SECF: Improving SPARQL Querying Performance with Proactive Fetching and Caching

Wei Emma Zhang, Quan Z. Sheng,
Yongrui Qin, Lina Yao, Ali Shemshadi
School of Computer Science
The University of Adelaide
Adelaide, SA 5005, Australia
wei.zhang01@adelaide.edu.au

Kerry Taylor
CS&IT Building 108
Australian National University
Canberra, ACT 2601, Australia
kerry.taylor@csiro.au

ABSTRACT

Querying on SPARQL endpoints may be unsatisfactory due to high latency of connections to the endpoints. Caching is an important way to accelerate the query response speed. In this paper, we propose SPARQL Endpoint Caching Framework (SECF), a client-side caching framework for this purpose. In particular, we prefetch and cache the results of similar queries to recently cached query aiming to improve the overall querying performance. The similarity between queries are calculated via an improved Graph Edit Distance (GED) function. We also adapt a smoothing method to implement the cache replacement. The empirical evaluations on real world queries show that our approach has great potential to enhance the cache hit rate and accelerate the querying speed on SPARQL endpoints.

CCS Concepts

•**Information systems** → **Database management system engines;** *Database query processing;*

Keywords

Caching, SPARQL, Query Suggestion, MSES

1. INTRODUCTION

Consuming structured data is a promising way to improve the effectiveness of search. RDF is widely accepted for modelling information in a structured way. It connects a *subject* to an *object* with a *predicate*, a labelled edge. SPARQL is the query language to retrieve information formatted in RDF from triple stores. SPARQL Endpoints are interfaces that enable users to query these publicly accessible knowledge bases. As the SPARQL 1.1 specification introduces the SERVICE keyword, federated queries can be realized by using SERVICE to access data offered by other SPARQL endpoints. However, network instability and latency affects the query efficiency. Therefore, the most typical way for consumers who want to query public data is to download data dump and set up their own local SPRAQL Endpoint. But data in a local endpoint is not up-to-date and hosting an endpoint requires expensive infrastructural support.

Many research efforts have been dedicated to circumvent this problem [9, 15, 8, 14] and caching is one of the popular directions [12]. While most research efforts focus on providing a server-side caching mechanism, being embedded in triple stores, client-side caching has not been fully explored [9]. In this paper, we propose a domain-independent client-side caching framework SECF for SPARQL endpoints to facilitate the query answering process. Our approach is based on the observation that end users who consume RDF-modelled knowledge typically use programmatic query clients to retrieve information from SPARQL endpoints [8]. These queries usually have same query patterns and only differ in specific attributes of triple patterns. Moreover, they are usually issued subsequently. Figure 1 shows example of two similar queries. Query 1 retrieves career start year from the actors of the movie *Rain Man* and the year should be later than 1980. Query 2 requests the same information for movie (*Eyes Wide Shut*). The differences between these two queries are the movie name (the underlined terms) and the year in Filter expression. We call the different attribute (movie name here) the *replacable attribute*. Thus, the similar queries are defined in our paper as queries with same pattern and different *replacable attributes* in their triple patterns.

By considering these observations, we propose a caching mechanism that is based on proactive fetching (i.e., prefetching) the query results of similar queries in advance. Since these similar queries are potentially subsequent queries, the cached results can be returned immediately rather than being retrieved from SPARQL endpoints if cache hit. Thus, the average query response time will be reduced.

The key challenge to improve the hit rate centers on how to effectively obtain similar queries that possibly be requested subsequently. We look into this issue and utilize a cluster-

SAC 2016, April 04-08, 2016, Pisa, Italy

```
Query 1
SELECT ?actor ?year WHERE {
:Rain_Man dbpedia-owl:starring ?actor .
?actor dbpedia-owl:activeYearsStartYear ?year .
}
FILTER(?year>1980)

Query 2:
SELECT ?actor ?year WHERE {
:Eyes_Wide_Shut dbpedia-owl:starring ?actor .
?actor dbpedia-owl:activeYearsStartYear ?year .
}
FILTER(?year > 1960)
```

Figure 1: Example Queries

ing algorithm to suggest similar queries. Firstly, we transform queries into vector representations by leveraging the idea in [4]. We improve their work by not only considering Basic Graph Patterns (BGPs), but also take other feature keywords e.g., FILTER, BIND and VALUES, into consideration. Secondly, we train a modified *K Nearest Neighbor* (KNN) model [4], essentially a clustering algorithm, using these feature vectors. Moreover, we adopt the Principal Component Analysis (PCA) [5] to project original queries to a lower dimensional subspace to accelerate the nearest neighbor calculations. In the last step, the trained model suggests similar queries. Our algorithm prefetches the results of these queries and caches the (query, results) pairs. The suggestion process runs in a background thread to the query process. The training and mining process can be performed only once as a pre-computing step.

We also introduce cache replacement algorithm because of the limited cache size. However, techniques for relational databases ([10]) cannot be directly applied into our caching framework because our caching is record based, rather than traditional page-based caching algorithms. Moreover, our client-side application is not based on RDBMS and is not designed for server side as traditional caching algorithms do. In this paper, we use a time-aware frequency based algorithm evaluated in our previous work [16]. More specifically, we use Modified Simple Exponential Smoothing (MSES) to evaluate the frequencies of cached queries and remove the ones with the lowest scores from the cache.

Our main contributions are summarized as follows:

- We address the problem of providing client-side caching for accelerating query answering process for SPARQL endpoints and design a caching mechanism that can either be deployed as a web browser plugin or be embedded in the firewall. We envisage ultimately it being embedded within SPARQL endpoints that act as clients to other SPARQL endpoints by interpreting the SERVICE keyword for SPARQL 1.1 federated queries.
- Our approach suggests similar queries by leveraging machine learning techniques. We combine GED and functions for special feature keywords to measure the similarity between SPARQL queries. KNN and PCA are utilized to learn similar queries. We also adopt a smoothing method in cache replacement.
- We perform extensive experiments on real world queries. The empirical results show that our approach has great potential to accelerate the querying speed on SPARQL endpoints.

The remainder of this paper is structured as follows. We present some backgrounds in Section 2. In Section 3, our methodology is introduced. The experimental results are reported in Section 4. Finally, we overview the related work in Section 5 and conclude this paper in Section 6.

2. PRELIMINARIES

The official syntax of SPARQL1.1 considers operators OPTIONAL, UNION, FILTER, SELECT and concatenation via a dot symbol (.) to GROUP patterns. VALUES and BIND are to define sets of variable bindings. We use B, I, L, V for denoting the (infinite) sets of blank nodes, IRIs, literals, and variables. A SPARQL graph pattern expression is defined recursively as follows [11]:

(i) A valid triple pattern $T \in (IVB) \times (IV) \times (IVLB)$ is a graph pattern,

(ii) If P_1 and P_2 are graph patterns, then expressions (P_1 AND P_2), (P_1 UNION P_2) and (P_1 OPTIONAL P_2) are graph patterns,

(iii) If P is a graph pattern and R is a SPARQL build-in condition, then the expression (P FILTER R) is a graph pattern.

A BGP is a graph pattern when it is represented by the conjunction of multiple triple patterns. Let $Q = (S_Q, P_Q)$ be the query where S_Q is the SELECT expression and $P_Q = P_1 \oplus ... \oplus P_n$ is the query pattern with $\oplus \in$ {AND, UNION, OPTIONAL, FILTER, BIND, VALUES, MINUS}. When pattern feature $\oplus \in$ {AND, UNION, OPTIONAL, MINUS}, graph pattern $P_i, i \in [1, n]$ can be recursively decomposed to sub-level graph patterns until the graph pattern is a BGP which can further be decomposed to triple patterns as $P_{bgp,i} = T_1 \oplus ... \oplus T_k$, where $\oplus$ = AND. When pattern feature $\oplus \in$ {FILTER, BIND, VALUES}, graph pattern P_i cannot be decomposed to BGPs and is represented as expressions. It is easy to observe that query Q can also be represented as $Q = (S_Q, \{P_{bgp}, P_{filter}, P_{bind}, P_{value}\})$ where P_{bgp}, P_{filter}, P_{bind}, P_{value} are BGP, FILTER, BIND and VALUE patterns in P_Q respectively. Note that each graph pattern can appear multiple times in a query pattern.

3. THE SECF METHODOLOGY

Figure 2 illustrates the process of SECF. When a new query is issued, SECF first checks if an identical query has been cached. In this case, the results are returned immediately. Otherwise, the process moves on. If query recording is enabled, a background process will log all queries by this user into a file for further learning processing. When query suggestion is enabled, during runtime, similar queries are suggested for the current query. The results of these queries will be retrieved from the SPARQL Endpoint in advance and cached in the form of (query, result) pairs ((q_i, r_i) in Figure 2). At the same time, current query results are returned to

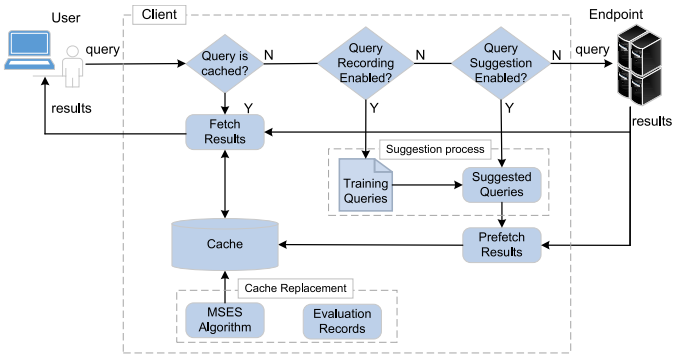


Figure 2: SECF

the client. A cache replacement algorithm is executed when the cache is full. It runs in a separate thread so that it does not affect the query answering process.

In this section, we discuss our method to measure similarity between queries (Section 3.1) and how to mine historical queries based on the similarity function to suggest similarly-structured queries (Section 3.2). We also introduce a cache replacement algorithm for cache update (Section 3.3).

3.1 Query Similarity

To find similar queries, we compute the distance between two given queries by calculating the distance between patterns of the two queries:

$$d(P_Q, P_Q^{'}) = d(P_{bgp}, P_{bgp}^{'}) + d(P_{filter}, P_{filter}^{'}) + d(P_{bind}, P_{bind}^{'}) + d(P_{value}, P_{value}^{'}) \quad (1)$$

Where P_Q contains $P_{bgp}, P_{filter}, P_{bind}, P_{value}$ and $P_Q^{'}$ contains $P_{bgp}^{'}, P_{filter}^{'}, P_{bind}^{'}, P_{value}^{'}$. $d(P_Q, P_Q^{'}) = 0$ denotes the two queries are structurally the same.

We calculate $d(P_{bgp}, P_{bgp}^{'})$ using GED [13] since SPARQL queries are graph-structured. We adopt the idea to build graphs from SPARQL query pattern in [4] and use a suboptimal solution integrated in the Graph Matching Toolkit[1] to compute $d(P_{bgp}, P_{bgp}^{'})$. Because the graph of BGPs in the two example queries are the same, their BGP distance (i.e., GED) equals 0, namely $d(P_{bgp_1}, P_{bgp_2}) = 0$ in this case.

When $d(P_{bgp}, P_{bgp}^{'}) = 0$, we further calculate $d(P_{filter}, P_{filter}^{'})$, $d(P_{bind}, P_{bind}^{'})$ and $d(P_{value}, P_{value}^{'})$. We define distance between two FILTER expressions as half of their *levenshtein* distance when the variables in these two expressions are identical, otherwise the distance is a fixed value 1. Thus the distance is in the range of $[0, 0.5]$ or equals to 1.

$$d(P_{filter,i}, P_{filter,i}^{'}) = \begin{cases} \frac{levenshtein(E(i), E^{'}(i))}{2max(length(E(i)), length(E^{'}(i)))}, \\ \qquad if \quad V(i) = V^{'}(i) \\ 1, \qquad else \end{cases} \quad (2)$$

where $E(i)$ and $E^{'}(i)$ represent the FILTER expression for $P_{filter,i}$ and $P_{filter,i}^{'}$. $V(i)$ and $V^{'}(i)$ are variables in these two FILTER patterns respectively. When there are multiple Filter expressions that can be compared, the total difference is defined as:

$$d(P_{filter}, P_{filter}^{'}) = \sum_{i=1}^{m} d(P_{filter,i}, P_{filter,i}^{'}) \quad (3)$$

We can also have similar functions for BIND and VALUE patterns. Filter expressions in Query 1 and Query 2 are similar as the distance is 0.05 using Equation 3. So $d(P_{Q1}, P_{Q2}^{'}) = 0.05$ (Equation 1).

3.2 Prefetch and Cache Similar Queries

We identify the most similar queries by mining historical queries. When a new query comes, we prefetch the results of its similar queries, and cache them along with its results. Two approaches are developed for this step.

3.2.1 Basic Approach

Basic approach includes three steps:

- We construct the feature vectors for training queries by adopting *k-medoids* algorithm [6] and applying Function 1. Specifically, we cluster training queries and obtain the center query of each cluster. Then we calculate the distance between a query and each center query, and use the distance as a dimension of the feature vector of this query. The number of clusters equals to the number of dimensions of feature vectors.
- We then train a modified KNN [4] to suggest similar queries. Euclidean distance is used in KNN to measure distance between feature vectors.
- When a new query is issued, we choose the K nearest neighbors obtained from Step 2 as suggested queries. We prefetch the results of these queries and put the (q_i, r_i) pairs into the cache during the caching process.

3.2.2 Improved Approach with PCA

To further reduce the suggestion time, we employ PCA to preprocess the queries. Given training queries $\{\mathbf{q}_{i=1}^{n}\} \in \mathbb{R}^m$, PCA calculates a linear transformation $\mathbf{q}_i \rightarrow \mathbf{L}\mathbf{q}_i$, which maps the original data to a variance-maximizing subspace. The variance of the projected data can be computed using the covariance matrix:

$$\mathbf{C} = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{q}_i - \mu)^T(\mathbf{q}_i - \mu) \quad (4)$$

where $\mu = \frac{1}{n}\sum_i^n \mathbf{q}_i$ is the mean value of all queries. To this point, the objective of PCA is to find an optimal linear transformation $\mathbf{L}$, which can maximize the variance of projected data. The objective function is formulated as:

$$\max_{\mathbf{L}} Tr(\mathbf{L}^T\mathbf{C}\mathbf{L}), \quad s.t. \quad \mathbf{L}\mathbf{L}^T = \mathbf{I} \quad (5)$$

thus, the top $T \leq m$ eigenvectors of covariance matrix $\mathbf{C}$ would be the optimal solutions of Equation 5.

The only difference compared to the basic approach is in the first step, constructing feature vectors. We use the same

[1]http://www.fhnw.ch/wirtschaft/iwi/gmt

clustering and similarity algorithms to construct basic feature vectors for training queries, but we further reduce dimensions by using PCA and generate multiple feature vectors. For example, if we use 20 clusters in the basic approach, the feature vector for a query is a 20-dimensional vector. Here by using PCA, we generate 19 feature vectors, with dimensions from 1 to 19, for this query separately. We choose one of these feature vectors as inputs of Step 2. In Section 4.3, we compare the performance using feature vectors of different dimensions.

3.3 Cache Replacement

The key point of cache replacement algorithm is to identify the most useful data. One way is to find most frequent used data. In [16], we use Modified Simple Exponential Smoothing (MSES) to evaluate the frequency of accessed triples and prove its simplicity and accuracy. In this paper, we use MSES (Equation 6) to estimate cached queries.

$$E_t = \alpha + E_{t_{prev}} * (1-\alpha)^{t_{prev}-t} \quad (6)$$

where E_t stands for smoothed observation of time t. t_{prev} represents the time when the query is last observed and $E_{t_{prev}}$ denotes the previous frequency estimation for the query at t_{prev}. α is a constant with $\alpha \in (0,1)$. From this equation, it is easy to observe that MSES assigns exponentially decreasing weights as the observation becomes older, which meets the requirement of caching the most frequently and recently issued queries.

We perform cache replacement based on the estimation score calculated by MSES. Each time a new query is executed, we examine the frequency of this query using MSES. If it is in the cache, we update the estimate for it. Otherwise, we just record the new estimate. We keep the estimate records for all processed queries if query recording is enabled. When the top H estimations are changed, the cache will be updated to reflect the new top H queries. Lower rank queries will be removed from the cache.

4. EVALUATION

4.1 Setup

We set up local SPARQL Endpoint using Virtuoso 7.2 as backing system, installed on Ubuntu 14.04 64 bit, 32GB RAM. All the learning algorithms are performed on a PC with Windows 7 64 bit, 8GB RAM and 2.40GHZ Intel i7-3630QM CPU using Java SE7 and Apache Jena-2.11.2.

4.2 Datasets

We evaluated our approach using real world queries provided by USEWOD 2014 challenge[2]. We focus on SELECT queries and extracted 198,235 valid queries from the query logs of DBPedia's SPARQL endpoint[3] (DBpedia3.9) and 1,790,047 valid queries from the query logs in Linked Geo Data's endpoint[4] (LinkedGeoData). Within the SELECT queries, except for patterns which can be finally decomposed to BGPs (e.g., AND, UNION, OPTIONAL and MINUS), FILTER, VALUES and BIND are used, especially for FILTER, which occurs in 83.97% DBpedia3.9 queries and 50.72% LinkedGeoData queries(Table 1). This actually provides a strong evidence that FILTER expressions should not be ignored when calculating similarity between queries.

[2]http://usewod.org
[3]http://dbpedia.org/sparql/
[4]http://linkedgeodata.org/sparql

Table 1: Selected Keywords from SELECT Queries

	FILTER	VALUES	BIND
DBpedia3.9	166,465 (83.97%)	1,615 (0.81%)	123 (0.06%)
LinkedGeoData	907,963 (50.72%)	101 (0.005%)	1 (0.0005%)

4.3 Experiments

The experiments aim to verify the effectiveness of our approach by comparing average hit rate, average query time and space usage. We evaluated the impact of K in KNN and the performance under the scenarios of applying and without applying suggestion/prefetching (Section 4.3.1). We investigated the impact on performance when using PCA to reduce the dimension of graph feature vectors (Section 4.3.2). We also compared our work with the Adaptive SPARQL Query Cache (ASQC) introduced in [9] (Section 4.3.3). In order to measure the impact on SPARQL Endpoint server, we monitored server workload and give our observation (Section4.3.4).

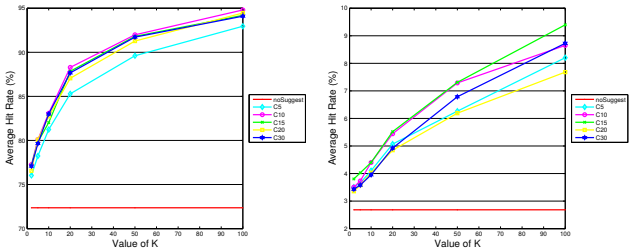
We randomly chose 21,600 training queries and 5,400 test queries from the two query sets separately. The cache replacement algorithm we used in all test cases is MSES (Section 3.3). We chose $\alpha = 0.05$ according to our previous experience [16]. Because the larger size of cache, the higher hit rate would achieve, we only show experiment results when the number of queries in cache is set to 1,000.

4.3.1 Impact of Cluster Number

As the generation of SPARQL feature vectors depends on the number of clusters, we compared the impact of different number of clusters on the average hit rates. Figure 3 shows the hit rates with different number of clusters (i.e., 5, 10, 15, 20, 30) and different K in KNN (i.e., 2, 5, 10, 20, 50, 100 as K). Although the hit rates on DBpedia3.9 queries change slightly when using the same K with different clusters, we can find that Cluster 10 (C10 for short thereafter) gives highest performances from 77.18% (K=2) to 94.18% (K=100). For LinkedGeoData queries, the highest hit rate is achieved by C15 from 38.03% (K=2) to 20.60% (K=100). In the following experiments, we used C10 for DBpedia3.9 query and C15 for LinkedGeoData for feature vector generation. Moreover, Figure 3 shows that the hit rates performed without suggestion is always lower than the ones with suggestions because cached similarly-structured queries contribute to the improvement of hit rates.

4.3.2 Impact of Dimension Reduction

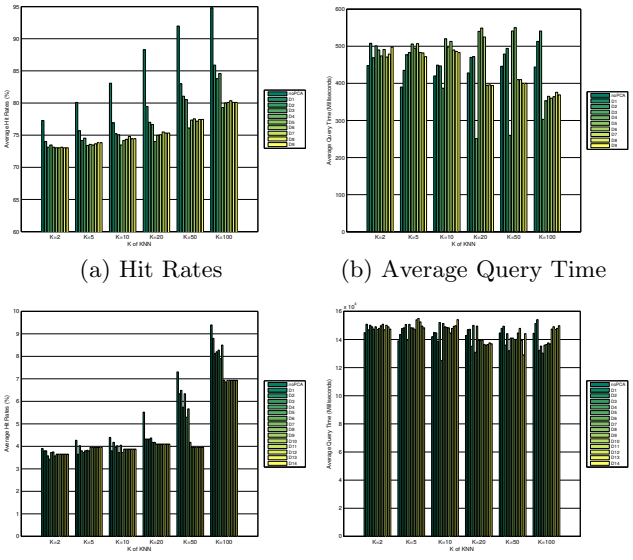
Dimensions of the feature vectors also affect the hit rate. We generated feature files with different dimensions for the two sets of queries, namely, files from Dimension 1 (D1) to D9 for DBpedia3.9 with 10 clusters (C10) and D1 to D14 for LinkedGeoData with 15 clusters (C15). We then trained



(a) DBPedia3.9 (b) LinkedGeoData

Figure 3: Hit Rates with Different Clusters and K

KNN model with these files. We compared the average hit rates and average query time between with and without using PCA to reduce the dimension of feature vectors. From



(a) Hit Rates (b) Average Query Time

(c) Hit Rates (d) Average Query Time

Figure 4: Performance Comparisons on Dimensional Reduction. (a) and (b) are for DBpedia3.9, (c) and (d) are for LinkedGeoData

Figure 4(a) and 4(c), it is evident that without dimension reduction, hit rates are higher for both query sets. The highest hit rate achieved for DBpedia3.9 queries is 93.22% when K=100 without dimension reduction. For LinkedGeoData, the highest hit rate is 39.03%. Usually, the higher the hit rate is, the less query time it takes. However, when taking K and dimension in PCA into consideration, this rule does not work. This is because it will take more time to compute when K and dimension increase, which counteract the time reduced by higher hit rate. In our experiment, we found out that for DBpedia3.9 queries, when K=20, D3 gives the least average query time, 251 ms. For LinkedGeoData, when K=10, D5 takes least time, 1251 ms on average. In following experiments, we will use parameters based on these results.

4.3.3 *Performance Comparison with ASQC*

In this experiment, we compared the average hit rate, average query time and space usage between our work, ASQC and no cache is used. For comparison, we modified the code of ASQC [5] to access our datasets and use MSES for cache replacement. DBpedia3.9 queries are used in this evaluation. Parameters are set to K=20, D=3 according to previous experiment. Table 2 presents the results. ASQC have slightly lower hit rate (72.63%) than SECF (76.26%). ASQC takes 264 ms in average for querying and SECF takes 251 ms. When no cache is implemented, the average query time increases to 625 ms. We did not include prefetching time as it is in separate thread. Space consumption evaluates how much memory the cache uses. In SECF, the total usage (before slash) for caching 1000 queries, as shown in Table 2, includes cached queries and answers as well as the estimation records for cache replacement (after slash). The numbers indicate that most space are consumed by cached (query, result) pairs.

Table 2: Performance Comparison

	ASQC	No Cache	SECF
Hit	72.63%	NA	76.65%
AvgTime	264ms	625ms	251ms
Space	7.15MB/0.45KB	NA	7.15MB

4.3.4 *Server Overhead*

In order to evaluate the impact of cache on the endpoint server, we monitored the memory and CPU usage as well as I/O on the server. We captured the usage every 20 seconds until the querying ends. Table 3 shows the average free memory (AvgFreeMem), average I/O (AvgIO) and average CPU time (AvgCPU) including system CPU and user CPU time. We only present the results on querying DBpedia3.9 dataset due to limit space. From the results we find out that SECF and ASQC cause higher computation overhead and memory usage on server compared to querying without cache and ASQC performs slightly better than SECF with more free memory (217.87MB vs 203.74MB), less I/O (11.49 vs 21.84) and less CPU time (9.37ms vs 10.68ms). It is because that SECF requires prefetching results for similar queries from server which leads to additional overhead.

Table 3: Server Performance

	ASQC	No Cache	SECF
AvgFreeMem	217.87MB	224.30MB	203.74MB
AvgIO	11.49	7.72	21.84
AvgCPU	9.37ms	10.09ms	10.68ms

5. RELATED WORK

In this section, we overview recent works in *Semantic Caching* and *Query Suggestion* that are related to our work.

Semantic Caching. Semantic Caching scheme involves techniques that keep previously fetched data for past queries. It was originally developed for relational databases [2]. In recent years, semantic caching technique is extended to triple

[5]http://wiki.aksw.org/Projects/QueryCache

stores that managing SPARQL queries. The work of Martin et al. [9] is the first step towards semantic caching in triple stores. It essentially builds a proxy layer between an application and a SPARQL endpoint. The proxy layer caches the query-result pairs. Yang and Wu [15] provide an approach that caches intermediate result of BGPs in SPARQL queries. More recently, Lorey and Naumann [8] propose approaches that generate queries with templates, then prefetch and cache the results of these queries. We draw on the notion of prefetching from this work. The Linked Data Fragments (LDF) approach [14] aiming at improving data availability can also be regarded as caching technique as it caches fragments of queryable data from servers that can be accessed by client. So that each client is able to process SPARQL queries on replicated fragments cached from servers.

Query Suggestion. Query Suggestion is an interactive approach used in search engines to better understand users' information needs. It plays an important role in improving the accuracy of searching. Query suggestions are usually made by mining query logs and session records of web users' searching history [1]. Researchers recently introduced these mining techniques into SPARQL processing. Lehmann et al. [7] propose a supervised machine learning framework to suggest SPARQL queries based on examples previously selected by users. No prior knowledge of the underlying schema or the SPARQL query language is required. Very recently, Hasan [4] uses a machine learning method to predict the performance of SPARQL query performance. We draw idea from it to build feature vectors for SPARQL queries.

6. CONCLUSION AND FUTURE WORK

In this paper, we have addressed the challenges of improving the querying performance of SPARQL endpoints. The proposed similarity measurement and mining based approach are evaluated effective. Dimensional reduction algorithm PCA also contributes to the reduction of overall query time. In the future, we plan to train larger query sets and further improve the query performance by reducing space overhead. We also plan to investigate ways to understand client intentions for more accurate suggestions of similar queries.

7. ACKNOWLEDGMENTS

This research has support from the Commonwealth Scientific and Industrial Research Organization (CSIRO), Australia (Top-up PhD Scholarship OCEPhD13/03446).

8. REFERENCES

[1] H. Cao, D. Jiang, J. Pei, Q. He, Z. Liao, E. Chen, and H. Li. Context-aware Query Suggestion by Mining Click-through and Session Data. In *Proc. of the 14th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD 2008)*, pages 875–883, Las Vegas, Nevada, USA, August 2008.

[2] S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan. Semantic Data Caching and Replacement. In *Proc. of the 22rd International Conference on Very Large Data Bases (VLDB1996)*, pages 330–341, Bombay, India, September 1996.

[3] E. S. Gardner. Exponential Smoothing: The State of The Art–Part II. *International Journal of Forecasting*, 22(4):637–666, 2006.

[4] R. Hasan. Predicting SPARQL Query Performance and Explaining Linked Data. In *Proc. of the 11th Extended Semantic Web Conference (ESWC 2014)*, pages 795–805, Anissaras, Crete, Greece, May 2014.

[5] I. Jolliffe. *Principal component analysis*. Wiley Online Library, 2005.

[6] L. Kaufman and P. Rousseeuw. *Clustering by Means of Medoids*. Dodge, Y. (ed.) Statistical Data Analysis based on the L1 Norm, 1987.

[7] J. Lehmann and L. Bühmann. AutoSPARQL: Let Users Query Your Knowledge Base. In *Proc. of the 8th Extended Semantic Web Conference (ESWC 2011)*, pages 63–79, Heraklion, Crete, Greece, May 2011.

[8] J. Lorey and F. Naumann. Detecting SPARQL Query Templates for Data Prefetching. In *Proc. of the 10th Extended Semantic Web Conference (ESWC 2013)*, pages 124–139, Montpellier, France, May 2013.

[9] M. Martin, J. Unbehauen, and S. Auer. Improving the Performance of Semantic Web Applications with SPARQL Query Caching. In *Proc. of the 7th Extended Semantic Web Conference (ESWC 2010)*, pages 304–318, Heraklion, Crete, Greece, 2010.

[10] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K Page Replacement Algorithm For Database Disk Buffering. In *Proc. of the International Conference on Management of Data (SIGMOD 1993)*, pages 297–306, Washington, D.C., USA, May 1993.

[11] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and Complexity of SPARQL. *ACM Transactions on Database Systems*, 34(3), 2009.

[12] Q. Ren, M. H. Dunham, and V. Kumar. Semantic Caching and Query Processing. *IEEE Transactions on Knowledge and Data Engineering*, 15(1):192–210, 2003.

[13] A. Sanfeliu and K. Fu. A Distance Measure between Attributed Relational Graphs for Pattern Recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, 13(3):353–362, 1983.

[14] R. Verborgh, O. Hartig, B. D. Meester, G. Haesendonck, L. D. Vocht, M. V. Sande, R. Cyganiak, P. Colpaert, E. Mannens, and R. V. de Walle. Querying Datasets on the Web with High Availability. In *Proc. of the 13th International Semantic Web Conference (ISWC 2014)*, pages 180–196, Riva del Garda, Italy, October 2014.

[15] M. Yang and G. Wu. Caching Intermediate Result of SPARQL Queries. In *Proc. of the 20th International World Wide Web Conference (WWW 2011)*, pages 159–160, Hyderabad, India, March 2011.

[16] W. E. Zhang, Q. Z. Sheng, K. Taylor, and Y. Qin. Identifying and Caching Hot Triples for Efficient RDF Query Processing. In *Proc. of the 20th International Conference on Database Systems for Advanced Applications (DASFAA 2015)*, pages 259–274, Hanoi, Vietnam, April 2015.