Computing (2020) 102:1743–1763
https://doi.org/10.1007/s00607-020-00814-9

REGULAR PAPER

A cache-based method to improve query performance of linked Open Data cloud

Usman Akhtar[1] · Anita Sant'Anna[2] · Chang-Ho Jihn[3] · Muhammad Asif Razzaq[1] · Jaehun Bang[1] · Sungyoung Lee[1]

Received: 9 July 2019 / Accepted: 25 April 2020 / Published online: 14 May 2020

Abstract
The proliferation of semantic big data has resulted in a large amount of content published over the Linked Open Data (LOD) cloud. Semantic Web applications consume these data by issuing SPARQL queries. One of the main challenges faced by querying the LOD web cloud on account of the inherent distributed nature of LOD is its high search latency and lack of tools to connect the SPARQL endpoints. In this paper, we propose an Adaptive Cache Replacement strategy (ACR) that aims to accelerate the overall query processing of the LOD cloud. ACR alleviates the burden on SPARQL endpoints by identifying subsequent queries learned from clients historical query patterns and caching the result of these queries. For cache replacement, we propose an exponential smoothing forecasting method to replace the less valuable cache content. In the experimental study, we evaluate the performance of the proposed approach in terms of hit rates, query time and overhead. The proposed approach is found to outperform existing state-of-the-art approaches, increase hit rates by 5.46%, and reduce the query times by 6.34%.

Keywords Query performance · Cache Replacement · Linked Open Data · SPARQL

Mathematics Subject Classification 68P20

1 Introduction

The Linked Open Data (LOD) cloud provides a global information space with a wealth of structured facts. The LOD cloud offers for example Geo-location facts[1] and cross-domain information (e.g. DBpedia, YAGO [32] and WIKIdata). Currently, it is estimated that more than thirty billion facts have been published over the LOD [33] cloud. The data in the LOD cloud is represented using the Resource Descrip-

Extended author information available on the last page of the article

[1] https://linkedgeodata.org/.

tion Framework (RDF)[2] and RDF Query Language (SPARQL)[3] is used as querying language. The rapid expansion of LOD use in academia and industry evidences the efficient retrieval of data as one of its major challenges. Although every LOD cloud supports SPARQL queries to access data from its publicly available interfaces, a central problem is the lack of trust regarding these endpoints due to network instability and latency. Therefore, the typical solution is to dump the data locally and maintain endpoints to process these data. Recent investigations [3,5,10,13,14,17] have shown that the content of LOD is dynamic over time and continuously evolving. However, the data stored at the local endpoints are not up-to-date and require constant updates, therefore, accurately hosting the endpoints requires expensive infrastructure support.

In recent years, many efforts have been made to circumvent the problem of effectively querying LOD [24,31,35], among these, caching [22,23] is the most popular technique to reduce query time by serving the requests from a cache (also called *cache hits*) [28]. In the literature, two types of caching have been proposed, client and server-side caching. In client-side caching, requests are immediately served from the nearest cache to reduce the latency and network traffic. However, client-side caching is not fully explored [24] and is still in the early days of research. Server-side caching is not flexible to support different querying patterns and design of server-side caching usually depends on the database. As cache has limited space, it is important to fill it with valuable content by replacing unnecessary content. Many cache replacement techniques have been developed for relational databases, such as LRU [8] and LFU [18]. The underlying structure of the LOD is different from relational databases. The caching algorithms designed for relational databases are not fully applicable in LOD scenarios [21]. To the extent of our knowledge, we believe, there is very limited work addressing the problem of efficiently querying and retrieving data from LOD cloud [22,29,36–38].

In this paper, we propose an Adaptive Cache Replacement strategy (ACR) in order to accelerate overall query processing. ACR works as a proxy between the querying agents and the SPARQL endpoint. We adopt client-side caching as it is a domain-independent approach that does not require underlying knowledge of the LOD. Typically, the queries issued by the end user are repetitive and follow similar patterns that only differ in specific element. The major challenge of this task is to find similar queries, as it is possible that two queries are structurally similar but may differ in content. To tackle this problem, we propose the use of a bottom-up matching approach to find similar queries. For the structural similarity, we first compute the distance between the triple patterns and prefetch the results of similar queries to be placed in cache. A cache has limited space, therefore it is advantageous to replace it with frequently-accessed data. In this work, our cache replacement utilizes exponential smoothing forecasting to calculate the frequency of the accessed data and replace content based on access frequencies. More specifically, we propose a full-record replacement strategy, in which at every new query the hit frequency of accessed triples is calculated using exponential smoothing and the cache is replaced with the highest access queries. The motivation behind adaptive cache replacement is to improve the querying efficiency and reduce the burden on the SPARQL endpoint. Repeated queries are cached

[2] https://www.w3.org/RDF/.

[3] https://www.w3.org/TR/rdf-sparql-query/.

locally, and the results of these queries are immediately answered to the user. Our approach optimizes the results of predicted potential queries and less-valuable queries are replaced from the cache.

We now summarize the key contributions of this paper.

- We propose a client-side caching that works as a proxy between the querying agent and SPARQL endpoint. To accelerate the querying answering process, our approach can either be deployed within SPARQL endpoint or querying agent to eliminate the burden on these endpoint.
- Our approach introduces a distance-based query similarity metric, which considers both content-wise and structural similarities for more accurate comparison between queries.
- We propose an exploratory prefetching to retrieve contents possibly requested at the future queries by identifying the concepts of the previously issued queries and issuing a single query for all required contents. Its benefit is to reduce the transmission overhead and improve the hit rate and query time by retrieving contents for future queries at once.
- We propose a frequency-based cache replacement method to rank each query according to its estimated access frequency. The most frequently accessed queries are kept in the cache. Thus, our work benefit the triple stores in replacing the cache.
- Comprehensive evaluation on real-world LOD datasets showcase the effectiveness of our approach. The evaluation result outperforms the state-of-the-art approaches such as (LRU) Least Recently Used, (LFU) Least Frequently Used and (SQC) SPARQL Query Caching [24] in terms of higher hit rate, shorter query time, and less overhead.

The remainder of this paper is organized as follows. Section 2 explains the background of representing and querying LOD and briefly explains the related work in comparison to the proposed approach. Section 3 explains the main phases of the approach to find similar-structured queries and perform cache replacement. The evaluation of the proposed approach is performed in Sect. 4. This article is concluded in Sect. 5.

2 Background

In this section, we briefly discuss the background needed to understand LOD, which includes the data representations and the querying of the LOD. Moreover, we also discuss an overview of related work in the area of semantic caching and query suggestion, and highlight the differences of the proposed approach with respect to state-of-the-art.

2.1 Data representation

The Semantic Web [2] is an extension of the Web of Data [1]. As described by Tim Berners-Lee [2], the Semantic Web enables the machine in such a way that data can be searched, interpreted and reused. LOD is another important concept in the Semantic

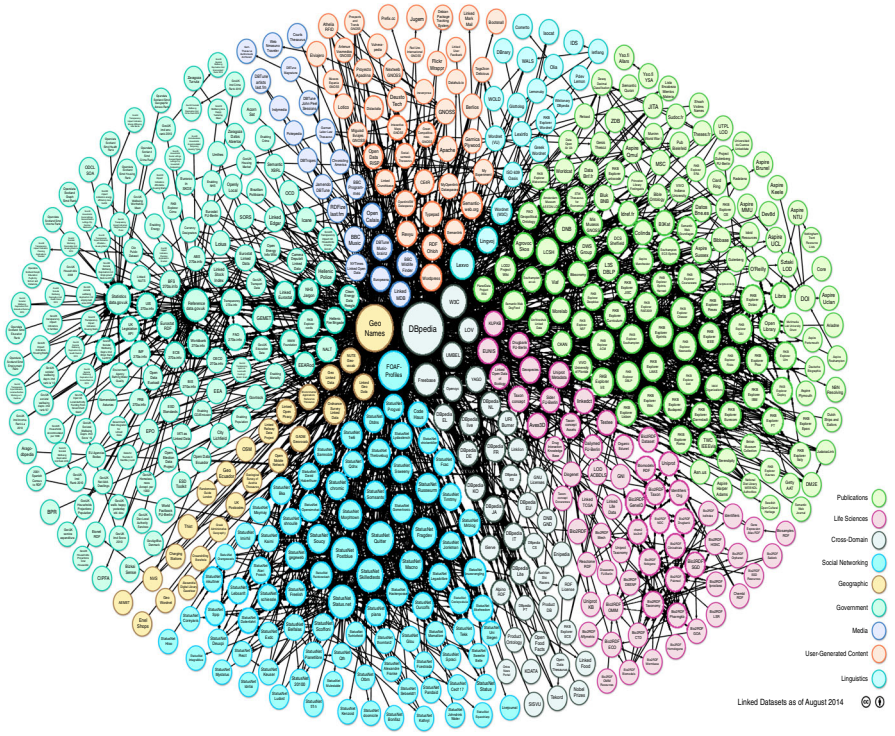

Fig. 1 Showing the LOD diagram containing interlinked data from multiple domains

Web, enabling the machine to browse the web, such as DBpedia.[4] LOD is collaboratively built from web corpus and represents knowledge in structured form. Facts are stored inside the Knowledge Base (KB) and an inference engine is used to assert conditions based on these facts. There are a number of different publicly available KBs, such as Freebase [4], DBpedia [20] and Yago [32]. KBs are further categorized into curated and open KBs. In curated KB, factual information is represented in the form of entity-relationship (e.g. https://en.wikipedia.org). In an open KB, the facts are automatically collected from web pages and are often linked together, e.g., by using LOD. With the current evolution of the Semantic Web 3.0, the LOD enables data to linked between sources, as shown in Fig. 1. The LOD cloud contains almost 570 datasets from different domains that are interlinked with each other.

The data representation in curated KBs follows a predefined schema, entities and relationships are modeled using a Resource Description Framework (RDF). The RDF is considered the standard representation for a curated KB, where the relationships are represented in many sets of triples, i.e., (*subject*, *predicate* and an *object*). These sets of triples are a fundamental part of a KB and are also known as a knowledge graph. The knowledge graph allows the sharing of data and the further linking of these data sets in the LOD cloud. However, in an open KB, facts are also represented but they

[4] https://wiki.dbpedia.org/.

```
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX dbpedia: <http://dbpedia.org/>
SELECT ?philosopher1 ?philosopher2
WHERE {
?philosopher1 foaf:name "Auguste_Comte" .
?philosopher1 ?relationshipWith :Paris .
} UNION {
?philosopher2 dbo:influenced ?philosopher1 .
OPTIONAL {
?philosopher2 foaf:givenName "Jean-Baptiste_Say"
}
}
```

Fig. 2 Showing the example of a SPARQL query

do not follow a strict schema, and these data are available in various formats, such as *N*-triples and the turtle format.

2.2 SPARQL

SPARQL[5] is a widely used graph based standard query language to retrieve and manipulate data that are stored in the RDF format. SPARQL is a structured query language standardized by the W3C for querying RDF triple stores.[6] The syntax of SPARQL contains different and disjoint query types such as $SELECT$, $CONSTRUCT$, ASK and $DESCRIBE$. To extract the values from the endpoints, $SELECT$ is widely used. As an example of SPARQL query illustrated in Fig. 2, the query pattern contains $SELECT$ statement that limit the projection to the certain variables used in the query such as $?philosopher1$ and $?philosopher2$. This query used the $UNION$ and $OPTIONAL$ as a basic operations for modifying the content of SPARQL query. LOD cloud also provides SPARQL endpoint for their datasets. However, querying SPARQL endpoints is troublesome due to network instability, and the connection to these endpoints can be temporarily lost, which affects the query efficiency. These endpoints do not provide any information about dataset modification. Therefore, long-running applications that use a cache must resubmit the queries to keep the local data cache up-to-date.

2.3 Related work

A number of works have dealt with issuing related to query LOD. This section present related work organized under two topics: (1) *query suggestion*, e.g., techniques to find the similar-structured queries, (2) *semantic caching*, e.g., the main idea of semantic caching is to maintain previously accessed data in a cache. We summarize the state-of-the-art approaches and briefly discuss their advantages and disadvantages in Table 1.

[5] https://www.w3.org/TR/sparql11-overview/.

[6] http://www.w3.org/TR/rdf-sparql-query/.

Table 1 A comparison of related work

Work	Advantages	Disadvantages	Method	Query similarity	Cache replacement	Pre-fetching
Martin et al. [24]	Cache complete triples query results; introduced the proxy layer between the application and SPARQL endpoint to cache repeated query results	Only considers repeated queries	Structure-based similarity	–	✓	–
Chun et al. [6]	Proposed maintenance policy that update the cache prior to query execution	Only update the local cache at the system idle time	Content-based similarity	–	✓	–
Godfrey et al. [12]	Define a general framework in logic for semantic query caching	The proposed algorithm has exponential time complexity	Content-based similarity	–	–	–
Yang et al. [35]	Adaptive cache to store intermediate results of a SPARQL query	No cache policy was introduced in their work	Result-based similarity	✓	✓	–
Dar et al. [7]	Proposed semantic region-based caching and a distance measure to update cache	Only considers a semantic region rather than tuples	Distance-based similarity	–	✓	–

Table 1 continued

Work	Advantages	Disadvantages	Method	Query similarity	Cache replacement	Pre-fetching
Shu et al. [31]	Introduced *query containment* which evaluated whether a query can be answered from the cache or not	Containment checking is a computationally expensive task	Content-based similarity	✓	–	✓
Papailiou et al. [26]	Work-load adaptive caching to reduce the SPARQL query response time; introduced canonical labelling for optimal join execution plan	No policy for cache replacement was introduced in their work	Result-based similarity	✓	–	–
Lehman et al. [19]	Proposed a machine learning approach to leverage the query processing over KBs; no knowledge of underlying schema is required	The Feature modeling method introduced in their paper is time-consuming	Structure-based similarity	–	–	✓
Fernández et al. [10]	Proposed an archiving system to efficiently retrieve data from evolving RDF	Unable to replace the cache during system idle time	Structure-based similarity	–	✓	–
Proposed	Proposed an adaptive cache replacement that accelerates the overall processing of querying over earlier works	Prefetching the previously issued queries degrades the performance of the system	Structure and content-based similarity	✓	✓	✓

2.3.1 Query suggestion

Recently, query suggestion has been introduced into SPARQL processing. It plays a vital role in improving the overall processing of the query. The suggestion is made based on the mining of similar queries from logs. Graph Edit Distance (GED) [30] is normally applied to measure the structural similarity between SPARQL queries. However, GED is very computationally expensive, and the use of structure similarity is insufficient. It is possible that two SPARQL queries are same but differ in their result. To overcome this drawback, Shu et al. [31] proposed a content-aware approach that utilized *query containment* to estimate whether the queries can be answered from the caches. However, this approach is not widely utilized by the semantic web community since the containment checking approach produces very significant overhead. Lorey et al. [23] proposed a query augmentation approach to alter SPARQL queries to detect frequently recurring patterns. The benefit of their approach is to answer the query from the cache without accessing the LOD. However, the major limitation is that it considers only the queries requested by the same agent and the hit rate of the template-based approach is only 39% [15]. In contrast to the aforementioned query suggestion methods, our approach considers both content-wise and structural similarities based on a simple distance score, which results in a higher hit rates, shorter query time, and less spatial overhead.

2.3.2 Semantic caching

Semantic caching was originally proposed for the Database Management System (DBMS) [7,24] and the purpose of the DBMS is to reduce the overhead of retrieving data from the cloud. Godfrey et al. [12] proposed the notion of semantic overlaps and introduced a caching approach that utilized client-server systems. To extend this idea, Dar et al. [7] proposed a semantic region-based caching technique and introduced a distance metric to update the cache such that the cold (e.g., less frequently access) regions are removed from the cache. Martin et al. [24] proposes to selectively invalidate cache objects on updates of the knowledge base by identifying the affected query results. However, their work does not consider query similarity for cache replacement. Yang et al. [35] proposed server-side caching to decompose the query into the basic graph patterns and cache their intermediate results. To prefetch similar-structured queries, Lehman et al. [19] proposed a supervised machine learning approach that performed analysis on the user's previously issued queries. Their approach filters the range of possible answers and utilizes a learning technique to ensure that no prior knowledge of the underlying schema of LOD is required. Nishioka et al. [25] proposed a periodic crawling strategy that predict whether the change occurs in RDF triples. However, Lehman et al. [19] and Nishioka et al. [25] did not consider the system overhead as their performance measure. Recently, a proactive policy for maintaining local cache is proposed by Chun et al. [6] that alleviates the expensive job of copying the LOD at idle time. In summary, only a few works have been reported to deal with the problems related to the semantic caching for SPARQL queries. We propose a client-side adaptive strategy to utilize caching for SPARQL query processing. The goal of our research is

to keep track of the access queries and evicts the less valuable content from the cache in an overhead-efficient way and regardless of system idle time.

3 Proposed methodology

Web-users increase the burden of SPARQL endpoints by issuing similar queries repeatedly and suffer high search latency due to the inherently distributed nature of LOD. To alleviate the burden and latency, we propose a query similarity based caching method and access frequency based cache replacement policy. In contrast to existing approaches [26,31], our proposed method is computationally efficient and more versatile since the similarity measure utilizes a simple distance score and considers content-wise similarity as well as structural similarity. These advantages are utilized to propose our performance-effective and overhead-efficient prefecthing policy with query template and offline processing. Based on our novel similarity metric, more specifically, we utilize exploratory prefetching to search and gather contents possibly requested at the future queries by identifying the concepts of the previously issued queries. Our cache replacement module is adaptive when the number of queries reaches a predefined limit, the cache replacement process is triggered, replacing infrequently accessed data in the cache. Our replacement is effective as compared to the existing approaches [6,24] in terms of less overhead and higher hit rates. Figure 3 illustrates the architecture of the proposed overall client-side cache replacement approach. Our methods are discussed in more detail in the following sections.

3.1 SPARQL query similarity

The existing approach [24] relies on the structural similarity of the query for improving the performance of the triple stores. We argue that the structural similarity is based on the ordering of the symbols and it is not sufficient as two queries may represent the same structure of ordering but share a different content.

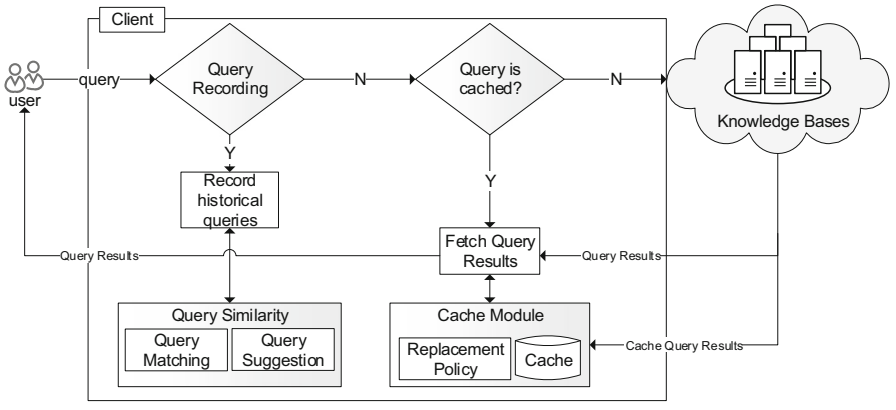

Fig. 3 Block diagram of the proposed approach

```
1 PREFIX :<http://dbpedia.org/resource/>
2 PREFIX dbpedia2:<http://dbpedia.org/property/>
3 PREFIX dbpedia:<http://dbpedia.org/>
4 SELECT ?city ?influence
5 WHERE {
6 ?city1 rdfs:label "Paris" .
7 ?person ?relationshipWith :city1 .
8 :Auguste_Comte foaf:givenName "Auguste" .
9 }
```

(a) Example of a SPARQL query with BGP_1

```
1 PREFIX :<http://dbpedia.org/resource/>
2 PREFIX dbpedia2:<http://dbpedia.org/property/>
3 PREFIX dbpedia:<http://dbpedia.org/>
4 SELECT ?city ?influence
5 WHERE {
6 :Auguste_Comte foaf:surname "Comte" .
7 ?city2 rdfs:label "Montpellier" .
8 ?Auguste_Comte ?association :city2 .
9 }
```

(b) Example of a SPARQL query with BGP_2

Fig. 4 Showing the example of the structure similar SPARQL query

To overcome this drawback of existing methods, we propose a query similarity metric considers both content-wise and structure similarity. Consider the two queries illustrated in the Fig. 4. Two queries Q_1 and Q_2 have a similar-structured if the ordering of their symbols is the same. To determine the similarity between two query patterns, we first compute the Levenshtein distance between their query patterns. Where the Line number 6, 7 and 8 shows the triple patterns exits in the query. Here, the most similar triple patterns can be determined by computing the minimum distance between the $\Delta(s_1, s_2)$, $\Delta(p_1, p_2)$, and $\Delta(o_1, o_2)$. The composition of the SPARQL query contains a number of different patterns. To find the similarity between queries, we need to decompose the SPARQL queries into subgraph patterns. The triple pattern distance is the minimum number of edit operations, such as addition, deletion, and insertion, to transform one graph to another. We introduce three functions AND, $UNION$ and $OPTIONAL$. These patterns take the input graph and decomposed it into three sets of non-empty patterns. As an example, consider the SPARQL query in Fig. 2 that contains the following graph patterns represented as Q_{AND}, $Q_{OPTIONAL}$, and Q_{UNION} and if no such triple patterns exist the result is $\emptyset$.

$$Q_{AND} = \left\{ \begin{array}{lll} ?philisopher1 & foaf:name & \text{“}AugusteComte\text{”} \; . \\ ?philisopher1 & ?relationshipWith & :Paris \quad . \end{array} \right\} \tag{1}$$

$$Q_{OPTIONAL} = \left\{ ?philisopher2 \quad foaf:givenName \quad \text{“}jean-Baptise\_say\text{”} \; . \right\} \tag{2}$$

$$Q_{UNION} = \{Q_{AND}, Q_{OPTIONAL}\} \tag{3}$$

More formally, we defined a $QueryDecomposition$ to deduce whether the decomposition of query patterns exits, as shown in Eq. (4).

$$QueryDecomposition := \begin{cases} \Theta_{UNION}(Q), & iff\ \Theta_{UNION}(Q) \neq \emptyset \\ \Theta_{OPTIONAL}(Q), & iff\ \Theta_{OPTIONAL}(Q) \neq \emptyset \\ \Theta_{AND}(Q), & else. \end{cases} \tag{4}$$

To calculate the similarity between two query patterns, we use the Levenshtein distance that is a string metric for assessing the difference between the two sequences. For example, the Levenshtein distance [9] of the two similar-structured queries is in the range of [0, 1]. The overall distance of the triple pattern is calculated by aggregating the individual score of the subjects, predicates, and objects. The general formula for the distance score is defined as:

$$Distancescore := \begin{cases} 0, \\ \frac{Levenshtein(Q_1, Q_2)}{max(stringlength(Q_1), stringlength(Q_2))} \\ 1, \end{cases} \tag{5}$$

By using this distance score in Eq. (5), we can determine the matching between the triples. Consider the triple matching between the two Basic Graph Patterns (BGP) as shown in Fig. 4. Here the most similar patterns for L_6, L_7, L_8 in BGP_1 are L_7, L_8, L_6 in BGP_2, respectively e.g., L represents the Line number correspond to Fig. 4a, b. For example, the minimum value for the edit distance is calculated as aggregating the individual distance score of subject, predicate and an object as follows: $\Delta(L6_{BGP1}, L7_{BGP2}) = (0 + 0 + \frac{4}{16}) = 0.75$. Complete matching is only possible in the case of bipartite graphs, where triples occur with the same number as for the triples patterns. Maximum matching can be determined in polynomial time. The matching of the triple is a computationally expensive process and existing approaches [31,37] do not consider the cost while performing the matching between two queries. In contrast, we use a cost threshold to cut off too expensive matching and utilize the classical algorithm called Hungarian Method [39] to solve the maximum matching of triples with minimum cost. This algorithm compute an optimal solution in a finite time. More specifically, we consider only contents of which the minimum matching cost does not exceed one. Thus, the maximum matching of the triples $\{(L6_{BGP1}, L7_{BGP2}), (L7_{BGP1}, L8_{BGP2})\}$ has a cost $\frac{0.75+\infty}{2}$, which shows that these BGP_1 and BGP_2 are unfit to match with each other.

3.2 Query prefetching

Similar queries occur frequently in real-world SPARQL query logs. This has also been reported previously [34]. The query prefetching approach is suitable for alleviating the burden on SPARQL endpoints by extracting the results of subsequent queries. In the common keyword-based search engines, the user is often not aware of the most suitable keyword to optimally extract information from the resource. In several iterations, the user is more likely to formulate their own keyword to find the correct answer. Similarly, in a LOD user might query for additional details based on the initial results, after making incremental changes the initial queries.

Definition 1 (*Query cluster*) To identify the similar-structured queries, we propose a query cluster. Consider $T_Q = \{Q_1, Q_2, \ldots, Q_n\}$ be the set of the SPARQL queries with corresponding query patterns $\{PQ_1, PQ_2, \ldots, PQ_n\}$. A query cluster is defined based on the pairwise matching with three constraints between the triple patterns such as $\Delta(s_i, s_j) \leq 1$, $\Delta(p_i, p_j) \leq 1$, and $\Delta(o_i, o_j) \leq 1$.

Given this definition, the query cluster only derived the query using parameter $\Delta_{max} = 0$ that can only be derived if the two queries are identical. Therefore, a query cluster consists of those query patterns that are structurally the same, based on the corresponding mapping $m \sqsubseteq \Theta\,(PQ_1) \times (PQ_2)$. To represent the query patterns as a

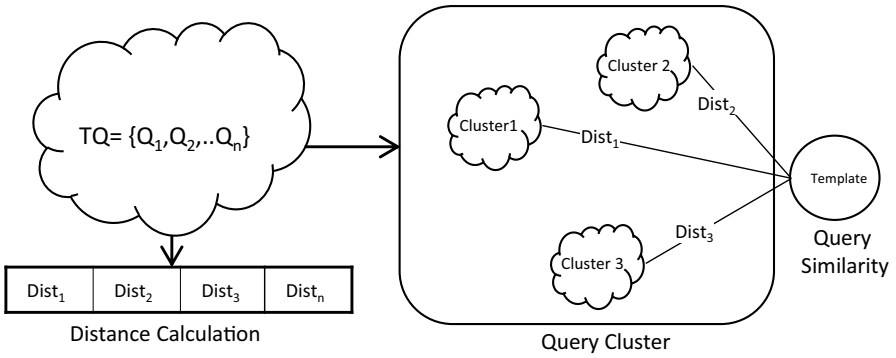

Fig. 5 Showing the example of query cluster of similar-structured queries

```
1  PREFIX : <http://dbpedia.org/resource/>
2  PREFIX dbo: <http://dbpedia.org/ontology/>
3  PREFIX foaf: <http://xmlns.com/foaf/0.1/>
4  SELECT ?p ?o ?birthPlace ?influence
5  WHERE {
6  :Auguste_Comte dbo:birthPlace ?birthplace .
7  ?birthplace dbo:country :France .
8  :Auguste_Comte dbo:influenceBy ?influence .
9  ?Auguste_Comte ?p ?o .
10 }
```

(a) Example of a SPARQL query preftech-ing

p	o
rdf:type	dbo:person
dbo:birthDate	1798-01-19
dbo:idea	:Positivism
dbo:influenced	:KarlMarx

(b) Showing the result of the query

Fig. 6 Showing the example of the SPARQL query prefetching

feature, we first cluster queries based on the content-wise and structural similarities as shown in Fig. 5 and distances between each pair of queries are computed by adopting the k-medoids algorithm [27]. We use this algorithm to cluster the training data of the query. This algorithm chooses the data points and allows us to utilize the distance function. To calculate the query distance, we utilize the distance score in Eq. (5) and define the similarity score of the query cluster as shown in the Eq. (6).

$$SimilarityScore(T_Q, Q_c) = \frac{1}{1 + Distancescore(T_Q, Q_c)} \tag{6}$$

Furthermore, we introduce an exploratory prefetching. More specifically, we identify a query cluster that previously issued queries belong to and construct a query template using all queries in the cluster. Then, we prefetch all contents that were used to answer the queries in the template since those queries are more likely to be requested by the same user in the future. This prefetching is completed by issuing one single query which includes all queries in the template. For example, we modified the content of the queries previously issued in Fig. 4 and retrieve all relevant contents that are useful for the future queries as shown in the Fig. 6. This query retrieves the additional information based on the central concept, instead of issuing the many

Algorithm 1: Central Concept Fetching (CCF)

Input : $T_Q = \{Q_1, Q_2, .., Q_n\}$
Output: Occurrence of most frequent subject

```
1  S.Count ← 0
2  foreach Q_p ∈ T condition do
3      S ← Θ(P_Qi)
4      while S ≠ 0 do
5          foreach Q_p ∈ S do
6              if Θ(P_Qi) > 1 then
7                  S ← S ∪ Θ(P_Qi)
8
9              else
10                 (S, P, O) ← Θ(P_Qi)
11
12             if S ∈ S.Count then
13                 S.Count.increasecount(S)
14             else
15                 S.Count.put(S,1)
16             end
17         end
18     end
19 end
20 return getHighestCount();
```

similar-structured queries, prefetching retrieve all the relevant information by issuing a single query.

For extracting the additional information for the specific resource, we propose an Algorithm 1 called central Concept Fetching (CCF) to generate the central concept from a query as shown in the Fig. 6a. In CCF algorithm, we first discovered the frequency of the subjects in all query patterns in Line 7 and aggregate whether the subject is already included in the $S.Count$. We further increase the count of the subject and add to $S.Count$ in Line 11 and analyzed all the triple patterns. This algorithm will analyze all the triple and give a good indication of a common theme in all the SPARQL queries.

3.3 Cache replacement

We propose an offline process for cache replacement to calculate the access frequency. Logging every record produces the most accurate result, however, it is computationally expensive. Existing approach [21] utilize forward scanning to identify record access with a time slice $[t_n, t_{n+1}]$.

However, this is not an efficient process and decreases the system performance, as it requires scanning and storage of the entire record. Moreover, the forward scanning approach requires a significant amount of time to classify the record. To optimize this process, we maintain partial records within a specific time period. We parallelize this task by splitting the logs into n consecutive periods and use a hash function to store the

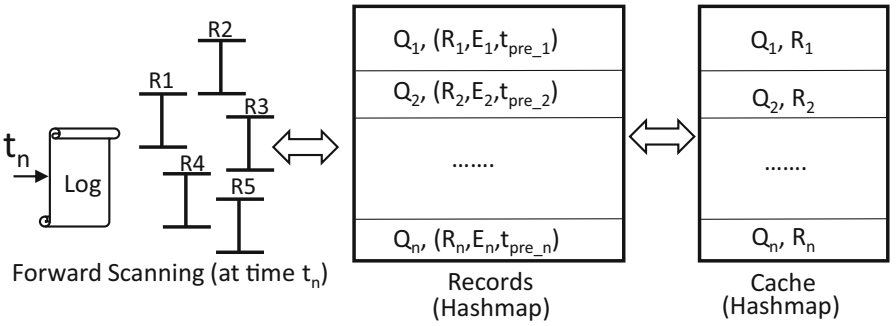

Fig. 7 Showing working example of the ACR algorithm maintaining cache and query access frequency

Algorithm 2: Adaptive Cache Replacement (ACR)

Input : Query Log Q, Job Scheduler H
Output: List of added cache triples

$t_{latest} \leftarrow max(LA_t, CA_t)$;
$t_{earliest} \leftarrow min(LA_t, CA_t)$;
$Records \leftarrow getRecords(t_{latest}, t_{earliest})$;
Function: $ModifiedForwardAlgo(Q, H)$;
if *newTriples in Records* **then**
    $max(estimation, cachedTripples)$;
    $Calculate(Frequency, LA_t)$;
    $update(Frequency, LA_t)$;
    $Remove = Leastaccessedtriples$;
else
    $Triples$ not in $Records$;
    $Calculate(Frequency, LA_t)$;
    $Add(newAccessTriples$;
return $addnewAccessTriples$;

frequency estimation for each query as shown in Fig. 7. Where Q_1 represents the query and R_1 denotes the results of the query. The estimation of the record is calculated by Eq. (7) and this algorithm ranks each query by its access frequencies. The storage of the access log is placed in a separate hash table. When each of the parallel executions finishes, the results of the most highly accessed frequencies are returned immediately and infrequently accessed queries are removed from the cache.

The overall flow of ACR is described in Algorithm 2, which explains the details of updating the cache by analyzing the access logs. The ACR algorithm takes previously access logs to calculate the access frequencies and provide the list of the updated cache triple, where LA_t represents the last access time of the triple and CA_t represents the current access time. ACR algorithm scans the records and updates the frequency. In the case of a cache miss, the algorithm first checks for the case of a record in the cache and updates the LA_t. Based on the access frequency ACR decides whether the new triples need to be added to the cache.

We have calculated the frequency of the data access using the exponential smoothing technique [11]. This method is widely utilized to predict economic data in financial

applications. The traditional approach [22] contains all the accessed queries in the cache. In our work, ACR serves each query according to the estimated frequency, the query with the highest frequencies are kept in the cache for future access. The general formula of exponential smoothing is as follows:

$$E_t = \alpha * x_t + (1 - \alpha) * E_{t-1} \tag{7}$$

As shown in Eq. (7), where E_t represents on exponentially moving average of access frequencies up to time t and x_t represents an access frequency observed at time t in discrete time with the smoothing constant $\alpha \in (0, 1)$. The high value of α gives significance to the new observations. By using the Eq. (7), we satisfies our requirement of selecting the highly accessed queries. We further modified Eq. (7) to represent the time of the last hit. In Eq. (8), t_{prev} represents the time of the last query hit and $X_{t_{prev}}$ represents the frequency estimate of the previous query at t_{prev}. For example, assume that $\alpha = 0.05$, $t = 12$, $t_{prev} = 3$, $x_t = 0.6$, and $x_{t_{prev}} = 0.5$. The value of E_t is calculated by $E_t = 0.05 * 0.6 + 0.05(1 - 0.05)^{12-3} * 0.5 = 0.046$.

$$E_t = \alpha x_t + \alpha(1 - \alpha)^{t - t_{prev}} x_{t_{prev}} \tag{8}$$

In case of the queries that are not similar to the previous ones stored in the cache, the result of these queries is served from the LOD and ACR algorithm store the access frequencies by using Eq. (8). When the cache becomes full, replacement is based on the access score; the top queries are kept in cache and less-frequent queries are removed from the cache.

4 Experimentation and results

This section is devoted to show the effectiveness of the proposed approach. We performed an evaluation on real-world datasets. The major goal of the experiment is to examine the hit rates and overhead comparison of the proposed approach with the current state-of-the-art cache replacement approaches.

4.1 Experimental setup

We conducted the experiments on an OpenLink Virtuoso Server 07.10 with a 4x AMD A8-7650K Radeon R7 graphics card, 64bit Ubuntu 16.04.2 LTS, and 32 GB of RAM. We utilized the *DBpedia3.6* and *Linked Geo Data (LGD)* query logs provided by the USEWOD 2014 challenge.[7] The query log contains a number of requests received by the SPARQL endpoint. The log is formatted in the form of the Apache common log format and contains the information about the query session that is used to retrieve the data from the endpoint. It is possible that in a single query session, two queries

[7] http://usewod.org/usewod2014.html.

Table 2 Showing the size of the query logs used in our evaluation

Source	Total queries	Valid queries	Unique queries
DBpedia 2013	28, 423, 201	27, 563, 105	12, 326, 855
DBpedia 2014	4, 132, 742	3, 708, 727	1, 517, 002
DBpedia 2015	31, 345, 875	30, 245, 552	3, 258, 671
LGD 2013	1, 721, 770	1, 512, 785	247, 731
LGD 2014	1, 730, 770	1, 513, 895	517, 530
Total	67, 354, 358	64, 544, 064	17, 867, 789

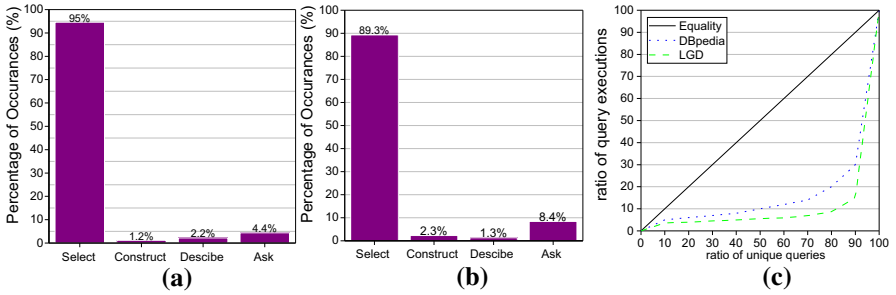

Fig. 8 Showing the patterns of the queries in **a** DBpedia and **b** LinkedGeoData, **c** the Lorenz curve for the impact of a unique query on query execution

are issued by the same user over time. The requests included in the *DBpedia3.6* query logs include the timestamp. The query log contains IP address, timestamp, query and userID. The valid queries were extracted from the query logs and the syntax of the query was checked according to the SPARQL1.1 specification.

The *DBpedia3.6* dataset contain the structure information extracted from the WIKIpedia and published over the LOD cloud. This dataset is obtained directly from the USEWOD query logs for DBpedia 2013, 2014 and 2015 as shown in the Table 2. The *DBpedia3.6* KBs contain 3.0M entities about the general knowledge. We first extracted the textual information from the log to get the previously issued queries then parse each query using *Apache Jena*.[8]

The *Linked Geo Data (LGD)* dataset holds the geographic sensor information mainly related to the OpenStreetMap and it is currently available as RDF format. We utilized the LGD 2013 and 2014 that consist of more than 10 billion triples. From the available LGD query logs, our evaluation contain repetitive and unique queries.

In both datasets, the majority of the queries are the SELECT queries in the DBpedia, and LinkedGeodata logs and within these SELECT queries, we identified the occurrences of BGPS, as in Fig. 8, which shows SELECT, CONSTRUCT, DESCRIBE and ASK. Most of the queries in both datasets are SELECT queries (95% in DBpedia and 89.3% in LinkedGeoData) and the most widely used features are ASK (4.4% in DBpedia and 8.4% in LinkedGeoData) followed by CONSTRUCT (1.2% in DBpedia and 2.3 % in LinkedGeoData).

[8] https://jena.apache.org/.

Figure 8c shows the impact of the unique query account for the query execution. Our aim is to ascertain the impact of the unique and frequently executed queries on the overall execution. We analyzed the execution using the DBpedia and LGD query logs. In DBpedia, 70% of the unique queries account for 30% of the overall executions, which shows that most of the execution instances involved the frequently accessed queries. Similarly, the impact of the unique queries on the overall execution is low, as almost 90% unique queries account for the 20% of the total query executions.

4.2 Performance evaluation

In this evaluation, we compare ACR with existing approaches, such as (LRU) Least Recently Used [16], (LFU) Least Frequently Used [18], and (SQC) SPARQL Query Caching [24] and measure the efficiency in terms of average hit rate and space overhead.

We evaluate the impact of existing cache replacement algorithms to improve the performance in terms of hit rates and overhead. Therefore, we compare ACR with three well-known cache replacement approaches (1) *LRU*: to replace the cache Least Recently Used (LRU) is applied to remove the items from the cache in order to provide space for the new item. This approach is simple to implement, especially when the objects are uniform. (2) *LFU*: with this method, the Least Frequently Used (LFU) resources are removed from the cache and the cache item is replaced with a new resource. However, LFU does not consider the size of the objects and CPU memory utilization. (3) *SQC*: SPARQL Query Caching (SQC) [24] improves the performance of triple stores by the selective invalidation of cache objects. This approach eliminates the cache objects that do not contain the predefined timestamp.

Figure 9a shows the hit rates achieved by the existing approaches. This experiment feeds the access logs of 3M to the ACR algorithm, whose job is to rank the access frequencies of the queries based on the exponential smoothing technique. It is noted that ACR outperforms existing approaches. However, the LFU technique remains accurate for a cache with a small size. The choice of the α effects the performance of the hit rate. We have set the value to 0.05 due to the higher accuracy of the results obtained, as the optimal value of α is almost certainly inversely proportional to the size of the cache, and perhaps related to the size of the database. If the cache is smaller, α should probably be larger as shown in Fig. 9b. Upon varying the size of the cache, our proposed approach outperform the other approaches, as shown in Fig. 9c. On average, our approach outperforms existing approaches in terms of higher hit rates, up to (80.65%).

Figure 10a depicts the space overhead used by the cache replacement algorithms for varying data set sizes. We measure the maximum space consumption of each approach based on the maximum number of records that each algorithm stores. It is observed that existing approaches consume more space to maintain the records. Figure 10b shows the time overhead average of the proposed ACR technique compared with state-of-the-art solutions. The existing solutions take a long time; on average the hit checking time of our approach takes (280 ms), which is almost 10 times better than other approaches.

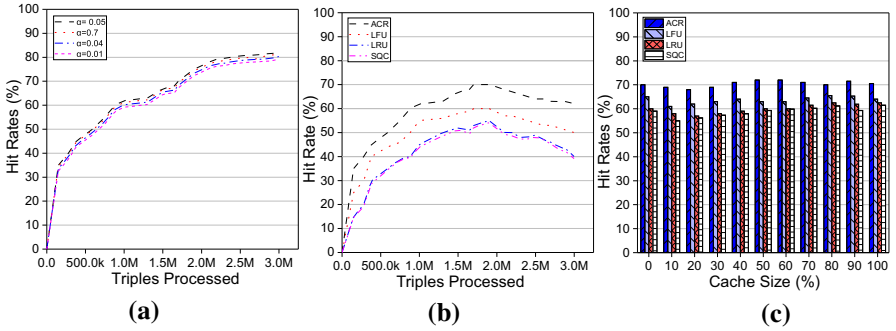

Fig. 9 Hit Rate achieved by ACR as compared to the LRU, LFU and SQC algorithms

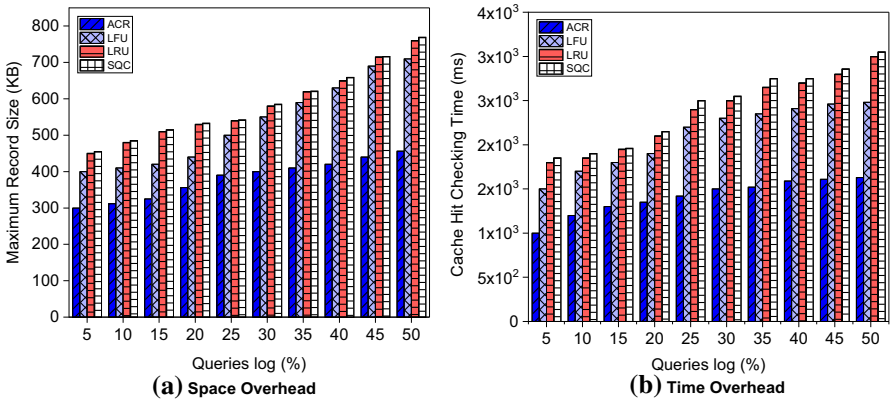

Fig. 10 Space and time overhead of existing as compared to ACR

5 Conclusion

In this paper, we proposed an Adaptive Cache Replacement (ACR) to improve SPARQL query processing on the LOD cloud. ACR algorithm parallelizes the task to calculate the access frequencies. To find similar queries, ACR utilizes the edit distance to identify clients similar querying patterns and place the frequently accessed queries in the cache to reduce the burden on SPARQL endpoints. Through experimental evaluation, we found that our approach outperforms the state-of-the-art approaches in terms of better query response time and less space overhead without losing the cache hit rate. This shows that on average, we achieve hit rates of 80.66%, which accelerates the querying speed by 6.34%. Specifically, our ACR technique is capable of classifying the access log with better space efficiency as compared to LFU, LRU, and SQC. In the future, we plan to investigate the effect of prefetching on system performance, this may lead to an improvement of ACR by parallelizing the algorithm to run on a separate machine as an offline process during system idle time.

Acknowledgements This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2017-0-01629) supervised by

the IITP (Institute for Information & communications Technology Promotion). This work was supported by the Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 2017-0-00655), NRF-2016K1A3A7A03951968 & NRF-2019R1A2C2090504.

References

1. Basu A (2019) Semantic web, ontology, and linked data. In: Web services: concepts, methodologies, tools, and applications, IGI Global, pp 127–148
2. Berners-Lee T, Hendler J, Lassila O (2001) The semantic web. Sci Am 284(5):34–43
3. Bizer C, Heath T, Berners-Lee T (2009) Linked data-the story so far. Int J Semant Web Inf Syst 5(3):1–22
4. Bollacker K, Evans C, Paritosh P, Sturge T, Taylor J (2008) Freebase: a collaboratively created graph database for structuring human knowledge. In: Proceedings of the 2008 ACM SIGMOD international conference on management of data, ACM, pp 1247–1250
5. Cho J, Garcia-Molina H (2003) Estimating frequency of change. ACM Trans Internet Technol 3(3):256–290
6. Chun S, Jung J, Lee KH (2019) Proactive policy for efficiently updating join views on continuous queries over data streams and linked data. IEEE Access 7:86226–86241
7. Dar S, Franklin MJ, Jonsson BT, Srivastava D, Tan M et al (1996) Semantic data caching and replacement. VLDB 96:330–341
8. Denning PJ (1968) The working set model for program behavior. Commun ACM 11(5):323–333
9. Dividino RQ, Gröner G (2013) Which of the following SPARQL queries are similar? why? In: LD4IE@ ISWC
10. Fernández JD, Umbrich J, Polleres A, Knuth M (2019) Evaluating query and storage strategies for RDF archives. Semant Web 10(2):247–291
11. Gardner ES Jr (2006) Exponential smoothing: the state of the art–part ii. Int J Forecast 22(4):637–666
12. Godfrey P, Gryz J (1999) Answering queries by semantic caches. In: International conference on database and expert systems applications, Springer, pp 485–498
13. Gottron T (2016) Measuring the accuracy of linked data indices. arXiv preprint arXiv:1603.06068
14. Gottron T, Knauf M, Scherp A (2015) Analysis of schema structures in the linked open data graph based on unique subject uris, pay-level domains, and vocabulary usage. Distrib Parallel Databases 33(4):515–553
15. Hasan R (2014) Predicting SPARQL query performance and explaining linked data. In: European semantic web conference, Springer, pp 795–805
16. Jelenković P, Radovanović A (2003) Optimizing the LRU algorithm for web caching. Charzinski J, Lehnert R, Tran-Gia P (eds) Teletraffic science and engineering, vol 5. Elsevier, pp 191–200, ISSN 1388–3437, ISBN 9780444514554
17. Konrath M, Gottron T, Staab S, Scherp A (2012) Schemex–efficient construction of a data catalogue by stream-based indexing of linked data. Web Semant Sci Serv Agents World Wide Web 16:52–58
18. Lee D, Choi J, Kim JH, Noh SH, Min SL, Cho Y, Kim CS (2001) LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies. IEEE Trans Comput 50(12):1352–1361
19. Lehmann J, Bühmann L (2011) Autosparql: let users query your knowledge base. In: Extended semantic web conference, Springer, pp 63–79
20. Lehmann J, Isele R, Jakob M, Jentzsch A, Kontokostas D, Mendes PN, Hellmann S, Morsey M, Van Kleef P, Auer S et al (2015) Dbpedia-a large-scale, multilingual knowledge base extracted from wikipedia. Semant Web 6(2):167–195
21. Levandoski JJ, Larson PÅ, Stoica R (2013) Identifying hot and cold data in main-memory databases. In: 2013 IEEE 29th international conference on data engineering (ICDE), IEEE, pp 26–37
22. Lorey J, Naumann F (2013) Caching and prefetching strategies for SPARQL queries. In: Extended semantic web conference, Springer, pp 46–65
23. Lorey J, Naumann F (2013) Detecting SPARQL query templates for data prefetching. In: Extended semantic web conference, Springer, pp 124–139
24. Martin M, Unbehauen J, Auer S (2010) Improving the performance of semantic web applications with SPARQL query caching. In: Extended semantic web conference, Springer, pp 304–318

25. Nishioka C, Scherp A (2017) Keeping linked open data caches up-to-date by predicting the life-time of RDF triples. In: Proceedings of the international conference on web intelligence, ACM, pp 73–80
26. Papailiou N, Tsoumakos D, Karras P, Koziris N (2015) Graph-aware, workload-adaptive SPARQL query caching. In: Proceedings of the 2015 ACM SIGMOD international conference on management of data, ACM, pp 1777–1792
27. Park HS, Jun CH (2009) A simple and fast algorithm for *k*-medoids clustering. Expert Syst Appl 36(2):3336–3341
28. Podlipnig S, Böszörmenyi L (2003) A survey of web cache replacement strategies. ACM Comput Surv 35(4):374–398
29. Ren Q, Dunham MH, Kumar V (2003) Semantic caching and query processing. IEEE Trans Knowl Data Eng 15(1):192–210
30. Sanfeliu A, Fu KS (1983) A distance measure between attributed relational graphs for pattern recognition. IEEE Trans Syst Man Cybern 3:353–362
31. Shu Y, Compton M, Müller H, Taylor K (2013) Towards content-aware SPARQL query caching for semantic web applications. In: International conference on web information systems engineering, Springer, pp 320–329
32. Suchanek FM, Kasneci G, Weikum G (2007) YAGO: a core of semantic knowledge. In: Proceedings of the 16th international conference on World Wide Web, ACM, pp 697–706
33. Umbrich J, Karnstedt M, Hogan A, Parreira JX (2012) Hybrid SPARQL queries: fresh versus fast results. In: International semantic web conference, Springer, pp 608–624
34. Yan L, Ma R, Li D, Cheng J (2017) RDF approximate queries based on semantic similarity. Computing 99(5):481–491
35. Yang M, Wu G (2011) Caching intermediate result of SPARQL queries. In: Proceedings of the 20th international conference companion on World wide web, ACM, pp 159–160
36. Zhang WE, Sheng QZ, Qin Y, Yao L, Shemshadi A, Taylor K (2016) SECF: Improving SPARQL querying performance with proactive fetching and caching. In: Proceedings of the 31st annual ACM symposium on applied computing, ACM, pp 362–367
37. Zhang WE, Sheng QZ, Taylor K, Qin Y (2015) Identifying and caching hot triples for efficient RDF query processing. In: International conference on database systems for advanced applications, Springer, pp 259–274
38. Zhang WE, Sheng QZ, Yao L, Taylor K, Shemshadi A, Qin Y (2018) A learning-based framework for improving querying on web interfaces of curated knowledge bases. ACM Trans Internet Technol 18(3):35
39. Zheng W, Zou L, Peng W, Yan X, Song S, Zhao D (2016) Semantic SPARQL similarity search over RDF knowledge graphs. Proc VLDB Endow 9(11):840–851

Affiliations

Usman Akhtar[1] · Anita Sant'Anna[2] · Chang-Ho Jihn[3] · Muhammad Asif Razzaq[1] · Jaehun Bang[1] · Sungyoung Lee[1]

✉ Jaehun Bang
jhb@oslab.khu.ac.kr

✉ Sungyoung Lee
sylee@oslab.khu.ac.kr

Usman Akhtar
usman@oslab.khu.ac.kr

Anita Sant'Anna
anita@viniam.se

Chang-Ho Jihn
jihn@khu.ac.kr

Muhammad Asif Razzaq
asif.razzaq@oslab.khu.ac.kr

1 Department of Computer Science and Engineering, Kyung Hee University, Seocheon-dong, Giheung-gu, Yongin-si, Gyeonggi-do 446-701, Republic of Korea

2 Viniam Consulting AB, Halmstad, Sweden

3 Department of Industrial and Management System Engineering, Kyung Hee University, Yongin-si, South Korea