A SPARQL client side caching system

Sri Harsha Anand Pushkala, Hari Krishn Gupta
Department of Computer Science, University of Southern California,
Los Angeles, United States
{ anandpus, harikgup}@usc.edu

Abstract. In semantic web, querying for data on online triple stores and SPARQL endpoints tends to be unreliable due to issues such as limited bandwidth or low availability. There have been a few attempts to develop a caching system but most of them focus on server side caching which does not address the problem of limited bandwidth. In this paper, we propose a novel approach of using client side cache which reduces the problem of low availability and decreases the response rate of online endpoints. This paper further proposes the usage of an algorithm called "GQR" to check for containment of conjunctive queries. This allows for faster querying of data stored. Also we propose a methodology to automatically update the data stored. Finally, we discuss a set of benchmark queries to evaluate the developed system.

1 Introduction

As per the current research done in the field, triple stores do not perform nearly as fast as relational database in terms of query processing speed, indexing or data retrieval [Michael Martin et. al ,2010]. The performance of online SPARQL endpoint for these triple stores is an even more time-consuming task, because of the network latency involved. Also, there are additional issues that affect these endpoints, such as low fault tolerance.

However the availability issue and the low bandwidth can be removed by making an offline copy of the data. But the offline copies of data pose additional issues which are as follows:

- Offline copied data becomes stale, hence needs periodic updates.
- Redundancy in data on a client machine and would require enormous space.

Another key insight which has seldom been explored by researchers is the user's affinity to perform repetitive SPARQL queries and their inclination to be focused on a particular domain. If this insight is leveraged, new queries can be first checked against previously performed queries and a union of these previously retrieved data can be used for the new query instead of performing a query on the SPARQL endpoint.

In order to address the issues, we postulate a caching system which can perform containment checks on the user query before retrieval. Therefore, if the stored data is available as data of contained queries in the cache, we get the following two additional advantages:

- Faster access to the stored data than the alternative of accessing it over a network.
- Concentration of queries only in the domains in which the user is interested. Thus, increasing the possibility of overlap of the performed queries, and by its effect, the average speed of query execution increases.

1.1 Definitions

1.1.1 GQR:

The key idea behind the algorithm GQR (Graph-based Query Rewriting) is to compactly represent common subexpressions in the views; while at the same time treating each subgoal atomically and taking into account the way each of the query variables interacts with the available view patterns, as well as the way these view patterns interact with each other. This representation is achieved by decomposing the query and the views to simple atomic subgoals, and depicting them as graphs.[George Konstantinidis et al., 2011]

1.2 Existing models

Existing caching systems perform primarily as a server side caching mechanism with an aim to improve performance of the online query engine. Some application [Michael Martin et. al.,2010] attempt to improve the performance of triple stores by caching query results and even complete application objects. But this solution cached only the most popular queries.

Another system developed [Johannes Lorey et. al.,2013] addresses the issue of faster querying using caching by modifying the structure of a query so that the altered query can be used to retrieve additional related information. This additional data is shown to be potentially interesting for subsequent requests in advance. But this solution has data redundancy and does not ensure that the subsequent queries would be done faster.

2 Our Approach

We propose a caching system which would be deployed on the client system. This system would allow users to cache the RDF data while fetching only the data not already cached from an online endpoint when required. Also we ensure that the data is not stale by regularly updating the based on the fixed frequency.

We achieve this with two modules that work in parallel independent of each other. The data retrieval module accepts the user query and returns its results to the client interface. We perform a containment check of the user query with respect to the previously cached queries using "GQR" algorithm. The autonomous data update module performs the update of data in the cache based on a fixed frequency.

2.1 Data Retrieval Module

The module first receives the SPARQL query and performs a containment check using GQR. If the query is contained within previous queries, we get the data based on rewritings produced by the containment sub-module. If the query is not contained within the previous queries performed on the system, the module executes the query on the SPARQL endpoint. The results of the new query, along with the query, are stored in the cache. Figure 1. Shows the overall process of the data retrieval module and we further discuss each sub-module involved in this process in the further sections.

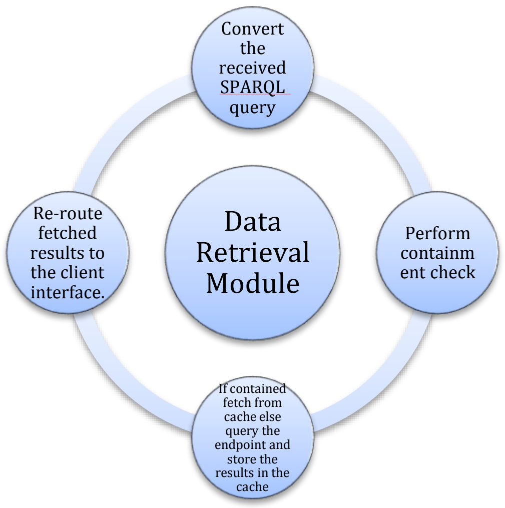

Convert the received SPARQL query
Data Retrieval Module
Re-route fetched results to the client interface.
Perform containm ent check
If contained fetch from cache else query the endpoint and store the results in the cache

Fig1. The process flow of the data retrieval module.

2.1.1 Conversion of SPARQL query

The GQR algorithm performs containment checks only on the conjunctive queries. SPARQL queries are inherently not conjunctive queries and would require conversion to conjunctive queries. This sub-module converts a SPARQL query into its equivalent conjunctive query. The current implementation of the GQR algorithm does not accept constant-values for predicates, hence SPARQL queries with constants cannot be allowed for containment check. It is for this reason that in our benchmark query-set (discussed in section 3.1) we don't include such queries. The algorithm to convert SPARQL query to its conjunctive form is shown below:

```
  program Conversion (Output)
    var   tokens: a list of tokens from SPARQL queries;
    begin
      transformedQuery = queryName+"("
      if tokens contains 'Select'
         start = index of 'Select' in tokens + 1
         if tokens contains 'Where'
             end = index of 'Where' in tokens
              repeat
                   insert tokens[start] to transformedQuery
                  as query variable
              until start = end
             start = startIndex of triples after 'Where'
             repeat
                if predicate at tokens[start] == "rdf:type"
                  or "a"
                    insert triple tokens[start] to
                     transformedQuery as object(subject)
                else
                    insert triple tokens[start] to
                     transformedQuery as predicate
                endIf
              until start = endIndex of tokens
         else
             return "Incorrect Format"
         endIf
      else
             return "Incorrect Format!"
      endIf
      return "Incorrect Format!"
end.
```

2.1.2 Containment check sub-module

Once the SPARQL queries are converted into its equivalent conjunctive form, they are passed to the GQR alogorthm which gives the final rewritings for the new query. For the containtment the LAV rules are required, which are produced using the previously cached queries where each of cached queries acts as a source, and the actual endpoint elements act as mediated views. The new query is tested for containment against these LAV rules. After performing the containment check, we get the query rewritings as seen in the figure 2.

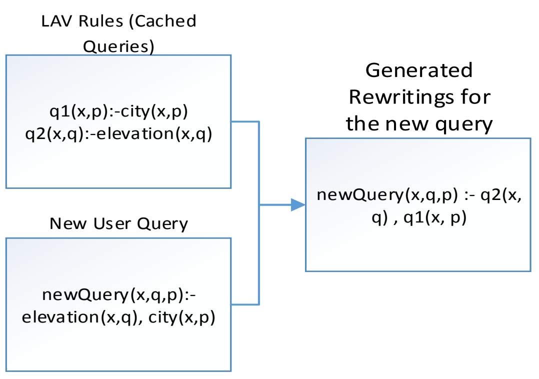

LAV Rules (Cached Queries)
q1(x,p):-city(x,p)
q2(x,q):-elevation(x,q)
New User Query
newQuery(x,q,p):-elevation(x,q), city(x,p)
Generated Rewritings for the new query
newQuery(x,q,p) :- q2(x, q) , q1(x, p)

Fig2: Demonstration of the rewritings produced by GQR algorithm when the LAV rules and the conjunctive query is provided.

2.1.3 Minimization of query rewritings

The produced rewritings also contain duplicate rewritings which have to be removed. For instance, if there is a query rewriting such as q(x,y) :- V1(x,f(x), g(x)), V1(h(x),y,l(x)), then it is converted into its equivalent conjunctive form q(x,y) :- V1(x,y,_). This ensures that unnecessary joining of data from different views is avoided. Due to a bug in the original GQR implementation the discussed feature was removed in their implementation. Therefore, we have fixed it in our implementation.

Program

```
program CombinePredicates (Output)
  var tokens:= a Map predicate and value pairs of all
tokens extracted from list of rewritten queries;
  var List:= will contain the final list of combined
queries
  begin
    repeat
       repeat
          if(token_key in tokens)
              if(match(tokens[token[index_inner]],
                         tokens[token[index_outer]]))
                 merge(tokens[token[index_inner]],
                                tokens[token[index_outer]]);
              endif
          end
      until index_inner < len(tokens)-2
    until index_outer < len(tokens)-1
  end
```

2.2 Autonomous Data Update Module

Once the cache has been created, the data is ready for any contained queries to run on the cache and retrieve the data. But over a period of time the data becomes stagnant, if the data is not periodically updated. Hence, the probability of the consumer observing stale data becomes higher [Hiroshi Wada et al, 2011].

Data consistency. Data consistency refers to process of the ensuring that the data cached in the repository is fresh. Data consistency must be ensured separate from the main process flow discussed in the previous section (2.1). In order to achieve parallelism, we propose a model which works in parallel to the main process and ensures that the data is consistent.

2.2.1 Cache Update Process:

The overall process flow of this module is seen in figure 3. First all the details associated with caches namely the cache names and the associated query are fetched. These details are stored as a singleton object in-order to obviate the redundant process reading these details again.

Next the SPARQL endpoint is queried to retrieve the actual data. As part of the initial configuration, if the batch insert mode is set to true, then based on the predefined number of batch insertion value, the script for insertion is dynamically created and the data is inserted into the cache.

Also the hash value of the entire record retrieved from the SPARQL endpoint is taken using MD5 checksum [Mary Cindy et al., 2013], which ensures that no duplicates of the data are present in the cache. Finally, the cache is updated and the process repeats.

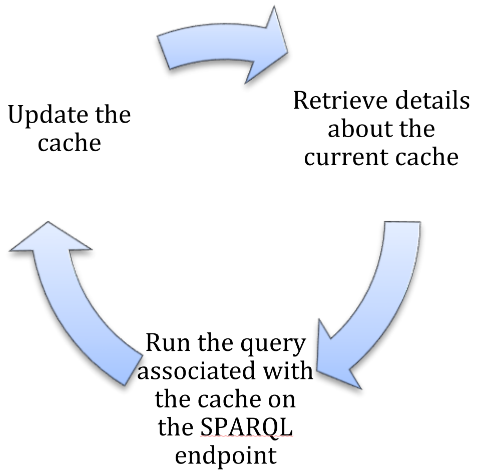

Update the cache
Retrieve details about the current cache
Run the query associated with the cache on the SPARQL endpoint

Fig 3. The process flow for the autonomous data update module.

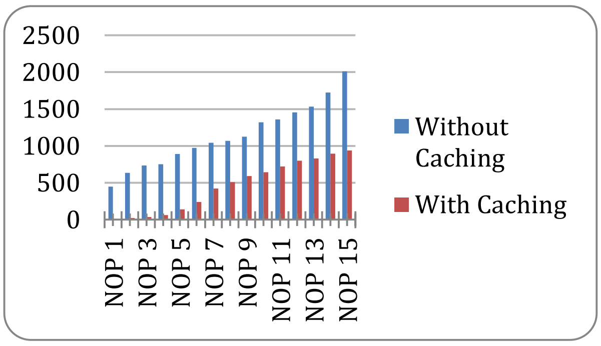
3 Experimental Evaluation

3.1 Benchmark Query-Set

The experimental evaluation phase started with the construction of the query set. Two major benchmarking query-sets were taken into consideration. One was the Lehigh University Benchmark and the other is the SP2Benchmark [Michael Schmidt et al., 2010]. But these benchmarking queries were heavily reliant on constants to get the results. Since current implementation of GQR could not handle constants, a constant free non-empty query set was needed. So we created a set of 30 queries with varying volumes, which helped us compare the results of normal cache access due to containment, and querying the SPARQL-endpoint directly. The highest number of predicates in a query is 15 predicates. Appendix 1 shows a sample of the queries that were chosen. These queries are broadly divided into high volume queries (viz. large input size) and low volume queries (viz. smaller input size).

3.2 Results Obtained

The results are discussed in this section. Figure 4 shows the performance increase obtained by comparing the results of using Sesame for running the SPARQL queries against using our caching approach to query the data. Only the time taken to query the endpoint or the cache is taken into consideration and it does not include any non-trivial tasks like insertion of the data in the cache. Also, only the high volume queries discussed in Appendix 1 are used.



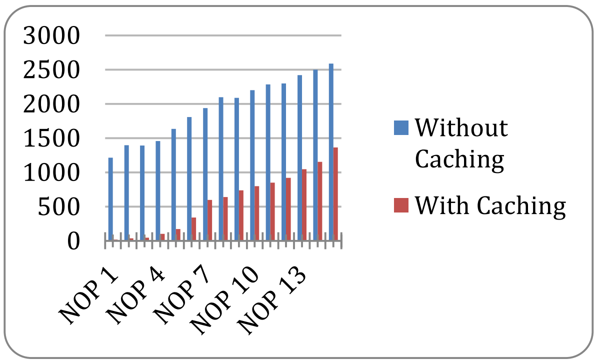


Fig4: Number of predicates compared against the time taken to query SPARQL endpoint using sesame versus time taken to query using the caching system (a) With a LIMIT of 100 records. (b) With a LIMIT of 1000 records.

In figure 5, a comparison of the number of joins needed to obtain the results, against the time taken to query the results is shown. The study reveals that the when the number of joins are lesser, the difference in time between the number of records queried is negligent. But when the number of joins are increased, the retrieval time increases.

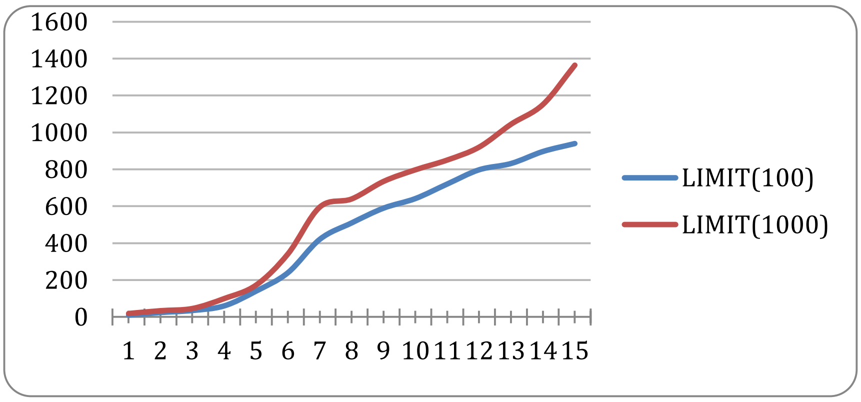


Fig5: Number of joins done using cache is plot against the time taken to query SPARQL endpoint using sesame. There is a LIMIT on the number of records as 100 records and 1000 records.

4 Conclusion

Thus, based on our experimental results, we conclude that the client-side caching mechanism is a better approach than directly querying the online SPARQL endpoint. It is the suggested approach for client side storage of data and would enable us to query faster for data. From our analysis we were able to conclude that a minimal of 47% increase in response time can be achieved.

5 Future Work

- There are some further enhancements for the system, which can be explored in the future research. Instead of saving the complete query data on the whole, we can store the individual predicates. This enables us to get data even for a part of the query, which may not be completely contained in the previous queries. This may greatly enhances the reusability provided by the maintained cache for future queries.
- Furthermore, we can modify the current implementation of G.Q.R. algorithm to accept queries with constant values.

Acknowledgments. *We sincerely thank Professor. Craig Knoblock and Professor Jose Luis Ambite for providing us with their valuable insights and for their time. We would also like to thank Dr. George Konstantinidis for his timely help.*

References

- George Konstantinidis and José Luis Ambite : Scalable Query Rewriting: A Graph-Based Approach, SIGMOD'11, June 12–16, 2011, Athens, Greece.
- Michael Martin, J¨org Unbehauen, and S¨oren Auer - Improving the Performance of Semantic Web Applications with SPARQL Query Caching - ESWC'10 Proceedings of the 7th international conference on The Semantic Web: research and Applications - Volume Part II – 2010.
- Johannes Lorey, Felix Naumann - Caching and Prefetching Strategies for SPARQL Queries - The Semantic Web: ESWC 2013 Satellite Events – 2013.
- Mary Cindy Ah Kioon, ZhaoShun Wang and Shubra Deb Das - Security Analysis of MD5 algorithm in Password Storage, Proceedings of the 2nd International Symposium on Computer, Communication, Control and Automation (ISCCCA-13).
- Michael Schmidt, Thomas Hornung, Georg Lausen, Christoph Pinkel - SP2Bench: A SPARQL Performance Benchmark, Semantic Web Information Management,2010 .
- Hiroshi Wada, Alan Fekete, Liang Zhao, Kevin Lee and Anna Liu - Data Consistency Properties and the Trade-offs in Commercial Cloud Storages: the Consumers' Perspective, CIDR'11 Asilomar, California, January 2011.

Appendix 1: Sample Query Set

Number of Predicates : 9
Low Volume : 92 records
SELECT * WHERE { ?e <http://dbpedia.org/ontology/series> ?seriesName.

?seriesName dbpedia-owl:director ?director. ?director dbpedia-owl:occupation ?occupation. ?e <http://dbpedia.org/ontology/releaseDate> ?date; <http://dbpedia.org/ontology/episodeNumber> ?number; <http://dbpedia.org/ontology/seasonNumber> ?season; <http://dbpedia.org/ontology/director> ?director; <http://dbpedia.org/ontology/abstract> ?abstract; <http://dbpedia.org/ontology/photographer> ?photographer. }
High Volume : 279109 records
SELECT count(*) WHERE { ?x a ?z ; dbpedia-owl:city ?p ; dbpprop:location ?t ;dbpprop:cityServed ?m. ?t dbpedia-owl:country ?country. ?country dbpedia-owl:capital ?capital. ?country dbpedia-owl:language ?language. ?country dbpprop:establishedEvent ?event; dbpprop:currency ?currency.}
Number of Predicates : 15
Low Volume : 16 records
SELECT * WHERE { ?e <http://dbpedia.org/ontology/series> ?seriesName. ?seriesName dbpedia-owl:director ?director. ?director dbpedia-owl:occupation ?occupation. ?e <http://dbpedia.org/ontology/releaseDate> ?date; <http://dbpedia.org/ontology/episodeNumber> ?number; <http://dbpedia.org/ontology/seasonNumber> ?season; <http://dbpedia.org/ontology/director> ?director; <http://dbpedia.org/ontology/abstract> ?abstract; <http://dbpedia.org/ontology/photographer> ?photographer; <http://dbpedia.org/ontology/previousWork> ?previousWork; <http://dbpedia.org/ontology/releaseDate> ?releaseDate; <http://dbpedia.org/ontology/subsequentWork> ?subsequentWork; <http://dbpedia.org/ontology/wikiPageID> ?wikiPageID; dbpedia-owl:writer ?writer. ?writer dbpedia-owl:occupation ?occupation. }
High Volume : 2418925 records
SELECT count(*) WHERE { ?x a ?z ; dbpedia-owl:city ?p ; dbpprop:location ?t ;dbpprop:cityServed ?m. ?t dbpedia-owl:country ?country. ?country dbpedia-owl:capital ?capital. ?country dbpedia-owl:language ?language. ?country dbpprop:establishedEvent ?event; dbpprop:currency ?currenc; dbpprop:callingCode ?callingCode; dbpprop:largestCity ?largestCity; dbpprop:leaderName ?leader; dbpprop:officialLanguages ?officialLanguages; dbpprop:legislature ?legislature; dbpprop:leaderTitle ?leaderTitle.}