Query Rewriting or Ontology Modification? Toward a Faster Approximate Reasoning on LOD Endpoints

Naoki YAMADA†∗a), Yuji YAMAGATA†∗∗b), Nonmembers, and Naoki FUKUTA††††c), Member

SUMMARY On an inference-enabled Linked Open Data (LOD) endpoint, usually a query execution takes longer than on an LOD endpoint without inference engine due to its processing of reasoning. Although there are two separate kind of approaches, query modification approaches, and ontology modifications have been investigated on the different contexts, there have been discussions about how they can be chosen or combined for various settings. In this paper, for reducing query execution time on an inference-enabled LOD endpoint, we compare these two promising methods: query rewriting and ontology modification, as well as trying to combine them into a cluster of such systems. We employ an evolutionary approach to make such rewriting and modification of queries and ontologies based on the past-processed queries and their results. We show how those two approaches work well on implementing an inference-enabled LOD endpoint by a cluster of SPARQL endpoints.

key words: semantic web, SPARQL, inference

1. Introduction

Linked Open Data (LOD) are retrieved by a query written in the standard query language called SPARQL*** for the retrieval of RDF data stored in an endpoint. Reasoning on LODs allows such queries to obtain implicitly stated knowledge or data from the given explicit relations [1]. These techniques are important to make intelligent agents accessible to data on its external world. Techniques to utilize reasoning capability based on ontology have been developed to overcome several issues, such as higher complexity in worst case [2]–[6].

On a retrieval of LOD using SPARQL, a client prepares a SPARQL query and directly submits the query to a SPARQL endpoint. Even when the query prepared by the client might need a long time for its execution, a standard SPARQL endpoint implementation will try to execute the query with lots of costs to return answers unless it reaches the limit for its timeout. If the endpoint receives lots of heavy queries and the specified timeout is too long, it might cause a server-down. This is especially important for endpoints that have inference engines to support OWL reasoning capability [7].

In order to avoid such executions of time-consuming queries at the endpoint’s side, it might be valuable to classify whether a query execution is time-consuming or not [8], [9]. An evolutionary approach could be useful for executing such queries approximately, while it may not return the exactly same result [10]. However, there is a limitation to obtain both fast execution and correct answers from a single SPARQL approximate processing engine [7]. Although some ontology-modification approaches (e.g., [11]) could also be applied to this issue, the limitation of approximation on a single SPARQL processing engine still remains.

In this paper, a mixed approximation approach for processing inference-enabled SPARQL queries is presented****. In the approach, to avoid variance of approximation accuracy and its runtime performance, a query is processed on a clustered inference-enabled SPARQL processing engines whose approximation profiles are slightly different. The results are aggregated for better performance as well as better correctness and coverage of the result. To implement each SPARQL processing engine in a cluster, we employ an evolutionary approach to make better approximation rules to be applied on SPARQL processing phase as well as their ontology modification phase. Also we utilize ensemble learning techniques to avoid poor performance scenarios.

We show how this approach can be implemented by presenting a prototype system and discuss about the performance on it. Then, we compare two promising methods: query rewriting and ontology modification for reducing query execution time with sufficient correctness and coverage of results on an inference-enabled LOD endpoint. We employ an evolutionary approach to make such modification of queries or ontologies based on the past-processed queries and their results. We show how those two approaches work well on implementing an inference-enabled LOD endpoint by a cluster of SPARQL endpoints and discuss about the result on it.

2. Preliminaries

2.1 Query Approximation

There are two possible approaches to manage long-running

****http://www.w3.org/TR/rdf-sparql-query/

† † † † This paper is based on our initial idea presented at [12].
queries. One is to utilize parallel and distributed computing techniques to make those executions faster [13]. Another possible approach is to apply approximate execution of a query that requires long execution time to a light-weight one.

A possible approach to approximate an inference processing is just deleting subsets in an ontology that would increase classification complexity [11]. To enable faster inference, a kind of ontology refinements could be applied by ontology engineers themselves. Ontology engineers are not often working on the side of endpoints but rather as independent positions and therefore they might not know how such ontologies were optimized and what kind of inferences could be run on those ontologies. Similar issues are on the use of destructive query optimization (i.e., approximate the result of query but does not guarantee obtaining the exact result).

To solve those issues, we have proposed an architecture and a mechanism to apply such query modifications on the fly [7]. Before summarizing those approaches, we briefly explain how other possible approaches could be applied for those issues. To do that, when we could predict the query execution performance on DL inference-enabled SPARQL endpoints to detect time-consuming queries, it could be realistic to apply some good approximation methods to those heavy queries selectively.

2.2 Query and Inference Performance Prediction

For reducing the time spent for reasoning, developing a faster inference algorithm is a direct way to solve the issues. For example, Baumgartner et al. proposed an efficient reasoning algorithm based on hypertableau [2]. HermiT [3] reasoner implemented such an improved hypertableau reasoning algorithm.

In [4], for the classification task on ontological inference, MORE has been presented that combines an OWL2 reasoner and an external efficient reasoner. MORE has several versions, one is using HermiT with ELK† [14], JFact with ELK, and other experimental versions††. MORE is optimized for classification task on ontologies by combining an extensively applicable reasoner (HermiT or JFact) and a more efficient and profile specific reasoner (ELK).

Although those implementation-level inference speed improvements are very useful, there is another difficult issue: the difficulty of predicting hardness of each inference problem. Kang et al. presented a systematic study to tackle the problem and argued that the hardness of reasoning about individual ontologies has not been easily characterized and there is a challenge of predicting ontology classification performance by using machine learning techniques [5].

In [5], they introduced a number of metrics that can be used to predict reasoning performance and evaluated various classifiers to know how they accurately predict classification time for an ontology based on its metric values. According to their results of evaluation, they have prepared prediction models which can predict in accuracy of over 80%, but there are still major difficulties to improve them.

In [5], they finally argued that the ontology classification is still a challenging task in spite of such progresses in the design and development of optimized algorithms and reasoners, and there are demands to be able to quantitatively analyze and predict reasoning performance using syntactic features.

In [6], it is introduced that reasoning tasks on ontologies constructed from an expressive description logic have a high worst case complexity, by analyzing experimental results that divided each of several ontologies into 4 and 8 random subsets of equal size and measured classification times of those subsets as increments. They reported that some ontologies exhibit non-linear sensitivity on their inference performance.

They also argued that there is no straightforward relationship between the performance of a subset of each isolated ontology and the contribution of each subset to the whole inference performance when they merged into the ontology, while they provided an algorithm that identifies ontology’s hot spots.

Hasan proposed a better prediction method to estimate the performance of each SPARQL query [9]. Hasan’s prediction approach is based on algebra features of a SPARQL query. Before executing a query, it has been decomposed into a graph structure that is called SPARQL algebra expression. In their work, they constructed a feature vector from the algebra expression and then applied machine learning techniques such as SVM. They reported that their prediction performance is nearly 0.94 in $R^2$ coefficient value among their predicted execution times and actual execution times. Also, Sande et al. proposed an approach to extend a metadata-based endpoint selection approach to allow clients to opportunistically choose appropriate endpoints to be searched when there are possible endpoints to be accessed [15].

Now we have a good prediction method for estimating query execution performance, however, their approach does not consider DL reasoning on those endpoints. We have proposed an ensemble learning approach to predict the query execution performance on DL inference-enabled SPARQL endpoints [7].

2.3 Architecture of Front-End Endpoint

In [7], the aim of classifying queries is to classify whether a SPARQL query execution is time-consuming or not based on the policies given by the owner of the endpoint. The classifier is implemented based on machine learning implementations (i.e., Weka [16], in this case). The classifier is placed at the front-end EP [7].

The classifier predicts whether or not the execution of the given query requires a time that is longer than the threshold set in advance. The Front-end EP extracts attributes
as input objects from a query sent from client. The classifier classifies the query as time-consuming or non-time-consuming one from extracted attributes.

The classifier is built based on training data generated from records of queries and their execution time. Here, we prepared an attribute, “whether the query execution is time-consuming or not,” as a desired output value. Other attribute values (e.g., the number of each class URIs and what first appeared class URI is) are extracted from records of queries as input objects.

If the input query seems to be time-consuming, the query will not be executed as is, and a notification will be sent to the client to notify that the query execution has been rejected or the query is going to be rewritten into a cost-friendly query and the client can resubmit the query rewritten by the Front-end EP when the client accepts it. If the input query seems not to be time-consuming, the query will be executed as it is. Figure 1 shows the process of query classification and optimized execution on the Front-end EP. We used Weka\cite{16} for the implementation of machine learning algorithms. On the classifier implemented in the Front-end EP, several learning algorithms such as (e.g., bagged J48, boosted J48, support vector machine, etc.) are available for use, and they can be configured on each Front-end EP. In [8], we have presented this classification approach produced up to approximately 99.9 percent for the applied dataset.

2.4 GA-Based Query Rewriting

There are some query rewriting approaches to improve the quality of queries [17]–[19]. Also, there are some heuristic techniques to approximate inference-enabled queries by modifying some hotspots in the query that prevent faster execution [8]. However, since those hotspots are also dependent on their individual ontologies, such query modification should take into account both query-structure and characteristics of the used ontologies.

It is demanded to obtain a rewriting rule that could be applied to some sort of queries generically, rather than that can only be applied to a particular query. The reason is that when the Front-end EP executes a heavy query on each time to get results for generating query rewriting rules, it makes no difference, or even worse to execute such a query as is. Although it is possible to reduce such an overhead by caching pair of a query and an optimized query, it still makes a cost for optimization when each query is executed at a first time. Furthermore, this approach can not be applied to a case that an original query was too heavy to be executed so that it is difficult to run the optimization process itself. To solve those issues, in [10] and [20] we have prepared a GA-based heuristic query-rewriting-rule generation engine [10], [20].

For generating heuristic query rewriting rules, each individual in the GA-based engine represents a set of possible query modification operators (i.e., rules), and then the engine applies individuals to some test queries. Individuals are evaluated by fitness values obtained by executing the rewritten queries. A ‘then’ part of each query rewriting rule is constructed with heuristic operations such as “change the n-th class appeared in the query to its superclass”, “change the n-th class to its randomly selected subclass”, “swap left and right sides of UNION operator”, and so on. Notice that, those may include some operations that do not guarantee to produce the same result as the original query generated. Therefore, we used the term ‘approximate’ rather than ‘optimize’ for applying those heuristic rules. Furthermore, sometimes some operations may not be applied to a specific query. For example, consider this scenario. At first, we applied an operation “change the n-th class appeared in the query to another class at random” to a specific query, it could be work well some times. However, when there is no “n-th class appeared in the query”, the operation could not be applied.

In [20] we have shown the basic query processing pro-
Therefore, we rather prepare multiple endpoints that store differentially-modified ontologies which will be presented in the next section.

3. Our Approach

3.1 Architecture of Clustered Front-End EndPoints

It is natural for an endpoint to have a policy to limit the maximum cost of execution for each query. If the query execution is classified as too time-consuming one, the endpoint may have an option to reject the execution of the query or transform that query into an optimized one. To implement such behaviors on an endpoint, some extensions should be provided to allow a notification to the client that the received query has been transformed into another one, or the query has been rejected due to a heavy-load condition.

To realize this idea, we initially implemented a prototype system to classify whether a query execution is time-consuming or not, rewriting the query to a more light-weight one, and extending the protocol to notify the rejection of the query, the applied query-transformation for the query, and so on [20]. We initially applied a pattern-based heuristic query rewriting technique that, for example, substitutes some named classes to subsets of their potential classes that are derived by the inference [8]. Our prototype system has a unique proxy module called “Front-end EP” between the client and the endpoint (called “Back-end EP” in this paper). Figure 1 shows a brief overview of the query execution process mediated by a Front-end EP [8], [20].

First, a client prepares a query for retrieving LOD. Then, the query prepared by the client is sent to the Front-end EP instead of its primary Back-end EP. Here, the Front-end EP predicts the cost for query execution. If the Front-end EP judges the received query be a heavy query, the Front-end EP can reject the query or run a modified one based on the specified policies.

When the Front-end EP modifies the submitted heavy query, an approximation process is applied and a modified query is sent to the Back-end EP. Then the Back-end EP executes the received query modified by the Front-end EP and returns the answer to the Front-end EP. After that, the Front-end EP sends the answer received from the Back-end EP to the client. When the query-approximation process has been applied, the modified query may not return the exactly equivalent result to the one from the original query. Here, an issue for the client is how to know the fact that the client’s original query has been rewritten by the approximation process. To realize this mechanism, we re-used some related HTTP response codes to keep consistency in the protocol. However, to work with this extended protocol, a client should support this extension. In this paper, we omit further details about this protocol extension due to make much focus on query rewriting mechanism itself.

The idea presented in [10] suffers an issue due to relatively large variance of the performance of applying the generated query rewriting rules for actual queries, since an endpoint might have very limited number of sample queries to be processed that would also be used for applying the GA-based query rewriting rule generations. Another issue is the limitation of computational resource for generating better rewriting rules. Since the computation for generating better rewriting rules will require more generations of GAs to be computed and sometimes this process produces a certain amount of loads on the Back-end EP for calculating fitness values of generated individuals (i.e., rules), we make a limit for such computation. This causes variance of quality on the generated rewriting rules and sometimes it might have some poorly performed rewriting rules for some queries that were not processed before.

To overcome this issue, we propose a cluster-based approach, i.e., it allows the master Front-end EP to use several Simple Front-end EPs in the middle tier and then aggregates those results based on several heuristic aggregation functions. The overview of this cluster-based approach is shown in Fig. 2*. In this approach, Front-end EPs are cascaded in two tiers (i.e., front tier and the middle tier) and in the middle tier it can employ two or more Front-end EPs with slightly different optimization parameters to generate different approximation results.

This architecture allows some good selections and aggregations on the front tier to choose faster responses from several middle tier Front-end EPs, aggregate two fastest responses for better accuracy or recalls, and some other heuristic aggregations among them. This mechanism reduces the chance to encounter a case to have a very poor approximation performance due to high variance of approximation quality.

Figure 3 shows the overview of our system running on a single computer with virtual machines. In Fig. 3, the window located in the upper-left is an interface to issue an query to the testing system. It also behaves as the top of the cluster to issue the query to other clustered middle-tier EPs. The results obtained from the query are shown in the lower-left window. The three virtual machine windows show how each middle-tier Front-end EP and its associated Back-end EP run on their virtual machines.

3.2 Representing Rewriting Rules Using SWRL

As discussed in [7], it is important to properly tell the clients about how the system applied an approximation on the processing. To represent the query rewriting and ontology modification rules to the clients, our system prepares a representation of the applied rules to the clients. For example, when the applied rule is as explained in (1), the applied rules are represented by the following SWRL [21]-xml coding and it

---

*Initial idea about this approach has been presented at [12].
will be send to the clients when necessary. Here, the predicate “removeAxiom” denotes that these specified axioms will be deleted from the specified ontology, while the predicate “includes” denotes the conditions when these modifications will be applied. Some examples and discussions about modifying ontology for inconsistency resolution and performance optimization have been presented in [24].

\[
\begin{align*}
\text{removeAxiom}(\text{axiom1}, \text{ontology}) \land \\
\text{removeAxiom}(\text{axiom2}, \text{ontology}) \land \\
\text{removeAxiom}(\text{axiom3}, \text{ontology}) & \leftarrow \\
\text{includes}(\text{query}, \text{axiom1}) \land \\
\text{includes}(\text{ontology}, \text{axiom3})
\end{align*}
\]

4. Experimental Analysis

4.1 Experimental Settings

In the experiments, as our initial experiments in [12], we used the dataset used in the conference track on Ontology Alignment Evaluation Initiative 2013 (OAEI 2013). Here we used Linklings ontology from the OAEI dataset in the preliminary experiment. To prepare datasets to evaluate the performance sensitivity of ontology-level simplification techniques, we reduced Linklings ontology by cutting several relational descriptions and added 10 instances for each named classes by using protége (v4.3)\(^6\).

\[^6\text{http://protege.stanford.edu/}\]
The evaluation data set was generated by queries getting the instances of a named class in the Linklings ontology. Each query gets the instances of one named class. We prepared these queries to cover all the named class to be retrieved. We also prepared queries to join instances of two named classes. In these queries, orders of getting instances of each class is taken into account. For example, a query:

```sql
SELECT ?x WHERE {
  {?x <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://linklings#Submission> }
UNION
  {?x <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://linklings#Administrator> }
}
```

is treated as not identical to the following query:

```sql
SELECT ?x WHERE {
  {?x <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://linklings#Submission> }
UNION
  {?x <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://linklings#Administrator> }
}
```

As an experimental environment we set up a SPARQL endpoint using Joseki (v3.4.4) in conjunction with server-side Pellet [22] reasoner to enable OWL-level inference capability on the endpoint.

We used an Intel Core-i7 2.9GHz dual core Mac-BookPro, with 6GB SDRAM dedicated to the Java Virtual Machine to run the endpoint.

4.2 Performance of Cluster-Based Approach

We conducted our experiment in the following settings. We formed a cluster which has three simple Front-end EPs.

Here, we prepared three aggregation scenarios for our cluster-based approach. One is ‘choose fastest response’ to have better response time. Another one is ‘choose the fastest two responses and aggregate results in OR’, i.e., the union of the two results. The last one is almost the same idea of the 2nd one, with AND aggregations instead of OR, for better accuracy but potentially worse recall.

Here, for each Front-end EP, we applied a very conventional Genetic Algorithm (GA) for generating good query rewriting rules as we applied in [10]. We simply encode some heuristic rules, such as, just replace the 1st appeared class to its superclass, and so on, onto each gene. The front-end endpoint assumed to have some time-consuming queries and their results from past query execution logs to be used for its evolutionary computation phase. Then the GA is applied to those data on each middle-tier Front-end EP. Here, we assume that we could have the correct result for calculating the fitness values for the individuals in GA. After generating the heuristic rules by GA, the rules are stored to be applied to the queries that may not return their results due to the timeout of their executions. To make difference of the GAs used in middle-tier Front-end EPs, we forced those Front-end EPs to have different initial individuals as far as possible through the communication among the top tier Front-end EP and other middle-tier Front-end EPs.

Fitness values for individuals [7], [10] are defined as

$$ V_q = \alpha F_q + \beta \frac{T_Q - T_q}{T_Q} $$

such that $\alpha + \beta = 1$

Here, $V_q$ is a fitness value of written query $q$. $F_q$ is a F-measure. $T_Q$ is an execution time of original query $Q$. $T_q$ is an execution time of rewritten query.

In this evaluation, we used 50 individuals for each middle-tier Front-end EPs, set inversion rate 0.2, mutation rate 0.15 (to superclass), 0.15 (to subclass), and 0.05 (random). Average number of generations was 35.3.

We calculated an average F-measure of queries rewritten in accordance with a rewriting rule generated by our GA approach. We also measured average times of query execution about an original query and rewritten queries when they were used separately or used in a clustered scenario. We applied this approach to two known heavy queries that we identified in [10]. Here, to compare the performance among the different approximation approaches, we show an ideal F-measure and query execution time that does not include any overheads within the clustered Front-end EP.

Table 1 shows the results that we conducted for this setting. As we mentioned, those values are ideal values that excluded any communication overheads among the tiers.

According to Table 1, our cluster-based GA approach can find an ideal query rewriting rule in most cases. As the Table 1 shows, there is a possibility that GA approach can find a query rewriting rule generating a query that can execute within shorter time than the original query. Table 2 shows the results when we applied our ontology approximation approach to the same settings used in Table 2. Here, compared to the query approximation approaches, the

<table>
<thead>
<tr>
<th>Table 1</th>
<th>Performance of cluster-based evolutionary query approximation.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original</td>
</tr>
<tr>
<td>Time (ms)</td>
<td>1841.5</td>
</tr>
<tr>
<td>Precision</td>
<td>0.93</td>
</tr>
<tr>
<td>Recall</td>
<td>1.0</td>
</tr>
<tr>
<td>F-measure</td>
<td>0.96</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2</th>
<th>Performance of cluster-based evolutionary ontology approximation.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original</td>
</tr>
<tr>
<td>Time (ms)</td>
<td>1841.5</td>
</tr>
<tr>
<td>Precision</td>
<td>1.0</td>
</tr>
<tr>
<td>Recall</td>
<td>0.86</td>
</tr>
<tr>
<td>F-measure</td>
<td>0.91</td>
</tr>
</tbody>
</table>

1For details about this parameter, please see [7].

11These two queries are naturally marked as the targets to be rewritten by the rewriting rules generated by GA. For details about the percentage of queries to be rewritten and the performance of classifiers have been given in [7] and [10].

†††Here we did not include the overhead of classifying and rewriting a query for execution. Also the modified ontologies should be loaded into the associated endpoints if necessary.
runtime performance on the ontology approximation approaches are very high (approximately 4 times faster) while their precision and recall are comparable. In contrast to the results on the query approximation approaches, the 2-AND gave worst F-measure performance on the ontology approximation approaches. Therefore, when we can apply our ontology approximation approach, ontology-based approximation performs better in those cases. Note that, in those experiments we did not consider any overheads on loading a modified ontology to the endpoint. In some cases we might have to relaunch the endpoint to load an ontology and also we might prepare a mirror of the endpoint within the clustered Front-end EP. To consider these limitations, the use of query approximation approaches in parallel is still meaningful on those clustered settings.

5. Conclusion

In this paper, a mixed approximation approach for processing inference-enabled SPARQL queries was presented. In the proposed approach, a query is processed on a clustered inference-enabled SPARQL processing engines whose approximation profiles are slightly different. We presented how a combination of query approximation and ontology modification could be effectively applied and how aggregating results on a cluster improved the runtime performance as well as the correctness of the results. To implement this approach, we employed a parallel evolutionary approach to make better approximation rules to be applied on the SPARQL processing phase and the ontology modification phase, and also utilized ensemble learning techniques to avoid poor performance scenarios. We have shown that how this approach can be implemented on our prototype system, and discussed about the performance on it.

For further improvement of approximation performance, the use of larger query logs for learning and optimization could be a good option [10]. However, making use of such query logs could have some privacy concerns due to sensitive information remaining in those queries. To realize a good anonymize method for gathered query logs is one of our future work.

Based on the SPARQL 1.1 specification [23], a query has been allowed to include sub-queries to another endpoints, called Federated-query. This makes each query very complicated and also it enforces consistent modifications of ontologies or sub-queries on a cluster of our Front-end EPs. Although sub queries could be intercepted by the Front-end EP for further optimization or approximation [20], it may require additional computation costs on consistent interception and approximation of such sub-queries. Presenting an architecture and a method to reduce such costs is also future work.

Acknowledgments

The work was partly supported by Grants-in-Aid for Challenging Exploratory Research 26540162 and JST CREST.

References


Naoki Fukuta received his B.E. and M.E. from Nagoya Institute of Technology in 1997 and 1999 respectively. He received his Doctor of Engineering from Nagoya Institute of Technology in 2002. Since April 2002, he has been working as a research associate at Shizuoka University. Since April 2007, he has been working as an assistant professor. Since April 2015, he has been working as an associate professor. In 2012, he received the IPSJ Yamashita SIG Research Award. His main research interests include Mobile Agents, Semantic Web, Knowledge-based Software Engineering, Logic Programming, Applications of Auction Mechanisms, and WWW-based Intelligent Systems. He served as a Deputy Chief Examiner (2011, 2012) and a Chief Examiner (2013) of Intelligent Technology Group, Editorial Committee of Journal of Information Processing. Since 2014, he is a senior member of IPSJ (Information Processing Society of Japan). He is a member of ACM (Association for Computing Machinery), IEEE-CS (IEEE Computer Society), JSIAI (Japanese Society for Artificial Intelligence), IEICE (Institute of Electronics, Information, and Communication Engineers), JSSST (Japan Society of Software Science and Technology), and ISSJ (Information Systems Society of Japan).

Naoki Yamada is a graduate student of Department of Informatics, Shizuoka University. He received the best paper award at 1st International Conference on Agents (ICA2016). His research interests include reasoning techniques and applications on the Semantic Web.

Yuji Yamagata is an engineer of UNITED, Inc. and he was a graduate student of Shizuoka University. On his student’s period, he had been working on realizing techniques and methods to realize better query processing on inference-enabled SPARQL endpoints.