Highly Parallel SPARQL Engine for RDF

Fan Feng¹, Weikang Zhou¹, Ding Zhang¹, and Jinhui Pang²(✉)

¹ Tianjin University, Tianjin, China

² Beijing Institute of Technology, Beijing, China
pangjinhui@bit.edu.cn

Abstract. In this paper, a highly parallel batch processing engine is designed for SPARQL queries. Machine learning algorithms were applied to make time predictions of queries and reasonably group them, and further make reasonable estimates of the memory footprint of the queries to arrange the order of each group of queries. Finally, the query is processed in parallel by introducing pthreads. Based on the above three points, a spill time prediction algorithm was proposed, including data processing, to better deal with batch SPARQL queries, and the introduction of pthread can make our query processing faster. Since data processing was added to query time prediction, the method can be implemented in any set of data-queries. Experiments show that the engine can optimize time and maximize the use of memory when processing batch SPARQL queries.

Keywords: SPARQL · Pthread · Multithreading · Performance prediction

1 Introduction

With the explosion of RDF data, SPARQL query related technologies are also advancing by leaps and bounds. Over the years, many have been devoted to how to build SPARQL storage, and how to effectively answer the graphical query mode expressed in SPARQL. Currently, stand-alone RDF engines, such as RDF3X [1], gStore [4, 5], can efficiently execute SPARQL queries serially. Their performance and stability are relatively high. With the continuous development of computer hardware, the memory of a single computer is getting larger and larger, and the parallel processing capability is getting higher and higher, but how to use the parallel capability to handle batch query problems has still not been solved well. But how to use the parallel ability to deal with the batch query problem has not been solved well.

The existing multi-query optimizations [MQO] [12] are based on finding common parts of batch queries and processing them to get answers. It uses query rewrite technology to achieve ideal and consistent MQO performance and guarantee its integrity on different RDF stores. However, the MQO on this SPARQL query is np-hard [11], and the equivalent relationship algebra that has been established between SPARQL and the relationship.

Our method chose to avoid this np-hard problem and switched to using pthreads for parallel processing. We use multiple threads, which use the same address space with each other, share most of the data, and start a thread. Space is much smaller than the time it takes to start a process, and the time required to switch between threads is much less than the time required to switch between processes. It is fast to create and destroy. Naturally, this method is extremely useful for batch processing of saprql queries. Big advantage. At the same time, in order to make better use of computer performance, we also performed time prediction [2] and memory estimation on the query to do reasonable grouping and query. In a nutshell, the main contributions of this work are summarized as follows:

- We introduced query-time predictions to initially group batches of questions. For a problem, we use machine learning algorithms to predict its time consumption based on its algebraic tree structure, BGD size, and database format. In this way, a batch of problems is grouped, which is convenient to use multi-threading to process.
- We have also increased memory estimation to avoid unbalanced query memory usage. It is a reasonable method to evaluate the memory usage of the query by its intermediate result size. We make a cardinality estimation of the query and use its triple structure to find its position occupation in the data graph.
- We use pthread technology to process spall queries in batches in parallel. Through multi-threading [16], we can improve the query speed very well, and also show exceptionally high practicality in batch processing queries.

2 Related Work

In relational databases [6–10], the problem of multi-query optimization has been well studied. The main idea is to identify common sub-expressions in a batch of queries. Construct a globally optimized query plan by reordering the join sequence and sharing intermediate results in the same set of queries, thereby minimizing the cost of evaluating common subexpressions. The same principle was also applied in [7], which proposed a set of heuristics based on dynamic programming to handle nested sub-expressions. The general MQO problem of a relational database is the NP problem. Even with the heuristic approach, the search space for a single candidate scheme and its combined hybrid scheme (i.e., the global scheme) is usually astronomical. Because of hardness, they proposed some heuristic methods, which have proven to be effective in practice.

All the above work is focused on MQO in a relational situation. To avoid this np problem, we chose to use pthread to process batches of SPARQL in parallel. At the same time, To make better optimization, we introduced query estimation to complete better distribution.

3 Preliminaries

In this section, we will discuss the main ideas for the design of this paper. We avoided MQO processing problems of np-hard difficulty and chose to use threads to process SPARQL in parallel in batches. However, since the usage time and memory usage of a single query are different, and even a large gap may occur, we must allocate it reasonably.

3.1 SPARQL

An RDF (Resource Description Framework) data model is proposed for modeling Web objects, which is part of the development of the Semantic Web. It has been used in various applications. For example, Yago and DBpedia automatically extract facts from Wikipedia and store the points in RDF format to support structured queries on Wikipedia [18]. In general, RDF data can be represented as a collection of triples represented by SPO (object, attribute, object). A running example G_D is given in follows.

```
Barack_Obama ⟨bornIn⟩ Honolulu.
Barack_Obama ⟨won⟩ Peace_Nobel_Prize.
Barack_Obama ⟨won⟩ Grammy_Award.
Honolulu ⟨locatedIn⟩ USA.
```

SPARQL is a query language and data acquisition protocol developed for RDF. It is defined by the RDF data model developed by the W3C [14] but can be used for any information resource that can be represented by RDF. A SPARQL query graph $G_Q(V_Q, E_Q, L, \text{Vars})$ is a labeled, directed multi-graph where V_Q is the set of query nodes; E_Q is the set of edges connecting nodes in V_Q , L is the set of edge and node labels. An RDF example in triplet form (TTL/N3 format) and an example query (which retrieves all the people who are born in a city that is located in “USA” and who won some prize) is expressed in SPARQL as follows.

```
SELECT  ?peron, ?city, ?prize WHERE {
    ?person ⟨bornIn⟩ ?city.
    ?city ⟨locatedIn⟩ USA.
    ?person ⟨won⟩ ?prize. }
```

Therefore, processing the SPARQL query graph G_Q for the RDF data graph G_D can solve the problem of finding the isomorphism of all subgraphs between G_Q and G_D . For example, the method of SPARQL query on our RDF data segment is as follows.

```
Barack_Obama, Honolulu, Peace_Nobel_Prize.
Barack_Obama, Honolulu, Grammy_Award.
```

3.2 Time Prediction

We use machine learning algorithms [2] and innovate existing time prediction algorithms. Wei Emma Zhang proposed the work of predicting SPARQL query execution time using machine learning techniques. In work, multiple regression using support vector regression (SVR) was adopted. The evaluation was performed on the open-source triple storage Jena TDB for benchmark queries. Feature models were extracted based on the graph edit distance (GED) [13] between each training query, and feature extraction was also performed on the query's algebraic tree structure [13]. Further, improve accuracy. But in fact, we found that its method does not perform relevant feature extraction on the data set. We believe that data also has a great influence on the query. The simplest example is that the larger the data, the slower the query will become.

In order to effectively predict SPARQL query time, we borrowed the ideas of Wei Emma Zhang. Still, we introduced a new feature vector of the data set to further introduce the characteristics of the data into the time prediction.

3.3 Memory Estimation and Thread Acceleration

We consider that the query memory usage of SPARQL lies in the storage of the intermediate results, so we have designed a cardinality estimation algorithm, which builds a b-tree index on the RDF data and maps the triples of the query to get the approximate answer range. Further, the connection cost of the triples is obtained to obtain the estimated cardinality of the query, which represents the memory usage of the query. Through the above two steps, the query is grouped, and the dynamic programming method is used for reasonable thread usage allocation to ensure that the query time is minimized and the occupied memory is minimized.

4 Time Prediction

We used Wei Emma Zhang to use machine learning technology to predict SPARQL query execution time. The method proposed that the prediction time mainly depends on the features in the training set, so how to extract the problem information is an important breakthrough point. The vector of data information is also introduced to improve the accuracy and applicability of problem prediction. In this study, we introduce algebraic features and BGP features, which are obtained by parsing query text (see Sects. 4.1 and 4.2). By applying algebraic and BGP feature-based selection algorithms to generate hybrid features (see Sect. 4.3), the dataset analyzes depth-width features (see Sect. 4.4).

4.1 Algebraic Features

Algebra refers to operators such as opt, union, etc. Algebra can be represented and extracted in the form of a tree:

Definition 1. (Algebra Tree) Given a SPARQL query Q , the algebra tree $T_{Algebra}(Q)$ is a tree where the leaves are BGP, and the nodes are the algebraic operators of the hierarchical representation. The parent of each node is the parent operator of the current operator.

We obtain the SPARQL algebra tree and then traverse the tree to build an algebraic set by recording the occurrence and hierarchical information of each algebraic operator.

Definition 2. (Algebra Set) Given an algebra tree represented as $T_{Algebra}(Q)$, the algebra set is a set of tuples $(opt_i, c_i, maxh_i, minh_i)$, where opt_i is the name of the operator and c_i is the number of occurrences in $T_{Algebra}(Q)$. Opt_i , $maxh_i$ and $minh_i$ are the maximum and minimum heights of opt_i in $T_{Algebra}(Q)$.

4.2 BGP Feature

Algebraic features can only represent part of the query information, and the specific structural features are still not obvious enough. Therefore we introduce the graph structure of BGP as a supplement. Obtain the features of the BGP structure and convert it into a vector representation.

Fig. 1. Mapping triple patterns to graphs. Left: eight types of triple patterns are mapped to eight structurally different graphs. Right: mapping example query in Figure ?? to a graph

We map all eight types of ternary patterns to eight different graphs on the structure, and we convert the SPARQL in Figure ?? to Fig. 1 as an example. The black rectangles in the figure are the connection nodes. After converting the query into a graph representation, we use the graph edit distance as its BGP feature vector. The graph edit distance between two graphs is the minimum amount of editing operations required to convert one graph to another (i.e., delete, insert And replace nodes and edges). Figure 2 shows a GED calculation process. We calculated the target query and 18 query templates separately. These 18 query templates were filtered from the DBPSB benchmark [15] test. So we can get an eighteen-dimensional vector.

Fig. 2. Graph edit path

4.3 Data Feature

In view of the acquisition of query feature vectors, we have also derived the way of extracting data information. An RDF data set, which is linked in the form of a graph, must also have class and instance information, so we separate the class and instance from it.

Definition 3. (Data Information) Given a data set G , its class information, including the number of child classes and parent classes of each class, is used as the width of the data set. The information of the instance, including all the attributes of the instance, and the number of objects, is used as the depth of the data set.

Width calculation: We traverse the entire graph, and record the class when it meets the definition of the class, and traverse its parent and child classes for this class. If there is no parent and child class, the width of the graph is increased by one. If there is a parent and child class, the width of the graph is increased by Third, and record this category to prevent secondary statistics. **Depth calculation,** for a data set, traverse the graph, using a recursive algorithm, if the depth is already available, directly return the value depth of the instance plus 1, query the instance to get all its objects; traverse the object, perform the object Processing and then following the recursive algorithm. Each time an object is traversed, the depth is added to the depth of the object.

5 Memory Estimation

We introduce the estimate of the cardinality of the query to determine how much intermediate results the query has, and then determine its memory footprint. The challenge we face is how to estimate the cardinality of the query [17]. We graph the query and map each triplet pattern in BGP to node v . If there is a common variable between the triplet patterns, add an edge e between the nodes. After processing, you will get the SPARQL The connection graph, which is an undirected connected graph, is numbered according to the triples in the graph pattern entered by the user.

1. **Node estimation.** Build a B-tree index of RDF data. Use AllGroGraph to build three indexing strategies: *spo*, *pos*, and *osp*. These three include six patterns of triples, which can also save space. For example, in the triple pattern $(?spo)$, if the subject's hash value range is set to $[0-0xff]$, the hash

range of the entire triple can be determined. and set to $[n1, n2]$. For this value range in the B-tree, The query above calculates the number of results N with key values between $n1$ and $n2$, and the data in the range contains the results of triple matching. The estimate of its cardinality is:

$$cart(t) = 3 * N/4 \quad (1)$$

2. **Edge weight estimation.** The weights for the edges are calculated using an estimate of the size of the result set of the join operation on the node words. For the estimation of the size of the result set of the join operation, the estimated value of the cardinality of the node is used, and the formula is as follows:

$$T(R \bowtie S) = s * |R| * |S| \quad (2)$$

Among them, R and S respectively represent the result set after pattern matching of triples, and $|R|$ and $|S|$, respectively represent the cardinalities of their triples results. s indicates the selection rate, which is divided into the following situations:

- No variables common to R and S . Then s is a constant coefficient of 1.

$$T(R \bowtie S) = |R| * |S| \quad (3)$$

- When left and right operands have common variables, suppose the set of public variables is the intersection of V_R and $V_S:V_{RS}$. $W(R, V_{RS})$ represents the number of different variables on R and V_{RS} . $W(S, V_{RS})$ represents the number of different variables on S and V_{RS} , take the maximum value between these two as max . And assuming that the variables are uniformly distributed, the formula becomes:

$$T(R \bowtie S) = \frac{|R| * |S|}{max} \quad (4)$$

6 Pthread Allocation

One of the reasons for using multi-threading is that it is a very “frugal” way of multitasking compared to processes. We know that starting a new process must allocate it an independent address space and create numerous data tables to maintain its code, stack, and data segments. This is an “expensive” multitasking way of working. And multiple threads running in a process use the same address space with each other and share most of the data. The space taken to start a thread is much less than the space taken to start a process. Moreover, the threads switch between each other. The time required is also much less than the time required for interprocess switching. According to statistics, in general, the cost of a process is about 30 times the cost of a thread. Of course, on a specific system, this data may vary greatly.

In our method, a batch of problems and their corresponding data are introduced. The problems are grouped by time estimation models into m groups, and

the m groups of problems are processed in parallel using multiple threads while ensuring that the total time spent in each group reaches an average. Before querying, first, make a memory estimation of the problems in each group, and sort the problems in each group to ensure that when one problem is taken out from m groups for processing, the memory occupation of these m problems can be processed in this batch of problems. The total occupancy in the medium reaches an average.

7 Evaluation

In this section, we provide an assessment of our proposed method. We first introduce the experimental setup. We then report the results of various evaluations performed on different data sets and query sets.

7.1 Setup

Data Preparation. Data sets and queries use dbpedia3.9 data and query records. We first randomly selected ten datasets containing 100,000 triples from the DBpedia dataset, and then USEWOD 2014 provided log files for querying using the DPPedia SPARQL query engine. We decoded the log files and selected the A total of 10,000 questions were found and correctly returned in the selected data set. From this, we have ten pairs of data, including questions and answers.

System. The backing system of our local triple store was Jena TDB, installed on 64-bit Ubuntu 16.04 Linux operation system with 32 GB RAM and 16 CPU cores.

Evaluation Metric. For all experiments, we measure the number of optimized queries and their end-to-end evaluation time, including query rewrite, execution, and result distribution. We compare our Pthread algorithm with an evaluation without any optimization (i.e., No-MQO) and an algorithm with MQO (including various existing optimizations). At the same time, in order to ensure the effective use of memory, we have set up indicators that specifically detect memory usage to ensure that our memory allocation is indeed effective.

7.2 Experimental Results

We took ten sets of queries and data to train the time prediction model, and each group randomly extracted 600 queries, parsed out the query features and data features to build a prediction model, and used the remaining 400 queries for model detection. We directly give the results of the time prediction model trained by machine learning, as shown in Fig. 3. From this figure, we can see that we have made a good prediction of the query time, and this predicted time can be directly processed. We choose structure-based features that can be obtained directly from

Fig. 3. Time predict effect

the query text. As stated in many works, similarly structured queries can have huge performance gaps due to the irregular distribution of values in the query data. However, based on our actual experience in this work, we have observed that although it may cause prediction distortion, based on the limited features we can obtain, the error rate is acceptable.

After the prediction model is processed, we randomly select a set of 1000 query data randomly, use the prediction model to group it, and then use memory estimation to sort within the group. Then use Pthread multi-thread batch processing to query our query performance. A comparison with other methods is shown in Table 1.

Table 1. Experimental results

Model	Processing time (s)
MQO	47
MQO-S	42
No-MQO	67
Pthread	38

8 Conclusion

We investigated batch query optimization in the context of RDF and SPARQL. We refer to the multi-query optimization processing method and choose to use Pthread to avoid its difficulties and handle these queries from a new perspective. Our method also introduces time prediction and memory estimation to make our query process faster, and more accurate and efficient. In addition, our technology is storage-independent, so it can be deployed on top of any RDF storage without modifying the query optimizer. The experimental results show that our method is effective.

Acknowledgments. This work is supported by Big Data Research Foundation of PICC.

References

1. Neumann, T., Weikum, G.: The RDF-3X engine for scalable management of RDF data. *VLDB J.* **19**(1), 91–113 (2010)
2. Zhang, W.E., Sheng, Q.Z., Qin, Y., Taylor, K., Yao, L.: Learning-based SPARQL query performance modeling and prediction. *World Wide Web* **21**(4), 1015–1035 (2017). <https://doi.org/10.1007/s11280-017-0498-1>
3. Le, W., Kementsietsidis, A., Duan, S., et al.: Scalable multi-query optimization for SPARQL. In: 2012 IEEE 28th International Conference on Data Engineering. IEEE Computer Society (2012)
4. Zou, L., Mo, J., Chen, L.: gStore: answering SPARQL queries via subgraph matching. *Proc. VLDB Endow.* **4**(8), 482–493 (2011)
5. Zou, L., Oezsu, M.T., Chen, L., et al.: gStore: a graph-based SPARQL query engine. *VLDB J.* **23**(4), 565–590 (2014)
6. Park, J., Segev, A.: Using common subexpressions to optimize multiple queries. In: International Conference on Data Engineering. IEEE (1988)
7. Roy, P., Seshadri, S., Sudarshan, S., et al.: Efficient and extensible algorithms for multi query optimization. *ACM SIGMOD Rec.* **29**(2), 249–260 (2000)
8. Sellis, T.K.: Multiple-query optimization. *ACM Trans. Database Syst.* **13**(1), 23–52 (1988)
9. Shim, K., Sellis, T.K., Nau, D.: Improvements on a heuristic algorithm for multiple-query optimization. *Data Knowl. Eng.* **12**(2), 197–222 (1994)
10. Zhao, Y., Deshpande, P., Naughton, J.F., Shukla, A.: Simultaneous optimization and evaluation of multiple dimensional queries. In: SIGMOD (1998)
11. Sellis, T., Ghosh, S.: On the multiple-query optimization problem. *IEEE Trans. Knowl. Data Eng.* **2**(2), 262–266 (1990)
12. Wang, M., Fu, H., Xu, F.: RDF multi-query optimization algorithm for query rewriting using common subgraphs. In: The 3rd International Conference (2019)
13. Hasan, R.: Predicting SPARQL query performance and explaining linked data. In: Presutti, V., d’Amato, C., Gandon, F., d’Aquin, M., Staab, S., Tordai, A. (eds.) *ESWC 2014. LNCS*, vol. 8465, pp. 795–805. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07443-6_53
14. Servidor web: World Wide Web Consortium (W3C) (2010)
15. Morsey, M., Lehmann, J., Auer, S., Ngomo, A.N.: Usage-centric benchmarking of RDF triple stores. In: Proceedings of the 26th AAAI Conference on Artificial Intelligence, Toronto, Canada (2012)
16. Filip, Z.: Parallel SPARQL query processing using bobox. *Int. J. Adv. Intell. Syst.* **5**, 302–314 (2012)
17. Gubichev, A., Neumann, T.: Exploiting the query structure for efficient join ordering in SPARQL queries. In: Proceedings of the 17th International Conference on Extending Database Technology (EDBT 2014), Athens, Greece, pp. 439–450 (2014)
18. Suchanek, F.M., Kasneci, G., Weikum, G.: Yago: a core of semantic knowledge. In: WWW, pp. 697–706 (2007)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.