Elastic and scalable processing of Linked Stream Data in the Cloud*

Danh Le-Phuoc, Hoan Nguyen Mau Quoc, Chan Le Van, and Manfred Hauswirth

Digital Enterprise Research Institute, National University of Ireland, Galway

Abstract. Linked Stream Data extends the Linked Data paradigm to dynamic data sources. It enables the integration and joint processing of heterogeneous stream data with quasi-static data from the Linked Data Cloud in near-real-time. Several Linked Stream Data processing engines exist but their scalability still needs to be in improved in terms of (static and dynamic) data sizes, number of concurrent queries, stream update frequencies, etc. So far, none of them supports parallel processing in the Cloud, i.e., elastic load profiles in a hosted environment. To remedy these limitations, this paper presents an approach for elastically parallelizing the continuous execution of queries over Linked Stream Data. For this, we have developed novel, highly efficient, and scalable parallel algorithms for continuous query operators. Our approach and algorithms are implemented in our CQELS Cloud system and we present extensive evaluations of their superior performance on Amazon EC2 demonstrating their high scalability and excellent elasticity in a real deployment.

Keywords: Cloud, Linked Data, linked stream processing, continuous queries

1 Introduction

Realistically, all data sources on the Web are dynamic (across a spectrum). Many current sources are of a slow update nature which is well supported by the existing batch-update infrastructure of Linked Data, e.g., geo-data or DBpedia. However, a fast increasing number of sources produce streams of information for which the processing has to be performed as soon as new data items become available. Examples of such data sources include sensors, embedded systems, mobile devices, Twitter, and social networks, with a steep, exponential growth predicted in the number of sources and the amount of data [22]. Integrating these information streams with other sources will enable a vast range of new "near-real-time" applications. However, due to the heterogeneous nature of the streams and static sources, integrating and processing this data is a difficult and labor-intensive task which gave rise to Linked Stream processing, i.e., extending the notion of Linked Data to dynamic data sources. Linked Stream Data processing engines, such as C-SPARQL [3], EP-SPARQL [1], SPARQL$_{stream}$ [5], and CQELS [19], have emerged as an effort to facilitate this seamless integration of heterogeneous stream data with the Linked Data Cloud using the same abstractions.

* This research has been supported by the European Commission under Grant No. FP7-287305 (OpenIoT) and Grant No. FP7-287661 (GAMBAS) and by Science Foundation Ireland under Grant No. SFI/08/CE/I1380 (Lion-II) and Grant No. SFI/12/RC/2289 (INSIGHT).

None of the above systems could yet systematically and satisfactorily address all scalability aspects existing in Linked Stream Data processing [20], such as the wide range of stream data production frequencies, the size of static data, the number of concurrent queries, elastic load profile support, etc. The existing approaches start to fail when some of these scalability aspects go beyond certain thresholds. For instance, C-SPARQL and EP-SPARQL only can deal with small RDF datasets (∼1 million triples) and most of the approaches are only able consume very slow input stream rates of around 100 triples/second when the number of concurrent queries grows up to 100–1000 [20]. These thresholds are rather modest for real-time and Web applications and must be improved to make the systems usable for practical applications in practical settings, e.g., each car in a city producing a data stream.

On top of this, further scalability issues come from practical limits of computer hardware such as memory size, network bandwidth, processor speed, etc. Even though these parameters will increase over time (Moore's Law), the general argument is likely to remain true for the foreseeable future [17]. However, today's typical computer hardware is cheap and almost indefinitely replicable [17]. The total-cost-of-ownership of 8 off-the-shelf commodity servers with 8 processing cores and 128GB of RAM each is much lower than that of a single system with 64 processors and 1TB of RAM. Therefore, distributing the processing load of a Linked Stream Data processing engine over networked computers is a promising strategy to achieve scalability.

Additionally, the trend to Cloud infrastructures, i.e., renting servers on a "pay-per-use" basis, provides a further argument in favor of this strategy. Amazon EC2, Google Cloud, and Microsoft Azure are prominent examples for this development. Building a Linked Stream Data processing engine running on such an elastic cluster potentially enables the engine to adapt to changing processing loads by dynamically adjusting the number of processing nodes in the cluster at runtime. This "elasticity" is vital for processing stream data due to its fluctuating stream rates (e.g., bursty stream rates) and the unpredictable number of parallel queries (e.g., queries can be registered/unregistered at run-time) which result in hard-to-predict computing loads and resource requirements. To enable elasticity in a Cloud environment with on-demand load profiles, the used algorithms and data access must lend themselves to parallelization or must be re-engineered to achieve this property.

To address the above problems, this paper introduces an elastic execution model based on a comprehensive suite of novel parallelizing algorithms for incremental computing of continuous query operators on Linked Stream Data. We present the CQELS Cloud implementation of this model and provide a comprehensive evaluation of its scalability and elasticity based on an extensive set of experiments on Amazon EC2. The results show that CQELS Cloud can scale up to throughputs of 100,000s of triples per second for 10,000s of concurrent queries on a cluster of 32 medium EC2 nodes. As we will demonstrate in the paper this is not the ultimate limit of scalability but our approach can scale gracefully to nearly arbitrary loads if more nodes are added.

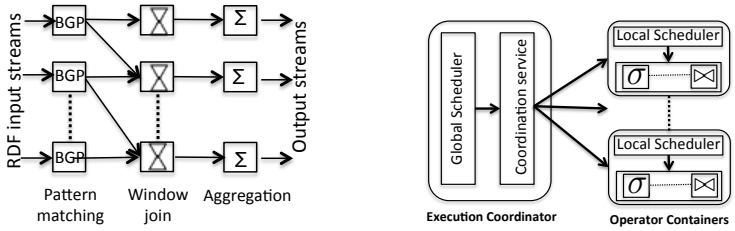
The remainder of this paper is organized as follows: Section 2 describes our abstract execution model for processing continuous queries over Linked Stream Data on a cluster of networked computers. The required parallelizing algorithms

for this execution framework are described in Section 3. Section 4 presents the implementation details of our CQELS Cloud engine and the results of our comprehensive experimental evaluation. We discuss related work in Section 5 and then present our conclusions in Section 6.

2 Elastic execution of continuous queries

This section describes the elastic execution model for processing continuous queries over Linked Stream Data. The execution model is based on the Linked Stream Data model and the query semantics of the CQELS query language (CQELS-QL) [19]. The Linked Stream Data model [3, 5, 19] is used to model both stream data represented in RDF and static RDF datasets. CQELS-QL is a declarative continuous query language and is a minimal extension of SPARQL 1.1 with additional syntactical constructs to define sliding window operators on RDF data streams.

Our execution model accepts a set of CQELS-QL queries over a set of RDF input streams which produce a set of output streams (RDF streams or relational streams in SPARQL-result formats). These queries will be compiled to a logical query network. The network defines which query algebras the input stream data should go through to return results in the output streams. Figure 1a shows and illustrating example: First triples are extracted from RDF streams by a set of pattern matching operators (basic graph patterns), which then are sent to a number of sliding window joins (derived from the original queries). The triples resulting from these operations are then sent to a set of aggregation operators. The outputs of the aggregation operators are the result streams of the original queries comprising the network.



(a) A continuous query network (b) Elastic execution architecture

Fig. 1: Elastic execution model for Linked Stream Data

Our execution model is based on a distributed architecture as shown in Figure 1b. The logical query network is mapped to a processing network distributed among processing nodes, called Operator Containers (OCs) (see Section 4). The Global Scheduler of the Execution Coordinator uses the Coordination Service to distribute the *continuous processing tasks* to OCs to trigger the corresponding executions concurrently. Similar to Eddies [2, 19], the continuous processing tasks are input stream elements associated with *operator signatures* that indicate which physical operators the stream elements need to be processed to satisfy their processing pipeline (mandated by the original queries). Each OC hosts a set of physical query operators that process input streams and forward the output to the consuming operators in the network. The Local Scheduler of an OC is responsible for scheduling the execution of processing tasks assigned

by the Global Scheduler to make the best use of computing resources allocated for that OC. This execution architecture supports elasticity by allowing the machines running OCs to join and leave the network dynamically at runtime. The Coordination Service monitors the OC instances for failure or disconnection. In the event that an OC instance leaves, the Coordination Service will notify the Global Scheduler to re-balance / re-assign the "missing" processing to the rest of the network. The Coordination Service maintains all processing state of the whole network whereas each OC only has a mirrored copy of the processing state necessary for its processing tasks. When an OC instance leaves, the Coordination Service will recover its processing state (progress) from the last successful processing state and reassign the tasks to other nodes in the network. When a new OC instance joins, it will notify the Coordination Service of its availability to receive tasks. To start processing assigned tasks, each OC has to synchronize its processing state with the processing state of the Coordination Service. To avoid a single point of failure problem through a failure of the Coordination Service, its processing state is replicated among a set of machines.

Distributing computing over multiple computers supports scalability but also incurs performance costs because bandwidth and latency of the network are several orders of magnitude worse than those of RAM (scalability outweighs the costs as demonstrated by Grids as the predecessor to Clouds). Therefore, to avoid the huge communication overheads of processing raw, "very wordy" RDF streams, we use the dictionary encoding approach of CQELS [19] for compression, i.e., the processing state – input triples, mappings, etc. – is represented as integers. Another way of reducing the communication cost is grouping the operators processing the same inputs into one machine. For example, instead of sending an input triple to multiple machines to apply different triple matching patterns, these triple matching patterns can be grouped to a single machine to avoid sending multiple copies of an input triple (see Section 3.1). Furthermore, the data exchanged among machines is combined into larger units to reduce the additional overhead of packaging and transferring single data items (this is standard "good networking" practise applied in any networking protocol, see Section 4). On the other hand, network latency is much lower than disk latency. Therefore, the performance cost of storing and retrieving data on/from other nodes in a network is comparable to the cost of local disk access. Thus, data that cannot be stored entirely in memory is distributed to multiple partitions on the disks of multiple computers "in parallel." For instance, the intermediate results from a sub-query to a static data set might have millions of mappings [19] which can be split and indexed (for future search and retrieval operations) on multiple nodes. As a result, the access bandwidth to persistent data can be increased if the number of processing node increases. Interestingly, on typical server hardware, the sequential access to a disk is comparably faster than completely random access to RAM [17]. Storing data to be fetched in sequential blocks and providing distributed indexes to such blocks will overcome the I/O bottleneck of accessing a single disk on a single computer. The availability of data to be searched and fetched can also be increased by increasing the data replicating ratio.

While these strategies may seem overly complicated and heavy, they ensure optimal resource usage, fault tolerance, elasticity, and scalability in line with the

accepted state of the art in distributed computing and cloud systems and thus are necessary to achieve our performance and scalability goals.

3 Parallelizing algorithms for incremental evaluation of continuous query operators

The continuous query operators of CQELS-QL, e.g., join, filter, and aggregate [19], are defined as operators on bags of mappings. Each of these operators consumes a set of bags of mappings and returns a bag of mappings which then can be used as intermediate mappings to be consumed in another operator of an execution pipeline, called *upper operator*. In each pipeline, the inputs at the input side are bags of mappings stored in the window buffers of the concerned sliding window operators and the operators return a bag of mappings on the output side. The execution pipelines are continually applied to the input streams. The parallelizing approaches for this continuous evaluation are presented and discussed in the following sections.

3.1 Parallelizing the incremental evaluation of sliding window operators

To support maximum throughput and efficiency in a Cloud environment, we designed incremental evaluation strategies for sliding window operators [10] which minimize computational efforts by avoiding re-computations and which can distribute the incremental computing tasks to multiple processing nodes. The incremental computing tasks are triggered by two types of events: the arrival or the expiration of a stream data item, e.g., a mapping. We use a *negative tuple* approach [10, 13] to signal expiration events, called *negative mappings*. To implement this, a continuous processing task is assigned to an OC as a mapping with extra information to indicate if it is a negative mapping or a new mapping and to provide operator signatures. The operator signatures are information about which operator instances should process the mapping [2]. These signatures are used by the Local Scheduler to construct a processing pipeline corresponding to that task.

Stateless operators, e.g., select, project, filter, and triple matching, do not have to maintain a processing state for incremental evaluation, i.e., they do not need to access any previous input. Therefore, parallelizing the incremental evaluation of stateless operators is straight-forward, since a new or expired mapping can be processed directly in parallel to produce the corresponding new or expired mappings. The time necessary for each execution of this kind is usually much shorter than the time spent for transferring an input data item. To save communication cost, we group executions that consume the same input in a single machine. Additionally, grouping executions might improve performance for evaluating concurrent queries. For instance, instead of iterating over all triple patterns to check if an input triple is matched with the constants of these triple patterns, the indexes of such constants can be used to efficiently find which triple patterns have constants being matched with values of the input triple. Grouping operators does not limit the possible degree of parallelization because the processing load is split and distributed per stream data item.

For stateful operators such as join and aggregation, the previous input data have to be consulted to compute the updates necessary when a new or a negative

mapping is received. To parallelize the incremental evaluation of these operators, the workers involved in the processing have to be coordinated to share a consistent processing state. How this can be done will be discussed in the next sections. Due to space limitations, we discuss only two operators in the the following which benefit the most from parallelizing their executions: multiway join and aggregation. Their parallel, incremental algorithms are presented below.

3.2 Parallel Multiway join

Based on an extensive analysis of queries and their executions, we have found that multiway joins are a dominating cost factor in typical queries [18, 20]. Inspired by the MJoin approach [23], we developed a parallel multiway join that works over more than two input buffers which are used to store data for sliding windows defined in the continuous execution model introduced in Section 2. As the multiway join is symmetric, without loss of generality, the evaluation of a new mapping μ_1 being inserted into the input buffer R^1 is illustrated in Figure 2. When a mapping μ_1 is inserted into the input buffer R^1, it will be used to probe one of the other input buffers $R^2 \cdots R^n$. Let us assume that R^2 is the next input buffer in the probing sequence. For each mapping μ_2^i in R^2 that is compatible with μ_1, an intermediate joined mapping in the form $\mu_1 \circ \mu_2^i$ is generated. Subsequently, $\mu_1 \circ \mu_2^i$ is recursively used to probe the other input buffers to generate the final mappings. When a buffer that does not return any compatible mapping is found, the probing sequence stops. Following the negative tuple approach [10, 13], if a negative mapping arrives, it is used to invalidate expired outputs. This invalidation operation is done by the next in the query pipeline, which consumes the multiway join output as its input buffer.

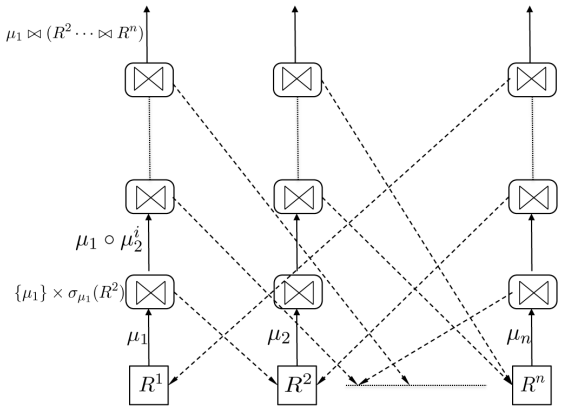


Fig. 2: Multiway join process.

The probing sequence and the invalidation operation are triggered by a new mapping or negative mapping and can be executed in parallel on multiple machines given that each machine can access the same set of input buffers. Algorithm 1 shows our incremental evaluation algorithm for the parallel multiway join with n input buffers. Lines 2–5 handle new mappings and line 9 is for forwarding a negative mappings to the upper operator. Lines 2 and 3 synchronize the window $W[i]$ in all machines running this operator. Line 4 checks if the current multiway join is supposed to compute the probing sequence triggered from

this new mapping by verifying its operator signatures. Line 5 calls the recursive sub-routine *probingPropagate* (see Algorithm 2) to initialize the probing sequence. The next step in the probing sequence is defined in the sub-routine $findNextProbWin$ in line 2. It chooses the next window buffer $W[i_{next}]$ to be probed to find a compatible mapping to forward to in the next step (line 3). The sub-routine $findNextProbWin$ can be used to inject additional adaptive optimization algorithms for multiway joins [8]. Note that, a multiway join might use buffers to store intermediate mappings from subqueries on static data. They are just a special case of buffers used for sliding windows.

Algorithm 1: Parallel Multi-Way Join

```
   Input: n input buffers W1,..,Wn
 1 if a new mapping μ arrives at window W[i] then
 2     remove expired tuples from all windows;
 3     W[i].insert(μ);
 4     if μ is assigned for this multiway join instance then
 5         probingPropagate(μ,{W[1],..,W[n]}\{W[i]}) ;
 6     end
 7 end
 8 else
 9     propagate negative mapping to upper operator;
10 end
```

For different window joins which share join predicates over the same RDF streams, we group them into a shared computing network, i.e., a shared join, to reduce network communication overhead as well as to avoid wasting resources due to redundant computation. A shared join has a single execution plan for multiple queries and produces multiple output streams for each separate query involved. The shared join operator consists of two components: the join component and the routing component.

Algorithm 2: Probing propagation *probingPropagate*

```
   Input: μ, k sliding windows {W[i1],..,W[ik]}
 1 if k==0 then
 2     i_next ← findNextProbWin( μ,{W[i1],..,W[ik]});
 3     for μ* ∈ W[i_next].probe(μ) do
 4         probePropagate(μ ∘ μ*, {W1,..,Wk}\{W[i_next]});
 5     end
 6 end
 7 else
 8     dispatch μ ;
 9 end
```

The join component produces a single intermediate output stream for all queries, and the routing component routes the valid output items to the corresponding output buffer of each continuous query [14]. The join component dominates the query load because the complexity of an m-way join operation is much higher than that of a filtering operation of the routing component for n

queries (n is usually smaller than $\prod_{i=1}^{i=m} W_i$ where W_i is the size of buffer i of the m-way join).

To share the computations and the memory when processing multiple joins that have the same set of input buffers, the multiway join algorithm can be used to build a shared join operator, i.e, the multiple join operator. Let us assume m multiple window joins $W_j^1 \cdots \bowtie W_j^m$ where j=1...k and W_j^i is a window buffer extracted from the RDF stream $S^i, i = 1...n$. Let W_{max}^i be the window buffer that has a window size equal to the maximum window size over all $W_j^i, j = 1...k$. Then, the following containment property [14] holds:

$$W_j^1 \cdots \bowtie W_j^n \subseteq W_{max}^1 \cdots \bowtie W_{max}^n$$

Due to this property, the processing of the query $W_{max}^1 \cdots \bowtie W_{max}^n$ produces an output that contains the outputs of all queries $W_j^1 \cdots \bowtie W_j^n, j = 1...k$. Therefore, the join component only has to execute a single multiway query for $W_{max}^1 \cdots \bowtie W_{max}^n$. In the routing component, each resulting mapping then has to be routed to the query that takes it as an input. We call the routing component of the multiple join operator *router*. The router maintains a sorted list of the windows relevant to each join. The windows are ordered by window sizes in increasing order. Each output mapping is checked if its constituent mappings are valid within valid time intervals of the windows of a query. When a mapping satisfies the time condition of the query, it is routed to the query's output buffer. Figure 3 illustrates a multiple join operator for 3 queries over 2 streams S^1 and S^2 where $Q_1 = W_1^1 \bowtie W_1^2$, $Q_2 = W_2^1 \bowtie W_2^2$ and $Q_3 = W_3^1 \bowtie W_3^2$. This multiple join operator connects the 2-way join operator $W_{max}^1 \bowtie W_{max}^2$ to its router where $W_{max}^1 = W_3^1$ and $W_{max}^2 = W_3^2$. The left-hand side of the figure shows how the router delivers the output mappings from the 2-way join to each query. For instance, when the new mapping $\langle a_1, b_6 \rangle$ arrives at the stream S^1, the 2-way join probes the input buffer W_{max}^2 to generate two output mappings $\langle a_1, b_6, c_1 \rangle$ and $\langle a_1, b_6, c_5 \rangle$. Based on the window conditions of each query, the router routes $\langle a_1, b_6, c_5 \rangle$ to Q_1 and Q_2 and both $\langle a_1, b_6, c_1 \rangle$ and $\langle a_1, b_6, c_5 \rangle$ to Q_3.

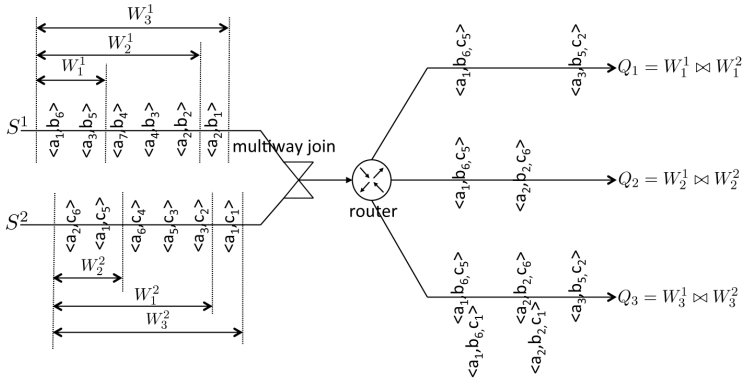


Fig. 3: Shared windows example

Generally, the concurrent queries registered to the system only share subsets of the streams involved in their queries. Therefore, we create a network of

multiple join operators to enable sharing of the execution of subqueries for a group of queries. For each group of joins that share the same set of streams a multiple join operator is created. Figure 4 illustrates a network of 4 queries over 4 streams S^1, S^2, S^3 and S^4, where $Q_1 = W_1^1 \bowtie W_1^2 \bowtie W_1^3$, $Q_2 = W_2^2 \bowtie W_2^3$ $Q_3 = W_3^2 \bowtie W_3^3 \bowtie W_3^4$ and $Q_4 = W_4^2 \bowtie W_4^3 \bowtie W_4^4$. This network is composed of three multiple join operators $\bowtie_1^M$, $\bowtie_2^M$ and $\bowtie_3^M$ where $\bowtie_1^M$ is for Q_1, $\bowtie_2^M$ is for Q_2 and $\bowtie_3^M$ for Q_3 and Q_4. Due to space limits, we refer the reader to [18] for a detailed technical discussion of this setting.

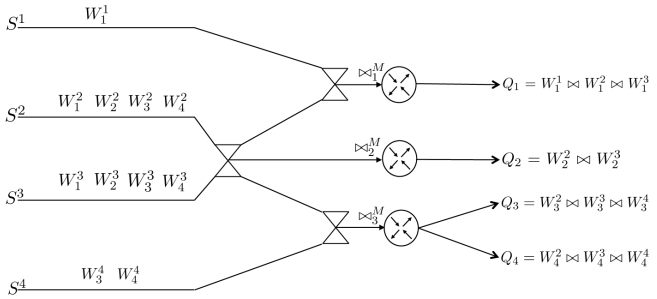


Fig. 4: A network of multiple join operators.

3.3 Aggregation

An aggregation operator $\mathcal{AGG}_{f_1(A_1),f_2(A_2),...,f_k(A_k)}$ maps each input mapping to a group G and produces one output mapping for each non-empty group G. The output has the form $<G, Val_1, .., Val_k>$, where G is the group identifier (grouping key) and Val_i is the group's aggregate value of the function $f_i(A_i)$. The value Val_i is updated whenever the set of mappings in G changes in the case of new and expired mappings. Both new mappings and expired mappings can result in an update to the value of a group and the aggregate operator needs to report the new value for that group. The incremental computation of each group can be done independently, therefore, they can be assigned to different machines to be computed in parallel. The Global Scheduler can distribute the incremental aggregation tasks by routing new mappings and expired mappings to processing nodes based on their grouping keys. A simple routing policy is splitting based on the hash values of grouping keys. The algorithms for incremental computation of aggregation over sliding windows in [13, 10] can be used to incrementally update the value of each group.

4 Implementation and Evaluation

We implemented our elastic execution model and the parallel algorithms using ZooKeeper [16], Storm[1] and HBase.[2] The architecture of CQELS Cloud is shown in Figure 5. The Execution Coordinator coordinates the cluster of OCs using coordination services provided by Storm and HBase which share the same Zookeeper cluster. The Global Scheduler uses Nimbus,[3] an open source EC2/S3-compatible Infrastructure-as-a-Service implementation, to deploy the operators'

[1] `http://storm-project.net/`
[2] `http://hbase.apache.org/`
[3] `http://www.nimbusproject.org/`

code to OCs and monitor for failures. Each OC node runs a Storm supervisor which listens for continuous processing tasks assigned to its machine via Nimbus. The processing tasks that need to process the persistent data use the HBase Client component to access data stored in HBase. The machines running an OC also host the HDFS DataNodes of the HBase cluster. The DataNodes are accessed via the OC's HRegionServer component of HBase.

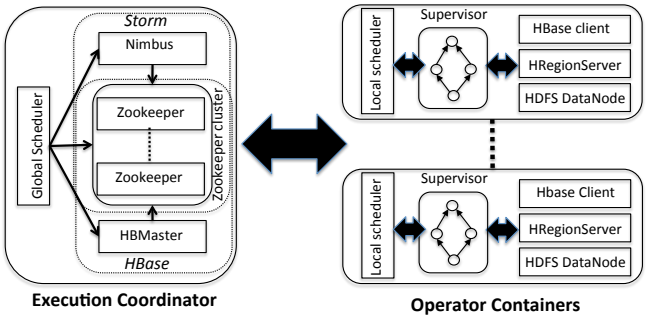


Fig. 5: CQELS Cloud architecture

Machines running OCs communicate directly without using intermediate queues via ZeroMQ[4] used inside Supervisors. Their available communication bandwidths are optimized by ZeroMQ's congestion detection mechanism. Based on ZeroMQ, OCs use inter-process communication interfaces as defined by Storm's "spouts" (stream source) and "bolts"(processing) infrastructure by sending tuples. Links among spouts and bolts in a Storm topology indicate how tuples should be passed among them. In CQELS Cloud spouts are used to stream data from sources. Bolts receive any number of input streams from upstream processes that trigger processing pipelines and continually output results as new streams. Processing queries works as follows: Using the parallelizing algorithms presented in Section 3 a logical query plan is transformed to a Storm topology composed of bolts for all query operators. The stream inputs of the query plan will be mapped to spouts which stream data to special spouts for encoding raw RDF stream. Inner query operators will be connected to the encoding spouts following the acyclic query graph of the logical query plan. Finally, the outermost bolts will be connected to the decoding bolts to transform the query output back into the decoded version.

Data is routed to a bolt using a routing policy, called *stream grouping*. In CQELS Cloud we use three stream grouping policies provided by Storm, namely, *shuffle grouping*, *all grouping* and *fields grouping*. The *shuffle grouping* policy is used to evenly distribute the input to the parallel stateless operator instances. The *all grouping* policy is used to synchronize the input buffers of the parallel multiway join algorithm in Section 3. The *fields grouping* policy is used to route mappings that have a certain key to the aggregation operator instance responsible for computing the aggregated value of the group corresponding to that key (see Section 3.3).

In CQELS Cloud, input mappings are ordered and put into batches carried by Storm tuples that are routed among bolts and spouts. For optimization

[4] `http://www.zeromq.org/`

purposes, we encode the data, thus, mappings contain only fixed-size integers. Consequently, batching a series of mappings in an array of integers will reduce the delay of consuming data from the network as well as the serialization and de-serialization. On top of that, this enables us to employs fast compression algorithms such as [9, 21] for further speed-up. As the input streams coming to the buffers of an operator instance running in one machine can be unordered, we use the well-established heart-beat approach [24] for guaranteeing the strict order of stream inputs to ensure the correct semantics of the continuous operators.

We use HBase to store the dictionary for the encoding and decoding operations. The dictionary is partitioned by the keys of RDF nodes to store on OC nodes. The encoding and decoding tasks for RDF nodes for the input and output streams are evenly distributed among the nodes using the *fields grouping* policy on hash values of the RDF nodes. HBase stores the written data in memory using its MemStore before flushing partitions of the dictionary into sequential disk blocks on the disks of the destined OC nodes. This allows the writing operations of the dictionary encoding to be carried out in parallel and has high throughput on OC nodes. Along with the dictionary, HBase is also used to store and index intermediate results from subqueries (which can be huge) on static RDF datasets [19]. This data is stored and indexed by the keys used for the lookup operation which facilitates high throughput of the probing and fetching operations for the parallel processes of the multiway join algorithm.

All state is kept in the Zookeeper cluster which enables high access availability through its built-in replication service. Its reliability, performance and availability can be tuned by increasing the number of machines for the cluster. However, Storm does not directly support state recovery, i.e., resuming a computation state of a node when it crashes. Therefore, we implemented the state recovery for OCs ourselves via timestamped checkpoints which is aided by Storm's concept of guaranteed message processing. Storm guarantees that every input sent will be processed by its *acknowledgment* mechanism. This mechanism allows a downstream processing node to notify its upstream processing node "up to which time point" it has processed downstream inputs successfully. The checkpoint timestamps are encoded in the acknowledgement messages of Storm. This provides an implicit recovery checkpoint when a new node taking over should restart the computation by reconstructing the processing state up to the last "successful" state.

It is important to note that while we use a sophisticated combination of top-class distributed infrastructures, this is "just" the programmatical foundation we base on, in order not to have to deal with classical distributed computing problems ourselves. The mere use of this infrastructures would neither provide elasticity nor scalability of stream processing. Our main contribution lies in the parallel algorithms, operator implementation, and data structures used for this and the highly optimized implementation of these concepts. For example, our operator implementations do not map 1-to-1 to Storm's bolts and spouts, but those are used as a communication components that trigger an execution pipeline from inputs tagged with data workflows.

4.1 Evaluation Setup

To demonstrate the efficiency, performance, scalability and elasticity of our approaches, we evaluated CQELS Cloud using a real deployment on the Amazon

EC2 cloud. Our current version of CQELS Cloud uses Zookeeper 3.4.5-cdh4.2, Storm 0.8.2 and HBase 0.94.2. The configuration of the Amazon instances we use for all experiments is "medium" EC2 instances, i.e., 3.5 GB RAM, 1 virtual core with 2 EC2 Compute Units, 410 GB instance storage, 64 Bit platform, moderate I/O performance. A cluster includes 1 Nimbus node, 2 Zookeeper nodes, 1 HBase Master node and 2-32 OC nodes within the same administrative domain. In each experiment, we registered a set of queries or operators and then stream a defined number of stream items to measure the average processing throughput (triples/second or mappings/second).[5] This process is repeated for 2, 4, 8, 16 and 32 OC nodes. To test the elasticity, OC nodes are added and removed during the experiments without stopping the CQELS Cloud engine. Our baseline is given by processing the same queries on a single node, i.e., we show how the global scalability can be improved by adding more nodes. This shows the benefits of our central contribution which is the data- and operator-aware load distribution algorithm for stream queries. Our evaluation focuses on showing how multi-joins scale – as in Linked Data star-shaped n-way joins are the dominating operation – when increasing the number of processing nodes, rather than comparing the performance of different join operators. We conducted sets of experiments:[6]

Operator scalability: We evaluate the scalability and overheads of our algorithms proposed in a controlled setting by increasing the number of machines running OCs. For each operator, we fix some parameters to have that kind of processing loads which shows a clear impact of the parallelization. The used data is generated randomly.

Parallel queries: We evaluate the performance and scalability of CQELS Cloud when processing multiple concurrent queries over the Social network scenario of LSBench [20]. LSBench is a benchmarking system for Linked Stream Processing engines which provides a data generator to simulate Social Network streams. We choose LSBench over SRBench [25], because LSBench enables us to control the experimental settings to demonstrate the scalability properties of our system. Furthermore, with LSBench, we can choose four queries with different complexities for our experiments: Q1 (simple matching pattern), Q4 (3-way join on streams only), Q5 (3-way join on stream and static data) and Q10 (3-way join and aggregation). We randomly vary the constant of Q1 to generate 100-100,000 queries and the window sizes of Q4, Q5, Q10 to generate 10-10,000 queries for each. We use LSBench to generate a dataset for 100k users. The dataset presents a social network profile of 121 million triples and 10 billion triples from 5 streams. This dataset can generate tens of millions of intermediate mappings, e.g., more than 33 million for Q5.

The validity of the throughput measurement for a stream processing engine is defined by the correctness and completeness of its output results [20]. For instance, the mismatches of the outputs generated from different implementations of time-based sliding windows may lead to incorrect interpretations when comparing throughput among them [7]. In our experiments, as we only measure

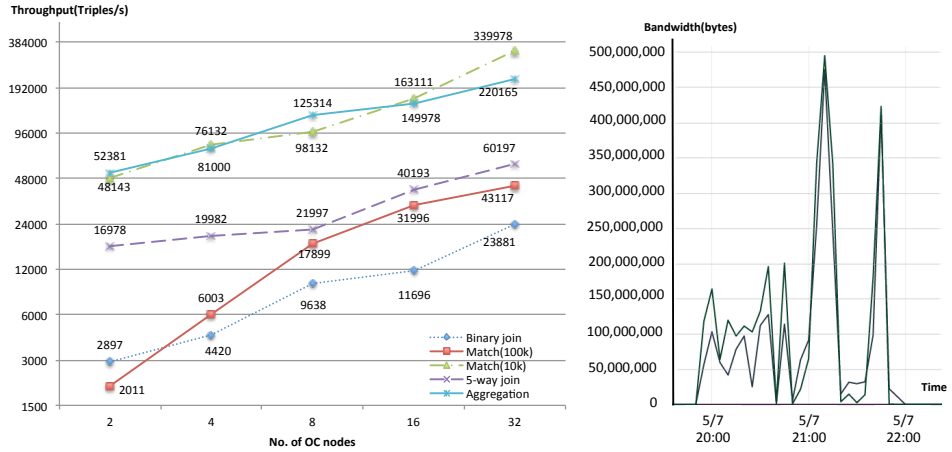
[5] In the following we mean average processing throughput when talking about processing throughput.

[6] A detailed guide for how to reproduce our experiments on Amazon EC2 can be found at `https://code.google.com/p/cqels/wiki/CQELSCloud`.

the throughput of one implementation with different configurations, we verify the validity by two types of tests on only count-based windows for any query: unit tests for operator implementations and mismatch tests for output results of queries defined in LSBench [20].

4.2 Evaluation Results

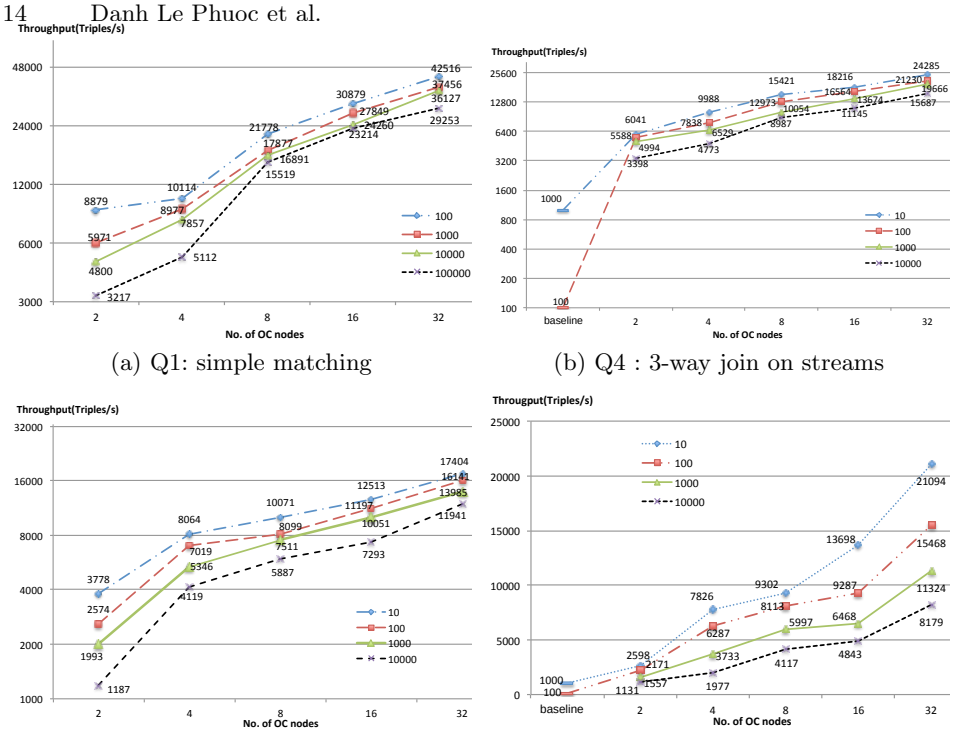
Figure 6a shows the results of 5 experiments: The first two experiments are for the triple matching operator with 10,000 and 100,000 concurrent triple patterns. The next two are for the 5-way join and aggregation. The 5-way join has 5 count-based windows of 100,000 mappings and the selectivity of the join predicates is 1%. The aggregation is connected to a window of 1 million mappings. The last one is for a binary join between a window buffer and a bag of 10 million mappings stored in HBase. Note that, the 5-way join and the binary join are two distinct examples to show how the multi-join scales when increasing the number of processing nodes rather than comparing the performance of different join operators.



(a) Throughput of Operators (b) Peak network traffic

Fig. 6: Operator behaviors

From the graphs we can see that the throughputs increase linearly with the increasing numbers of OC nodes (logscale on y-axis). This means that the performance increases nearly linearly with the number of processing nodes which is close to the optimal theoretical limit. In terms of throughput, the light-weight operators like triple matching (for 10k triple patterns) and aggregation achieve more than 100k inputs/second with 8 nodes. The ones with heavier query load like triple pattern matching for 100k patterns or the binary join deliver even better scale factors. To confirm that network congestion did not effect the scalability significantly, we show the accumulated network bandwidth of the experiments in Figure 6b. The maximum bandwidth used is less than 500MB/sec for all experiments. This means that we use only a maximum of 4GBit/sec of the 10GBit/sec network offered by the Amazon EC2 cluster and thus the effects incurred by network communication are negligible.



(a) Q1: simple matching

(b) Q4 : 3-way join on streams

(c) Q5: 3-way join on streams and static data

(d) Q10: Join and Aggregation

Fig. 7: Multiple queries on Social Network streams

Figure 7 shows more results for four types of queries. Each graph shows the throughputs of an experiment for a certain number of concurrent queries. Similarly to the previous experiments, the throughput of CQELS Cloud increases linearly for a constant query load when we increase the number of OC nodes. When we increase the query load, e.g., by a factor 10, the throughput only decreases approximately by a factor of 2. Considering the best throughputs of standalone systems reported in [20] with the same settings for query Q4 and Q10 as baselines, approximately 1000 triples/second for 10 queries and 100 triples/second for 100 queries, the CQELS Cloud can achieve more than 24 times and 210 times of these baseline throughputs for 10 and 100 queries respectively. Furthermore, the more concurrent queries the system handles, the better is the scalability in relation to the baselines. This may be explained by the increasing reuse of intermediate results by multiple queries. This demonstrates excellent scalability and shows the advantages of our data- and query-aware load distribution strategy.

5 Related work

To the best of our knowledge, CQELS Cloud is the first system addressing elastic and scalable processing for Linked Stream Data but our approaches touch on a number of areas.

Linked Stream Data processing: CQELS Cloud has a data model and query semantics similar to Streaming SPARQL [4], C-SPARQL [3], CQELS [19], etc. However, all these are designed to run on a single machine, while CQELS Cloud goes beyond that and specifically focuses on scalability issues – discussed in

detail in [20] – by distributing the computing and defining an architecture and algorithms suitable for Cloud deployments. There is also preliminary work for distributed stream reasoning on S4 [15] which provides some scalability results for certain queries and reasoning but is not a complete system like ours.

Distributed stream processing engines: Classical distributed stream processing engines such as Borealis [6] and StreamCloud [12] are the distributed versions of the stand-alone engines and only support the relational data model or very generic stream data primitives and operators. They can be used as black-boxes to delegate Linked Data Stream processing tasks, but, as shown in [19, 20], the overhead of data transformation and query rewriting seriously impact on scalability, rendering them no competitive option. However, approaches and techniques from distributed stream processing such as load balancing, operator placement and optimizations [11] can be used to improve the performance of CQELS Cloud.

Generic parallel stream computing platforms: Storm and S4 are the most popular elastic stream computing platforms and provide very generic primitives and data types for representing stream elements. None supports declarative query languages nor the Linked Data model. On top of that, neither Storm nor S4 support correlating data from distributed storages as CQELS Cloud does with HBase. There are also other systems such as Kafka[7] and Scribe[8] that target specific, narrow application domains (and do so efficiently). For instance, Kafka and Scribe are used to programmatically create reliable and scalable processing pipelines for stream logs in LinkedIn and Facebook, respectively. We consider these works as complimentary work that we can draw on to potentially improve our implementation.

6 Conclusions

Our goal was to devise scalable algorithms and an infrastructure for Linked Stream processing that scales to realistic scenarios with high stream frequencies, large numbers of concurrent queries and large dynamic and static data sizes along with the possibility to deploy them in a hosted Cloud environment to achieve elasticity in the load profiles and enable "pay-as-you-go" scenarios. The experimental evaluations show that we have achieved this aim to a large degree: Our algorithms and implementation exhibit excellent scalability in the Cloud, essentially supporting arbitrary loads only limited by the number of nodes and the hardware and software characteristics of the used Cloud platform. We achieved this through a completely distributed design with novel parallel algorithms for Linked Stream processing, along with a number of optimization techniques adapted for our purpose and a well-justified combination of sophisticated distributed computing infrastructures. CQELS Cloud provides the same or even better scalability as the established stream processing approaches outside the Linked Data world and will help to make the Linked Data paradigm an increasingly serious competitor in this area.

[7] `http://kafka.apache.org/`
[8] `https://github.com/facebook/scribe`

References

1. D. Anicic and P. Fodor. EP-SPARQL: a unified language for event processing and stream reasoning. In *WWW*, New York, NY, USA, 2011. ACM.
2. R. Avnur and J. M. Hellerstein. Eddies: continuously adaptive query processing. *SIGMOD Rec.*, 29:261–272, May 2000.
3. D. F. Barbieri, D. Braga, S. Ceri, and M. Grossniklaus. An execution environment for C-SPARQL queries. In *EDBT 2010*, New York, NY, USA, 2010. ACM.
4. A. Bolles, M. Grawunder, and J. Jacobi. Streaming SPARQL – extending SPARQL to process data streams. In *ESWC*, pages 448–462, Berlin, Heidelberg, 2008.
5. J.-P. Calbimonte, O. Corcho, and A. J. G. Gray. Enabling ontology-based access to streaming data sources. In *ISWC*, Berlin, Heidelberg, 2010. Springer-Verlag.
6. Y. A. Daniel J. Abadi. The Design of the Borealis Stream Processing Engine. In *CIDR 2005*, pages 277–289, 2005.
7. D. Dell'Aglio, J.-P. Calbimonte, M. Balduini, Ó. Corcho, and E. Della Valle. On Correctness in RDF stream processor benchmarking. In *ISWC*, 2013.
8. A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1, January 2007.
9. P. Elias. Universal codeword sets and representations of the integers. *IEEE Trans. Inf. Theor.*, 21(2):194–203, Sept. 2006.
10. T. Ghanem, M. Hammad, M. Mokbel, W. Aref, and A. Elmagarmid. Incremental Evaluation of Sliding-Window Queries over Data Streams. *TKDE*, 19(1), 2007.
11. L. Golab and M. T. Özsu. *Data Stream Management*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.
12. V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, and P. Valduriez. Streamcloud: A large scale data streaming system. In *ICDCS*, 2010.
13. M. Hammad, W. G. Aref, M. J. Franklin, M. F. Mokbel, and A. K. Elmagarmid. Efficient execution of sliding-window queries over data streams. Technical Report 03-035, Purdue University, Dept. of Computer Science, 2003.
14. M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid. Scheduling for shared window joins over data streams. In *VLDB*. VLDB Endowment, 2003.
15. J. Hoeksema and S. Kotoulas. High-performance Distributed Stream Reasoning using S4. In *1st International Workshop on Ordering and Reasoning, ISWC*, 2011.
16. P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *USENIX*, 2010.
17. A. Jacobs. The pathologies of big data. *Queue*, 7(6):10:10–10:19, July 2009.
18. D. Le Phuoc. *A Native And Adaptive Approach for Linked Stream Processing*. PhD thesis, National University of Ireland, Galway, 2013.
19. 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.
20. D. Le Phuoc, M. Dao-Tran, M.-D. Pham, P. A. Boncz, T. Eiter, and M. Fink. Linked Stream Data Processing Engines: Facts and Figures. In *ISWC*, 2012.
21. D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *CoRR*, abs/1209.2137, 2012.
22. J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers. Big data: The next frontier for innovation, competition, and productivity. Technical report, McKinsey Global Institute, June 2011.
23. V. J. F. Naughton and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB*. VLDB Endowment, 2003.
24. U. Srivastava and J. Widom. Flexible time management in data stream systems. In *ACM SIGMOD-SIGACT-SIGART*, New York, NY, USA, 2004. ACM.
25. Y. Zhang, M.-D. Pham, Ó. Corcho, and J.-P. Calbimonte. SRBench: A Streaming RDF/SPARQL Benchmark. In *ISWC*, 2012.