Strider: A Hybrid Adaptive Distributed RDF Stream Processing Engine

Xiangnan Ren[1,2], and Olivier Curé[2]

[1] ATOS - 80 Quai Voltaire, 95870 Bezons, France
xiang-nan.ren@atos.net
[2] LIGM (UMR 8049), CNRS, UPEM, F-77454, Marne-la-Vallée, France
olivier.cure@u-pem.fr

Abstract. Real-time processing of data streams emanating from sensors is becoming a common task in Internet of Things scenarios. The key implementation goal consists in efficiently handling massive incoming data streams and supporting advanced data analytics services like anomaly detection. In an on-going, industrial project, a $24/7$ available stream processing engine usually faces dynamically changing data and workload characteristics. These changes impact the engine's performance and reliability. We propose Strider, a hybrid adaptive distributed RDF Stream Processing engine that optimizes logical query plan according to the state of data streams. Strider has been designed to guarantee important industrial properties such as scalability, high availability, fault tolerance, high throughput and acceptable latency. These guarantees are obtained by designing the engine's architecture with state-of-the-art Apache components such as Spark and Kafka. We highlight the efficiency (*e.g.*, on a single machine machine, up to 60x gain on throughput compared to state-of-the-art systems, a throughput of 3.1 million triples/second on a 9 machines cluster, a major breakthrough in this system's category) of Strider on real-world and synthetic data sets.
Keywords: RDF Stream Processing, SPARQL, Adaptive Query Processing, Distributed Computing, Apache Spark

1 Introduction

With the growing use of Semantic Web Technology in Internet of Things (IoT) contexts, *e.g.*, for data integration and reasoning purposes, the requirement for almost real-time platforms that can efficiently adapt to large scale data streams, *i.e.*, continuous SPARQL query processing, is gaining more and more attention. In the context of the FUI (Fonds Unique Interministeriel) Waves project[1], we are processing data streams emanated from sensors distributed over the drinking 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. Obviously, our RDF Stream Processing (RSP) engine should satisfy some common industrial features, *e.g.*, high throughput, high availability, low latency, scalability and fault tolerance.

Querying over RDF data streams can be quite challenging. Due to fast generation rates and schema free natures of RDF data streams, a continuous SPARQL query usually involves intensive join tasks which may rapidly become a performance bottleneck.

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

Existing centralized RSP systems like C-SPARQL [4], CQELS [13] and ETALIS [3] are not capable of handling massive incoming data streams, as they do not benefit from task parallelism and the scalability of a computing cluster. Besides, most streaming systems are operating $24/7$ with patterns, *i.e.*, stream graph structures, that may change overtime (in terms of graph shapes and sizes). This can potentially have a performance impact on query processing since in most available distributed RDF streaming systems, *e.g.*, CQELSCloud [17] and Katts [9], the logical query plan is determined at compile time. Such a behavior can hardly promise long-term efficiency and reliability, since there is no single query plan that is always optimal for a given query.

A general approach for large scale data stream processing is performed over a distributed setting. Such systems are better designed and operated upon when implemented on top of robust, state-of-the-art engines, *e.g.*, Kafka [10] and Spark [26,27]. Moreover, the system has to adapt to unpredictable input data streams and to dynamically updated execution plans while ensuring optimal performance. A time-driven/batch-driven [5] approach could be a solution for adaptive streaming query. In that context, it becomes possible to reconstruct the logical plan for each query execution. Furthermore, compared to data-driven systems [5], time-driven/batch-driven provides a more coarse operation granularity. Although this mechanism inevitably causes higher query latency, it also brings high system throughput, inexpensive cost and low latency to achieve fault tolerance and system adaptivity [27].

Our system, Strider, possesses the aforementioned characteristics. In this paper, we present three main contributions concerning this system: (1) the design and implementation of a production-ready RSP engine for large scale RDF data streams processing which is based on the state-of-the-art distributed computing frameworks (*i.e.*, Spark and Kafka). (2) Strider integrates two forms of adaptation. In the first one, for each execution of a continuous query, the system decides, based on incoming stream volumes, to use either a query compile-time (rule-based) or query run-time (cost-based) optimization approach. The second one concerns the run-time approach and decides when the query plan is optimized (either at the previous query window or at the current one). (3) an evaluation of Strider over real-world and synthetic data sets.

2 Background Knowledge

Strider follows a classical streaming system approach with a messaging component for data flow management and a computing core for real-time data analytics. In this section, we present and motivate the use of Spark Streaming and Kafka as these two components. Then, we consider streaming models and adaptive query processing.

Kafka & Spark Streaming. Kafka 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.

Spark is a MapReduce-like cluster-computing framework that proposes a parallelized fault tolerant collection of elements called Resilient Distributed Dataset (RDD) [26]. 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) [27] 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. To achieve continuous SPARQL query processing on Spark Streaming, we bind the SPARQL operators to the corresponding Spark SQL relational operators. Moreover, the data processing is based on DataFrame (DF), an API abstraction derived from RDD.

Streaming Models. At the physical level, a computation model for stream processing has two principle classes: Bulk Synchronous Parallel (BSP) and Record-at-a-time [25]. From a logical level perspective, a streaming model uses the concept of a *Tick* to drive the system in taking actions over input streams. [5] defines a Tick in three ways: data-driven (DD), time-driven (TD) and batch-driven (BD). In general, the physical BSP is associated to the TD and/or BD models, *e.g.*, Spark Streaming [27] and Google DataFlow with FlumeJava [1] adopt this approach by creating a micro-batch of a certain duration T. That is data are cumulated and processed through the entire DAG within each batch. The record-at-a-time model is usually associated to the logical DD model (although TD and BD are possible) and prominent examples are Flink [6] and Storm [23]. The record-at-a-time/DD model provides lower latency than BSP/TD/BD model for typical computation. On the other hand, the record-at-a-time model requires state maintenance for all operators with record-level granularity. This behavior obstructs system throughput and brings much higher latencies when recovering after a system failure [25]. For complex tasks involving lots of aggregations and iterations, the record-at-a-time model could be less efficient, since it introduces an overhead for the launch of frequent tasks. Given these properties and the fact that in [7], the authors emphasize that latencies in the order of few seconds is enough for most extreme use cases at Facebook, we have decided to use Spark Streaming.

Adaptive Query Processing (AQP) is recognized as a complex task, especially in the streaming context [8]. Moreover, AQP for continuous SPARQL query needs to cope with some cross-field challenges such as SPARQL query optimization, stream processing, *etc*.. Due to structure unpredictability, schema-free and real-time features of RDF data streams, conventional optimizations for static RDF data processing through data pre-processing, *e.g.*, triple indexing and statistic summarizing, become impractical. However, the perspectives from [16,21] show that most parts of RDF graphs have tabular structure, especially in the IoT domain. This opens up several perspectives concerning selectivity/cardinality estimation and the possibility to use Dynamic Programming (DP) approaches. Inspired by [22,14,11,24,25], we propose a novel AQP optimizer for RDF stream processing.

3 Strider overview

In this section, we first present a Strider query example, then we provide a system's overview, detail the data flow and query optimization components.

3.1 Continuous query example

Listing 1.1 introduces a running scenario that we will use throughout this paper. The example corresponds to a use case encountered in the Waves project, *i.e.*, query Q_8 continuously processes the messages of various types of sensor observations.

We introduce new lexical rules for continuous SPARQL queries which are tailored to a micro-batch approach.The `STREAMING` keyword initializes the application context of Spark Streaming and the windowing operator. More precisely, `WINDOW` and `SLIDE` respectively indicate the size and sliding parameter of a time-based window. The novelty comes from the `BATCH` clause which specifies the micro-batch interval of discretized stream for Spark Streaming. Here, a sliding window consists of one or multiple micro-batches.

```
STREAMING { WINDOW [10 Seconds] SLIDE [10 Seconds] BATCH [5 Seconds] }
REGISTER { QUERYID [Q8] 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 ?o2 ?o3
  WHERE {   ?s ssn:hasValue ?o1 (tp1); ssn:hasValue ?o2 (tp2);
               ssn:hasValue ?o3 (tp3).
            ?o1 rdf:type cuahsi:flow (tp4).
            ?o2 rdf:type cuahsi:temperature (tp5).
            ?o3 rdf:type cuahsi:chlorine    (tp6). }] }
```

Listing 1.1: Strider's query example (Q_8)

The `REGISTER` clause is used to register standard SPARQL queries. Each query is identified by an identifier. The system allows to register several queries simultaneously in a thread pool. By sharing the same application context and cluster resources, Strider launches all registered continuous SPARQL queries asynchronously by different threads.

3.2 Architecture

Strider contains two principle modules: (1) data flow management. In order to ensure high throughput, fault-tolerance, 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 represent different types of RDF events. (2) 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.

Figure 1 gives a high-level overview of the system's architecture. The upper part of the figure provides details on the application's data flow management. In a nutshell, data sources (IoT sensors) are sending messages to a publish-subscribe layer. This layer emits messages for the streaming layer which executes registered queries. The sensor's metadata are converted into RDF events for data integration purposes. We use Kafka to design the system's data flow management. Kafka is connected to Spark Streaming using a *Direct Approach*[2] to guarantee exactly-once semantics and parallel data feeding. The input RDF event streams are then continuously transformed to DataFrames.

[2] https://spark.apache.org/docs/latest/streaming-kafka-integration.html

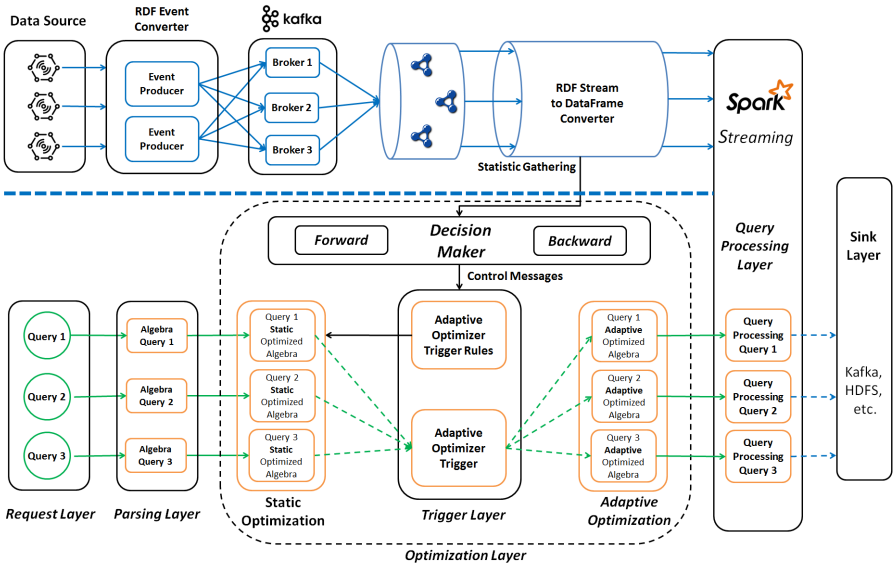


Fig. 1: Strider Architecture

The lower part of Figure 1 presents components related to the implementation of the computing core. The Request layer registers continuous queries. 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. These queries are later sent to the Parsing layer to compute a first version of a query plan. These new plans are pushed to the Optimization layer which consists of four collaborating sub-components: static and adaptive optimizations as well as a trigger mechanism and a Decision Maker for adaptation strategy. Finally, the Query Processing layer sets the query execution off right after the optimized logical plan takes place.

4 Strider's continuous SPARQL processing

In this section, we detail the components of the Strider's optimizer layer. Two optimization components are proposed, *i.e.*, static and adaptive, which are respectively based on heuristic rules and (stream-based) statistics. The trigger layer decides whether the query processing adopts a static or an adaptive approach. Two strategies are proposed for AQP: backward (B-AQP) and forward (F-AQP). They mainly differ on when, *i.e.*, at the previous or current window, the query plan is computed.

4.1 Query processing outline & trigger layer

Intuitively, Strider's optimizers search for the optimal join ordering of triple patterns based on collected statistics. Both static (query compile-time) and adaptive (query run-time) optimizations are processed using a graph $G^U = (V, E)$, denoted Undirected Connected Graph (UCG) [22] where vertices represent triple patterns and edges symbolize joins between triple patterns. Naturally, for a given query q and its query graph

$G^Q(q)$, $G^U(q) \subseteq G^Q(q)$. A UCG showcases the structure of a BGP and the join possibilities among its triple patterns. That query representation is considered to be more expressive [14] than the classical RDF query graph. The weight of UCG's vertices and edges correspond to the selectivity of triple patterns and joins, respectively. Once the weights of an UCG are initialized, the query planner automatically generates an optimal logical plan and triggers a query execution. For the sake of a better explanation, the windowing operator involved in this section is considered as a tumbling window.

Strider's static optimization retains the philosophy of [24]. Basically, static optimization implies a heuristics-based query optimization. It ignores data statistics and leads to a static query planner. In this case, unpredictable changes in data stream structures may incur a bad query plan. The static optimization layer aims at giving a basic performance guarantee. The predefined heuristic rules set empirically assign the weights for UCG vertices 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 represents the query execution pipeline which is permanently kept by the system. More details about UCG creation and query logical plan generation are given in Sec. 4.2.

The Trigger layer supports the transition between the stages of static optimization and adaptive optimization. In a nutshell, that layer is dedicated to notify the system whether it is necessary to proceed with an adaptive optimization. Our adaptation strategy requires collecting statistical information and generating an execution logical plan. The overhead coming with such actions is not negligible in a distributed environment. 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.

4.2 Run-time query plan generation

Here, we first briefly introduce how we collect stream statistics and construct query plan. Then, we give an insight into the AQP optimization, which is essentially a cardinality-based optimization.

Unlike systems based on greedy and left-deep tree generation, *e.g.*, [22,13], Strider makes a full usage of CPU computing resources and benefits from parallel hardware settings. It thus creates query logical plans in the form of general (bushy) directed trees. Hence, the nodes with the same height in a query plan p_n can be asynchronously computed in a non-blocking way (in the case where computing resources are allowed). Coming back to our Listing 1.1 example, Figure 2 refines the procedure of query processing (F-AQP) at w_n, $n \in N$. If w_n contains multiple RDDs (micro-batches), the system performs the union all RDDs and generates a new combined RDD. Note that the union operator has a very low-cost in Spark. Afterward, the impending query plan optimization follows three steps: (a) UCG (weight) initialization; (b) UCG path cover finding; (c) query plan generation.

UCG weight initialization is briefly described in Algorithm 1 and Figure 3 (step (a), step (b)). Since triple patterns are located at the bottom of a query tree, the query evaluation is performed in a bottom-up fashion and starts with the selection of triple

patterns $\sigma(tp_i)$, $1 \leq i \leq I$ (with I the number of triple patterns in the query's BGP). The system computes $\sigma(tp_i)$ asynchronously for each i and temporally caches the corresponding results ($R^{\sigma}(tp_i)$) in memory. $Card(tp_i)$, *i.e.*, the cardinality of $R^{\sigma}(tp_i)$, is computed by a Spark count action. Thence, we can directly assign the weight of vertices in $G^U(Q)$. Note that the estimation of $Card(tp_i)$ is exact.

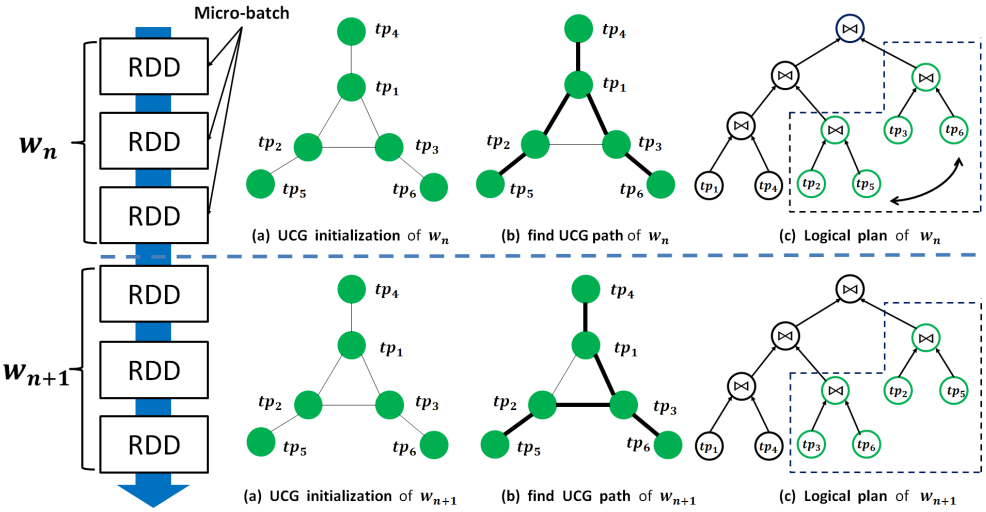


Fig. 2: Dynamic Query Plan Generation for Q_8

Once all vertices are set up, the system predicts the weight of edges (*i.e.*, joined patterns) in $G^U(q)$. We categorize two types of joins (edges): (i) star join, includes two sub-types, *i.e.*, star join without bounded object and star join with bounded object; (ii) non-star join. To estimate the cardinality of join patterns, we make a trade-off between accuracy and complexity. The main idea is inspired by a research conducted in [22,14,11]. However, we infer the weight of an edge from its connected vertices, *i.e.*, no data pre-processing is required. The algorithm begins by iteratively traversing $G^U(q)$ and identifies each vertex $v \in V$ and each edge $e \in E$. Then we can decompose $G^U(q)$ into the disjoint star-shaped joins and their interconnected chains (Figure 3, step (b)). The weight of an edge in a star join shape is estimated by the function getStarJoinWeight. The function first estimates the upper bound of each star join output cardinality (*e.g.*, , $Card(tp_1 \bowtie tp_2 \bowtie tp_3)$), then assigns the weight edge by edge. Every time the weight of the current edge e is assigned, we mark e as visited. This process repeats until no more star join can be found. Then, the weight of unvisited non-star join shapes is estimated by the function getNonStarJoinWeight. It lookups the two vertices of the current edge, and chooses the one with smaller weight to estimate the edge cardinality. The previous processes are repeated until all the edges have been visited in $G^U(q)$.

UCG path cover finding & Query plan generation. Figure 3 step (c) introduces path cover finding and query plan generation. The system starts by finding the path cover in $G^U(q)$ right after $G^U(q)$ is prepared. Intuitively, we search the undirected path cover which links all the vertices of $G^U(q)$ with a minimum total edge weight.

The path searching is achieved by applying Floyd–Warshall algorithm iteratively. The extracted path $Card(G^U(q)) \subseteq G^U(q)$, is regarded as the candidate for the logical plan generation. Finally, we construct p_n, the logical plan of $G^U(q)$ at w_n, in a top-down manner (Figure 3, step (c)). Note that path finding and plan generation are both computed on the driver node and are not expensive operations (around 2 - 4 milliseconds in our case).

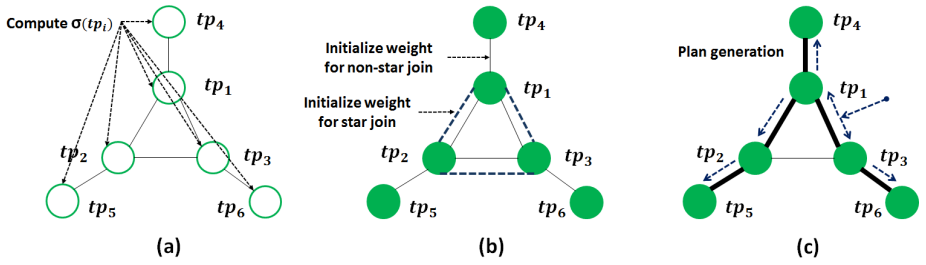


Fig. 3: Initialized UCG weight, find path cover and generate query plan

Algorithm 1: UCG weight initialization

Input: query q, $G^U(q) = (V, E) \subseteq G^Q(q)$, current buffered window w_n
Output: $G^U(q)$ with weight-assigned

while $\exists v$ *unvisited* $\in V$ **do**
  mark v as visited, $R^\sigma(v) \leftarrow$ `compute` (v) ;
  `buffer` $(v, R^\sigma(v)) \wedge v$.weight $\leftarrow Card(v)$;
while $\exists e$ *unvisited* $\in E$ **do**
  mark e as visited ;
  if *(*$\exists$ *star join* S_J*)* $\wedge e \cap S_J \neq \emptyset$ **then**
    locate each $S_J \in G^U(q)$
    foreach $\forall e_S \in S_J$ **do**
      mark e_S as visited ;
      e_S.weight $\leftarrow$ `getStarJoinWeight`$(S_J, e_S.vertices)$;
  else e.weight $\leftarrow$ `getNonStarJoinWeight`(S_J);

4.3 B-AQP & F-AQP

We propose a dual AQP strategy, namely, **backward** (B-AQP) and **forward** (F-AQP). B/F-AQP depict two philosophies for AQP, Figure 4 roughly illustrates how B/F-AQP switching is decided at run-time, *i.e.*, this is the responsibility of the Decision Maker component. Generally, B-AQP and F-AQP are using similar techniques for query plan generation. Compared to F-AQP, B-AQP delays the process for query plan generation.

Our B-AQP strategy is inspired by [25]'s pre-scheduling. Backward implies gathering, feeding back the statistics to the optimizer on the current window, then the optimizer constructs the query plan for the next window. That is the system computes the query plan p_{n+1} of a window w_{n+1} through the statistics of a previous window

w_n. Strider possesses a time-driven execution mechanism, the query execution is triggered periodically with a fixed update frequency s (*i.e.*, sliding window size). Between two consecutive windows w_n and w_{n+1}, there is a computing barrier to reconstruct the query plan for w_{n+1} based on the collected statistics from a previous window w_n. Suppose the query execution of w_n consumes a time t_n (*e.g.*, in seconds), then for all $t_n < s$, the idle duration $\delta_n = s - t_n$ allows to re-optimize the query plan. But δ_n should be larger than a configurable threshold Θ. For $\delta_n < \Theta$, the system may not have enough time to (i) collect the statistic information of w_n and (ii) to construct a query plan for w_{n+1}. This potentially expresses a change of incoming steams and a degradation of query execution performance. Hence, the system decides to switch to the F-AQP approach.

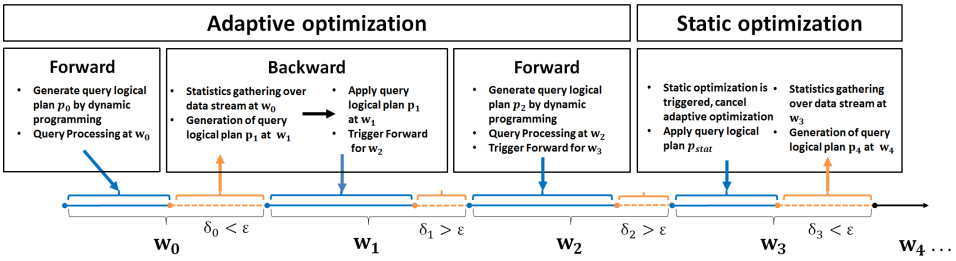


Fig. 4: Decision Maker of Adaptation Strategy

F-AQP applies a DP strategy to find the optimal logical query plan for the current window w_n. The main purpose of F-AQP is to adjust the system state as soon as possible. The engine executes a query, collects statistics and computes the logical query plan simultaneously. Here, the statistics are obtained by counting intermediate query results, which causes data shuffling and DAG interruption, *i.e.*, the system has to temporally cut the query execution pipeline. In Spark, such suspending operation is called an *action*, which immediately triggers a job submission in Spark application. However, a frequent job submission may bring some side effects. The rationale is, for a master-slave based distributed computing framework (*e.g.*, Spark, Storm) uses a master node (*i.e.*, driver) to schedule jobs. The driver locally computes and optimizes each submitted DAG and returns the control messages to each worker node for parallel processing. Although the "count" action itself is not expensive, the induced side effects (*e.g.*, driver job-scheduling/submission, communication of control message between driver and workers) will potentially impact the system's stability. For instance, based on our experience, F-AQP's frequent job submission and intermediate data persistence/unpersistence put a great pressure on the JVM's Garbage Collector (GC), *e.g.*, untypical GC pauses are observed from time to time in our experiment.

Decision Maker. Through experimentations of different Strider configurations, we understood the complementarity of both the B-AQP and F-AQP approaches. Real performance gains can be obtained by switching from one approach to another. This is mainly due to their properties which are summarized in Table 1.

We designed a decision maker to automatically select the most adapted strategy for each query execution. The decision maker takes into account two parameters: a configurable switching threshold $\Theta \in]0, 1[$; $\gamma_n = \frac{t_n}{s}$, the fraction of query execution

Strategy	Advantage	Drawback
B-AQP	No dynamic programming overhead	Approximate query plan generation through previously-collected statistics
F-AQP	Query plan generation through real-time collected statistics	Overhead for dynamic programming, side-effects caused by pipeline interruption

Table 1: B/F-AQP summarization

time t over windowing update frequency s. For the query execution at w_n, if $\gamma_n < \Theta$, the system updates the query plan from p_n to p_{n+1} for the next execution. Otherwise, the system recomputes p_{n+1} by DP at w_{n+1} (see Algorithm 2). We empirically set $\Theta = 0.7$ by default.

Algorithm 2: B-AQP and F-AQP Switching in Decision Maker

Input: query q, switching threshold Θ, sliding window $W = \{w_n\}_{n \in N}$, update frequency s of W

```
foreach w_n ∈ W do
    t_n ← getRuntime { execute (q) } // executionTime ;
    λ_n ← getAdaptiveStrategy (Θ,t_n,s) // adaptiveStrategy;
    if λ_n == Backward then
        update query plan p_n of q at w_n
        p_{n+1} ← update (p_n);
    if λ_n == Forward then Recompute p_{n+1} at w_{n+1};
```

The decision maker plays a key role for maintaining the stability of the system's performance. Our experiment (Sec. 5.3) shows that, the combination of F/B-AQP through decision maker is able to prevent the sudden performance declining during a long running time.

5 Evaluation

5.1 Implementation details

Strider is written in Scala, the code source can be found here[3]. 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 the 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. By traversing the AST, we bind the SPARQL operators to the corresponding Spark SQL relational operators for query evaluation.

5.2 Experimental Setup

We test and deploy our engine on Amazon EC2/EMR cluster of 9 computing nodes and Yarn resource management. The system holds 3 nodes of m4.xlarge for data flow management (*i.e.*, Kafka broker and Zookeeper [12]). Each node has 4 CPU virtual cores of 2.4 GHz Intel Xeon E5-2676, 16 GB RAM and 750 MB/s bandwidth. We use

[3] https://github.com/renxiangnan/strider

Apache Spark 2.0.2, Scala 2.11.7 and Java 8 as baselines for our evaluation. The Spark (Streaming) cluster is configured with 6 nodes (1 master, 5 workers) of type c4.xlarge. Each one has 4 CPU virtual cores of 2.9 GHz Intel Xeon E5-2666, 7.5 GB RAM and 750 MB/s. The experiments of Strider on local mode, C-SPARQL and CQELS are all performed on a single instance of type c4.xlarge.

Datasets & Queries. We evaluated our system using two datasets that are built around real world streaming use cases: *SRBench* [28] and *Waves*. SRBench, one of the first available RSP benchmarks, comes with 17 queries on LinkedSensorData. The datasets consists of weather observations about hurricanes and blizzards in the United States (from 2001 to 2009). Another dataset considered in our evaluation comes from aforementioned project Waves. The dataset describes different water measurements captured by sensors. Values of flow, water pressure and chlorine levels are examples of these measurements. The value annotation uses three popular ontologies: SSN, CUAHSI-HIS and QUDT. Each sensor observes and records at least one physical phenomenon or a chemical property, and thus generates RDF data stream through Kafka producer. Our micro-benchmark contains 9 queries, denoted from Q_1 to Q_9[4]. The road map of our evaluation is designed as follow: (1) injection of structurally stable stream for experiment of Q_1 to Q_6. Q_1 to Q_3 are tested by SRBench datasets. Here, a comparison between Strider and the state of the art RSP systems *e.g.*, C-SPARQL and CQELS are also provided. Then we perform Q_4 to Q_6 based on Waves dataset. (2) Injection of structurally unstable stream. We generate RDF streams by varying the proportion of different types of Kafka messages (*i.e.*, sensor observations). For this part of the evaluation, queries Q_7 to Q_9 are considered.

Performance criteria. In accordance with *Benchmarking Streaming Computation Engines at Yahoo!* [5], we choose the system throughput and query latency as two primary performance metrics. Throughput indicates how many data can be processed in a unit of time. Throughput is denoted as "triples per second" in our case. Latency means how long does the RSP engine consumes between the arrival of an input and the generation of its output. The reason why we abandoned existing RSP performance benchmarking systems [18,2] is that, none of them is tailored for massive data stream. This limitation is contrary to our original intention of using distributed stream processing framework to cope with massive RDF stream. We did not record the latency of C-SPARQL, CQELS and Strider in local mode for two reasons: (1) given the scalability limitation of C-SPARQL, we have to control input stream rate within a low level to ensure the engine can run normally [18]. (2) due to its design, based on a so-called eager execution mechanism and *DStream* R2S operator, the measure of latencies in CQELS is unfeasible [18]. Moreover, given reasons provided in Sec. 4.3, we have not done any comparisons of B/F-AQP versus F-AQP approaches.

Performance tuning on Spark is quite difficult. Inappropriate cluster configuration may seriously hinder engine performance. So far we can only empirically configure Spark cluster and tune the cluster settings step by step. We briefly list some important performance settings based on our experience. First of all, we apply some basic

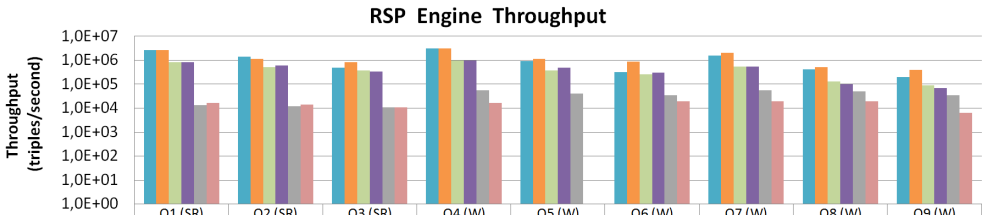
[4] Check the wiki of our github page for more details of the queries and datasets

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

optimization techniques. *e.g.*, using Kryo serializer to reduce the time for task/data serialization. Besides, we generally considered adjustments of Spark configuration along three control factors to achieve better performance. The first factor is the size of micro-batch intervals. Smaller batch sizes can better meet real-time requirements. However, it also brings frequent job submissions and job scheduling. The performance of a BSP system like Spark is sensitive to the chosen size of batch intervals. The second factor is GC tuning. Set appropriately, the GC strategy (*e.g.*, using Concurrent Mark-Sweep) and storage/shuffle fraction may efficiently reduce GC pressure. The third factor is the parallelism level. This includes the partition number of Kafka messages, the partition number of RDD for shuffling, and the upper/lower bound for concurrent job submissions, *etc.*.

5.3 Evaluation Results & Discussions

Figures 5 and 6 respectively summarize the RSP engines throughput and latency. Note that CQELS gives a parsing error for Q_5. This is due, at least for the version that we have evaluated, to the lack of support for the UNION operator in the source code. In view of the centralized designs of C-SPARQL and CQELS, a direct performance comparison to Strider with distributed hardware settings seems unfair. So we also evaluated Strider in local mode, *i.e.*, running the system on a single machine (although it should not be its forte, Strider still gets an advantage from the multi-core processor). Based on this preliminary evaluation, we try to give an intuitive impression and reveal our findings about these three RSP systems.

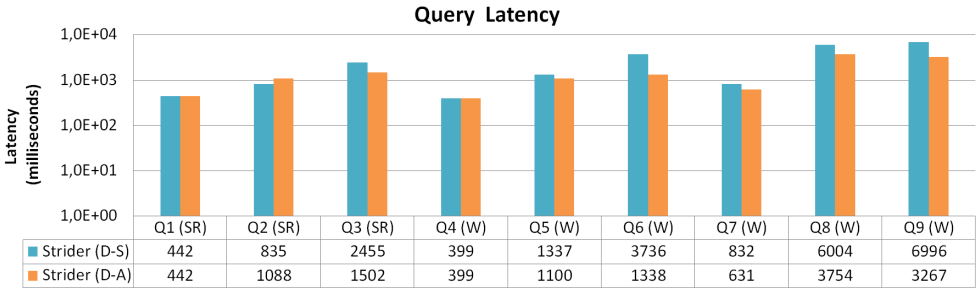


	Q1 (SR)	Q2 (SR)	Q3 (SR)	Q4 (W)	Q5 (W)	Q6 (W)	Q7 (W)	Q8 (W)	Q9 (W)
Strider (D-S)	2620005	1431808	495380	3116147	939506	325328	1555542	422591	194225
Strider (D-A)	2620005	1164471	838531	3116147	1148896	886769	2005617	524181	387742
Strider (L-S)	837860	510231	382972	965986	370904	264665	538524	127276	91710
Strider (L-A)	837860	608228	329683	965986	481945	310039	538524	98573	68345
C-SPARQL	13142	12008	10934	56179	40469	34882	54917	49985	33988
CQELS	16817	14211	10637	16541	0	19574	19534	19472	6527

Fig. 5: RSP engine throughput (triples/second). **D/L-S:** Distributed/Local mode Static Optimization. **D/L-A:** Distributed/Local mode Adaptive Optimization. **SR:** Queries for SRBench dataset. **W:** Queries for Waves dataset.

In Figure 5, we observe that Strider generally achieves million/sub-million-level throughput under our test suite. Note that both Q_1 and Q_4 have only one join, *i.e.*, optimization is not needed. Most tested queries scale well in Strider. Adaptive optimization generates query plans based on the workload statistics. In total, it provides a more efficient query plan than static optimization. But the gain of AQP for the simple queries that have less join tasks (*e.g.*, Q_1, Q_5) becomes insubstantial. We also found out that, even if Strider runs on a single machine, it still provides up to 60x gain on throughput compared to C-SPARQL and CQELS. Figure 6 shows Strider attains a second/sub-second

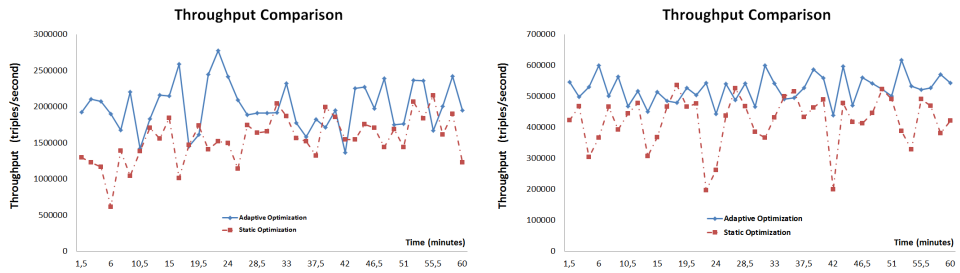
delay. Obviously, for queries with 2 triple patterns in the query's BGP, we can observe the same latency between static and adaptive optimizations, Q_1 and Q_4. Query Q_2 is the only query where the latency of the adaptive approach is higher than the static one. This is due to the very simple structure of the BGP (2 joins in the BGP). In this situation, the overhead of DP covers the gain from AQP. For all other queries, the static latency is higher than the adaptive one. This is justified by more complex BGP structures (more than 5 triple patterns per BGP) or some union of BGPs.



	Q1 (SR)	Q2 (SR)	Q3 (SR)	Q4 (W)	Q5 (W)	Q6 (W)	Q7 (W)	Q8 (W)	Q9 (W)
Strider (D-S)	442	835	2455	399	1337	3736	832	6004	6996
Strider (D-A)	442	1088	1502	399	1100	1338	631	3754	3267

Fig. 6: Query latency (milliseconds) for Strider (in distributed mode)

On the contrary, the average throughput of C-SPARQL and CQELS is maintained in the range of 6.000 and 50.000 triples/second. The centralized designs of C-SPARQL and CQELS limit the scalability of the systems. Beyond the implementation of query processing, the reliability of data flow management on C-SPARQL and CQELS could also cause negative impact on system robustness. Due to the lack of some important features for streaming system (*e.g.*, back pressure, checkpoint and failure recovery) once input stream rate reaches to certain scale, C-SPARQL and CQELS start behaving abnormally, *e.g.*, data loss, exponential increasing latency or query process interruption [18,19]. Moreover, we have also observed that CQELS' performance is insensitive to the changing of computing resources. We tested CQELS on different EC2 instance types, *i.e.*, with 2, 4 and 8 cores, and the results evaluation variations were negligible.



(a) (b)

Fig. 7: Record of throughput on Strider. (a)-throughput for q_7; (b)-throughput for q_8

Figure 7 and Figure 8 concern the monitoring of Strider's throughput for Q_7 to Q_9. We recorded the changes of throughput over a continuous period of time (one hour).

The source stream produces the messages with different types of sensor observations. The stream is generated by mixing temperature, flow and chlorine-level measurement with random proportions. The red and blue curves denote query with respectively static and adaptive logical plan optimization. For Q_7 and Q_8 (Figure 7), except when some serious throughput drops have been observed in 7b, static and adaptive planners return a close throughput trend. For a more complex query Q_9 (Figure 8), which contains 9 triple patterns and 8 join operators. Altering logical plans on Q_9 causes significant impact on engine performance. Consequently, our adaptive strategy is capable to handle the structurally unstable RDF stream. Thus the engine can avoid a sharp performance degradation.

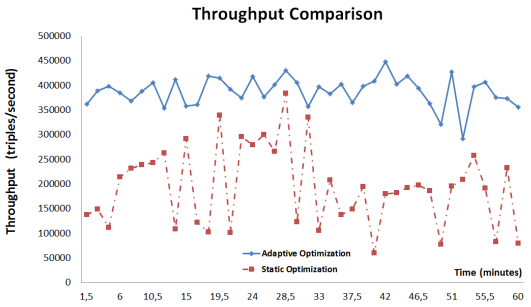


Fig. 8: Throughput for q_9 on Strider

Through this experiment, we identified some shortcomings in Strider that will be addressed in future work: (1) the data preparation on Spark Streaming is relatively expensive. It costs around 0.8 to 1 second to initialize before triggering the query execution in our experiment. (2) Strider has a more substantial throughput decreasing with an increasing number of join tasks. In order to alleviate this effect, the possible solution is enlarging the cluster scale or choosing a more powerful driver node. (3) Strider does not support well high concurrent requests, although this is not at the moment one of our system design goals. *E.g.*, some use cases demand to process a big amount of concurrent queries. Even through Strider allows to perform multiple queries asynchronously, it could be less efficient.

6 Related Work

In the recent years, a variety of RSP systems have been proposed which can be divided into two categories: centralized and distributed.

Centralized RSP engines. For the last few years, some contributions have been done to satisfy the basic needs of RDF stream processing. RSP engines like C-SPARQL, CQELS, ETALIS, *etc*., are developed to run on a single machine. None of them targets the scenario that involves massive incoming data stream.

Distributed RSP engines. CQELS-Cloud [17] is the first RSP system which mainly focuses on the engine elasticity and scalability. The whole system is based on Apache Storm. Firstly, CQELS-Cloud compresses the incoming RDF streams by dictionary encoding in order to reduce the data size and the communication in the computing cluster. The query logical plan is mapped to a Storm topology, and the evaluation is done through a series of SPARQL operators located on the vertex of the topology. Then, to overcome the performance bottlenecks on join tasks, the authors propose a *parallel multiway join* based on probing sequence. From the aspect of implementation, CQELS-

Cloud is designed as the streaming service for high concurrent requests. The capability of CQELS-Cloud to cope with massive incoming RDF data streams is still missing. Furthermore, to the best of our knowledge, CQELS-Cloud is not open source, customized queries and data feeding are not feasible. Katts is another RSP engine based on Storm. The implementation of Katts [9] is relatively primitive, it is more or less a platform for algorithm testing but not an RSP engine. The main goal of Katts is designed to verify the efficiency of graph partitioning algorithm for cluster communication reduction.

Although the SPARQL query optimization techniques have been well developed recently, CQELS is still the only system which considers query optimization to process RDF data stream. However, the greedy-like left-deep plan leads to sequential query evaluation, which makes CQELS benefit from few additional computing resources. The conventional SPARQL optimization for static data processing can be hardly applied in a streaming context. Recent efforts [22,15,14,20] possess long data preprocessing stage before launching the query execution. The proposed solutions do not meet real-time or near real-time use cases. The heuristic-based query optimization in [24] totally ignores data statistics and thus does not promise the optimal execution plan for 24×7 running streaming service.

7 Conclusion and Future Work

In this paper, we present Strider, a distributed RDF batch stream processing engine for large scale data stream. It is built on top of Spark Streaming and Kafka to support continuous SPARQL query evaluation and thus possesses the characteristics of a production-ready RSP. Strider comes with a set of hybrid AQP strategies: *i.e*., static heuristic rule-based optimization, forward and backward adaptive query processing. We insert the trigger into the optimizer to attain the automatic strategy switching at query runtime. Moreover, with its micro-batch approach, Strider fills a gap in the current state of RSP ecosystem which solely focuses on record-at-a-time. Through our micro-benchmark based on real-word datasets, Strider provides a million/sub-million-level throughput and second/sub-second latency, a major breakthrough in distributed RSPs. And we also demonstrate the system reliability which is capable to handle the structurally instable RDF streams.

There is still room for improving the system's implementation. As future work, we aim to add stream reasoning capacities and the ability of combining static data.

References

1. T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *PVLDB*, 2015.
2. M. I. Ali, F. Gao, and A. Mileo. Citybench: A configurable benchmark to evaluate rsp engines using smart city datasets. In *ISWC*, 2015.
3. D. Anicic, S. Rudolph, P. Fodor, and N. Stojanovic. Stream reasoning and complex event processing in etalis. *Semant. web*, 2012.

4. D. F. Barbieri and al. C-SPARQL: SPARQL for continuous querying. In *WWW*, 2009.
5. I. Botan, R. Derakhshan, N. Dindar, L. Haas, R. J. Miller, and N. Tatbul. Secret: A model for analysis of the execution semantics of stream processing systems. *PVLDB*, 2010.
6. P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 2015.
7. G. J. Chen, J. L. Wiener, S. Iyer, A. Jaiswal, R. Lei, N. Simha, W. Wang, K. Wilfong, T. Williamson, and S. Yilmaz. Realtime data processing at facebook. In *SIGMOD*, 2016.
8. A. Deshpande, Z. G. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 2007.
9. L. Fischer and al. Scalable linked data stream processing via network-aware workload scheduling. In *SSWS@ISWC*, 2013.
10. 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.*, 2012.
11. A. Gubichev and T. Neumann. Exploiting the query structure for efficient join ordering in sparql queries. In *EDBT*, 2014.
12. P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX*, 2010.
13. 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.
14. T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for rdf queries with multiple joins. In *ICDE*, 2011.
15. T. Neumann and G. Weikum. Scalable join processing on very large rdf graphs. In *SIGMOD*, 2009.
16. M. Pham and P. A. Boncz. Exploiting emergent schemas to make RDF systems more efficient. In *ISWC*, 2016.
17. D. L. Phuoc and al. Elastic and scalable processing of linked stream data in the cloud. In *ISWC*, 2013.
18. 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*, 2012.
19. X. Ren, H. Khrouf, Z. Kazi-Aoul, Y. Chabchoub, and O. Curé. On measuring performances of C-SPARQL and CQELS. In *SWIT@ISWC*, 2016.
20. A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic, and G. Lausen. S2rdf: Rdf querying with sparql on spark. *PVLDB.*, 2016.
21. E. Siow, T. Tiropanis, and W. Hall. Sparql-to-sql on internet of things databases and streams. In *ISWC*, 2016.
22. M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. Sparql basic graph pattern optimization using selectivity estimation. In *WWW*, 2008.
23. A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@twitter. SIGMOD, 2014.
24. P. Tsialiamanis, L. Sidirourgos, I. Fundulaki, V. Christophides, and P. Boncz. Heuristics-based query optimisation for sparql. In *EDBT*, 2012.
25. S. Venkataraman, A. Panda, K. Ousterhout, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Spark Summit*, 2016.
26. 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*, 2012.
27. M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, 2013.
28. Y. Zhang, P. M. Duc, O. Corcho, and J.-P. Calbimonte. Srbench: A streaming rdf/sparql benchmark. In *ISWC*, 2012.