A Unified Interface for Optimizing Continuous Query in Heterogeneous RDF Stream Processing Systems

Seungjun Yoon, Sejin Chun, Xiongnan Jin, Kyong-Ho Lee

Department of Computer Science, Yonsei University, Republic of Korea
{sjyoon, sjchun, wnkim}@icl.yonsei.ac.kr, khlee89@yonsei.ac.kr

Abstract. The W3C RDF Stream Processing (RSP) community has proposed a common model and a language for querying RDF streams. However, the current RSP systems significantly differ from each other in terms of performance. In this paper, we propose a unified interface for optimizing a continuous query in heterogeneous RSP systems. To enhance the performance of RSP, a unified interface decomposes a query, reassembles partial queries and assigns them to appropriate RSP systems. Experimental results show that the proposed approach performs better in terms of memory consumption and latency.

Keywords: RDF stream processing, Unified query interface, RSP system

1 Introduction

The importance of RDF Stream Processing (RSP) is magnified by the extension of Linked Open Data (LOD) [6]. Since RSP systems have been proposed, we are able to integrate data sources and process a continuous query using the RSP systems. In order to provide a coherent semantic model for various RSP systems [1,2], the RDF Stream Processing Query Language (RSP-QL) was proposed by [3]. We face various features of RDF streams, e.g., the number of streams and the execution time of a stream, when processing a continuous query using heterogeneous RSP systems. The performance of existing RSP systems varies in terms of memory consumption and latency. We have to evaluate them to select an optimal RSP system that processes with the smallest memory and lowest latency.

We propose a unified interface for processing a continuous query in three steps: Firstly, we analyze a global query [4] using the query form (e.g. SELECT etc.) that is defined by the SPARQL syntax. After analysis, the global query is divided into partial queries according to the streams contained in the global query. Secondly, each partial query is continuously evaluated based on stream features to select an optimal RSP system, for which a partial query is registered. Thirdly, each RSP system returns an RDF graph as the answer to a partial query. However, each partial RDF graph is incomplete to answer the global query since it has only partial information. So we integrate partial RDF graphs into a global RDF graph. Furthermore, we demonstrate that the proposed method has advantages in comparison with C-SPARQL [2] and CQELS [1] in terms of the growth rate of memory consumption and latency.

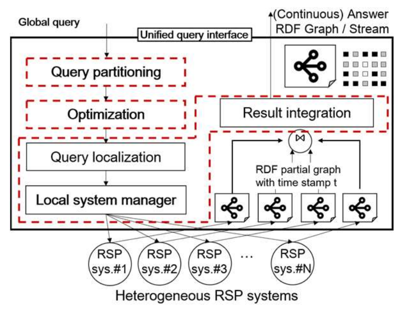

Fig. 1. Proposed Architecture

```
Data: Global query Q
Result : a set of partial queries F
ℙ = constructQueryplan(Q)
nextOp := null
while nextOp is not Root
 nextOp := ℙ.nextOp()
 if nextOp is named window w then
  Stream S = F.serachRelevantStream(nextOp)
  F.addPartialQuery(nextOP, S)
 end if
end while
return F
```

Algorithm 1. Query partitioning

2 Our Approach

Fig. 1 shows the overall process of the proposed unified interface of optimizing a global query, given by a user for heterogeneous RSP systems. The process consists of five stages: query partitioning, optimization, query localization, local system management and result integration. Our main contributions lie on the three cores, including query partitioning, optimization and local system management. In the following, the details of the three core stages are described.

Query partitioning. To assign a query to a suitable RSP system that has better performance than the other RSP systems, we divide a global query into partial queries. As shown in Algorithm 1, by dividing a given global query, our method returns a set of partial queries that should be delivered to appropriate RSP systems. Specifically, the tree of a query plan is constructed using the semantics of a given query. Each node of the tree contains operators (e.g., window, join etc.), which perform respective roles. And, each node finds a stream corresponding to an operator as an input parameter. Since a partial query must include a given window operator and a stream, thus we repeatedly constitute a pair of a window and a stream. Then a set of partial queries is returned to be registered to the RSP systems.

Optimization. In this stage, partial queries are optimized based on the stream features to be allocated to appropriate RSP systems. Specifically, to check whether a partial query is suitable to be processed on a specific RSP system or not, we obtain a set of RSP systems, which support a certain operator. We also use the performance history (e.g. memory consumption) to select an RSP system that can process the partial query most efficiently in accordance with the execution time. The number of streams is not considered by this stage because the partial query generated in the stage of *Query partitioning* only has a single stream. Furthermore, we classify RSP systems into four classes for registering a partial query to an optimal RSP system in accordance with two factors: *the initial memory consumption* and *the increasing rate of memory consumption*. If all of the factors are high, the corresponding RSP system is not selected because the performance is always the worst. In contrast, the RSP system with the lower values is an ideal choice. Generally, if one of the factors is high, then the other one is low.

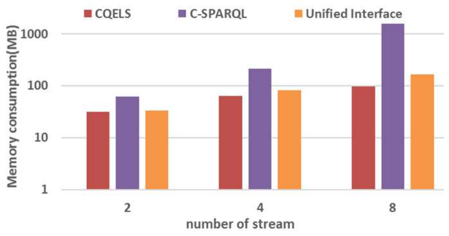

Fig. 2. Memory consumption with varying number of streams

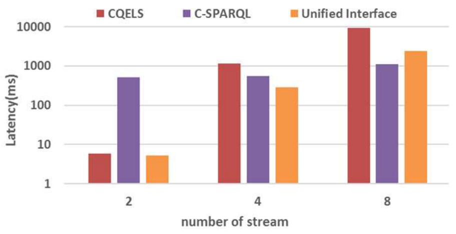

Fig. 3. Latency with varying number of streams

Moreover, it is reasonable to register a partial query with short execution time to a system, which has the high increasing rate and the low initial cost of memory consumption. By using the classification, we determine the type of the systems with the optimal performance. We define a set of timestamps $\mathrm{T} = \{t_1, t_2, \dots, t_i, \dots, t_n\}$, where t_i indicates the time when two RSP systems consume the same amount of memory. Therefore, we try to derive the optimal performance to assign a partial query to the selected RSP system based on t_i.

Local system manager. This stage includes the query localization and the result integration. In the previous step, we determine how to distribute partial queries. To deal with the partial queries in the given RSP system, the proposed method supports the translation into the language form of the given RSP system. Note that, since the RSP systems may not require translation, the query localization is optional. And then, we distribute each partial query into an appropriate RSP system. Besides, to answer a global query, we integrate the graphs generated from partial queries into a graph. The integrated graph provides the answer to the global query.

3 Experimental Result

To verify our approach, we evaluated the performance of the proposed unified interface in comparison with C-SPARQL and CQELS by using multiple stream queries. The performance was evaluated in terms of memory consumption and latency. The latency indicates the time of delay until our method provides an answer after query execution.

Experimental Setting. Our experiments were conducted with Intel Core i7-4790k 4.0GHz CPU, 8 GB RAM, 250 GB SSD, Windows 10 OS and Eclipse for Java. Also we used the real time cities dataset provided by CityBench [5]. We experimented by varying the number of streams for global queries Query5 and Query10 of CityBench.

Result Analysis. As shown in Fig. 2 and Fig. 3, we compared the proposed interface with CQELS and C-SPARQL in terms of the number of streams. The results showed that our method consumed the latency approximately 85% less, but the memory approximately 40% more than CQELS. And, our method consumed the memory 80% less, but the latency approximately 95% more than C-SPARQL. In this respect, we observed that our proposed method outperformed C-SPARQL in terms of memory consumption and CQELS in terms of latency. The reason for this is that partial queries were distributed by the evaluated performance of systems. Also, each RSP system

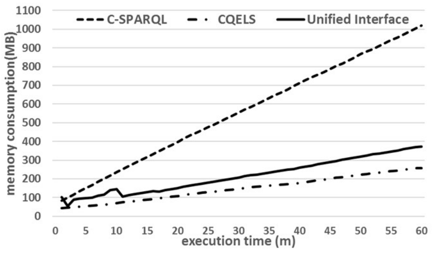

Fig. 4. Memory consumption while increasing execution time

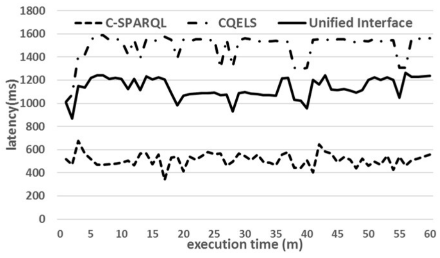

Fig. 5. Latency while increasing execution time

processed the smaller number of streams as compared to the number of streams of a global query and reduced the load for processing streams.

We measured the performance using queries consisting of five streams according to the execution time of streams as shown in Fig. 4 and Fig. 5. A unified interface required the increased memory consumption during the initial one minute. It is due to that a unified interface needs the processing steps of query writing and query registering. After some time, the experimental results were optimized in other parts, respectively. The proposed method was better than C-SPARQL as much as 80% in terms of memory consumption. Also, the proposed method was better than CQELS as much as 60% in terms of latency. This is because a partial query was assigned to an optimal RSP system.

In conclusion, the conventional RSP systems have to trade-off between memory consumption and latency. The proposed method showed stable performance between the memory consumption and the latency without being biased to one side.

Acknowledgement

This work was supported by the ICT R&D program of MSIP/IITP, Republic of Korea. [B0101-16-1276, Access Network Control Techniques for Various IoT Services]

References

1. Le-Phuoc, Danh, et al.: A native and adaptive approach for unified processing of linked streams and linked data. In: Web–ISWC, 2011. p. 370-388.
2. BARBIERI, Davide Francesco, et al.: C-SPARQL: SPARQL for continuous querying. In: WWW. ACM, 2009, p. 1061-1062.
3. Dell'Aglio, D., Calbimonte, J. P., Della Valle, E., & Corcho, O.: Towards a Unified Language for RDF Stream Query Processing. In: ESWC, 2015, p. 353-363.
4. Daum, Michael.: Deployment of Global Queries in Distributed and Heterogeneous Stream Processing Systems. In.: DEBS Workshop, 2009.
5. Ali, M. I., Gao, F., & Mileo, A..: CityBench: a configurable benchmark to evaluate RSP engines using smart city datasets. In. ISWC, 2015. p. 374-389.
6. Sejin Chun, Seungmin Seo, Wonwoo Ro, and Kyong-Ho Lee, "Proactive Plan-Based Continuous Query Processing over Diverse SPARQL Endpoints," Proc. of the IEEE/WIC/ACM Web Intelligence conference (WI 2015) , pp.161-164, 2015.