Scalable RDF Stream Reasoning in the Cloud

Ren Xiangnan [a,b,*], Curé Olivier [b], Naacke Hubert [c] and Ke Li [a]
[a] *Innovation Lab Atos, Bezons France*
E-mails: xiang-nan.ren@atos.net, li_ke@yahoo.com
[b] *LIGM (UMR 8049) CNRS, ENPC, ESIEE, UPEM, Marne la Vallée, France*
E-mail: olivier.cure@u-pem.fr
[c] *LIP6 (UMR 7606) CNRS, Sorbonne Universités, UPMC, Paris, France*
E-mail: hubert.naacke@lip6.fr

Abstract. Reasoning over semantically annotated data is an emerging trend in stream processing aiming to produce sound and complete answers to a set of continuous queries. It usually comes at the cost of finding a trade-off between data throughput, latency and the cost of expressive inferences. Strider^R proposes such a trade-off and combines a scalable RDF stream processing engine with an efficient reasoning system. The main reasoning services are based on a query rewriting approach for SPARQL that benefits from an intelligent encoding of an extension of the RDFS (*i.e.*, RDFS with owl:sameAs) ontology elements. Strider^R runs in production at a major international water management company to detect anomalies from sensor streams. The system is evaluated along different dimensions and over multiple datasets to emphasize its performance.

Keywords: RDF stream processing, Reasoning, Scalability

1. Introduction

This paper deals with an emerging problem in the design of Big Data applications: reasoning over large volumes of semantically annotated data streams. The main goal amounts to producing sound and complete answers from a set of continuous queries. This problem is quite important for many Big data applications in domains such as science, finance, information technology, social networks and Internet of Things (IoT) in general. For instance, in the Waves[1] project, we are dealing with “real-time” anomaly detection in large water distribution networks. By working with domain experts, we found out that such detections can only be performed using reasoning services over data streams. Such inferences are performed over knowledge bases (KB) about the sensors used in water networks, *e.g.*, sensor characteristics together with their measure types, locations, geographical profiles, events occurring nearby, etc..

Tackling this issue implies to find a trade-off between high data throughput and low latency on the one hand and reasoning over semantically annotated data streams on the other hand. This is notoriously hard and even though it is currently getting some attention, it still remains an open problem. RDF Stream Processing (RSP) engines are the prominent systems tackling this problem where annotation are using the RDF (Resource Description Framework)[2] data model, queries are expressed in a continuous SPARQL[3] dialect and reasoning are supported by RDFS[4] (RDF Schema)/ OWL[5] (Web Ontology Language) ontologies. Existing RSP engines are either not scalable (*i.e.*, they do not distribute data and/or processing) or do not support expressive reasoning services.

Our system, Strider^R, combines our Strider RSP engine with a reasoning approach. Strider [27] is a stream processing engine for semantic data taking the form of RDF graphs. It is designed on top of state-of-the-art

[*] Corresponding author. E-mail: xiang-nan.ren@atos.net.
[1] http://www.waves-rsp.org/

[2] https://www.w3.org/RDF/
[3] https://www.w3.org/TR/sparql11-query/
[4] https://www.w3.org/TR/rdf-schema/
[5] https://www.w3.org/TR/owl2-overview/

Big Data components such as Apache Kafka[20] and Apache Spark[34]. It is thus the first RSP engine capable of handling at scale high throughput with relatively low latency. Strider is capable of processing and adaptively optimizing continuous SPARQL queries. Nevertheless, it was not originally designed to perform inferences. Hence, a main goal of this work is to integrate stream reasoning services that can support the main RDFS inferences together with the `owl:sameAs` property (henceforth denoted `sameAs`).

Intuitively, this property enables to define aliases between RDF resources. This is frequently used when a domain's (meta) data is described in a collaborative way, *i.e.*, a given object has been described with different identifiers (possibly by different persons) and are later reconciled by stating their equivalence. Reasoning with the `sameAs` property is motivated by the popularity of `sameAs` across many datasets, including several domains of the Linked Open Data (LOD). For instance, the `sameAs` constructor is frequently encountered to practically define or maintain ontologies. In [16], the authors measured the frequency of `sameAs` triples in an important repository of LOD. That property was involved in more than 58 million triples over 1,202 unique domain names with the most popular domains being biology, *e.g.*, Bio2rdf and Uniprot (respectively 26 and 6 million `sameAs` triples), and general domains *e.g.*, DBpedia (4.3 million `sameAs` triples). Moreover, the knowledge management of LOD, estimated to more than 100 billion triples, clearly amounts to big data issues. In our Waves running example, we also found out that, due to the cooperative ontology building, many `sameAs` triples were necessary to re-conciliate ontology designs. We discovered several of these situations in the context of the IoT Waves project. For instance, we found out that sensors or locations in water distribution networks could be given different identifiers.

These `sameAs` triples are generally persisted in RDF stores[11] but data streams are providing dynamic data streams about these resources. Such metadata are needed to perform valuable inferences. In the Waves project, they correspond to the topology of the network, characteristics of the network's sensors, etc. We consider that the presence of static metadata can be generalized to many domains, *e.g.*, life science, finance, social, cultural, and is hence important when designing a solution that reasons over their data streams. Strider^{R} thus needs to reason over both static KBs, *i.e.*, a set of facts together with some ontologies, and dynamic data streams, *i.e.*, a set of facts which once annotated with ontology concepts and properties can be considered as an ephemeral extension of the KB fact base. Apart from `sameAs` inferences, the most prevalent reasoning services in a streaming context are related to ontology concept and property hierarchies. We are addressing these inferences tasks via a trade-off between the query rewriting and materialization approaches.

The main contributions of this paper are (i) to combine a scalable, production-ready RSP engine that supports reasoning services over RDFS plus the `sameAs` property, (ii) to minimize the reasoning cost, and thus to guarantee high throughput and acceptable latency, and (iii) to propose a thorough evaluation of the system and thus to highlight its relevance.

The paper is organized as follows. In Section 2, we present some background knowledge in the fields of Semantic Web and stream processing. Section 3 provides an overview of the system's architecture. In Section 4, we detail a running example. Then Sections 5 and 6 provide reasoning approaches with respectively concept/property hierarchies and `sameAs` individuals. Section 7 evaluates Strider^{R} and demonstrates its relevancy. Some related works are proposed in Section 8. The paper concludes with Section 9.

2. Background knowledge

2.1. RDF and SPARQL

Resources present on the Web are generally represented using RDF, a schema-free data model. Assuming disjoint infinite sets I (Internationalized Resource Identifier (IRI) references), B (blank nodes) and L (literals), a triple (s,p,o) $\in$ (I $\cup$ B) x I x (I $\cup$ B $\cup$ L) is called an RDF triple with s, p and o respectively being the subject, predicate and object.

We now also assume that V is an infinite set of variables and that it is disjoint with I, B and L. SPARQL is the W3C query language recommendation for the RDF format. We can recursively define a SPARQL[1] triple pattern (tp) as follows: (i) a triple $tp \in$ (I $\cup$ V) x (I $\cup$ V) x (I $\cup$ V $\cup$ L) is a SPARQL triple pattern, (ii) if tp_1 and tp_2 are triple patterns, then $(tp_1.tp_2)$ represents a group of triple patterns that must all match, (tp_1 `OPTIONAL` tp_2) where tp_2 is a set of patterns that may extend the solution induced by tp_1, and (tp_1 `UNION` tp_2), denoting pattern alternatives, are triple patterns and (iii) if tp is a triple pattern and C is a built-in condition, then, (tp `FILTER` C) is a triple pattern enabling to restrict the solutions of a triple pattern match according to the expression C. A set of tps is denoted a Basic Graph Pattern (BGP). The SPARQL syntax follows the select-from-where approach of SQL queries.

2.2. Semantic Web KBs and reasoning

We consider that a KB consists of an ontology, aka terminological box (Tbox), and a fact base, aka assertional box (Abox). The least expressive ontology language of the Semantic Web is RDF Schema[2] (RDFS). It allows to describe groups of related resources (concepts) and their relationships (properties). RDFS entailment can be computed using 14 rules. But practical inferences can be computed with a subset of them. The one we are using is ρdf which has been defined and theoretically investigated in [23]. In a nutshell, ρdf considers inferences using `rdfs:subClassOf`, `rdfs:subPropertyOf` as well as `rdfs:range` and `rdfs:domain` properties.

An RDF property is defined as a relation between subject and object resources. RDFS allows to describe this relation in terms of the classes of resources to which they apply by specifying the class of the subject (*i.e.*, the domain) and the class of the object (*i.e.*, the range) of the corresponding predicate. The corresponding `rdfs:range` and `rdfs:domain` properties allow to state that respectively the subject and the object of a given `rdf:Property` should be an instance of a given `rdfs:Class`. The property `rdfs:subClassOf` is used to state that a given class (*i.e.*, `rdfs:Class`) is a subclass of another class. Similarly, using the `rdfs:subPropertyOf` property , one can state that any pair of resources (*i.e.*, subject and object) related by a given property is also related by another property.

Other ontology languages, OWL and its fragments, of the Semantic Web stack extend RDFS expressiveness, *e.g.*, by supporting properties such as `sameAs` or `owl:TransitiveProperty`. The deductive process then has a higher computational cost which can become incompatible with low latency constraints expected in a stream processing engine.

Two main approaches are generally used to support inferences in KBs. The first approach consists in materializing all derivable triples before evaluating any queries. It implies a possibly long loading time due to running reasoning services during a data preprocessing phase. This generally drastically increases the size of the buffered data and imposes specific dynamic inference strategies when data is updated. Besides, data materialization also potentially increases the complexity for query evaluation (*e.g.*, longer processing to scan the input data structure). These behaviors can seriously impact query performance. The second approach consists in reformulating each submitted query into an extended one including semantic relationships from the ontologies. Thus, query rewriting avoids costly data preprocessing, storage extension and complex update strategies but induces slow query response times since all the reasoning tasks are part of a complex query preprocessing step.

In a streaming context, due to the possibly long lifetime of continuous queries, the cost of query rewriting can be amortized. On the other hand, materialization tasks have to be performed on each incoming streams, possibly on rather similar sets of data, which implies a high processing cost, *i.e.*, lower throughput and higher latency.

2.3. RDF Stream Processing (RSP)

The nature of stream processing is to run a set of operations over unbounded data streams. Such processing are generally performed within a windowing operator that slices the incoming infinite data streams into finite chunks. These windows can be defined over some temporal constraints, *e.g.*, take the last 3 minutes of incoming data, or over non-temporal parameters, *e.g.*, counting or session based.

In the last decade, much effort has been devoted to improving RSP engines. In these systems, streams are represented are as RDF graphs and queried with a continuous version of the SPARQL query language. For instance, this amounts to introduce novel clauses that permit to define a streaming window of different types (*e.g.*, sliding, tumbling) with different parameters (*e.g.*, duration of a window, advancing step). Various systems have been defined (the main ones are presented in Section 8) and they can be differentiated by their capacity to support distribution of the data streams as well as workload and their capacity to reason over streams. These two dimensions have to be considered together due to the high temporal constraints of processing windows and the cost of reasoning (usually performed with query rewriting or materialization). To the best of our knowledge, Strider^R is the first system that addresses these two aspects with a being ready for production perspective (using state-of-the art engines such as Kafka and Spark) and expressive ontologies.

In summary, although RSP engines have substantially improved in recent years, none of them cover both scalability and expressive reasoning services.

2.4. Kafka and Spark Streaming

Apache Kafka[13] is a distributed message queue which aims to provide a unified, high-throughput, low-latency real-time data management. Intuitively, producers emit messages which are categorized into adequate *topics*. The messages are partitioned among a cluster to support parallelism of upstream/downstream operations. Kafka uses *offsets* to uniquely identify the location of each message within the partition.

Apache Spark is a MapReduce-like cluster-computing framework that proposes a parallelized fault tolerant collection of elements called Resilient Distributed Dataset (RDD) [31]. An RDD is divided into multiple partitions across different cluster nodes such that operations can be performed in parallel. Spark enables parallel computations on unreliable machines and automatically handles locality-aware scheduling, fault-tolerant and load balancing tasks.

Spark Streaming extends RDD to Discretized Stream (DStream) [32] and thus enables to support near real-time data processing by creating *micro-batches* of duration T. DStream represents a sequence of RDDs where each RDD is assigned a timestamp. Similar to Spark, Spark Streaming describes the computing logics as a template of RDD Directed Acyclic Graph (DAG). Each batch generates an instance according to this template for later job execution. The micro-batch execution model provides Spark Streaming second/sub-second latency and high throughput.

3. Strider^R overview

In this section, we first give a high-level overview of the Strider^R system. Its architecture has been designed to support the distribution the processing of RDF data streams and to provide guarantees on fault tolerance, high throughput and low latency. Moreover, Strider^R aims to integrate efficient reasoning services into an optimized continuous query processing solution.

Figure 1 shows 3 vertical "columns" or groups of functions: (a), (b), and (c). On the middle and the right, (a) and (b) are off-line pre-processing functions. On the left, (c) is the on-line stream processing pipeline. We detail the three groups below, in the order they participate to the whole workflow:

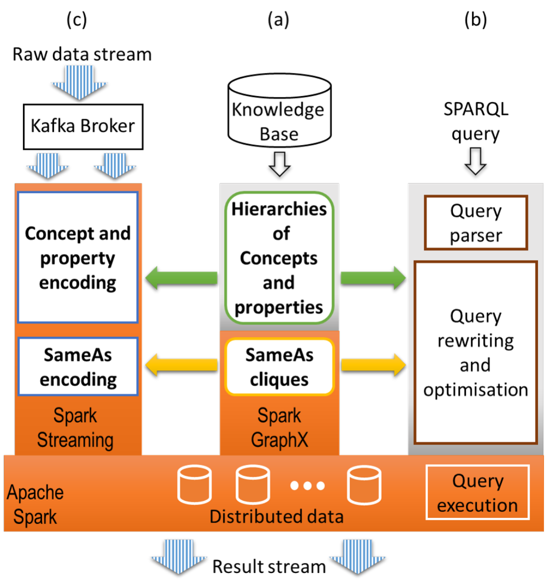


Fig. 1. Strider^R Functional Architecture

(a) Off-line KB encoding. It consists in reading the static knowledge base to get the classification of concepts and properties, both organized into a hierarchy. The knowledge base also contains `sameAs` predicates from which `sameAs` cliques are detected. This step generates the identifiers for each concept, property and individual clique that is later used in steps (b) and (c). Note that we use the GraphX library of Apache Spark to efficiently process clique detection in parallel.

(b) Off-line query preparation. Once a SPARQL query is registered into the system, it is parsed then rewritten into a plan composed of basic RDF processing operations. The plan is extended with dedicated operations to support the reasoning over properties and concepts, using the semantic identifiers generated at step (a). The plan also relies on the `sameAs` cliques information to support the `sameAs` reasoning for various use cases.

(c) On-line stream semantic encoding. The data stream is encoded based on the hierarchical codes generated from the static KB at step (a). Each concept and property is replaced by an identifier that allows for fast reasoning over concept and property hierarchies. The stream is also completed with `sameAs` clique membership information. For the purpose of ensuring high throughput and fault-tolerance, we use Apache Kafka to manage the data flow. The incoming raw data are assigned to so-called Kafka *topics*. The Kafka broker distributes the topics and the corresponding data over a

cluster of machines to enable parallelism of upstream/-downstream operations. Then the distributed streams seamlessly enter the Spark Streaming layer which encodes them in parallel.

Continuous query processing. The logical plan obtained at step (b) is pushed into the query execution layer (*i.e.*, the base layer of Figure 1 on which the three "groups" (*i.e.*, a,b and c) of previously defined functions rely). To achieve continuous SPARQL query processing on Spark Streaming, we bind the SPARQL operators to the corresponding Spark SQL relational operators that access a distributed compressed in-memory representation of the data stream (through the DataFrame and the RDD APIs provided by the Spark platform). Note that, Strider^{R} is capable of adjusting the query execution plan at-runtime via its adaptive optimization component.

Figure 1 also serves as a map to better outline our main contributions:

– The green arrows highlight the contributions about **reasoning over concepts and properties** presented in Section 5: generating hierarchies of concepts and properties (Section 5.2.1), concept and property encoding (Section 5.2.2), and the corresponding query rewriting method (Section 5.2.3).
– The yellow arrows highlight the contributions about **reasoning over `sameAs` facts** presented in Section 6: `sameAs` clique detection (Section 6.1) and two alternative methods (Sections 6.2 and 6.3) for `sameAs` encoding and query rewriting.

4. Running example with continuous queries

In this section, we present a running example that will be used all along the remaining of the paper. A first issue concerns the selection of a supporting benchmark. Two characteristics prevent us from using well-established RSP benchmarks [3, 25, 35]: their lack of support for the considered reasoning tasks and their inability to cope with massive RDF streams. We thus selected a benchmark with which the Semantic Web community is confident with, namely the Lehigh University Benchmark (henceforth LUBM)[15], and extended it in two directions. First, we created a stream generator based on the triples contained in the LUBM Abox. Second, we extended the LUBM generator with the ability to create individuals related by the `sameAs` property. Intuitively, novel individuals are generated and stated as being equivalent to some other LUBM individuals. This generator is configurable and one can decide how many sameAs cliques and how many individuals per clique are created.

The LUBM ontology has not been extended and in Figure 2 we provide an extract of it. It contains a subset of the property hierarchy (*i.e.*, memberOf, worksOf and headOf) as well as a subset of the concept hierarchy. We will emphasize in Section 5 on the encoding of this extract of the Tbox. This figure also presents elements of the Abox,*i.e.*, RDF triples concerning individuals. That extract highlights the creation of individuals related by `sameAs` property, thus creating individual cliques. We have three cliques in this figure: (pDoc1, pDoc2, pDoc3), (pDoc4,pDoc5,pDoc6) and (pDoc7, pDoc8 and pDoc9). This example will be used in Section 6 when detailing inferences concerned with the `sameAs` property.

In order to evaluate our Strider^{R} engine, we also created a set of continues SPARQL queries (eight in total, two of them are presented here and the rest are in Appendix A). The SPARQL query Q4 (Listing 1) requires reasoning over both the concept and property hierarchies. Intuitively, the query retrieves professor names and the organization they are a member of. In LUBM, no individuals are directly typed as a Professor but many individuals are stated as a sub-concept of the Professor concept (*e.g.*, Associate, assistant, full professors). Moreover, the `memberOf` property has one direct (namely `worksFor`) and one indirect (namely `headOf`) sub-properties. Hence, it is necessary to perform some inferences to obtain the complete answer to this query.

As shown in Q4, we have extended the standard SPARQL query language with some clauses for a continuous query processing (more details in Appendix B).

```
STREAMING { WINDOW [10 Seconds]
            SLIDE [10 Seconds]
            BATCH [5 Seconds] }
REGISTER { QUERYID [Q1]
           REASONING [U,SM]
SPARQL [ PREFIX rdf: <http://...ns#>
         PREFIX lubm: <http://..owl#>
         SELECT ?o ?n WHERE {
             ?x rdf:type lubm:Professor;
                lubm:memberOf ?o;
                lubm:name ?n.} }] }
```

Listing 1: Query Q4 involving concept hierarchy inference

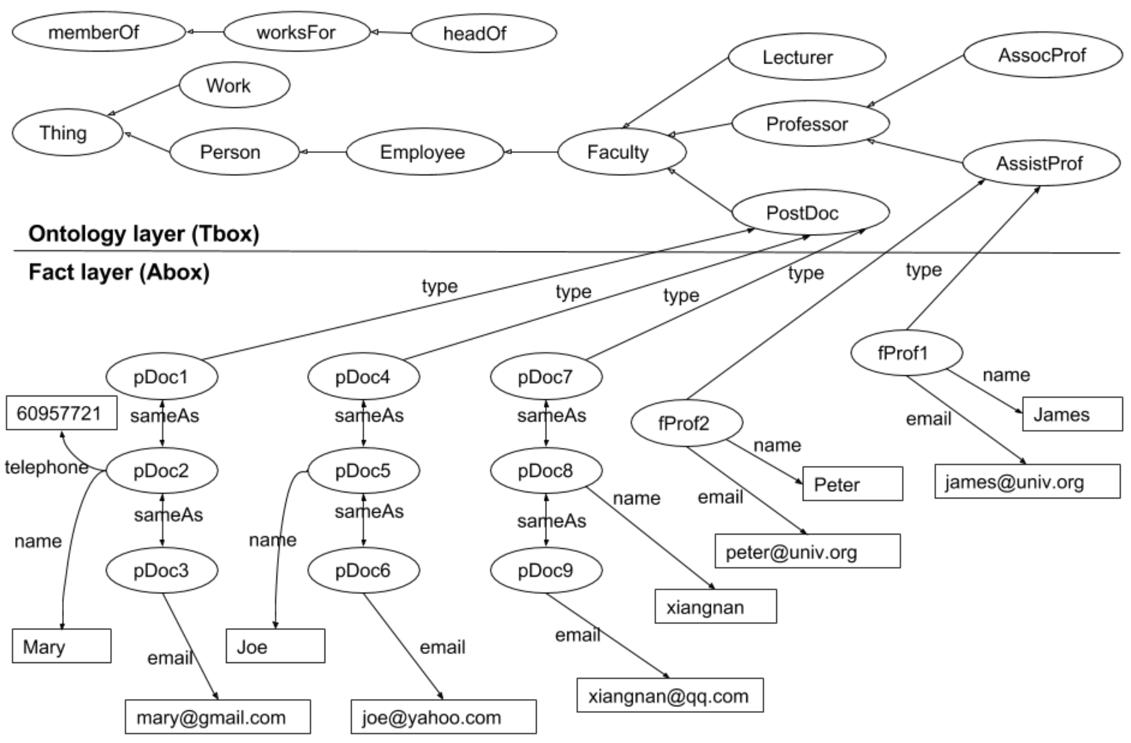


Fig. 2. LUBM's Tbox and Abox running example

In Listing 2, we present the SPARQL part of query Q6 (*i.e.*, the streaming and register clauses are not presented since they do not provide any new information). This query retrieves names and email addresses of resources typed as PostDoc. It requires `sameAs` inferences since several individuals stated as PostDoc belong to `sameAs` cliques (namely pDoc1 to pDoc9).

```
PREFIX rdf: <http://...ns#>
PREFIX lubm: <http://..owl#>
SELECT ?n ?e
WHERE { ?x rdf:type lubm:PostDoc;
           lubm:name ?n;
           lubm:emailAddress ?e.}
```

Listing 2: Query Q6 involving sameAs inference

5. Reasoning over concept and property hierarchies

In the following, we consider approaches for reasoning over concept and property hierarchies. We first present the classical approach consisting in the standard query rewriting. Then, we present an extension of our LiteMat reasoner [10] which, compared to the standard approach, provide better performances in most queries.

In both approaches, encoding the elements of the Tbox, *i.e.*, concept and property hierarchies, is needed upfront to any data stream processing. The KB Encoding component encodes concepts, properties and instances of registered static KBs. This aims to provide a more compact (*i.e.*, replacing string-based IRIs or literals with integer identifiers) representation of the Tbox and Abox as well as supporting more efficient comparison operations. In the general case, each concept and property is mapped to an arbitrary unique integer identifier. We will emphasize that our LiteMat approach produces a semantic encoding scheme that supports important reasoning services. In the following, we consider inferences pertaining to the ρdf subset of RDFS (the `sameAs` property is considered in Section 6) and the input ontology is considered to be the union of (supposedly aligned) ontologies necessary to operate over one's application domain.

5.1. Standard rewriting: add UNION Clauses

The standard rewriting approach to perform inferences over concept and property hierarchies on SPARQL queries consists in a reformulation according to an analysis of the Tbox. Intuitively, for a query Q with BGP B. For all triples $t \in B$, the system searches in the Tbox if the property (resp. concept) in t has sub-properties (resp. sub-concepts). In the affirmative, a set of UNION clauses is appended to Q, thus producing a new query Q'. A new UNION clause contains a rewriting of B where the property (resp. concept) is replaced with a sub-property (resp. sub-concept). A UNION clause will be added for each direct and indirect sub-properties (resp. sub-concepts) and their combinations if the BGP contains several of them.

In Listing 3, we provide an example of this rewriting for Q4 and the extract of the LUBM ontology (see Figure 2). We display only six (out of the twelve, three properties times four concepts) UNION clauses present in the rewriting. Note that for each UNION clause, two joins are required.

```
SELECT ?o ?n
WHERE {{ ?x rdf:type lubm:Professor;
            memberOf ?o;
            lubm:name ?n. }
 UNION { ?x rdf:type lubm:Professor;
            worksFor ?o;
            lubm:name ?n. }
 UNION { ?x rdf:type lubm:Professor;
            headOf ?o;
            lubm:name ?n. }
 UNION { ?x rdf:type lubm:AssistantProfessor;
            memberOf ?o;
            lubm:name ?n. }
 UNION { ?x rdf:type lubm:AssistantProfessor;
            worksFor ?o;
            lubm:name ?n. }
  UNION { ?x rdf:type lubm:AssistantProfessor;
            headOf ?o;
            lubm:name ?n. }  ...  }
```

Listing 3: Strider's query example (Q_4)

This approach guarantees the completeness of the query result set but comes at the high cost of executing a potentially very large queries (due to an exponential increase of original query). Those constraints are not compatible with executions in a streaming environment. In the next section, we present a much more efficient approach.

5.2. LiteMat adapted to stream reasoning

In the following, we dedicate two subsections to our encoding scheme: one for the static KB and one for the streaming (dynamic) data. Then a rewriting dedicated to this encoding scheme is detailed.

5.2.1. Static encoding

In LiteMat, inferences drawn from properties such as `rdfs:subClassOf` and `rdfs:subPropertyOf` are addressed by attributing numerical identifiers to ontology terms, *i.e.*, concepts and properties. The compression principle of this term encoding lies in the fact that subsumption relationships are represented within the encoding of each term. This is performed by prefixing the encoding of a term with the encoding of its direct parent (a workaround using an additional data structure is proposed to support multiple inheritance). The generation of the identifiers is performed at the bit level.

More precisely, the concept (resp. property) encoding are performed in a top-down manner, *i.e.*, starting from the top concept of the hierarchy (the classification is performed by a state-of-the-art reasoner, *e.g.*, HermiT[12], and hence supports all OWL2 logical concept subsumptions), such that the prefix of any given sub-concept (resp. sub-property) corresponds to its super-concept (resp. super-property). Intuitively, for the entity hierarchy (*i.e.*, concept or property), we start from a top entity and assign it the value 1 (see the raw id of `owl:Thing` in Table 1) and process its N direct sub-entities. These sub-entities will be encoded over $\lceil log_2(N+1) \rceil$ bits and their identifiers will be incremented by 1. This approach is performed recursively until all entities in the TBox are assigned an identifier. It is guaranteed at the end of this first phase that, for 2 entities A and B with $B \sqsubseteq A$, the prefix of *idB* matches with the encoding *idA*. Note that for the property hierarchy, a reasoner is not needed to access the direct sub-properties. Moreover, we distinguish between object and datatype properties by assigning different starting identifiers, respectively '01' and '10'. Finally, some RDF and OWL properties, *e.g.*, `rdf:type` are assigned identifiers in the '00' range.

In a second step, to guarantee a total order among the identifiers of a given concept or property hierarchy, the lengths of these identifiers have to be normalized. This is performed by computing the size of the longest branch in each hierarchy and by encoding each identifier on this length of bits (*i.e.*, filling '0' on the right most bit positions).

These normalized identifiers are stored as integer values in our dictionary. The characteristics of this encoding scheme ensures that from any concept (resp. property) element, all its direct and indirect sub-elements can be computed with only two bit shift operations and are comprised into a discrete interval of integer values, namely its lower and upper bound (resp. LB,UB). Table 1 presents the identifiers of Figure 2's LUBM concept hierarchy extract. The first step of the encoding generates raw ids (column 1). We can observe that the `Faculty`'s prefix 110101 corresponds to the `Employee`'s identifier, and his hence one of its direct sub-concept. Moreover, `Employee` is a direct sub-concept of `Person` and indirect sub-concept of `owl:Thing`. These raw ids are normalized to produce column 2 of Table 1. Finally, integer values contained in the id column are stored in the dictionary.

Raw ids	Normalized ids	id	Term
1	1000000000000	4096	`owl:Thing`
1001	1001000000000	4608	Schedule
1010	1010000000000	5120	Organization
1011	1011000000000	5632	Publication
1100	1100000000000	6144	Work
1101	1101000000000	6656	Person
110101	1101010000000	6784	Employee
110101001	1101010010000	6800	Faculty
11010100101	1101010010100	6804	Lecturer
11010100110	1101010011000	6808	PostDoc
11010100111	1101010011100	6812	Professor
1101010011101	1101010011101	6813	AssistantProf.
1101010011110	1101010011110	6814	AssociateProf.

Table 1

Encoding for an extract of the concept hierarchy of the LUBM ontology

The encoding scheme of individuals of static KBs can take two forms. It depends on whether an individual is involved in a triple with a `sameAs` property or not. The encoding scheme for `sameAs` cliques is detailed in Section 6.1. For non-sameAs individuals, we apply a simple method which attributes a unique integer identifier (starting from 1) to each individual. In [10], we provided an efficient distributed method to perform this encoding.

5.2.2. *Dynamic partial encoding*

In the previous section, we detailed the generation of dictionaries for the elements of static KBs, *i.e.*, their concepts, properties and individuals. These maps are being used to encode, on the fly, incoming data streams. That is concept and property IRIs of a data stream are being replaced with their respective integer identifier. The same transformation is processed for individual IRIs or literals. This approach permits to drastically compress the streams without incurring high compression costs.

In practical use cases, some entries of data streams may not correspond to an entry in one of the dictionaries. For instance, due to their infinite nature, numerical values, *e.g.*, sensor measures in the IoT, can not possibly all be stored in the individual dictionary. Other cases are possible where a stream emits a message where concepts and/or properties are not present in our dictionaries. Note that such situations prevent the system from performing any reasoning tasks upon the missing ontology elements.

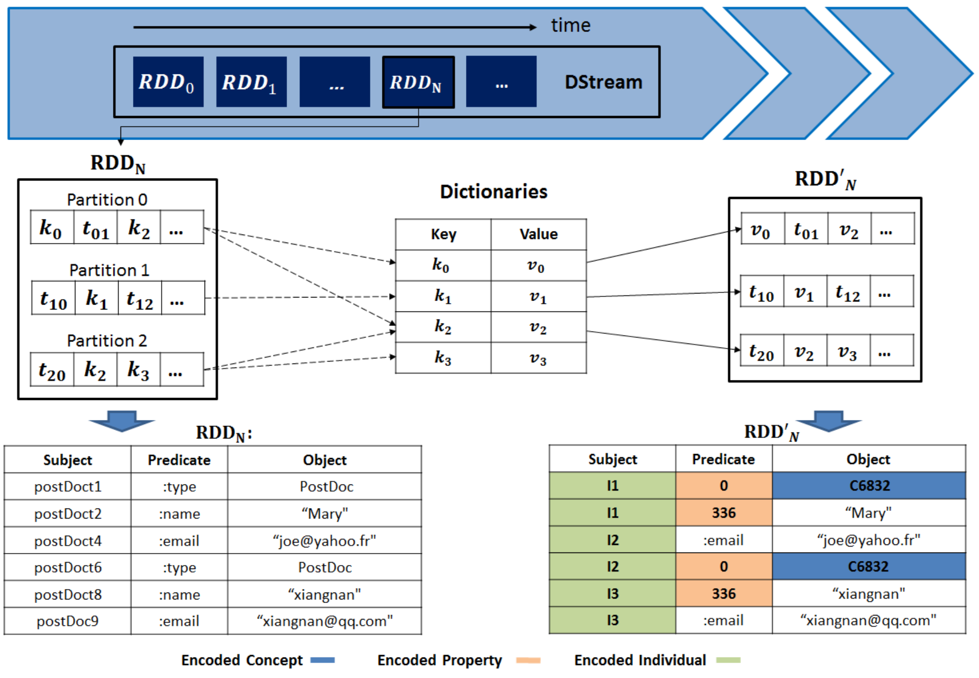
When facing the absence of a dictionary entry, we are opting for a partial stream encoding. Intuitively, this means that we are not trying to create a new identifier on the fly but rather decide to leave the original data as is, *i.e.*, as an IRI, literal or blank node. After some experimentations, we found out that this is good trade-off between maintaining ever growing, distributed dictionaries and favoring an increase of incoming data streams rate.

Figure3 provides some details on how this partial encoding is implemented into Strider^{R}. Intuitively, it uses the Discretized stream (Dstreams) [32] abstraction of Apache Spark Streaming where each RDD is composed of a set of RDF triples. For each RDD, a transformation is performed which takes a IRI/literal based representation to a partially encoded form. This transformation lookups into the TBox and Abox dictionaries precomputed from static KBs. The bottom right table of the figure emphasizes that some triple elements are encoded while some other are not. The dictionaries are broad-casted to all the machines in the cluster. The encoding for each partition of data is thus performed locally.

We briefly summarize important advantages of the partial encoding of RDF streams: (i) an efficient parallel encoding to meet real-time request; (ii) no extra overhead for dictionary generation.

5.2.3. *Query rewriting: FILTER clauses and UDF*

In Section 5.2.1, we highlighted that to each concept and property corresponds a unique integer identifier. Moreover, one characteristic of our encoding method guarantees that all sub-concept (resp. sub-property) identifiers of a given concept (resp. property) are included into an interval of integer values, denoted lower



Key	Value
k_0	v_0
k_1	v_1
k_2	v_2
k_3	v_3

RDD_N:

Subject	Predicate	Object
postDoct1	:type	PostDoc
postDoct2	:name	“Mary"
postDoct4	:email	“joe@yahoo.fr"
postDoct6	:type	PostDoc
postDoct8	:name	“xiangnan"
postDoc9	:email	“xiangnan@qq.com"

RDD'_N

Subject	Predicate	Object
I1	0	C6832
I1	336	“Mary"
I2	:email	“joe@yahoo.fr"
I2	0	C6832
I3	336	“xiangnan"
I3	:email	“xiangnan@qq.com"

Fig. 3. Parallel partial encoding over DStream

bound (LB) and upper bound (UB) of that ontology element.

We now concentrate on the query rewriting for concepts. In order to speed up the rewriting, we take advantage of the following context: since we are only considering that data streams are representing elements of the Abox, concepts are necessarily at the object position of a triple pattern and the property must be `rdf:type`. Intuitively, if a concept has at least one sub-concept then it is replaced in the triple pattern by a novel variable and a SPARQL `FILTER` clause is added to the query's BGP. That filter imposes that the new variable is included between the LB and UB values (which have been previously computed at encoding-time and stored in the dictionary) of that concept.

The overall approach is quite similar for the rewriting concerning the property hierarchy but no specific context applies, *i.e.*, all triple patterns have to be considered. For each triple pattern, we check whether the property has some sub-properties. If it is the case then the property is replaced by a new variable in the triple pattern and a SPARQL `FILTER` clause is introduced in the BGP. That filter clause restricts the new variable to be included in the LB and UB of that property.

As a concrete example of this rewriting, we are using query Q4 of our benchmark since it requires inferences over both the concept and property hierarchies.

The rewriting Q4' of Q4 contains two `FILTER` clauses, one for the `Professor` concept and one for the `memberOf` property (LB() and UB() functions respectively return the LB and UB of their parameter). Given p, the parameter submitted to LB and UB, these functions respectively return the identifier of p and an identifier computed using two bit shift operations on p. So, the computation of the UB function is quite fast. Note the introduction of the $?p$ and $?m$ variables, respectively replacing the `Professor` concept and `memberOf` property. The lower and upper bounds for $?p$ correspond to the identifier of the Professor and AssociateProfessor, respectively (equal to 6812 and 6814 in Table 1).

```
SELECT ?o ?n
WHERE { ?x rdf:type ?p; ?m ?o;
           lubm:name ?n.
         FILTER (?p>=LB(Professor)
              && ?p<UB(Professor)).
         FILTER (?m>=LB(memberOf)
              && ?m<UB(memberOf)).}
```

Listing 4: LiteMat query rewriting for query Q4

Finally, this rewriting is much more compact and efficient than the classical reformulation which would require twelve `UNION` clauses and twenty four joins[6].

6. Reasoning with the `sameAs` property

This section concerns inferences performed in the presence of triples containing the `sameAs` property. In Section 6.1 a distributed, parallelized approach to encode `sameAs` cliques and a naive approach to materialize inferred triples are proposed. We emphasize that this approach is not adapted to a streaming context. Therefore, the challenge is to support `sameAs` reasoning efficiently while answering queries over streaming data. In Strider^R, we address this challenge and propose two solutions. The first one (Section 6.2) aims for efficiency, the second one (Section 6.3) aims to handle reasoning applied to a "provenance awareness" scenario which is not supported by the first solution.

6.1. *SameAs clique encoding*

Consider, in a KB, a set of `sameAs` triples that represents a graph denoted G_{sa}. The nodes V of G_{sa} are set of individuals (either at the subject or object position of a triple) of the `sameAs` triples. Let $x \in V$ denote such a node. For convenience, every node $x \in V$ is uniquely identified by an integer value $Id(x) \in [1, |V|]$.

Let C be the set of connected components that partition G_{sa}. Although a component may not be fully connected, it represents a `sameAs` clique because of the semantic equivalence (*i.e.*, transitivity and symmetry) of the `sameAs` property. Let C_i denote the connected component containing the node identified by i.

In Strider^R, we assume that the `sameAs` triples are in the static KB. We detect the C_i using a parallel algorithm to compute connected components [14].

The principle of that algorithm is to propagate through the graph a numeric value representing a component id, such that every connected component will end up with a component id assigned to its members.

Initially, each node x is assigned with $Id(x)$. Then for each node x, the group that comprises x and its neighbors is considered and the minimum number among the group members is assigned to all the group members. The algorithm ends when no more update occurs at any group, *i.e.*, for any group (or connected component) all the members share the same component id. About the computational cost of clique detection, note that it is computed in a distributed and parallel manner (using the GraphX library of the Apache Spark engine). Hence it is able to scale to very large static KBs.

Once the connected components are detected, we define the $Cl(x)$ mapping that associates the IRI of x with its clique id. Table 2 summarizes the notations used in this section.

Notation	Description
G_{sa}	The `sameAs` graph of individuals
$Id(x)$	The integer ID of individual x
C_i	The clique st. i is the minimal ID among the members
$Cl(x)$	The clique ID of individual x.
$IRI(i)$	The IRI of ID i (or set of IRIs if i is in a clique)
S	The average size of a clique

Table 2

Notations used for `sameAs` reasoning

In order to reason over `sameAs`, an obvious solution is to materialize all inferences. However that is not tractable because the number of inferred triples is far too high in general. Consider a triple $t = (x, p, y)$ where x (resp. y) belongs to the `sameAs` clique C_x (resp. C_y). Let S be the average size of a clique. The number of triples inferred from t is $2 \times S^2$. Therefore, the number of triples inferred from the entire dataset D (of size $|D|$) can grow up to $2 \times |D| \times S^2$ in the worst case. For instance, from Figure 2 we obtain Figure 4(a) where all dashed edges correspond to a materialization of all inferences induced by `sameAs` explicit triples. We can easily witness the increase of represented triples. This naïve approach is generally not adopted in RDF database systems storing relatively static datasets due to its ineffectiveness. This is even more relevant in a dynamic, streaming context. First, it may not be feasible to generate all materialization within the time constraint of a window execution. Second, the execution of a continuous query over such stream sets would be inefficient.

6.2. *Representative-based (RB) reasoning*

Based on the detected cliques, the principle of the representative-based (RB) reasoning is to fuse the dataset such that all the individuals that belong to the same clique appear as a single (representative) individual. As a consequence, the fused graph implicitly carries the `sameAs` semantics. Then, a regular evaluation of any query on the fused graph is guaranteed to comply with the `sameAs` semantics.

[6]Rewriting available on our github page

6.2.1. Stream encoding

Stream encoding according to `sameAs` individuals consists of two steps:

1. First, select a single individual per clique C_i, which acts as the clique *representative*. The representative can be any member, as long as there is only one representative per clique. Without loss of generality, we assume that the representative for C_i is the member whose node number equals to i. Therefore, given the IRI x of any individual, its representative is numbered $Cl(x)$. In Figure 4(a) shows three cliques in dotted boxes, and $pDoc1$ individual can serve as the representative of the clique $(pDoc1, pDoc2, pDoc3)$.
2. Second, encode the input stream: replace every (x, p, y) triple by its corresponding representative-based triple: $(Cl(x), p, Cl(y))$. Fig. 4(b) shows the result of the encoding where individuals $pDoc1$, $pDoc5$ and $pDoc9$ are the so-called representatives of the cliques.

This approach has many advantages, especially in a data streaming context:

(i) The inferred graph is more compact without loss of information from the original graph,

(ii) The dictionary data structure that implements $Cl(x)$ is light. The dictionary size equals to the size of G_{sa} which is in practice very small compared to the number of triples to process in a streaming window. The computing overhead of encoding the input stream is negligible,

(iii) Clique updates (*e.g.*, removing or adding an individual from a clique) does not imply to update the input stream since the data streams are ephemeral. This assumes that an update of the static KB is taken into account starting from the next window that only contains data produced after the update. For instance, consider the clique named C_1 with 3 members $(pDoc1, pDoc2, pDoc3)$. At time t, the KB is updated: $pDoc4$ is declared to be `sameAs` $pDoc2$ thus $pDoc4$ joins C_1. The data already streamed before t are not updated, *i.e.*, the triples mentioning $pDoc4$ are not updated. Whereas, in the window following t, $pDoc4$ will be translated to $Cl(pDoc4)$.

6.2.2. Query processing

Based on the above encoding, a standard query processing is performed where variable bindings concern both standard individuals and `sameAs` representative.

Note that because the `sameAs` reasoning is fully supported by the representative-based encoding, we can simplify the query by removing the `sameAs` triple patterns that it may contain.

To evaluate a filter clause that refers to an IRI value, *e.g.*, `FILTER {?x like '*w3c.org*'}`, we rewrite it into an expression that refers back to the IRI value(s) instead of the encoded identifier. Let define $IRI(x)$ as the IRI (or the set of IRIs in case of a clique) associated with encoded ID x. Let $f(x)$ be a `FILTER` condition on variable $?x$, $f(x)$ is then rewritten into $\{\exists e \in IRI(x) | f(e)\}$.

A final step decodes the bindings: each encoded value is translated to its respective IRI or literal value. If the encoded value is a clique number, then it translates to the IRI of the clique representative.

6.3. SAM reasoning

SAM stands for SAM for `sameAs` Materialization and aims to handle reasoning in the case of origin-preserving (or provenance-awareness) scenario which are not supported by the RB solution introduced above (Section 6.2).

SAM reasoning targets the use cases that require to make the distinction between the *original* dataset triples and the *inferred* triples. That distinction is necessary for a user investigating which part (or domain) of the dataset contributes to the query, *i.e.*, brings some piece of knowledge, when the IRIs within a clique have different domains. In this section, we begin by motivating SAM using a concrete example then we detail a method to evaluate queries in this setting.

6.3.1. Motivation

We briefly sketch an example showing the limitations of RB approach and the need for the proposed SAM approach. For instance, consider the dataset of Figure 2. Suppose all the triples about email addresses come from domain1 (*e.g.*, mail.univ.edu), then the IRIs $pDoc3$, $pDoc6$, and $pDoc9$ are in that domain. Similarly, suppose all telephone numbers come from domain2 (*e.g.*, phone.com), then $pDoc2$ is in domain2. Let consider a query searching the IRI and the email of a person named Mary. We could write that query Q as follows:

```
Q: select ?x, ?y where {
   ?x name "Mary" . ?x email ?y}
```

The result is $?x = pDoc2$ and $?y = mary@gmail.com$. Based on that result and on the clique membership information, we know that the possible bindings for $?x$ are also $pDoc1$ or $pDoc3$. However the result does not

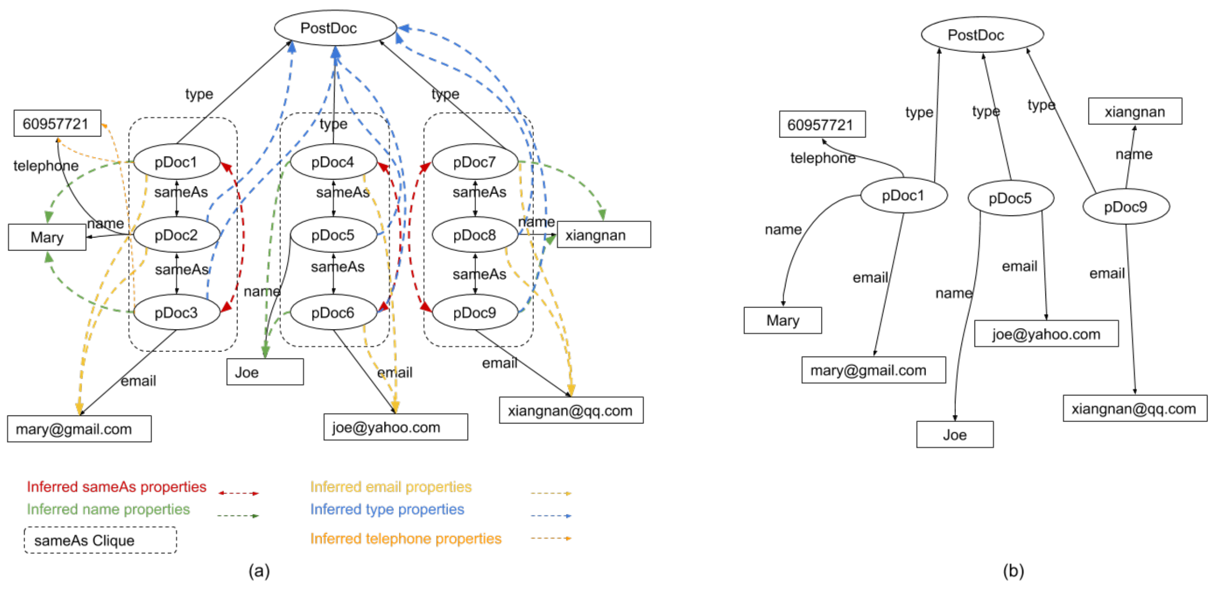


Fig. 4. `sameAs` representation solutions

inform us that *pDoc*3 as well as its *domain*1 were originally concerning the `email` triple. To get this provenance information, we could write the query as Q':

```
Q': select ?x1, ?y where {
    ?x name "Mary" .
    ?x sameAs ?x1 . ?x1 email ?y}
```

However, through the RB approach, the result is still $?x1 = pDoc2$ and $?y = mary@gmail.com$ because *pDoc*2 is the representative of *pDoc*3. The goal of the SAM approach is to make Q' return the binding $?x1 = pDoc3$ instead of $?x1 = pDoc2$. Doing this way, we will get that the IRI *pDoc*3 and *domain*1 directly relate to the email information within the dataset. To sum up, the RB approach (*c.f.* §6.2) does not support the origin-preserving use case, because a query result only binds to individuals that are clique representatives or not member of a clique at all. The result lacks information about which IRI originally exists in the triples that match the query.

To overcome this drawback, we propose the SAM approach that keeps track of the individuals that match the query even if they are part of a `sameAs` clique. The principle of the SAM approach is to **explicitly handle the `sameAs` equivalence** such that the equivalent individuals that match a query are preserved in the query result. From a logical point of view, this means to manage explicit `sameAs` information both in the dataset and in the query.

- `sameAs` in the dataset: complete the input stream with explicit information representing the `sameAs` equivalences between IRIs.
- `sameAs` in the query: complete the query with triple patterns explicitly expressing the `sameAs` matching.

6.3.2. *Materialize `sameAs` data streams*

A general method consists in completing the input stream with `sameAs` information. We devise an efficient solution that guarantees to materialize only the necessary triples and prevents from exploding the size of the data stream. Moreover, the IRIs are encoded to get a more concise representation to save on query execution time.

We now detail the steps of our solution. Let W be the current streaming window. The idea is to express each clique that has at least one member in W by a minimal set of triples. Let $\mathcal{C}$ be the set of cliques used in W and a clique $C_i \in \mathcal{C}$. For each member x of C_i such that $Id(x) \neq i$ (the identifier $Id(x)$ is defined in Table 2), add the triple $<$ i `sameAs` $Id(x)$ $>$ into W. For instance, consider the three `sameAs` cliques of Figure 4(a). Let denote C_1 the clique containing $(pDoc1, pDoc2, pDoc3)$, the minimal Id in C_1 is $Id(pDoc1) = 1$. Suppose the input stream window contains:

```
pDoc1 type PostDoc
pDoc2 name "Mary"
pDoc3 emailAddress "mary@gmail.com"
```

While applying the SAM approach, the window is completed with only two triples:

```
1 sameAs Id(pDoc2)
1 sameAs Id(pDoc3)
```

Let S be the average clique size. Notice that only $S-1$ edges of C_i out of S^2 (those with subject i) are added into the stream. The added `sameAs` edges represent a directed star centered at i the member of C_i with minimal Id. As explained below in Section 6.3.3, that light materialization is sufficient to fully enable the `sameAs` reasoning during query processing.

Cost analysis. We analyze the materialization cost in terms of data size. The total amount of materialized triples in W is $|\mathcal{C}| \times (S-1)$. The space overhead of SAM is indeed far smaller than a full materialization of every triple inferred from the `sameAs` reasoning which would add $2 \times |W| \times S^2$ triples (*c.f.* Table 2). Moreover, our solution materializes S times less triples compared to materializing all the clique edges. This low memory footprint makes our solution more scalable.

6.3.3. *Query rewriting: add `sameAs` patterns*

Consider a BGP query represented by a graph of triple patterns where nodes are variables, IRIs or literals. In a query, a join node is a variable that connects at least two triple patterns. The query rewriting method consists in extending a BGP query with `sameAs` patterns that could match the materialized stream. The principle is to "inject" the `sameAs` semantics into each join appearing in the query, *i.e.*, to decompose a **direct** join on one variable into an **indirect** join through a path of two `sameAs` triple patterns. Consequently, each join is decomposed into three join operations. Intuitively, the join nodes of the BGP are split to be replaced by a star of `sameAs` triple patterns such that the join "traverses" the star center.

The shape of the added `sameAs` patterns is a star because it has to match the stars `sameAs` triples that have been materialized into the stream. We consistently adopt a star-shaped representation of the `sameAs` information both in the materialized stream and in the query pattern. This guarantees any `sameAs` relation within a clique to be expressed by a path of length two (the diameter of a star). Besides, a star triple pattern is guaranteed to match any path of length two. Therefore, our proposed rewriting is guaranteed to match any `sameAs` path within the stream, *i.e.*, the rewritten query is semantically equivalent to the initial one.

We next detail the query rewriting algorithm. Let V be the set of join variables of a query. For each $v \in V$, (i) *Split the join variable*: replace each occurrence of v in the query by a distinct variable. Let $v_1, \cdots, v_n$ denote the variables replacing the n occurrences of v. (ii) *Express the indirect join*: For each v_i add the $(?v$ `sameAs` $?v_i)$ triple pattern.

For example, consider the following query Q6 and its graphical representation shown in Figure 5:

```
select ?x where {
?x type PostDoc.
?x name ?n.
?x emailAddress ?y.}
```

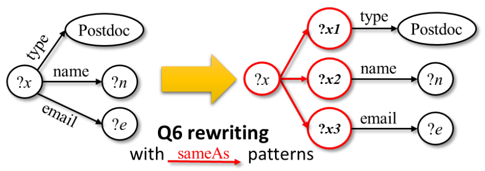


Fig. 5. SAM rewriting for the Q_6 query

The $?x$ join variable is split into $?x_1, ?x_2, ?x_3$ and these new variables are connected through `sameAs` patterns. The rewritten equivalent query is:

```
select ?x, ?x1, ?x2, ?x3
where {
?x1 type PostDoc.      ?x sameAs ?x1.
?x2 name ?n.           ?x sameAs ?x2.
?x3 emailAddress ?y.  ?x sameAs ?x3. }
```

Another example, on Figure 6, shows the rewriting case of a join variable that appears both as an object and as a subject position.

6.3.4. *Query evaluation: join with `sameAs` patterns*

Evaluating a rewritten `sameAs` query requires special attention in order to ensure that the result is complete. Our SAM approach minimizes the amount of materialized `sameAs` triples for better efficiency. Thus, the dataset does not contain any `sameAs` reflexive triple x `sameAs` x. Remind that our solution aims to bring `sameAs` reasoning capability to query engines that do not support `sameAs` reasoning natively. Such query engine do not infer x `sameAs` x for any individual. Therefore, a regular evaluation of a `sameAs` query may lead to incomplete result. For instance, let us remind the example dataset of § 6.3.2 including the materialized `sameAs` triples:

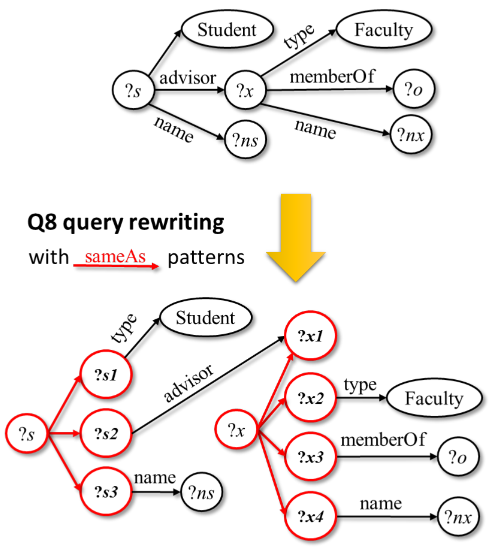


Fig. 6. SAM rewriting for Q_8 query

```
Id(pDoc1) type PostDoc
Id(pDoc2) name "Mary"
Id(pDoc3) email "mary@gmail.com"
Id(pDoc1) sameAs Id(pDoc2)
Id(pDoc1) sameAs Id(pDoc3)
```

Consider the query:

```
select ?x, ?x1, ?x2 where {
?x sameAs ?x1.  ?x1 type PostDoc.
?x sameAs ?x2.  ?x2 name ?n. }
```

That query result is empty because the triple pattern $?x$ `sameAs` $?x_1$ does not bind to $?x = Id(pDoc1)$ and $?x_1 = Id(pDoc1)$ due to the absence of the reflexive `sameAs` triple in the dataset.

To overcome this limitation, while keeping the materialized data as small as possible, we devise an extended query evaluation process. The idea is to take into account the implicit reflexive `sameAs` triples while joining a non-`sameAs` triple pattern with a `sameAs` one in order to ensure that the result is complete. The key phases of the query evaluation are:

(i) Decomposition. A query containing n non-`sameAs` triple patterns is decomposed into n chains (or sub-queries). A chain contains one non-`sameAs` triple pattern and the `sameAs` patterns it is joined to. A chain has exactly one non-`sameAs` triple pattern and at most 2 `sameAs` triple patterns. For example, the decomposition for query Q_8 in Section 6 has 6 chains, among which a chain of length 2 is `?x sameAs ?x3. ?x3 memberOf ?o` and a chain of length 3 is `?s sameAs ?s2. ?s2 advisor ?x1. ?x sameAs ?x1.`

(ii) Planning. Based on the chains that somehow hide the `sameAs` patterns, the query planner assesses a join order and generates an execution plan as usual (ignoring the `sameAs` patterns).

(iii) Execution. During the query execution phase, if a chain contains a `sameAs` pattern then a dedicated operator ensures that all the bindings are produced. More precisely, to execute the chain `?x sameAs?y. ?y p ?z` consists in evaluating the triple pattern `?y p ?z` which results in a set of $(?y, ?z)$ bindings. Then, for each binding, produce a set of $(?x, ?y, ?z)$ bindings such that $?x$ binds to the $?y$ value and also to each individual equivalent to $?y$ value. This ensures a complete result.

7. Evaluation

Putting together the contributions presented in Sections 5 and 6, we are able to combine LiteMat with one of the two methods to reason over `sameAs` individuals, denoted RB (representative-based) and SAM (SameAs Materialization). It thus defines two forms of reasoners for RDFS with `sameAs`:

– the LiteMat + RB approach is, in most use cases, the best performing approach and is hence the default approach.
– the LiteMat + SAM provides additional features, *e.g.*, a need for origin-preserving scenario, and improves the processing performance of BGPs containing a single triple pattern with inference.

7.1. Computing Setup

We evaluate Strider^R [28] on an Amazon EC2/EMR cluster of 11 machines (type m3.xlarge) and manage resources with Yarn. Each machine has 4 CPU virtual cores of 2.6 GHz Intel Xeon E5-2670, 15 GB RAM, 80 GB SSD, and 500 MB/s bandwidth. The cluster consists of 2 nodes for data flow management via the Kafka broker (version 0.8.x) and Zookeeper (version 3.5.x)[18], 9 nodes for Spark cluster (1 master, 8 workers, 16 executors). We use Apache Spark 2.0.2, Scala 2.11.7 and Java 8 in our experiment. The number of partitions for message topic is 16, generated stream rate is around 200,000 triples/second.

7.2. Datasets, Queries and Performance metrics

As explained in Section 4, we can not use any existing RSP benchmarks[3, 25, 35] to evaluate the performances of Strider^R. Hence, we are using our LUBM-based stream generator configured with 10 universities, *i.e.*, 1.4 million triples. For the purpose of our experimentation, we extended LUBM with triples containing the `sameAs` property. This extension requires to set two parameters: the number of cliques in a dataset and the number of distinct individuals per clique. To define these parameters realistically, we ran an evaluation over different LOD datasets. The results are presented in Table 3. It highlights that although the number of cliques can be very large (over a million in Yago), the number of individuals per clique is rather low, *i.e.*, a couple of individuals. Given the size of our dataset, we will run most of our experimentations with 1,000 cliques and an average of 10 individuals per clique, denoted 1k-10. Nevertheless, on queries requiring this form of reasoning, we will stress Strider^R with up to 5,000 cliques and an average of 100 individuals per clique (see Fig.9 for more details). More precisely, we will experiment with the following configurations: 1k-10, 2k-10, 5k-10, 1k-25, 1k-50 and 1k-100. For the SAM approach, the number of materialized triples can be computed by $nc * ipc$ with nc the number of cliques and ipc the number of individuals per clique.

We have defined a set of 8 queries[7] to run our evaluation (see Appendix for details). Queries Q1 to Q5 are limited to concept or/and property subsumption reasoning tasks. Query Q6 implies sameAs only inferences while Q7 and Q8 mix subsumptions and sameAs inferences.

Finally, we need to define which dimensions we want to evaluate. According to *Benchmarking Streaming Computation Engines at Yahoo!*[8], a recent benchmark for modern distributed stream processing framework, we take system throughput and query latency as two performance metrics. In this paper, throughput refers to how many triples can be processed in a unit of time (*e.g.*, triples per second). Latency indicates the time consumed by an RSP engine between the arrival of the input and the generation of its output. More precisely, for a windowing buffer w_i of the i-th query execution containing N triples and executed in t_i, then $throughput = \frac{N}{t_i}$ and the $latency = t_i$.

[7]https://github.com/renxiangnan/strider/wiki

[8]https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at

7.3. Quantifying joins and unions over reasoning approaches

As stated before, we can not compare Strider^R to other available RSP systems. This is mainly due to the high stream rate generated of our experiment which can not be supported by state-of-the-art reasoning-enabled RSPs, *e.g.*, C-SPARQL and SparqlStream. This is probably due to the lack of data flow management scalability of in these RSPs. In fact, their design was not intended for large-scale streaming data processing. Moreover, RSP system that could handle such rate either do not support reasoning or are not open source, *e.g.*, CQELS-cloud.

To assess the performance benefit of our solution for processing complex queries, specially comprising many joins, we compare LiteMat + RB and Lite + SAM with a more classical query rewriting approach. This combines SAM with UNION clauses between combinations of BGP reformulation (this approach is henceforth denoted UNION + SAM). Notice that the UNION + SAM approach acts as a baseline for our experiments. Such a rewriting comes at the cost of increasing the number of joins. Table 4 sums up the join and union operations involved in the 8 queries of our experimentation. In particular, queries Q5, Q7 and Q8 present an important number of joins (resp. 90, 45 and 180) due to a large number of union clauses (resp. 17, 14, 29).

datasets	#triples	#sameAs cliques	max	avg
Yago*	3696623	3696622	2	2
Drugbank	4215954	7678	2	2
Biomodels	2650964	187764	2	1.95
SGD	14617696	15235	8	3
OMIM	9496062	22392	2	2

Table 3

SamesAs statistics on LOD datasets (ipc = number of distinct individuals per sameAs clique, max and avg denotes resp. the maximum and average of ipc,*: subsets containing only `sameAs` triples with DBpedia, Biomodels contains triples of the form *a* `sameAs` *a*

7.4. Results evaluation & Discussion

The window size for involved continuous SPARQL queries with LiteMat reasoning support is set to 10 seconds, which is large enough to hold all the data generated from the dataset. However, since the impacts of extra data volume and more complex overheads are introduced in SAM query processing, we have to increase the window size (up to 60 seconds) to en-

Queries	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
# Joins								
LMRB	1	4	0	2	5	2	2	5
USAM	7	84	0	42	210	2	120	420
USAM*	3	24	0	18	90	2	60	210
# Union keywords								
USAM	6	20	3	20	41	0	29	59
USAM*	2	5	2	8	17	0	14	29
# Filter clauses								
LMRB	1	2	1	2	3	0	2	3

Table 4

Number of joins, unions and filter per query for LiteMat + RB (LMRB) and UNION + SAM (USAM) approaches. Here, the number of UNIONs correspond to the number of UNION keywords. USAM* relies on a simplified LUBM ontology

sure that both LiteMat and SAM approaches return the same result. In a nutshell, we approximately adjust the window size and the incoming stream rate by checking the materialized data volume.

All the evaluation results include the cost of LiteMat encoding and, for the LiteMat + SAM solution, the cost of `sameAs` triple materialization. Figure 7 reports the throughput and query latency of Q1 to Q5. Reasoning LiteMat + RB achieves the highest throughput (up to 2 millions triples/seconds) and the query latency remains at the second-level. To the best of our knowledge, such performances have not been achieved by any existing RSP engines. When both original and rewritten query patterns are relatively simple, *e.g.*, Q1 and Q2, LiteMat + RB has 30% gain over UNION+SAM on throughput and latency. The improvement gain of LiteMat + RB over UNION + SAM is increasing for queries involving multiple inferences, *i.e.*, Q4 is 75% faster. For Q5, UNION + SAM does not even terminate. This is mainly due to the insufficient computing resources (*e.g.*, number of CPU cores, memories) on Spark driver nodes, which is not capable of handling such intensive overheads for jobs/-tasks scheduling and memory management. Nevertheless, UNION + SAM is more efficient than LiteMat + RB on Q3 which contains a single triple pattern needing to reason over a property hierarchy of length two. In fact, a filter operator to evaluate a range predicate (LiteMat + RB) is longer to process than evaluating three equality predicates (UNION + SAM).

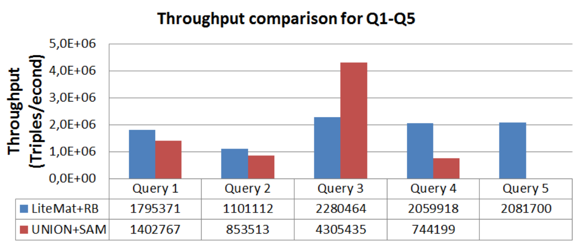
Figures 8 and 9 illustrate the impact on engine throughput and latency of Q6 to Q8 with varying `sameAs` clique sizes. As noted previously, “1K-10" means 1,000 cliques, and 10 individuals per clique. The number of materialized triples for `sameAs` reasoning support follows the $nc * ipc$ formula presented in Section 7.2. The number of materialized triples obviously increases with greater number of cliques and/or number of individuals per clique. The data throughput and latency can only be compared on Q6 since on Q7 and Q8, LiteMat + SAM does not terminate. The same non termination issue than on Q5 is observed (Q7 and Q8 respectively have 60 joins and 210 joins). Although stream rate is controlled at a low level, the system quickly fails after the query execution is triggered. Data throughput and latency is always better for LiteMat + RB than LiteMat + SAM by up to respectively two and three times. For the same computing setting, when the number of individuals per clique increases for a given number of cliques or when the number of cliques increases for the same number of individuals per clique, the performances of the LiteMat approaches decrease.

The same evolution for LiteMat + RB is witnessed on the more complex Q7 and Q8 queries. Nevertheless, for these queries, a throughput of over 800,000 triples per second can be achieved. Given our computing setting of 11 machines, this is still a major breakthrough compared to existing RSP engines. Moreover these systems are currently not able to support important constructors such as `sameAs`.

7.5. Cost analysis of the SAM approach

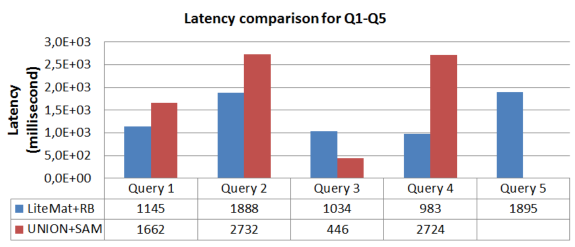
The SAM approach allows for retrieving the specific members of a clique that match the original dataset. As explained in Section 6.3.4, we propose an evaluation strategy that efficiently generates the result of a join operation between a non-`sameAs` pattern and its adjacent `sameAs` patterns, without actually processing the join. For example, SAM only executes 5 out of 12 joins for query Q_8, the result of each of the remaining 7 joins is directly obtained from the clique metadata, (see $Cl(x)$ in Table 2). The SAM approach implies to customize the query engine and add the specific logic for joining `sameAs` triple patterns. Therefore, SAM only suits to extensible query processors and prevents a ‘black box’ SPARQL processors from being used.

Due to the extensible Spark APIs, it is possible to implement such an approach. It would more difficult to obtain such a behavior with an out-of-the box SPARQL query processor.



	Query 1	Query 2	Query 3	Query 4	Query 5
🟦 LiteMat+RB	1795371	1101112	2280464	2059918	2081700
🟥 UNION+SAM	1402767	853513	4305435	744199	

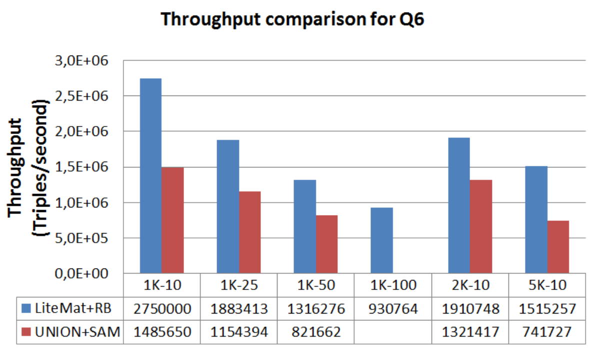
(a) Throughput Comparison for **Q1** to **Q5**



	Query 1	Query 2	Query 3	Query 4	Query 5
🟦 LiteMat+RB	1145	1888	1034	983	1895
🟥 UNION+SAM	1662	2732	446	2724	

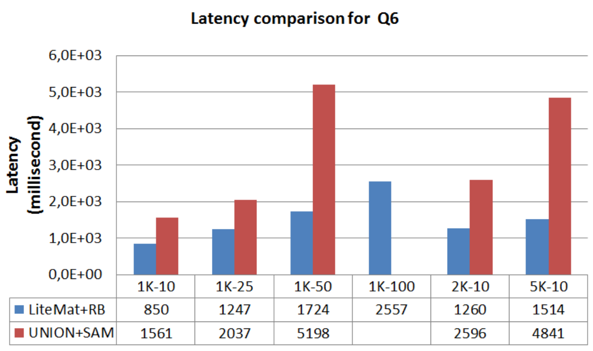
(b) Latency Comparison for **Q1** to **Q5**

Fig. 7. Throughput, Latency Comparison between LiteMat+RB and UNION+SAM for **Q1** to **Q5**



	1K-10	1K-25	1K-50	1K-100	2K-10	5K-10
🟦 LiteMat+RB	2750000	1883413	1316276	930764	1910748	1515257
🟥 UNION+SAM	1485650	1154394	821662		1321417	741727

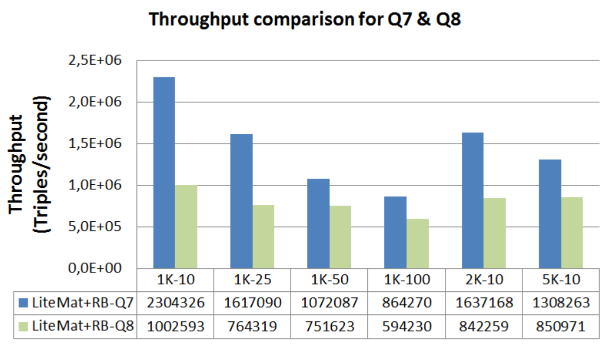
(a) Throughput Comparison for **Q6**



	1K-10	1K-25	1K-50	1K-100	2K-10	5K-10
🟦 LiteMat+RB	850	1247	1724	2557	1260	1514
🟥 UNION+SAM	1561	2037	5198		2596	4841

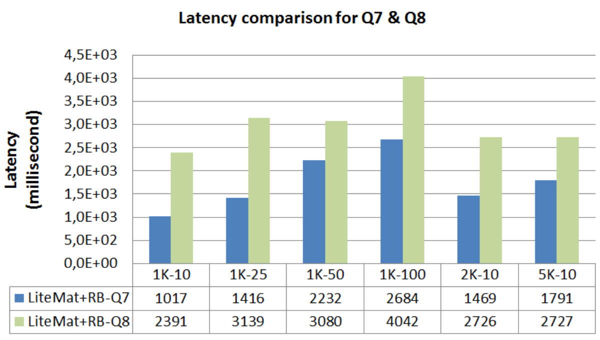
(b) Latency Comparison for **Q6**

Fig. 8. Throughput, Latency Comparison between LiteMat+RB and UNION+SAM for **Q6** by varying the size of clique.



	1K-10	1K-25	1K-50	1K-100	2K-10	5K-10
🟦 LiteMat+RB-Q7	2304326	1617090	1072087	864270	1637168	1308263
🟩 LiteMat+RB-Q8	1002593	764319	751623	594230	842259	850971

(a) Throughput Comparison for **Q7** and **Q8**



	1K-10	1K-25	1K-50	1K-100	2K-10	5K-10
🟦 LiteMat+RB-Q7	1017	1416	2232	2684	1469	1791
🟩 LiteMat+RB-Q8	2391	3139	3080	4042	2726	2727

(b) Latency Comparison for **Q7** and **Q8**

Fig. 9. Throughput, Latency Comparison of LiteMat+RB for **Q7** and **Q8** by varying the size of clique.

8. Related work

This section considers two related work fields: reasoning over ontologies supporting `sameAs` and RSP. Most RDF stores are using a more or less advanced form of encoding and do not adopt a full materialization approach due to its inefficiency. Concerning the support of `sameAs` inferences, GraphDB Enterprise Edition [9] and RDFox [24] are using a representative-based approach but do not handle stream processing. For instance, RDFox elects a representative among elements of a `sameAs` clique using a naive lexicographic order. Then all occurrences of individuals of a

[9]http://graphdb.ontotext.com/

clique are replaced by the representative. This presents drawbacks when updates are performed on the clique, *e.g.*, removing the representative from the clique. In such situations, the original dataset has to be processed again. In a streaming context, data streams are ephemeral, *i.e.*, do not need to be stored, and the update ontology issue is not a limitation. Intuitively, if a concept or property is updated (inserted, deleted or modified), it is going to be considered in the next processing window and the previous stream query answer sets will still hold in the context of the previous ontology's state. [26] follows on the work of RDFox but considers a distributed approach. Nevertheless, the system is not fault tolerant and does not addresses stream processing. The Kognac system[29] proposes an intelligent encoding of RDF terms for large KBs. It is designed on a combination of estimated frequency-based encoding and semantic clustering. Nevertheless, Kognac is not designed with inferences in mind and its implementation is not distributed and thus can not scale to very large KBs. Laser[6] is a stream reasoning system based on a tractable fragment of LARS[7], *i.e.*, an extension of Answer Set Programming for stream processing). The reasoning services of Laser are supported by a set of rules which are expressed in a specific syntax. We consider that this may prevent Laser's adoption by end-users. Moreover, Laser is not distributed and is thus not able to process very large data streams.

The first RSP engines [4, 5, 8, 21] have emerged around 2009. Their original focus was on the design of continuous query languages based on SPARQL. Scalability and reasoning are now considered as primordial features. [19] was among the first systems to concentrate on the scalability of RDFS stream reasoning. The engine is able to reach throughputs around hundreds of thousand triples/seconds within 32 computing nodes. However, [19] does not include `sameAs` in the scope of consideration. C-SPARQL [5] , CQELS [21], SparqlStream [8] and ETALIS [4] are prominent RSP systems. C-SPARQL and SparqlStream tackle the requirements of stream reasoning via respectively data materialization and query rewriting. The proposed straightforward solutions, *i.e.*, data materialization and query rewriting show limitations on system's scalability. Moreover the centralized design of C-SPARQL and SparqlStream can not support complex reasoning task over massive RDF data stream. On the other hand, distributed RSP engines, *e.g.*, CQELS-Cloud, address flexibility and scalability issues but do not possess any real-time stream reasoning. Using standards such Apache Kafka and Spark streaming enabled Strider^R to increase throughput with less computing power while being able to reason over RDFS with `sameAs`.

9. Conclusion

In this paper, we have presented the integration of several reasoning approaches for RDFS plus `sameAs` within our Strider RSP engine. For most queries, LiteMat together with the representative-based (RB) approach for `sameAs` cliques is the most efficient. Nevertheless, LiteMat + SAM proposes an unprecedented provenance-awareness feature that can not be obtained in other approaches. Lite + SAM can also be useful for very simple queries, *e.g.*, a single triple patter in the WHERE clause. To the best of our knowledge, this is the first scalable, production-ready RSP system to support such an ontology expressiveness. Via a thorough evaluation, we have demonstrated the pertinence of our system to reason with low latency over high throughput data streams.

As future work, we will investigate novel semantic partitioning solutions. This could be applied to elements such as dictionaries, streaming data and continuous queries. We are aiming to support data streams that would update the ontology and thus our dictionaries. Finally, we are also working on increasing the expressiveness of supported ontologies, *e.g.*, including transitive properties.

References

[1] Sparql 1.1 query language. http://www.w3.org/TR/sparql11-query/, 2013.

[2] Rdf 1.1 schema. http://www.w3.org/TR/2014/REC-rdf-schema-20140225/Overview.html, 2014.

[3] M. I. Ali, F. Gao, and A. Mileo. Citybench: A configurable benchmark to evaluate rsp engines using smart city datasets. In *ISWC*, pages 374–389, 2015.

[4] D. Anicic, S. Rudolph, P. Fodor, and N. Stojanovic. Stream reasoning and complex event processing in ETALIS. *Semantic Web journal*, pages 397–407, 2012.

[5] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, and M. Grossniklaus. C-SPARQL: SPARQL for continuous querying. In *WWW*, pages 1061–1062, 2009.

[6] H. R. Bazoobandi, H. Beck, and J. Urbani. Expressive stream reasoning with laser. *ISWC*, pages 87–103, 2017.

[7] H. Beck, M. Dao-Tran, T. Eiter, and M. Fink. LARS: A logic-based framework for analyzing reasoning over streams. In *AAAI*, 2015.

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

[9] O. Curé and G. Blin. *RDF Database Systems: Triples Storage and SPARQL Query Processing, 1st Edition*. Morgan Kaufmann, 2014.

[10] O. Curé, H. Naacke, T. Randriamalala, and B. Amann. Litemat: A scalable, cost-efficient inference encoding scheme for large RDF graphs. In *IEEE Big Data*, pages 1823–1830, 2015.

[11] O. Curé and G. Blin. *RDF Database Systems: Triples Storage and SPARQL Query Processing*. Morgan Kaufmann Publishers Inc., 1st edition, 2014.

[12] B. Glimm, I. Horrocks, B. Motik, and G. Stoilos. Optimising ontology classification. In *ISWC*, pages 225–240, 2010.

[13] K. Goodhope, J. Koshy, J. Kreps, N. Narkhede, R. Park, J. Rao, and V. Y. Ye. Building linkedin's real-time activity data pipeline. *IEEE Data Eng. Bull.*, 35(2):33–45, 2012.

[14] J. Greiner. A comparison of parallel algorithms for connected components. In *ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 16–25. ACM, 1994.

[15] Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *Journal of Web Semantics*, 2005.

[16] H. Halpin, P. J. Hayes, J. P. McCusker, D. L. McGuinness, and H. S. Thompson. When owl: sameas isn't the same: An analysis of identity in linked data. In *ISWC*, pages 305–320, 2010.

[17] P. Hayes. RDF semantics, W3C recommendation. http://www.w3.org/tr/rdf-mt/, 2004.

[18] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX*, 2010.

[19] H. Jesper and K. Spyros. High-performance distributed stream reasoning using s4. In *Ordring Workshop at ISWC*, 2011.

[20] M. Kleppmann and J. Kreps. Kafka, samza and the unix philosophy of distributed data. *IEEE Data Eng. Bull.*, 38(4):4–14, 2015.

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

[22] A. Mileo, A. Abdelrahman, S. Policarpio, and M. Hauswirth. Streamrule: A nonmonotonic stream reasoning system for the semantic web. In *Int'l Conf. on Web Reasoning and Rule Systems*, pages 247–252, 2013.

[23] S. Muñoz, J. Pérez, and C. Gutierrez. Minimal deductive systems for rdf. In *ESWC*, 2007.

[24] Y. Nenov, R. Piro, B. Motik, I. Horrocks, Z. Wu, and J. Banerjee. Rdfox: A highly-scalable RDF store. In *ISWC*, pages 3–20, 2015.

[25] D. L. Phuoc, M. Dao-Tran, M. Pham, P. A. Boncz, T. Eiter, and M. Fink. Linked stream data processing engines: Facts and figures. In *ISWC*, pages 300–312, 2012.

[26] A. Potter, B. Motik, Y. Nenov, and I. Horrocks. Distributed RDF query answering with dynamic data exchange. In *ISWC*, pages 480–497, 2016.

[27] X. Ren and O. Curé. Strider: A hybrid adaptive distributed RDF stream processing engine. In *ISWC*, pages 559–576, 2017.

[28] X. Ren, O. Curé, L. Ke, J. Lhez, B. Belabbess, T. Randriamalala, Y. Zheng, and G. Képéklian. Strider: An adaptive, inference-enabled distributed RDF stream processing engine. *PVLDB*, 10(12):1905–1908, 2017.

[29] J. Urbani, S. Dutta, S. Gurajada, and G. Weikum. Kognac: Efficient encoding of large knowledge graphs. pages 3896–3902, 2016.

[30] J. Urbani, A. Margara, C. Jacobs, F. Harmelen, and H. Bal. Dynamite: Parallel materialization of dynamic rdf data. In *ISWC*, pages 657–672, 2013.

[31] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.

[32] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: fault-tolerant streaming computation at scale. In M. Kaminsky and M. Dahlin, editors, *SOSP*, pages 423–438. ACM, 2013.

[33] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, 2013.

[34] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache spark: a unified engine for big data processing. *Commun. ACM*, pages 56–65, 2016.

[35] Y. Zhang, P. M. Duc, O. Corcho, and J.-P. Calbimonte. Srbench: A streaming rdf/sparql benchmark. In *ISWC*, pages 641–657, 2012.

Appendix A. Queries

In this Appendix section, we provide six out of eight (Q4 and Q6 have already been presented in Section 4) SPARQL queries evaluated in Section 7. In all of them, we are using the `rdf` and `lubm` namespaces which respectively correspond to http://www.w3.org/1999/02/22-rdf-syntax-ns# and http://swat.cse.lehigh.edu/onto/univ-bench.owl#.

A.1. Queries with inferences over concept hierarchies

Q1: Inferences are required on the Professor concept which has no direct instances in LUBM datasets.

```
SELECT ?n WHERE {
  ?x rdf:type lubm:Professor;
     lubm:name ?n.}
```

Q2: Inferences are required on both the Professor and Student concepts.

```
SELECT ?ns ?nx WHERE {
 ?x rdf:type lubm:Professor; lubm:name ?nx.
 ?s lubm:advisor ?x; rdf:type lubm:Student.
 ?s lubm:name ?ns. }
```

A.2. Query with inferences over property hierarchies

Q3: Inferences are required for the memberOf property which has on direct sub property and one indirect sub property.

```
SELECT ?x ?o  WHERE { ?x lubm:memberOf ?o.}
```

A.3. Queries with inferences over both concept and property hierarchies

Q5: This query goes further than Q4 by mixing Q2 and Q3, i.e., it requires reasoning over the Professor and Student concept hierarchies and the memberOf property hierarchy.

```
SELECT ?ns ?nx ?o WHERE {
 ?x rdf:type lubm:Professor; lubm:name ?nx;
 lubm:memberOf ?o.
 ?s lubm:advisor ?x; rdf:type lubm:Student;
 lubm:name ?ns. }
```

A.4. Queries with inferences over concept, property hierarchies and owl:sameAs

Q7: Inferences over the Faculty concept hierarchy, which includes PostDoc sameAs individuals and the memberOf property.

```
SELECT ?o ?n WHERE {
 ?x rdf:type lubm:Faculty; memberOf ?o;
 lubm:name ?n.}
```

Q8: The most complex query of our evaluation with two inferences over concept hierarchies (Faculty and Student), with the former containing sameAs individual cliques, and inferences over the memberOf property hierarchy.

```
SELECT ?ns ?nx ?o WHERE {
 ?x rdf:type lubm:Faculty; lubm:name ?nx;
 lubm:memberOf ?o.
 ?s lubm:advisor ?x; rdf:type lubm:Student;
 lubm:name ?ns.}
```

Appendix B. Details on our continuous query extension

The `STREAMING` clause is used to initialize a Spark Streaming context. As in other RSP query languages, the `WINDOW` and `SLIDE` keywords respectively specify the range and size of a windowing operator. Since Spark Streaming is based on a micro-batch processing model, we defined a `BATCH` clause to assign the time interval of each micro-batch. Basically, a single micro-batch represents a RDD, Spark's main abstraction. For each triggered query execution, Spark Streaming receives a segment of Dstreams which essentially consists of an RDD sequence.

The `STREAMING` clause is used to initialize a Spark Streaming context. As in other RSP query languages, the `WINDOW` and `SLIDE` keywords respectively specify the range and size of a windowing operator. Since Spark Streaming is based on a micro-batch processing model, we defined a `BATCH` clause to assign the time interval of each micro-batch. Basically, a single micro-batch represents a RDD, Spark's main abstraction. For each triggered query execution, Spark Streaming receives a segment of Dstreams which essentially consists of an RDD sequence.

The `REGISTER` clause concerns the SPARQL queries to be processed. Strider^R allows to register multiple queries, and uses a thread pool to launch all registered queries asynchronously. However, the optimization of multiple SPARQL queries is beyond the scope of this paper. Inside `REGISTER`, each continuous SPARQL query possesses a query ID. The `REASONING` clause enables the end-user to select a combination of concept/property hierarchy and `sameAs` inferences. Once `REASONING` service is triggered, Strider^R automatically rewrites the given SPARQL query to its LiteMat mapping. Moreover, incoming data stream will also be encoded within the rules of LiteMat KBs.