Strider: An Adaptive, Inference-enabled Distributed RDF Stream Processing Engine

Xiangnan Ren[1,2] Olivier Curé[2] Li Ke[1] Jeremy Lhez[2]
Badre Belabbess[1,2] Tendry Randriamalala[1] Yufan Zheng[1] Gabriel Kepeklian[1]
[1]ATOS, 80 quai Voltaire, 95870 Bezons, France. {xiang-nan.ren, firstname.lastname}@atos.net
[2]UPEM LIGM - UMR CNRS 8049, 77454 Marne-la-Vallée, France. {firstname.lastname}@u-pem.fr

ABSTRACT

Real-time processing of data streams emanating from sensors is becoming a common task in industrial scenarios. An increasing number of processing jobs executed over such platforms are requiring reasoning mechanisms. The key implementation goal is thus to efficiently handle massive incoming data streams and support reasoning, data analytic services. Moreover, in an on-going industrial project on anomaly detection in large potable water networks, we are facing the effect of dynamically changing data and work characteristics in stream processing. The Strider system addresses these research and implementation challenges by considering scalability, fault-tolerance, high throughput and acceptable latency properties. We will demonstrate the benefits of Strider on an Internet of Things-based real world and industrial setting.

1. INTRODUCTION

The vast amount of data produced by the Internet of Things (IoT) generally needs to be processed in almost real time. This is the case for an increasing number of industrial scenarios requiring to detect anomalies, identify commercial trends, diagnose machines' conditions, etc.. In the context of the Waves FUI (Fonds Unique Interministeriel) project[1], we are processing data streams emanating from sensors distributed over the potable water distribution network of a resource management international company. For France alone, this company distributes water to over 12 million clients through a network of more than 100.000 kilometers equipped with thousands (and growing) of sensors. Our system's main objective is to automatically detect anomalies, *e.g.*, water leaks, from analyzed data streams. Obviously, the promptness and accuracy of our anomaly discoveries potentially impacts ecological (loss of cleaned up water) as well as economical (price of clients' consumed water) aspects.

These anomaly detections frequently depend on reasoning services. That is meta-data emitted by IoT sensors are first mapped to predefined vocabularies which are supporting logic-based inferences. For data integration, inference and stack homogeneity reasons, we have opted for the Semantic Web technology stack, *i.e.*, RDF(S), SPARQL and OWL. Hence, one of our goals is to design a system that efficiently integrates reasoning services within a real-time processing platform. This is performed by finding a trade-off between the two common reasoning mechanisms adopted in knowledge-based management systems, namely inference materialization and query rewriting. Our solution consists in creating semantic-aware dictionaries for the concepts, predicates and instances of managed vocabularies and to use the properties of attributed identifiers to optimize query rewriting. We thus minimize the size of stored knowledge bases (KBs) and of data streams, as well as reduce the size and the time needed to reformulate continuous queries.

In real world scenarios, we are frequently facing dynamically changing data and workload characteristics, *e.g.*, a sensor could emit different types of messages based on the user requests. These changes impact the execution performance of continuous queries executed over data streams. To ensure the efficient query execution, the execution plan may have to change during its usually long-term lifetime. Our problem is exacerbated by the hardness of generating optimized query plans for SPARQL queries due to their potentially high number of joins compared to SQL queries. To support this behavior, we have implemented an adaptive query processing (henceforth AQP) [3] for continuous SPARQL queries and RDF data.

Moreover, processing large scale data streams is usually performed over a distributed setting which guarantees scalability, automatic work distribution and fault tolerance but also has to satisfy high throughput and acceptable latency constraints. Such systems are better designed and operated upon when implemented on top of robust, proven engines such as Apache Kafka (a distributed commit log which efficiently stores and delivers data streams) and Apache Spark (a general purpose and unified cluster computing framework). Hence, our inference-enabled SPARQL AQP needs to fit efficiently into this distributed setting. We consider that such an AQP does not exist due to a high expertise entry point (distributed systems, database management systems and Semantic Web) when starting this kind of project. Thus, it is not a surprise that, to the best of our knowledge, no industry targeted production-ready systems are currently available. In fact, the RDF Stream Processing (RSP) ecosystem regroups engines that are either (i) distributed but lacking important features, *e.g.*, Katts[4], or are not open-source,

[1]http://www.waves-rsp.org/

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.
Proceedings of the VLDB Endowment, Vol. 10, No. 12
Copyright 2017 VLDB Endowment 2150-8097/17/08.

e.g., CQELS Cloud[5], or (ii) centralized and hence can not support high throughput (C-SPARQL[1]). A common characteristics of these engines is to consider that the structure of the data stream does not change. While facing Waves' real-world use cases, we found out that one can not assume the regularity of incoming streaming data. Finally, we consider that the best of breed in SPARQL AQP necessarily needs to mix static (heuristic-based) and dynamic (cost-based) query optimization approaches.

2. THE STRIDER SYSTEM

In this section, we first present an overview of the Strider system, detail the reasoning as well as query optimization components. Finally implementation aspects are considered.

2.1 Architecture overview

Figure 1 gives a high-level overview of the system's architecture. The left hand-side gives details on the application's data flow. Its design is relatively standard and follows the approach generally adopted in stream processing systems. In a nutshell, data sources (IoT sensors) are sending messages to a publish-subscribe layer. That layer emits messages for the streaming layer which executes registered queries. The main originality of our approach consists in transforming the data source messages, *e.g.*, csv files, into an RDF serialization for data integration and reasoning purposes.

On the right hand-side of Figure 1, we concentrate on components related to the system's implementation. The Encoding layer runs off-line and pre-processes the encoding of all KB elements, *i.e.*, concepts, predicates and instances, into integer values using a semantic-aware approach. This component interacts with the RDF event converter and the Inference layer. The Request layer registers continuous queries which are later sent to the Parsing layer to compute a first version of a query plan. Plans involving any form of reasoning are extended by the Inference layer. These new plans are pushed to the Optimization layer which consists of three collaborating sub-components: static and adaptive optimizations as well as a trigger mechanism. Finally, the Query Processing layer sets off the query execution right after the optimized logical plan takes place.

2.2 Encoding and Inference layers

Upfront to any data stream processing, the Encoding Layer encodes concepts, predicates and instances of registered KBs. In the remaining of this paper, we consider that a KB consists of a schema, aka ontology or terminological box (Tbox) and a fact base, aka assertional box (Abox). With this Encoding layer, we aim to provide efficient encoding scheme and data structures to support the reasoning services associated to the input ontology of an application. The input ontology is considered to be union of (potentially aligned) ontologies necessary to operate over one's application domain. In the current version of our work, we address the ρdf subset of RDFS, meaning that we are only interested in the `rdfs:subClassOf`, `rdfs:subPropertyOf`, `rdfs:domain` and `rdfs:range` constructors. To address inferences drawn from these first two RDFS predicates, we attribute numerical identifiers to ontology terms, *i.e.*, , concepts and predicates. 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

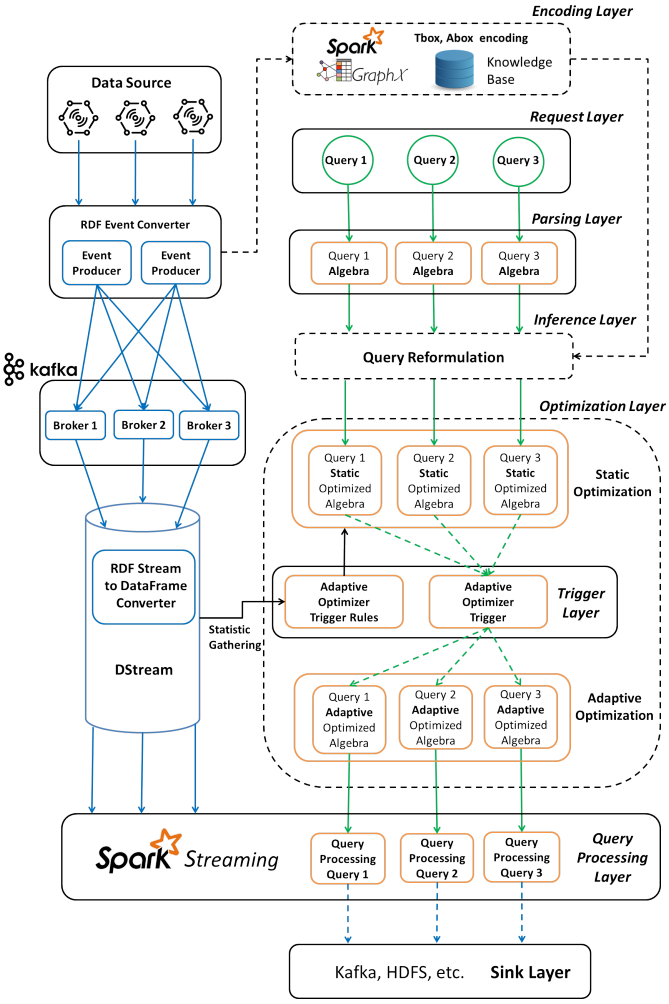

Figure 1: Strider Architecture

prefixing the encoding of a term with the encoding of its direct parent (a workaround is proposed to support multiple inheritance). This approach only works if an encoding is computed using a binary representation. More details on the encoding scheme can be obtained in [2].

Once this ontology pre-processing has been computed, we can use the generated dictionaries to encode all incoming data streams. In fact, the schema of all our domain's sensor messages are known in advanced and have been mapped to ontology elements. Thus these messages can be represented as subject, predicate and object triples. Using our dictionaries, we can represent these triples with element identifiers. Not all triple entries are transformed using our dictionaries. For instance, numerical values (highly frequent in IoT) and blank nodes are not transformed. All predicates, concepts and domain constants are transformed. Following an analysis of the queries associated to Kafka topics, some RDF triples can be introduced in the data streams, and hence potentially make the query satisfiable.

The semantic-aware encoding of concepts and predicates supports the reasoning services required at query processing time. For instance, consider a query asking for the pressure value of sensors of type S1. This would be expressed as the following two triple patterns: `?x pressureValue ?v. ?x type S1`. In the case sensor concept S1 has n sub-concepts,

then a naive query reformulation requires to run the union of n+1 queries. With our semantic-aware encoding, we are able, using two bit-shift operations, to compute the identifier interval, *i.e.*, [lowerBound, upperBound[, of all direct and indirect sub-concepts of S1. And thus we can compute this query with a simple reformulation: (i) replacing the concept S1 with a new variable : `?x type ?newVar` and (ii) introducing a filter clause constraining values of this new variable: `FILTER (?newVar>=lowerBound && ?newVar<upperBound)`.

The main benefits of our tight inference and stream processing integration supports the retrieval of all correct answers of continuous queries based on a trade-off between inference materialization and query reformulation.

2.3 Optimization layer

Conventional SPARQL query optimization techniques with data pre-processing, *e.g.*, data indexing, statistic information maintaining, are not appropriate within the requirement of real-time or near real-time aspect data processing. Besides, due to the schema-free, graph nature of RDF data, a navigation-based query language such as SPARQL potentially involves a large amount of joins and self-joins. In a distributed streaming context, this behavior becomes a performance bottleneck since a join task might lead to data shuffling which causes heavy network communications. To cope with the previously mentioned performance issue, and inspired by the state of the art efforts [3, 6, 7], we build our query optimizer with hybrid static and adaptive optimization strategies.

In the Request layer, Strider allows to register multiple queries at once. The input SPARQL queries are submitted through different threads, and executed concurrently with respect to the execution plans which are generated by the optimizer. Currently, we consider that the input queries are independent, thus a multi-query optimization approach (*e.g.*, sub-query sharing) is not in the scope of the current state of Strider.

As described in Section 2.1, the Optimization layer possesses three sub-layers: static and adaptive optimizations, trigger. Fundamentally, both static and adaptive optimizations are processed using a graph G, denoted as Undirected Connected Graph (UCG) [6], which is formed of vertexes (triple patterns) and edges (joins between triple patterns). The weight of UCG's vertexes and edges correspond to the selectivity of triple patterns and join patterns, respectively. Once a UCG is initialized, the query planner automatically generates an optimal logical plan and triggers a query execution.

Static Optimization creates a UCG graph using a set of heuristic rules. The predefined heuristic rules set empirically assigns the weights for UCG vertexes and edges. Next, the query planner determines the shortest traversal path in the current UCG and generates the logical plan for query execution. The obtained logical plan presents the query execution pipeline which is kept by the system permanently.

Our static optimization strategy depends solely on the query shape and is independent from the data set. Practically, this approach could be an acceptable approximation, since it applies a basic optimization and simplifies the implementation. However, the static optimization can not guarantee to return the optimal query plan for all input streams, and it does not cover the situation when the structure of data stream may change. In order to remedy this defect, we add an adaptive query optimization component.

Trigger layer is the transition between the stages of static optimization and adaptive optimization. In a nutshell, the trigger layer is dedicated to notify the system whether it is necessary to proceed the adaptive optimization. Our adaptive strategy requires collecting statistic information at run-time which in a distributed environment involves a computation cost that is not negligible. The Strider prototype provides a set of straightforward trigger rules,*i.e.*, the adaptive algebra optimization is triggered by a configurable workload threshold. The threshold refers to two factors: 1) the input number of RDF events/triples; 2) the fraction of the estimated input data size and the allocated executors' heap memory.

When **Adaptive Optimization** is activated, the statistic information gathering and query evaluation occur synchronously, and the UCG elements will be assigned a so-call *statistic weight*. Next, at the same, the optimizer recomputes and evaluates the optimal algebra tree in a bottom-up fashion. Therefore, adaptive logical plan scheduling ensures that the system always executes a query in an optimal way. Here is an example of a Strider query:

```
STREAMING { WINDOW [ 20 SECONDS ]
            SLIDE [ 20 SECONDS ] BATCH [ 5 SECONDS ] }
REGISTER { QUERYID [ Q5 ]
SPARQL [
    prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
    prefix ssn: <http://purl.oclc.org/NET/ssnx/ssn/>
    prefix cuahsi: <http://www.cuahsi.org/waterML/>
    select ?s ?o1
    where { ?s ssn:hasValue ?o1 ; ssn:hasValue ?o2 ;
               ssn:hasValue ?o3 .
            ?o1 rdf:type cuahsi:flow .
            ?o2 rdf:type cuahsi:temperature .
            ?o3 rdf:type cuahsi:chlorine . }]}
```

Figure 2 illustrates Strider's adaptive optimization for `Q5`. The query UCG (Figure 2(a),(b)) and logical plan (Figure 2(c),(d)) change as the structure of data stream changes dynamically.

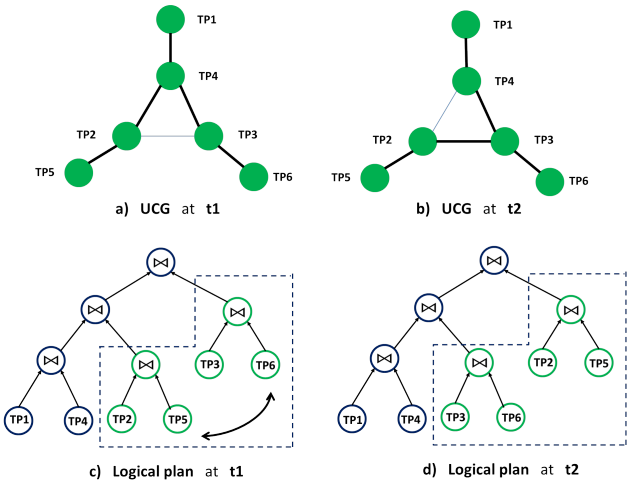

Figure 2: The query UCG and logical plan (TP denotes a triple pattern and t1,t2 computation times)

2.4 Implementation

Strider is written in Scala and contains two principle modules: *i*) data flow management. In order to ensure high throughput, fault-tolerant, and easy-to-use features, Strider uses Apache Kafka to manage input data flow. The incoming RDF streams are categorized into different *message topics*, which practically presents different types of RDF events. *ii*) computing core. Strider core is based on the Spark programming framework. Spark Streaming receives, maintains messages emitted from Kafka in parallel, and generates data processing pipeline. Comparing to other distributed stream processing frameworks, Spark represents a certain maturity and a rich ecosystem. This allows to achieve a horizontal expansion of the platform functionality. Besides, due to the coarse-grained feature of the micro-batch model, Spark Streaming provides high throughput and acceptable latency for tasks involving complex analytics.

The Encoding layer uses Apache Jena to parse the RDFS or OWL ontologies and an external reasoner, namely HermiT, to classify concept hierarchies. This part of Strider is the only component that does not run in parallel. This is due to the absence of an efficient, expressive, *i.e.*, higher than RDFS and OWLRL, reasoner that runs in parallel. This can not be considered as a limitation of the system since the processing is only performed when the set of ontologies are updated and Tboxes are known to be rather small compared to Aboxes. For instance, the encoding of Wikipedia, which contains over 213,000 triples and 350 predicates, takes less than 2 minutes. The Abox is encoded in parallel using the dictionaries stored as Spark's DataFrames.

To enable SPARQL query processing on Spark, Strider parses a query with Jena ARQ and obtains a query algebra tree in the Parsing layer. The system reconstructs the algebra tree into a new Abstract Syntax Tree (AST) based on Visitor model. Basically, the AST represents the logical plan of a query execution. Once the AST is created, it is pushed into the algebra Optimization layer. As stated in section 2.3, the system applies a hybrid static and adaptive optimization strategy on original query algebra. By traversing the AST, we bind the SPARQL operators to the corresponding Spark SQL relational operators for query evaluation.

3. DEMONSTRATION SCENARIOS

The demonstration[2] concentrates on real-world IoT use cases from the Waves project. We highlight two aspects of the system. In the first one, we show some user-oriented features, *e.g.*, Strider's user-friendly graphical interface. This allows fast, easy and intuitive Spark cluster configuration and deployment. In the second one, we demonstrate Strider's reasoning capability and hybrid optimization strategies for continuous SPARQL query execution via the the following scenarios:

Scenario 1 concentrates on the inference component. We begin with by emphasizing the efficiency of our KB encoding: Tbox encoding and Abox encoding. Then, we focus on the stream materialization and query reformulation aspects. This is demonstrated on data streams requiring some data materialization and on query reformulation of registered queries. Both processing durations and internal representations of the streams and queries will accessible.

[2]details of the demo at `https://github.com/renxiangnan/reference-vldb-demo-2017/wiki`

Scenario 2 mainly focuses on continuous SPARQL query processing with stable stream structure. In the ideal case, incoming RDF streams are supposed to be structurally stable. The proportions of variant types of RDF triples does not change over time. By deploying our system on a small cluster of Amazon EMR (one driver node, 3 to 4 worker nodes), we show that Strider can achieve a throughput between 400,000 and 600,000 triples per second with real queries.

Scenario 3 highlights the efficiency of our engine's adaptive query optimization. A group of Kafka message producers are configured. By randomly modifying the proportion of the different types of messages, we feed the structurally unstable RDF stream to the engine. To give an intuitive view on the system's adaptivity, we demonstrate the changing of the query execution plan in real-time. We also provide a comparison between conventional static optimization and adaptive query optimization. For instance, in Figure 3, an adaptive query execution shows a steady performance during a relative long (one hour) running time. On the contrary, the engine performance fluctuates substantially over time, and we have observed a large Garbage Collection pressure through log monitoring.

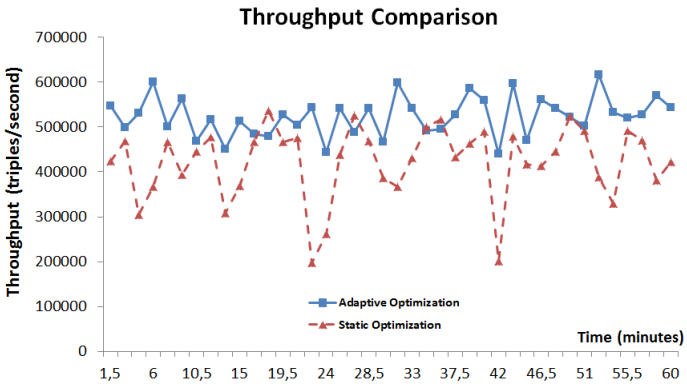

Figure 3: Throughput Comparison between Static and Adaptive Optimization (query `Q5`)

4. REFERENCES

[1] D. F. Barbieri and al. C-SPARQL: SPARQL for continuous querying. In *18th WWW*, 2009.

[2] O. Curé and al. Litemat: A scalable, cost-efficient inference encoding scheme for large RDF graphs. In *IEEE Big Data, 2015*.

[3] A. Deshpande and al. Adaptive query processing. *Foundations and Trends in Databases*, 2007.

[4] L. Fischer and al. Scalable linked data stream processing via network-aware workload scheduling. In *SSWS*, 2013.

[5] D. L. Phuoc and al. Elastic and scalable processing of linked stream data in the cloud. In *ISWC 2013*.

[6] M. Stocker and al. SPARQL basic graph pattern optimization using selectivity estimation. In *17th WWW, 2008*.

[7] P. Tsialiamanis and al. Heuristics-based query optimisation for SPARQL. In *15th EDBT, 2012*.