Hindawi
Journal of Sensors
Volume 2022, Article ID 4624856, 12 pages
https://doi.org/10.1155/2022/4624856

Research Article

A Two-Phase Method for Optimization of the SPARQL Query

Xiaoqing Lin[1] **and Dongyang Jiang**[2]

[1]*School of Information Engineering, Liaodong University, Dandong 118003, China*
[2]*Department of Information Engineering, Liaoning Mechatronics College, Dandong 118009, China*

Correspondence should be addressed to Xiaoqing Lin; linxiaoqing@elnu.edu.cn

Received 20 June 2022; Accepted 5 August 2022; Published 25 August 2022

Academic Editor: Yuan Li

With a rapid growth in the available resource description framework (RDF) data from disparate domains, the SPARQL query processing with graph structures has become increasingly important. In this pursuit, we designed a two-phase SPARQL query optimization method to process the SPARQL query. The structural characteristics of RDF data graphs, predicate path sequence indices (PPS-indices), were used to efficiently prune the search space, which captured the inherent features of the RDF data graphs, while the database is updated. Our storage model was based on a relational database. Compared to a baseline solution, the proposed method effectively reduced the cardinalities of the intermediate results during the query processing, and at least an order of magnitude improvement is achieved in filtering performance, thereby improving the efficiency of the query execution.

1. Introduction

After the advent of RDF as a standard model of data representation and exchange in the Semantic Web, the availability of the RDF data has immensely increased. RDF data is a collection of statements, in the form of a triple of subject, predicate, and object, where predicate connects the subject and the object. SPARQL is a standard RDF data query language. SPARQL is widely used due to the useful web information that can be obtained from RDF data. However, large amounts of redundant intermediate results appear during executing SPARQL queries, especially in the processing of the large-scale RDF data, which greatly reduces efficiency of SPARQL query. Therefore, optimization of the SPARQL query has attracted a significant attention. With the rapid development of Semantic Web technology, ontologies have been widely applied in semantic data mining. Semantic data mining is the searching process of information hidden in a large amount of data by incorporating the domain knowledge. Ontology is actually a formal representation of a set of concepts in a specific field and the relationships between them. The present studies exploited the domain knowledge to filter out the useless intermediate results. In our two-phase optimization method, in the first stage, an input SPARQL query is divided into different blocks, and the triple pattern containing the same variable is classified into one block where each block has only one variable. The triple patterns in each block are sorted by selectivity to obtain the optimal query execution plan, and then, the triple with the largest selectivity is chosen to execute the query. In the second stage, for the remaining triple pattern containing two variables, the redundant vertices are filtered according to prebuilt PPS-indices, and finally query results are generated. Along with the structural characteristics of RDF data graph, PPS-indices are employed, and the storage based on relational databases is combined together to optimize the SPARQL query.

2. Overview and Related Work

Our storage model consists of a relational database suitable for handling large-scale RDF data. Earlier RDF search engines used relational database management systems to store RDF triples, such as Jena [1] and Sesame [2] which store RDF triples in the form of triple tables, and used the query processing module of the relational database management system to process RDF data. However, these systems exhibit poor scalability for large-scale data. SW-store [3], RDF-3x [4], x-RDF-3X [5], Hexastore [6], and gStore [7] can effectively handle large-scale Semantic Web data; SW-store uses vertical partition storage, wherein splitting the triple table by the predicate can quickly merge and join. At present, most SPARQL optimization techniques mainly reduce the intermediate results produced in the join process by formulating an optimal query execution plan. The connection order of the query affects the amount of the intermediate results. In earlier studies [4, 5], dynamic programming was used to estimate the optimal query execution plan. This type of method has an exponential amount of time and space overhead and does not use the RDF data graph structure. In another study [6], six permutation and combination indices of triples (subject, predicate, and object) were built, which could perform effective connection operations, but the index storage overhead was relatively large. The system used MapReduce and HBase, and an abstract RDF data was exploited to decrease the amount of intermediate data [8].

Graphs have become increasingly important in SPARQL query processing. In this pursuit, the VS tree and VS∗ tree indices were developed, which used the subgraph matching to process the wildcard or precise SPARQL query processing [7]. A directed supersemantic query graph was used to extract the structural query intention of the question, based on which Q/A on RDF was reduced to the graph matching problem [9]. Nowadays, Semantic Web ontologies have become an indispensable technology for intelligent processing knowledge and provide a framework for conceptual models of shared domains. Domain knowledge can be used as a set of prior knowledge for constraints to help guide the search path and reduce search space, during the search and pattern generating process [10–16]. A set of preconstructed semantics-aware indexes were constructed [17]. In order to capture the ontology information, as well as the data information, an RDF hypergraph representation was developed [18], which is a little similar to [19]. Further, the ontologies were used to prune space and filter the association rule mining results [16, 20, 21]. Ontologies were later exploited to help the subjective analysis for the association rule postpruning task [22]. In the study [23], the redundant intermediate results were filtered out through the RDF graph predicate path sequence indices, but we do not deal with the triple pattern containing text and empty vertices. Index information method implements a query optimization, dynamically calculating the query cost for large-scale RDF data, then formulating the optimal query plan by the connection order cost. However, large amounts of storage space overhead are required [24, 25]. An inductive learning algorithm was later developed that could automatically select the useful join paths and properties to construct rules from an ontology with many concepts [26], thereby leading to an effective optimization of the SPARQL query.

Motivated by these observations, we exploited the feature of RDF graph and advantages of relational database to refine the RDF triples, so as to improve the efficiency where on the one hand we implemented the join operation based on relation model storage, and on the other hand, we constructed RDF graph indices (PPS-indices) using graph exploration. The traditional indexing approach may encounter challenges such as graphs. This is because a complex RDF graph may contain an exponential number of paths. Though six permutation indices of triples could speed up the query processing, the storage overhead was increased [6]. In order to help processing graph queries, our method constructed the PPS-indices that only kept all discriminative segments which were refined from the set of frequent structures, thus scaling down the number of frequent paths to be indexed. By the PPS-indices, a lot of useless intermediate results could be greatly reduced. We calculated all indices in advance, and our method was very effective for the query processing of long path patterns in RDF data. TripleBit dynamically computed optimal execution orderings for join queries. However, it was not very effective in processing these long path patterns [27]. In the study [4], only given triple patterns could be processed. Any graph structure information in RDF data was not considered in [4, 5, 27]. Our method used the RDF graph structure characteristics, and our experimental results have shown that our proposed method was more effective for RDF data than theirs.

2.1. SPARQL and Query Execution Plan. SPARQL has been recommended by the W3C as the standard query language for RDF data. It is a query for obtaining the results through matching triples on the RDF data [28]. A basic SPARQL query mainly consists of two parts, the SELECT clause and the WHERE clause. Among them, "pattern1", "pattern2", etc. are all triple patterns, and the operator "." connects the two triple patterns. Each triple pattern is composed of a subject, predicate, and object, and these three items are either text or variables. The query uses the text to represent the known and employs variables to denote the unknown, as shown in the query example:

SELECT ?variable1 ?variable2 … WHERE { pattern1. pattern2…}

The following is the ninth query example Q9 from the LUBM benchmark. This example queries all students, teachers, and courses and satisfies that the courses of students are taught by the teacher and the teacher is the students' instructor. Among them, the first two lines "rdf" and "ub" are all prefixes, and "?X rdf : type ub : Student" is a triple pattern. The SPARQL queries processed in this study include triple patterns with the predicate as a variable (see Section 2.4).

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub:<http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X ?Y ?Z
WHERE{?X rdf:type ub:Student.
?Y rdf:type ub:Faculty.
?Z rdf:type ub:Course.
?X ub:advisor ?Y.
?Y ub:teacherOf ?Z.
?X ub:takesCourse ?Z }
```

A query execution plan for the above example is shown in Figure 1, which is an operation tree. A SPARQL query includes multiple joins, so an improper query join order

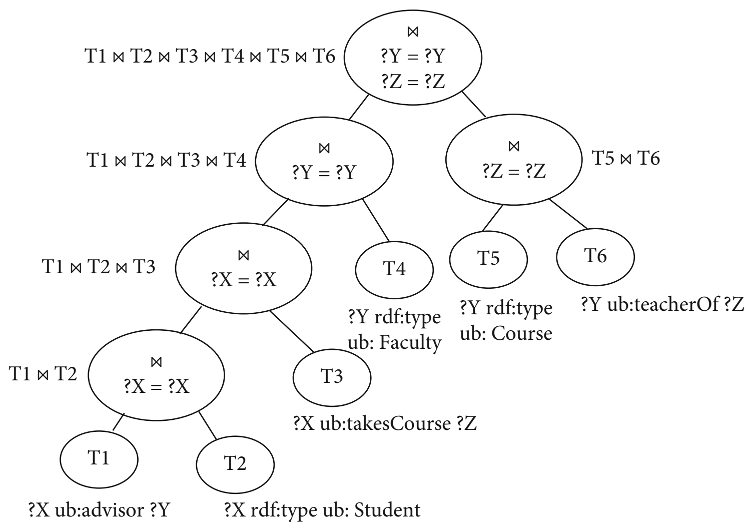


Figure 1: A query plan for Q9.

can produce large amounts of redundant intermediate results. As shown in Table 1, the third and fourth rows present the amount of intermediate results during the join process. The fourth row "T4⋈T5⋈T6" join produces 600,000 intermediate results, but only 2000 of the final results, at least 598,000 redundant intermediate results, are generated in this process. As a result, the size of the result set of each connection affects the next join operation. Therefore, in order to reduce the intermediate results as much as possible, in the first stage, it is important to adjust the join sequence and put the join producing fewer intermediate results in the front. In the second stage, the redundant vertices of the remaining triple patterns are filtered out using predicate path sequence indices (see Section 2.4 for details).

2.2. SPARQL Query Optimization. We employed the SPARQL queries as graphs instead of trees. The system process flow is shown in Figure 2. In the first stage, the SPARQL query graph is divided into blocks, reordered according to the cost of each join within the block, and partial joins are executed. The cost is estimated based on the amount of intermediate results produced by the joins in the second stage, wherein we used the PPS-indices that were built beforehand to refine the triples. This process is somewhat similar to the filtering of the predicate path [23]. While filtering, the longest PPS-indices were employed for matching the filtering instead of all predicate paths within a path length range. This avoided excessively double calculation, so as to improve the query efficiency (see Section 2.4).

The query execution plan is a graph instead of a tree, which is convenient for reusing the results of the intrablock joins. Figure 3(a) shows the Q9 query graph. In the first stage, the SPARQL query graph is divided into different blocks containing the same join variables. Figure 3(b) is a part of the Q9 query graph. According to the triple pattern containing the same join variable, "$?X$" are classified into one block. Of course, a triple pattern with two variables, such as "$?X$ ub : advisor$?Y$", can be classified into one block by the variable "$?X$", or it can be classified into one block by the variable "$?Y$". Each block belongs to a star join. A star denotes a join between two subjects. For instance, in the above example, "T1⋈T2" and "T1⋈T3" all belong to the star join, which is the ternary description of the predicate in the star join. The group mode (including a variable) is reordered according to selectivity, and partial joins are performed to filter out the useless intermediate results.

Table 1: The cardinality of intermediate results.

Different joins	The cardinality of intermediate results
T1⋈T2⋈T3⋈T4⋈T5⋈T6	2000
T1⋈T2⋈T3	1,000,000
T4⋈T5⋈T6	600,000

2.3. Estimation of Selectivity. The selectivity is based on the number of triples of the matching join, Tn, as shown in formula (1), where the degree of selectivity is inversely proportional to the number of matched triples. That is, the number of corresponding matched triples is larger, and the degree of selection is smaller. For example, if $\#(T1) < \#(T2) < \#(T3)$, then $\text{sel}(T1) > \text{sel}(T2) > \text{sel}(T3)$. In order to save the space and time, the calculation of SELECTIVITY (selection degree) generally contains at most one query variable triple pattern, and the most frequent triple pattern is $(?s, p, o)$. Given the object o and the predicate p of a triple, the value of the subject s in the corresponding triple can be obtained.

$$\text{SELECTIVITY} = \text{minimum}(\text{sel}(T1), \text{sel}(T2), \cdots, \text{sel}(Tn)). \tag{1}$$

We built an OPS (object, predicate, and subject) index, as presented in [6]. By specifying the object and predicate, the corresponding subject Id list and cardinals of matching triples were obtained, so as to complete the optimal execution within the block. The plan and structure are shown in Table 2. The joins with a small number of connections were performed, so as to minimize the unnecessary joins.

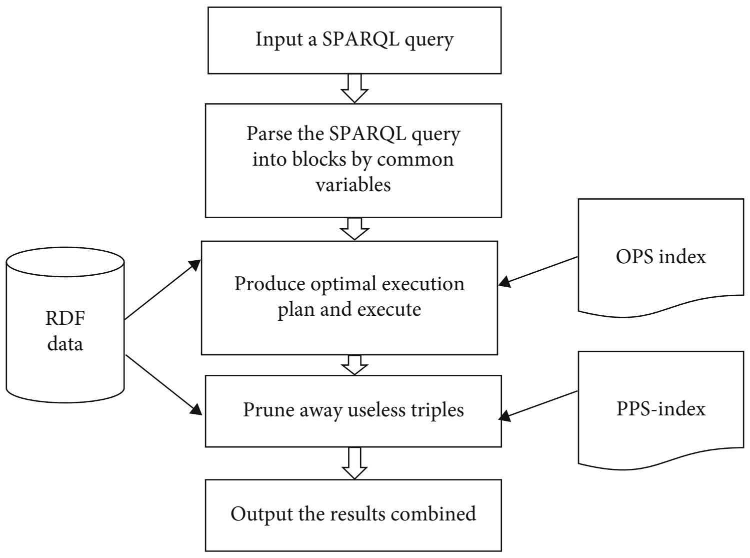

Input a SPARQL query
Parse the SPARQL query into blocks by common variables
RDF data
Produce optimal execution plan and execute
OPS index
Prune away useless triples
PPS-index
Output the results combined

FIGURE 2: The system process flow.

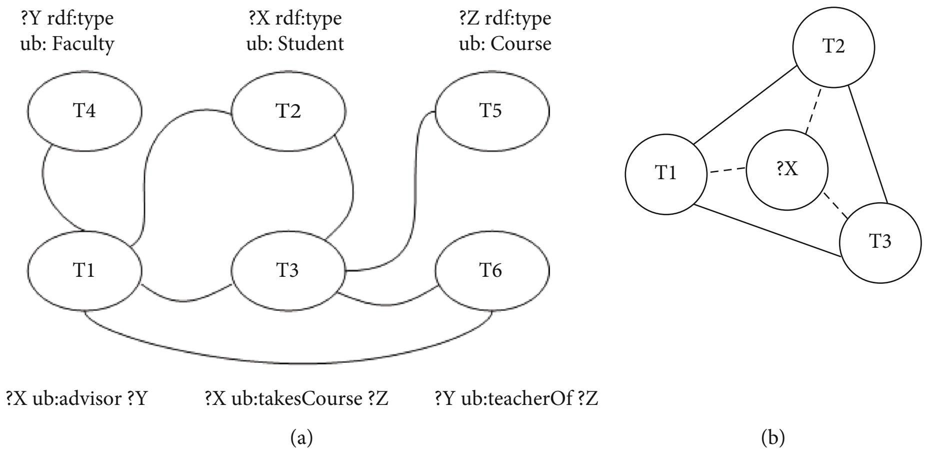


FIGURE 3: A query graph for Q9 and one of its blocks: a query graph for Q9 (a) and a block for Q9 query graph (b).

2.4. Predicate Path Sequence Indices (PPS-Indices). In the second stage, we construct the predicate path sequence indices to filter the remaining joins in the second stage. PPS-indices were extracted from large-scale RDF data graphs by the depth first search (DFS). The definition of the RDF data graph was adopted according to the study [29], as shown by Definition 1. We defined an RDF entity-relationship graph for building the PPS-indices, as defined in Section 2.

TABLE 2: OPS index.

Object	Predicate	SubjectId
"FullProfessor0"	Name	1(FullProfessor0)
"University0"	Name	2(University0)
"Department0"	Name	3(Department0)

Definition 1 (RDF graph). An RDF data graph is a tuple (V, L, E), where V is a finite set of vertices, V_E represents entity vertices, V_C represents class vertices, and V_V represents the data vertices, forming a disjoint union $V_E \uplus V_C \uplus V_V$. L is a limited set of edge labels, $L = L_R \uplus L_A$ {type, subclass}, where L_R represents the edge between entities and L_A represents the edge between entities and data values. E is a finite edge set; the form of the edge is $e(v1, v2)$, where $v1, v2 \in V_E$, $e \in L$, only $v1, v2 \in V_E$, $e \in L_R$; $e \in L_A$ if and only if $v1 \in V_E$ and $v2 \in V_V$; $e =$ type only when $v1 \in V_E$ and $v2 \in V_c$; and $e =$ subclass only when $v1, v2 \in V_C$.

Definition 2 (RDF entity-relationship graph, G'). An RDF entity relationship graph G' is a triple (V', L', E'), where vertex $V' = V_E$, edge label $L' = L_R$, and $E' \subset E$, E' connects the two entity vertices. V_E represents a collection of entity vertices (e.g., IRIs). Only when the edge $e(v1, v2) \in E'$ and $v1, v2 \in V_E$ exist in the RDF data does the edge $e(v1, v2) \in V_E$ exist in the RDF entity relationship graph.

PPS-indices are based on the RDF entity relationship graph, excluding the data value and type triples, as shown

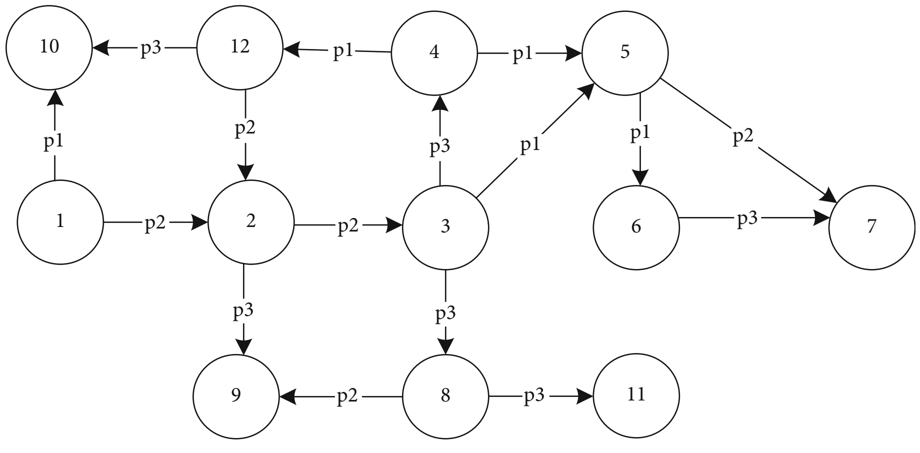


FIGURE 4: RDF entity relationship graph.

in Figure 4. The triples containing values are processed in the first stage.

Definition 3 (inbound predicate path sequence). The inbound predicate path is a sequence of triple predicates. If the terminal vertex of a path is *e*, it is said to be an inbound predicate path of *e*.

For example, for the join of "$?X$ ub : advisor$?Y.?Y$ ub : teacherOf$?Z$" in Q9, there is an inbound predicate path sequence of the variable "$?Z$", "ub : advisor $\longrightarrow$ ub : teacherOf". Table 3 shows the list of vertices pointed to by the predicate path index in Figure 4. Through such a list of vertices, useless intermediate vertices can be filtered out firstly, thereby avoiding more unnecessary join operations. For example, if there is a link in a SPARQL query, "$?n1\,p1?n2.?n2.p2?n3.?n3?p3?n4$", the predicate path "$<p1, p2, p3>$" can be found that contains the vertex Id list "8, 9" for filtering the result set, without having to merge a large number of intermediate result sets.

We obtained an n edge predicate sequence using the DFS algorithm. All bold edges in Figure 5 form a searching tree. The searching order of each vertex is marked below it. It can be seen from Figure 6 that the order of being searched for forward edges is $(1, 2)$, $(2, 3)$, and $(2, 4)$. We specify that for any vertex v', all of its backward edges should be behind the forward edge that points to v'. For vertex 3, its backward edge $(3, 1)$ should appear after $(2, 3)$. Among the backward edges from the same vertex, an order is enforced: given vertex Id i and its two backward edges, (i, j), (i, k); if $j < k$, then edge (i, j) will appear before edge (i, v). Thus, we obtain the ordering of edges in the RDF data graph. Based on the above order, a complete edge sequence is formed (Figure 5) as $<(1, 2), (2, 3), (3, 1), (2, 4)>$. We call this sequence the DFS code. Since there are a lot of different searching trees in a graph, each of which can form a DFS sequence code; a lexicographic order was designed to order the DFS code [30, 31]. We denote a labeled edge by its predicate without vertices. Thus, the edge predicate path sequence of Figure 5 can be written as $<p1, p1, p2, p2>$.

A PPS-index tree storing PPS-indices is shown in Figure 6, where each vertex stands for a sequential segment

TABLE 3: Vertex lists for predicate path indices.

Path length	Predicate path sequence	Vertex Id lists
1	$<p1>$	2, 5, 6, 12
	$<p2>$	2, 3, 7, 9
	$<p3>$	4, 7, 8, 9, 10, 11
2	$<p1, p2>$	2, 3, 7
	$<p2, p1>$	5
	$<p2, p2>$	3
	$<p2, p3>$	4, 8, 9
	$<p3, p1>$	5
	$<p3, p2>$	9
3	$<p1, p2, p3>$	8, 9
	$<p2, p1, p1>$	6
	$<p2, p1, p2>$	7
	$<p2, p3, p3>$	11
	$<p3, p1, p2>$	2, 7
	$<p3, p1, p3>$	10

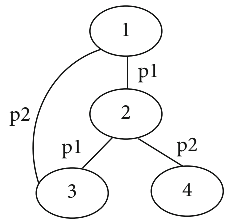


FIGURE 5: DFS code generation.

(a minimal sequence code). Here, two discriminative segments $f1 = <p1>$ and $f9 = <p1, p2, p1>$ are kept in the PPS-index tree. Pi is used to denote predicate of edges in DFS codes. Though segments $f2, f3$ on level 1 are not the discriminative segments, $f2$ and $f3$ have to be kept for connecting the vertices $f5$, $f6$, and $f7$.

The PPS-index tree is a prefix that keeps all discriminative segments with size n in level n (segments with size 0

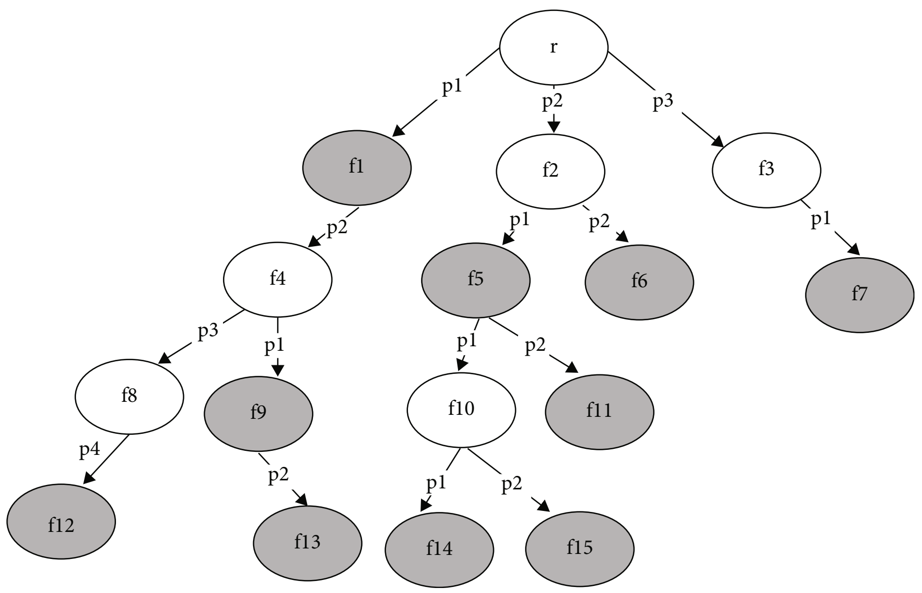


Figure 6: PPS-index tree.

denote subgraphs that have one vertex and no edges). This discriminative predicate path is adapted from the concept in an earlier study [32]. In this prefix tree, if t is a prefix of t', t is an ancestor of t'. All black vertices represent the discriminative segments, and white vertices represent the intermediate vertices of the whole PPS-index tree. In each black node, fi denotes a list of predicate path sequence; for instance, $f1$ is $<p1>$ and $f5$ is $<p2, p1>$. PPS-indices store the discriminative segments as well as useless ones. By using the PPS-index tree, the search time will get significantly reduced.

In Figure 7, the vertex variable "$?Z$" has two-direction inbound predicates, "<advisor, teacherOf >" and "<takesCourse >". According to a study [23], all incoming predicate paths with incoming predicates less than 3 were calculated, including "<advisor, teacherOf >", "<teacherOf >", and "<takesCourse >". The intermediate results were filtered by the longest incoming predicate path index in each direction, and finally, the result sets traversed are merged, as shown in Figure 8.

A prefix tree structure storing the PPS-indices is similar to an inverted index in information retrieval. Each inverted predicate path can be regarded as a word in the inverted index, and the vertex list is equivalent to the document Id sorted in the inverted index. There are many repeated parts in PPS-indices, as shown in Figure 5: $<p1, p2, p3>$, $<p2, p1, p1>$, $<p2, p1, p2>$, $\cdots$. All of them contain $p1, p2$ without the prefix tree; more storage units are needed to store duplicate path information. Therefore, by the prefix tree, the same predicate prefix can be stored only once, which can save space. By PPS-indices, the vertex list corresponding to the specified predicate path could be quickly found, so as to filter the redundant vertices. For the length of the predicate path, the maximum value of 3 is used according to the empirical value [23]. In order to compress and store data better, the actual vertex Id number is not stored, but the vertex Id number difference is similar to that of the compressed triples [4, 5].

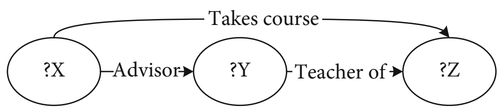
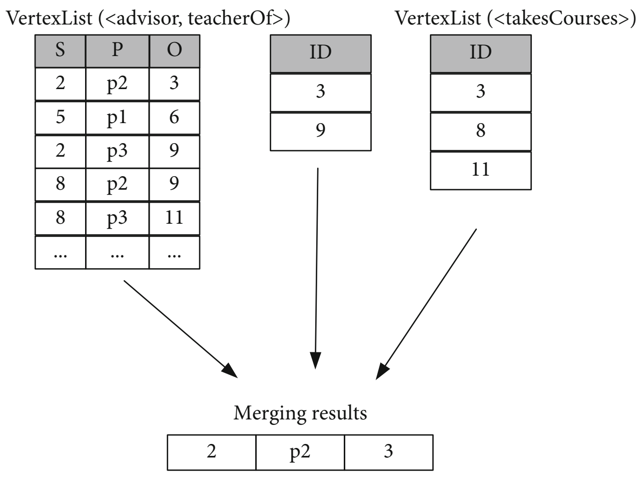


Figure 7: SPARQL query graph.


VertexList (<advisor, teacherOf>)

S	P	O
2	p2	3
5	p1	6
2	p3	9
8	p2	9
8	p3	11
...	...	...

ID
3
9

VertexList (<takesCourses>)

ID
3
8
11

Merging results

2	p2	3

Figure 8: Combined results.

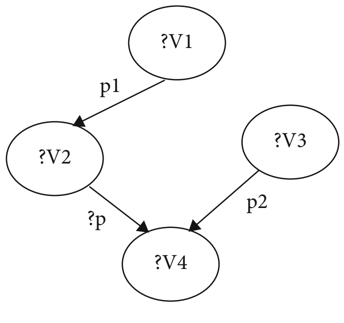


Figure 9: A SPARQL query with a predicate variable.

```
Input:SPARQL queries, queryStr, Trie;
Output:combined Results;
1 Parse queryStr;
2 Block queryStr into n blocks;
3 For i ←0 to n-1
4     Filter out joins with (?s, p, o) by getVariableSelectivity(queryStr.variable[i]) in order;
5 endFor
6 For each variable v ϵ Variables
7     filterList ← getVertexList(Trie,v);
8     Merge results using filterList; // By using Trie, useless vertices are filtered by the longest predicate path index of each variable
      and then the results will be combined.
9 endFor
10 Output Results;
```

Algorithm 1: Two-phase algorithm for SPARQL query processing.

Table 4: Statistics for datasets.

Datasets	The number of predicates	The number of triples	Size (GB)
YAGO2	94	195,948,254	7.8
LUBM	17	1,243,781,345	72
DBSPB	39,675	183,000,000	25
SP2B	77	1,399,000,000	123

Table 5: Size of PPS-index.

	Yago2	LUBM	DBSPB	SP2B
The number of vertex list (#)	24.977	132	145,157	120,065
Size (GB)	0.78	1.45	7.03	23.86

Table 6: Evaluation results of LUBM for Q1-Q5.

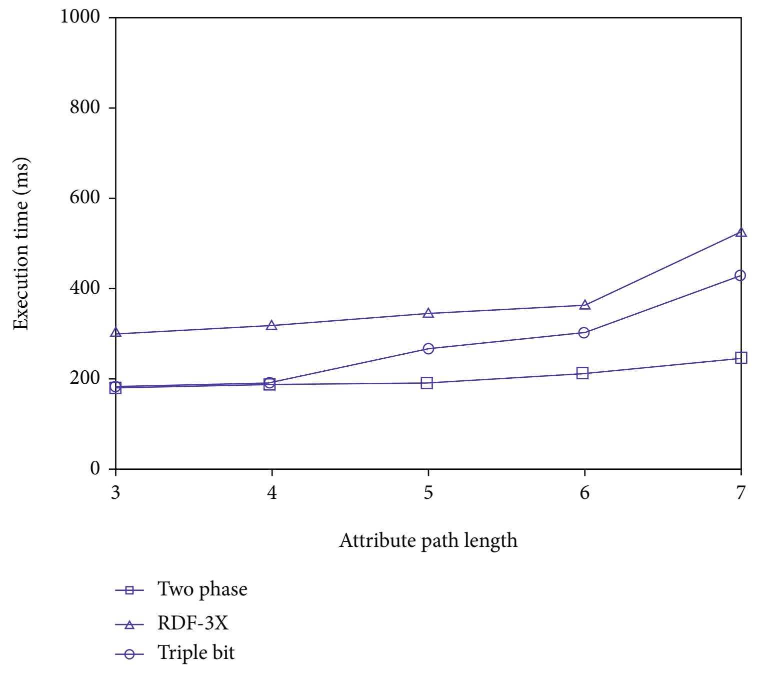
	Q1	Q2	Q3	Q4	Q5
RDF-3x	392,688,864	424,747,108	334,234,879	245,123,469	575,821,432
Two-phase	212,452,134	234,876,67	301,567,121	135,877,334	376,223,562
TripleBit	212,404,452	372,245,356	313,458,135	148,356,765	376,278,263

Table 7: Evaluation results of LUBM for Q6-Q9.

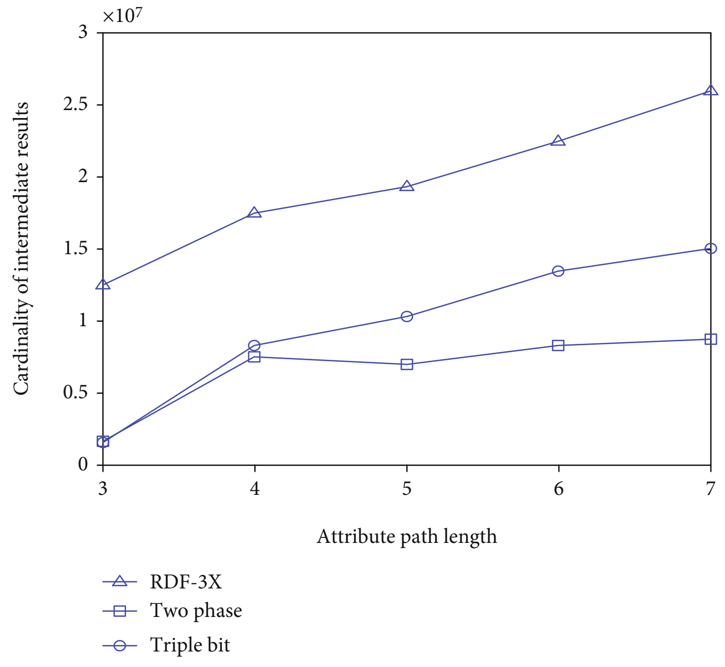
	Q6	Q7	Q8	Q9
RDF-3x	659,738,158	543,786,113	380,103,352	667,922,234
Two-phase	415,234,133	324,455,729	145,235,246	576,245,458
TripleBit	455,673,778	362,135,567	208,356,113	621,472,876

The SPARQL query includes a join whose predicate is a variable, as shown in Figure 9. One method is to remove all paths which include the predicate variables. Although this is relatively simple, the ability of predicate path filtering is limited. The other method is to consider the predicate variable as a special predicate and to replace it with *pv*, which is a bit like the wildcard "?". In Figure 9, the predicate path of all incoming edges of the vertex "?*v*4" is $<p1, pv>$, $<p2>$, and $<pv>$; replace "?*p*" with *pv*, where VertexList $(p1, pv)$ is the set of vertices corresponding to the predicate path, $p1 \longrightarrow pv$, and the first predicate of the predicate path was *p*1. We adopted the second method to deal with the predicate path containing predicate variables.

Our two-phase query processing algorithm is shown in Algorithm 1. The algorithm is mainly divided into three steps. In the first step, the input SPARQL query is analyzed and divided into blocks (steps 1-2). In the second step for each triple pattern in the block, $(?s, p, o)$ is filtered by the selectivity and getVariableSelectivity (steps 3-5), where the selectivity function reorders the joins of the form $(?S, P, O)$



(a)



(b)

Figure 10: Evaluation results for YAGO2: execution time (a) and cardinality of intermediate results (b).

, and the current query mode with the least join cost in the query pattern will be executed in order to reduce the amount of intermediate results generated in the query process as much as possible. In the third step, the remaining triples are filtered using PPS-indices Trie (step 6-9) which is the second round filtering of intermediate results. Because the predicate path considers at least one triple pattern, the predicate path filtering can filter out more intermediate results than the single triple pattern. The final merged result output is displayed in step 10. The join structure types of SPARQL are generally classified into star structure, string structure, and mixed structure type. In this study, the above types of SPARQL queries are selected for comparison experiments from the four datasets, LUBM, YAGO2, SPARQL benchmark dataset (SP2B), and DBpedia SPARQL benchmark dataset (DBSPB). From the experiments on all types of SPARQL queries, we verify the correctness and effectiveness of the two-phase algorithm proposed. The time complexity

Table 8: SPARQL queries for DBSPB and SP2B.

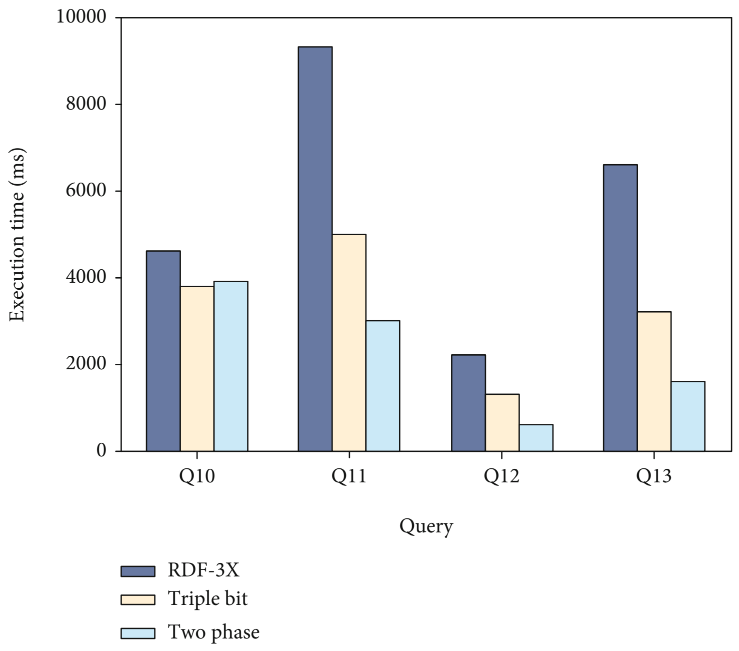
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>	
PREFIX dbpprop:http://dbpedia.org/property/	
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>	
PREFIX dbpowl:http://dbpedia.org/ontology/	
PREFIX geo:http://www.w3.org/2003/01/geo/wgs84_pos#	
PREFIX foaf:http://xmlns.com/foaf/0.1/	
PREFIX dc:<http://purl.org/dc/elements/1.1/>	
PREFIX dcterms:http://purl.org/dc/terms/	
PREFIX bench:http://localhost/vocabulary/bench/	
PREFIX swrc:<http://swrc.ontoware.org/ontology#>	
Q10	SELECT ?x ?y ?z ?m ?n ?p ?q ?r ?s ?t WHERE{?x dbpprop:ground ?y. ?x foaf:homepage ?z. ?m rdfs:label ?n. ?m dbpowl:postalCode ?p. ?m geo:lat ?q. ?m geo:long ?r. ?y dbpowl:location ?m. ?y foaf:homepage ?s. ?t dbpprop:clubs ?x}
Q11	SELECT ?x ?y ?z ?m ?n ?p ?q ?s ?t WHERE{?x rdf:type dbpowl:Person. ?x dbpprop:name ?z. ?n rdfs:label ?p. ?x dbpprop:placeOfBirth ?m. ?n dbpprop:isbn ?t. ?n dbpprop:author ?x. ?q dbpprop:author ?s. ?s rdfs:label ?y. ?n dbpprop:precededBy ?q. ?s dbpprop:name ?z. ?s dbpprop:placeOfBirth ?m}
Q12	SELECT ?x ?y ?z ?m ?n ?p ?q ?r ?s WHERE{?x foaf:name ?y. ?x rdfs:comment ?z. ?x rdf:type ?m. ?x dbpprop:series ?n. ?n dbpowl:starring ?p. ?p rdf:type ?s. ?q dbpowl:starring ?p. ?r dbpowl:previousWork ?q}
Q13	SELECT ?x ?y ?z ?m ?n ?p ?q ?r ?s WHERE{?x foaf:name ?y. ?x rdfs:comment ?z. ?x rdf:type ?m. ?x dbpprop:series ?n. ?n dbpowl:starring ?p. ?p rdf:type ?s. ?q dbpowl:starring ?p. ?r dbpowl:previousWork ?q}
Q14	SELECT ?x ?y ?z ?m ?n ?p ?q ? r ?s ?t WHERE{?x dcterms:references ?y. ?x a bench:Inproceedings. ?y rdf:_1 ?z. ?y rdf:_2 ?m. ?z dcterms:references ?n. ?n rdf:_1 ?p. ?n rdf:_2 ?q. ?m dcterms:references ?r. ?r rdf:_1 ?s. ?r rdf:_2 ?t}
Q15	SELECT ?x ?y ?z WHERE{?x swrc:editor ?y. ?z dc:creator ?y. ?z dcterms:partOf ?x}
Q16	SELECT ?x ?y ?z ?m ?n ?p ?q WHERE{?x dc:creator ?y. ?y foaf:name ?z. ?x dc:title ?m. ?x bench:abstract ?n. ?x dcterms:references ?p. ?p rdf:_50 ?q}
Q17	SELECT ?x ?y ?z ?m ?n ?p ?q WHERE{?x swrc:editor ?y. ?z swrc:editor ?y. ?y foaf:name ?m. ?x dc:creator ?y. ?x dc:title ?n. ?x dcterms:references ?p. ?p rdf:_10 ?q}

of the algorithm is mainly determined by step 6 to step 9, which is $O\ (n * e^{-})$, where n is the number of variables in the input SPARQL query, and e^{-} is the average inbound edge of each variable in the SPARQL query. The results indicate that the time of the algorithm is affected by the number of variables in the SPARQL query and the number of incoming edges of the variables.

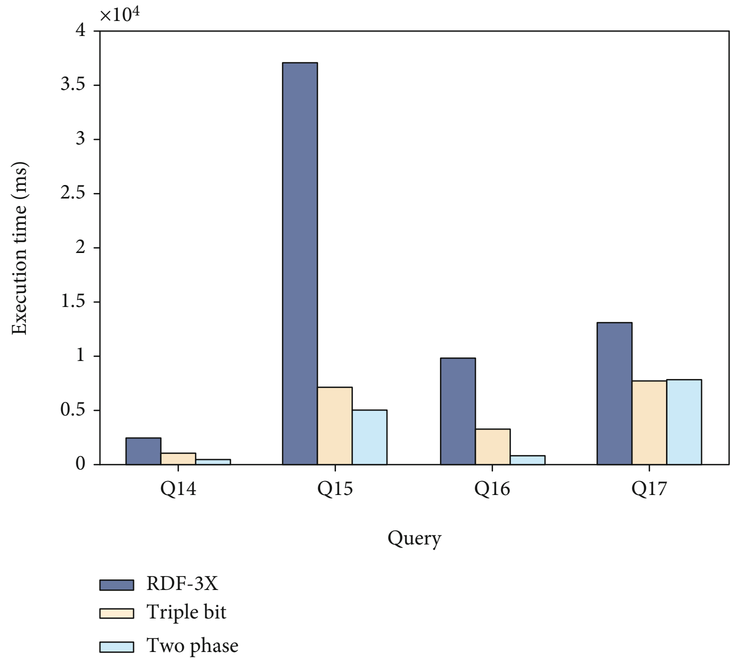
2.5. Evaluation. Public datasets, LUBM, YAGO2, SPARQL benchmark dataset (SP2B) [33], and DBpedia SPARQL benchmark dataset (DBSPB) [33], were utilized for the study. The open RDF-3X [4, 5] system (version 0.3.5) and TripleBit [27] were used for the evaluation experiments. Through the generator of LUBM, LUBM datasets of 10,000 universities are obtained. Table 4 shows the statistical information of these four datasets. Among them, YAGO2 contains 94 predicates, LUBM and SP2B contain 17,77 predicates, respectively, and DBSPB contains 39,675 predicates. DBSPB is a merger of multiple domain datasets, YAGO2 is a merger of heterogeneous datasets WordNet and Wikipedia, and LUBM and SP2B are descriptions of the relationship between similar domains. The size of the PPS-indices with a maximum length of 3 is shown in Table 5, including the number of vertices pointed to by the predicate path sequence index and the size of the entire index. Because the LUBM dataset exhibits the characteristics of a single structured model, it contains fewer vertices.

We employed the top nine queries Q1-Q9 in the benchmark query of the LUBM dataset in the comparative experiments. Tables 6 and 7 show the number of intermediate results for queries Q1-Q9. From the comparison, it was revealed that a simple structure query or a complex structure query such as Q2 and Q9 generated fewer intermediate results than RDF-3X. For queries Q1 and Q5, the number of intermediate results by our method was relatively close to TripleBit, because no filterable predicate path index was found. In future work, we will try to continuously update the PPS-indices. From YAGO2, we selected 10 queries each with a predicate path sequence length of 3-7 and calculated the average query execution time and the number of intermediate results.

We compared the proposed methods with RDF-3X and TripleBit. From Figure 10, it can be observed that except for the predicate path sequence length of 3, the execution time and the number of intermediate results were generally lower than those of RDF-3X, and the execution time and the amount of intermediate results increased slowly with the variation in the length of the predicate path. We noted an increase in the time curves of our method, RDF-3X, and TripleBit (Figure 10), with an increase in the predicate path length. Although our approach did not outperform TripleBit as a whole (Figure 10), when the path length was 3 or 4, our curve was very close to that of TripleBit. Furthermore, the execution time curve was better than that of RDF-3X and TripleBit. TripleBit chooses the query pattern corresponding to the smallest



(a)



(b)

FIGURE 11: Execution time: DBSPB(a) and SP2B(b).

selective variable, to reduce the size of the intermediate result set in the query process. The main reason for the performance degradation of TripleBit is that the query contains more joins or accesses more index blocks. As a result, for queries with a small number of predicate path lengths, the filtering effect was similar to that of TripleBit. For queries with a long predicate path, our proposed method outperformed TripleBit.

In order to verify that our method is applicable to various RDF datasets, queries Q10-Q13 and Q14-Q17 were constructed for the datasets DBSPB and SP2B, which are shown in Table 8. These queries consist of 4 to 5 join times and have a long predicate path. The execution time of the query is shown in Figures 11(a) and 10(b). We found that except queries Q10 and Q17, the query execution time achieved in these two datasets was better than that of RDF-3X and TripleBit. The execution time of queries Q11, Q12, Q13, Q15, Q14, and Q16 in Table 8 is significantly less than that of RDF-3X and TripleBit. Since we filtered out a large number of useless intermediate results using the PPS-indices, TripleBit dynamically calculated the optimal selective triple pattern for queries with more joins and accessed more index blocks. However, for queries Q10 and Q17, although there was no TripleBit with high execution efficiency, there was no significant difference. Additionally, no filterable predicate paths were found in the PPS-indices.

3. Conclusion

In the present study, we proposed a two-phase SPARQL query processing solution. On the one hand, the join with the smallest number of joins was found through the calculation of the selectivity degree, and on the other hand, the PPS-indices were leveraged to prune away the redundant intermediate results. Although the PPS-indices consumed a certain amount of storage space, using space for time greatly improved the execution efficiency of the SPARQL queries. Furthermore, the proposed PPS-indices can be incrementally updated, with an update of the data. In addition, experiments using different datasets show that our solution is feasible and very effective in filtering the intermediate results. In the future work, the accuracy of the calculation of the selectivity of the triple connection will be further improved. In addition, we will try to filter the SPARQL queries with more complex structures through PPS-indices and apply the two-phase SPARQL query optimization to the parallel and distributed environments, such as MapReduce.

Data Availability

The datasets generated during and/or analyzed during the current study are available from the corresponding author on reasonable request.

Conflicts of Interest

The authors declare that there is no conflict of interest regarding the publication of this paper.

Acknowledgments

The work was supported by a grant from Basic Research Programs of Higher Education Institutions Launched by Office of Department of Education of Liaoning Province, China (No. LJKZ1121).

References

[1] K. Wilkinson, “Jena property table implementation,” Citeseer, 2006.

[2] J. Broekstra, A. Kampman, and F. van Harmelen, “Sesame: a generic architecture for storing and querying RDF and RDF schema,” *International semantic web conference*, 2002, pp. 54–68, Springer, Berlin, Heidelberg, 2002.

[3] D. J. Abadi, A. Marcus, S. Madden, and K. Hollenbach, “SW-Store: a vertically partitioned DBMS for semantic web data management,” *The VLDB Journal*, vol. 18, no. 2, pp. 385–406, 2009.

[4] T. Neumann and G. Weikum, “The RDF-3X engine for scalable management of RDF data,” *VLDB Journal*, vol. 19, no. 1, pp. 91–113, 2010.

[5] T. Neumann and G. Weikum, “x-RDF-3X,” *Proceedings of the VLDB Endowment*, vol. 3, 2010no. 1-2, pp. 256–263, 2010.

[6] C. Weiss, P. Karras, and A. Bernstein, “Hexastore: sextuple indexing for semantic web data management,” *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 1008–1019, 2008.

[7] L. Zou, M. T. Özsu, L. Chen, X. Shen, R. Huang, and D. Zhao, “gStore: a graph-based SPARQL query engine,” *The VLDB Journal*, vol. 23, no. 4, pp. 565–590, 2014.

[8] H. Oh, S. Chun, S. Eom, and K. Lee, “Job-optimized map-side join processing using MapReduce and HBase with abstract RDF data,” in *2015 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, pp. 425–432, Singapore, 2015.

[9] S. Wang, J. Jiao, and X. Zhang, “A semantic similarity-based subgraph matching method for improving question answering over RDF,” in *Companion Proceedings of the Web Conference 2020*, pp. 63-64, Taipei, Taiwan, 2020.

[10] N. Balcan, A. Blum, and Y. Mansour, “Exploiting ontology structures and unlabeled data for learning,” *International Conference on International Conference on Machine Learning*, vol. 28, pp. 1112–1120, 2013.

[11] A. Bellandi, B. Furletti, V. Grossi, and A. Romei, “Ontology-driven association rule extraction: a case study,” *Context & Ontologies:Representation and Reasoning*, vol. 10, pp. 1–10, 2007.

[12] C. Nikas, P. Fafalios, and Y. Tzitzikas, “Open domain question answering over knowledge graphs using keyword search, answer type prediction, SPARQL and pre-trained neural models,” in *International Semantic Web Conference*, pp. 235–251, Cham, 2021.

[13] X. Hu, J. Duan, and D. Dang, “Natural language question answering over knowledge graph: the marriage of SPARQL query and keyword search,” *Knowledge and Information Systems*, vol. 63, no. 4, pp. 819–844, 2021.

[14] H. Tran, L. Phan, J. Anibal, B. T. Nguyen, and T. S. Nguyen, “SPBERT: an efficient pre-training BERT on SPARQL queries for question answering over knowledge graphs,” in *International Conference on Neural Information Processing*, pp. 512–523, Cham, 2021.

[15] K. Joseph and H. Jiang, “Content based news recommendation via shortest entity distance over knowledge graphs,” in *Companion Proceedings of The 2019 World Wide Web Conference*, pp. 690–699, San Francisco, USA, 2019.

[16] H. S. Sheu, Z. Chu, D. Qi, and S. Li, “Knowledge-guided article embedding refinement for session-based news recommendation,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 99, pp. 1–7, 2021.

[17] M. Mountantonakis and Y. Tzitzikas, “Content-based union and complement metrics for dataset search over RDF knowledge graphs,” *Journal of Data and Information Quality*, vol. 12, no. 2, pp. 1–31, 2020.

[18] H. Liu, D. Dou, R. Jin, P. LePendu, and N. Shah, “Mining biomedical ontologies and data using RDF hypergraphs,” *Machine Learning and Applications (ICMLA)*, vol. 1, pp. 141–146, 2013.

[19] X. Jian, Y. Wang, X. Lei, L. Zheng, and L. Chen, “SPARQL rewriting: towards desired results,” in *SIGMO '20: International Conference on Management of Data*, pp. 1979–1993, Portland, OR, USA, 2020.

[20] C. Marinica and F. Guillet, “Knowledge-based interactive post-mining of association rules using ontologies,” *Knowledge and Data Engineering*, vol. 22, no. 6, pp. 784–797, 2010.

[21] C. Marinica, F. Guillet, and H. Briand, "Post-processing of discovered association rules using ontologies," in *2008 IEEE International Conference on Data Mining Workshops*, pp. 126–133, Pisa, Italy, 2008.

[22] G. Mansingh, K.-M. Osei-Bryson, and H. Reichgelt, "Using ontologies to facilitate post-processing of association rules by domain experts," *Information Sciences*, vol. 181, no. 3, pp. 419–434, 2011.

[23] K. Kim, B. Moon, and H. J. Kim, "R3F: RDF triple filtering method for efficient SPARQL query processing," *World Wide Web*, vol. 18, no. 2, pp. 317–357, 2015.

[24] P. Yuan, C. Xie, H. Jin, L. Liu, G. Yang, and X. Shi, "Dynamic and fast processing of queries on large-scale RDF data," *Knowledge and Information Systems.*, vol. 41, no. 2, pp. 311–334, 2014.

[25] Z. B. Ozger and N. Y. Uslu, "An effective discrete artificial bee colony based SPARQL query path optimization by reordering triples," *Journal of Computer Science and Technology*, vol. 36, no. 2, pp. 445–462, 2021.

[26] R. Singh, "Inductive learning-based SPARQL query optimization," *ICDSIA*, vol. 52, pp. 121–135, 2021.

[27] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu, "Triple-Bit," *VLDB*, vol. 6, no. 7, pp. 517–528, 2013.

[28] S. Harris and A. Seaborne, *SPARQL 1.1 query language*, Recommendation, World Wide Web Consortium, 2013, http://www.w3.org/TR/sparql11-query/.

[29] T. Tran, H. Wang, S. Rudolph, and P. Cimiano, "Top-k exploration of query candidates for efficient keyword search on graph-shaped (RDF) data," in *2009 IEEE 25th International Conference on Data Engineering*, pp. 405–416, Shanghai, China, 2009.

[30] X. Yan and J. Han, "gSpan: graph-based substructure pattern mining," in *2002 IEEE International Conference on Data Mining, 2002. Proceedings*, pp. 721–724, Maebashi City, Japan, 2002.

[31] X. Yan and J. Han, "CloseGraph: mining closed frequent graph patterns," in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 286–295, Washington, D.C, 2003.

[32] X. Yan, P. S. Yu, and J. Han, "Graph indexing based on discriminative frequent structure analysis," *ACM Transactions on Database Systems (TODS)*, vol. 30, no. 4, pp. 960–993, 2005.

[33] M. Morsey, J. Lehmann, S. Auer, and A. C. Ngonga Ngomo, "DBpedia SPARQL benchmark – performance assessment with real queries on real data," in *International semantic web conference*, pp. 454–469, Berlin, Heidelberg, 2011.