Minimizing Query Execution Time of SPARQL Queries with CS-index Approach

Khin Myat Kyu[1], Kay Thi Yar[2], Aung Nway Oo[3]

University of Information Technology
Yangon, Myanmar
[1]khinmyatkyu@uit.edu.mm
[2]kaythiyar@ uit.edu.mm
[3]aungnwayoo@ uit.edu.mm

Abstract - **RDF is a W3C's standardized data model for Semantic Web, and provides a graph-based descriptive way for representing resources on the web and their relationships. With the increasing use of RDF data, SPARQL query processing over the data becomes a critical issue. This paper proposes an indexing and searching approach that can support both chain and star shaped SPARQL query. Our approach considers graph structural nature of the RDF data. The RDF data is firstly decomposed into chain and star shaped subgraph patterns based on nature of edges for each vertex. These subgraphs are stored as index, called CS-index. When a SPARQL query is given, it is decomposed into query subgraph patterns based on common join variable among all triple patterns. And the query results are finally obtained by matching these query subgraphs against with CS-index. The proposed approach tends to minimize data loading/indexing time, and query execution time by reducing number of join operations needed to perform for a query's processing.**

Keywords - **Semantic Web, RDF, SPARQL, graph-based index, join optimization**

I. Introduction

Resource Description Framework (RDF) is a schema-free and graph-structured data model for describing resources on the Web. Resources may be person, places, organizations, or anything on the Web. These RDF data can be accessed by SPARQL is a declarative query language recommended by W3C. As the highly interconnected nature of Web data, many RDF data management systems have been proposed with different techniques [1], i.e., relation-based RDF store, the clustered property table, vertical partitioning, and indexing.

Relation-based RDF stores such as Jena-SDB, Sesame, RDF-3X, manage the data in relational tables and process SPARQL queries using relational operators, such as scan and join operators. The main problem of relation-based RDF stores is that they need too many join operations for processing SPARQL queries when the queries contain many triple patterns. This kind of queries is called complex queries.

To process complex SPARQL queries, many approaches have been considered to solve this issue by emphasizing on: (i) reducing number of join, (ii) reducing inputs of join operators, and (iii) optimizing join order [3]. However, the graph-structured nature of the RDF and the graph pattern matching nature of SPARQL queries still have significant challenges for efficient processing of complex SPARQL queries over the interlinking RDF data. A question arises to ask how to find efficiently all matches of a query graph in a large database graph, i.e., reducing time of query processing as much as possible.

In this paper, an indexing structure and querying algorithm is proposed for processing chain and star shaped SPARQL query. Blank nodes are not considered in this paper as they represent a resource without specifying its URI. Our proposed indexing structure collects vertices/literals, and predicates for each vertex in the RDF data graph based on their incoming and outgoing edges. These combinations (vertices/literals and predicates) are stored into an index table as key-value form to quickly access the data. And we also propose a search algorithm based on our indexing structure. The proposed method could minimize query execution time, and requires little memory usage as it stores all structural information of one vertex with one key-value pair.

The remainder of the paper is organized as follows: Section 2 describes literature review and explanation of RDF data and SPARQL query is given in Section 3. Our proposed indexing structure and search algorithm are presented in Section 4. Section 5 explains the query evaluation with proposed method. Finally, Section 6 concludes the whole paper and discusses future perspectives.

II. Related Work

Many triple stores, such as RDF-3X [4] and Hexastore [5], store all RDF triples data (S,P,O) in a single three-column table. For efficient data access, one-dimensional indexes (B+ trees) are used for each of the six permutations (i.e., SPO, SOP, PSO, POS, OSP, OPS), known as sextuple indexing technique. However, this querying efficiency comes at the cost of excessive storage requirements and maintenance overhead since the complete data set is stored replicated six times. It degrades the efficiency of query processing as it requires expensive self-joins when SPARQL queries consist multiple triple patterns [1].

X. Wang et. al [6] proposed a RDF storage and indexing scheme, called CHex. CHex uses sextuple indexing and binary association table (BAT) for a column-oriented database system. It not only provides efficient single triple pattern lookups, but also allows fast merge-joins for any pair of two triple patterns. But the additional processing becomes substantial as the queries become complex. And it incurs space overhead in data storing.

X. Lyu et. al [7] proposed the efficient subgraph matching method for star queries. The method decomposes both data graphs and query graphs into sets of star graphs, and encode each star subgraph into a fingerprint. Fingerprints were used to effectively reduce the data searching space. But the method takes too much time in fingerprints encoding, and can handle only star shaped queries.

In [8], a graph indexing approach, Extended Characteristic Sets, is proposed for SPARQL query optimization. Extended Characteristic Sets the authors considered is based on the work in [9]. It aims to accelerate query processing time for conjunctive queries with multi-chain-star patterns, called double chain-star queries. The approach had advantages on queries' processing time. But it can process only double chain-star queries and has processing overhead as it can extract extended characteristic sets after generating characteristic sets. And it cannot support data updates.

In [10], RP-filter was proposed for reducing the redundant intermediate results of join operations. RP-filter uses a path-based index which indexes the incoming path information of RDF graph. However, it has limitation that it could not exploit the graph structural information of RDF data. The additional processing becomes substantial as the queries become complex. In order to overcome this limitation, RG-index was proposed in [11]. The RG-index indexes the graph patterns by using adapted gSpan algorithm - is a frequent subgraph mining algorithm was originally proposed for graph transaction data set eg. chemical compounds. But the method takes too much time for mining discriminative and frequent graph patterns from RDF data.

To overcome these limitations, we propose CS-index which extracts chain and star shaped graph patterns by counting the incoming and outgoing degrees of vertices while parsing and dictionary encoding. We assume that extraction of chain and star shaped patterns need the time than RG-index because our proposed method does not use subgraph mining technique. And the proposed method could support both chain and star shaped SPARQL queries.

III. Preliminary concepts

In this section, formal definition of RDF data and SPARQL query are provided. Assume that there are three pairwise disjoint sets: a set of uniform resource identifiers (URIs) U, a set of literals L, and a set of variables VAR.

A. *RDF Data*

A RDF data set is a collection of statements in the form of subject (s), predicate (p), and object (o). A statement t ϵ U × U × (U ʋ L) (without variables) is called a triple. Table 1 presents an example of RDF data set, LUBM - is a standard data set which was developed to evaluate the performance of Semantic Web repositories. For simplicity, we use prefix for each URI as many triples share the same URI.

Prefix:

uni0 = "http://www.University0.edu#"

uni241= "http://www.University241.edu#

dept0= "http://www.Department0.University0.edu#"

rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"

owl= "http://swat.cse.lehigh.edu/onto/univ-bench.owl#"

TABLE I
Example Rdf Data Set

Subject	Predicate	Object
uni0:University0	rdf:type	owl:University
uni0:University0	owl:name	"University0"
dep0:Department0	rdf:type	owl:Department
dept0:Department0	owl:name	"Department0"
dept0:Department0	owl:subOrganizationOf	uni0:University0
uni0:FullProfessor0	rdf:type	owl:FullProfessor
uni0:FullProfessor0	owl:name	"FullProfessor0"
uni0:FullProfessor0	owl:teacherOf	uni0:GraduateCourse0
uni0:FullProfessor0	owl:teacherOf	uni0:GraduateCourse1
uni0:FullProfessor0	owl:doctoralDegreeFrom	uni241:University241
uni0:FullProfessor0	owl:worksFor	dept0:Department0
uni0:FullProfessor0	owl:researchInterest	"Research20"
uni0:GraduateStudent1	owl:advisor	uni0:FullProfessor0

These RDF data can be considered as directed, labelled graph. Figure 1 shows the RDF data graph for example RDF data set in Table. 1. In this paper, blank nodes are omitted as they represent a resource without specifying its URI.

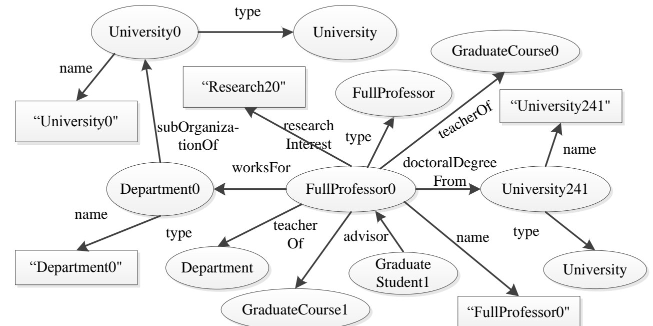

Fig. 1 RDF data graph

B. *SPARQL Query*

A SPARQL query consists of one or more triple patterns (tps). A statement tp ϵ (U ʋ VAR) × U × (U ʋ L ʋ VAR) (triple with variables) is called a triple pattern. Variable symbols start with ''?'' to distinguish them from URIs and literals. SPARQL queries can be classified based on the shape of the query graph. In this paper, chain and star query are considered. Chain queries include subject-object join (the join is between a tp's subject and another tp's object). A star query includes subject-subject join, i.e. join variable is at the subject's position of all the tps.

Figure 2. shows example of SPARQL query. It retrieves the university's name where FullProfessor0 got doctoral degree. It consists in the chain query type.

```
SELECT ?X WHERE {
uni:FullProfessor0
owl:doctoralDegreeFrom ?X.
        ?X  rdf:type "University" .
}
```

Fig. 2 Example SPARQL query

The query graph of the example SPARQL query in Figure 2 is shown in Figure 3.

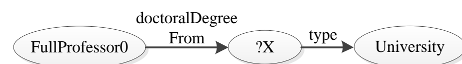

Fig. 3 Example SPARQL query graph

IV. Proposed Method

In our proposed method, there are two main phases: (i) index construction and (ii) query searching. The proposed indexing and searching algorithm are designed to process

both chain and star query. Algorithms for the proposed method are described in Fig. 5 and Fig. 6, respectively.

A. Index Construction

There are some tasks in CS-index construction phase: parsing RDF triples, constructing dictionaries, and extracting graph patterns based on the incoming and outgoing degree of each vertex. All these tasks are carried out in parallel. As first task, each RDF triple (v_i,e_i,v_j) is parsed into three parts: subject, predicate, and object. Values of the subject/predicate/object are URIs or literals. Thus, we store integer values instead of these URIs and literals because they are complex and long string values.

Two dictionaries are needed for mapping the RDF triples with integer values. The first one is for subjects and objects, and the other one is for predicate, called subject/object dictionary and predicate dictionary, respectively. Key-value (id, value) mappings are used to construct the dictionaries. The notation for 'id' is defined as V_{id} and E_{id} where V_{id} is the integer value for subjects and objects, E_{id} is the integer value for predicates.

V_{id}	URI/Literal
1	uni0:University0
2	owl:University
3	University0
4	dept0:Department0
5	owl:Department
6	Department0
7	uni0:FullProfessor0
8	owl:FullProfessor
9	FullProfessor0
10	uni0:GraduateCourse0
11	uni0:GraduateCourse1
12	uni241:University241
13	Research20
14	uni0:GraduateStudent1

(a)

E_{id}	URI
1	rdf:type
2	owl:name
3	owl:subOrganizationOf
4	owl:teacherOf
5	owl:doctoralDegreeFrom
6	owl:worksFor
7	owl:researchInterest
8	owl:advisor

(b)

Fig. 4 (a) Subject/Object dictionary, (b) Predicate dictionary

While constructing the dictionaries, in-degree and out-degree are computed for each subject and object. Degree computation is not need to consider for predicate. If the parsed one (v_i) is subject, we increase the out-degree and add the pair (e_i,v_j) into outgoing-edges of vi. If it is object, we increase the in-degree and add the pair (v_j,e_i) into incoming-edges of v_i. If it is predicate, next triple is read to parse.

Algorithm 1: CS-index construction algorithm

```
1.  Input: RDF data set D
2.  Output: CS-index, subject/object dictionary,
          predicate dictionary
3.  begin
4.  for each triple t in D
5.    parse and encode each URI/literal
6.    for each encoded URI/literal v_i
7.      if encoded URI/literal v_i is subject
8.        if out-degree of v_i is zero
9.        out-degree of v_i ++
10.       outgoing-edges of v_i = {(e_i, v_j)}
11.       end if
12.       else
13.         out-degree of v_i ++
14.         merge (e_i,v_j) to existing outgoing-edges of v_i
15.       end else
16.     end if
17.     else if encoded URI/literal v_i is object
18.       if in-degree of v_i is zero
19.         in-degree of v_i ++
20.         incoming-edges of v_i = {(v_j, e_i)}
21.       end if
22.       else
23.         in-degree of v_i ++
24.         merge (v_j, e_i) to existing incoming-edges of v_i
25.       end else
26.     end else if
27.     else break;
28.     end for
29.   end for
30.   for each encoded subject/object vertex v_i
31.     store outgoing-edges and incoming-edges,
      and v_i
32.   end for
33.   end
```

Fig. 5 Algorithm for CS-index construction

After all RDF triples have been processed completely, we store incoming-edges and outgoing-edges as compound key and v_i as value in CS-index. And then, CS-index is sorted in ascending order based on the in-degree and out-degree pair of each v_i. CS-index of example RDF data in Table I is shown in Table II.

TABLE II
CS-INDEX ARCHITECTURE

	outgoing-edges (e_i,v_j)	incoming-edges (v_j, e_i)	V_{id}
#1	-	{(4,1)}	5
#2	-	{(4,2)}	6
#3	-	{(7,1)}	8
#4	-	{(7,2)}	9
#5	-	{(7,4)}	10
#6	-	{(7,4)}	11
#7	-	{(7,5)}	12
#8	-	{(7,7)}	13
#9	-	{(1,1)}	2
#10	{(1,2)}	-	3
#11	{(8,7)}	-	14
#12	{(1,2),(2,3)}	{(4,3)}	1
#13	{(1,5),(2,6),(3,1)}	{(7,6)}	4
#14	{(1,8),(2,9),(4,11),(4,11),(5,12),(6,4),(7,13)}	{(14,8)}	7

B. Query Processing

When a query arrives, the query processor finds common join variable which include as a variable in more than one triple pattern. And the triple patterns are grouped based on the common join variable. And each subject/predicate/object values are encoded using two dictionaries constructed in index construction stage. Then, in-degree, out-degree, incoming-edges, and outgoing edges are computed for each common join variable.

Algorithm 2: Query processing algorithm

```
1. Input: SPARQL query Q, CS-index
2. Output: result of the query result_Q
3. begin
4.   find common join variable var_i
5.   decompose triple patterns tps based on var_i
6.   compute in-degree, out-degree, incoming-edges,
        outgoing-edges of var_i
7.    result_Q = match(in-degree_vari, out-degree_vari,
```

incoming-edges$_{vari}$, outgoing-edges$_{vari}$)
8. decode result$_Q$
9. return result$_Q$
10. end

match(in-degree$_{vari}$, out-degree$_{vari}$, incoming-edges$_{vari}$, outgoing-edges$_{vari}$)

1. begin
2. access the CS-index based on the in-degree and out-degree of var$_i$
3. retrieve the values which match with incoming-edges, outgoing-edges
4. return result$_{vari}$
5. end

Fig. 6 Algorithm for query processing

When these four values are obtained, the results are searched in CS-index. The location of CS-index where the result can be exist are easily accessed as the CS-index is sorted in ascending order according to the degree of vertices. After the matched value (vertex id) is obtained, all vertex id need to be decoded into the original strings by the dictionary lookups. And the system displays the result to user. In this way, the proposed method could optimize the query response time by reducing number of join operations.

V. Performance Study

A. Experimental Setup

We have conducted an experimental evaluation with synthetic data set, LUBM [10], was used for performance testing and all tests were run 10 times to calculate the average results. The algorithm was implemented in Java SDK 1.8 and all tests were performed on a PC with an Intel Core i3 1.90 GHz processor, 4 GB RAM, and operating system is Windows 8.1. The data nature of three different LUBM data set is described in Table III.

TABLE III
DATA CHARACTERISTICS OF THREE LUBM DATA SET

Data Set	#triples	#class instance	#property instance
LUBM10	1,316,511	263,427	1,052,895
LUBM15	2,021,508	404,743	1,616,472
LUBM20	2,781,724	556,572	2,224,750

Test queries (Q1-Q5) were designed for SPARQL query processing. Table IV lists all the benchmark queries. Q1 consists of only one triple pattern. Q2 is chain shaped query. Q3 and Q4 are star shaped queries with two triple patterns and seven triple patterns, respectively. The last Q5 is a query which contains both subject-object join and object-object join.

TABLE IV
SPARQL TEST QUERIES

ID	*Query*

Q_1	SELECT ?X WHERE {
	?X rdf:type owl:University .
	}

Q_2	SELECT ?X WHERE { uni:AssociateProfessor0 owl:worksFor ?X . ?X rdf:type owl:Department . }
Q_3	SELECT ?X WHERE { ?X rdf:type owl:GraduateStudent . ?X owl:takesCourse uni0:GraduateCourse20> .}

Q_4	SELECT ?X WHERE {
	?X rdf:type owl:AssociateProfessor .
	?X owl:researchInterest "Research2" .
	?X owl:teacherOf uni0:Course23 .
	?X owl:teacherOf uni0:GraduateCourse23 .
	?X owl:teacherOf owl:GraduateCourse24 .
	?X owl:doctoralDegreeFrom uni290:University290 .
	?X owl:worksFor uni0:University0 .
	}

Q_5	SELECT ?X WHERE {
	?X rdf:type owl:Course .
	uni0:FullProfessor2 owl:teacherOf ?X .
	uni0:UndergraduateStudent118 owl:takesCourse ?X .
	}

TABLE V
DATA LOADING/INDEXING TIME FOR THREE LUBM DATA SET

Data set	Data Loading/Indexing Time (sec)
LUBM10	11.72
LUBM15	20.34
LUBM20	30.24

Table V describes the time for data loading/indexing of three different LUBM data set. It takes a few seconds to load and index the input RDF data. We found that the proposed method had slightly time difference although the number of triples contained in the data set is significantly varied.

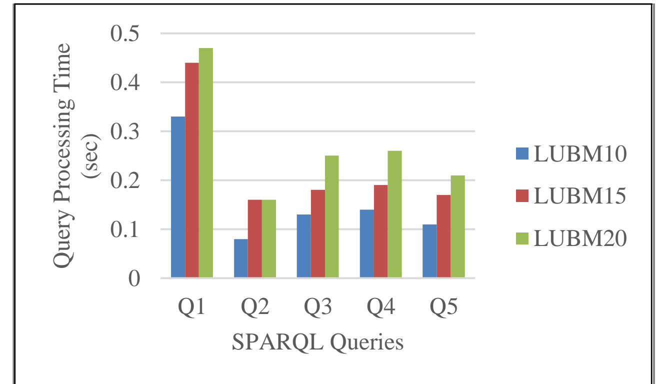

Fig. 7 Query processing time of Q1, Q2, Q3, Q4, and Q5 over three LBUM data set

Fig. 7 shows the time performance of five test queries in three different data set. According to our experiment, our proposed method can efficiently enhance the query run time even if the number of triple patterns contained in a given SPARQL query is large.

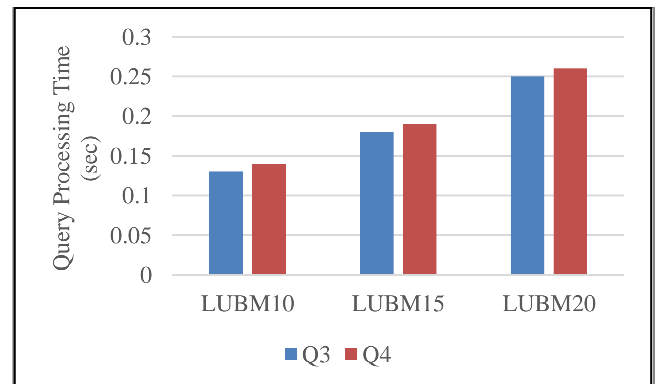

Fig. 8 Query processing time of Q3 and Q4 over three LUBM data set

To obviously evaluate, we again compared only Q3 and Q4. They are both star shaped query. But Q3 has two triple patterns and Q4 has seven triple patterns. The difference between number of triple patterns in Q3 and Q4 is three times, but the processing time was not different too much. The comparison result is shown in Fig. 8.

When we made an evaluation, it showed that our proposed method can handle the queries with many triple patterns as in processing the queries with less triple patterns. The query processing time do not differ too much. So, we conclude that our proposed method could process both chain and star shaped SPARQL queries. Even when the star shaped queries have many triple patterns, it can process well.

VI. CONCLUSIONS

The proposed approach is designed to gain high-performance query processing for chain and star shaped SPARQL queries. Formally, when a query with n triple patterns is processed, (n-1) join operations are needed to execute to get the query's result. It takes too much time for query processing. So, our proposed CS-index and querying approach intend to minimize query processing time by avoiding join operations. The proposed method has index construction time, but it requires only one unit cost to get the result as explained in the query evaluation in Section V. And it uses reasonable memory space as two dictionaries (subject/object, predicate) and CS-index are needed to store instead of original data set.

In future work, we will compare to validate that our proposed method could efficiently minimize query execution time than other state-of-art RDF indexing and querying approach.

REFERENCES

[1] M.T. Ozsu, "A survey of RDF data management systems", Frontiers of Computer Science, Vol. 10, No. 3, June 2016, pp. 418-432.

[2] T. Neumann and G. Weikum, "RDF-3X: a RISC-style engine for RDF," In Proc. VLDB, pp. 647–659, 2008.

[3] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: sextuple indexing for semantic web data management," In Proc. VLDB, pp. 1008–1019, 2008

[4] X. Wang, S. Wang, P. Du, and Z. Feng, "CHex: An Efficient RDF Storage and Indexing Scheme for Column-Oriented Databases", International Journal of Modern Education and Computer Science, Vol. 3, No. 3, June 2011, p. 55.

[5] X. Lyu, X. Wang, Y.F. Li, Z. Feng, and J. Wang, "GraSS: An efficient method for RDF subgraph matching", In International Conference on Web Information Systems Engineering, 1 Nov 2011, pp. (108-122), Springer, Cham.

[6] Marios. et. al, "Extended Characteristic Sets: Graph Indexing for SPARQL Query Optimization", In Data Engineering (ICDE), 2017 IEEE 33rd International Conference on (pp. 497-508). IEEE.

[7] Neumann, T. and Moerkotte, G., 2011, April. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In Data Engineering (ICDE), 2011 IEEE 27th International Conference on (pp. 984-994). IEEE.

[8] K. Kim, B. Moon, and H.J.Kim, "RP-Filter: A path-based triple filtering method for efficient SPARQL query processing", In Joint International Semantic Technology Conference, 4 Dec 2011, pp. 33-47

[9] K. Kim, B. Moon, and H. J. Kim, "RG-index: An RDF graph index for efficient SPARQL query processing", Expert Systems with Applications, Vol. 41, August 2014, pp. 4596 – 4607.

[10] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A benchmark for OWL knowledge base systems", Web Semantics: Science, Services and Agents on the World Wide Web, Vol. 3, 31 Oct 2005, pp. 158-182.

[11] K. M. Kyu, K. T. Yar, A. N. Oo, "A Proposal of CS-index Approach for SPARQL Queries Considering Chain and Star Shaped Subgraphs", The 10th International Conference on Advances in Information Technology (IAIT2018), Vol. 10.