Article

Massive RDF Query Processing Efficiently in Spark Environment Based on Semantic Connection Set

Jiuyun Xu [1,†,‡], Chao Zhang [1,‡]

[1] College of Computer and Communication Engineering, China University of Petroleum (East China),Tsingdao 266580, China;jiuyun.xu@computer.org(X.J.);czhang0815@gmail.com(C.Z.)
* Correspondence:jyxu@upc.edu.cn;
‡ These authors contributed equally to this work.

Abstract: Resource Description Framework(RDF) is a data representation format of the Semantic Web, and its data volume is growing rapidly. Cloud-based systems provide a rich platform for managing RDF data. However, the distributed environment has performance challenges when it is processing with RDF queries that contain multiple join operations, such as network reshuffle, memory overhead. To get over these challenges, this paper proposed a spark-based RDF query architecture, which is based on Semantic Connection Set (SCS). First of all, this spark-based query architecture adopts the mechanism of re-partitioning class data based on vertical partitioning, which can reduce memory overhead and fast index data. Secondly, a method for generating query plans based on semantic connection sets is proposed in this paper. In addition, statistics and broadcast variable optimization strategies are used to reduce shuffling and data communication costs. The experiment of this paper is based on the latest SPARQLGX on the spark platform RDF system, two synthetic benchmarks are used to evaluate the query. The experiment result illustrates that the proposed approach in this paper is more efficient in data search than SPARQLGX.

Keywords: RDF; Semantic Web; Basic Graph Pattern; Distributed SPARQL Query Processing; Spark

1. Introduction

With the development of Web 3.0 and knowledge graph, the data represented by the Resource Description Framework(RDF)[1] is increasing rapidly. There are many reasons for this phenomenon ,such as search engine like Google add semantics to web pages to improve query accuracy .In addition, a growing number of communities driven projects are constructing large knowledge bases with billions of facts in many domains to implement more useful applications, for example DBpedia, Probase and so on.

SPARQL(SPARQL Protocol and RDF Query Language)[2] recommended by W3C is the main query language for RDF data. The SPARQL query sentences contain a cluster of triple patterns. The system searches triple that match on the conditions. One variable can be in multiple triple patterns, the system has to check all the conditions about a variable.

During the past decade, most traditional RDF management system executed in a single machine can not handle large-scale RDF data and answer complex query. But to handle large-scale RDF data, several methods are presented in terms of storage strategies and query strategies, such as RDF-3X[3],Hexastore[4],SW-Store[5]. Now industry dealing with large-scale data storage usually considers a distributed environment by partitioning RDF data in many compute nodes and evaluating queries in a distributed fashion,e.g., Spark, can be used [6]. SPARQL query is decomposed into multiple subqueries that are evaluated by each node independently. Since data is distributed, the nodes may

need to exchange intermediate results during query evaluation. Consequently, queries with large intermediate results incur high communication cost, which is detrimental to the query performance.

In this paper, we demonstrate that the system is mainly consider querying for large-scale RDF data efficiently in the distributed environment. The main problem are considered about the following two points:

- 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 access methods directly affect the efficiency of the query.
- Query part, how to reduce SPARQL query processing costs and communication costs. The SPARQL query processing can be transformed to the problem of iterative matching and joining of sub-queries.

The contributions of this paper are summarized as follows:

- Unlike existing most systems that use a set of permutations of (S,P,O) indexes, we introduce a VP-based storage schema for management massive RDF data by further partitioning `rdf:type` predicate based on Vertical Patitioning(VP)[7]. The 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. Semantic connection set generates query plan and uses broadcast variables method to avoid lots of communication costs. These optimizations ensure high performance for our system.
- We perform an experimental evaluation by comparing this system and current state-of-the-art SPARQLGX that query RDF data on the Spark platform. We test the performance of current works over LUBM[8] datasets and Watdiv[9] datasets via standard benchmark queries. The results prove the effectiveness of this system.

The rest of this paper is organized as follows: section2 presents the related work. section 3 mainly introduces basic knowledge RDF data and SPARQL queries. The section 4 mainly introduces system architecture. The section 5 mainly introduces data preparation. The section 6 mainly introduces query processing. The section 7 is the experimental analysis, with latest distributed query SPARQLGX system for comparison. Finally, we conclude in section 8.

2. Related Work

In recent years, many rdf systems capable of evaluating sparql queries have been developed. These stores can be divided in two categories: centralized systems and distributed ones, running on single machine or on a computing cluster. RDF-3x[3] creates an exhaustive set of indexes for all RDF triple permutations and aggregates indexes for subsets, resulting in a total of 15 indexes stored in compressed clustered B trees. Besides, it also employs additional indexes to collect statistical information for pairs and stand-alone entities to eliminate the problem of expensive self-joins and provides great performance improvement. However, storing all the indexes is expensive and query efficiency highly depends on the amount of main memory. VP is another representation for RDF data proposed by SW-Store[5]. The triples table is vertically partitioned into n tables, where n is the number of distinct predicates. A two columns table is created for each predicate where a row is a pair of subject-object values connected through the predicate. It provides good performance for queries with bounded predicates. However, it does not consider special class predicates to achieve further fine-grained data. HadoopRDF[10] partitions the input RDF graph to create a pos or pso index. Then the predicate files are splitted into smaller files according to the type of objects. It performs the SPARQL query by a serious of iterative MapReduce jobs, each of which implements a join between two TPs on a variable. However, its use of greedy algorithms to reduce the number of MapReduce connections required for each step raises many unnecessary intermediate results. H2RDF+[11] stores and indexes

RDF data in HBase. Then it implements MapReduce-based multi-way Merge and ,Sort-Merge join algorithms to process SPARQL query. However, since each MapReduce job requires a given input and output, it will take a lot of time to read and write IRs in HBase. SPARQLGX[12] is designed to leverage existing Hadoop infrastructures for evaluating sparql queries. It uses VP to process data and then store it in HDFS. In addition, the system's query optimization statistics is to read all the data for statistics, which ignores the query information only needs to calculate the size of the query associated data. S2RDF[13] is built on Spark and uses its SQL interface to execute SPARQL queries. Its main goal is to address efficiently all SPARQL query shapes. Its data layout corresponds to the VP approach. So-called ExtVP relations are computed at data load-time using semi-joins, to limit the number of comparisons when joining triple patterns. Considering query processing, each triple pattern of a query is translated into a single SQL query and the query performance is optimized using the set of statistics and additional data structures computed during this pre-processing step. but, the data pre-processing step generates an important data loading overhead which might be up to 2 orders of magnitude larger than our solution.

3. Preliminary

3.1. RDF

RDF is the W3C recommended standard model for data interchange on the Web. 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 or a class of resources. Predicate denotes relationship associated between fates or classes, and object is an entity,a class, or a literal value.

According this structure, the RDF dataset represents the information of the semantic web as a directed graph. RDF graph is a finite set of RDF triples. Figure 1 shows an example of RDF graph. Ellipse nodes represent resource, edges represent relationship, and rectangular nodes represent literal values.

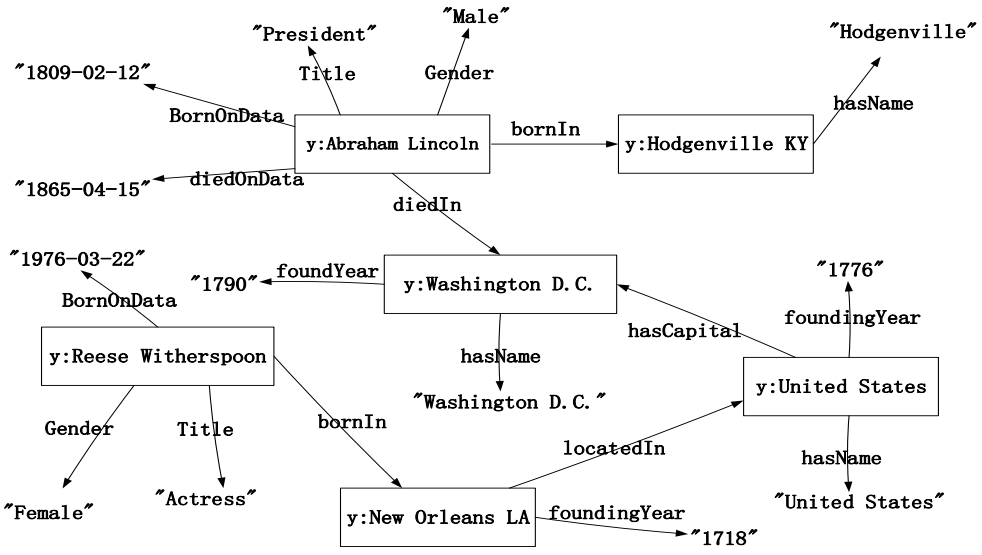

Figure 1. RDF Graph

3.2. SPARQL

SPARQL is the W3C recommended query language for RDF. 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, then a group of queries corresponds to a group of n subqueries, each subqueries contains a triple, according to relevant variable of the triple can query information. We summarize the query process as follows: compute binding values for every TP, implement joins intermediate result, and build the final result of the query. For example, the SPARQL query statement is shown in Figure 2.

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

Figure 2. SPARQL Query Statement

Corresponding to above SPARQL query, the SPARQL query graph is shown in Figure 3.

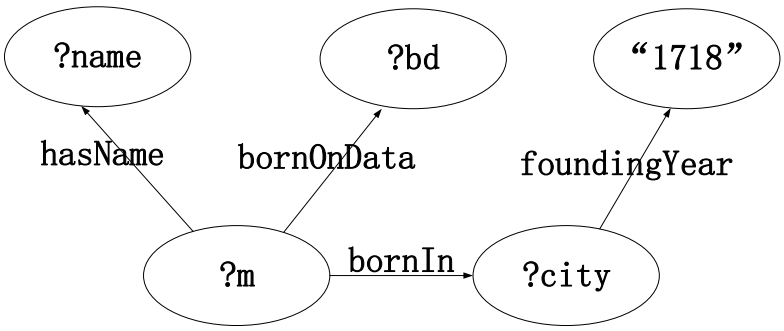

Figure 3. SPARQL Query Graph

4. System Architecture

In this section, we will introduce the system architecture. For the massive RDF dataset, a single node is difficult to support access data. Distributed system can precede of the low cost, fault tolerance, stability, scalability. Therefore, we propose a system for querying large-scale RDF data based on Spark. Figure 4 shows the architecture of the our system. On the whole, the system structure includes four aspects: data preparation module, persistent data module, query parser module and distributed processing module.

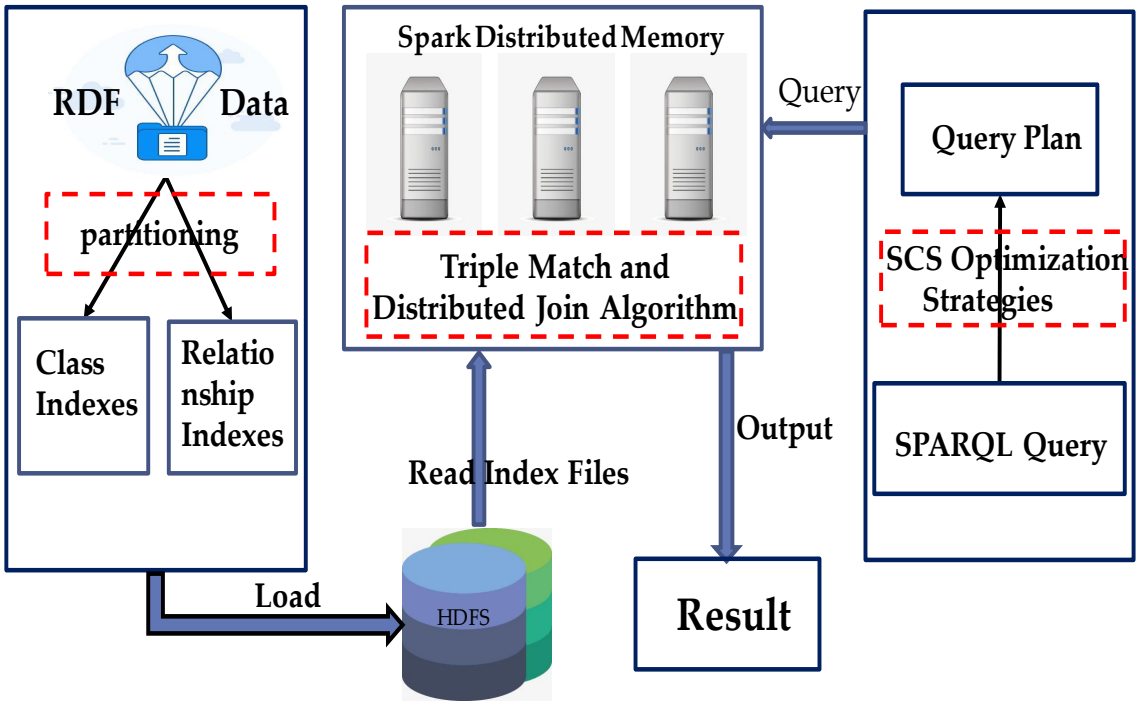

Figure 4. System Architecture

In this system architecture, the data preparation module is designed to convert RDF data in the form of XML into n-triples 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.

5. Data Partitioning

Data layout plays a significant role for efficient SPARQL queries in a distributed environment. The most straight forward representation of RDF in a relational model is a named triple table with three columns, containing one row for each RDF triple(s,p,o). Generally for efficient query, it will be accompanied by several indexes over some or six triple permutations as query evaluation essentially boils down to a series of a join on this large table. For example, well-known system RDF-3X[3], this system creates a set of indexes on RDF data. However, this indexing approach can take up several times the storage space. Since the size of its indexes files is still large which will lead to the overhead of memory. Many cloud-based systems [14] use VP that it introduced by Abadi et al in [7]. such as [15],[13],[12]. It uses a two-column table instead of three-column for every RDF predicate. The predicate is the name of the file, subject and object is the two columns of data in the file. In the RDF data, the number of predicates is generally small. So when we retrieve a triple containing this predicate, we can quickly find the corresponding index file.

Different data partitioning and access methods directly affect the efficiency of the 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.
- Fasts retrieval of related index files.

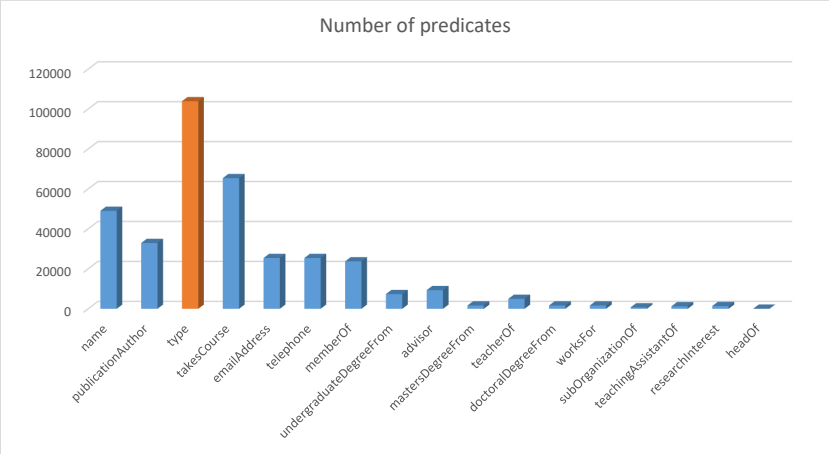

Figure 5. Sample Number of Each Predicates

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-columns table for every RDF predicate, e.g.workfor(s,o). On this basis, we further divide the triples with predicate of type. We divide them into small class files based on the triple's object representing a specific class. The instances belonging to the same class are stored in corresponding class index files. Object will be the name of the file. Table 123 shows 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)[16].

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

As important as data partitioning, the strategies for processing SPARQL query are focused on searches. 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.

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

Table 1. Sample of Predicate Type

Indexname:University
Harvard
MIT
Cambridge

Table 2. Spilt1

Indexname:City
New York
Los Angeles
Beijing

Table 3. Spilt2

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.

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

$$Parse\left(TP_i\right) = Read\left(TP_i\right) + Match\left(TP_i\right) \tag{2}$$

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

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

Where,

n = number of TP in SPARQL query.

Read(TP_i) = load the relevant index file.

Match(TP_i) = the result of matching TP_i.

shuffle(IR_1,IR_2) = the size of data that needs to be moved in a distributed environment.

compute(IR_1,IR_2) = the size of data required to 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 the cost of parsing a triple pattern. Equation 3 represents the cost of implementing the 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 matching TP through data partitioning. In addition, the cost of joining is roughly proportional to the size of their matching result, so we can reduce the connection cost by reducing the size of intermediate results and data communication costs.

6.2. Triple Patterns Matching

Spark[17] is a general-purpose in-memory cluster computing system that runs on Hadoop and can process data from any Hadoop data source. Compared to map-reduce-based systems[18][19][20], 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. In the example query showed in preliminary section 2, the BGP contains a set of triple patterns. For this, The goal of query is to compute the bindings for all variable. As every TP can be regarded as a sub-query, the problem of SPARQL query processing can be transformed to the problem of triple patterns matching and iterative joining of sub-queries. Calculating the binding for all variables means matching the variables in each triple pattern separately. Jena ARQ[21] is used to parse SPARQL query to generate the corresponding triple patterns. Each triple contains three parts of the subject, predicate, and object, including constants and variables, in which the variables contain ? of special characters. When the value of the predicate is constant, 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, we read 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 is 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.

Algorithm 1: Triple Matching Algorithm

```
Input: tp:(s,p,o)
Output: IR
if p is constant then
    if p is rdf:type then
        if o is constant then
            tq = spark.textFile(o.txt);
        else
            tq = spark.textFile(p.txt);
    else
        tq = spark.textFile(p.txt);
else
    tq = spark.textFile(D.txt);
if s is constant then
    tq = tq.Filtercase(s, o) => s.equals(constant);
    size = tq.count;
if o is constant then
    tq = tq.Filtercase(s, o) => o.equals(constant);
    size = tq.count;
new IR(tq, size);
return IR
```

The triple matching algorithm is showed in algorithm 1. In summary, 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.

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 according to the size of IR. 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 intermediate result size to save I/O and reduce the total number of connection comparisons to save CPU.

As shown in algorithm 2, the line 2 represents the smallest IR from the set of semantic connection. From line 3 to line 6 indicate that the matching results in the set that contains same variable and the smallest IR are extracted. Lines 8 through 10 represent the two result sets that implement the join operation and remove it from the set, after which the join result is loaded into the connection set.

Algorithm 2: SCS Algorithm

Input: List(IRs)
Output: Query Result

```
1  while list.size > 1 do
2      IR_1 = list.minby(_.getSize);
3      canJoin = list.filter(rdd =>
         rdd.ne(IR_1)&&rdd.getVarSet.intersect(IR_1.getVarSet).nonEmpty);
4      if canjoin.isEmpty then
5          print("can not join into one");
6      IR_2 = canjoin.minby(_.getSize);
7      broadcast(IR_2);
8      join = IR_1.join(IR_2);
9      list.delete(IR_1, IR_2);
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 small 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[22], Hadoop2.7.3 and Spark2.1.0. A variety of experimental data sets are proposed in [23] .In our experiment, we compare the presented systems using two synthetic benchmarks: LUBM [8] and Watdiv[9].Test dataset are WatDiv10M and LUBM100. WatDiv10M contains 10 million triple data and LUBM100 contains 12 million triple data. We test the query time over the datasets above via the standard Watdiv and LUBM queries to prevent some queries absence of a final result. In addition, we classify queries into three types:linear (L), star (S), snowflake (F).

Our system runs in a distributed environment, and the latest RDF distributed query engines based on Spark are S2RDF and SPARQLGX. [12] shows that SPARQLGX performs better in both the preprocessing and query stages than S2RDF. In addition, the SPARQLGX system adopts the method of VP similar to our system, so our experimental results are mainly considered for comparison with it.Since our system only further divides type predicate compared with SPARQLGX in data partitioning, the comparison between the two in data preprocessing can be ignored. We compare the two systems in terms of response time for the three query types.

For dataset LUBM100, our experiments run standard queries on the data with 12 million tuples and results are shown in figure 6.

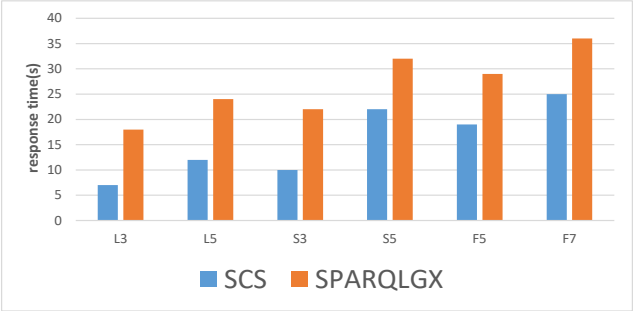

Figure 6. Query Time over LUBM100

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 our system and SPARQLGX is shown in Figure 6, we can see that our system outperforms the comparison system in query response time regardless of the query type or the number of triples contained. This can be attributed to the following two reasons: i)finer-granined partitioning schema. Our system matches a Tp by inputting a smaller index file than sparqlgx; ii)optimal query plan. Compared with sparqlgx, the query plan made by our system and the method of broadcasting variables can avoid a lot of communication costs.

For WatDiv10M, our experiments run queries on the data with 10 million tuples and results are shown in figure 7.

As shown in Figure 7, we can find that our system in all the types of standard queries is better efficiency than SPARQLGX. 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 Figure 8, in which Watdiv divides more index files than LUBM. Therefore, when doing an evaluation query under a data set of

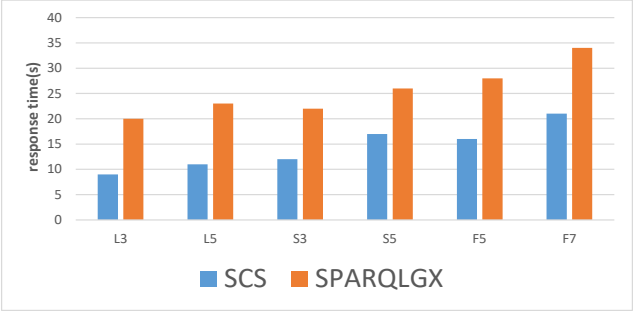

Figure 7. Query Time over Watdiv10M

the same size, the time required to read the index files divided by the LUBM dataset is likely to be greater than the index files divided by the Watdiv dataset.

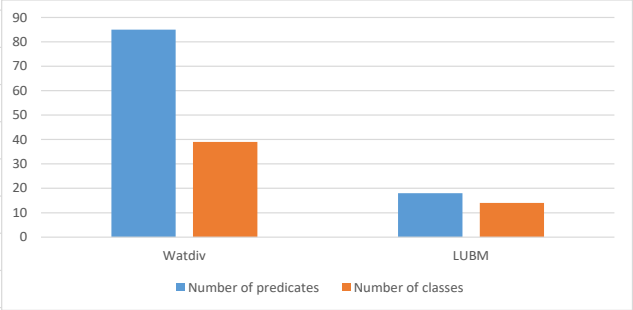

Figure 8. Number of Predicates category

8. Conclusion

In this paper, we introduce the SCS, 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 semantic connection set 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.

Author 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.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

RDF	Resource Description Framework
SCS	Semantic Connection Set
SPARQL	SPARQL Protocol and RDF Query Language
VP	Vertical Partitioning
TPs	Triple Patterns
BGP	Basic Graph Pattern
IRs	Intermediate Results
L	Linear
S	Star
F	Snowflake

References

1. Hayes, P. RDF Semantics. *Recommendation* **2004**.
2. Prud, E.; Seaborne, A.; others. Sparql query language for rdf **2006**.
3. Neumann.; Thomas.; Weikum.; Gerhard. The RDF-3X engine for scalable management of RDF data. *Vldb Journal* **2010**, *19*, 91–113.
4. Weiss, C.; Karras, P.; Bernstein, A. Hexastore: sextuple indexing for semantic web data management. *Proceedings of the Vldb Endowment* **2008**, *1*, 1008–1019.
5. Abadi, D.J.; Marcus, A.; Madden, S.R.; Hollenbach, K. SW-Store: a vertically partitioned DBMS for Semantic Web data management. *Vldb Journal* **2009**, *18*, 385–406.
6. Agathangelos, G.; Troullinou, G.; Kondylakis, H.; Stefanidis, K.; Plexousakis, D. RDF Query Answering Using Apache Spark: Review and Assessment. 2018 IEEE 34th International Conference on Data Engineering Workshops (ICDEW), 2018.
7. Abadi, D.J.; Marcus, A.; Madden, S.R.; Hollenbach, K. Scalable semantic web data management using vertical partitioning. Proceedings of the 33rd international conference on Very large data bases. VLDB Endowment, 2007, pp. 411–422.
8. Guo, Y.; Pan, Z.; Heflin, J. LUBM: A benchmark for OWL knowledge base systems. *Social Science Electronic Publishing* **2005**, *3*, 158–182.
9. Aluç, G.; Hartig, O.; Özsu, M.T.; Daudjee, K. Diversified Stress Testing of RDF Data Management Systems. 2014.
10. Husain, M.; McGlothlin, J.; Masud, M.M.; Khan, L.; Thuraisingham, B.M. Heuristics-based query processing for large RDF graphs using cloud computing. *IEEE Transactions on Knowledge and Data Engineering* **2011**, *23*, 1312–1327.
11. Papailiou, N.; Konstantinou, I.; Tsoumakos, D.; Karras, P.; Koziris, N. H 2 RDF+: High-performance distributed joins over large-scale RDF graphs. IEEE International Conference on Big Data, 2013.
12. Graux, D.; Jachiet, L.; Geneves, P.; Layaïda, N. Sparqlgx: Efficient distributed evaluation of sparql with apache spark. International Semantic Web Conference. Springer, 2016, pp. 80–87.
13. Schätzle, A.; Przyjaciel-Zablocki, M.; Skilevic, S.; Lausen, G. S2RDF: RDF querying with SPARQL on spark. *Proceedings of the VLDB Endowment* **2016**, *9*, 804–815.
14. Kaoudi, Z.; Manolescu, I. RDF in the clouds: a survey. *The VLDB Journal—The International Journal on Very Large Data Bases* **2015**, *24*, 67–91.
15. Schätzle, A.; Przyjaciel-Zablocki, M.; Hornung, T.D.; Lausen, G. PigSPARQL: a SPARQL query processing baseline for big data. Th International Conference on Posters & Demonstrations Track, 2013.
16. Shvachko, K.; Radia, S.; Cox, A.L. The hadoop distributed file system. IEEE Symposium on Mass Storage Systems & Technologies, 2010.
17. Zaharia, M.; Chowdhury, M.; Franklin, M.J.; Shenker, S.; Stoica, I. Spark: cluster computing with working sets. Usenix Conference on Hot Topics in Cloud Computing, 2010.
18. Rohloff, K.; Schantz, R.E. High-performance, massively scalable distributed systems using the MapReduce software framework: the SHARD triple-store. Programming Support Innovations for Emerging Distributed Applications, 2010.

19. Zhang, X.; Chen, L.; Wang, M. Towards efficient join processing over large RDF graph using mapreduce. International Conference on Scientific & Statistical Database Management, 2012.
20. Zhang, X.; Lei, C.; Tong, Y.; Min, W. EAGRE: Towards scalable I/O efficient SPARQL query evaluation on the cloud. IEEE International Conference on Data Engineering, 2013.
21. Mcbride, B. Jena: a semantic Web toolkit. *IEEE Internet Computing* **2002**, *6*, 55–59.
22. Weston, T. The Scala Language **2018**.
23. Abdelaziz, I.; Harbi, R.; Khayyat, Z.; Kalnis, P. A survey and experimental comparison of distributed SPARQL engines for very large RDF data. *Proceedings of the Vldb Endowment* **2017**, *10*, 2049–2060.