Optimizing the Performance of Concurrent RDF Stream Processing Queries

Chan Le Van(✉), Feng Gao, and Muhammad Intizar Ali

INSIGHT Centre for Data Analytics, National University of Ireland, Galway, Ireland
{chan.levan,feng.gao,ali.intizar}@insight-centre.org

Abstract. With the growing popularity of Internet of Things (IoT) and sensing technologies, a large number of data streams are being generated at a very rapid pace. To explore the potentials of the integration of IoT and semantic technologies, a few RDF Stream Processing (RSP) query engines are made available which are capable of processing, analyzing and reasoning over semantic data streams in real-time. This way, RSP mitigates data interoperability issues and promotes knowledge discovery and smart decision making for time-sensitive applications. However, a major hurdle in the wide adoption of RSP systems is their query performance. Particularly, the ability of RSP engines to handle a large number of concurrent queries is very limited which refrains large scale stream processing applications (e.g. smart city applications) to adopt RSP. In this paper, we propose a shared-join based approach to improve the performance of an RSP engine for concurrent queries. We also leverage query federation mechanisms to allow distributed query processing over multiple RSP engine instances in order to gain performance for concurrent and distributed queries. We apply load balancing strategies to distribute queries and further optimize the concurrent query performance. We provide a proof of concept implementation by extending CQELS RSP engine and evaluate our approach using existing benchmark datasets for RSP. We also compare the performance of our proposed approach with the state of the art implementation of CQELS RSP engine.

Keywords: Linked Data · RDF Stream Processing · Query optimization

1 Introduction

A merger of semantic technologies and stream processing has resulted into RDF Stream Processing (RSP). RSP engines refer to the stream processing engines which have the capability to process semantically annotated RDF data streams. Over the past few years, different RSP engines have been proposed, such as CQELS [16], C-SPARQL [4], and SPARQLStream [6] etc. RSP engines continuously consume streaming RDF data as an input and generate query results as an output. An RSP query is registered once and executed constantly. The continuous query execution is a resource intensive task and most of the existing RSP engines suffer from scalability and performance issues while processing

E. Blomqvist et al. (Eds.): ESWC 2017, Part I, LNCS 10249, pp. 238–253, 2017.
DOI: 10.1007/978-3-319-58068-5_15

multiple concurrent RSP queries. The inability of the RSP engines to handle a large number of concurrent queries is a major performance bottleneck, hence, a main obstacle to the wider adoption of RSP engines. This RSP performance bottleneck has different causes. For example, in CQELS, each input query is separately compiled and parsed into an individual execution plan with a designated data buffer assigned to that particular query. Therefore, the performance of the engine would definitely degrade whenever multiple concurrent queries are registered to the engine. Many real-time applications consuming continuous data streams often share input data streams for a large number of concurrent queries. Therefore, we foresee a great opportunity to address the performance and scalability issues of RSP engines by proposing a resource sharing strategy where multiple concurrent queries could share the resources required to process common input data streams. These resource sharing strategies will certainly ease the burden over RSP engines and consequently improve the performance.

In this paper, we propose resource sharing and load balancing techniques to optimize the performance of concurrent RSP queries. We take the concept of the shared join operator (also known as multiple join operator) introduced in [16], to support sharing memory and computation resources. Taking the advantage of queries having the shared input streams, the multiple join operator uses a join component to produce shared results for all queries, and a routing component/router is used to route the output items to the corresponding query. We extend the approach of shared join operator and introduce the creation of join sequences using a *reutilization* metric. This way, a network of shared join operators are formed for multiple queries. The network contains different shared join operators consuming the same input streams, and in order to produce the query results a query evaluation process involves traversing a path in the network, we called this process a join sequence. In our approach, some intermediate results can also propagate to other queries, if they contain triple patterns referring to the same stream data buffers.*Reutilization* metric is used to choose optimal join sequences. We extended the existing implementation of CQELS engine, and named it as CQELS+. We evaluated CQELS+ in comparison to the existing state-of-the-art CQELS implementation, which showcases that CQELS+ is capable of handling a larger number of concurrent queries. We further evaluated CQELS+ performance for different load balancing techniques for multiple instances of RSP engines deployed in a distributed environment. We evaluated the performance of multiple instances of CQELS+ and compare it with the performance of a single engine instance. The evaluation results have revealed that we can have lower query latency by deploying parallel engine instances.

The remainder of the paper is organized as follows; Sect. 2 lays the theoretical foundations for our approach. Section 3 elaborates the details for the shared join operations and load balancing strategies for parallel RSP engines. Section 4 presents the experimental design to evaluate our approach. This section also provides experimental results analysis. Section 5 reviews the literature before we conclude and discuss future plans in Sect. 6.

2 Foundations

In this section, we introduce the basic concepts of RSP, including multi-way join, shared join operator and network of multiple join operators, which forms the basis of our proposed approach [16].

Following the principles of Linked Data [5], the concept of Linked Stream Data [25] was introduced in order to bridge the gap between data streams and Linked Data. Linked stream data helped to simplify the data integration process among heterogeneous streaming sources as well as between streaming and static sources. Below we present the basic concepts of RDF Stream Processing model, which are also followed in CQELS data model.

- **RDF stream** is a bag of elements $\langle (s, p, o) : [t] \rangle$, where (s, p, o) is an *RDF triple* [22] and t is a timestamp.
- **Window operators** are inspired by the time-based and triple-based window operators of CQL data stream management system [3].
- **Stream Graph Pattern** is supplemented into the *GraphPatternNotTriples* pattern[1] to represent window operators on RDF Stream.

2.1 Multi-way Join

The concept to multi-way join was introduced in [19], same concept was re-used for CQELS [16], where authors propose a single multi-way join that works over more than two stream data buffers (also known as window buffers). Window buffers generate and propagate results in a single step rather than creating the join results by passing through a multiple-stage binary joins in the query execution pipeline. Suppose we have n window data buffers W_1, .., W_n involved in a join operation. A multi-way join operates as follows:

1. When a new stream data item (tuple) comes to any window buffers, it is used to recursively probe other window buffers to generate the join output, i.e., when a data item M arrives at W_1, the join processing starts to look for next join candidate in W_2, W_3, ..., W_n.
2. Let us assume W_2 is the next joining candidate, the algorithm will check data inside W_2 to see if there are any existing tuples which are compatible with M.
3. If there is any compatible data item available, the intermediate result sets are created and the algorithm recursively continues until all of the involved window buffers are visited or stops if no compatible data items found.

2.2 Shared Join Operator and Network of Shared Join Operators

Chapter 7 in [16] introduces the *shared join operator* to share the computation among multi-way join operations from queries with the same set of window buffers of RDF streams. As discussed in [16], let us assume we have k multi-way join operators, each involves m window buffers i.e., we have k multi-way

[1] SPARQL 1.1 Grammar: https://www.w3.org/TR/sparql11-query/#grammar.

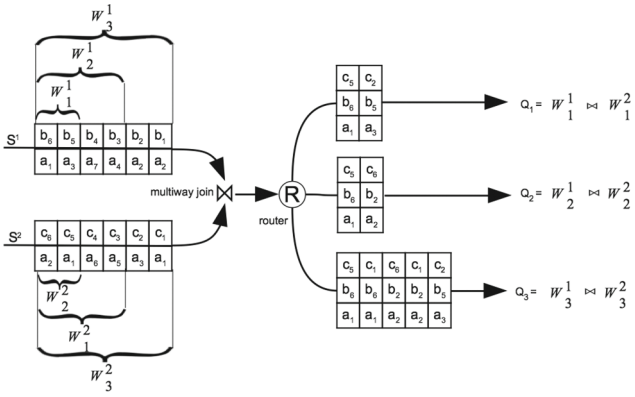


Fig. 1. Shared join operator example

join operators: $(W_1^1 \cdots \bowtie W_1^m)$, ..., $(W_k^1 \cdots \bowtie W_k^m)$. Additionally, let us assume with any $1 \leq i \leq m$, we always have W_1^i, W_2^i, ..., W_k^i referring to the same window buffer (regardless to the window alignment), then the equation below (containment property [12]) is always true:

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

In Eq. 1, W_{max}^i refers to the window buffer in the set $\{W_1^i, W_2^i, ..., W_k^i\}$) with the maximum size. We call W_{max}^i is a shared window buffer.

Intuitively, Eq. 1 tells us that the join results of any multi-way join operator for a fixed set of window buffers is included in the join results of the operator for the set of windows with the maximum length. For example, Fig. 1 shows a shared join operator for three queries Q_1, Q_2 and Q_3 containing 3 multi-way joins $W_1^1 \bowtie W_1^2$, $W_2^1 \bowtie W_2^2$ and $W_3^1 \bowtie W_3^2$, respectively. These queries receive input from two RDF streams: S^1 and S^2. As we can see in the picture, W_{max}^1 will be W_3^1 and W_{max}^2 will be W_3^2. By applying Eq. 1, we have:

$$\forall j \in \{1, 2, 3\}, W_j^1 \bowtie W_j^2 \subseteq W_3^1 \bowtie W_3^2 \tag{2}$$

Leveraging the containment property, the approach in [16] designed a shared join operator. This operator targets in queries sharing the same set of window buffers. It consists of two components: join component and routing component. The join component has the duty of generating the shared results that contain results of all involved queries. The routing component is responsible for filtering the shared results and routing the filtered results to proper queries.

It has been discussed in the literature that in practice the queries registered to the processing engine often share only subsets of input streams. In [16], authors introduced the concept of *network of shared join operators* (NSJO). NSJO can share the execution for sub-queries (part of the whole queries) from a group of queries. NSJO includes a set of shared join operators, where each shared join operator consumes the same window buffers. When a new tuple comes to a shared window buffer, it triggers the join components of related shared join operators in the

NSJO consuming it. The join component then chooses the best cost join sequence (the order of joined window buffers) in all possible join sequences and generates the join output. During the join process, in case the generated results are the output of any shared join operator, the routing component of that shared join operator routes them to proper queries. Otherwise, if the results are further consumed by other shared join operators in the network, the algorithm recursively continues until all related shared join operators have been visited or no results are created.

3 Optimization for Concurrent CQELS Queries

In this section, we elaborate our approach for optimizing CQELS performance under concurrent queries. More specifically, we first discuss how we facilitate NSJO in CQELS+ (the extended CQELS), including how to create the join sequences (the order of joined window buffers) in a join graph, how to calculate the reutilization metric, and provide an example of the join graph. Then, we elaborate how we deploy multiple CQELS+ engines in parallel and distribute the workload among different engine instances in order to obtain better performance.

Algorithm 1. *create_Join_Sequences*(*curr_Join_Vertex*, *queries*)

```
1  while coverage < number_of_queries do
2  |  HAQ ← 0; considering_Patterns ← ∅ ;
3  |  for j ∈ [1..number_of_physical_windows] do
4  |  |  cal ← cal_Reuse_Metric(cur_join_Vertex, physical_Windows[j]);
5  |  |  patterns = get_Considering_Patterns(cur_join_Vertex,
   |  |    physical_Windows[j]) ;
6  |  |  if cal > HAQ then
7  |  |  |  HAQ = cal; next_Buffer = physical_Windows[j];
8  |  |  |  considering_Patterns ← patterns;
9  |  |  end
10 |  end
11 |  if HAQ > 0 then
12 |  |  next_Join_Vertex ← create_Join_Vertex(next_Buffer) ;
13 |  |  next_Join_Vertex.set_Considered_Patterns(considering_Patterns) ;
14 |  |  curr_Join_Vertex.add_Nexts(next_Join_Vertex) ;
15 |  |  prev_Join_Vertex ← curr_Join_Vertex ;
16 |  |  curr_Join_Vertex ← next_Join_Vertex ;
17 |  |  coverage += count_Covered_Queries(curr_Join_Vertex) ;
18 |  |  create_Join_Sequences(current_Join_Vertex, queries) ;
19 |  |  curr_Join_Vertex ← prev_Join_Vertex ;
20 |  end
21 |  else
22 |  |  return;
23 |  end
24 end
```

3.1 CQELS+: Network of Shared Join Operators

According to [16], when a new tuple comes to the shared window buffer(a physical window in our approach) in the NSJO, join sequences in the join graph of this physical window are evaluated to generate the join results for involved queries. In our approach, this join graph is built at the query registration time based on a metric called *reutilization*. This metric is defined to create the optimal join sequences in the graph. More specifically, in each step of forming a join sequence, one physical window is chosen to be the next join candidate if the generated join results can be consumed by the most queries. The process of creating a join graph has two steps including *create_Join_Sequences* as described in Algorithm 1 and supplement join sequences for *special queries* containing multiple stream patterns referring to the same physical window.

Each Shared Window Buffer (SWB) from the involved queries triggers Algorithm 1 to create the join graph. The *curr_Join_Vertex* parameter for the first call is the vertex containing the SWB itself and SWB's graph patterns. The *coverage* variable in line 1, initialized by 0, holding the number of queries has been considered, is used to escape the algorithm. Line 2 initializes the *HAQ* and *considering_Patterns* variables. HAQ is intended to hold the *H*ighest *A*mount of *Q*ueries reusing the generated join results. The *considering_Patterns* keeps the considering stream patterns in the queries sharing join variables with the current join vertex. From line 3 to line 8, the algorithm iterates over all of the involved physical windows and calculates the *reutilization metric* detailed in the next section. The HAQ and *considering_Patterns* variables are updated when we found a higher calculated value. If the condition in line 6 evaluates to *true* (i.e. the next join candidate is found), we reset the pointers for the current vertex and previous vertexes based on the newly found candidate as from line 12 to 16. With the chosen join candidate, we count the number of covered queries and update the coverage variable (line 17). At line 18, we recursively call the algorithm with the input parameter as the found join candidate. Line 19 assigns the previous state of the considering join candidate to create other sequences (if any). Finally, the algorithm terminates in line 23 if we are not able to find any join candidate, or all the queries have been already considered, i.e. *coverage* equals or is greater than *number_of_queries*.

Calculating the Reutilization Metric: Given n queries (Q_1,..,Q_n), m join sequences (S_1, .., S_m) that can share the join results for a set of sub-queries in the current vertex C, t patterns (P_1, .., P_t) referring to the physical window in C. We define $J^i_j(k, l, v)$ as a counter for calculating the reutilization for variable v, on sequence S_i and pattern P_j, where $v \in V = \{$set of variables in all queries$\}$, $1 \leq i \leq m, 1 \leq j \leq t$, $l =$ index of join variable in P_j, e.g., in the first triple pattern in PW1 (Fig. 2), the index of variable (?person_q1) and (?loc_q1) are 1 and 3, respectively[2]. $k =$ index of the joining variable in S_i, $J^i_j(k, l, v)$ is given in the equation below:

[2] The index of join variable in join sequence is similarly defined.

$$J_j^i(k,l,v)\begin{cases} = 1 \text{ , if } P_j \notin S_i \wedge S_i \in Q_y \wedge p_j \in Q_y \\ \qquad \wedge var(S_i,k) = v \wedge var(P_j,l) = v \\ = 0 \text{ , if otherwise.} \end{cases} \tag{3}$$

The functions $var(S_i, k)$ and $var(P_j, l)$ allow to retrieve the variable of a sequence S_i at index k or the variable of a pattern P_j at index l. We calculate the metric based on the Eq. 4:

$$R = Max(\sum_{v \in V} \sum_{k \in [1,|S_i|]} \sum_{l \in [1,|P_j|]} J_j^i(k,l,v)), \tag{4}$$

where $|S_i|$ and $|P_j|$ gives the size of S_i and P_j, respectively.

Supplement Join Sequences: Special queries mentioned in Sect. 2.2 requires more than one join sequences to generate the join results because any arrived tuple in the shared window buffer is matched with multiple stream patterns inside one query. As Algorithm 1 only considers one join sequence for each query, we need to supplement the rest of join sequences. To do this, we count the number of special queries and rerun Algorithm 1 with the condition that created join sequences are not taken into account.

Join Graph Example: Now we show an example for building the join graph for 4 queries: Q1, Q2, Q3 and Q4[3] with 5 data buffers shown in Fig. 2. In this example, we need to build the join graph for the physical window PW1 because this one stores the streaming data. Figure 3 demonstrates how the graph is constructed. Starting from physical window PW1, the algorithm calculates the highest reutilization metric and chooses the corresponding physical window. As a result, PW2 is chosen with a reutilization value of 3, the highest value among all candidates. There are three shared join variables in three join sub-sequences. In this case, the join results are reused by three queries, which are Q1, Q3 and Q4. Continuing on this path, PW3 is visited next because it has highest reutilization of 2, and the join results on this path are re-used by Q1 and Q3. At this point, all patterns in Q1 are visited and Q1 is covered. After this buffer, PW1, PW4 and PW4 (again) are respectively chosen and Q3 is also covered. Similar processes are repeated to cover all involved queries, then the algorithm stops. Q2, Q3 and Q4 are covered more than once as shown in the dashed branches in Fig. 3. These three special queries require multiple join sequences to generate enough join results. Therefore, we have to run the supplementation step to create join sequences starting from these patterns to make sure, that we do not miss the generated join results. In Fig. 3, the red color indicates the queries covered after running Algorithm 1, while the blue color demonstrates the queries covered after the *supplement join sequence* step.

[3] Q4 in this paper is actually Q5 in [20], we do not use Q4 in [20] because its join sequence is too long for the demonstration.

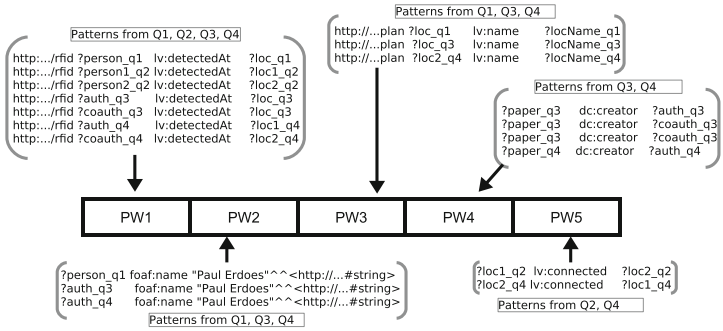


Fig. 2. Shared window buffers for patterns extracted from 4 queries

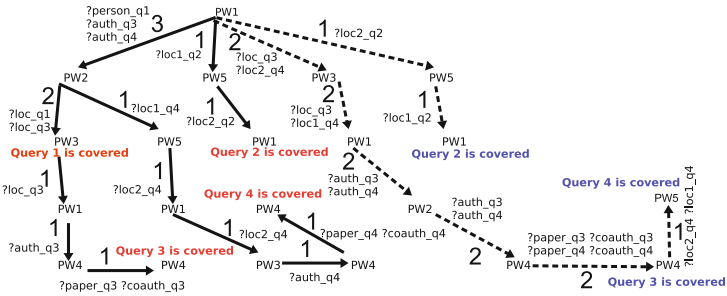


Fig. 3. Join graph example (Color figure online)

3.2 Load Balancing for Parallel CQELS+ Instances

In Sect. 2.2 we discussed the means for optimizing RSP performance leveraging shared join operations. However, the described approach is still a centralized one, i.e., the join graphs and all queries are managed by a single machine. While this is feasible for small-scale applications, it cannot satisfy the need for large-scale systems, where multiple servers are deployed, possibly at different geographical locations, to handle the excessively high concurrency. Evidence for the concurrency limitations for RSP can be found in CityBench [1]. Notice that in CityBench only duplicated queries are tested over a single engine instance. To cope with large-scale applications, a distributed way of processing RDF streams is necessary. In [9,11], a service-oriented approach was proposed to realize RSP federation via service composition. This way, multiple engine instances, even with different engine types, can be deployed in parallel to collaboratively answer an RSP query. In this paper, we follow this principle and develop means for an efficient distributed RSP.

In [10], the problem of multi-modal QoS optimization for the event service composition plan is studied, i.e., it provides means for creating optimal RSP query federation. However, it does not address the problem of the overall system performance. In particular, it does not provide heuristics on which specific engine

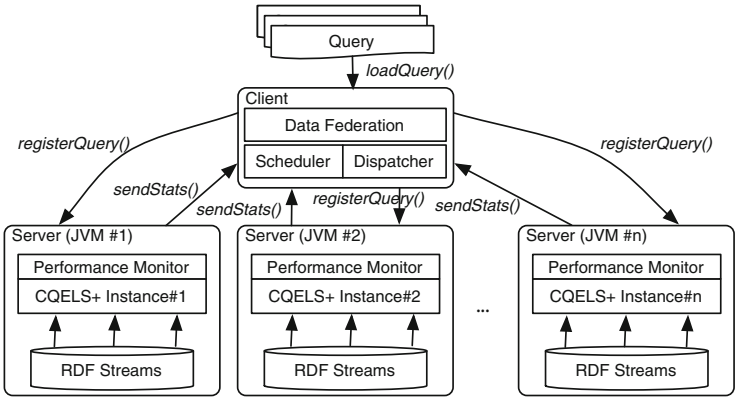


Fig. 4. Architecture for load balancing over parallel CQELS+

instance should be used when a new federated query is generated. To determine the workload assigned to multiple RSP engine instances, we implemented a query scheduler for RSP engines, as illustrated in Fig. 4.

The architecture shown in Fig. 4 consists of a client and multiple servers hosting RSP engines (CQELS+ instances in this paper). The client is a centralized controlling component, which accepts RSP queries from applications and utilizes the data federation component (described in [9]) to create the query federation plan. Then, the scheduler determines which engine instance should be picked for evaluating the plan (which is also a CQELS query) at runtime. The scheduler communicates with the performance monitors implemented on the servers and continuously receives information on the status of the servers. Currently, we implemented three different heuristics (evaluated in Sect. 4.2) for the load balancing strategy:

- **Rotation:** each instance takes its turn to deploy a new query in a rotation, i.e., we distribute queries evenly among the engine instances;
- **Minimal average latency:** the engine instance with the lowest average query latency is picked to deploy the new query. For the latency monitoring mechanism, we follow the approach provided by CityBench [1], and
- **Minimal data buffer size:** this strategy chooses the current instance which has the minimum total number of elements in data buffers, to register the next query, as the processing time of the join operator depends significantly on the join selectivity of the data buffers. Heuristically, smaller data buffers typically have smaller join selectivity.

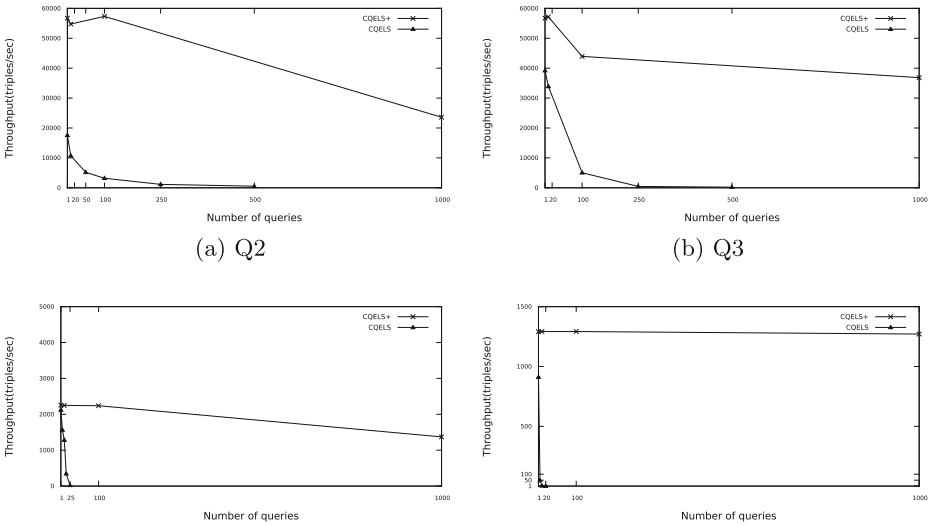
4 Evaluation

In this section, we conduct three experiments. First we compare our shared join approach in the CQELS+ engine with the original CQELS. Then, we show the

performance of the load balancing over CQELS+ engines. Finally, we evaluate the query registration time for multiple queries. All experiments are deployed on a machine running Ubuntu 12.04 with Intel Quad-Core i7-3520M CPU (2.90 GHz) and 16 GB RAM. The tests are compiled for 64-bit Java Virtual Machine (JVM build 1.7.0_80b15). All experimentation results are reproducible[4]. In the following, we present the detailed design of the experiments and analyze the results.

4.1 Evaluating Shared Joins in CQELS+

In this experiment, we reproduce the experiments in LS-Bench[5] and compare the performance between the publicly available version of CQELS[6] with our implementation of CQELS+. The performance is measured in throughput, i.e., the number of triples processed per unit time. We choose 4 queries: Q2, Q3, Q5 and Q6 from LS-Bench for this evaluation, since they involve the streaming and static data. For each query, we increase the number of duplicated queries registered to the engines.



(a) Q2 (b) Q3

(c) Q5 (d) Q6

Fig. 5. Throughput comparison between CQELS+ and CQELS

Figure 5 shows the results of this experiment. CQELS+ outperforms CQELS in throughput as well as in handling concurrent queries. The results in Fig. 5a and b

[4] CQELS+: https://github.com/chanlevan/CqelsplusLoadBalancing.
[5] LS-Bench: https://code.google.com/archive/p/lsbench/.
[6] CQELS engine: https://code.google.com/archive/p/lsbench/wikis/howto_cqels.wiki.

point out that for Q2 and Q3, CQELS is not able to process more than 500 queries and the throughput is about 20000 and 40000 triples per second, respectively. As indicated in Fig. 5c and d, CQELS responses if the number of queries is not higher than 25 and the maximum throughput is about 2200 triples per second for Q5 and 800 for Q6. The results show that the throughput of CQELS decreases drastically when we increase the number of concurrent queries. Conversely, CQELS+ can handle one thousand queries with higher throughput, about 60000 triples per second for Q2 and Q3, nearly 2300 for Q5 and 1300 for Q5. The performance of CQELS+ reduces slowly when we increase the number of queries.

These results are because of two main factors: the join operator and the scheduling mechanism of the JVM. In the CQELS version we evaluated, the join operators are repeatedly triggered by the arrivals of data elements and the consecutive stream windows and data arriving at those windows are processed independently. Furthermore, if the input data coming to stream windows has the high join selectivity and a large number of involving stream windows, a lot of intermediate results are generated. The resources used by those intermediate results have to be released frequently, which causes considerable overhead for the JVM. All of these factors have limited the capability of CQELS for handling concurrent queries. CQELS+ instead, on the same data input (same join selectivity and involved stream windows), manipulates the join strategy by using the incremental evaluation over the network of shared join operators. This strategy only processes the changes of data in stream windows rather than processing consecutive stream windows independently. When multiple continuous queries are registered inside the engine, CQELS+ shares the processing for queries whenever possible. Notably in Fig. 5c, the throughput of two engines are almost equal when they evaluate one query. This is because the window size in the query is too long (1 day) and contains very big data buffers, which makes it difficult for CQELS+ to look for compatible join elements.

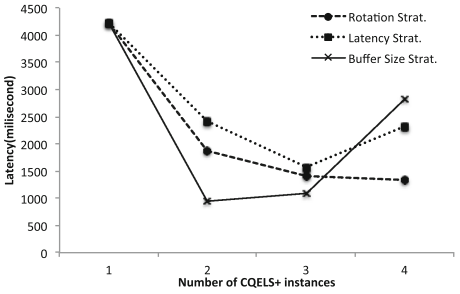


Fig. 6. Latency with increasing CQELS+ instances

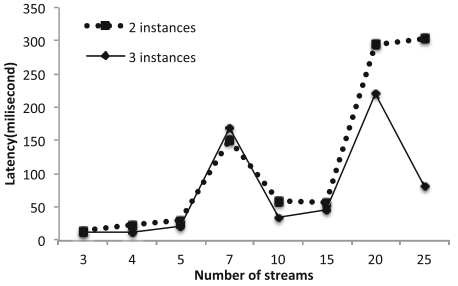


Fig. 7. Latency with increasing number of streams

4.2 Evaluating Load Balancing over CQELS+

In this experiment, we run two different tests. We show the results of the load balancing model presented in Sect. 3.2 in the first test. Then, we choose the best policy and test the scalability of the system with increasing number of streams. To demonstrate the both tests, we implement the client-server architecture as shown in Fig. 4. A new query is registered only after the previous query has been successfully registered to the engine. After all the queries are deployed, we keep the system running for 15 min and monitor the latency. For the input data, we use the CityBench [1] as they support end-to-end query latency monitoring. The latency is captured as follows: each query in this test has some joins over sensor data streams, and the queries are slightly adjusted (e.g., removing numerical filters) so that each sensor observation (in the form of a group of triples) can produce at least one result, then, the latency can be derived by comparing the timestamp of the first result produced by each sensor observation and the timestamp of observation entering the triple streams. Notice that all sensor streams produce observations simultaneously with a same frequency.

In the first test, we deploy different CQELS+ engines in different JVMs and apply the aforementioned load balancing strategies, i.e. rotation, minimum average latency and total buffers size. We choose 4 different queries: Q1, Q2, Q3 and Q4 (from CityBench). We register 400 queries (randomly picked from Q1 to Q4) to the system. Figure 6 illustrates the results. With a single instance, the latency for 400 queries is up to 4.5 s. The latency decreases about 4 times when we deploy 2 and 3 CQELS+ instances. With this configuration, the buffer size load-balancing strategy is the most efficient one, while the other two strategies also improve the latency for 2 and 3 instances. The reason behind this is perhaps that the number of data buffer size is more accurate in representing the workload, and the latency observed may fluctuate in time. When we use 4 instances, the latency tends to increase. This indicates that the overhead of multiple JVMs and CQELS+ instances starts to outweigh the benefit of load balancing.

The second test aims to check the latency of the multiple CQELS+ instances when we vary the number of streams. Previous experiments (including the LSBench results) tested only a limited number of streams. Now we increase the number of streams and monitor the latency over 2 and 3 CQELS+ instances with the data buffer size policy. These streams are picked in turns by 400 randomly generated different queries with Q1 as a template. Figure 7 shows the experiment results. Generally, the latency increases when more streams are used, and with more than 7 streams some unstable processing states can be observed (e.g., latency "spikes" for 7 and 20+ streams). We also observed that with more than 30 streams, the engine often stops responding. The latency increases when we scale the number of streams from 3 to 25 streams in the both configurations (2 and 3 instances). The more streams are involved, the more concurrent threads are created and invoked to stream the data into the system. This makes the system response slower when the number of streams increases. However, the abnormal increase in latency appears with the 7 streams and 25 streams in Fig. 7.

This abnormal behavior is perhaps caused by an incorrect engine implementation, which may also be the cause for the system failure when the number of streams is higher than 30.

4.3 Evaluating the Query Registration Time

In the first experiment, we showed that the CQELS+ outperforms CQELS at runtime. However, this is because we build the join probing graphs before query execution. This introduces an overhead for registering new queries: the graph must be updated constantly. Also, the more queries are deployed, the more time it takes to update the graph. Figure 8 shows the time taken for the query registration using 400 random queries from the template of Q1 in the second experiment, using different load balancing strategies. From the results, we can see that for the data buffer strategy, it takes more time than the other two, possibly because this strategy has a higher chance of resulting in the different number of queries deployed on the engine instances.

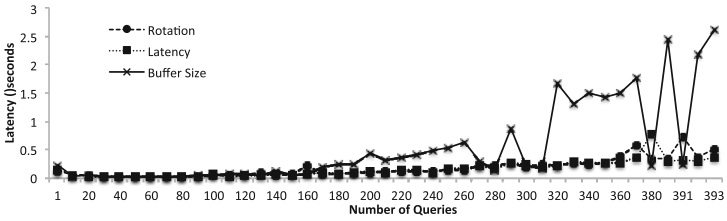


Fig. 8. Query registration time for 4 CQELS+ instances

5 Related Work

Sharing query results is not a novel for database community, e.g. optimization for multiple queries for static database systems by sharing the query operations have been discussed in [7,8,24]. However, for stream processing systems more complicated query semantics are required to be incorporated and additionally the dynamicity of data buffers and input streams have to be considered. Existing RSP engines like C-SPARQL [4] uses a black box approach for handling the semantic queries, thus not able to provide much optimization for multiple queries. Benchmark results like CityBench [1] also showed that with a native join operator implementation, CQELS is better than C-SPARQL at handling multiple queries. Although an adaptive join routing is discussed in [16], where an estimated cost for the join sequences is calculated by monitoring the index of data buffers at run-time. However, it is currently not implemented in CQELS, thus we are not able to compare the performance difference to our approach. Also, we argue that a static join graph has less overhead at run-time while we acknowledge that we are relocating the complexity to query registration time.

Distributed query processing has long been discussed in database community [15] and also in linked data community [2,23]. Regarding distributed RSP, In [13] a C-SPARQL version with parallel streaming is developed mainly to optimize RDFS reasoning by splitting and filtering sub-streams. In CQELS Cloud [17], where extensions for CQELS are made for utilizing the elasticity of cloud environments and allow processing nodes to join and leave the network on-demand. However, the load shedding in CQELS Cloud relies on existing DSMS systems (e.g., Storm[7]), whereas in our approach different strategies are designed and tested. Load balancing techniques have been studied extensively in the literature [14,18,21]. Various metrics, from basic execution latency and bandwidth usage [14] to sophisticated service correlations [21] and network path analysis [18] have been proposed to evaluate the load and determine the shedding strategy.

6 Conclusions and Future Work

In this paper, we realized shared join operations for CQELS in order to improve its performance when handling multiple concurrent queries. Particularly, we have discussed when and how stream and static inputs can share the data. We provided a solution to share join operators. Our approach pre-processes the queries and builds a join graph before constructing the network of shared join operators. The join graph is constructed based on the heuristics that each vertex should generate join results reusable by as many queries as possible. Our experiments showed that CQELS+ can handle more concurrent queries with higher throughputs, compared to the original CQELS. In order to further improve the performance for concurrent RSP queries, we followed the principle of RSP query federation and applied load balancing strategies over distributed RSP engines. Evaluations for the load balancing strategies showed that deploying multiple CQELS+ instances can lower the latency, but the memory overhead for too many parallel instances may outweigh its benefit. Also, we found that while the minimal data buffer size strategy performs best at reducing the query latency, it has longer query registration time.

In future work, we plan to investigate on implementing the probing graph based on both the statistics of data in window buffers and join variable. Real-time data from the stream sources come to the system unpredictably, which means the data inside the window buffers is changing and consequently changes the join selectivity. Therefore, the join probing graph built based merely on join variable in patterns is not able to guarantee the optimal join probing sequences. On the other hand, we must consider the overhead when monitoring the dynamics of the data buffers. We also plan to investigate more sophisticated load balancing strategies, e.g., defining a more precise cost model for the query processing pipeline and use it to estimate the total cost after a new query registration. This may involve checking the similarity between queries using metrics like the

[7] Twitter Storm: http://storm.apache.org/.

number of shareable data buffers, the graph edit distance between the join probing graphs before and after the query registration etc.

Acknowledgment. Authors are extremely thankful to John Breslin, Alessandra Mileo and Danh-Le Phouc for their valuable feedback and guidance. The work conducted during this study is supported by Science Foundation Ireland (SFI) under grant No. SFI/12/RC/2289.

References

1. Ali, M.I., Gao, F., Mileo, A.: CityBench: a configurable benchmark to evaluate RSP engines using smart city datasets. In: Arenas, M., et al. (eds.) ISWC 2015. LNCS, vol. 9367, pp. 374–389. Springer, Cham (2015). doi:10.1007/978-3-319-25010-6_25
2. Ali, M.I., Pichler, R., Truong, H.-L., Dustdar, S.: DeXIN: an extensible framework for distributed XQuery over heterogeneous data sources. In: Filipe, J., Cordeiro, J. (eds.) ICEIS 2009. LNBIP, vol. 24, pp. 172–183. Springer, Heidelberg (2009). doi:10.1007/978-3-642-01347-8_15
3. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. VLDB J. **15**(2), 121–142 (2006)
4. Barbieri, D.F., Braga, D., Ceri, S., Della Valle, S.E., Grossniklaus, M.: C-SPARQL: SPARQL for continuous querying. In: Proceedings of WWW, pp. 1061–1062. ACM (2009)
5. Bizer, C., Heath, T., Berners-lee, T.: Linked data - the story so far. Int. J. Semant. Web Inf. Syst. **5**, 1–22 (2009)
6. Calbimonte, J.-P., Jeung, H., Corcho, O., Aberer, K.: Enabling query technologies for the semantic sensor web. Proc. IJSWIS **8**(1), 43–63 (2012)
7. Deen, S.M., Al-Qasem, M.: A query subsumption technique. In: Bench-Capon, T.J.M., Soda, G., Tjoa, A.M. (eds.) DEXA 1999. LNCS, vol. 1677, pp. 362–371. Springer, Heidelberg (1999). doi:10.1007/3-540-48309-8_34
8. Diao, Y., Franklin, M.J.: High-performance XML filtering: an overview of YFilter. IEEE Data Eng. Bull. **26**, 41–48 (2003)
9. Gao, F., Ali, M.I., Mileo, A.: Semantic discovery and integration of urban data streams. In: Proceedings of the 13th International Semantic Web Conference (ISWC 2014), Workshop on Semantics for Smarter Cities (2014)
10. Gao, F., Curry, E., Ali, M.I., Bhiri, S., Mileo, A.: QoS-aware complex event service composition and optimization using genetic algorithms. In: Franch, X., Ghose, A.K., Lewis, G.A., Bhiri, S. (eds.) ICSOC 2014. LNCS, vol. 8831, pp. 386–393. Springer, Heidelberg (2014). doi:10.1007/978-3-662-45391-9_28
11. Gao, F., Curry, E., Bhiri, S.: Complex event service provision and composition based on event pattern matchmaking. In: Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems, Mumbai, India. ACM (2014)
12. Hammad, M.A., Franklin, M.J., Aref, W.G., Elmagarmid, A.K.: Scheduling for shared window joins over data streams. In: VLDB. VLDB Endowment (2003)
13. Hoeksema, J., Kotoulas, S.: High-performance distributed stream reasoning using S4. In: Ordering Workshop at ISWC (2011)
14. Koerner, M., Kao, O.: Multiple service load-balancing with openflow. In: 2012 IEEE 13th International Conference on High Performance Switching and Routing (HPSR), pp. 210–214. IEEE (2012)

15. Kossmann, D.: The state of the art in distributed query processing. ACM Comput. Surv. (CSUR) **32**(4), 422–469 (2000)
16. Le-Phuoc, D.: A native and adaptive approach for linked stream data processing. Ph.D. thesis, National University of Ireland Galway, IDA Business Park, Lower Dangan, Galway, Ireland (2012)
17. Le-Phuoc, D., Nguyen Mau Quoc, H., Le Van, C., Hauswirth, M.: Elastic and scalable processing of linked stream data in the cloud. In: Alani, H., et al. (eds.) ISWC 2013. LNCS, vol. 8218, pp. 280–297. Springer, Heidelberg (2013). doi:10.1007/978-3-642-41335-3_18
18. Matsuba, H., Joshi, K., Hiltunen, M., Schlichting, R.: Airfoil: a topology aware distributed load balancing service. In: 2015 IEEE 8th International Conference on Cloud Computing (CLOUD), pp. 325–332. IEEE (2015)
19. Naughton, V.J.F., Burger, J.: Maximizing the output rate of multi-way join queries over streaming information sources. In: VLDB. VLDB Endowment (2003)
20. Le-Phuoc, D., Xavier Parreira, J., Hauswirth, M.: Linked stream data processing. In: Eiter, T., Krennwallner, T. (eds.) Reasoning Web 2012. LNCS, vol. 7487, pp. 245–289. Springer, Heidelberg (2012). doi:10.1007/978-3-642-33158-9_7
21. Porter, G., Katz, R.H.: Effective web service load balancing through statistical monitoring. Commun. ACM **49**(3), 48–54 (2006)
22. Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF. W3C Recommendation **4**, 1–106 (2008)
23. Schwarte, A., Haase, P., Hose, K., Schenkel, R., Schmidt, M.: FedX: optimization techniques for federated query processing on linked data. In: Aroyo, L., et al. (eds.) ISWC 2011. LNCS, vol. 7031, pp. 601–616. Springer, Heidelberg (2011). doi:10.1007/978-3-642-25073-6_38
24. Sellis, T.K.: Multiple-query optimization. ACM Trans. Database Syst. **13**(1), 23–52 (1988)
25. Sequeda, J.F., Corcho, O.: Linked stream data: a position paper. In: SSN (2009)