Regular Paper

An Approach to Dynamic Query Classification and Approximation on an Inference-enabled SPARQL Endpoint

Yuji Yamagata[1,a]   Naoki Fukuta[1,b]

Abstract: On a retrieval of Linked Open Data using SPARQL, it is important to consider an execution cost of query, especially when the query utilizes inference capability on the endpoint. A query often causes unpredictable and unwanted consumption of endpoints' computing resources since it is sometimes difficult to understand and predict what computations will occur on the endpoints. To prevent such an execution of time-consuming queries, approximating the original query could be a good option to reduce loads of endpoints. In this paper, we present an idea and its conceptual model on building endpoints having a mechanism to automatically reduce unwanted amount of inference computation by predicting its computational costs and allowing it to transform such a query into a more speed optimized one by applying a GA-based query rewriting approach. Our analysis shows a potential benefit on preventing unexpectedly long inference computations and keeping a low variance of inference-enabled query executions by applying our query rewriting approach. We also present a prototype system that classifies whether a query execution is time-consuming or not by using machine learning techniques at the endpoint-side, as well as rewriting such time-consuming queries by applying our approach.

Keywords: SPARQL, inference, ontology mapping

1. Introduction

Due to the rapid emergence of the use and publish of Linked Open Data (LOD), constructing and refining efficiently querying systems to the LOD is becoming more important than before. LOD is generally retrieved by sending a query written in a standard query language called SPARQL [*1] for the retrieval of RDF data stored in an endpoint. Reasoning on LODs allows such queries to obtain unstated knowledge from the distinct one [1]. Techniques to utilize reasoning capability based on ontology have been developed to overcome several issues, such as its high complexity in worst case [2], [7], [10], [12], [14].

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, since the query is submitted directly to the endpoint, a standard SPARQL endpoint implementation will try to execute the query with lots of costs to return answers. If the endpoint receives lots of heavy queries, it might cause a server-down. This is especially important for endpoints that have inference engines to support OWL reasoning capability.

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. There are two approaches to predict the execution time for a query. One approach is using machine learning techniques to predict the execution performance from existing query logs. Another approach is to lookup some similar queries executed before and take an average of their execution time. In this paper, we take the former approach to classify whether a query execution is time-consuming or not.

In this paper, we present an idea and its conceptual model on building endpoints having a mechanism to automatically reduce unwanted amount of inference computation by predicting its computational costs and allowing it to transform such a query into a more speed optimized one by applying a GA-based query rewriting approach. Our analysis shows a potential benefit on preventing unexpectedly long inference computations but keeping low variance of inference-enabled query executions by applying our query rewriting approach. We also present a prototype system that classifies whether a query execution is time-consuming or not by using machine learning techniques at the endpoint-side, as well as rewriting such time-consuming queries by applying our approach.

The rest of the paper is organized as follows. Section 2 presents backgrounds of our work as well as presenting some existing works related to the issues, and clarify the aim of our research. Section 3 shows the detailed design of our approach and overview of our prototype implementation. Section 4 conducts evaluations of the approach and presents some analysis on them. Section 5 discusses about other possible approaches and difficulties

---

[1]   Graduate School of Informatics, Shizuoka University, Hamamatsu, Shizuoka 432–8011, Japan
[a]   gs14044@s.inf.shizuoka.ac.jp
[b]   fukuta@cs.inf.shizuoka.ac.jp

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

on them. Section 6 concludes our work.

2. Background

2.1 Difficulty of Inference-performance Prediction

For reducing time of reasoning, a direct way is to develop a faster inference algorithm. For example, Baumgartner et al. proposed an efficient reasoning algorithm based on hypertableau [2]. HermiT [12] reasoner implemented such an improved hypertableau reasoning algorithm.

In Ref. [14], 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 [*2] [11], JFact with ELK, and other experimental versions [*3]. 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 [10].

In Ref. [10], 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 they reported that there are still major difficulties to improve them.

In Ref. [10], 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 Ref. [7], 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 these subsets as increments. They reported that some ontologies exhibit nonlinear 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 hotspots [7], that are the parts that consume most of such huge amount of computation in inference.

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

---

[*2] ELK is OWL 2 EL reasoner.
[*3] https://code.google.com/p/more-reasoner/wiki/MOReCli

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 between the actual execution times and their estimated values for them. Now we have a good prediction method for estimating query execution performance, however, their approach does not consider Description Logic (DL) reasoning on those endpoints.

In this paper, we aim to solve the above mentioned issues and difficulties on predicting reasoning performance as follows. First, we focus on the prediction of performance on executing an inference-enabled query but do not aim to apply it to arbitrary reasonings in DL. Second, we apply machine learning-based approaches based on the features of a query and the used ontologies. However, unlike Hasan's approach, we do not aim to predict the exact cost of executing queries but rather try to predict whether the query is a time-consuming one based on those features and a given threshold for the execution time. After that, we aim to manage such queries that require a very long execution time.

2.2 Query Approximation

There are two possible approaches to managing long-running queries. One is to utilize parallel and distributed computing techniques to make those executions faster [16]. Another possible approach is rewriting a query that requires a long execution time to a light-weight one.

A possible approach to approximate an inference problem is deleting subsets of ontology that would increase the classification complexity [13]. Ontology refinements to enable a faster inference could be done by ontology engineers themselves. Ontology engineers are often working on the side of endpoints. However, clients do not know how such ontologies were optimized and what kind of inferences could be run on those ontologies. On the other hand, endpoints cannot apply some approximate optimizations since the endpoint could not fully predict clients' expectations that were not expressed in SPARQL, even when some dynamic ontology simplifications could be done for the queries. For example, endpoints do not know about time constraint for each query sender or demands for accuracy on those answers. Same issues are on the use of destructive query optimization (i.e., approximate the result of query but no guarantee of the exact result).

There are some query rewriting approaches to improve the quality of queries [3], [5], [6]. Also, there are some heuristic techniques to approximate inference-enabled queries by modifying some hotspots in the query that prevent faster execution [19]. However, since those hotspots are also dependent on their individual ontologies, those query modifications should take into account both the query-structure and characteristics of the ontologies used.

3. Our Approach

3.1 Outline and System Architecture

If a query seems not to be a time-consuming one, the endpoint
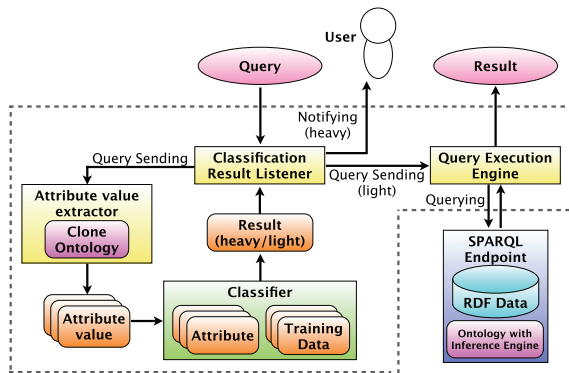
Fig. 1   Architecture of Front-end EP.

executes the query. If the query execution is classified as time-consuming, 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 in 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 the idea, we are implementing a preliminary system to classify whether a query execution is time-consuming or not, rewriting the query to a lighter-weight one, and extending the protocol to notify the rejection of the query, the applied query-transformation for the query, and so on. We 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. 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[19].

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 an optimized one based on the specified policies.

When the Front-end EP optimizes the submitted heavy query, an optimization process is applied and an optimized query is sent to the Back-end EP. Then the Back-end EP executes the received query optimized 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-optimization process has been applied, the optimized query may not return the exactly equivalent result to the one from the original query. Here, an issue is how the client knows that the client's original query is rewritten by the optimization 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 making much focus on query rewriting mechanism itself.

3.2   Query Classification

The aim of classifying queries is to classify whether a SPARQL

query execution is time-consuming or not. The classifier is implemented based on machine learning implementations (i.e., Weka[8], in this case). The classifier is placed at the front-end EP.

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.

We used Weka[8] 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.

3.3   GA-based Query Rewriting

It is demanded to get 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 those rewriting rules cannot be applied to a specific query, the Front-end EP executes each different heavy query on each time to get results for generating query rewriting rules for it. This might be even worse to execute such queries 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 cannot 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, we prepared a GA-based heuristic query-rewriting-rule generation engine that can produce heuristic rules to optimize some heavy queries that have never been executed yet.

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 [*4]. A 'then' part of each query rewriting rule is constructed with heuristic op-

---

[*4]   We execute the queries just once and memoize those values. When they are executed twice ore more, the memoized values are used to avoid further computation cost of executions.

```
select ?x where {
{?x rdf:type <http://linklings#ConferenceSession>}
UNION
{?x rdf:type <http://linklings#Administrator>}
}
```

Fig. 2   Target query A.

```
select ?x where {
{?x rdf:type <http://linklings#Administrator>}
UNION
{?x rdf:type <http://linklings#ConferenceSession>}
}
```

Fig. 3   An approximated query.

```
select ?x where {
{?x rdf:type <http://linklings#ConferenceSession>}
UNION
{?x rdf:type <http://linklings#AdminRegistered>}
}
```

Fig. 4   Target query B.

```
select ?x where {
{?x rdf:type <http://linklings#ConferenceSession>}
UNION
{?x rdf:type <http://linklings#Administrator>}
UNION
{?x rdf:type <http://linklings#Person>}
}
```

Fig. 5   Target query C.

```
select ?x where {
?x rdf:type <http://linklings#ConferenceSession>
}
```

Fig. 6   Target query D.

erations such as "change the n-th class appeared in the query [*5] 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.

A more detailed description about how our approach works is following. **Figure 2** shows a target query for generating a query rewriting rules. We may obtain an approximated query shown in **Fig. 3** and a query rewriting rule applied here is: "swap the first class description and second class description." Our GA-based approach aims to produce the query rewriting rules that will produce the query shown in Fig. 3 from the original query shown in Fig. 2. Here, this query rewriting rule can be applied to other similar queries. For example, we consider the case that we want to approximate a query shown in **Fig. 4**. This query has the same structure of the query shown in Fig. 2. We will get an approximated query by applying the query rewriting rule that we previously got by approximating the query shown in Fig. 2. Since the rule used here does not see the semantic structure of the target queries, it can be applied to the queries which do not have the exactly same structure of the original query, such as shown in **Fig. 5**. This query has similar structure to the query shown in Fig. 2. We will get an approximated query by applying the query rewriting rule that we previously got by approximating the query shown in Fig. 2. Unfortunately, this rule cannot be applied to some queries,

such as shown in **Fig. 6**. Here, we can not obtain an approximated query by applying the query rewriting rule because in this query a description of a class name appears only once.

Here let, "swap the first class description and second class description" be rule 1, another rewriting rule, "change the n-th class appeared in the query to its superclass" be rule 2, our approach generates a rule: "First try to apply rule 1 and if it can not be applied, try to apply rule 2" for wider applicability of generated rewriting rules.

4.   Experimental Analysis

4.1   Baseline Inference Performance

As a first step of our evaluation, we conducted a set of experiments to know a potential benefit on speed optimizations for query executions by applying our query rewriting approach.

In the preliminary experiments, 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 protégé (v4.3) [*6].

As an experimental environment we set up a SPARQL endpoint using Joseki (v3.4.4) in conjunction with server-side Pellet [17] reasoner to enable OWL-level inference capability on the endpoint.

We used an Intel Core 2 Duo 2.6 GHz MacBookPro, with 6 GB 667 MHz DDR2 SDRAM dedicated to the Java Virtual Machine (JVM v1.7) to run the endpoint. The system runs on OS X 10.8.5.

Here, we need to consider a caching effect, in other words, materializing [4] the inference results to be reused for later queries on the query processing at the endpoint-side. The first time execution performance is calculated from the sum of 100 times of query execution and the sum of 200 times of query execution from the first time.

Figure 7 shows the result of average performance on each query to get the specified class instances. Here, we can see that the query to get instances of class <http://linklings#ConferenceSession> is time-consuming in spite of the query to get instances of superclass <http://linklings#Session> is not so
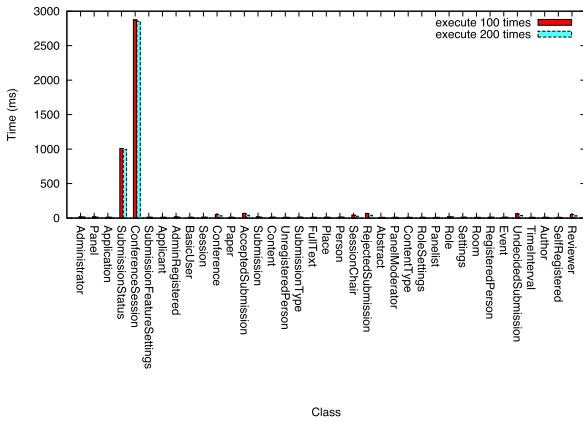

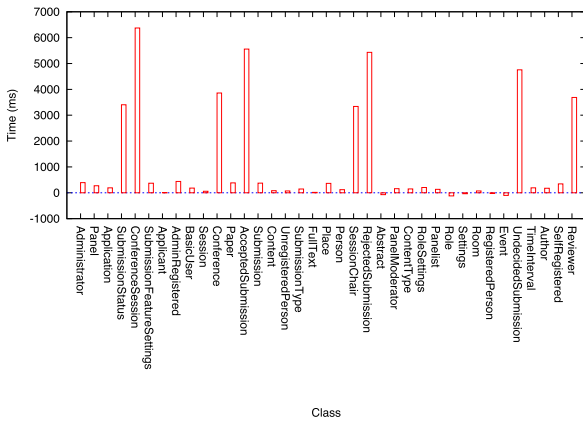**Fig. 7** Baseline query execution performance (average).

Fig. 8 Baseline query execution performance (initial execution).

time-consuming. This observation shows the potential applicability of our query rewriting approach. In this case, when we assume <http://example#ConferenceSessionName> is mostly equal to data property whose domain is ConferenceSession, we can make a rewriting suggestion of the alternate query getting the instances of <http://linklings#ConferenceSession> as follows:

```
SELECT ?x WHERE {
 ?x rdf:type <http://linklings#ConferenceSession> .
}
to
SELECT ?x WHERE {
 ?x rdf:type <http://linklings#Session> .
 ?x <http://example#ConferenceSessionName> ?y
}
```

Figure 8 shows the required time for the first time of query execution measured by the above way.

Here, we can see that when we consider a case without caching effect, a number of classes could also be time-consuming.

4.2 Query Classification Performance

The evaluation data set was generated by queries getting the instances of a named class in the Linklings ontology. Each query gets the instance 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 are taken into account. For example a query:

Table 1 Classification performance on our approach (leave-one-out cross validation).

Classifier	Recall	Precision	F-Measure
C4.5	0.895	0.896	0.896
Bagged C4.5	0.959	0.977	0.964
Boosted C4.5	0.999	0.999	0.999

```
SELECT ?x WHERE {
{?x rdf:type <http://linklings#Submission> }
UNION
{?x rdf:type <http://linklings#Administrator>}
}
```

is treated as not identical to the following query:

```
SELECT ?x WHERE {
{?x rdf:type <http://linklings#Administrator> }
UNION
{?x rdf:type <http://linklings#Submission>}
}
```

In this experiment, as the attributes of the queries, we just used two simple heuristic functions. One is just counting-up the number of classnames in the queries and the other is obtaining first-appearing classnames and encoding it to a value.

We measured average elapsed time to execute these queries 100 times for light queries and 10 times for some quite heavy queries. Here, we treat queries whose execution times exceed 200 ms as heavy queries [*7].

Table 1 shows the result of our query classifier performance, based on the dataset that queries to the instances used in previous experiments. Here, we conducted the experiment for total 1,369 queries [*8] on leave-one-out cross validation. We used two classifiers: Bagged C4.5, and Boosted C4.5, implemented in Weka [8], with default parameters. Also we added C4.5 for a reference classifier to know the baseline difficulty of the problem. Here, we can see that this performance is very good, especially on Boosted C4.5. On Bagged C4.5, all 'heavy' queries were correctly classified but 54 non-heavy queries were wrongly classified as 'heavy' ones. On Boosted C4.5, all non-heavy queries were correctly classified but two 'heavy' queries were wrongly classified as non-heavy ones. Note that, in this experiment we did not consider any noise caused by the variance of loads on the endpoints. Further detailed evaluations considering those realistic conditions are our future work.

4.3 Query Rewriting Performance

Here, we applied a very conventional GA [15] for generating good query rewriting rules. We simply encode some heuristic rules, such as, just replace the 1st appeared class to its superclass, and so on. The Front-end EP stores some time-consuming queries and their results. Then the GA is applied to those data. 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 could be applied to the queries that may not return their results due to the timeout of their executions.

[*7] We assume that this threshold value can be decided statically as the policy of operating an endpoint to protect it from denial of services.

[*8] In these queries, 74 were marked as 'heavy' queries.

Table 2   Performance of GA-based query approximation (Case 1).

	Original	GA based	Optimal
Execution Time	4,760 ms	2 ms	2 ms
F-measure of results	–	0.93	1

Table 3   Performance of GA-based query approximation (Case 2).

	Original	GA based	Optimal
Execution Time	1,991 ms	12.5 ms	10.4 ms
F-measure of results	–	0.87	1

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. A query prepared for this evaluation is one of queries that we manually found as heavy queries. Since we manually found an ideal query rewriting rule for this query, we are able to show an ideal F-measure and query execution time. We show them in **Tables 2** and **3**. In this evaluation, we used 50 individuals, set inversion rate 0.2, mutation rate 0.15 (to superclass), 0.15 (to subclass), and 0.05 (random). Average generations was 35.3. According to Tables 2 and 3, our GA approach can find an ideal query rewriting rule in most cases. Tables 2 and 3 also show there is a possibility that GA approach can find a query rewriting rule generating a query that can execute within short time compared with an original query and have a high F-measure.

Fitness values for individuals are defined as

$$V_q = \alpha F_q + \beta \frac{T_Q - Tq}{T_Q} \text{ such that } \alpha + \beta = 1$$

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.

The following is an overview of our GA-based algorithm:

1: **function** GA-Optimizer($Q, GAParam$)
2:   $CAns := getCorrectAnswers(Q)$;
3:   $N := GAParam.topN$
4:   $Individuals := initInvividuals(Q, GAParam)$;
5:   $Evals := getFitnessValues(Individuals, Q, CAns)$
6:   **while** stopping condition is false
7:     $Individuals = topN(Individuals, Evals, N)$
8:     $Individuals = applyGeneticOPs(Individuals, GAParam)$
9:     $Evals := getFitnessValues(Individuals, Q, CAns)$
10:   **end while**
11:   **return** $topN(Individuals, 1)$

Here, the function *GA-optimizer* has two arguments, Q that contains the sample queries, and *GAParam* that specifies GA parameters as we described. The function gathers correct answers of Q as well as their execution time and stores them into *CAns*. The *CAns* is used to evaluate *individuals* by using the fitness function that we described above. Then, Genetic operators are applied to the selected individuals and continue this process while stopping condition is false. Then the function returns the best one.

5.   Discussion

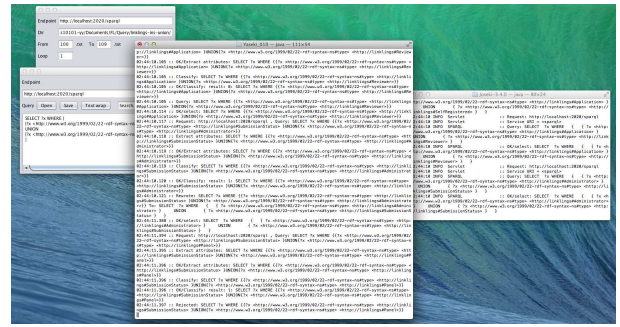We implemented our Front-end EP which behaves as a proxy

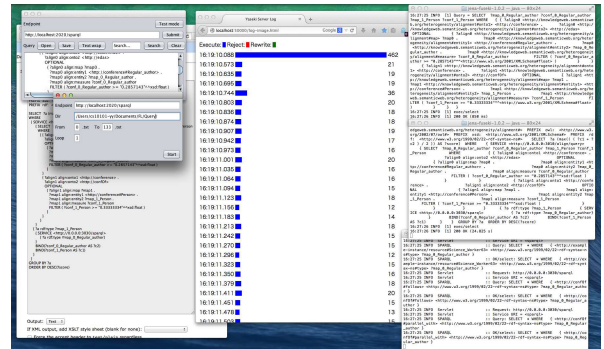**Fig. 9**   Overview of implemented Front-end EP.

Fig. 10   Overview of our system.

endpoint. **Figure 9** shows an overview of running the Front-end EP. Since it behaves as a proxy endpoint, it can be placed to be run on both a client-side and a server-side environment.

We also implemented a preliminary prototype client-system that has some abilities to support the coding process of SPARQL queries to make faster execution of queries, as well as helping developers who do not know much about the stored LOD on the endpoint. **Figure 10** shows an overview of our system.

We implemented an approach which uses an advance investigation to identify hotspots that are limiting the speed of query executions, as well as using machine learning techniques. The module consists of several sub modules such as endpoint-investigator, query classifier, etc.

Identifying hotspots often requires the actual loads of endpoints, as well as the structural hotspots existed in the ontology. A possible approach, for example, is downloading the ontology and identifying hotspots by analyzing the downloaded ontology if the ontology is not so large and bulk-download of the whole ontology is allowed. Based on such possible approaches, we implemented a client-system to support developers to prepare better SPARQL queries considering their reasoning times on the server-side.

The prediction method implemented on our system uses some of characteristic attributes of the queries that are helpful to predict their query execution time to be used on machine learning algorithms. However, avoiding time-consuming query executions by a client-side has some limitations. For example, some people may not care about the loads of an endpoint and send time-consuming queries as is, even when they knew it. Even when an endpoint receives only a limited number of such queries, the endpoint may not be able to respond to even simple queries. Therefore, as shown in **Fig. 11**, our Front-end EP is also designed to be used to cover multiple Back-end EPs which might share some
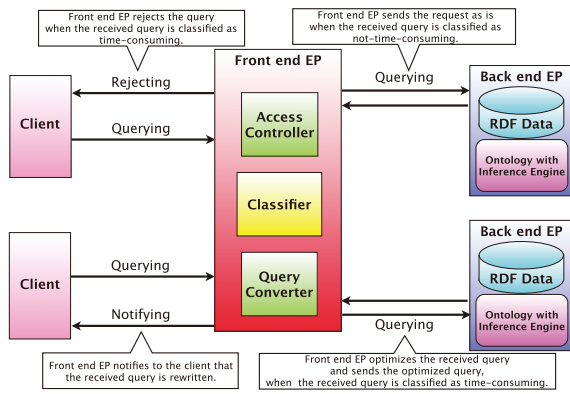
Fig. 11 Basic query processing procedure.

computational resources.

6. Conclusion

In this paper, we presented a preliminary idea and concept on building SPARQL endpoints having a mechanism to automatically decrease or reject unwanted inference computation by using machine learning and dynamic query rewriting techniques. Our preliminary experiment showed a potential benefit on speed optimizations of query executions by applying our query rewriting approach. We also presented a preliminary prototype system that distinguishes whether a query execution is expected to be time-consuming or not by using machine learning techniques at the endpoint side.

Future works include gathering and using various queries and its resulting attributes among multiple endpoints and clients. That involves issues on protecting anonymity to protect privacy of clients and the efficiency of such query gathering process itself. The one of approach to cope with that issue is separating results of machine learning from training data that are queries and their execution time. Front-end EP only uses results to predict the received query's execution time.

Reducing overheads on the Front-end EP is also one of future work. It might require a further analysis the actual computational costs to classify whether a SPARQL query execution is time-consuming or not based on different machine learning techniques, and how the query rewriting method reduces costs by keeping its query rewriting performance.

From the SPARQL 1.1 specification [18], a query has been allowed to include sub-queries to another endpoints, called Federated-query. Although our approach could be extended to accept such Federated Queries, it makes each query very complicated so that it might be difficult to directly optimize or approximate queries on an Front-end EP. When typical each sub query is very simple, we could apply our approach to those sub queries [20]. However, although we have done some discussions about evaluations [21], currently our evaluation has been done on rather simple queries. Therefore, the applicability of our approach to such complex queries should additionally be evaluated. Giving evaluation for further complex and complicated conditions is one of our future works. Also, even when such sub queries could be intercepted and processed by the Front-end EP for further optimization or approximation, it may spend a certain com-

putation cost on each interception. Presenting an architecture and a method to reduce such costs is also our future work.

References

[1] Baader, F. and Suntisrivaraporn, B.: Debugging SNOMED CT Using Axiom Pinpointing in the Description Logic $\mathcal{EL}^+$, *Representing and sharing knowledge using SNOMED, Proc. 3rd International Conference on Knowledge Representation in Medicine KR-MED 2008*, Cornet, R. and Spackman, K. (Eds.), Vol.410, CEUR-WS, pp.1–7 (2008).

[2] Baumgartner, P., Furbach, U. and Niemelä, I.: Hyper Tableaux, *Logics in Artificial Intelligence, LNCS*, Alferes, J.J., Pereira, L.M. and Orlowska, E. (Eds.), Vol.1126, pp.1–17, Springer-Verlag (1996).

[3] Bischof, S. and Polleres, A.: RDFS with Attribute Equations via SPARQL Rewriting, *The Semantic Web: Semantics and Big Data, LNCS*, Cimiano, P., Corcho, O., Presutti, V., Hollink, L. and Rudolph, S. (Eds.), Vol.7882, pp.335–350, Springer-Verlag (2013).

[4] Dell'Aglio, D. and Valle, E.D.: IMaRS: Incremental Materialization for RDF Streams, *Tutorial on Stream Reasoning for Linked Data at ISWC 2014* (2014).

[5] Fujino, T. and Fukuta, N.: SPARQLoid – A Querying System using Own Ontology and Ontology Mappings with Reliability, *Proc. 11th International Semantic Web Conference* (*Poster & Demos*) (*ISWC 2012*) (2012).

[6] Fujino, T. and Fukuta, N.: Utilizing Weighted Ontology Mappings on Federated SPARQL Querying, *The 3rd Joint International Semantic Technology Conference* (*JIST2013*), *LNCS*, Kim, W., Ding, Y. and Kim, H.-G. (Eds.), Vol.8388, pp.331–347, Springer International Publishing (2013).

[7] Gonçalves, R.S., Parsia, B. and Sattler, U.: Performance Heterogeneity and Approximate Reasoning in Description Logic Ontologies, *The Semantic Web–ISWC 2012 Part I, LNCS*, Cudré-Mauroux, P., Heflin, J., Sirin, E., Tudorache, T., Euzenat, J., Hauswirth, M., Parreira, J.X., Hendler, J., Schreiber, G., Bernstein, A. and Blomqvist, E. (Eds.), Vol.7649, pp.82–98, Springer-Verlag (2012).

[8] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P. and Witten, I.H.: The WEKA Data Mining Software: An Update, *ACM SIGKDD Explorations Newsletter*, Vol.11, No.1, pp.10–18 (2009).

[9] Hasan, R. and Gandon, F.: A Machine Learning Approach to SPARQL Query Performance Prediction, *Proc. IEEE/WIC/ACM International Conference on Web Intelligence 2014* (*WI 2014*), pp.266–273 (2014).

[10] Kang, Y.-B., Li, Y.-F. and Krishnaswamy, S.: Predicting Reasoning Performance Using Ontology Metrics, *The Semantic Web–ISWC 2012 Part I, LNCS*, Cudré-Mauroux, P., Heflin, J., Sirin, E., Tudorache, T., Euzenat, J., Hauswirth, M., Parreira, J.X., Hendler, J., Schreiber, G., Bernstein, A. and Blomqvist, E. (Eds.), Vol.7649, pp.198–214, Springer-Verlag (2012).

[11] Kazakov, Y., Krötzsch, M. and Simančík, F.: Concurrent Classification of $\mathcal{EL}$ Ontologies, *The Semantic Web–ISWC 2011*, pp.305–320, Springer (2011).

[12] Motik, B., Shearer, R. and Horrocks, I.: Hypertableau Reasoning for Description Logics, *Journal of Artificial Intelligence Research*, Vol.36, pp.165–228 (2009).

[13] Nikitina, N. and Schewe, S.: Simplifying Description Logic Ontologies, *Proc. 12th International Semantic Web Conference* (*ISWC 2013*), pp.411–426 (2013).

[14] Romero, A.A., Grau, B.C. and Horrocks, I.: MORe: Modular Combination of OWL Reasoners for Ontology Classification, *The Semantic Web–ISWC 2012 Part I, LNCS*, Cudré-Mauroux, P., Heflin, J., Sirin, E., Tudorache, T., Euzenat, J., Hauswirth, M., Parreira, J.X., Hendler, J., Schreiber, G., Bernstein, A. and Blomqvist, E. (Eds.), Vol.7649, pp.1–16, Springer (2012).

[15] Russell, S.J. and Norvig, P.: *Artificial Intelligence: A Modern Approach* (*2nd Edition*), pp.116–119, Prentice Hall (2002).

[16] Schätzle, A., Przyjaciel-Zablocki, M., Hornung, T. and Lausen, G.: PigSPARQL: A SPARQL Query Processing Baseline for Big Data, *Proc. 12th International Semantic Web Conference* (*Poster & Demos*) (*ISWC 2013*) (2013).

[17] Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A. and Katz, Y.: Pellet: A practical OWL-DL reasoner, *Journal of Web Semantics*, Vol.5, No.2, pp.51–53 (2007).

[18] The W3C SPARQL Working Group: SPARQL 1.1 Overview.

[19] Yamagata, Y. and Fukuta, N.: A Dynamic Query Optimization on a SPARQL Endpoint by Approximate Inference Processing, *Proc. 5th International Conference on E-Service and Knowledge Management* (*ESKM 2014*), pp.161–166 (2014).

[20] Yamagata, Y. and Fukuta, N.: Approximating Inference-enabled Federated SPARQL Queries on Multiple Endpoints, *Proc. 13th International Semantic Web Conference* (*Poster & Demos*) (*ISWC 2014*) (2014).

[21] Yamagata, Y. and Fukuta, N.: Evaluating a GA-based Approach to Dynamic Query Approximation on an Inference-enabled SPARQL Endpoint, *Proc. 14th IEEE/ACIS International Conference on Computer and Information Science* (*ICIS 2015*), pp.143–148 (2015).

Yuji Yamagata   is a graduate student of Shizuoka University.   He received his Bachelor of Informatics degree from Shizuoka University in 2014. His research interests include Semantic Web and AI applications.   He is a student member of IPSJ.

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, SemanticWeb, Konwledge-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. He is a member of ACM, IEEE-CS, JSAI, IEICE, JSSST, and ISSJ.