Xu and Zhang *et al. EURASIP Journal on Wireless Communications and Networking* (2019) 2019:256
https://doi.org/10.1186/s13638-019-1588-9

EURASIP Journal on Wireless Communications and Networking

RESEARCH Open Access

Semantic connection set-based massive RDF data query processing in Spark environment

Jiuyun Xu*† and Chao Zhang†

Abstract

Resource Description Framework (RDF) is a data representation of the Semantic Web, and its data volume is growing rapidly. Cloud-based systems provide a rich platform for managing RDF data. However, there is a performance challenge in the distributed environment when RDF queries, which contain multiple join operations, such as network reshuffle and memory overhead, are processed. To get over this challenge, this paper proposes a Spark-based RDF query architecture, which is based on Semantic Connection Set (SCS). First of all, the proposed Spark-based query architecture adopts the mechanism of re-partitioning class data based on vertical partitioning, which can reduce memory overhead and spend up index data. Secondly, a method for generating query plans based on semantic connection set is proposed in this paper. In addition, some statistics and broadcast variable optimization strategies are introduced to reduce shuffling and data communication costs. The experiments of this paper are based on the latest SPARQLGX on the Spark platform RDF system. Two synthetic benchmarks are used to evaluate the query. The experiment results illustrate that the proposed approach in this paper is more efficient in data search than contrast systems.

Keywords: Semantic Web, RDF, Basic graph pattern, Distributed SPARQL query processing

1 Introduction

Due to the rapid development of semantic web and knowledge graph, the amount of data represented by the Resource Description Framework (RDF) [1] has exploded. RDF is a set of knowledge representation model proposed by W3C to describe the content and structure of network resources. Typically, search engines add semantic information to web pages to return accurate query results to users and building large knowledge bases to support smart applications.

SPARQL (SPARQL Protocol and RDF Query Language) [2], recommended by W3C, is one of the standardized languages for RDF data retrieval and query. The SPARQL query statement contains multiple triple patterns, each of which contains one or more variables, and the same variable can exist in multiple triple patterns at the same time. The purpose of our query is to match variables to values from a large number of RDF data.

Over the past decade, due to the relatively small amount of data, RDF data storage and some complex query operations can be handled on a single machine. This usually refers to the operation of traditional RDF data management systems, such as RDF-3X [3], Hexastore [4], and SW-Store [5]. Nowadays, as the amount of RDF data grows, the processing of large-scale RDF data cannot be supported by a single machine. Therefore, both academia and industry begin to explore the scheme of distributed processing by dividing the data into multiple compute nodes and adopting distributed RDF query, e.g., [6, 7], Spark, can be used [8]. In a distributed environment, SPARQL statements are divided into sub-queries based on the multiple triple patterns they contain, that is, one triple pattern is one sub-query. Each sub-query is evaluated and the end result is the intersection of multiple sub-queries. However, the data is distributed among multiple nodes. it may be necessary to exchange data between nodes during query evaluation. Therefore, SPARQL queries that contain

*Correspondence: jyxu@upc.edu.cn
†Jiuyun Xu contributed equally to this work.
School of Computer & Communication Engineering, China University of Petroleum, 66# The Changjiang West Road, 266580 Qingdao, China

a large number of intermediate results incur communication costs that affect query performance.

In this paper, we demonstrate that the system primarily considers querying for large-scale RDF data efficiently in the distributed environment. In addition, we mainly consider the following two problems:

- Storage part, how to reduce memory overhead through partition and index data and achieve a balance between data preprocessing and fast indexing. Different storage and query algorithms directly affect the query efficiency of RDF data.
- Query part, how to reduce SPARQL query processing costs and communication costs. SPARQL query can be viewed as iterative matching and join issues for sub-queries on distributed platforms.

The contributions of this paper are summarized as follows:

- Unlike most existing systems that use a set of permutations of triples (subject, property, object) indexes, a VP-based storage schema is introduced which is for management massive RDF data by further partitioning `rdf:type` predicate based on vertical partitioning (VP) [9]. This strategy is designed to minimize the size of the input data to achieve the goal of reducing memory overhead and supporting the fast indexing.
- Cost estimation and optimization query strategies are presented in this paper. SCS generates query plan and uses broadcast variables method to avoid lots of communication costs. These optimization methods improve the performance of our system.
- We perform an experimental evaluation by comparing this system and two other systems that query RDF data on the distributed platform. We tested the performance of the system on LUBM [10] data sets and WatDiv [11] data sets via standard benchmark queries. The results prove the effectiveness of this system.

The rest of the paper is organized as follows: Section 2 presents the related work. Section 3 mainly introduces basic concepts RDF and SPARQL. Section 4 presents system architecture. In Section 5, we propose a novel data partitioning method. Query processing is covered in Section 6. Section 7 reports the experimental analysis. Finally, we conclude this paper in Section 8.

2 Related work

In recent years, the academic community has done a lot of research work on the RDF data query system. It can be roughly divided into a centralized system running on a single machine and a distributed system running on a cluster.

RDF-3x [3] creates six indexes permutations based on triple (subject, predicate, object) and stores all RDF data under the six indexes. In addition, the statistics of the entity are collected to avoid the cost of self-joining. However, storing six indexes will cause unnecessary data redundancy, and the efficiency of the query is directly related to the size of the main memory. Vertical partitioning (VP) is one of data representations for RDF data proposed by SW-store [5]. The triple table is vertically partitioned into n tables, where n is the number of different predicates. In each predicate, a two-list generated with a row is a pair of subject-object values joined by predicate. With this strategy, it provides good performance for queries with bounded predicates. However, it does not consider the use of special class predicates to achieve finer-grained data, so selectivity is not efficient.

HadoopRDF [12] divides native RDF data into POS index or PSO index and, in addition, classifies predicates based on their attributes of the object. In SPARQL query process, it links the results of multiple triple patterns together through a large number of iterative operations to produce the final result. But the flaws in the Hadoop platform itself cause a lot of intermediate results to be written back to disk, and the greedy algorithm used by the system itself produces unnecessary intermediate results. H2RDF+[13] uses the Hbase database and creates six index tables to store all RDF data. It also maintains index statistics to estimate the selectivity of the triple pattern. Based on the above, in SPARQL query process, H2RDF+ can automatically select whether to execute the query on a single node or on the cluster. However, reading and writing intermediate results to Hbase in distributed mode, which consumes a lot of cost. SPARQLGX [14] is RDF query system based on Spark platform. In terms of data storage, it adopts VP method for data partition and then stores the data into HDFS. In addition, the query optimization of the system is to read all the data for statistics. However, during the query process, the amount of associated data is limited, and excessive statistics can cause poor query performance. S2RDF [15] uses the Spark SQL [16] interface to perform SPARQL queries. First, it adopts VP method for data partition and then performs semi-join processing on these VP tables, and finally generates multiple tables named ExtVP. The above data operation can speed up matching of each triple pattern. During the query process, this system will convert each triple pattern of the query into a corresponding single SQL statement, and the final result is the intersection of the results of each SQL query. However, the data preprocessing step creates significant data loading overhead, which may be two orders of magnitude larger than our solution.

3 Preliminary

3.1 RDF

RDF is a set of knowledge representation model proposed by W3C to describe the content and structure of network resources. It helps search engines to understand the relation of these resources. The underlying structure of the data model is simple and flexible. Any expression in RDF is a collection of triples , including a subject(s), a predicate(p), and an object(o). Subject is a fact. Predicate indicate the relationship between fates, and object may be an entity or literal value, or it may be a class.

Based on this structure, RDF data can be represented as directed graph. RDF graph is a finite set of RDF triples. Figure 1 shows an example of RDF graph.

3.2 SPARQL

SPARQL is one of the standardized languages for RDF data retrieval and query. Its syntax is similar to the syntax of a relational query. SPARQL query usually contains multiple triple patterns (TPs). A set of triple patterns forms a basic graph pattern (BGP). Each tuple contains variables represented as ?v, and information is then queried based on the associated variables for each TP. We summarize the query process as follows: First, match the binding values for each TP, then implement the join of the intermediate results, and finally, generate the final SPARQL results. For example, the SPARQL query statement is shown in Fig. 2.

The SPARQL query graph corresponding to the above SPARQL query is shown in Fig. 3.

4 System architecture

In this section, we will introduce the system architecture. With the rapid growth of RDF data, it is difficult for a single machine node to support data processing. The distributed system has the advantages of low cost, strong fault-tolerance, good stability and expansibility. So, we propose a large-scale RDF data query system based on Spark. Figure 4 shows the architecture of our system. On the whole, the system architecture includes four aspects: data preparation module, persistent data module, query parser module, and distributed processing module.

In this system architecture, the data preparation module is designed to convert RDF data in the form of XML into n-triple format, further divide classes and relationships based on vertical partition, and generate relational index files and class index files. The persistent data module is responsible for loading the index files divided by the above modules into HDFS. The details of the above two modules are described in Section 5. The query parser module is used to generate a query plan based on the SCS optimization strategy, including the triple patterns join order, and the broadcast variable information. Based on the parsing information, we loaded the corresponding index files from HDFS into Spark distributed memory and persisted them. The distributed processing module performs local matching and iterative join operation according to the query plan and finally generates the query result. More details will be introduced in Section 6.

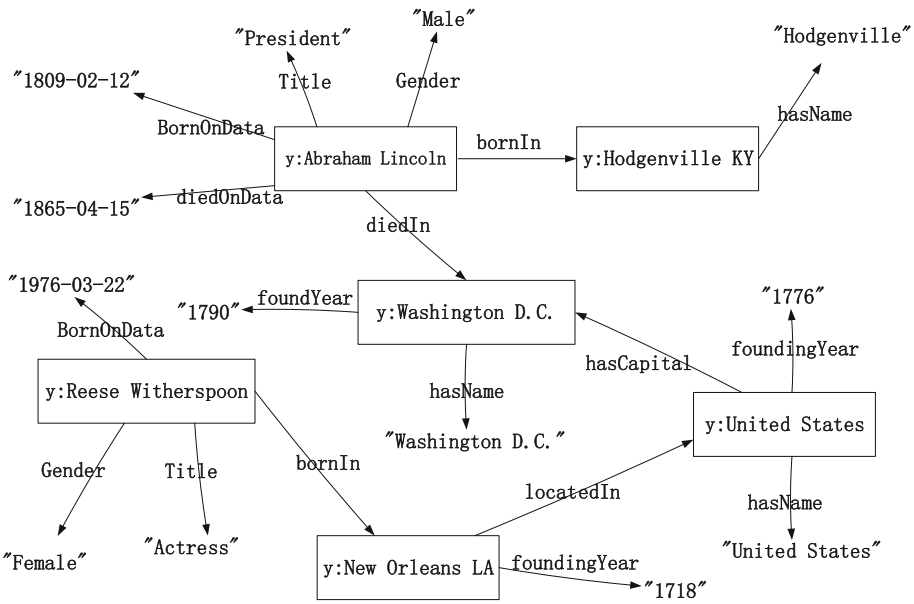

Fig. 1 RDF graph

```
SELECT ?name
WHERE {
  ?m <bornIn> ? city .   ?m <hasName> ?name .
  ?m <bornOnDate> ?bd .
  ? city <foundingYear> "1718" .
 }
```

Fig. 2 SPARQL query statement

5 Data partitioning

In a distributed environment, data partitioning plays a significant role in efficient SPARQL queries. The most straight forward representation of RDF in a relational model is a named triple table with three columns. Generally, for efficient query, it creates a series of indexes because query evaluation can be represented as a series of joins on a large table. For example, well-known system RDF-3X[3], this system creates six indexes permutations based on triple. However, this indexing approach can take up several times the storage space. This can result in memory overhead due to the size of its index files is still large. Many cloud-based systems [17] use VP that it introduced by Abadi et al. in [9] such as [14, 15, 18]. It uses a two-column table instead of three-column table for every RDF predicate and the predicate for the file name. In addition, subject and object are the two columns of data in the index file. The number of predicates in the data set is usually small. Therefore, when SPARQL query includes this predicate, we can quickly retrieve the index file.

Different data partitioning and query algorithms directly affect the efficiency of SPARQL query. In this paper, we propose several design goals:

- Reduce the time required to convert raw data to target data while ensuring finer-grained partitioning schema.
- Reduce the size of input data to avoid the overhead of memory.
- Speed up retrieval of related index files.

Take the LUBM benchmark as an example, which contains 355,823 triples. Figure 5 shows the number and proportion of each predicate. We can see that the number of predicate of `rdf:type` is at most. Therefore, further partitioning of this predicate will speed up the related indexes. Thus, we introduce a storage schema for management massive RDF data by further partitioning `rdf:type` predicate based on VP. In general, VP uses a two-column table for every RDF predicate, e.g., work for(s,o). On this basis, we further divide the triples with predicate of type. According to the triple's object representing a specific class, we divide them into small class files. Tables 1, 2, and 3 show the partitioning of the type predicate into smaller index files (class index files). We stored the partitioned data into the file system of Hadoop (HDFS)[19].

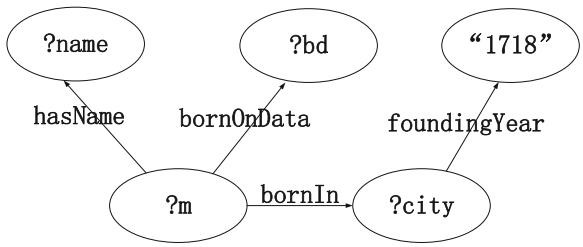

Fig. 3 SPARQL query graph

This data partitioning method allows the system to quickly match each triple pattern by selecting the relevant small index file when executing the SPARQL query, which reduces the cost of reading indexes and avoids the overhead of memory. In addition, the data compression performance is excellent because the data is not stored in the triple form. We will save two thirds of the RDF data storage space.

6 Query processing

In this section, we will introduce the cost estimation and then triple patterns matching based on Spark, and finally, the query optimization strategies based on the cost estimation.

6.1 Cost estimation

From the above introduction, we can divide SPARQL query into two aspects: triple patterns matching and join intermediate results. So, we define the first part as parse TPs, and the second part as join IRs. The cost of parsing TP includes the cost of reading related index files and matching TP. The cost of joining IRs includes shuffle communication costs and computing costs.

$$\text{Cost} = \sum_{i=1}^{n} \text{Parser}\,(\text{TP}_i) + \sum_{j=2}^{n} \text{Join}\left(IR_{j-1}, \text{Match}\,(TP_i)\right) \tag{1}$$

$$\text{Parse}\,(TP_i) = \text{Read}\,(TP_i) + \text{Match}\,(TP_i) \tag{2}$$

$$\text{Join}\,(IR_1, IR_2) = \text{Shuffle}\,(IR_1, IR_2) + \text{Compute}\,(IR_1, IR_2) \tag{3}$$

$$\text{IR}_i = \begin{cases} \text{join}\,(IR_{i-1}, \text{Match}\,(TP_i))\,, & 2 \le i \le n \\ \text{Match}\,(TP_i)\,, & i = 1 \end{cases} \tag{4}$$

where,
n = number of TP.

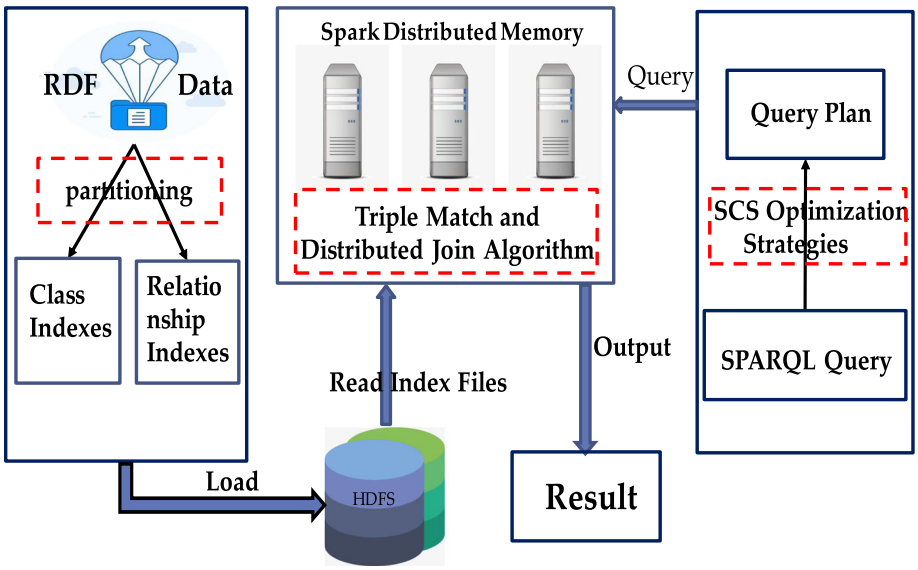

Fig. 4 System architecture

TP_i = the ith triple pattern in query.
Read (TP_i) = load the relevant index file.
Match (TP_i) = the result of matching TP_i.
Shuffle (IR_1,IR_2) = the data that needs to be moved in a distributed environment.
Compute (IR_1,IR_2) = implement the join operation.
IR_i = the IR of TP_i.
Equation 1 estimates the overall cost of a SPARQL query. Equation 2 specifically estimates cost of parsing triple patterns. Equation 3 represents the cost of performing join operation. Equation 4 represents iterative computation of IRs. Therefore, from the perspective of total cost estimation, we reduce the cost of loading data and TP matching through data partitioning. In addition, the larger the size of the matching result, the higher the cost of the join, so we can reduce the connection cost by reducing the size of IRs and data communication costs.

6.2 Triple patterns matching

Spark [20] is a in-memory cluster computing system. Compared to map-reduce-based systems [21–23], our SPARQL query systems based on Spark does not need to write intermediate results back to disk, which causes a large number of disk I/O problems. Instead, they are cached in memory to avoid disk I/O costs. The example showed in preliminary Section 2, the BGP contains

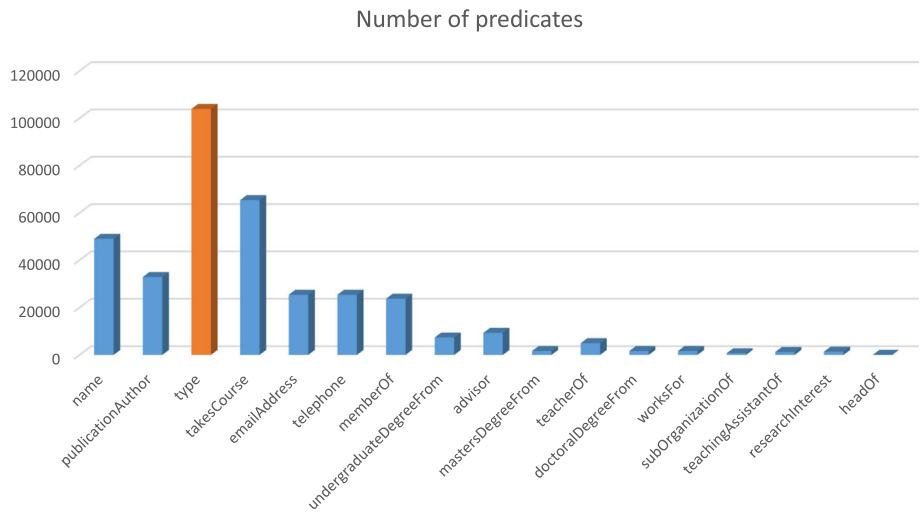

Fig. 5 Sample number of each predicates

Table 1 Sample of predicate type

Predicate:type	
Harvard	University
MIT	University
Cambridge	University
New York	City
Los Angeles	City
Beijing	City

a set of TPs. For this, the purpose of SPARQL query is to find the values of all variables. Since each TP is a sub-query, so SPARQL query operation can be viewed as TPs matching and sub-query iterative join. Calculating the binding for all variables means matching the variables in each triple pattern separately. Jena ARQ [24] is used to parse SPARQL query to generate the corresponding triple patterns. Each triple contains constants and variables, in which the variables contain ? of special characters. When the predicate is not a variable, we can obtain the relevant data of each tuple according to our previous data partitioning strategy, and further filtering-related data based on whether the subject and object are constants and using common operators in Spark. Then, we will count the size of the matching result and use it in the next optimization strategy.

For example, in the former Section 2 showed, the tuple {?*city* ⟨*foundingYear*⟩ 1718.}, where *foundingYear* and 1718 are constants, human reads the index file in the file system based on the fact that the predicate *foundingYear*, and then filter the related data just read based on the fact that object mean a number of 1718. After each triple pattern is matched to the result, iterative join according to query plan we will describe in detail in the query optimization section.

The triple matching algorithm is showed in Algorithm 1. From general viewpoint, line 1 through line 8 represent the case where the predicate is constant, and 9 through 10 represent the case where the predicate is variable. Line 2 through line 6 consider the special case where the predicate is type. Line 11 through line 16 represent triples that are filtered by the given subject or object.

Table 2 Spilt1

Indexname:University
Harvard
MIT
Cambridge

Table 3 Spilt2

Indexname:City
New York
Los Angeles
Beijing

Algorithm 1: Triple Matching Algorithm

```
Input: tp:(s,p,o)
Output: IR
1  if p is not variable then
2      if p is rdf:type then
3          if o is not variable then
4              tq = spark.textFile(o.txt);
5          else
6              tq = spark.textFile(p.txt);
7      else
8          tq = spark.textFile(p.txt);
9  else
10     tq = spark.textFile(T.txt);
11 if s is not variable then
12     tq = tq.Filtercase(subject, object) => subject.equals(s);
13     size = tq.count;
14 if o is not variable then
15     tq = tq.Filtercase(subject, object) => object.equals(o);
16     size = tq.count;
17 new IR(tq, size);
18 return IR
```

6.3 Query optimization

In the cost estimation, we mentioned reducing the cost of join by reducing the size of the results in the process and data communication. Due to BGP query contains multiple triples, we join them based on shared variables. But in the process of query, different connection order has different efficiency on query result. Therefore, we propose a SCS optimization strategy to generate the join order in the RDF query.

6.3.1 Semantic connection set

The join order of SPARQL sub-queries has a significant impact on query performance, so the semantic connection set (SCS) optimization method needs to be built. The SCS contains multiple intermediate results (IRs) obtained after multiple TP matches and then sorted in ascending order based on the size of matching result. The size of IRs in the initial set is statistical in Subsection 6.2, and then, the two smaller intermediate results that contain common variables are selected to join first, and the generated results are added to the set. Remove the previously connected IRs and sort by size. Iterate join through the intermediate results until only one result remains in the set, which is the final result of the SPARQL query. Finally, we return the columns of interest to the user. We use the SCS method to generate an optimized query plan to improve the performance of RDF queries. This approach will reduce the IR size to reduce I/O cost and reduce the total number of connection comparisons to improve system performance.

As shown in Algorithm 2, line 2 represents the smallest IR from the set of semantic connection. Line 3 to line 6 indicate that the matching results in the set that contains the same variable and the smallest IR are extracted. Lines 8 through 10 represent the two result sets that implement

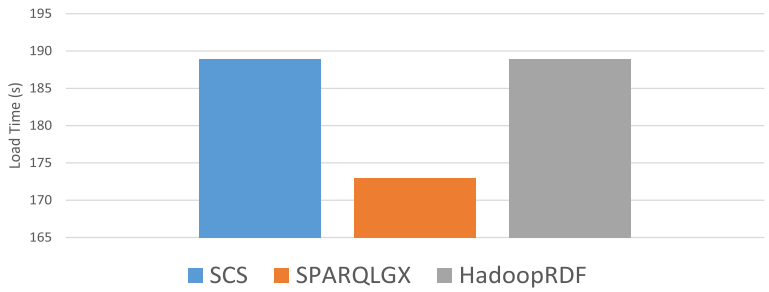

Fig. 6 load time over LUBM100

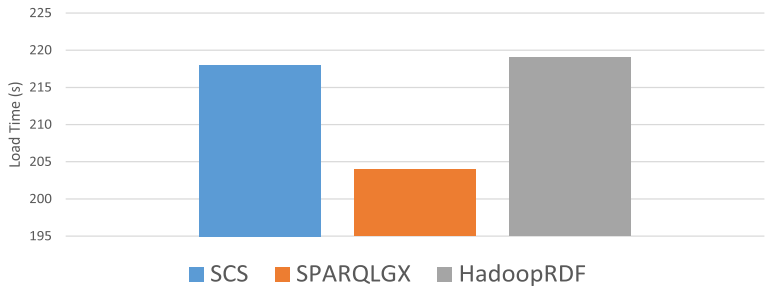

Fig. 7 Load time over WatDiv10M

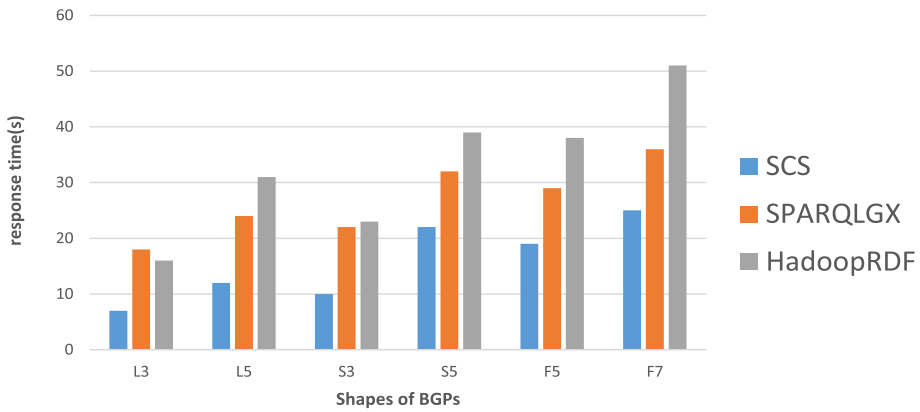

Fig. 8 Query time over LUBM100

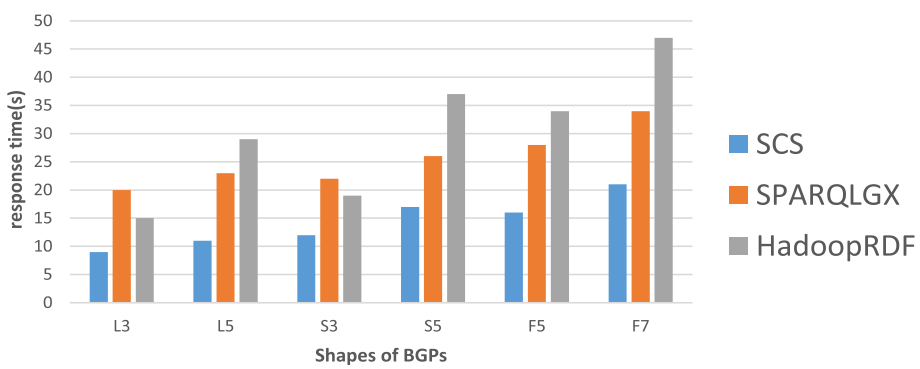

Fig. 9 Query time over WatDiv10M

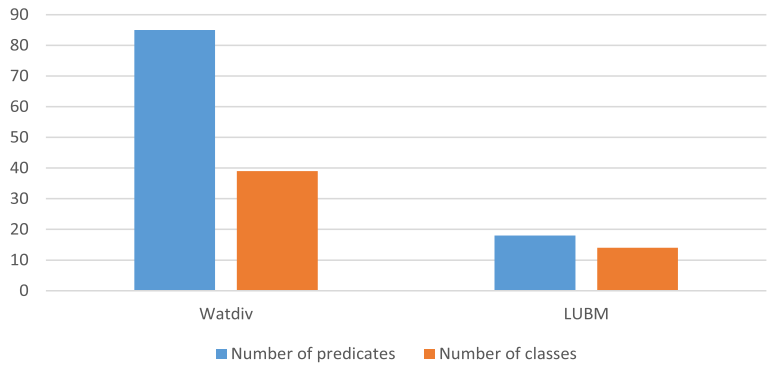

Fig. 10 Number of predicate category

the join operation and remove it from the set, after that the join result is loaded into the connection set.

Algorithm 2: Connection Set Algorithm

```
Input: List(IRs)
Output: Query Result
1  while list.size > 1 do
2      IR1 = list.minby(_.getSize);
3      canJoin = list.filter(rdd =>
         rdd.ne(IR1)&&rdd.getVarSet.intersect(IR1.getVarSet).nonEmpty);
4      if canjoin.isEmpty then
5          print("can not join into one");
6      IR2 = canjoin.minby(_.getSize);
7      broadcast(IR2);
8      join = IR1.join(IR2);
9      list.delete(IR1, IR2);
10     list.add(join);
11 QueryResult = list.head.getTriple;
12 return Query Result
```

In addition, in line 7 of the algorithm, we use the method of broadcasting variables to reduce the network cost in data communication. When doing the join intermediate result operation, we compress the data of the smaller result set A and broadcast it to the node of the result set B and make a local connection. This operation can reduce a certain amount of network communication cost caused by data shuffle.

7 Experiments

In this section, we will describe the performance evaluation of the SCS. The experiment is implemented on a cluster with five machines. Each node with an Inter Xeon E5-2670 CPU @2.6GHz,4 cores, 16GB RAM running Ubuntu 16.04.5 LTS with the software Scala2.11.1 [25], Hadoop2.7.3, and Spark2.1.0. A variety of experimental data sets are proposed in [26]. In our experiment, we used two synthetic benchmarks, LUBM [10] and WatDiv [11], to evaluate our system and two other comparison systems, and we evaluated the query response time on the above data sets with standard WatDiv and LUBM queries to prevent some queries absence of a final result. In addition, the BGPs in the SPARQL query can have different shapes. According to the position of the variables in the TPs, they can be divided into linear (L), star (S), and snowflake (F).

Similar to our system and typical RDF query engines based on distributed environments are HadoopRDF, S2RDF, and SPARQLGX. Graux et al. [14] shows that SPARQLGX performs better in both the preprocessing and query stages than S2RDF. In addition, the data partitioning methods used by the SPARQLGX and HadoopRDF systems are similar to those in this paper, so our experimental results will be compared with them.

Figures 6 and 7 illustrate the load time of the two data sets. Notably, SPARQLGX method is the fastest among three of them no matter the type of data sets. The loading time of HadoopRDF and SCS is very close. This is because SPARQLGX only uses VP to process data, and HadoopRDF and SCS continue to process data based on this. In addition, the loading time of the three systems under the WatDiv data set is longer than that under the LUBM data set, because the WatDiv data set contains more predicates and is more time-consuming to process.

Relative to the load time of the data, we are more concerned about the response time of the query. Next, we compare the three systems based on the response time of the three query types.

For the data set LUBM100 with 12 million tuples of data, the experimental results of our execution of the standard query are shown in Fig. 8.

The numbers after the letters, such as L3, represent a triple pattern in a SPARQL query that contains the corresponding numbers. The performance comparison between the three systems is shown in Fig. 8, In the L3 query phase, because the shape of BGPs is not complicated and the number of triple patterns that need to be connected is small, the HaoopRDF system with better algorithm efficiency is lower in response time than SPARQLGX. However, as the number of connection tasks increases, the disadvantages of Hadoop framework are gradually revealed. The query response time of HadoopRDF is higher than that of two systems based on Spark.

In short, we can see that our system is superior to the other two systems in query response time regardless of the query type or the number of triples contained. There are three reasons for this: (i) memory computing:

Spark framework is based on memory calculation, which processes the data much faster than Hadoop. For example, Spark puts the intermediate data into memory, so the iterative operation is efficient. But MapReduce saves the results to disk, which affects the overall speed. (ii) Finer-granined partitioning schema: our system matches a TP by inputting a smaller index file than SPARQLGX, which reduces the size of intermediate results. (iii) Optimal query plan: during the execution of the query plan by HadoopRDF, many unnecessary IRs were generated because the joining selectivity was not considered, but our system optimized the query plan based on the SCS to reduce the IR size and thus reduce the cost of the join operation. In addition, compared with SPARQLGX, the broadcast variables added to the query plan algorithm developed by our system can avoid a large amount of communication costs, thereby improving query efficiency.

For the data set WatDiv10M with 10 million tuples of data, the experimental results of our execution of the standard query are shown in Fig. 9.

As shown in Fig. 9, we can see that our system has better efficiency than SPARQLGX and HadoopRDF in all the types of standard queries. In addition, the difference between LUBM and WatDiv data sets is the number of predicates, where LUBM contains 17 different predicates and WatDiv 86 different predicates. In this system, the data with predicate `rdf:type` is further divided, as shown in the Fig. 10, in which WatDiv divides more index files than LUBM. Therefore, when evaluating queries under the same size data set, the intermediate results produced under the LUBM data set may be larger than the intermediate results produced by the WatDiv data set.

8 Conclusion

In this paper, we introduce the SCS, an RDF query processing engine based on Spark. We present a schema for further partitioning data with predicate of `rdf:type` based on VP to avoid the overhead of memory and speed up indexing. Then, the intermediate results size affects the performance of the system, so, a SCS method is built to handle the query process in a distributed environment. We propose a cost model and other optimization strategies to specify the query order to speed up the response time. For future work, we will increase the filtering of extraneous RDF data to further reduce the amount of data read and investigate more efficient query join algorithm.

Abbreviations

BGP: Basic graph pattern; F: Snowflake; IRs: Intermediate results; L: Linear; RDF: Resource Description Framework; S: Star; SCS: Semantic Connection Set; SPARQL: SPARQL Protocol and RDF Query Language; TPs: Triple patterns; VP: Vertical partitioning

Authors' contributions

CZ conceived the idea. JX conducted the analyses. All authors contributed to the writing and revisions. All authors read and approved the final manuscript.

Availability of data and materials

The data sets used or analyzed during the current study are available from the corresponding author on reasonable request.

Competing interests

The authors declare that they have no competing interests.

Received: 28 June 2019 Accepted: 30 October 2019
Published online: 27 November 2019

References

1. E. Miller, An introduction to the resource description framework. Bulletin Am. Soc. Inf. Sci. Technol. **25**(1), 15–19 (1998)
2. J. Pérez, M. Arenas, C. Gutierrez, Semantics and complexity of SPARQL. ACM Trans. Database Syst. (TODS). **34**(3), 16 (2009)
3. Neumann, Thomas, Weikum, Gerhard, The RDF-3x engine for scalable management of RDF data. Vldb J. **19**(1), 91–113 (2010)
4. C. Weiss, P. Karras, A. Bernstein, Hexastore: sextuple indexing for semantic web data management. Proc. Vldb Endowment. **1**(1), 1008–1019 (2008)
5. D. J. Abadi, A. Marcus, S. R. Madden, K. Hollenbach, SW-Store: a vertically partitioned DBMs for Semantic Web data management. Vldb J. **18**(2), 385–406 (2009)
6. L. Qi, X. Zhang, W. Dou, Q. Ni, A distributed locality-sensitive hashing-based approach for cloud service recommendation from multi-source data. IEEE J. Sel. Areas Commun. **35**(11), 2616–2624 (2017). https://doi.org/10.1109/JSAC.2017.2760458
7. A. Madkour, A. M. Aly, W. G. Aref, in *The Semantic Web – ISWC 2018*, ed. by D. Vrandečić, K. Bontcheva, M. C. Suárez-Figueroa, V. Presutti, I. Celino, M. Sabou, L.-A. Kaffee, and E. Simperl. WORQ: workload-driven RDF query processing (Springer, Cham, 2018), pp. 583–599
8. G. Agathangelos, G. Troullinou, H. Kondylakis, K. Stefanidis, D. Plexousakis, in *2018 IEEE 34th International Conference on Data Engineering Workshops (ICDEW)*. RDF query answering using Apache Spark: review and assessment (IEEE, 2018), pp. 54–59. https://doi.org/10.1109/icdew.2018.00016
9. D. J. Abadi, A. Marcus, S. R. Madden, K. Hollenbach, in *Proceedings of the 33rd International Conference on Very Large Data Bases*. Scalable semantic web data management using vertical partitioning (VLDB Endowment, 2007), pp. 411–422
10. Y. Guo, Z. Pan, J. Heflin, LUBM: a benchmark for OWL knowledge base systems. Soc. Sci. Electron. Publ. **3**(2), 158–182 (2005)
11. G. Aluç, O. Hartig, M. T. Özsu, K. Daudjee, in *International Semantic Web Conference*. Diversified stress testing of RDF data management systems (Springer, 2014), pp. 197–212. https://doi.org/10.1007/978-3-319-11964-9_13
12. M. Husain, J. McGlothlin, M. M. Masud, L. Khan, B. M. Thuraisingham, Heuristics-based query processing for large RDF graphs using cloud computing. IEEE Trans. Knowl. Data Engineer. **23**(9), 1312–1327 (2011)
13. N. Papailiou, I. Konstantinou, D. Tsoumakos, P. Karras, N. Koziris, in *2013 IEEE International Conference on Big Data*. H 2 RDF+: high-performance distributed joins over large-scale RDF graphs (IEEE, 2013), pp. 255–263. https://doi.org/10.1109/bigdata.2013.6691582
14. D. Graux, L. Jachiet, P. Geneves, N. Layaïda, in *International Semantic Web Conference*. SPARQLGX: efficient distributed evaluation of SPARQL with Apache Spark (Springer, 2016), pp. 80–87. https://doi.org/10.1007/978-3-319-46547-0_9
15. A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic, G. Lausen, S2RDF: RDF querying with SPARQL on Spark. Proc. VLDB Endowment. **9**(10), 804–815 (2016)
16. M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al., in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. Spark SQL: relational data processing in Spark (ACM, 2015), pp. 1383–1394. https://doi.org/10.1145/2723372.2742797
17. Z. Kaoudi, I. Manolescu, RDF in the clouds: a survey. VLDB J. Int. J. Very Large Data Bases. **24**(1), 67–91 (2015)
18. A. Schätzle, M. Przyjaciel-Zablocki, T. Hornung, G. Lausen, in *Proceedings of the 12th International Semantic Web Conference (Posters & Demonstrations Track)-Volume 1035*. PISPARQL: a SPARQL query processing baseline for big data, (2013), pp. 241–244. CEUR-WS. org

19. K. Shvachko, H. Kuang, S. Radia, R. Chansler, in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. The Hadoop distributed file system, (2010), pp. 1–10. IEEE Computer Society
20. M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, Spark: cluster computing with working sets. HotCloud. **10**(10-10), 95 (2010)
21. K. Rohloff, R. E. Schantz, in *Programming Support Innovations for Emerging Distributed Applications*. High-performance, massively scalable distributed systems using the MapReduce software framework: the shard triple-store (ACM, 2010), p. 4. https://doi.org/10.1145/1940747.1940751
22. X. Zhang, L. Chen, M. Wang, in *International Conference on Scientific and Statistical Database Management*. Towards efficient join processing over large RDF graph using MapReduce (Springer, 2012), pp. 250–259. https://doi.org/10.1007/978-3-642-31235-9_16
23. X. Zhang, L. Chen, Y. Tong, M. Wang, in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. EAGRE: towards scalable i/o efficient SPARQL query evaluation on the cloud (IEEE, 2013), pp. 565–576. https://doi.org/10.1109/icde.2013.6544856
24. B. Mcbride, Jena: a semantic web toolkit. IEEE Internet Comput. **6**(6), 55–59 (2002)
25. M. Odersky, L. Spoon, B. Venners (2011). http://blog.typesafe.com/why-scala. (last accessed: 28 Aug 2012)
26. I. Abdelaziz, R. Harbi, Z. Khayyat, P. Kalnis, A survey and experimental comparison of distributed SPARQL engines for very large RDF data. Proc. Vldb Endowment. **10**(13), 2049–2060 (2017)

Publisher's Note