

A Highly Parallel Engine for RDF

Shujun Wang, Yongxin Yu, Xiaowang Zhang*, and Zhiyong Feng

College of Intelligence and Computing, Tianjin University, Tianjin 300350, China
Tianjin Key Laboratory of Cognitive Computing and Application, Tianjin, China

* Corresponding author: xiaowangzhang@tju.edu.cn

Abstract. In this paper, we designed a highly parallel engine for RDF, which includes a new RDF storage model with little memory dependence and a Machine Learning algorithm. Based on the above two points, we present a heuristic query decomposition algorithm to split a SPARQL query into subqueries and then process these subqueries parallelly. Experiments show that our engine perform better on SPARQL query execution with maximizing the usage of memory.

1 Introduction

With the continuous development of computer hardware, the memory of a single computer becomes larger and the parallel processing capability becomes higher. Some application of knowledge graph, such as RDF Q/A system, which has strict requirements on system efficiency and stability. However, the amount of data it needs is not very large, a single computer is sufficient.

Currently, single-machine RDF engine, such as RDF3X [2], gStore [5], SMat [4], they can efficiently execute SPARQL queries serially. Their performance and stability are relatively high. However, due to the improvement of single-machine parallel capability, how to efficiently execute SPARQL queries in parallel on a computer has become a very interesting problem. At the same time, although many parallel processing schemes for SPARQL are presented in distributed RDF Engines, the design bottleneck of SPARQL is quite different from that of single-machine parallel processing, Specifically:

- Parallel execution of SPARQL queries on a computer will cause tremendous pressure on computer memory, while the main optimization direction of distributed systems is how to reduce communication costs.
- It is necessary to design a reasonable and efficient query decomposition scheme for parallel execution of SPARQL queries on a single computer.
- In parallel environment, single-machine RDF Engine should choose more suitable joining order strategy.

In this paper, we present a new storage model of RDF Engine with low memory dependence. Moreover, we develop a heuristic SPARQL decomposition algorithm based on Machine Learning to split a SPARQL query into subqueries.

Copyright © 2019 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

2 RDF Storage

In this paper, we propose a novel approach to manage RDF data (named TriStore). Figure 1 is an example of TriStore used to store RDF data. In order to save memory usage, instead of storing entire strings or URIs, we use shortened versions or keys. For each RDF element value, TriStore maintains a mapping table that maps these keys to their corresponding strings. After encoding, we convert Figure 1(a) into Figure 1(b). In order to match queries more efficiently,

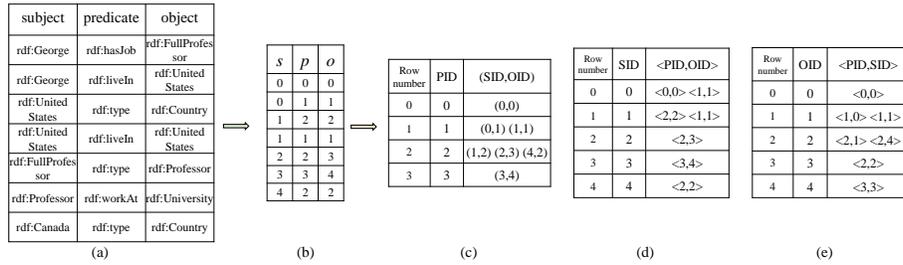


Fig. 1. An example of TriStore storing RDF data

we use three tables (Figure 1(c), Figure 1(d), Figure 1(e)) to store set of triples D_i in memory, which support the following query operations, where s , p and o are subject, predicate and object:

1. given p , return set $\{(s, o) \mid \langle s, p, o \rangle \in D_i\}$
2. given s and p , return set $\{o \mid \langle s, p, o \rangle \in D_i\}$
3. given o and p , return set $\{s \mid \langle s, p, o \rangle \in D_i\}$

Where Figure 1(c), Figure 1(d) and Figure 1(e) are used to support query types 1, 2 and 3 above, respectively. For the first type of query, only the predicate p is known, and we want to get all p -related s and o . For TriStore, we first get the encoding number m of string p , and then we only need to output all the data of line m in Figure 1(c) (it can be observed that the line number of the table is equal to the encoding of the predicate). For example, we perform this SPARQL query (*Select*where*{ $?x$ *rdf:type* $?y$.}). First find the digital encoding of the predicate *rdf:type*, after finding that the encoding is 2, then directly output all the data in the second row of Figure 1(c). Therefore, after obtaining the encoding of string p , corresponding data can be quickly found with $O(1)$ time complexity.

Figure 1(d) used to support fast matching of the second type of query, where s and p are known. What we want to get is o which is related to both s and p . For TriStore, the encoding m and n of string s and p should be obtained first. As we can see from the running examples, line m in Figure 1(d) stores all PIDs and OIDs associated with the SID m (there are a mapping table that maps PIDs to OIDs). Hence, we only need to output all the data in the m -th line with the key n .

3 Query Optimization

In order to maximize the efficiency of SPARQL queries in parallel environment, we split Q into several subqueries with the same cost. We accept the machine learning based method in [1] to estimate the time cost of the SPARQL query. We use $T(Q)$ to represent the time cost of executing query Q .

Heuristic SPARQL Decomposition Using exhaustive method to decompose the SPARQL is less efficient. In this section, we present a heuristic query decomposition algorithm, which can efficiently split query into two subqueries with equal execution cost. We use degree (denoted as d_n) to represent the number of

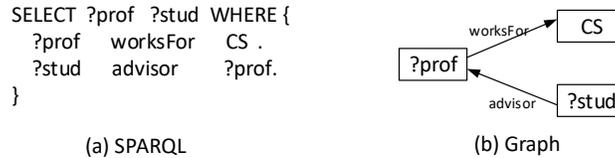


Fig. 2. A SPARQL query that finds CS professors with their advisees.

edges connected to node n in SPARQL graph. For example, in Figure2 (b), $d_{?prof} = 2, d_{CS} = 1, d_{?stud} = 1$. It is easy to see that the nodes with higher degree are usually the center of SPARQL query graph (or regional center). Hence, we try to split the query from the nodes with higher degree. **Note that**, the query Q is decomposed only when the formula: $\max\{T(q_1), T(q_2)\} < 0.8 \cdot T(Q)$ is satisfied. 0.8 is the value we obtained through experiments. Since a Q is decomposed into q_1 and q_2 , an additional join operation must be performed. Next, we will try to decompose the subqueries q_1 and q_2 obtained in this round until the formula cannot be satisfied.

Joining Order There are three types of Join processing trees commonly used in databases: left-deep tree, right-deep tree and bushy tree. In order to improve efficiency as much as possible, and to use the parallel processing power of the computer. it is essential to use a strategy that can create bushy join trees (rather than focusing on left-deep or right-deep trees).

4 Experiments and Evaluation

In the case of not processing the SPARQL query in parallel, we compare our engine with RDF3X in query efficiency. As can be seen from Figure 3, our engine is more efficient than RDF3X even without parallel acceleration. In Figure 4, we tested our engine in a parallel environment. We have found that our engine can indeed improve the efficiency of query execution in a parallel environment.

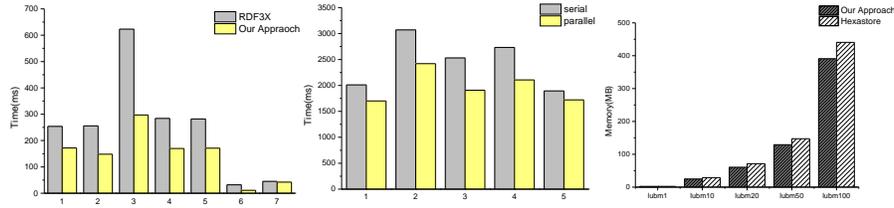


Fig. 3. Efficiency comparison with RDF3x **Fig. 4.** Comparison of Serial and Parallel Efficiency **Fig. 5.** Comparing memory usage with Hexastore

Finally, we compare the memory usage of our storage model with Hexastore [3] on different data sets. Hexastore has six tables to quickly respond to any type of query. Because our method only supports normal SPARQL queries. For equivalence comparison, we only calculated the memory occupancy of three tables in Hexastore. Figure 4 shows that our storage model takes up less memory than Hexastore does.

5 Conclusion

In this paper, we design a highly parallel RDF engine for RDF by our designed RDF storage model, to take less memory and respond faster to queries in a highly parallel way. We believe that our approach is helpful to maximize the performance of limited computing resources.

Acknowledgments

This work is supported by the National Key Research and Development Program of China (2017YFC0908401) and the National Natural Science Foundation of China (61972455,61672377). Xiaowang Zhang is supported by the Peiyang Young Scholars in Tianjin University (2019XRX-0032).

References

1. Hasan R., Gandon F.: A machine learning approach to sparql query performance prediction. In *Proc. of WI 2014*, pp. 266–273.
2. Neumann T., Weikum G.: RDF-3X: A RISC-style engine for RDF. *PVLDB*, 1(1):647–659 (2009).
3. Weiss C., Karras P., Bernstein A.: Hexastore: Sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019 (2008).
4. Zhang X., Zhang M., Peng P., Song J., Feng Z., Zou L.: A scalable sparse matrix-based join for SPARQL query processing. In *Proc. of DASFAA 2019*, pp.510–514.
5. Zou L., Tamer Özsu M., Chen L., Shen X., Huang R., Zhao D.: gStore: A graph-based SPARQL query engine. *VLDB J.*, 23(4):565–590 (2014).