Expert Systems with Applications 41 (2014) 4596–4607

Contents lists available at ScienceDirect

Expert Systems with Applications

journal homepage: www.elsevier.com/locate/eswa

RG-index: An RDF graph index for efficient SPARQL query processing

Kisung Kim *, Bongki Moon, Hyoung-Joo Kim

School of Computer Science and Engineering, Seoul National University, 1 Gwanak-ro, Seoul 151-742, Republic of Korea

ARTICLE INFO

Keywords:
RDF
SPARQL
Query optimization
Triple filtering
Intermediate results

ABSTRACT

As the size of Resource Description Framework (RDF) graphs has grown rapidly, SPARQL query processing on the large-scale RDF graph has become a more challenging problem. For efficient SPARQL query processing, the handling of the intermediate results is the most crucial element because it generally involves many join operators. Recently, a triple filtering method, called the RP-filter, which uses a path-based index, was proposed. It can reduce the intermediate results effectively by filtering out irrelevant triples in advance. However, its filtering power is limited, because it uses only the path information of the RDF graph. In this paper, we extend the triple filtering method to exploit the graph-structural information, and propose the RDF graph index (RG-index). We address the problem of the RG-index, which is caused by the indexing of the graph patterns, by indexing only effective graph patterns for the triple filtering. In addition, we propose an efficient method for building the RG-index in which a frequent graph pattern mining algorithm is adapted. We conducted comprehensive experiments on large-scale RDF datasets and demonstrated that the RG-index can reduce redundant intermediate results more effectively than can the RP-filter.

1. Introduction

The Resource Description Framework (RDF) (Klyne & Carroll, 2004) was proposed as the core data model of the Semantic Web, and SPARQL (Prud'hommeaux & Seaborne, 2008) was recommended by W3C as the standard query language for RDF data. RDF is a flexible, schema-free and graph-structural data model. It has been utilized as a unified data model in various areas, such as bioinformatics (Belleau, Nolin, Tourigny, Rigault, & Morissette, 2008; Redaschi & Consortium, 2009), media data (Kobilarov et al., 2009), Wikipedia (Hoffart, Suchanek, Berberich, & Weikum, 2013), social networks (Mika, 2004), and government open data (Sheridan, 2010). Today, RDF data on the Web has become web-scale graph data, so that we can call it big graph data. However, the graph-structural data model of the RDF and the graph pattern matching nature of SPARQL queries pose significant challenges for efficient processing of SPARQL queries for large-scale RDF data.

In order to process large-scale RDF data, many RDF systems have been proposed, i.e., Jena (Carroll et al., 2004), Sesame (Broekstra, Kampman, & van Harmelen, 2002), SW-store (Abadi, Marcus, Madden, & Hollenbach, 2009), RDF-3X (Neumann & Weikum, 2008), etc. They store RDF data in relational tables and process SPARQL queries using relational operators, such as scan and join operators. We call these RDF systems relation-based RDF stores, because they use the relational model. The main problem of relation-based RDF stores is that they need too many join operations for evaluating SPARQL queries, especially those having complex and large graph patterns. Thus, to address this problem, many techniques have been proposed, i.e., the clustered property table (Carroll et al., 2004), vertical partitioning (Abadi et al., 2009), multiple indexing (Neumann & Weikum, 2008; Weiss, Karras, & Bernstein, 2008). These techniques have been focused on the optimized storage layout, indexing methods and efficient join processing. However, recently the handling method of the intermediate results during SPARQL query processing is also recognized as an important issue. It is because a plenty of redundant intermediate results can be generated during SPARQL query evaluation.

In order to handle the redundant intermediate results problem, U-SIP (Neumann & Weikum, 2009) and RP-filter (Kim, Moon, & Kim, 2011), have been proposed. Their objectives are to improve the query evaluation by reducing redundant intermediate results as early as possible. U-SIP uses a dynamic filtering method exploiting the natures of the merge join and the hash join. However, it has limitation that the filters are generated run-time and could not exploit the graph structural information of RDF graphs. In order to overcome this problem, RP-filter uses a path-based index which indexes the incoming path information of RDF graphs. It uses additional filtering operators in the execution plan to filter out

* Corresponding author. Tel.: +82 28801830.
E-mail addresses: kskim@idb.snu.ac.kr (K. Kim), bkmoon@snu.ac.kr (B. Moon), hjk@snu.ac.kr (H.-J. Kim).

http://dx.doi.org/10.1016/j.eswa.2014.01.027
0957-4174/

irrelevant triples among the input triples and determines the irrelevance of triples for the given query. However, RP-filter also has limitations that it uses only the path information.

In order to explain the limitation of RP-filter, let us consider the example in Fig. 1. It shows a SPARQL query graph, its execution plan, and four fragments of an RDF graph: R_1, R_2, R_3, and R_4. In the execution plan, $Join_1$ produces the intermediate results matching the subgraph q_1 in the query graph: g_1, g_2, and g_3 in the RDF graph. However, because only R_1 is matched to the query graph, it is the final result for the query, and g_2 and g_3 become redundant intermediate results. RP-filter can reduce these intermediate results using the necessary condition for the final results that the matching vertices for $?v_3$ should have two incoming predicate paths: $\langle p_3, p_2\rangle$ and $\langle p_4, p_2\rangle$. Using this necessary condition, RP-filter can avoid producing g_3 in $Join_1$, because v_{14} does not have the incoming predicate path $\langle p_4, p_2\rangle$. However, note that g_2 is still produced because v_6 has both incoming predicate paths and satisfies the necessary condition. In order to remove these intermediate results, we should be able to consider the graph-structural information.

In this paper, we propose a graph index called *RG-index* (RDF graph index). The RG-index indexes the graph patterns in the RDF graph rather than the path information, and therefore, it can enhance the filtering effects. It indexes the graph patterns in RDF graphs and can provide vertex lists matching for a specific graph patterns. In order to index the graph patterns, we adapt the gSpan (Yan & Han, 2002) algorithm, one of the most well known algorithms for mining frequent graph patterns. Originally, gSpan was developed for treating a transactional graph database, which comprises many small-size graphs. Thus, to apply the gSpan to the RDF graph, which is a single large graph, the gSpan algorithm has to be modified. We propose an adaptation method of gSpan for RDF graphs, such as a canonical representation of RDF graph patterns, calculating their support and reducing redundant pattern generation. Further, to efficiently processing graph pattern mining, we also propose a mechanism for caching the intermediate results.

The main problem arising from indexing the graph patterns is that the index size can grow prohibitively large. This is because there exists a large number of graph patterns, and the number of graph patterns grows exponentially with its size. We solve this size problem of the RG-index by indexing a fraction of the graph patterns rather than all possible graph patterns. This approach is applicable because the objective of the RG-index is to provide the filter data, and therefore, it is enough to index graph patterns that are effective for the triple filtering. Then, the problem becomes how to select the graph patterns that are effective for the triple filtering. To address this problem, we propose several techniques to reduce the size of the RG-index while retaining its filtering power.

The triple filtering is performed in a relational operator called *RFLT* (RDF Filter). The RFLT operator filters its input triples using the vertex lists from the RG-index. It uses a fast merging algorithm for the triple filtering, and therefore the filtering can be performed very efficiently. In order to integrate the RFLT operators into the cost-based query optimizer, we also elaborate the cost model and the cardinality estimation method for the RFLT operator.

The contributions of this paper are as follows.

1. We describe the design of the RG-index and propose an efficient building algorithm adapted from the gSpan algorithm;
2. We present the RFLT operator, which conducts triple filtering efficiently. In addition, we develop the cost model and the cardinality estimation method for the RFLT operator;
3. We describe the implementation of the RG-index in the RDF-3X system, which is an open source RDF store, and comprehensive experiments with very large-scale real-life and synthetic RDF datasets, which demonstrate that the performance of our methods is superior to that of the existing methods.

The remainder of the paper is organized as follows. In Section 2, we survey existing works for the SPARQL query processing. In this section, a detailed description of the RDF-3X is also provided. In Section 3, we provide a formal data model of the RDF and SPARQL and introduce some necessary notation. In Section 4, we present the proposed RG-index in detail and describe the algorithm used to build the RG-index. The processing of triple filtering and the RFLT operator are described in Section 6. We present a performance study in Section 7 and our conclusions in Section 8.

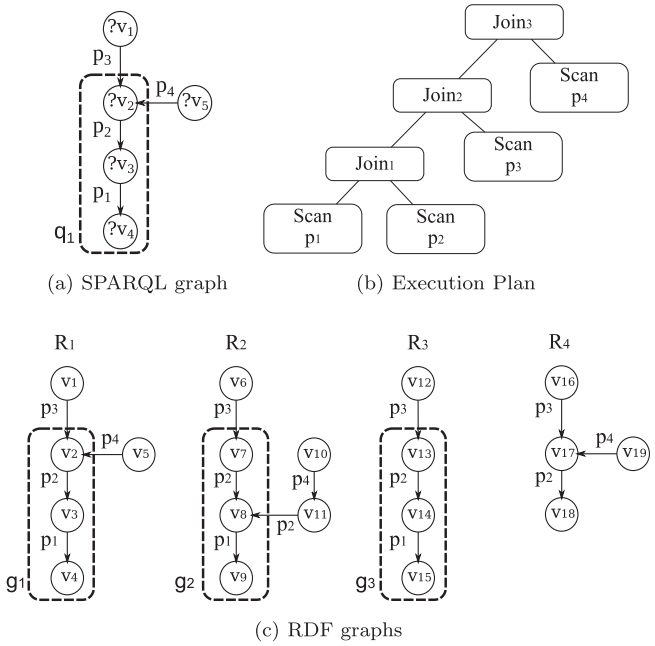


Fig. 1. RDF graph and SPARQL query graph.

2. Related work

In this section, we present previous work on RDF stores, and graph pattern mining and graph indexing.

2.1. RDF stores

Relation-based RDF stores use relational models to store RDF data and translate SPARQL queries into relational algebraic expressions (Chebotko, Lu, & Fotouhi, 2009). The main problem of relation-based RDF stores is that they need too many join operators to process SPARQL queries. Several approaches have been proposed for resolving this problem, which can be summarized as: (1) Reducing the number of joins; (2) making the join operators themselves efficient; and (3) reducing the inputs of join operators.

The property table can be considered as belonging to the first approach. It was proposed by Jena (Carroll et al., 2004) and Oracle (Chong, Das, Eadon, & Srinivasan, 2005), and reduces the number of joins by clustering several properties accessed together in a single property table. Because it stores the join results in a single table, it can reduce the number of joins. However, the property table approach has several problems in that it requires the users' clustering decisions and the previous knowledge about the query workload (Abadi et al., 2009). In addition, it incurs many null values or multi-values, which are hard to process, because it is created by denormalizing the triple table (Abadi et al., 2009).

Several methods of processing joins efficiently also exist. In SW-Store (Abadi et al., 2009), the triple table is partitioned vertically according to the predicate values. Since it uses a column-oriented store as its underlying store, triples are stored as sorted by the subject column. Therefore, the subject-subject joins can be processed efficiently using the fast merge join in SW-Store. However, the merge joins can be used only for the subject-subject joins in SW-Store. To extend the possibilities of using merge joins, in Hexastore (Weiss et al., 2008) and RDF-3X (Neumann & Weikum, 2008), the multiple indexing approach is applied. They index triples by all six possible orderings of (subject, predicate, object). Triples can be retrieved for any orderings, and merge joins can be used for joins other than the subject-subject join.

U-SIP (Ubiquitous Sideways Information Passing) (Neumann & Weikum, 2009) is used for reducing the inputs of join operators. U-SIP dynamically builds filters to provide information about the subject IDs or object IDs to be read next (we call this the next information). RDF-3X uses this next information to skip reading unnecessary disk blocks. While scanning the leaf blocks sequentially, if it determines that the next block can be skipped, it performs the B+tree index look-up to skip unnecessary blocks. In these ways, U-SIP can prune the triples that are irrelevant for the given query and reduce the input size of joins. U-SIP and our triple filtering method share the same objective of reducing the inputs of joins. However, our method exploits the graph-structural information, and therefore, it is more effective for RDF graphs.

There are also other approaches for SPARQL query processing. GRIN index (Udrea, Pugliese, & Subrahmanian, 2007), DOGMA (Bröcheler, Pugliese, & Subrahmanian, 2009), PIG (Tran & Ladwig, 2010), and gStore (Zou, Mo, Chen, Özsu, & Zhao, 2011) use graph-traversal approaches and graph indexing. These systems focus on reducing the search space of the graph traversing algorithms using the graph indices. While we also use a graph index (RG-index), our approach is different from these systems in that we focus on reducing the input size of joins in relation-based RDF stores.

Recently, RDF stores based on a clustered environment, such as MapReduce, have also been proposed, i.e., HadoopRDF (Husain, McGlothlin, Masud, Khan, & Thuraisingham, 2011), SHARD (Rohloff & Schantz, 2010), multi-node extension of RDF-3X (Huang, Abadi, & Ren, 2011), and Rya (Punnoose et al., 2012). In these distributed RDF systems, reducing the join inputs can improve the query performance more than do the single-node RDF stores, because it can reduce the network overhead for transporting intermediate results. RG-index can be applied in these systems.

2.2. Triple filtering method: RP-filter

The triple filtering method, RP-filter (Kim et al., 2011), was proposed for reducing the redundant intermediate results. In this method, an execution plan uses filtering operators, called *RFLT*, which conduct triple filtering for scan operators. Fig. 2 shows an execution plan using triple filtering. It uses two RFLT operators. The RFLT operators check that the subjects or the objects of the input triples satisfy the necessary structural conditions. For example, the $RFLT_1$ operator in Fig. 2 checks that the subjects of the input triples have two incoming predicates: $\langle p_3, p_2 \rangle$ and $\langle p_4, p_2 \rangle$. In order to check this condition, the method uses a path-based index called the *RP-filter*. The RP-filter contains the vertex lists, which contain vertex IDs with a specific incoming predicate path. The triple filtering can be performed by merging the input triples and the vertex lists, because the vertex lists are stored as sorted in the RP-filter.

2.3. Frequent graph pattern mining and graph indexing

There exist numerous bodies of literature focused on frequent graph pattern mining (cf. Cheng, Yan, & Han (2010) for detailed surveys). There are two problem formulations for graph mining (Kuramochi & Karypis, 2004): the graph-transaction setting, that is, many small graphs in a database; and the single-graph setting, that is, a large single graph. The graph-transaction setting has drawn more attention than the single-graph setting. The single-graph setting is more general, because several graphs can be combined into a single graph, and therefore, algorithms developed for the single-graph setting can be used for the graph-transactional setting (Kuramochi & Karypis, 2004). A frequent graph pattern mining algorithm first generates the candidate graph patterns, and then checks that its support is larger than the minimum support. If this condition is satisfied, the pattern is included in the results. The main focuses of the designers of frequent graph pattern mining algorithms are how to generate candidate graph patterns without generating duplicate patterns and how to prune infrequent patterns efficiently. To achieve these goals, they exploit the a priori principle (Cheng et al., 2010; Kuramochi & Karypis, 2004; Yan & Han, 2002), and canonical labeling mechanisms for representing the graph patterns are proposed. We adapt the gSpan (Yan & Han, 2002) algorithm to build the RG-index. It uses the DFS codes (Yan & Han, 2002) as the canonical representation of the candidate graph patterns and the depth-first manner of pattern generation. We discuss gSpan in more detail in Section 2.3.1.

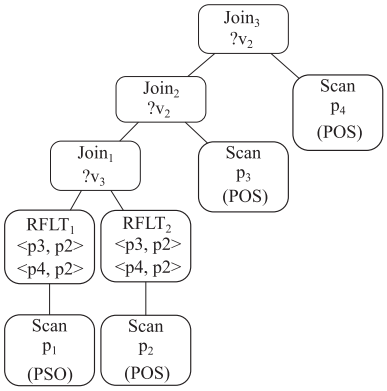


Fig. 2. Execution plan using the triple filtering.

Many graph indices have also been proposed for graph data. Most graph indices that have been proposed are for the graph-transaction setting, and they are focused on reducing the number of the graph isomorphism testing (e.g., GraphGrep (Shasha, Wang, & Giugno, 2002), gIndex (Yan, Yu, & Han, 2005), etc.). It is not easy to apply these indices to the RDF graph, because a single large graph is involved. Recently, graph indices for a large graph have also been proposed, such as SAGA (Tian, McEachin, Santos, States, & Patel, 2007), GraphQL (He & Singh, 2008), GADDI (Zhang, Li, & Yang, 2009), and SPath (Zhao & Han, 2010). Although these indices can be used in graph-based RDF stores, it is not trivial to apply them in relational-based RDF stores, because they were designed for use in the context of graph traversing algorithms.

2.3.1. Overview of gSpan

As already mentioned, we adapt gSpan algorithm (Yan & Han, 2002) to build RG-index. Therefore, we present its overview in this section. gSpan generates graph patterns in a depth-first fashion. That is, it starts from a 1-edge pattern and grows the pattern into larger patterns by adding one edge to the pattern. The most important issue in gSpan is minimizing the generation of the same graph patterns. Because a graph pattern can be generated in several ways, for efficient mining it is essential not to generate the patterns in duplicate. To achieve this, the pattern generation of gSpan is limited to the minimum DFS codes; otherwise, it is possible that the same patterns can be generated several times. If gSpan can ensure that all minimum DFS codes are generated, the generations of the

non-minimum DFS codes are redundant because any graph pattern can be represented by the minimum DFS code. Therefore, for each generated DFS code, gSpan checks whether it is the minimum DFS code for the generated pattern, and if not, the DFS code is pruned and not extended further.

In addition, to reduce the generation of non-minimum DFS codes, gSpan uses the rightmost extension when adding an edge to a graph pattern. The rightmost extension restricts the pattern growth as follows. For a DFS code, the first and the last vertices of the DFS traversal are called the *root* and the *rightmost vertex*, respectively. The path from the root to the rightmost vertex is called the *rightmost path*. The patterns can be grown such that a forward edge can be added to vertices in the rightmost path, and a backward edge can be added only to the rightmost vertex. If g is extended by adding e according to the rightmost extension, the extended pattern is denoted by $g \diamondsuit_r e$.

The reason why gSpan uses the rightmost extension is that patterns that are generated not by the rightmost extension are non-minimum DFS codes. Further, the rightmost extension guarantees that all minimum DFS codes are generated. Thus, gSpan guarantees the completeness of the mining results while reducing the duplicate pattern generation.

3. Preliminaries

3.1. RDF and SPARQL

In this section, we present the formal data model of the RDF and SPARQL. We assume the existence of three pairwise disjoint sets: a set of uniform resource identifiers (URIs) U; a set of literals L; and a set of variables VAR. We assume that blank nodes have their local URIs and treat them same as the resources. Variable symbols start with "?" to distinguish them from URIs and literals. A triple $t \in U \times U \times (U \cup L)$ (without variables) is called an RDF triple, and a triple $tp \in (U \cup VAR) \times U \times (U \cup L \cup VAR)$ (triple with variables) is called a triple pattern. It should be noted that in our model the joins that have predicate variables are not considered, because this join type is rarely used. In addition, various features of the RDF and SPARQL are omitted for simplicity. For example, some features of the RDF, such as data types, are not considered. We focus on SPARQL queries with basic graph patterns. A basic graph pattern is a set of triple patterns (Prud'hommeaux & Seaborne, 2008). Optional graph patterns and union graph patterns are not considered. However, our approaches can be applied to queries having these features with minor modifications.

An RDF database D is a set of RDF triples, and a SPARQL query Q is a set of triple patterns. For the RDF database D, a subset of U, P_D, is defined as a set of predicates. Formally, $P_D = \{p | p \in U \wedge \exists t(s,p,o) \in D\}$. An RDF database and a SPARQL query are mapped into an RDF graph and a query graph, respectively, as follows.

Definition 1 (*RDF graph*). We define an RDF graph for the RDF database D as $G_D = (V_D, E_D, L_D)$, where V_D is a set of vertices corresponding to the subjects and objects of all triples in D ($V_D \subseteq (U \cup L)$), E_D is a set of directed edges corresponding to all triples that are from the subjects to the objects, and L_D is an edge-label mapping, $L_D : E_D \to P_D$, such that $t(s,p,o) \in D, L_D(s,o) = p$.

The vertices in an RDF graph correspond to URIs or literals. It should be noted that URIs or literals are not considered vertex labels; rather, they are unique identifiers for vertices.

Definition 2 (*Query graph*). A query graph for a SPARQL query Q is defined as $G_Q = (V_Q, E_Q, L_Q)$, where V_Q is a set of vertices corresponding to the subjects and objects of all triple patterns in Q ($V_Q \subseteq (U \cup L \cup VAR)$), E_Q is a set of directed edges corresponding to all triples that are from the subjects to the objects, and L_Q is an edge-label mapping, $L_Q : E_Q \to P_D$, such that $tp(s,p,o) \in Q, L_Q(s,o) = p$.

As in the RDF graph, the vertices in the query graph are identified by the variable names, URIs or literals. Therefore, both G_D and G_Q are edge-labeled directed graphs. Figs. 1a and 1c show a SPARQL query graph and an RDF graph, respectively.

The evaluation of a SPARQL query consists of finding all possible variable bindings that satisfy the given query patterns. For the SPARQL query Q, substitution θ is a mapping $V_Q \cap VAR \to (U \cup L)$. $\theta(G_Q)$ is a graph whose variables are substituted by θ. The answer set for a SPARQL query is defined as follows.

Definition 3 (*SPARQL query answer*). The answer set for the SPARQL query Q w.r.t RDF database D is $Ans(Q) = \{\theta | \theta(G_Q)$ is a subgraph of $G_D\}$. For $v \in V_Q, Ans(Q, v)$ denotes the projection of $Ans(Q)$ over $v, Ans(Q, v) = \{\theta(v) | \theta \in Ans(Q)\}$, where $\theta(v)$ is the projection of mapping θ over v.

Example 1 (*SPARQL query answer*). For the RDF graph in Fig. 1c, the answer set of the SPARQL query in Fig. 1a is $Ans(Q) = \{(?v_1 \to v_1, ?v_2 \to v_2, ?v_3 \to v_3, ?v_4 \to v_4, ?v_5 \to v_5)\}$. Furthermore, the projection over $?v_1$ of $Ans(Q)$ is $Ans(Q, ?v_1) = \{v_1\}$.

3.2. Candidate vertex set

Irrelevant triples are filtered using the candidate vertex set concept. The candidate vertex set for a query vertex is a subset of the data vertices that could be included in the final results. The triple filtering is intended to remove irrelevant triples that are not included in the candidate vertex set, which can be defined as the neighborhood structural information of the query graph. We can define the candidate vertex set in various ways provided that it can be guaranteed that it is included in the final results. In this paper, we define the candidate vertex set using the neighbor subgraph information. We define the k-neighborhood subgraph as follows.

Definition 4 (*k-neighborhood subgraph*). Given a vertex v in a graph G, the k-neighborhood subgraph, denoted by $N(v,k)$, is a set of subgraphs that contain v and whose size is no more than k.

The k-neighborhood subgraph is applied to both the RDF graph and the query graphs. For example, Fig. 3 shows $N(?v3, 3)$ of the query graph in Fig. 1c.

We define the candidate vertex set using the subgraph patterns as follows.

Definition 5 (*Candidate vertex set*). Given a vertex v in a query graph G_Q and $maxL$, the candidate vertex set using the subgraph patterns, denoted by $CV(v, maxL)$, is a set of data vertices whose k-neighborhood subgraphs are the same as $N(v, maxL)$.

4. RG-index

In this section, we present the design of RG-index, and we also discuss its physical structure.

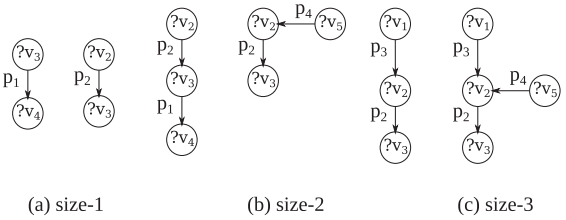


Fig. 3. k-Neighborhood subgraph.

4.1. Design of RG-index

RG-index is designed to provide the direct access to the filter data for the triple filtering. It maintains a set of vertex lists for subgraph patterns in the RDF database. A vertex list is built for every vertex of a graph pattern, and contains vertex IDs matching its query vertex. A vertex list is formally defined as follows.

RG-index indexes graph patterns in the RDF database. Only graph patterns whose vertices are all variables and the edge labels are all bounded, that is, not variable, are considered. We define the graph patterns as follows.

Definition 6 (*Graph pattern*). A graph pattern is a connected graph whose vertices are all variables and the labels of edges are all URIs.

It should be noted that a graph pattern can be viewed as a SPARQL query gp whose triple patterns satisfy the conditions: $\forall tp(s,p,o) \in gp, s \in VAR \wedge o \in VAR \wedge p \in P_D$. The vertex lists for a graph pattern are formally defined as follows.

Definition 7 (*Vertex list*). Given a graph pattern $G(V,E,L)$ and a vertex $v \in V$, a vertex list $Vlist(G,v)$ is $Ans(G,v)$, the projection over v for the answer set of G. A set of all vertex lists for G is denoted by $VS(G) = \{Vlist(v,G) | \forall v \in V\}$

In this definition, we treat a graph pattern as a query graph and use $Ans(G,v)$ to define the vertex list. RG-index is defined as follows.

Definition 8 (*RG-index*). RG-index for RDF database D with the maximum length $maxL$ is a set of pairs $\langle G, VS(G)\rangle$, where G is a graph pattern in D whose size is less than or equal to $maxL$.

Example 2 (*RG-index*). Fig. 4 shows an example of RG-index for the RDF graph in Fig. 1c. This RG-index indexes five graph patterns for the RDF graph and has fourteen Vlists.

Using RG-index, the candidate vertex sets for each vertex of the query graph can be obtained. The candidate vertex sets are obtained by intersecting relevant Vlists.

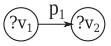
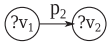
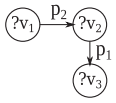
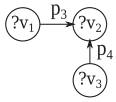
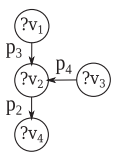
Size	Graph Pattern	Vlist
Size 1	gp_1 	$Vlist(gp_1,?v_1)=\{v_3, v_8, v_{14}\}$ $Vlist(gp_1,?v_2)=\{v_4, v_9, v_{15}\}$
	gp_2 	$Vlist(gp_2,?v_1)=\{v_2, v_7, v_{13}, v_{17}\}$ $Vlist(gp_2,?v_2)=\{v_3, v_8, v_{14}, v_{18}\}$
Size 2	gp_3 	$Vlist(gp_3,?v_1)=\{v_2, v_7, v_{13}\}$ $Vlist(gp_3,?v_2)=\{v_3, v_8, v_{14}\}$ $Vlist(gp_3,?v_3)=\{v_4, v_9, v_{15}\}$
	gp_4 	$Vlist(gp_4,?v_1)=\{v_1, v_{16}\}$ $Vlist(gp_4,?v_2)=\{v_2, v_{17}\}$ $Vlist(gp_4,?v_3)=\{v_5, v_{19}\}$
Size 3	gp_5 	$Vlist(gp_5,?v_1)=\{v_1, v_{16}\}$ $Vlist(gp_5,?v_2)=\{v_2, v_{17}\}$ $Vlist(gp_5,?v_3)=\{v_5, v_{19}\}$ $Vlist(gp_5,?v_4)=\{v_3, v_{18}\}$

Fig. 4. RG-index ($maxL = 3$).

Lemma 1 (*Candidate vertex set*). *Given a vertex v in a query graph G_Q and maxL, we can obtain a superset of $CV(v, maxL)$ by intersecting Vlists for k-neighborhood subgraphs of v.*

$$\bigcap_{gp \in N(v,maxL)} Vlist(gp,v) \subseteq CV(v,maxL) \quad (1)$$

Proof. By the definition of the k-neighborhood subgraph and its *Vlists*, for all v in gp, $Vlist(gp,v) \subseteq CV(v,maxL)$. Therefore, $\bigcap_{gp \in N(v,maxL)} Vlist(gp,v) \subseteq CV(v,maxL)$. □

4.2. Physical structure of RG-index

In this section, we discuss the physical structure of the RG-index. First, we describe how the graph patterns are represented in the RG-index. Then, we explain the storage of the RG-index.

4.2.1. DFS code representation

We use the minimal DFS code proposed for gSpan (Yan & Han, 2002) as the canonical representation of graph patterns. The minimal DFS code for a graph pattern gp is defined as follows. First, all nodes in gp are given DFS subscripts while they are traversed by a depth-first search. If two nodes are subscripted as v_i and v_j, and $i < j$, then v_i is traversed before v_j. It should be noted that, for a graph pattern gp, many different subscripts can be made, because there can exist several DFS trees for gp.

By using this subscription, each edge in gp is represented by a DFS edge. Originally, gSpan was designed to treat undirected graphs (Yan & Han, 2002), and DFS edge representation for undirected graphs was proposed. However, we are treating directed edge-labeled graphs. Therefore, the edge representation $\langle i,j,l_{(i,j)},d_{(i,j)}\rangle$, where i and j are DFS subscripts, $l_{(i,j)}$ is the edge label, and $d_{(i,j)}$ is the edge direction, is used. If the edge is from v_i to v_j, $d_{(i,j)} = \rightarrow$; otherwise, $d_{(i,j)} = \leftarrow$. A DFS edge with $i < j$ is called a forward edge, and a DFS edge with $i > j$ is called a backward edge. Forward edges are edges that are visited during the DFS search, and edges that are not visited become backward edges.

Using this DFS subscription and the DFS edge representation, a graph pattern can be mapped into a DFS code, which is a sequence of DFS edges. In the DFS code, DFS edges for edges of the graph patterns are sequenced as follows. Forward edges are ordered as they are discovered. Given a vertex v, all of its backward edges should appear after the forward edge pointing to v. Among the backward edges from the same vertex, say $(v_i, v_j), (v_i, v_k)$, if $j < k$, then (v_i, v_j) should appear before (v_i, v_k). gSpan defines a lexicographic order among DFS codes (Yan & Han, 2002). For two given DFS edges, the order is determined first by their two subscripts, then by edge labels, and finally by directions. We define the order between directions such that $\rightarrow$ is smaller that $\leftarrow$. gSpan defines the canonical label of gp as its lexicographically minimum DFS code.

Example 3 (*DFS code*). Fig. 5 shows a graph pattern (Fig. 5 (a)) and its three DFS subscriptions (Fig. 5 (b)–(d)). Each vertex is annotated by its subscription. Forward edges are represented by thick edges, while backward edges are represented by thin edges. Table 1 shows the DFS codes for three subscriptions. The order among the DFS codes is (d) < (c) < (d); (d) is the minimum DFS code for the graph pattern.

4.2.2. Storage of RG-index

Each vertex in the RDF database is assigned a four-byte integer ID. Physically, Vlists are stored as the sorted lists of these vertex IDs. Vlists are stored in a disk as sorted by vertex IDs so that the

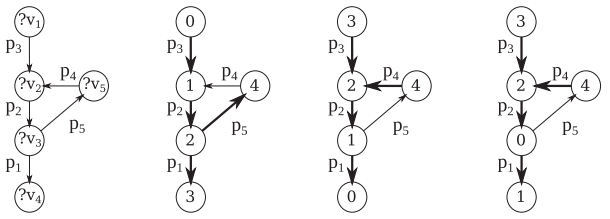


Fig. 5. DFS subscriptions.

Table 1
DFS codes of the graph pattern in Fig. 5(a).

Edge	(b)	(c)	(d)
1	$\langle 0,1,p_3,\rightarrow\rangle$	$\langle 0,1,p_1,\leftarrow\rangle$	$\langle 0,1,p_1,\rightarrow\rangle$
2	$\langle 1,2,p_2,\rightarrow\rangle$	$\langle 1,2,p_2,\leftarrow\rangle$	$\langle 0,2,p_2,\leftarrow\rangle$
3	$\langle 1,4,p_4,\leftarrow\rangle$	$\langle 1,4,p_5,\rightarrow\rangle$	$\langle 0,4,p_5,\rightarrow\rangle$
4	$\langle 2,3,p_1,\rightarrow\rangle$	$\langle 2,3,p_3,\leftarrow\rangle$	$\langle 2,3,p_3,\leftarrow\rangle$
5	$\langle 2,4,p_5,\rightarrow\rangle$	$\langle 2,4,p_4,\leftarrow\rangle$	$\langle 2,4,p_4,\leftarrow\rangle$

Vlist can be read from the disk as sorted. The reason for storing Vlists as sorted is to allow the filter data to be obtained by simply merging the relevant Vlists. Another benefit of sorting is that sorted Vlists can be compressed by the delta-based byte-level compression scheme similarly to compressed triples in RDF-3X (Neumann & Weikum, 2008). The delta between two vertex IDs is encoded with one header byte and the minimum number of bytes for the delta (1 byte $\sim$ 4 bytes). If the delta is smaller than 128, it is stored directly in the header byte, consuming only one byte. Otherwise, the header byte stores the byte length of the delta with its most significant bit set as 1 to indicate the delta is not small. This compression scheme alleviates the overall size overhead of Vlists and reduces disk I/O overhead for reading the Vlists.

The DFS codes of the graph patterns in RG-index are organized in a trie (or prefix tree) data structure. Each node in level l in the trie has a pointer to the Vlist for its associated length-l DFS code. The trie provides compact storage for the DFS codes, because the duplicated parts of the DFS codes can be shared. In addition, it provides an efficient way to access the Vlist for a given DFS code. The location in the disk of the Vlist for a graph pattern can be found by traversing the trie using the DFS code. The number of the nodes in the trie is equal to the number of the DFS codes in RG-index. For real-life datasets and a small *maxL* value, the trie is relatively small and can be resident in the main memory.

4.3. Handling the size problem of RG-index

Even if the graph patterns are limited to size *maxL*, it is still infeasible to index all possible subgraph patterns in the RDF database D, due to the exponential number of the possible graph patterns. Because the RG-index is designed to provide the filter data for the triple filtering, it does not have to index all possible graph patterns in D. Instead, by choosing and indexing only graph patterns with effective filtering power, its size can be reduced while its filtering power is retained. We discuss how to choose the graph patterns in Section 4.3.1.

In addition, there exist some graph patterns that need not be indexed, and redundant Vlists. We also discuss the handling of these redundant graph patterns and Vlists in this section.

4.3.1. Discriminative patterns

The first criterion is to store only Vlists with enough filtering power as compared to other replaceable Vlists. If $Vlist_i \supset Vlist_j$, $Vlist_i$ can be used in place of $Vlist_j$, because $Vlist_i$ has all vertices in $Vlist_j$. Therefore, it is possible to store only $Vlist_i$ and remove $Vlist_j$ from RP-index. However, this replacement can degrade the filtering power because the replacing filter is prone to produce more false positives than the replaced filter. Therefore, it is important to choose graph patterns that do not degrade the filtering power significantly. A discriminative Vlist is one whose Vlist cannot be replaced by another Vlist without significantly degrading the filtering power. We define the discriminative Vlist as follows.

Definition 9 (*Discriminative Vlist*). Given discriminative ratio γ $(0 < \gamma \leqslant 1)$ and a set of Vlists V, $vlist$ is discriminative w.r.t V iff $\forall vlist_s \in V \wedge vlist_s \in vlist$, $|vlist| < \gamma \times |vlist_s|$.

Example 4 (*Discriminative Vlist*). Let us assume that there exists two Vlists in $V = \{vlist_1, vlist_2\}$, $vlist_1 = \{v_1, v_2\}$ and $vlist_2 = \{v_1, v_2, v_3, v_4\}$. $vlist_1$ is included in $vlist_2$ and $|vlist_1|/|vlist_2| = 0.5$. Therefore, if $\gamma > 0.5$ $vlist_2$ is discriminative, otherwise it is not discriminative.

4.3.2. Frequent patterns

The second criterion is to store only frequent graph paths. A graph path is frequent iff its support is larger than the minimum threshold defined by the user. Infrequent graph paths are unlikely to be useful, because they are rare in RDF graphs and would not be queried frequently. Therefore, their removal from RG-index does not degrade the overall performance for most queries. Additionally, because infrequent graph patterns tend to be abundant, their removal can reduce the size of RG-index significantly. Since the number of patterns increases with their size, a size-increasing function is used to provide the threshold value for identifying frequent graph patterns. Thus, the overall index size can be reduced. We define a frequent graph pattern as follows.

Definition 10 (*Frequent Graph Pattern*). Given size-increasing function $\psi(l)$, a graph pattern G is frequent if and only if $sup(G) \geqslant \psi(|G|)$.

Example 5 (*Frequent Graph Pattern*). Let us assume that *maxL* is 10 and the frequency threshold function is $\psi(l) = ((l-1)/maxL)^2 \times 100$, which we use in the experiments. Also assume that there exists a Vlist whose size if 100 and whose graph pattern is size-5. Then, the Vlist is frequent because its size is larger than $\psi(5) = 16$.

5. Building RG-index

We build RG-index using the subgraph mining algorithm, gSpan, which was originally proposed for use in the transactional setting. In this section, we briefly review gSpan, and discuss how to adapt it in order to build the RG-index for the single large RDF graph.

5.1. RDF graph pattern mining using gSpan

We adapt the gSpan algorithm to mine frequent graph patterns in the RDF graph in order to build the RG-index. The modifications are (1) the support definition, (2) the restriction for the pattern generation, and (3) caching the intermediate results.

5.1.1. Support for the RDF graph

First, in order to apply the frequent pattern mining algorithm for the RDF graph, we need to measure the support of the generated patterns. gSpan was proposed for use in the context of the transactional setting, and the support for the transactional setting is easily defined as the number of graphs in the database matched for a graph pattern. This definition has the anti-monotonicity property, which is essential for efficient mining. However, it is not easy to define the support that satisfies this property in the single large graph setting (Kuramochi & Karypis, 2004). Several support definitions for the single large graph setting have been proposed. We use the definition of graph pattern frequency in Bringmann and Nijssen (2008).

Definition 11 (*Support of graph pattern*). Given a graph pattern $G(V, E, L)$, the support of G is $sup(G) = \min_{v \in V}(|Vlist(G, v)|)$.

This definition uses the minimum number of vertices of the graph pattern as the support, and is computationally efficient as compared to other support definitions for the single graph (Fiedler & Borgelt, 2007). In addition, it ensures the anti-monotonicity of the support. Using this support, only the size of Vlists for the graph pattern is required.

5.1.2. Avoiding redundant patterns

We restrict the pattern generation of gSpan such that it does not generate all possible patterns in the RDF graph. There exist graph patterns that become redundant due to the semantics of SPARQL. In fact, evaluating the basic graph patterns of SPARQL queries is not exactly the same as subgraph isomorphism. This is because pattern mapping is not bijective; that is, the different vertices in a query graph can be matched to a same vertex in the RDF graph. Let us take a look at the example in Fig. 6. In this figure, there are an RDF graph and three graph patterns: G_1, G_2, and G_3. These three graph patterns are all matched to the RDF graph, although G_2 and G_3 have more edges than the RDF graph. v_2 in the RDF graph is matched to several vertices in these patterns; i.e., $?v_2$ and $?v_3$ in G_2 are matched to v_2. Therefore, the Vlists for these vertices are identical; $Vlist(G_1, ?v_2) = Vlist(G_2, ?v_2) = Vlist(G_2, ?v_3) = Vlist(G_3, ?v_2) = Vlist(G_3, ?v_3) = Vlist(G_3, ?v_4) = \{v_2\}$. G_2 and G_3 are redundant because they have the same filtering power as G_1.

Formally, graph patterns having non-trivial automorphisms are redundant.

Lemma 2. *If a graph pattern G has a non-trivial automorphism θ, then $\forall v \in G \wedge \theta(v) \neq v, s.t. Vlist(G, v) = Vlist(G, \theta(v))$, where $\theta(v)$ is the matching vertex by θ.*

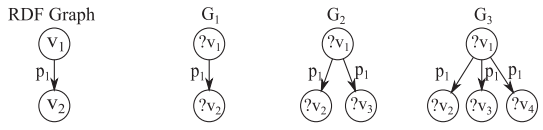


Fig. 6. Redundant graph pattern.

Proof. By the definition of the non-trivial automorphism, if G is a non-trivial automorphism θ and $v' = \theta(v)$, then $N(v, maxL) = N(v', maxL)$. Therefore, we can conclude that $Vlist(G, v) = Vlist(G, \theta(v))$. □

In this case, G has the same Vlists as the maximum subgraph of G, which does not have a non-trivial automorphism. Therefore, it is not necessary to generate graph patterns having non-trivial automorphisms.

In order not to generate graph patterns having automorphisms, automorphism checking should be performed for each generated graph pattern that is known to be NP-complete. Since this is too costly, we take an approximate approach instead. Patterns whose vertices have edges of the same type, i.e., edges with the same label and the same direction, are not generated. However, this method can remove graph patterns that are not redundant. For example, the graph pattern in Fig. 7 is removed because v_1 has two edges of the same type. However, it does not have non-trivial automorphism. Although the exclusion of these types of patterns can degrade the filtering power of the RG-index, in order to achieve efficient construction, these patterns are not considered.

5.1.3. Caching the intermediate results

The support of each generated pattern should be calculated and Vlists built for it. However, this process is very time-consuming because it requires finding all occurrences of the pattern in the RDF graph. The easiest way to perform this is to make and execute a SPARQL query for the generated pattern. However, this incurs many duplicate computations. Let us take a look at the example in Fig. 8. This figure illustrates a forward extension in which G_1 is extended to G_2. If the occurrences of these two patterns are calculated separately using two SPARQL queries generated for them, the subgraph of G_2, which corresponds to G_1, is calculated twice. This is because G_2 contains G_1.

In order to reduce these redundant computations, we propose caching the occurrences of a graph pattern and reusing them for

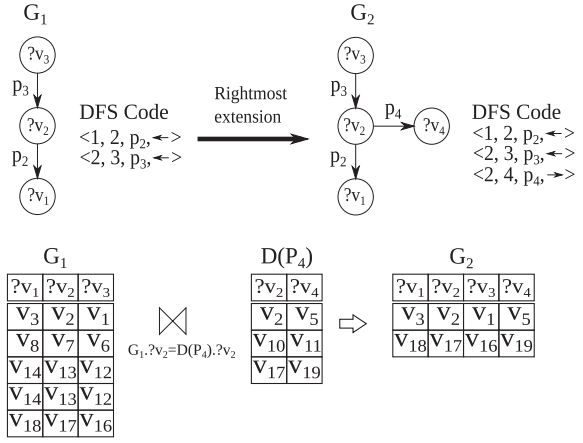


Fig. 8. Rightmost extension.

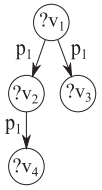


Fig. 7. Non-redundant graph pattern.

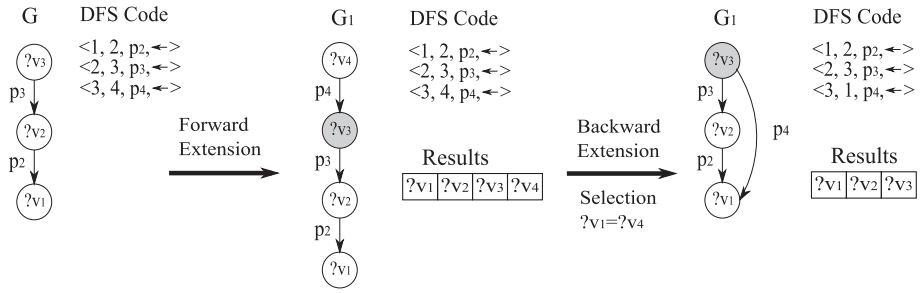


Fig. 9. Backward extension using the results of the forward extension.

its child graph patterns. Fig. 8 shows the entire process. The occurrences of the graph pattern are stored as a table whose columns correspond to each vertex in the graph pattern. Then, the occurrences of the child patterns can be obtained by a join operation for the table. In this example, the table G_1, which contains the occurrence results of G_1, is joined with the triple whose predicate is p_4, and the results become the occurrences of G_2.

In general, for a forward extension, the results can be obtained as

$$G_2 = \begin{cases} G_1 \bowtie_{v_i=S} D(p) & \text{if add} \quad \langle v_i, v_j, p, \leftarrow \rangle \\ G_1 \bowtie_{v_i=O} D(p) & \text{if add} \quad \langle v_i, v_j, p, \rightarrow \rangle \end{cases} \tag{2}$$

For a backward extension, the results are calculated as

$$G_2 = \begin{cases} \sigma_{v_j=O}(G_1 \bowtie_{v_i=S} D(p)) & \text{if add} \quad \langle v_i, v_j, p, \leftarrow \rangle \\ \sigma_{v_j=O}(G_1 \bowtie_{v_i=O} D(p)) & \text{if add} \quad \langle v_i, v_j, p, \rightarrow \rangle \end{cases} \tag{3}$$

For a backward extension, a selection operation is needed in addition to the join operation. The results of a backward extension can be obtained from the selection operation for the results of a forward extension. Fig. 9 shows an example of the rightmost extension for the rightmost vertex, $?v_3$. First, the results of the forward extension are calculated. Then, if the extension is for the rightmost vertex, it can be used for the backward extension (the backward extension is possible only for the rightmost vertex). If the backward extension is to add $\langle 3, 1, p_4 \rangle$, then the results can be obtained by performing the selection operation with the condition $?v_4 = ?v_1$ for the results of the forward extension. This is efficient, because only the selection operation is required, and the results of the forward extension are reused.

It should be noted that the depth-first fashion of gSpan makes this approach more attractive, because it exploits the parent's results for its children. The results can be stored in table-form in the main-memory. If their size is too great to store in the main-memory, they can be saved on disk. The number of results set to be kept is bounded as *maxL*.

Algorithm 1. gSpanRDF (s, D, minSup, RGindex)

Input: an RDF database D and *minSup*
1: $n \leftarrow |V|$;
2: /* rightmost extension */
3: **for all** $v \in$ the rightmost path of s **do**
4: **for all** $p \in P$ and $d \in \{\leftarrow, \rightarrow\}$ **do**
5: /* forward extension */
6: $e \leftarrow \langle v, v_{n+1}, p, d \rangle$
7: **if** $s \diamondsuit_r e$ is not minimal **then**
8: continue;
9: **end if**
10: $G' \leftarrow$getOccurrenceForward(G, p, e);
11: **if** $s \diamondsuit_r e$ is frequent and discriminative **then**

* (*continued*)

Algorithm 1. gSpanRDF (s, D, minSup, RGindex)

12: Insert into RGindex
13: **end if**
14: **if** v is the rightmost vertex **then**
15: /* backward extension */
16: **for all** $v_j \in V \wedge v_j \neq v$ **do**
17: $e_b \leftarrow \langle v, v_j, p, d \rangle$
18: **if** $s \diamondsuit_r e_b$ is not minimal **then**
19: continue;
20: **end if**
21: $G_b \leftarrow$ getOccurrenceBackward(G', p, e);
22: **if** $s \diamondsuit_r e$ is frequent and discriminative **then**
23: Insert into RGindex
24: **end if**
25: **end for**
26: **end if**
27: **end for**
28: **end for**
29: **return**

Algorithm 1 shows the overall process of building RG-index. The function gSpanRDF is called recursively to generate graph patterns from 1-size to *maxL*-size. It adds an edge to the input DFS code. First, it performs the forward extension for every vertex in the rightmost path. It adds edges, varying the label with the predicates in the RDF database and its direction. Then, it checks that the generated DFS code is the minimal DFS code of its corresponding graph pattern. If not, the DFS code is pruned. Then, it calculates the occurrences of the graph patterns using Eq. (2). If the graph pattern is frequent and discriminative, the pattern and Vlists are inserted into the RG-index. Then, if the extension is for the rightmost vertex, it performs the backward extension. The edge for the backward extension is from the rightmost vertex and to the other vertex in the graph pattern. The DFS code should be also checked as to whether it is minimal. Then, the occurrences of the DFS code are calculated by performing the selection operation for the results of the forward extension, as previously explained.

6. Triple filtering using RG-index

In this section, we describe how the triple filtering is processed. First, we introduce the RFLT operator, which is the relational operator for performing the triple filtering, and then explain how to incorporate the query optimizer to make execution plans using RFLT operators.

6.1. RFLT operator

The RFLT operator is a relational operator that conducts the triple filtering. RFLT operators are added to an execution plan as a

parent operator of the scan operators. It filters out irrelevant triples retrieved from its child scan operators using the Vlists of RG-index, which are assigned by the query optimizer. It checks whether the subjects or objects of the input triples are included in the candidate vertex set for the corresponding query vertex, which are obtained from the intersection of the assigned Vlists.

The triple filtering is conducted using the merge process for the input triples and the assigned Vlists. We assume that the input triples are sorted according to their subjects or objects. This assumption is satisfied in many RDF stores because they store triples sorted in order to allow efficient retrieval of the matching triples and to use merge joins to combine them. We call the vertex corresponding to the subjects or objects by which the triples are sorted the *sortkey* of the scan operator. Because the Vlists are stored as sorted in the RG-index and the input triples are also sorted, the triple filtering can be performed by simply merging the Vlists and input triples; the triples not matched to Vlists are filtered out. This requires only the sequential reading of Vlists and input triples, and therefore, the triple filtering can be performed very efficiently.

An RFLT operator can perform triple filtering for multiple scan operators with the same sortkey. By making the RFLT operator perform the triple filtering for several scan operators, redundant processing of triple filtering can be reduced. In addition, because they have the same sortkey, their outputs should be joined; the sortkey becomes also the join variable. It can be also processed by the merge join because the input triples are all sorted. Hence, we design the RFLT operator to process merge join operations and triple filtering simultaneously.

6.2. Generating an execution plan with RFLT operators

Many RDF stores use a cost-based query optimizer to find optimal (or near-optimal) plans for SPARQL queries (Neumann & Weikum, 2008). In order to make plans that use RFLT operators, we need to provide the query optimizer with (1) its cost function, and (2) the estimation method for the output cardinalities.

6.2.1. Cost function of the RFLT operator

First, we discuss the cost function of the RFLT operator. The RFLT performs only the merge process for its inputs, i.e., Vlists and input triples. Therefore, its cost is linear with respect to its input size, as follows.

$$\text{I/O cost}: \ O\left(\sum_{g\in GS}\|Vlist(g,v)\|\right) \tag{4}$$

$$\text{CPU cost}: \ O\left(\sum_{scan\in ChildOP}|scan| + \sum_{g\in GS}|Vlist(g,v)|\right) \tag{5}$$

where $\|vlist\|$ is the number of blocks of *vlist*, *GS* is the set of assigned graph patterns, ChildOP is the set of child scan operators, and $|scan|$ is the cardinality of the *scan* operator.

6.2.2. Output cardinality estimation of the RFLT operator

The output cardinality of an RFLT operator is estimated as follows.

First, it is assumed that the following statistics are available: (1) The cardinalities of scan operators, i.e., the number of triples matching triple patterns; (2) the number of distinct values of the sortkey column; and (3) the number of vertices in a Vlist. These statistics are already available from indices in RDF-3X and the RP-index.

We first consider the RFLT operator having one scan operator. The set of distinct values for the sortkey column of the scan operator is denoted by *S*. The intersection of *S* and Vlists of the RFLT is denoted by $C = \cap_{g\in GS} Vlist(g,v)$. Then, the output cardinality of the RFLT operator can be estimated as

$$|\text{RFLT}| = |\text{Scan}| \ \times \frac{C}{S} \tag{6}$$

If the RFLT has several child scan operators, it performs not only the triple filtering but also the join operation for all child operators. Let us denote the intersection of the sortkey columns for child operators and Vlists of the RFLT by $J = \bigcap_{child\in RFLT.childs}|child| \cap C$. Then, the output cardinality of the RFLT operator can be estimated as

$$|\text{RFLT}| = |J| \times \prod_{scan\in Childs}\frac{|\text{scan}|}{S} \tag{7}$$

Briefly, the output cardinality of an RFLT operator is estimated using (1) the assumption of a uniform distribution for the values of the sortkey column, and (2) the estimation of the sortkey column values remaining after triple filtering, that is, the intersection size of the values of the sortkey column and Vlists.

Our method is very similar to the Characteristic Set (Neumann & Moerkotte, 2011), which was proposed for estimating the cardinalities of star-join queries. However, our method does not aim to replace the Characteristic Set, but rather to reflect the filtering effect in the cardinality estimation. We expect that exploiting the Characteristic Set in our estimation method will improve the estimation accuracy. Therefore, our method and the Characteristic Set have a complementary relationship.

6.2.3. Adding RFLT operators

We use the query optimization of RDF-3X, which is based on the bottom-up dynamic-programming (DP) framework (Neumann & Weikum, 2008). The query compiler maintains the DP table, in which the optimal plans for the subproblems of the query are stored. First, the optimizer seeds its DP table with scan operators for the triple patterns as solutions of the 1-size subproblems. For each scan operator created in the seeding phase, an RFLT operator is added as its parent operator. The query optimizer should assign to RFLT operators the Vlists for the triple filtering. It assigns to an RFLT operator the Vlists for the graph patterns, which are *k*-neighborhood subgraphs of the query graph for the sortkey of the child scan operators. However, it is not necessary to assign all *k*-neighborhood subgraphs. If there are two subgraphs, *s.t.* $g_i \subset g_j$, $Vlists(g_i, v)$ does not need to be assigned because $Vlists(g_j, v) \subset Vlists(g_i, v)$. For an RFLT operator, $RFLT.Vlist = \{Vlist(g,v) | g \in N(v, maxL) \wedge \nexists g' \in N(v, maxL)\ s.t.\ g \subset g'\}$.

Larger plans are then created by joining two plans from smaller problems. After making the join operator for two smaller problems, if the join is a merge join, the operator is converted into an RFLT operator and the child operators of the join operator become the child operator of one RFLT operator.

7. Experimental results

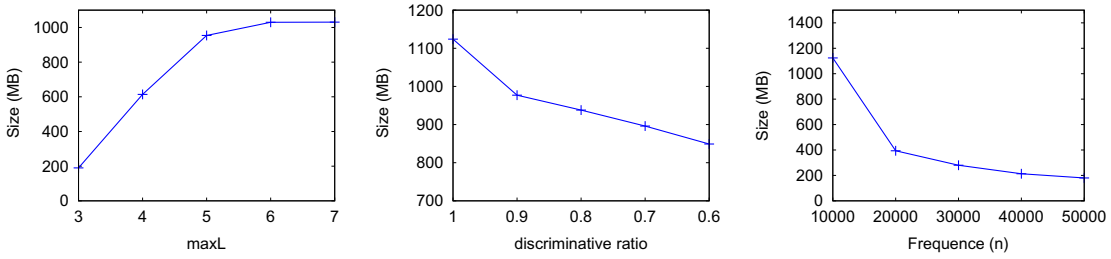
We implemented R3F on top of the open-source RDF-3X system (version 0.3.6).[1] R3F was written in C++ and compiled with g++ with the -O3 flag for the experiments. Our implementation included the RFLT operator, extension of the query optimizer, and the RP-index builder.

All the experiments were conducted on a hardware platform with eight 3.0 GHz Intel Xeon processors, 16 GB of memory, and running the 64-bit 2.6.31-23 Linux Kernel. We conducted the experiments using three datasets: Lehigh University Benchmark (LUBM) (Guo, Pan, & Heflin, 2005), Yet Another Great Ontology 2

[1] http://code.google.com/p/rdf3x/.

Table 2
Statistics about datasets.

	Predicates	URIs	Literals	Triples (millions)	RDF-3X size (GB)
LUBM	18	217,006,887	111,613,881	1335	77
YAGO2	93	6,872,931	22,452,390	37	9
SP2B	77	267,134,673	523,228,402	1399	123



(a) $maxL$ (b) γ (c) ψ

Fig. 10. RG-index size (YAGO2).

(YAGO2) (Hoffart et al., 2013), and SPARQL Performance Benchmark (SP2B) (Schmidt, Hornung, Lausen, & Pinkel, 2009). LUBM is a benchmark dataset whose domain is the university, YAGO2 is a knowledge-base derived from Wikipedia,[2] WordNet (Fellbaum, 1998), and GeoNames,[3] and SP2B is a benchmark that simulates the DBLP scenario.[4]

The benchmark datasets (LUBM and SP2B) have their own scale factors. We generated 10,000 universities for LUBM, and 96 GB triples for SP2B. These datasets have different characteristics, as shown in Table 2.

7.1. RG-index size

In this section, we present the experimental results for the size of RG-index. We built several RG-indices for SP2B varying *maxL*, γ (the discriminative ratio), and ψ (the frequency function). We used $\psi(l) = ((l-1)/maxL)^2 \times n$, which is the size-increasing function (l is the size of the graph pattern and n is determined properly for each dataset). We excluded some predicates for which there existed a large number of triples in order to reduce the overhead of building RG-index.

Fig. 10 shows the effects of three parameters: *maxL*, γ, and ψ (actually n in the function) for the size of RG-index of YAGO2 dataset. As can be seen in this figure, the size of the RG-index grows exponentially for the size of the graph patterns. However, we can adjust the size adequately using the two parameters, γ and ψ, for our purpose. Also, note that the size of the RG-index with adequate γ and ψ value is small as compared to that of the RDF database. We describe the effects of the parameters for the query performance in the next section.

7.2. Query evaluation performance

In this section, we present the effects of the RG-index for the query evaluation performance. First, we compare the query evaluation performances of RDF-3X, RP-filter, and RG-index. We built RG-indices and RP-filters for the three datasets. RP-filters index all incoming path patterns whose length is up to 7 (i.e., $maxL = 7$). We used the same *maxL* for RG-indices. We use $\gamma = 0.7$ and $\psi(l) = ((l-1)/maxL)^2 \times n$, where n is determined for each dataset to adjust the index size. Table 3 shows the statistics of RG-indices and RP-filters.

Because we remove some graph patterns by using the discriminative ratio and the frequency, RG-indices index fewer patterns than RP-filter. However, the number of Vlists in RG-index is larger than the number of graph patterns because a single graph pattern can have several Vlists.

In order to measure the query performance, we extracted graph patterns from each dataset and used them as the test queries. Table 4 shows the test query statistics. In this table, the queries are divided according to their execution times in RDF-3X. YAGO2 has many queries with short execution times because it is small and has many predicates. However, LUBM has a small number of predicates and a large number of triples. Therefore, its queries take a long time to evaluate. SP2B has intermediate characteristics between YAGO2 and LUBM.

Table 5 shows the averaged execution times. Both RP-filter and RG-index improve the query performance by more than about 30%. In addition, it can be seen that RG-index is more effective than RP-filter for YAGO2 and SP2B. In LUBM, RP-filter and

Table 3
Index statistics.

	Size	Number of patterns	Number of Vlists
(*a*) *YAGO2*			
RP-filter	341 MB	486,508	486,508
RG-index	1.1G	82,534	416,497
(*b*) *LUBM*			
RP-filter	1.4G	77	77
RG-index	2.4G	118	590
(*c*) *SP2B*			
RP-filter	1.3G	68,277	68,277
RG-index	1.3G	32,436	149,812

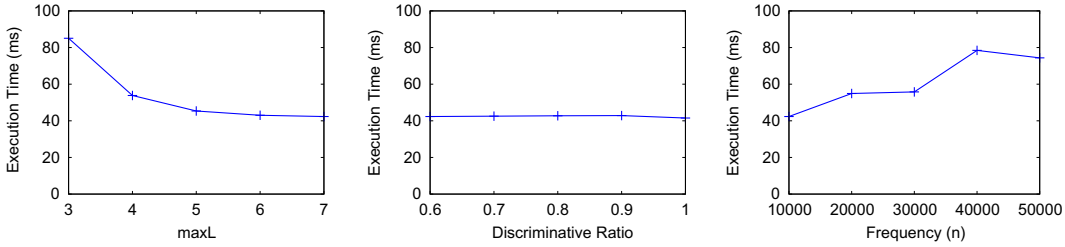
Table 4
Query Statistics.

Group Execution Time (ms)	A 0 ∼ 10	B 10 ∼ 100	C 100 ∼ 1000	D 1000∼	Total
Count (YAGO2)	638	126	12	0	1913
Count (LUBM)	0	5	15	51	71
Count (SP2B)	178	203	189	7	577

[2] http://www.wikipedia.org.
[3] http://www.geonames.org.
[4] http://www.informatik.uni-trier.de/ley/db/.

Table 5
Query Execution Time (ms).

Group	A	B	C	D	Total
(*a*) *YAGO2*					
RDF-3X	5.09	28.19	122.5	N/A	21.08
RP-filter	4.22 (17%)	19.38 (31%)	59.91 (51%)	N/A	14.58 (30%)
RG-index	3.56 (30%)	14.86 (47%)	43.58 (64%)	N/A	11.27 (46%)
(*b*) *LUBM*					
RDF-3X	N/A	53	540.8	134,490	114,385
RP-filter	N/A	50 (5%)	479.6 (11%)	90,290 (32%)	76,808 (32%)
RG-index	N/A	50 (5%)	477.2 (11%)	89,587 (33%)	76,209 (33%)
(*c*) *SP2B*					
RDF-3X	2.65	31.94	238.47	1361.42	106.68
RP-filter	2.37 (10%)	25.78 (19%)	168.19 (29%)	547.28 (59%)	71.53 (32%)
RG-index	2.32 (12%)	16.19 (49%)	103.60 (56%)	99.14 (92%)	41.55 (61%)



(a) $maxL$ (b) γ (c) ψ

Fig. 11. Query execution time (YAGO2).

RG-index show similar effects. This is because LUBM has a relatively structured data model, and therefore, there exists a small amount graph pattern that is effective for triple filtering. In addition, it should be noted that RG-index is more effective for queries with longer execution times. This is because the queries with long execution times have more intermediate results, which RG-index can reduce effectively.

Next, we measured the query performance varying RG-index parameters. We used RG-index, which was presented in Section 7.1, in order to present the effect of the parameters on the size of RG-index. Fig. 11 shows the results. We can improve the query performance by increasing the *maxL* value. However, the improvement decreases as the *maxL* value increases. Therefore, we should choose an adequate *maxL* value for the query workload. The discriminative ratio rarely affects the query performance. This is because the effective graph patterns remain for the small discriminative ratio. The frequency affects the query performance. Therefore, we should adapt the frequency considering the trade-off between the size and the query performance.

7.3. Discussions

As we can see in the previous section, RG-index can improve the query performance significantly by reducing the redundant intermediate results. By our observation, RG-index is specially effective for the queries with a large size of intermediate results. Surely, for queries with a small size of redundant intermediate results, RG-index does not improve the query performance much. Although our approach has apparent strengths, there also exist some issues to be discussed more.

7.3.1. Determining the parameter values

RG-index has three parameters, *maxL*, γ and $\psi(l)$. As we can see in the experimental results, these parameters affect the size of RG-index and the query evaluation performance. And there exists a trade-off between these metrics. Therefore it could be somewhat difficult to determining the values of these parameters. It could require some experiments and analyzing the characteristics of dataset. Also, it is also probably possible to automate the decisions using the statistical information of the data. We leave this issues for the future work.

7.3.2. Workload-aware index building

RG-index extracts the existing graph patterns in RDF graphs. However, as mentioned before, due to the size problem we could not index all existing patterns in RDF-graphs. Rather, we should limit the maximum size of indexed patterns or adapt several parameters in order to make the indices affordable to maintain. However, this approach could limit the performance of the triple filtering. For example, for the cases that the large size queries are general, RG-index with small *maxL* could be ineffective.

In addition, because we do not include infrequent patterns in the indices with the hypotheses that infrequent patterns are not liable to be queried, some effective infrequent patterns could be removed eventually. If these patterns are frequently used in the query workload, it will be better to decide to include this infrequent pattern in the index.

Therefore, we need a method which can make RG-index appropriate for the current query workload. It would be an interesting research topic to analyze the SPARQL query load and generate patterns to be indexed regarding the workload.

7.3.3. Accurate output cardinality estimation

The estimation of the output cardinality for each operator in an execution plan is very crucial for generating of an optimal execution plan. Actually, we have observed some cases that the query optimizer of RDF-3X generates non-optimal plans and the query performance degrades seriously. Therefore it is essential to estimate the output cardinalities to generate an optimal plan.

In addition, RFLT operator which conducts the triple filtering changes the output cardinalities of the target scan operators. Although we propose a cardinality estimation method for RFLT

operator, it has a limitation that it assumes the uniform distribution. We also observed some cases that the assumption does not hold, and therefore the estimation results are very poor. In our estimation method, we use the set intersection calculation. Although we use the upper bound for estimating the set intersection, there are other set intersection estimation methods. We can also apply other set intersection methods in order to make the cardinality estimation more accurate.

8. Conclusions and future work

In this paper, we proposed the RG-index, which we designed to improve the filtering power of the triple filtering method, RP-filter. The RG-index indexes the graph patterns in the RDF graph, and therefore, it can improve the query performance more than a triple filtering method that uses a path-based index. In order to address the size problem of the RG-index, we proposed indexing only the discriminative and frequent patterns. In addition, we proposed an efficient algorithm for building the RG-index, which is an adaptation of the frequent graph pattern mining algorithm, gSpan.

In our future studies, we plan to build RG-index considering the query workload. As already mentioned, to solving the size problem of RG-index, we should choose the patterns to be indexed. If we could consider the query workload for choosing the patterns, we expect that the filtering effects could be enhanced.

Also we need more accurate estimation method of the output cardinality. Actually, we have observed some cases that the query performance degrades seriously due to the non-optimal plans. Therefore it is essential to estimate the output cardinalities more accurately to generate an optimal plan.

Finally, as the big data emerges, the parallel distribution framework like MapReduce is used extensively for data processing. There exist already several methods for processing RDF data in these environments. In this distributed systems, the network transfer cost is very important factor for the performance. And in the MapReduce framework, the intermediate results should be materialized in the dist storage for provide query fail-over, handling the intermediate results is more serious problem. We expect that our triple filtering method could be very effective and its effect is more apparent in this environment. However, it requires how to store the indices and access the index data in the distributed index. We plan to extend our triple filtering method for the distributed environments.

Acknowledgement

This work was supported by Samsung Electronics Co. Ltd.

References

Abadi, D. J., Marcus, A., Madden, S., & Hollenbach, K. (2009). SW-Store: A vertically partitioned DBMS for semantic web data management. *VLDB J., 18*(2), 385–406.

Belleau, F., Nolin, M.-A., Tourigny, N., Rigault, P., & Morissette, J. (2008). Bio2RDF: Towards a mashup to build bioinformatics knowledge systems. *Journal of biomedical informatics, 41*(5), 706–716.

Bringmann B., & Nijssen S. (2008). What is frequent in a single graph? In *Advances in knowledge discovery and data mining, 12th Pacific-Asia conference, PAKDD 2008* (pp. 858–863).

Bröcheler, M., Pugliese, A., & Subrahmanian, V. S. (2009). DOGMA: A disk-oriented graph matching algorithm for RDF databases. In *Proceedings of the 8th international semantic web conference (ISWC 2009)*.

Broekstra, J., Kampman, A., & van Harmelen, F. (2002). Sesame: A generic architecture for storing and querying RDF and RDF Schema. In *Proceedings of the first international semantic web conference (ISWC 2002)*.

Carroll, J. J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., & Wilkinson, K. (2004). Jena: Implementing the semantic web recommendations. In *Proceedings of the 13th international conference on World Wide Web – alternate track papers & posters (WWW 2004)*.

Chebotko, A., Lu, S., & Fotouhi, F. (2009). Semantics preserving SPARQL-to-SQL translation. *Data Knowledge and Engineering, 68*(10), 973–1000.

Cheng, H., Yan, X., & Han, J. (2010). Mining graph patterns. In C. C. Aggarwal & H. Wang (Eds.), *Managing and mining graph data. Advances in database systems* (Vol. 40, pp. 365–392). Springer.

Chong, E. I., Das, S., Eadon, G., & Srinivasan, J. (2005). An efficient SQL-based RDF querying scheme. In *Proceedings of the 31st international conference on very large data bases (VLDB 2005)*. ACM.

Fellbaum, C. (Ed.). (1998). *WordNet: An electronic lexical database*. The MIT Press.

Fiedler, M., & Borgelt, C. (2007). Subgraph support in a single large graph. In *Workshops proceedings of the 7th IEEE international conference on data mining (ICDM 2007)*.

Guo, Y., Pan, Z., & Heflin, J. (2005). LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics, 3*(2-3), 158–182.

He, H., & Singh, A. K. (2008). Graphs-at-a-time: Query language and access methods for graph databases. In *Proceedings of the ACM SIGMOD international conference on management of data (SIGMOD 2008)*.

Hoffart, J., Suchanek, F. M., Berberich, K., & Weikum, G. (2013). YAGO2: A spatially and temporally enhanced knowledge base from Wikipedia. *Artificial Intelligence, 194*, 28–61.

Huang, J., Abadi, D. J., & Ren, K. (2011). Scalable SPARQL querying of large RDF graphs. *PVLDB, 4*(11), 1123–1134.

Husain, M., McGlothlin, J. P., Masud, M., Khan, L., & Thuraisingham, B. (2011). Heuristics-based query processing for large RDF graphs using cloud computing. *IEEE Transactions on Knowledge and Data Engineering, 23*(9), 1312–1327.

Kim, K., Moon, B., & Kim, H.- J. (2011). RP-filter: A path-based triple filtering method for efficient SPARQL query processing. In *Proceedings of the 2011 joint international semantic technology conference (JIST 2011)*.

Klyne, G., & Carroll, J. J. (2004). Resource description framework (RDF): Concepts and abstract syntax. In *W3c recommendation, World Wide Web consortium*.

Kobilarov, G., Scott, T., Raimond, Y., Oliver, S., Sizemore, C., Smethurst, M., Bizer, C., & Lee, R. (2009). Media meets semantic web — how the bbc uses dbpedia and linked data to make connections. In *Proceedings of the 6th European semantic web conference on the semantic web (ESWC'09)* (pp. 723–737).

Kuramochi, M., & Karypis, G. (2004). Finding frequent patterns in a large sparse graph. In *SDM*.

Mika, P. (2004). Social networks and the semantic web. In *Proceedings of international conference on web intelligence (WI'04)* (pp. 285–291).

Neumann, T., & Moerkotte, G. (2011). Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *Proceedings of the 27th international conference on data engineering (ICDE 2011)*.

Neumann, T., & Weikum, G. (2008). RDF-3X: A RISC-style engine for RDF. *PVLDB, 1*(1), 647–659.

Neumann, T., & Weikum, G. (2009). Scalable join processing on very large RDF graphs. In *Proceedings of the ACM SIGMOD international conference on management of data (SIGMOD 2009)* (pp. 627–640). Springer.

Prud'hommeaux, E., Seaborne, A. (2008). SPARQL query language for RDF. In *W3c recommendation, W3C recommendation*.

Punnoose, R., Crainiceanu, A., & Rapp D. (2012). Rya: A scalable RDF triple store for the clouds. In *Proceedings of the 1st international workshop on cloud intelligence (colocated with VLDB 2012) (Cloud-I 2012)*.

Redaschi, N., Consortium, U. (2009). UniProt in RDF: tackling data integration and distributed annotation with the semantic web. In *Nature precedings*.

Rohloff, K., & Schantz, R. E. (2010). High-performance, massively scalable distributed systems using the MapReduce software framework: The SHARD triple-store. In *SPLASH workshop on programming support innovations for emerging distributed applications*.

Schmidt, M., Hornung, T., Lausen, G., & Pinkel, C. (2009). SP2Bench: A SPARQL performance benchmark. In *Proceedings of the 25th international conference on data engineering (ICDE 2009)*.

Shasha, D., Wang, J. T.- L., & Giugno, R. (2002). Algorithmics and applications of tree and graph searching. In *Proceedings of the twenty-first ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems (PODS 2002)*.

Sheridan J. (2010). Linking UK government data. In *WWW workshop on linked data* (pp. 1–4).

Tian, Y., McEachin, R. C., Santos, C., States, D. J., & Patel, J. M. (2007). SAGA: A subgraph matching tool for biological graphs. *Bioinformatics, 23*(2), 232–239.

Tran, T., & Ladwig, G. (2010). Structure index for RDF data. In *Workshop on semantic data management (SemData@VLDB2010)*.

Udrea, O., Pugliese, A., & Subrahmanian, V. S. (2007). GRIn: A graph based RDF index. In *Proceedings of the twenty-second AAAI conference on artificial intelligence (AAAI 2007)*. AAAI Press.

Weiss, C., Karras, P., & Bernstein, A. (2008). Hexastore: Sextuple indexing for semantic web data management. *PVLDB, 1*(1), 1008–1019.

Yan, X., & Han, J. (2002). gspan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE international conference on data mining (ICDM 2002)*.

Yan, X., Yu, P. S., & Han, J. (2005). Graph indexing based on discriminative frequent structure analysis. *ACM Transactions on Database Systems, 30*(4), 960–993.

Zhang, S., Li, S., & Yang, J. (2009). Gaddi: Distance index based subgraph matching in biological networks. In *Proceedings of the 12th international conference on extending database technology (EDBT 2009)*.

Zhao, P., & Han, J. (2010). On graph query optimization in large networks. *PVLDB, 3*(1), 340–351.

Zou, L., Mo, J., Chen, L., Özsu, M. T., & Zhao, D. (2011). gstore: Answering SPARQL queries via subgraph matching. *PVLDB, 4*(8), 482–493.