World Wide Web (2015) 18:317–357
DOI 10.1007/s11280-013-0253-1

R3F: RDF triple filtering method for efficient SPARQL query processing

Kisung Kim · Bongki Moon · Hyoung-Joo Kim

Received: 6 September 2012 / Revised: 3 July 2013 /
Accepted: 26 August 2013 / Published online: 11 September 2013

Abstract With the rapid growth in the amount of graph-structured Resource Description Framework (RDF) data, SPARQL query processing has received significant attention. The most important part of SPARQL query processing is its method of subgraph pattern matching. For this, most RDF stores use relation-based approaches, which can produce a vast number of redundant intermediate results during query evaluation. In order to address this problem, we propose an RDF Triple Filtering (R3F) method that exploits the graph-structural information of RDF data. We design a path-based index called the RDF Path index (RP-index) to efficiently provide filter data for the triple filtering. We also propose a relational operator called the RDF Filter (RFLT) that can conduct the triple filtering with little overhead compared to the original query processing. Through comprehensive experiments on large-scale RDF datasets, we demonstrate that R3F can effectively and efficiently reduce the number of redundant intermediate results and improve the query performance.

Keywords RDF · SPARQL · Query optimization · Triple filtering · Intermediate results

K. Kim (✉) · B. Moon · H.-J. Kim
Department of Computer Science and Engineering, Seoul National University, Seoul, Korea
e-mail: kskim@idb.snu.ac.kr

B. Moon
e-mail: bkmoon@snu.ac.kr

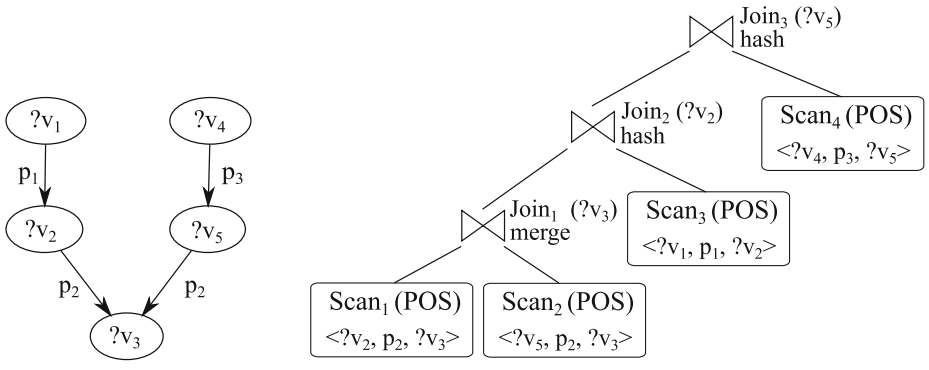
H.-J. Kim
e-mail: hjk@snu.ac.kr

1 Introduction

The Resource Description Framework (RDF) [23] is the core data model for the Semantic Web, and SPARQL [34] is the standard query language for RDF data. Currently, RDF is widely used to represent and integrate data from various domains in a flexible way. In general, RDF data can be modeled as a graph, and the evaluation of SPARQL queries can be considered as subgraph pattern matching on the RDF graph. Although its inherent graph structure gives a strong expressive power and flexibility to RDF, it also poses significant challenges for the processing of large-scale RDF data.

Most RDF systems employ a relational model. Examples of this are Jena [8], Sesame [7], SW-Store [1], Virtuoso [11], and RDF-3X [31]. Although the physical structures and detailed implementations are different for each RDF engine, they share a common framework for processing RDF data. That is, the RDF graph is stored in the form of relations (for example, relational tables in Jena and Sesame, or clustered B+tree indices in RDF-3X). SPARQL queries are processed using execution plans consisting of (1) operators for retrieving the matching triples, and (2) operators for combining the retrieved triples (the specific plans are different, as for different RDF engines, according to the physical storage layout and optimization techniques). For example, RDF-3X uses scan operators to retrieve matching triples and join operators to combine them. In this framework, each operator of the execution plan generates partially matching subgraphs for the query graph as its output. Let us consider the example of a SPARQL query and its execution plan in RDF-3X, as shown in Figure 1 (in this figure, each join operator is annotated with its join variable). $Join_1$ joins triples retrieved from $Scan_1$ and $Scan_2$ for variable $?v_3$, and outputs matched subgraphs for the subgraph pattern consisting of the two triple patterns $\langle ?v_2, p_2, ?v_3\rangle$ and $\langle ?v_5, p_2, ?v_3\rangle$.

Although this style of SPARQL query processing is widely used, it can produce a vast number of redundant intermediate results, especially when processing large-scale RDF data. The redundant intermediate results are those obtained from operators that are not included in the final results. In the previous example, not all



(a) Query Graph (b) Execution Plan

Figure 1 SPARQL query graph and execution plan

subgraphs generated from $Join_1$ contribute to the final results, because some of them are removed by subsequent join operators, i.e., $Join_2$ or $Join_3$. These redundant intermediate results waste processing resources without contributing to the query results. Moreover, for large-scale RDF data, it is possible that the overhead due to the redundant intermediate results dominates the overall query processing time. The main cause of this problem is that each operator simply generates all subgraphs matching its assigned subgraph pattern without considering any graph-structural information available in the RDF data.

In this paper, we propose a novel filtering method called *RDF Triple Filtering* (R3F) to address the problem of redundant intermediate results. R3F can reduce these unwanted results by filtering out irrelevant triples retrieved from the scan operators before they are passed to the join operators. We check the relevance of triples for a particular query using *incoming predicate path* information. Consider, for example, vertex $?v_3$ in Figure 1a, which has two path patterns: $\langle p_1, p_2 \rangle$ and $\langle p_3, p_2 \rangle$. These are called *incoming predicate paths* because they are composed of and represented by a sequence of predicates. In this example, the result vertices matching $?v_3$ must have these two incoming predicate paths. Using this necessary condition, R3F can filter out irrelevant triples, and consequently reduce redundant intermediate results.

In order to provide filter data for R3F, we design a new indexing structure called the *RDF Path index* (RP-index). The RP-index stores the precomputed incoming predicate path information in order to efficiently provide the filter data required for triple filtering. It consists of several vertex lists built for a set of predicate paths, each of which contain all vertices having the specified predicate path as their incoming path. The RP-index is a sort of path-based index, and appears very similar to previous path-based indices proposed for semi-structured data, such as DataGuide [13], 1-index [27], A(k)-index [21], D(k)-index [35], and M(k)-index [18]. Although these indices can be used for R3F, the RP-index has different goals, aiming to provide filter data efficiently rather than obtain query results from the index. Thus, it is specially designed to achieve this goal, and can also take different approaches to address the size problem, which is an important issue in several path-based indices. More specifically, we deal with the size problem of the RP-index using the discriminative fragment concept applied in gIndex [48]. We discuss the differences between the RP-index and other indices in more detail in Section 2.

We also propose a new relational operator called the *RDF Filter* (RFLT) that conducts triple filtering for its child operators using vertex lists from the RP-index. It is a very lightweight operator, designed to minimize the additional overhead to the original query processing caused by triple filtering. Execution plans using RFLT operators are generated by a cost-based query optimizer based on their costs and filtering effects. For this, we also elaborate on the cost measure and estimation method for the output cardinality of the RFLT operator.

We implement R3F on top of RDF-3X [31], the fastest RDF engine according to the published numbers (we discuss RDF-3X in Section 2). Many RDF stores including RDF-3X store triples as sorted to permit the efficient retrieval of matching triples and to allow efficient merge join operations [1, 31, 46]. For efficient triple filtering, R3F uses the manner in which retrieved triples are sorted in RDF-3X. In addition, RDF-3X already has several indices for efficient retrieval of matching triples. Whereas these indices aim to retrieve matching triples for a given triple pattern, the RP-index is designed to supply the filter data. The RP-index is a sort of

supplementary index for pruning irrelevant triples retrieved from the triple indices (or aggregated indices) using the incoming predicate path information. Hence, the RP-index and the indices in RDF-3X are in a complementary relationship. We focus on the graph pattern matching component of SPARQL query processing, especially the basic graph pattern [34]. However, we also discuss how to apply our approach to other types of queries.

Our main contributions are summarized as follows:

- We propose a novel triple filtering method called R3F to reduce the amount of redundant intermediate results using the incoming predicate path information of RDF data.
- For efficient and effective triple filtering, we design a path-based index called *RP-index*. Additionally, we deal with the size problem of the RP-index using the discriminative and frequent fragment concept from gIndex [48], and also consider maintenance issues.
- We propose the *RDF Filter* (RFLT) operator for conducting triple filtering, and integrate this into the cost-based query optimizer.
- We implement R3F on RDF-3X [31] and present comprehensive performance evaluation results using various large-scale RDF datasets.

This paper is an extended version of a previous paper [22]. In this research, we extend the previous work as follows: (1) We extend the RP-index to include reverse predicates, increasing the capability of triple filtering. (2) We deal with the size problem and maintenance issues of the RP-index that were not dealt with in the previous work. (3) We extend the RFLT operator to conduct triple filtering and merge joins at the same time, and also propose its cost function.

The remainder of the paper is organized as follows. Section 2 reviews related work. In this section, an overview of the target RDF-3X system is also presented. Section 3 gives some preliminary notation and discusses the data model related to our work. Section 4 describes the overall process of R3F and presents the design of the RP-index. Section 5 introduces the RFLT operator and discusses the generation of execution plans using this operator. Section 6 covers the building and incremental update method of the RP-index. Section 7 presents some experimental results from our approach, and Section 8 concludes the paper and discusses future work.

2 Related work

In this section, we review previous work on RDF stores, the handling of intermediate results in SPARQL query processing, and path-based and graph indices.

2.1 RDF stores

We can divide RDF stores into two categories, relation-based RDF stores and graph-based RDF stores, based on their query processing method. Relation-based RDF stores use the logical relational model to store RDF data and translate SPARQL queries into equivalent relational algebraic expressions [9]. On the other

hand, graph-based RDF stores process SPARQL queries using subgraph matching algorithms. They usually use graph indices to reduce the search space of subgraph matching algorithms.

Early relation-based RDF stores such as Jena [8] and Sesame [7] use relational databases as their underlying stores (currently, they also provide native RDF stores [33]). However, because relational database management systems (RDBMSs) are not optimized for processing RDF data, they have scalability problems for large-scale RDF data. SW-Store [1] partitions the triple table vertically according to the predicate value. By partitioning the triple table, SW-store can easily retrieve matching triples for triple patterns with predicate constants. However, SW-Store is not scalable for queries with predicate variables [39]. Hexastore [46] stores RDF triples in a set of vectors. Triples are indexed by six possible orderings of three columns so that they can be retrieved for any type of triple pattern. This method can also extend the possibility of using merge joins. BitMat [2] stores RDF data as a compressed bit-matrix structure. The authors present a pruning method using bit-matrices that does not generate intermediate results. RDF-3X [31] is another relation-based RDF store, that we discuss in more detail in Section 2.1.1. SWIM (Semantic Web Information Management) [20] proposes the scalable and extensible framework for RDF data that stores the semantic web data in a relational DBMS. The approximate query answering problem for RDF data has also been studied and experiments on relational RDF stores were conducted in [45].

Recently, a few *graph-based* RDF stores have also been proposed. In the GRIN index [44], an RDF graph is partitioned into several subgraphs. Those relevant to a query can then be chosen by the GRIN index. DOGMA [6] is a disk-based graph index used to retrieve the neighboring vertices of a specific vertex. The DOGMA index exploits distance information to restrict the search space. PIG [43] constructs an index that summarizes the structure of an RDF graph, and processes queries using the structure index. gStore [51] uses an approach similar to PIG. gStore reduces the search space by transforming an RDF graph and query graphs into signature graphs, and then matches the query signature graphs against the data signature graph.

In summary, relation-based RDF stores mainly use join operations, whereas graph-based RDF stores use graph exploration for the graph pattern matching. Using join operations, substructures can be joined in batch, and so relation-based RDF stores are more suitable for handling large-scale RDF data [41]. However, the graph indices used in graph-based RDF stores can effectively reduce the search space of the graph pattern matching algorithms, and can be used to reduce the number of redundant intermediate results. Our proposed R3F is designed for relation-based RDF stores, and also uses a kind of graph index, the RP-index. Therefore, R3F can be regarded as an attempt to hybridize the advantages of relation-based and graph-based approaches. To the best of our knowledge, there has been little effort to integrate the two approaches.

2.1.1 Overview of RDF-3X

RDF-3X [31] is an open source RDF engine and it is known as the fastest RDF engine according to the published numbers. In RDF-3X, Uniform Resource Identifiers (URIs) and literals are replaced by integer IDs using a mapping dictionary, and triples are stored using these IDs. Therefore, URIs and literals are treated in the same way in RDF-3X. RDF triples are stored in six clustered B+tree indices, built

for each of the six permutations of subject (S), predicate (P), and object (O): SPO, SOP, PSO, POS, OSP, and OPS. Each index stores triples in the leaf blocks as sorted by its ordering. Additionally, there also exist nine aggregated indices (SP, PS, SO, OS, PO, OP, S, P, O) that index partial triples and their occurrence counts.

By storing triples in six indices, RDF-3X can retrieve matching triples for any triple pattern in any ordering using range scans. For example, if a scan operator reads triples from the PSO index, the retrieved triples are ordered by (P, S, O). Furthermore, if the triple pattern assigned to a scan operator has a predicate constant, the retrieved triples are totally ordered by the S column.

RDF-3X uses two types of join operators: hash join and merge join. If both inputs of a join operator are ordered by columns corresponding to the join variable, RDF-3X uses the merge join; otherwise, the hash join is used. Let us consider the example in Figure 1b. $Scan_1$ and $Scan_2$ use the POS index, and the retrieved triples are totally ordered by the O column. The vertex corresponding to the O column is $?v_3$, which is also the join variable of $Join_1$. Therefore, $Join_1$ uses the merge join. However, the results of $Join_1$ are ordered by v_3 and the join variable of $Join_2$ is $?v_2$. Thus, $Join_2$ uses the hash join. $Join_3$ also uses the hash join because the results of $Join_2$ are not ordered.

RDF-3X alleviates the space overhead caused by redundancy (six triple indices and nine aggregated indices) by compressing the triples in the leaf blocks using a delta-based byte-level compression scheme. This compression scheme exploits the fact that it usually takes fewer bytes to encode the delta between triples than to store the triples directly. The delta between two triples is encoded with a header byte, which contains the size of three delta values, and three deltas between values in the triples (subject, predicate, and object). The delta between two values consumes between 0 bytes (unchanged) and 4 bytes (the ID of a URI or literal consumes four bytes), and therefore there are 125 size combinations for the delta between two triples. This delta size combination is stored in the header byte, with its most significant bit set to 1. If only the last value of the triple changes and the delta is less than 128, it is directly stored in the header byte (with its most significant bit set to 0), and so it can be encoded with only one byte. For a more detailed description, readers can refer to [31].

In addition, to reduce the overhead of index scans and the number of intermediate results, RDF-3X uses a kind of sideways information passing (SIP) technique called U-SIP. SIP refers to techniques that reduce the inputs of a join operator using information passed from another operator outside the normal execution flow (this is why they are called sideways information passing) [3, 4, 10]. The passed information usually contains domain information about the join variable so that inputs that will not be joined can be pruned in advance. U-SIP builds filters that provide information about the next triples to be read (called next information). The next information is the subject or object ID to be read next. RDF-3X uses this next information to skip the reading of unnecessary disk blocks. While scanning the leaf blocks sequentially, if the next block is considered to be unnecessary based on the next information, rather than continuing the sequential scan, it looks up the B+tree index from the root node and directly accesses the leaf blocks containing the next triples to be read. In this way, U-SIP can avoid reading unnecessary leaf blocks and reduce the number of redundant intermediate results.

2.2 Handling the intermediate results

In a traditional RDBMS, the redundant intermediate result problem is dealt with by finding the optimal join orderings for the queries [37]. Following this approach, several selectivity estimation techniques for SPARQL query processing have also been proposed [26, 40]. In RDF-3X, several specialized histograms for RDF are used [30–32]. They provide cardinality information for specific triple patterns and selectivities for specific patterns of joins.

The SIP techniques discussed in the previous section, including U-SIP, can also be considered as techniques for handling the intermediate results. However, SIP techniques are dynamic, runtime methods [3, 4, 10], whereas the join ordering technique is a static method determined in the query compile time.

These two previous approaches for handling the intermediate results have the limitation that they do not consider any graph structures in RDF data. Our R3F method exploits the graph-structural information, and can therefore be more effective for graph-structured RDF data than these approaches.

2.3 Path-based and graph indices

There exist numerous bodies of work in the literature proposing path-based indices for semi-structured data, e.g., DataGuide [13], 1-index [27], A(k)-index [21], D(k)-index [35], and M(k)-index [18] (cf. [14, 47] for detailed surveys). These indices summarize path information in graph-structured data, and provide a concise summary of the original graph that can be used for query processing in place of the original graph. Therefore, these indices focus on reducing the index size for efficient query processing, and avoid storing vertices several times in the index.

Although the RP-index can be considered reminiscent of these path-based indices, it aims to provide the filter data efficiently, not to obtain query results from the index. Hence, it incorporates a different structure than previous path-based indices: vertices can be stored several times, and they are stored as sorted and compressed to minimize the space and processing overheads of triple filtering. To prevent the indices from growing larger than the original graph, the path-based indices except DataGuide map a vertex to exactly one index node. Therefore, when using these indices, union operations are required to obtain vertices which are reached by a given path. In contrast, RP-index allows overlaps between vertex lists to be able to get filter data directly. To address the size problem of DataGuide, 1-index partitions vertices based on their B-bisimilarity. Intuitively, it stores vertices which have a same set of incoming paths into a index node. And to reduce the size of index further, A(k)-index indexes paths whose length are no longer than k using k-bisimilarity. D(k)-index and M(k)-index propose methods to apply k values adaptively. However, the RP-index applies a different approach to address the size problem. Because it provides filter data, it does not need to index all existing paths, and can index only effective paths for triple filtering selectively. Using this fact, we store only vertex lists having enough filtering power, based on the discriminative and frequent fragment concept used in gIndex [48]. Thus, the RP-index has a different structure from previous path-based indices and takes a different approach to handling the size problem.

Many graph indices have also been proposed for graph data. There are two problem formulations for graph indexing: the graph-transaction setting (many small graphs in a database) and the single-graph setting (a large single graph) [25]. The single-graph setting is more general because several graphs can be combined into a single graph, and the algorithms developed for the graph-transaction setting cannot be used for the single-graph setting [25]. Most graph indices have been proposed for the graph-transaction setting, and focus on reducing the number of tests conducted on the graph isomorphism, which is a very costly operation (e.g., GraphGrep [38], gIndex [48]). Hence, it is not trivial to apply these indices to an RDF graph, which is a single large graph. Recently, graph indices for large graphs were also proposed, such as SAGA [42], GraphQL [17], GADDI [49], and SPath [50]. Although these indices can be used in graph-based RDF stores, it is not trivial to apply these indices in relational-based RDF stores because they were designed in the context of graph-traversing algorithms.

3 Preliminaries

3.1 RDF and SPARQL

In this section, we present the core fragments of RDF and SPARQL that are relevant to our approach. We omit some features of RDF and SPARQL for simplicity. For example, we do not consider some features of RDF, such as blank nodes and the literal data type. For SPARQL, we focus on the basic graph patterns [34]. A basic graph pattern is a set of conjunctive triple patterns, which means its results should be matched to all triple patterns [34]. We assume that there is no join with predicate variables, because this join type is rarely used. However, with minor modifications, our approach can be applied to RDF data and SPARQL queries without these restrictions (we will discuss this issue in Section 4.2).

We assume the existence of three pairwise disjoint sets: a set of URIs U, a set of literals L, and a set of variables VAR. A variable symbol starts with ? to distinguish it from a URI. A triple $t(s, p, o) \in U \times U \times (U \cup L)$ (without variables) is called an RDF triple, and a triple $tp(s, p, o) \in (U \cup VAR) \times U \times (U \cup L \cup VAR)$ (triple with variables) is called a triple pattern. We treat literals in the same way as URIs, as in RDF-3X. That is, all URIs and literals are mapped to integer IDs using a dictionary mapping, and URIs and literals are treated in the same way.

The RDF database D is a set of RDF triples, and SPARQL query Q is a set of triple patterns. We denote the set of URIs that are used as predicates of triples in D as P_D. Formally, $P_D = \{p \mid p \in U \wedge \exists t(s, p, o) \in D\}$. Additionally, we denote as $D(p_i)$ the set of triples in D whose predicates are p_i. Namely, $D(p_i) = \{t(s, p, o) \mid t \in D \wedge p = p_i\}$.

We map RDF database D into a graph $G_D = (V_D, E_D, L_D)$, where V_D is a set of vertices corresponding to the subjects and objects of all triples in D, $E_D \subseteq V_D \times V_D$ is a set of directed edges that connect the subject and object vertices for triples in D, and $L_D : E_D \rightarrow P_D$ is an edge-label mapping such that, for all $t(s, p, o) \in D$, $L_D(s, o) = p$. SPARQL query Q is also mapped into graph $G_Q = (V_Q, E_Q, L_Q)$, where V_Q is a vertex set containing the subjects and objects of triple patterns in Q, E_Q is a set of directed edges that connect vertices corresponding to the subjects and objects of triple patterns in Q, and L_Q is an edge-label mapping such that, for all

$tp(s, p, o) \in Q$, $L_Q(s, o) = p$. Both G_D and G_Q are edge-labeled directed graphs. Figures 1a and 2 show a SPARQL query graph and an RDF graph, respectively. In these figures, we represent URIs and literals using simple notation such as v_n, p_n for readability.

For SPARQL query Q, the substitution θ is a mapping $V_Q \cap VAR \rightarrow U$. $\theta(G_Q)$ is a graph whose variables are substituted according to θ. The answer set for a SPARQL query is defined as follows.

Definition 1 (SPARQL Query Answer) The answer set for SPARQL query Q w.r.t RDF database D is $Ans(Q) = \{\theta \mid \theta(G_Q)$, which is isomorphic to a subgraph to $G_D\}$. For $v \in V_Q$, $Ans(Q, v)$ denotes the projection of $Ans(Q)$ over v, $Ans(Q, v) = \{\theta(v) \mid \theta \in Ans(Q)\}$, where $\theta(v)$ is the projection of mapping θ over v.

Example 1 (SPARQL Query Answer) For the RDF graph in Figure 2, the answer set of the SPARQL query in Figure 1a is $Ans(Q) = \{(?v_1 \rightarrow v_1, ?v_2 \rightarrow v_2, ?v_3 \rightarrow v_6, ?v_4 \rightarrow v_7, ?v_5 \rightarrow v_8), (?v_1 \rightarrow v_8, ?v_2 \rightarrow v_9, ?v_3 \rightarrow v_{12}, ?v_4 \rightarrow v_7, ?v_5 \rightarrow v_8), (?v_1 \rightarrow v_{10}, ?v_2 \rightarrow v_{11}, ?v_3 \rightarrow v_{15}, ?v_4 \rightarrow v_{13}, ?v_5 \rightarrow v_{14}), (?v_1 \rightarrow v_{11}, ?v_2 \rightarrow v_{14}, ?v_3 \rightarrow v_{15}, ?v_4 \rightarrow v_{13}, ?v_5 \rightarrow v_{14})\}$. Furthermore, the projection over $?v_3$ of $Ans(Q)$ is $Ans(Q, ?v_3) = \{v_6, v_{12}, v_{15}\}$.

3.2 Incoming predicate path

We define an RDF-specific path, called a *predicate path*, as follows.

Definition 2 (Predicate Path) A predicate path is a sequence of predicates. Given a predicate path *ppath*, the length of *ppath*, denoted as $|ppath|$, is the number of predicates in *ppath*.

We also define a set of incoming predicate paths for a vertex as follows.

Definition 3 (Incoming Predicate Path) Given a graph $G = (V, E, L)$, for $v \in V$, an incoming predicate path for v is a predicate path consisting of the predicates of the incoming path of v in G. We denote a set of incoming predicate paths of v as $InPPath(v)$. When the maximal path length $maxL$ is given, a variant of the notation, $InPPath(v, maxL)$, is used to denote a subset of $InPPath(v)$, such that $InPPath(v, maxL) = \{ppath \mid ppath \in InPPath(v) \wedge |ppath| \leq maxL\}$.

Note that the definition of the incoming predicate path can be applied to both RDF and query graphs.

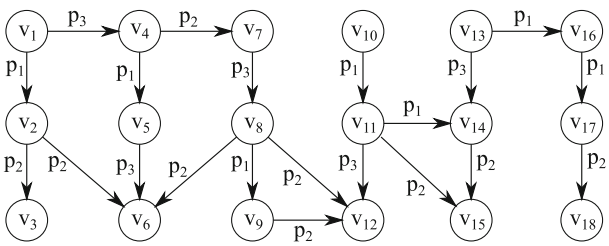


Figure 2 RDF graph

Example 2 (Incoming Predicate Path) For the RDF graph in Figure 2, the incoming path set of v_{12} with maximum length 3 is $InPPath(v_{12}, 3) = \{\langle p_2\rangle, \langle p_3\rangle, \langle p_1, p_2\rangle, \langle p_3, p_2\rangle, \langle p_1, p_3\rangle, \langle p_2, p_3, p_2\rangle, \langle p_3, p_1, p_2\rangle\}$. For the SPARQL query graph in Figure 1a, $InPPath(?v_3, 3) = \{\langle p_1, p_2\rangle, \langle p_3, p_2\rangle\}$.

3.3 Candidate vertex set

For $v \in V_Q$, the candidate vertex set for query vertex v is the set of vertices that could be results for v. Essentially, the candidate vertex set for v is a superset of the answer set $Ans(Q, v)$. The candidate vertex set can be defined in various ways, as long as it is a superset of the answer set. In this paper, we define the candidate vertex set using the incoming predicate path as follows.

Definition 4 (Candidate Vertex Set) Given the RDF database D, SPARQL query Q, and maximum length of the incoming predicate path $maxL$, the candidate vertex set for $v \in V_Q$ is $C_{InPPath}(v, maxL) = \{v_g \mid v_g \in V_D \wedge InPPath(v, maxL) \subseteq InPPath(v_g, maxL)\}$.

The following lemma ensures that the definition of $C_{InPPath}$ satisfies the previous condition of the candidate vertex set (i.e., it should be a superset of the answer set).

Lemma 1 *Given the RDF database D and SPARQL query Q,* $\forall v \in V_Q$, $Ans(Q, v) \subseteq C_{InPPath}(v, maxL)$.

Proof We prove that if vertex $v_D \in G_D$ is in $Ans(Q, v)$, v_D must have all incoming predicate paths of v. That is, $\forall v_D \in Ans(Q, v)$, $InPPath(v, maxL) \subseteq InPPath(v_D, maxL)$. If $v_D \in Ans(Q, v)$, there exists a substitution $\theta \in Ans(Q)$ that ensures graph $\theta(G_Q)$ is isomorphic to a subgraph to G_D and $\theta(v) = v_D$. From the definition of a subgraph isomorphism, if there exists an incoming path of v, $\langle e_1, \ldots, e_n\rangle$ $(n \leq maxL)$ in G_Q, there must exist a matching incoming path of v_D $\langle e'_1, \ldots, e'_n\rangle$ in $\theta(G_Q)$, such that $\forall i, 0 \leq i \leq n, l(e_i) = l(e'_i)$, where e_i is an edge and $l(e_i)$ is the label of e_i. Therefore, $\forall v_D \in Ans(Q, v), InPPath(v, maxL) \subseteq InPPath(v_D, maxL)$; that is, all $v_D \in Ans(Q, v)$ contain all incoming predicate paths of v, and $Ans(Q, v) \subseteq C_{InPPath}(v, maxL)$. □

Example 3 (Candidate Vertex Set) The candidate vertex for $?v_3$ in Figure 1a should have two incoming predicate paths, $\langle p_1, p_2\rangle$ and $\langle p_3, p_2\rangle$. For the RDF graph in Figure 2, there are three vertices that have these incoming predicate paths, so $C_{InPPath}(?v_3, 2) = \{v_6, v_{12}, v_{15}\}$. We can see that $Ans(Q, ?v_3) = \{v_6, v_{12}, v_{15}\} \subset C_{InPPath}(?v_3, 2)$ (i.e., satisfying the condition for the candidate vertex set).

4 R3F and the RP-index

In this section, we present an overview of R3F and discuss the design of the RP-index. We present a logical description of the RP-index in Section 4.2, and discuss its physical implementation in Section 4.2.1.

4.1 Overall process of R3F

Our goal is to filter out triples that are irrelevant to the query from among those retrieved from the scan operators. To decide the relevance of a triple for a given query, we use the definition of the candidate vertex set, $C_{InPPath}$. Suppose that $?v_S$ and $?v_O$ are the subject and the object, respectively, of a triple pattern in the query. The triples retrieved for this triple pattern are checked to see if their subjects or objects exist in $C_{InPPath}(?v_S, maxL)$ or $C_{InPPath}(?v_O, maxL)$, respectively. If either condition is not true, this triple is irrelevant, and so it can be filtered out safely.

To implement this type of triple filtering, we design the RP-index and RFLT operator. The RP-index is designed to provide $C_{InPPath}$ efficiently, and is presented in Section 4.2. RFLT operators conduct triple filtering for their child scan operators. In order to apply triple filtering, the query optimizer analyzes the query graph and adds appropriate RFLT operators to the execution plan based on the filtering effects, costs, and output cardinalities of the RFLT operators. We will discuss the *RFLT* operator and the query optimization method in Section 5.

4.2 RP-index definition

The RP-index is an index structure used to obtain $C_{InPPath}(v, maxL)$ efficiently. It consists of a set of vertex lists for predicate paths existing in the RDF database D. The vertex list of predicate path *ppath* is defined as follows.

Definition 5 (Vertex List) Given the RDF database D, the vertex list for the predicate path *ppath* is a set of vertices that have *ppath* as their incoming predicate paths, i.e., $Vlist(ppath) = \{v \in V_D \mid ppath \in InPPath(v)\}$.

The RP-index for D is defined as follows.

Definition 6 (RP-index) Given the RDF database D, the RP-index of D with maximum length $maxL$, denoted by $RP\text{-}index(D, maxL)$, is a set of pairs $\langle ppath, Vlist(ppath)\rangle$, where *ppath* is a predicate path in D whose length is less than or equal to $maxL$.

Example 4 (RP-index) Figure 3 shows the Vlists in $RP\text{-}index(D, 3)$ for D in Figure 2 with $maxL = 3$. There are 15 Vlists in $RP\text{-}index(D, 3)$.

We introduce $maxL$ to limit the size of the RP-index. As $maxL$ increases, the number of predicate paths in the RP-index increases and, as a result, the quality of the triple filtering can be improved. However, the space overhead of the RP-index also increases. In other words, there is a tradeoff between the quality of the triple filtering and the space overhead of the RP-index. This tradeoff can be adjusted by $maxL$ (we also use another method to address the size problem of the RP-index, discussed in Section 4.3).

A Vlist can be used to obtain candidate vertex sets. Given $RP\text{-}index(D, maxL)$ and query Q, we can obtain $C_{InPPath}(v, maxL)$ for $v \in V_Q$ by computing the intersection of $Vlist(ppath)$ for all $ppath \in InPPath(v, maxL)$.

Figure 3 Vlists in RP-index(D, 3)

Length	Predicate Path	Vlist
1	$<p_1>$	$v_2, v_5, v_9, v_{11}, v_{14}, v_{16}, v_{17}$
	$<p_2>$	$v_3, v_6, v_7, v_{12}, v_{15}, v_{18}$
	$<p_3>$	$v_4, v_6, v_8, v_{12}, v_{14}$
2	$<p_1, p_1>$	v_{14}, v_{17}
	$<p_1, p_2>$	$v_3, v_6, v_{12}, v_{15}, v_{18}$
	$<p_1, p_3>$	v_6, v_{12}
	$<p_2, p_3>$	v_8
	$<p_3, p_1>$	v_5, v_9
	$<p_3, p_2>$	v_6, v_7, v_{12}, v_{15}
3	$<p_1, p_1, p_2>$	v_{15}, v_{18}
	$<p_2, p_3, p_1>$	v_9
	$<p_2, p_3, p_2>$	v_6, v_{12}
	$<p_3, p_1, p_2>$	v_{12}
	$<p_3, p_1, p_3>$	v_6
	$<p_3, p_2, p_3>$	v_8

4.2.1 Physical structure of the RP-index

The vertices in a Vlist are represented by their integer IDs (4 bytes), which are produced by the dictionary mapping used in RDF-3X (Section 3). Vlists are sorted and stored on disk by vertex IDs, enabling the Vlist to be read from disk in its sorted form. The reason to store Vlists as sorted is to obtain $C_{InPPath}$ by simply merging the relevant Vlists (recall that $C_{InPPath}$ can be obtained by the intersection of the Vlists). Another benefit of sorting is that sorted Vlists can be compressed by the delta-based byte-level compression scheme, similar to the compressed triples in RDF-3X [31] (see Section 2.1.1). The delta between two vertex IDs is encoded with 1 header byte and the minimum number of bytes for the delta (1–4 bytes). If the delta is smaller than 128, it is directly stored 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 to 1

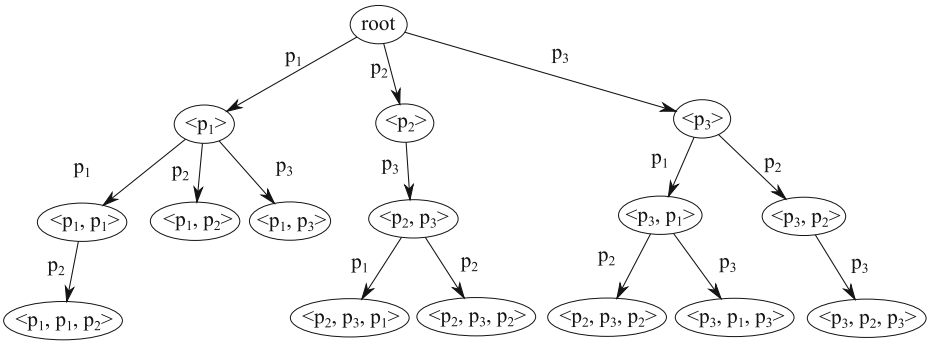


Figure 4 A trie for predicate paths

to indicate the delta is not small. This compression scheme alleviates the overall size overhead of Vlists and reduces the disk I/O overhead in reading the Vlists.

We organize the predicate paths of the RP-index 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 predicate path. Figure 4 shows the trie for RP-index(D, 3) in Figure 3. The trie provides compact storage for the predicate paths, because duplicated parts of predicate paths can be shared. In addition, it provides an efficient way to access the Vlist for a given predicate path. We can find the disk location of the Vlist for a predicate path by traversing the trie using the predicate path. The number of nodes in the trie is equal to the number of predicate paths in the RP-index. For real-life data sets and a small $maxL$ value, the trie is of relatively small size and can reside in the main memory.

4.3 Discriminative and frequent predicate paths

Due to their exponential number, it would be infeasible to generate Vlists for all predicate paths in an RDF database, even if we restricted their maximum length. Hence, we should choose a subset of Vlists to be stored in the RP-index. To establish criteria for choosing Vlists, we define the *discriminative and frequent predicate path*, which is adapted from the discriminative and frequent fragment concept in gIndex [48].

The first criterion is to store only Vlists with enough filtering power. If $Vlist_i \supset Vlist_j$, we can use $Vlist_i$ in place of $Vlist_j$, because $Vlist_i$ has all of the vertices in $Vlist_j$. Therefore, we can store only $Vlist_i$ and remove $Vlist_j$ from the RP-index. However, this replacement can degrade the filtering power, because the replacement filter is prone to produce more false positives than the replaced filter. Therefore, it is important to choose predicate paths that do not significantly degrade the filtering power. A discriminative predicate path is one whose Vlist cannot be replaced by another Vlist without degenerating the filtering power to an unacceptable degree. We define the discriminative predicate path as follows.

Definition 7 (Discriminative Predicate Path) Given a discriminative ratio γ ($0 < \gamma \leq 1$), predicate path $ppath$ is discriminative iff, $\forall ppath_{suf}$ that are proper suffixes of $ppath$, $|Vlist(ppath)| < \gamma \times |Vlist(ppath_{suf})|$.

In other words, predicate path $ppath$ is discriminative if $Vlist(ppath)$ is smaller (according to γ) than the Vlist for the longest proper suffix predicate path of $ppath$. Note that if $|ppath| = 1$, $ppath$ is discriminative because it does not have any proper suffix predicate path.

Example 5 (Discriminative Predicate Path) For the RP-index in Figure 3, suppose that the discriminative ratio is $\gamma = 0.7$. Then, $\langle p_1, p_2 \rangle$ is not discriminative because $|Vlist(\langle p_1, p_2 \rangle)| = 5$, $|Vlist(\langle p_2 \rangle)| = 6$, and $|Vlist(\langle p_1, p_2 \rangle)| / |Vlist(\langle p_2 \rangle)| > 0.7$.

The second criterion is to store only frequent predicate paths. A predicate path is frequent iff its Vlist has more vertices than the minimum threshold defined by the user. Infrequent predicate paths are not likely to be useful, because they are rare in RDF graphs and would not be queried frequently. Therefore, removing them

does not degrade the overall performance for most queries. Additionally, because there are a large number of infrequent predicate paths, removing them can reduce the size of the RP-index significantly. Since the number of paths increases with path length, we use a size-increasing function to provide the threshold value for identifying frequent predicate paths. In this way, we can reduce the overall index size. We define a frequent predicate path as follows.

Definition 8 (Frequent Predicate Path) Given a size-increasing function $\psi(l)$, predicate path *ppath* is frequent if and only if $|Vlist(ppath)| \geq \psi(|ppath|)$.

4.4 Reverse predicate

Because R3F utilizes the incoming predicate path information, triple filtering cannot be applied to a vertex having no incoming predicate path. For example, vertex $?v_3$ in Figure 5 has no incoming predicate path, and so triple filtering cannot be applied to $?v_3$, even though it has four edges (ignoring the dashed edges). In order to increase the capability of triple filtering, we extend an RDF database and SPARQL query as follows to consider the reverse predicates.

Definition 9 (Extended RDF Database and Query) For RDF database D, $\forall t(s, p, o) \in D$, we assume the existence of a virtual triple $t'(o, p^R, s)$. For SPARQL query Q, $\forall t(s, p, o) \in Q \wedge p \in P_D$, we assume the existence of a virtual triple $t'(o, p^R, s)$. We call p^R the *reverse predicate* of p.

In order to use reverse predicates, we build the RP-index on the extended RDF database and generate the incoming predicate paths using the extended SPARQL query. Note that the virtual triples do not need to exist in the RDF store. Instead, we only suppose that they exist in the RDF store by reversing the subject and the object of a triple when building the RP-index.

Although the introduction of reverse predicates can increase the applicability of triple filtering, it can also result in many redundant predicate paths. We call a predicate path redundant if its Vlist is always the same as some Vlists of its suffix predicate paths. For example, $Vlist(\langle p_1, p_2, p_3\rangle)$ is always the same as $Vlist(\langle p_1^R, p_1, p_2, p_3\rangle)$. This is because they have a suffix relationship

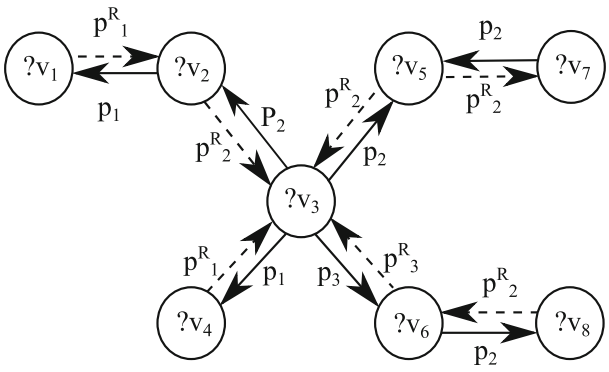


Figure 5 Extended SPARQL query

($Vlist(\langle p_1^R, p_1, p_2, p_3\rangle) \subset Vlist(\langle p_1, p_2, p_3\rangle)$), and vertices that have $\langle p_1, p_2, p_3\rangle$ as their incoming predicate paths must also have $\langle p_1^R, p_1, p_2, p_3\rangle$ as their incoming predicate paths (i.e., $Vlist(\langle p_1^R, p_1, p_2, p_3\rangle) \supset Vlist(\langle p_1, p_2, p_3\rangle)$). In general, $Vlist(ppath)$ is the same as the Vlists for predicate paths having $ppath$ as their suffix, and their remaining parts are cyclic paths using the reverse predicates, as in the previous example (we omit a formal definition and proof for simplicity). These redundant predicate paths are due to the cycles caused by reverse predicates

Besides the redundant predicate paths, reverse predicates also cause too many non-redundant incoming predicate paths. For example, $?v_8$ in Figure 5 has incoming predicate path $\langle p_3, p_2\rangle$. Also, $\langle p_3, p_2, p_2^R, p_2\rangle$ and $\langle p_3, p_2, p_2^R, p_2, \cdots, p_2^R, p_2\rangle$ are incoming predicate paths of $?v_8$ (note that these predicate paths are not redundant, because they do not have $\langle p_3, p_2\rangle$ as their suffix). Although they are not redundant and may be helpful, these incoming predicate paths are not likely to be used in normal queries. As a result, in order to prevent the formation of redundant predicate paths and the generation of too many incoming predicate paths, we do not generate predicate paths containing the pattern p_i, p_i^R.

In Figure 5 the dashed edges denote those with reverse predicates. Considering the reverse predicates, $InPPath(?v_3) = \{\langle p_2^R\rangle, \langle p_1^R, p_2^R\rangle, \langle p_1^R\rangle, \langle p_3^R\rangle, \langle p_2^R, p_3^R\rangle\}$.

4.5 Handling other types of queries

We have considered queries consisting of only the basic graph patterns without predicate variables (see Section 3). As already mentioned, R3F can also be applied to other types of queries with minor modifications. Queries with predicate variables can be handled as follows. The first and easiest way is to simply exclude edges with predicate variables from considerations when making the incoming predicate paths for the triple filtering. That is, we do not generate the incoming predicate paths with predicate variables. Let us consider the SPARQL query in Figure 6. This query has one edge with a predicate variable $?p$. If we exclude this edge when

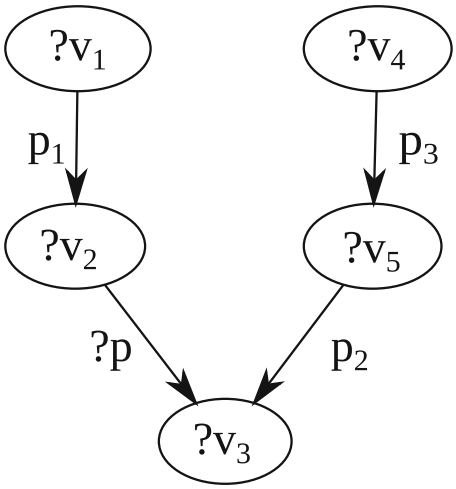


Figure 6 A SPARQL query with a predicate variable

generating the incoming predicate paths, then $InPPath(?v_3) = \{\langle p_3, p_2\rangle, \langle p_2\rangle\}$. Note that because we exclude the predicate variable, we have fewer incoming predicate paths. As we can see from this example, the first approach is simple, but it can also limit the capability of triple filtering. The second way is to consider the variable predicate as a special predicate, say p_v, whose triples are the entire set of triples in the database. Hence, when building the RP-index, the predicate paths containing this variable predicate also need to be indexed. When generating incoming predicate paths for the query graph, the triple patterns with predicate variables are considered as edges with the label p_v. For example, if we use the edge with the predicate variable, then $InPPath(?v_3) = \{\langle p_1, p_v\rangle, \langle p_v\rangle, \langle p_3, p_2\rangle, \langle p_2\rangle\}$. The set $Vlist(\langle p_1, p_v\rangle)$ is a set of vertices that have 2-length incoming predicate paths and where the predicate of the first edge of the path is p_1.

Queries with optional or union patterns can also be handled in a similar way. We can apply R3F to these queries by generating incoming predicate paths for the fragments of query graphs that consist of only the basic graph patterns. We can then apply triple filtering to these queries.

4.6 Determining the RP-index parameters

Until now, we have only discussed the design of the RP-index. In this section, we discuss its tuning issues. The RP-index has three tuning parameters: the maximum path length $maxL$, the discriminative ratio γ, and the minimum frequency function $\psi(l)$. These parameters affect the size and performance of the RP-index. It is important to make the RP-index as small as possible while maintaining its filtering power. The size of the RP-index is highly dependent on $maxL$, as the number of path patterns grows exponentially with the pattern length. However, for most cases, a small $maxL$ is sufficient because long paths are not common in real-world SPARQL queries. We study the effects of $maxL$ empirically in Section 7. From our experience, $maxL = 3$ is sufficient in most cases.

Although we use small $maxL$, it is still possible for the RP-index to grow prohibitively large. This is particularly likely to occur when there are a large number of predicates as in the case of the DBSPB dataset used in the experiments in Section 7. In this case, the number of possible predicate paths becomes abundant even for small $maxL$, because of the large number of predicates. In addition, there might be some cases in which queries with long paths are used and we need to index long path patterns by using large $maxL$. However, the size problem of the RP-index with large $maxL$ can be controlled by adjusting γ and $\psi(l)$. The effects of these two parameters have already been discussed, in Section 4.3. They can reduce the size of the RP-index; however, they can also degrade its performance by removing some necessary predicate paths. Hence, these parameters should be tuned carefully by considering the size and performance of the RP-index.

When the RP-index does not have some necessary predicate paths that users can identify, it is possible to add such paths to the RP-index based on user decisions. That is, rather than adjusting the parameters, users can indicate some necessary predicate paths to be indexed. However, this requires previous knowledge of the query workload. In most cases, using γ and $\psi(l)$, the size of the RP-index can be effectively controlled while retaining its filtering power. We see the effects of the parameters in the experimental results (Section 7.2.2).

5 Processing triple filtering

In this section, we describe how the triple filtering is processed. First, we introduce the RFLT operator, and then explain how to generate an execution plan using RFLT operators.

5.1 RFLT operator

The RFLT operator is a relational operator that conducts triple filtering for its child scan operators. It exploits the sorted property of the retrieved triples to efficiently process the triple filtering. Recall that the output triples of a scan operator in RDF-3X are sorted by the S or O column, depending on which index the scan operator reads. We define the *sortkey* for an operator as follows.

Definition 10 (Sortkey) The *sortkey column* of an operator is defined as the column by which the results of the operator are sorted. We use the term *sortkey vertex* to indicate the vertex in a query graph corresponding to the sortkey column. We also use *OP.sortkey* interchangeably to denote the sortkey column or the sortkey vertex of operator *OP*, depending on the context.

Example 6 (Sortkey) $Scan_1$ in Figure 1b uses the POS index and its triple pattern has the predicate constant p_2. Therefore, the result of $Scan_1$ is totally ordered by the O column. The sortkey column and the sortkey vertex of $Scan_1$ is the O column and $?v_3$, respectively. In the same way, $Scan_2.sortkey = ?v_3$, $Scan_3.sortkey = ?v_2$, and $Scan_4.sortkey = ?v_5$.

Basically, the RFLT operator conducts triple filtering for its child scan operator using their sortkey vertices. The query optimizer indicates to the RFLT operator which predicate paths it should use for triple filtering as follows. The RFLT operator for $Scan_i$ is assigned only predicate paths in $InPPath(Scan_i.sortkey, maxL)$, i.e., the incoming predicate paths of the sortkey vertex of $Scan_i$. The RFLT operator will compute the intersection of Vlists for all assigned predicate paths (this will be a superset of $C_{InPPath}(Scan_i.sortkey, maxL)$) to obtain the filter data for triple filtering. The input triples are then checked to determine whether the values of the sortkey column are included in the intersection (i.e., the filter data). This triple filtering can be processed by simply merging the assigned Vlists and the input triples, because they are all sorted by the sortkey column.

An RFLT operator can perform triple filtering for multiple scan operators as long as their sortkey vertices are the same. Note that the filter data for scan operators with the same sortkey vertex is also the same, because they will be assigned the same set of Vlists. Thus, if we make several RFLT operators for these scan operators, which conduct triple filtering separately using the same filter data, this causes redundant processing of triple filtering. To avoid this, we design the RFLT operator to process several child scan operators. Additionally, because the child scan operators share the sortkey vertex, their output triples should be joined for their sortkey columns, which 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 at the same time.

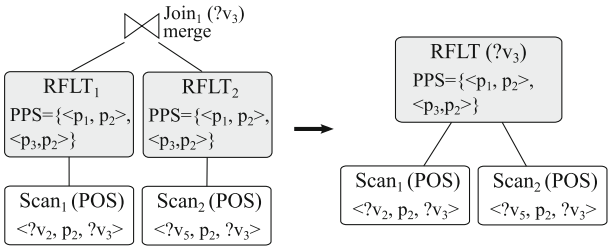


Figure 7 Execution plan using RFLT

Figure 7 shows part of the execution plan using RFLT operators for the query in Figure 1a. The predicate path set (PPS) in the RFLT operator is assigned by the query optimizer. In the plan on the left, there are two RFLT operators with the same PPS and one merge join operator. The sortkey vertices of $Scan_1$ and $Scan_2$, and the join variable of the merge join operator, are all $?v_3$. Therefore, these three operators can be combined into one RFLT operator, as in the plan on the right.

Figure 8 illustrates the filtering process of the RFLT operator. The intersection of two Vlists forms $C_{InPPath}(?v_3, maxL)$, and this is used as the filter data. The outputs of $Scan_1$ and $Scan_2$ are filtered using these Vlists, and the filtered triples are also joined by the operator.

RFLT only performs the merge process for its inputs (Vlists and input triples). Therefore, its cost is linear with respect to its input size, as follows.

$$\text{I/O cost: } O\left(\sum_{p \in PPS} \|Vlist(p)\|\right) \tag{1}$$

$$\text{CPU cost: } O\left(\sum_{scan \in ChildOP} |scan| + \sum_{p \in PPS} |Vlist(p)|\right) \tag{2}$$

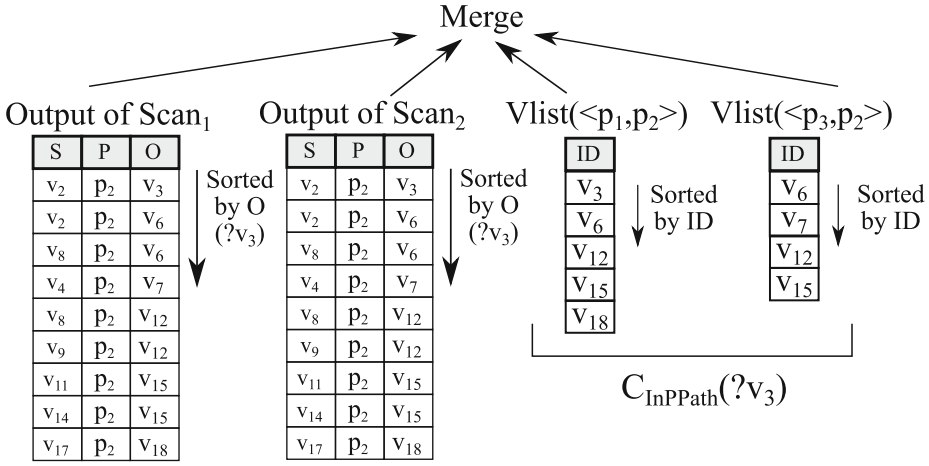


Figure 8 RFLT operator

where $\|vlist\|$ is the number of blocks of *vlist*, PPS is a set of predicate paths assigned for the RFLT operator, ChildOP is a set of child scan operators, and $|scan|$ is the cardinality of the *scan* operator. The Vlists are usually much smaller than the input triples. Therefore, the triple filtering process incurs little overhead, and the RFLT operator is very efficient and lightweight. In Section 7.2.1, we compare the size of Vlist and the input triples.

5.1.1 Implementation of the RFLT operator

We have implemented our RFLT operator in RDF-3X. RDF-3X adapts the iterator model of the query execution [15], and the operators in RDF-3X have a common interface with the first and next functions. first initializes the operator and returns the first tuple, and next returns the next tuples. RFLT operator also has been implemented as an iterator like other operators in RDF-3X so that it can be integrated with its query plans. When the first function of the RFLT operator is called, it performs some initializations for the triple filtering and returns the first tuple which passes the triple filtering. And then it returns the resulting tuples when its next function is called.

The results of the RFLT operator are the joined results of child input operators that pass the triple filtering. In order to conduct triple filtering, the RFLT operator reads Vlists from disk and gets the input triples from child operators by calling their next function. It generates results by performing the N-way merge joins for the assigned Vlists and input triples of the child operators, as discussed in the previous section. Note that the RFLT operator could generate the results and conduct the triple filtering simultaneously by performing only the N-way merge joins.

5.1.2 RFLT operator and U-SIP

RDF-3X exploits a type of SIP technique called *U-SIP* (see Section 2.1.1). In U-SIP, a scan operator can skip the reading of irrelevant blocks by utilizing the next information provided by other scan operators. With R3F, the filter data $C_{InPPath}$ can be used as another source for the U-SIP next information.

Let us look at the example in Figure 9. This figure illustrates the POS index for $Scan_1$ to read, and the filter data of the RFLT operator $C_{InPPath}(?v_3, maxL)$. The POS index is a clustered B+tree index in which triples are stored in its leaf blocks as sorted by the POS ordering. In this figure, the boxes represent leaf blocks of the index, and we represent the interval of the object values of the triples stored in each block. In this example, from $C_{InPPath}(?v_3, maxL)$, the scan operator scanning the POS index can determine that there is no need to read blocks whose objects are between v_7 and v_{11}, because the triples whose objects are in the interval would be filtered out in the RFLT operator. Therefore, it can skip two blocks whose objects are less than v_{12} by performing the look-up operation for the index. In this manner, R3F can provide the next information for scan operators. Consequently, R3F and U-SIP can utilize synergy effects.

5.2 Generating an execution plan with RFLT operators

Many RDF stores, including RDF-3X, use a cost-based query optimizer to find optimal (or near-optimal) plans for SPARQL queries [31]. In order to make a query

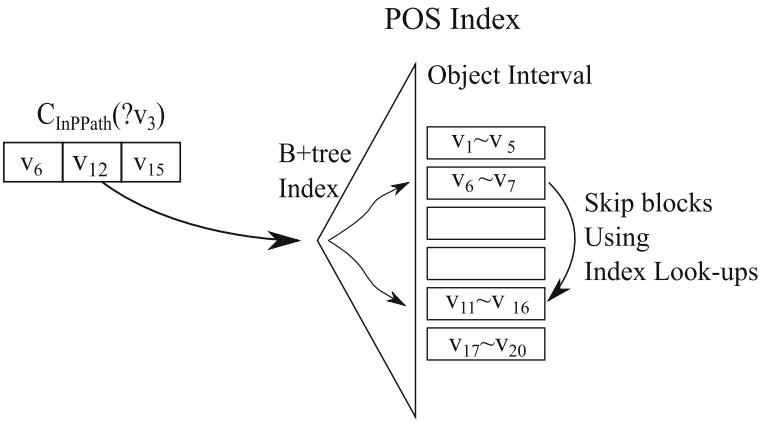


Figure 9 RFLT operator and U-SIP

optimizer that considers triple filtering, we need to provide the query optimizer with (1) the cost function of the RFLT operator, which was given in the previous section, and (2) the estimated cardinalities of the RFLT operators. In this section, we extend the query compiler of RDF-3X to consider RFLT operators. We first discuss the estimation method for the output cardinalities of RFLT operators, and then consider how to extend the query compiler to generate a plan using RFLT operators.

To begin, we assume that the following statistics are available: (1) the cardinalities of scan operators (the number of triples matching to 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. In addition, we form another statistic similar to the characteristics set [30]. We define the characteristics set for $v \in G_D$, $S_C(v)$, as the set of incoming predicates of v, including reverse predicates. Formally, $S_C(v) = \{p \mid \exists s : t(s, p, v) \in D\}$. For example, for v_{14} in Figure 2, $S_C(v_{14}) = \{p_1, p_3, p_2^R\}$. The number of vertices which have the characteristics set S is called the occurrence count [30] and is denoted as $count(S)$. We store the occurrence counts of all characteristics sets in D. The size of this information is minuscule compared to the database size [30].

5.2.1 Filtering effect of Vlists

We define the filtering effect of Vlist V for $Scan_i$, $E(Scan_i, V)$, as the fraction of the remaining values of the sortkey column after filtering. Let us denote the sortkey column of $Scan_i$ and the set of its distinct values as K, interchangeably. Then, $E(Scan_i, V)$ can be represented as follows:

$$E(Scan_i, V) = |V \cap K|/|K|. \tag{3}$$

We can estimate this value using the statistics of Vlists. First, we can obtain $|K|$ as follows. Let us assume that the predicate of the triple pattern of $Scan_i$ is p. If the sortkey of $Scan_i$ is the O column, $|K| = Vlist(\langle p \rangle)$, and if the sortkey of $Scan_i$ is the S column, $|K| = Vlist(\langle p^R \rangle)$. To simplify the notation, we use p_{scan}, which is defined depending on the sortkey column S as follows: if K is the O column, $p_{scan} = p$; if K is the S column, $p_{scan} = p^R$. Then, we can represent $|K| = |Vlist(\langle p_{scan} \rangle)|$.

The numerator of (3) can also be estimated using Vlists. Figure 10 shows the relationship between V and K. We denote the last predicate of the predicate path of V as p_v. If $p_v = p_{scan}$, $|V \cap K|$ can be easily computed as $|V|$ because $\langle p_{scan} \rangle$ is the suffix of the predicate path of V, and therefore $V \subseteq K$.

Otherwise ($p_v \neq p_{scan}$), we should estimate the intersection in other ways because $V \not\subseteq K$. The filtering effect of V against $Vlist(\langle p_v \rangle)$ can be computed as $|V|/|Vlist(\langle p_v \rangle)|$ (i.e. V would filter the values in $Vlist(\langle p_v \rangle)$ as the ratio of $|V|/|Vlist(\langle p_v \rangle)|$). We can also assume that V filters the values in $Vlist(\langle p_v \rangle) \cap Vlist(\langle p_{scan} \rangle)$ with the same filtering effect, because it is contained in $Vlist(\langle p_v \rangle)$. Then, we can estimate that $|V \cap K| = |V|/|Vlist(\langle p_v \rangle)| \times |Vlist(\langle p_v \rangle) \cap Vlist(\langle p_{scan} \rangle)|$. $|Vlist(\langle p_v \rangle) \cap Vlist(\langle p_{scan} \rangle)|$ is the number of vertices which have both p_v and p_{scan} as their incoming predicates, and it can be obtained from the characteristics set, $count(\{p_v, p_{scan}\})$.

5.2.2 Cardinality of the RFLT operator

If an RFLT operator has one child operator, it conducts only triple filtering. Let us denote the intersection of all assigned Vlists for an RFLT operator as $C = \bigcap_{ppath \in PPS} Vlist(ppath)$. In this case, if we assume that the values of the sortkey column of the child scan operator are distributed uniformly, the cardinality of the RFLT operator can be estimated as follows.

$$|RFLT| = |Scan_i| \times |C \cap K|/|K| \tag{4}$$

where K is the set of values of the sortkey column of $Scan_i$. To compute this value, we should be able to estimate the set of intersections, $C \cap K$. Although there are a few techniques [24] for estimating this set, they require some additional operations, such as sampling. In this case, we take a rather simple approach by using the upper bound of $|RFLT|$ as the estimated value. This means that we conservatively underestimate the effect of triple filtering. The upper bound can be estimated as $|C \cap K| \leq \min\left(\min_{path \in PPS} |Vlist(ppath)|, |K|\right)$, and we use this value for the estimated output cardinality of an RFLT operator.

If an RFLT operator has multiple child operators, we should be able to estimate the join size for the filtered triples. If we can estimate the number of joined values of

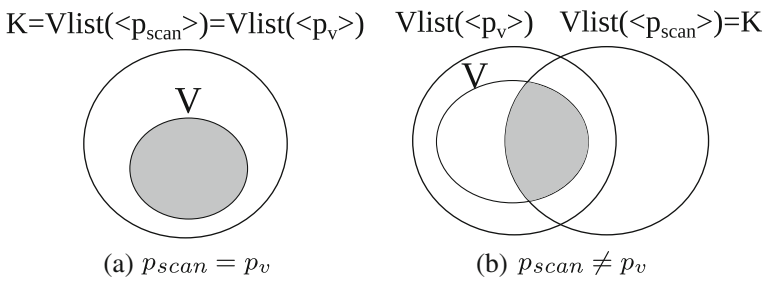


Figure 10 Filtering effect

the filtered triples, and assume that the values are distributed uniformly, the output cardinality can be estimated as follows:

$$|RFLT| = |J| \times \prod_{Scan_i \in ChildOP} |Scan_i|/|K_i| \quad (5)$$

where J is the set of joined values, and K_i is the set of sortkey column values of $Scan_i$.

J can be represented as $J = \left(\bigcap_{p \in P_s} Vlist(\langle p \rangle)\right) \bigcap \left(\bigcap_{ppath \in PPS} Vlist(ppath)\right)$, where P_s is a set of predicates of the child scan operators. Here, we again take the upper bound of $|J|$. We can easily obtain $|\bigcap_{p \in P_s} Vlist(\langle p \rangle)|$ from the characteristics set $U_1 = count(P_s)$. Also, we define $U_2 = \min_{ppath \in PPS} |Vlist(p)|$. Then, $|J| \leq \min(U_1, U_2)$.

In brief, we estimate the output cardinality of an RFLT operator 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 (the intersection size of the values of the sortkey column and Vlists). We find the accuracy of our estimation in Section 7.2.3.

Our method is very similar to the Characteristic Set [30], which was proposed to estimate the cardinalities of star-join queries. However, our method does not aim to replace the Characteristic Set, but to reflect the filtering effect in the cardinality estimation. We expect that exploiting the Characteristic Set with our estimation method would improve the estimation accuracy. Therefore, our method and the Characteristic Set have a complementary relationship.

5.2.3 Generating an execution plan

The query optimization of RDF-3X is based on the bottom-up dynamic-programming (DP) framework [31]. There are two ways to make plans using RFLT operators. The first is to add RFLT operators to plans generated from normal query optimization. This method is simple, but has the limitation that the plan cannot reflect the changed cardinalities due to triple filtering. Hence, we integrate RFLT operators into DP operator placement.

Before we discuss the addition method of RFLT operators, we briefly present the DP query optimization framework, shown in Algorithm 1. The input of the algorithm is a SPARQL query Q having n triple patterns ($tp_0 \cdots tp_{n-1}$), and it returns the cheapest plan for Q (line 19). The query compiler maintains the DP table (denoted as dpTable in Algorithm 1), in which the optimal plans for the subproblems of the query are stored. At first, the optimizer seeds its DP table with scan operators for the triple patterns as solutions of the 1-size subproblems (lines 1–3). The buildScan function makes scan operators for the input triple patterns. Larger plans are then created by joining two plans from smaller problems (lines 10–15), and these are added to the entries in dpTable. The buildJoin function makes join operators for two input plans. The added plans are maintained as follows. Each entry in dpTable keeps only the cheapest plans for its subproblem. However, there can be multiple plans in an entry of the DP table if there are several plans with different interesting orders (the order of output tuples). Basically, a plan in an entry is dominated and replaced by cheaper plans. However, more expensive plans with different interesting orders can be used to make final plans with lower overall costs. Hence, plans with different interesting

Algorithm 1 Dynamic-Programming based Query Optimization

procedure DPsize ($Q = \{tp_0, \ldots, tp_{n-1}\}$)

```
1:  for each tp_i ∈ Q do
2:    dpTable[{tp_i}]=buildScan(tp_i);
3:  end for
4:  for 1 ≤ i ≤ n do
5:    for 1 ≤ j < i do
6:      for each S_1 ⊂ Q : |S_1| = i − j, S_2 ⊂ Q : |S_2| = j do
7:        if S_1 ∩ S_2 ≠ ∅ or
            S_1 and S_2 cannot be joined then
8:          continue;
9:        end if
10:       for each p_1 ∈ dpTable[S_1] do
11:         for each p_2 ∈ dpTable[S_2] do
12:           P ← buildJoin(p_1, p_2);
13:           addPlan(dpTable[S_1 ∪ S_2], P);
14:         end for
15:       end for
16:     end for
17:   end for
18: end for
19: return dpTable[Q]
```

orders do not dominate each other and are kept in dpTable. The addPlan function (line 13) maintains the plans in an entry of dpTable.

We modify buildScan and buildJoin, and add a buildRFLT function, which is presented in Algorithm 2, to add RFLT operators. First, for each scan operator created in the seeding phase, an RFLT operator is added as its parent operator (line 3). When adding an RFLT operator in buildRFLT, the query optimizer finds the incoming predicate paths for the sortkey vertex of the scan operator by traversing the query graph and choosing only Vlists that are more effective than the user-defined threshold in getEffectivePPath function. We refer to Vlists that are expected to filter inputs more than a user-defined ratio as effective Vlists. The effect of Vlists is estimated from (3). By only using effective Vlists, we can avoid the overhead incurred by Vlists with an insignificant pruning effect. From our experience, a threshold value of about 0.7 is adequate.

Next, 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 (line 10) (recall the merge process in Figure 7). Furthermore, the intersection of the PPSs of the merged RFLT operators becomes the PPS of the new RFLT operator (line 11). We take the intersection in order to use only Vlists that are effective for all scan operators.

This extension of the query optimizer to incoporate RFLT operators does not incur much additional computation. It requires the traversing of the query graph, which is small-sized (in getEffectivePPath function), and accessing the statistical

Algorithm 2 Operator Build Functions

procedure buildScan (tp)

```
1: P ← a set of all possible scan operators for tp
2: for ∀p ∈ P do
3:     p ← buildRFLT(p)
4: end for
5: return P
```

procedure buildJoin (p_1, p_2)

```
1: P ← a set of all possible join plans for p1 and p2
2: for ∀p ∈ P do
3:     p ← buildRFLT(p)
4: end for
5: return P
```

procedure buildRFLT (p)

```
1: op ← the root operator of p
2: if op is scan operator then
3:     v ← the sortkey vertex of op;
4:     rootOp.ChildOP ← {op};
5:     rootOp.PPS ← getEffectivePPath(InPPath(v, maxL), scan.predicate);
6:     rootOp.Cost ← getCost(rootOp);
7:     return rootOp
8: else if op is merge join operator then
9:     v ← the sortkey vertex of op;
10:    rootOp.ChildOP ← ⋃_{c∈p.ChildOP} c.ChildOP;
11:    rootOp.PPS ← ⋂_{c∈p.ChildOP} c.PPS;
12:    rootOp.Cost ← getCost(rootOp);
13:    return rootOp
14: end if
```

information for estimating the output cardinalities which is resident in memory (in getCost function).

6 Maintaining the RP-index

In this section, we present the method of building the RP-index and discuss a technique for its incremental update.

6.1 RP-index building

Building the RP-index creates Vlists for predicate paths whose length is up to $maxL$ in the RDF database. A Vlist for a predicate path can be built using the path-pattern query corresponding to the predicate path. That is, we can build $Vlist(\langle p_1, p_2\rangle)$ by a query joining $D(p_1)$ and $D(p_2)$ ($D(p)$ is a relation containing triples in the RDF database D whose predicates are p). However, if we build each Vlist separately using its corresponding query, many computations would be performed in duplicate. For example, to build $Vlist(\langle p_1, p_2, p_3\rangle)$, we have to join $D(p_1)$ and $D(p_2)$ again,

which was computed during the building of $Vlist(\langle p_1, p_2 \rangle)$. To reduce these duplicate computations, we build a Vlist for an i-length predicate path ($i > 1$) using the Vlist for the $(i-1)$-length predicate path (its longest proper prefix) as follows:

$$Vlist(ppath) = \rho_{ID}\left(\pi_O\left(Vlist(ppath_{pre}) \rtimes_{ID=S} D(p)\right)\right) \quad (6)$$

where $ppath_{pre}$ is the longest proper prefix of $ppath$ and p is the last predicate of $ppath$. In this equation, we view a Vlist as a relation with an ID column and $D(p)$ as a relation with S, P, and O columns. We build Vlists in a breadth-first fashion (that is, from 1-length Vlists to $maxL$-length Vlists) and reuse Vlists built in the previous step. In this way, we can reduce the number of duplicate computations.

There are some implementation issues related to the discriminative and frequent predicate paths. As discussed in Section 4.3, we only store Vlists for discriminative and frequent predicate paths in an attempt to address the size problem of the RP-index. Due to this, there are some cases where it is impossible to build Vlists using (6). For example, if $Vlist(\langle p_1, p_2 \rangle)$ is infrequent, then we cannot use (6) to build $Vlist(\langle p_1, p_2, p_3 \rangle)$ because $Vlist(\langle p_1, p_2 \rangle)$ is not stored in the RP-index. In this case, we build $Vlist(\langle p_1, p_2, p_3 \rangle)$ from scratch.

We can skip the building of some infrequent Vlists using their suffix predicate paths. The sizes of Vlists have the following relationship:

$$|Vlist(ppath)| \leq |Vlist(ppath_{suf})| \quad (7)$$

where $ppath_{suf}$ is the proper suffix of $ppath$. That is, $|Vlist(ppath_{suf})|$ is the upper bound of $|Vlist(ppath)|$. Therefore, if $|Vlist(ppath_{suf})|$ is less than the frequency threshold $\psi(|ppath|)$, we do not need to create $Vlist(ppath)$.

Algorithm 3 outlines the process of building the RP-index. BuildRPindex generates the predicate paths in the BFS manner using a queue structure PQ (line 9–11, 23–25). A size-l predicate path is generated by appending a predicate to a size-$(l-1)$ predicate path in PQ. For each generated predicate path $ppath$, the pruning condition (7) is checked (line 12) and, if satisfied, $ppath$ is skipped. Otherwise, $Vlist(ppath)$ is created by calling CreateVlist($ppath$) (line 18) (for building, isUpdate is false). CreateVlist($ppath$) builds a Vlist for $ppath$ using the Vlist of the longest proper prefix of $ppath$ as described in (6). If $Vlist(ppath)$ is not empty, the algorithm checks whether $ppath$ is discriminative and frequent, and if this condition is satisfied, it is stored in the RP-index (lines 20–28).

6.2 Incremental maintenance of the RP-index

In order to ensure the correctness of query results, the RP-index should be consistent with the RDF database and updated concurrently. The easiest way to obtain the newest version of the RP-index is to rebuild it using the updated RDF store. However, it would be very inefficient to rebuild the entire RP-index for every update. In this section, we discuss the incremental maintenance of the RP-index.

We assume that RDF applications have read-mostly workloads in which the updates for RDF stores are usually batched [31]. A batch update is modeled as a set of updated triples U, whose triples are flagged as 'inserted' or 'deleted.' U is divided into two subsets, a set of inserted triples U^+ and a set of deleted triples U^-.

Basically, given a set of updated triples U, all Vlists for the predicate paths containing $p \in P_U$ (the set of predicates in U) should be updated. As we can see

Algorithm 3 RP-index Build

procedure BuildRPindex (isUpdate, D, *maxL*)

```
1: /* We share the building algorithm for updating */
2: /* When building, isUpdate is false*/
3: /* D: RDF database */
4: /* maxL: the maximum length of the predicate path */
5: /* P_D: the set of all predicate in D */
6: /* PQ: a queue of predicate paths */
7: enqueue(⟨⟩, PQ) /* enqueue an empty predicate path in PQ */
8: while PQ ≠ ∅ do
9:    ppath_pre ← dequeue(PQ)
10:   for each p ∈ P_D do
11:      ppath ← append p to ppath_pre
12:      if Vlist(ppath) can be skipped (7) then
13:         continue;
14:      end if
15:      if isUpdate then
16:         vlist ← UpdateVlist(ppath); /* Incremental update (Table 1) */
17:      else
18:         vlist ← CreateVlist(ppath); /* Build using (6) */
19:      end if
20:      if vlist is not empty then
21:         if ppath is discriminative and frequent then
22:            RP-index.insert(ppath, vlist);
23:         end if
24:         if |ppath| < maxL then
25:            enqueue(ppath, PQ);
26:         end if
27:      end if
28:   end for
29: end while
```

from (6), $Vlist(ppath)$ is built from both $Vlist(ppath_{pre})$ and $D(p)$, where p is the last predicate of $ppath$ and $ppath_{pre}$ is the longest proper prefix of $ppath$. Therefore, if neither of these components is changed during the update, $Vlist(ppath)$ does not need to be updated. For example, assume that $p_1 \in P_U$ and that there exist two predicate paths, $\langle p_1, p_2, p_3\rangle$ and $\langle p_1, p_2, p_3, p_4\rangle$, in the RDF graph. If $Vlist(\langle p_1, p_2, p_3\rangle)$ is not changed by the updates, and $p_4 \notin P_U$, then $Vlist(\langle p_1, p_2, p_3, p_4\rangle)$ is not affected by update U, even though $\langle p_1, p_2, p_3, p_4\rangle$ includes p_1.

Table 1 summarizes the update methods of $Vlist(ppath)$ for $|ppath| > 1$. Δ^+ and Δ^- are the sets of inserted and deleted vertices of $Vlist(ppath_{pre})$, respectively. In Table 1, 'rebuild' means that the Vlist should be rebuilt using the createVlist function. Note that if $p \in P_{U^-}$ or $\Delta^- \neq \emptyset$, $Vlist(ppath)$ should be rebuilt. Additionally, note that there are four cases that do not require rebuilding. For three of them, the Vlist can be updated by adding some vertices; for one case, there is no need to update.

We share the procedure with the building process (using isUpdate = true). For each predicate path, the UpdateVlist($ppath$) function checks the delta of the prefix Vlist and the existence of the last predicates in P_{U^+} and P_{U^-}, and updates the Vlists

Table 1 Incremental update method of *Vlist*(*ppath*)

	$p \in P_{U+}$ $p \notin P_{U-}$	$p \notin P_{U+}$ $p \notin P_{U-}$	$p \in P_{U-}$
$\Delta^+ \neq \emptyset$ $\Delta^- = \emptyset$	Add $\Delta^+ \rtimes D(p)$ and $Vlist(ppath_{pre}) \rtimes U^+(p)$	Add $\Delta^+ \rtimes D(p)$	Rebuild
$\Delta^+ = \emptyset$ $\Delta^- = \emptyset$	Add $Vlist(ppath_{pre}) \rtimes U^+(p)$	None	Rebuild
$\Delta^- \neq \emptyset$	Rebuild	Rebuild	Rebuild

$\Delta^+(\Delta^-)$: the set of inserted (deleted) vertices of $Vlist(ppath_{pre})$

according to Table 1. Besides updating the existing Vlists, some Vlists should be created by the update. The Vlists for the newly created predicate paths should be created. It is also possible that a non-discriminative or infrequent predicate path in the old version becomes discriminative and frequent in the updated RP-index, and vice versa. UpdateVlist(*ppath*) creates these Vlists, as well as updating existing Vlists.

There are several ways to reduce the updating overhead of the RP-index. For example, the RP-index can be updated in the background, while accepting user-queries. When committing the updates of the RDF-store, all Vlists to be updated are marked as 'stale'. Then, a background process starts to update the stale Vlists, and updated Vlists become 'normal'. The query compiler should check the status of the Vlists to be used. If the considered Vlist is stale, the query compiler does not use it. Using this method, we can reduce the downtime incurred by updating the RP-index. Additionally, note that updating caused by deletion can be deferred. This is because the vertices to be deleted in Vlists do not cause false negatives and do not affect the query results.

7 Experimental results

We have 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. Implementation includes the RFLT operator, extension of the query optimizer, and the RP-index builder.

All 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 ran the experiments using five datasets: DBpedia SPARQL Benchmark (DBSPB) [29], Lehigh University Benchmark (LUBM) [16], Social Network Intelligence Benchmark[2] (SNIB), Yet Another Great Ontology 2 (YAGO2) [19], and SPARQL Performance Benchmark (SP2B) [36]. DBSPB is a synthetic dataset, but it simulates the data distribution of DBpedia [5] and has the characteristics of a real-world dataset [29]. LUBM is a benchmark dataset whose domain is the university, and SNIB is another synthetic dataset whose domain is a social network site. YAGO2

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

[2] http://www.w3.org/wiki/Social_Network_Intelligence_BenchMark

is a knowledge-base derived from Wikipedia,[3] WordNet [12], and GeoNames,[4] and SP2B is a benchmark that simulates the DBLP scenario.[5]

The benchmark datasets (DBSPB, LUBM, SNIB, and SP2B) have their own scale factors. We used the database size parameter of 200 % for DBSPB, generated 10,000 universities for LUBM, 30,000 users for SNIB, and 144 GB-size triples for SP2B. These datasets have different characteristics, as shown in Table 2. DBSPB has a large number of predicates, while the others have a relatively small number of predicates. This is because DBSPB is a collection of data from various domains. In contrast, LUBM, SNIB, and SP2B are single-domain datasets, and YAGO2 is made from three data sources. Using DBSPB, we can evaluate our approach with a more realistic and heterogeneous dataset.

7.1 RP-index size

We built three RP-indices ($maxL = 3$) for each dataset by varying the following parameters: γ, $\psi(l)$, and reverse predicates. Table 3 shows the three different settings for the RP-indices. We use the frequent threshold function $\psi(l) = ((l-1)/maxL)^2 \times n$, where n is chosen appropriately for each dataset (we use 1,000 for DBSPB, SNIB and SP2B, and 10,000 for LUBM and YAGO2). We call RP-indices under Setting 3 *reduced RP-indices*, because they are built for the discriminative and frequent predicate paths.

Tables 4 and 5 show the number of Vlists and the size of RP-indices built for each dataset. Note that the number of Vlists in LUBM under Setting 1 is only 122, which is much smaller than the number of possible predicate paths (18^3). This is because LUBM has a relatively structured scheme, almost similar to the relational table. Next, as this table shows, the inclusion of reverse predicates increases the number of Vlists and the size of the RP-index significantly (comparing Setting 1 with Setting 2). For DBSPB, we could not even build an RP-index under Setting 2, as it was too large to complete the construction (more than 200 GB). This is because the addition of the reverse predicates causes an increase in the possible predicate paths to be indexed. Nonetheless, we could reduce the size of the RP-index effectively by storing only Vlists for the discriminative and frequent predicate paths (Setting 3).

7.1.1 RP-index with the predicate variable

We propose some methods for handling queries with predicate variables in Section 4.5, one of which is to index the predicate variables when building the RP-index. In this section, we discuss the effects of indexing the predicate variable on the size of the RP-index. We built the RP-index with the predicate variables for the LUBM and YAGO2 datasets, using Setting 3 for the building parameters.

Table 6 shows the size of the RP-index with the predicate variables and the number of Vlists with the predicate variables. We can see that including predicate variables significantly increases the size of the RP-index and the number of Vlists. However, if we adjust the parameters of the RP-index appropriately, we expect to

[3] http://www.wikipedia.org

[4] http://www.geonames.org

[5] http://www.informatik.uni-trier.de/~ley/db/

Table 2 Statistics about datasets

	Predicates	URIs	Literals	Triples (millions)	RDF-3X size (GB)
DBSPB	39,675	38,401,849	154,972	183	25
LUBM	18	217,006,887	111,613,881	1,335	77
SNIB	44	35,197,509	2,119,818	387	17
YAGO2	93	6,872,931	22,452,390	37	9
SP2B	77	267,134,673	523,228,402	1,399	123

Table 3 RP-index parameter settings

Setting	$maxL$	γ	$\psi(l)$	Reverse predicate
1	3	1	0	Not included
2	3	1	0	Included
3	3	0.7	$(l - 1/maxL)^2 \times n$	Included

Table 4 Number of Vlists in RP-indices

Setting	DBSPB	LUBM	SNIB	YAGO2	SP2B
1	34,205,462	122	1,193	8,479	4,875
2	N/A	1,718	10,070	167,114	389,070
3	120,424	63	253	10,023	86,050

Table 5 Total size of RP-indices (GB)

Setting	DBSPB	LUBM	SNIB	YAGO2	SP2B
1	2.85	0.307	1.46	0.08	2.05
2	N/A	19.12	8.83	2.20	87.99
3	6.52	1.39	0.47	0.79	21.97

Table 6 RP-index with the predicate variables (setting 3)

	LUBM	YAGO2
Size (GB)	11	6.3
# of Vlists with predicate vars.	314	12,562

be able to reduce the size overhead due to the inclusion of the predicate variables. We leave the tuning and optimization techniques of indexing predicate variables for future work.

7.2 Query evaluation performance

In this section, we present the query performance of R3F using the three RP-indices built in the previous section. For the experiments, we made four test queries for each dataset (included in the Appendix). Our test queries have many joins (4–5) and relatively long paths. Each query was executed a total of 10 times, and the average execution time is presented.

Figure 11 shows the execution times (for DBSPB, Setting 2 is not included because it could not be built). In this figure, we can see that R3F reduces the execution time of most queries. In particular, there are some queries for which R3F reduces the execution times significantly, by a factor of more than 5 (for instance, Q1 of LUBM, Q2 of SNIB, Q1 of YAGO2, and Q2 of SP2B). These queries have selective path patterns, which R3F can use to effectively filter redundant triples. However, there also exist queries (e.g., Q1 of DBSPB, Q3 of LUBM, and Q4 of SP2B) for which R3F is not very effective. These queries do not have the selective path patterns that R3F uses for triple filtering.

In most cases, the RP-indices under Settings 2 and 3 (with the reverse predicates) are more effective than those under Setting 1 (for Q2 of DBSPB, Q4 in LUBM, and Q1 and Q3 of SNIB, Q3 and Q4 of YAGO2, and Q2 and Q3 of SP2B). This is because RP-indices with reverse predicates index more predicate paths for use in triple filtering. Additionally, we can observe that, although the reduced RP-indices under Setting 3 are much smaller than the RP-indices under Setting 2, their filtering power is not significantly degraded. This is because the criteria proposed in Section 4.3 do not harm the filtering power of the RP-index much. However, for some queries (for example, Q4 of DBSPB), the execution time of Setting 3 is longer than that of Setting 1. This is because the reduced RP-index removes Vlists that are effective for the queries. Nonetheless, the performance of Setting 3 is still good compared to RDF-3X.

Figure 12 shows the intermediate results generated during query evaluation for each query. The intermediate results counted in these experiments are the outputs of scan operators and join operators. We can see that the results have some correlation with the execution times, and that the number of redundant intermediate results is reduced considerably for queries where R3F is effective.

7.2.1 Filter usage

Table 7 shows the usage information of Vlists for SNIB queries: the number of Vlists for each length of predicate path and for predicate paths with reverse predicates, the size of Vlists and the triples read in scan operators. From this table, we can see that the size of the Vlists is generally small compared to the size of the triple data, and therefore R3F incurs little overhead above the original query processing. By comparing Setting 1 and Setting 2, we can see that the number of Vlists to be applied is increased by the reverse predicates. Also, note that only 3-length predicate paths are used in Setting 1 and Setting 2, whereas in Setting 3, 1-length and 2-length predicate paths are used. This is because 3-length predicate paths are removed, as

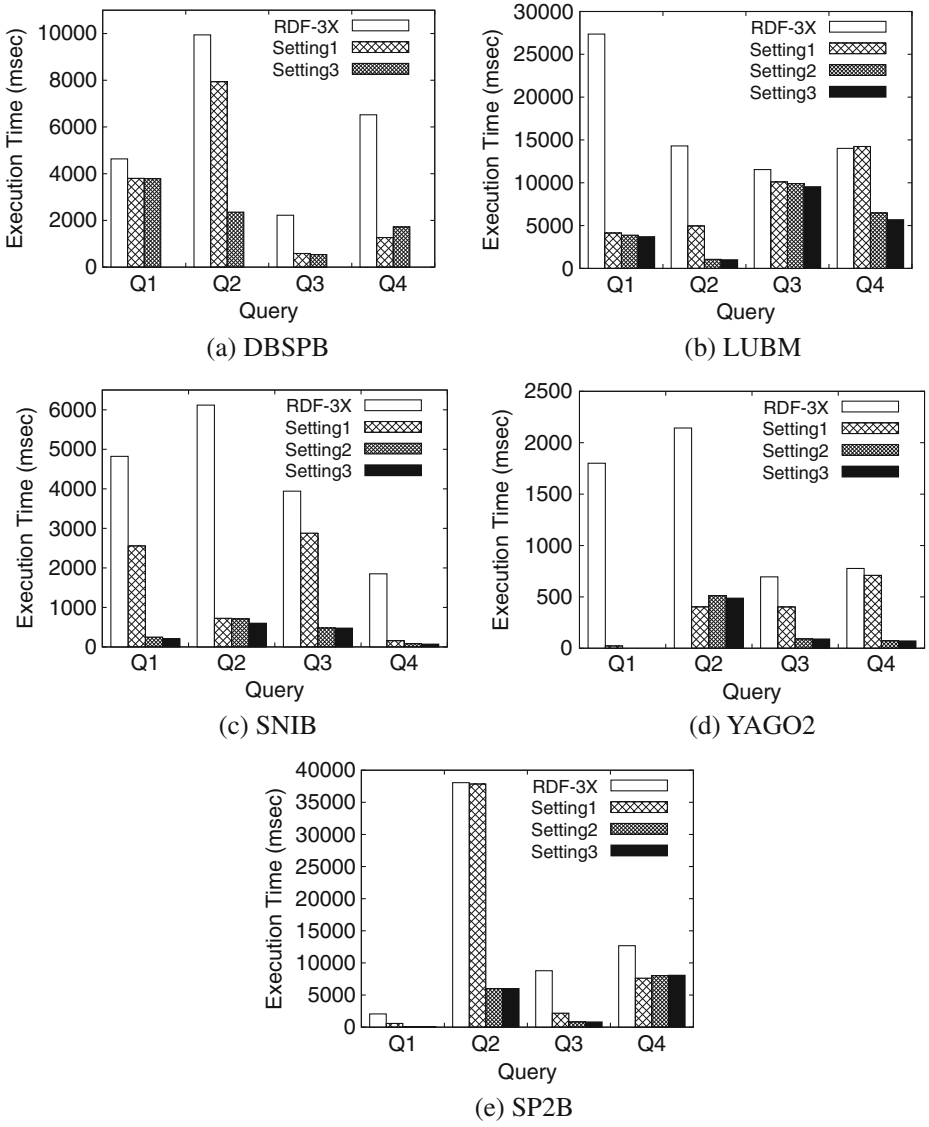


Figure 11 Query execution time

they are not discriminative or frequent, and replaced by shorter predicate paths in Setting 3.

7.2.2 Path query

In order to evaluate R3F for more general cases, we generated random path-pattern queries with lengths of 4, 6, 8, and 10 (an n-length path-pattern query has n triple patterns connected as a path) for the YAGO2 dataset. For each length, we generated 100 queries by varying the predicates (including reverse predicates). We

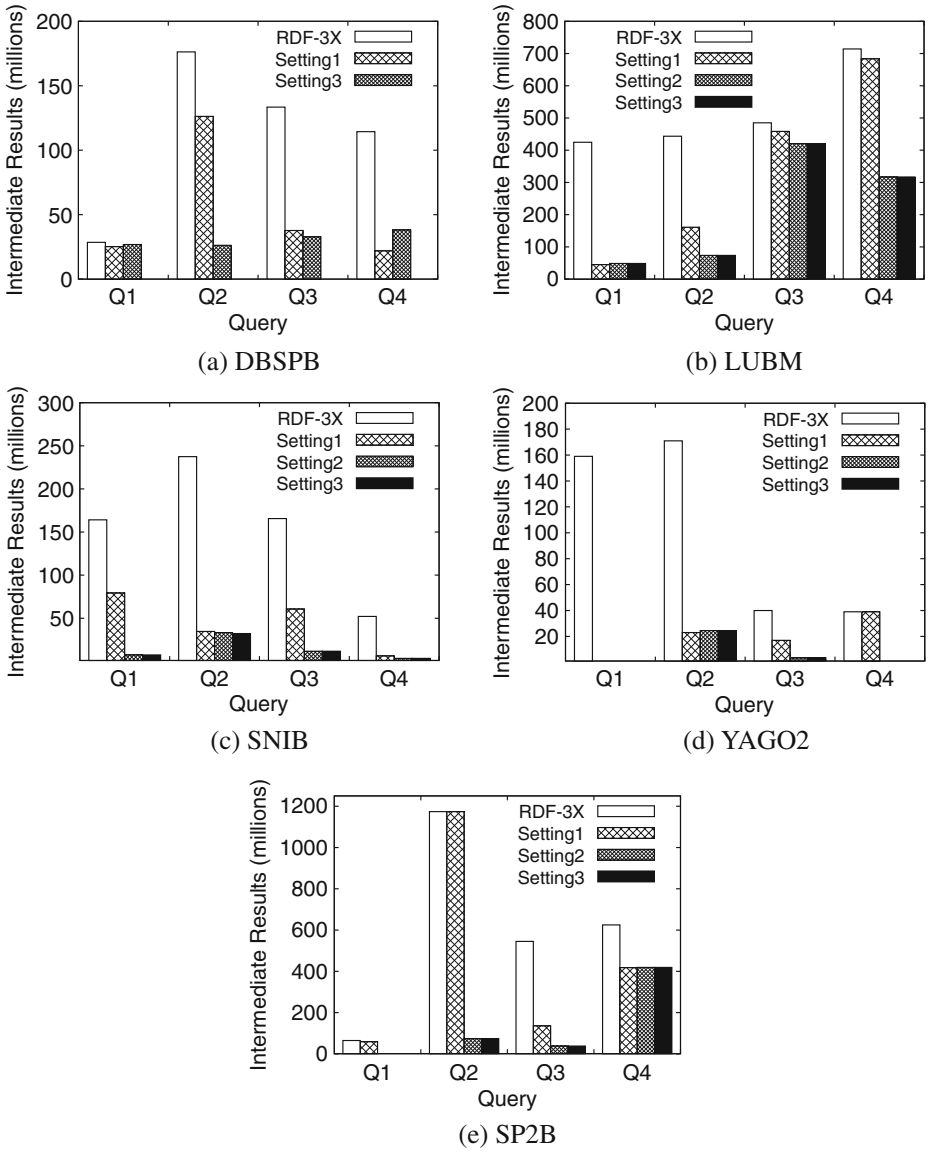


Figure 12 Intermediate results

also evaluated the effects of user-defined parameters of the RP-index ($maxL$, γ, and $\psi(l)$) using these queries.

Figure 13 shows the average execution time for each path length. We can see that path queries are processed more efficiently using R3F. The results are similar to those in the previous experiments using the test queries. The RP-index under Setting 2 is most effective, and the RP-index under Setting 3 is next. However, we can see that the evaluation times do not improve as much as in the previous experiments.

Table 7 Filter usage (SNIB)

Setting	Query	Path length			Reverse	Vlist size	Data size
		1	2	3		(MB)	(MB)
1	1	0	0	4	0	4.98	62.92
	2	0	0	1	0	1.03	204.43
	3	0	0	4	0	3.14	97.78
	4	0	0	4	0	0.09	11.32
2	1	0	0	11	7	6.15	40.48
	2	0	0	7	6	16.75	180.23
	3	0	0	5	3	8.08	78.60
	4	0	0	18	12	0.55	11.32
3	1	2	1	1	2	0.75	40.48
	2	0	2	2	2	16.36	186.18
	3	0	3	1	2	8.47	78.60
	4	4	0	0	2	0.05	11.32

This is because that we generated 100 path queries, and the averaged times are presented. In the query sets, there exist queries without selective path patterns, for which R3F is not effective. And some of queries have no results. These queries tend to be processed quicker than queries with results, and RDF-3X process these queries very fast. As a result, the averaged improvement is not as impressive as the previous experiments. Also, we can observe that the execution times do not increase linearly with the path length (the execution times for 8-length queries are longer than those of 10-length queries). This is because, as the path queries increase, the possibility that they have no results also increases.

Figures 14, 15, and 16 show the effect of the three RP-index parameters on its performance and size. In Figure 14, we decrease the discriminative ratio γ with fixed $maxL = 3$ and $\psi(l) = 0$. From Figure 14a, we can see that the execution times increase as γ decreases. However, the degradation is slight compared to the decreased size of the RP-index (Figure 14b). In Figure 15, we increase n in the frequency function $\psi(l) = (l - 1/maxL)^2 \times n$ with fixed $maxL = 3$ and $\gamma = 1$. From this figure, we can see that the execution times increase as n increases. Again, the degradation is tolerable considering the decreased size of the RP-index. Figure 16 shows the effects of $maxL$. Contrary to the previous two parameters, an increase in $maxL$ does not give a notable increase in performance, although the size of the RP-

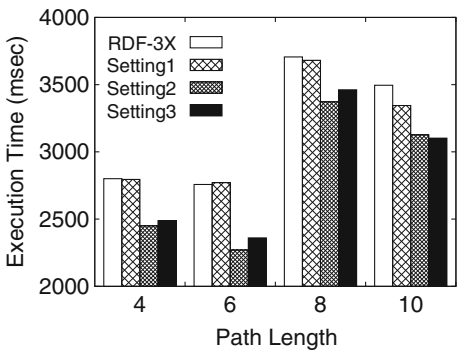
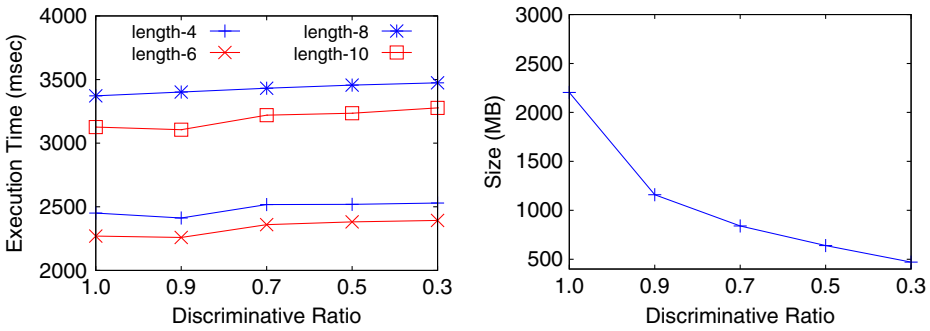
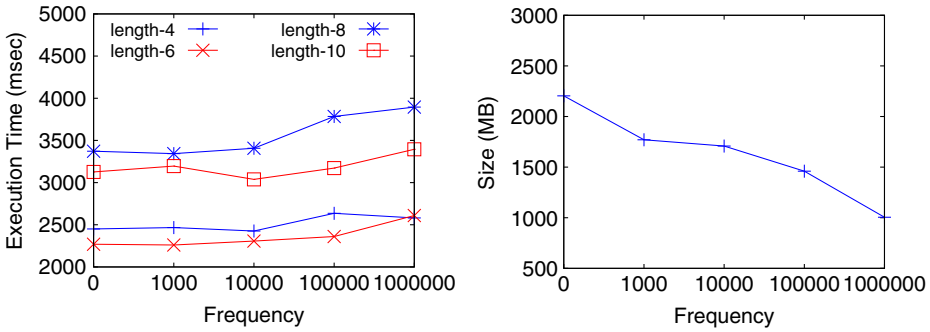


Figure 13 Path query (YAGO2)



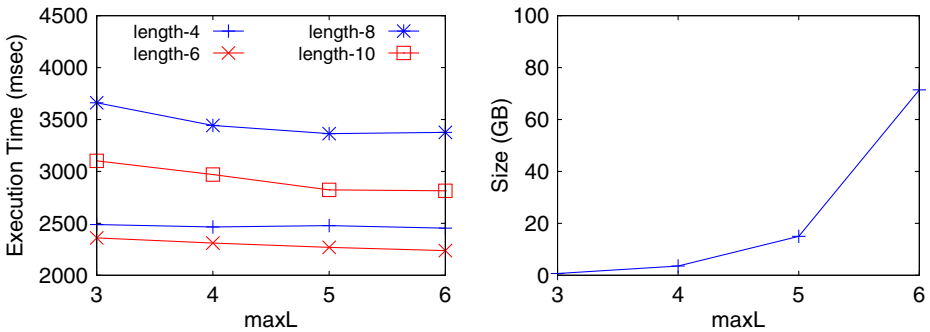
(a) Execution Time (ms) (b) Size (GB)

Figure 14 Effects of discriminative ratio (YAGO2)



(a) Execution Time (ms) (b) Size (GB)

Figure 15 Effects of frequency function (YAGO2)



(a) Execution Time (ms) (b) Size (GB)

Figure 16 Effects of *maxL* (YAGO2)

Table 8 Cardinality estimation errors (using upper bound)

Query	DBSPB	LUBM	SNIB	YAGO2	SP2B
1	1.81	2.13	2.07	146.04	24.84
2	10.77	1.14	3.56	3593.3	1.28
3	13.00	2.63	12.3	1.92	6.22
4	11.31	13.57	1.09	2.14	186.6

index increases exponentially. Therefore, as we discussed in Section 4.6, we do not need a large *maxL* value.

7.2.3 Accuracy of cardinality estimation

In this section, we study the accuracy of the cardinality estimation technique discussed in Section 5.2. We calculate the q-error $\max(c/\hat{c}, \hat{c}/c)$ [28], where c is the real cardinality and $\hat{c}$ is the estimated cardinality. This is the method used in [30] to evaluate estimation techniques. In Section 5.2, we need to estimate the intersection of Vlists and the input sortkey columns, and for this, we use the upper bound of the intersection. Tables 8 and 9 show the q-errors for the experimental queries. The q-errors in Table 8 are calculated using the upper bound, as in Section 5.2, and those in Table 9 are a result of using the exact intersections. From these tables, we can observe that the estimations are more accurate when using the exact intersections, except for YAGO2. The exception of YAGO2 is because the uniform distribution assumption does not hold. We can also note, from Table 9, the estimations are more accurate for the benchmark datasets (LUBM, SNIB and SP2B) than for the real-world datasets (DBSPB and YAGO2). This is because the assumption of the uniform distribution of sortkey values is more adequate for the benchmark datasets. Except for query 2 in YAGO2, we can see that the estimations are generally accurate.

From these results, we can deduce that we need a more accurate estimation of the intersection size and a method to handle cases in which the uniform distribution assumption does not hold.

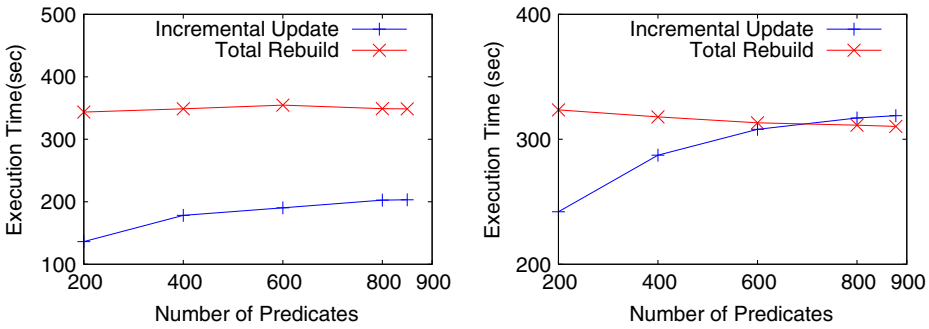
7.3 Incremental update of the RP-index

In this section, we present experimental results for the incremental update of the RP-index. We measured the incremental update times of the RP-index and compared them to the total rebuilding times.

First, we measured the update time, varying the number of predicates in the updates (we refer to a set of updated triples as an update). We use a subset of the DBSPB dataset as D. This has 3,000,000 triples and 1,000 predicates ($|D| = 3,000,000$, $|P_D| = 1,000$). We generated five insert updates, each of which has 100,000 triples ($|U^+| = 100,000$, $|U^-| = 0$), increasing the number of predicates in

Table 9 Cardinality estimation errors (using exact intersection)

Query	DBSPB	LUBM	SNIB	YAGO2	SP2B
1	1.74	1.03	1.13	134.84	1.52
2	10.76	1.14	1.34	5212.7	1.01
3	12.30	2.07	1.09	2.38	1.07
4	11.31	1.25	1.09	6.83	16.90

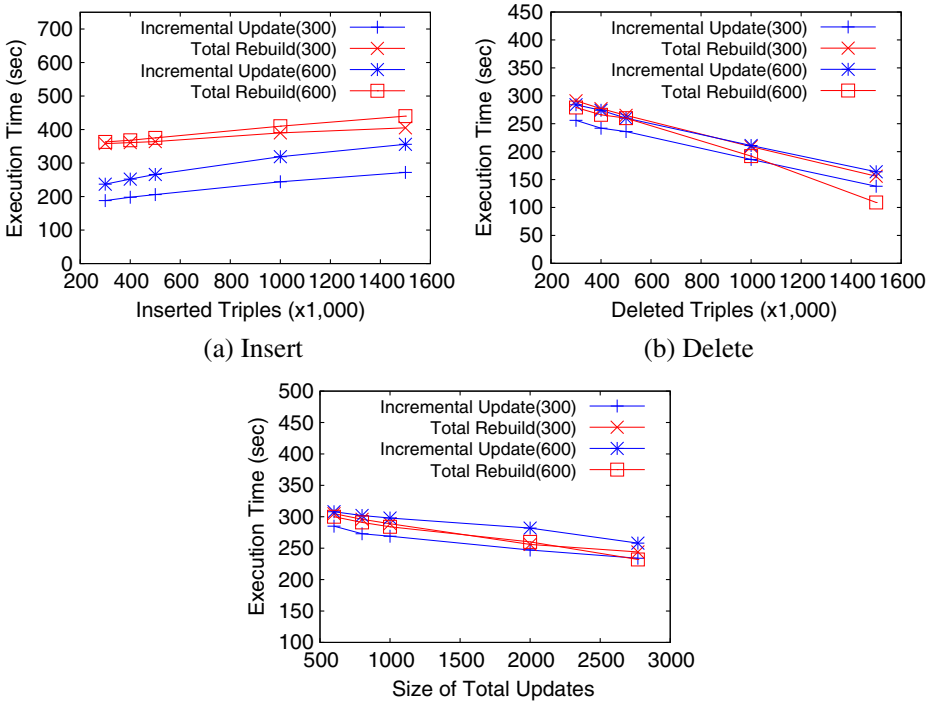


(a) Insert (b) Delete

Figure 17 Update time (predicates)

the updates. Additionally, we generated another five delete updates, each of which has only 100,000 deleted triples ($|U^+| = 0, |U^-| = 100,000$).

Figure 17 shows the update times. As we can see, the update times are proportional to the number of predicates in the updates. This is because the number of Vlists to update increases with the number of predicates. However, the total



(a) Insert (b) Delete

(c) Insert and Delete

Figure 18 Update time (update size)

rebuilding times are almost equal, as the number of predicates in D is not different. Furthermore, note that the update times for insert updates are less than those for delete updates. This is because a Vlist can be updated using the delta of the Vlist for the prefix predicate path, whereas, for delete updates, Vlists are updated using rebuilding.

Next, we measured the effect of the update size on the update time. In this experiment, we generated three types of update: insert-only updates, delete-only updates, and updates with both inserts and deletes, increasing the number of updated triples. Additionally, for each type, updates with 300 predicates and 600 predicates were generated. Figure 18 shows the update times. For insert updates, both the incremental update times and the rebuild times increase as the sizes of the updates increase. In contrast to insert updates, for delete updates and updates with inserts and deletes, the incremental update times are similar to the rebuild times. For updates with inserts and deletes, because of the deleted triples, the results are similar to the delete updates. To alleviate the overhead of the deleted triples, we can use the workarounds in Section 6.2.

8 Conclusions and future work

In this paper, we proposed a novel triple filtering framework called R3F in order to reduce the number of redundant intermediate results in SPARQL query processing. R3F filters out redundant triples using a necessary condition for results based on the incoming predicate path information. To organize the filter data for R3F, we designed an RP-index and considered its size problem and maintenance issues. In addition, we presented the RFLT operator, which conducts the triple filtering, and proposed a cost function to integrate it with the cost-based query optimizer. Through comprehensive experiments using various large-scale RDF datasets, we demonstrated that R3F is very effective in reducing the number of redundant intermediate results, and improved query performance for complex SPARQL queries.

In future research, we plan to extend R3F to exploit the graph features of RDF data and to explore the application of R3F in parallel and distributed environments, such as MapReduce. We will also investigate a more accurate estimation method for the output cardinalities of filtered triples.

Acknowledgement This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea Government(MSIP) (No. 20120005695).

Appendix

Query sets

We include the queries used in our experiments. For the queries, we use the following prefixes.

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX lubm:<http://www.lehigh.edu#>
PREFIX dbpowl:<http://dbpedia.org/ontology/>
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbpprop:<http://dbpedia.org/property/>
```

```
PREFIX foaf:<http://xmlns.com/foaf/0.1/>
PREFIX sioc:<http://rdfs.org/sioc/ns#>
PREFIX sib:<http://www.ins.cwi.nl/sib/>
PREFIX sibv:<http://www.ins.cwi.nl/sib/vocabulary/>
PREFIX geo:<http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX yago2:<http://www.mpii.de/yago/resource/>
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#>
```

DBSPB

```
Q1 ?a dbpprop:ground ?b. ?a foaf:homepage ?c. ?b rdf:type ?v8.
   ?d rdfs:label ?e. ?d dbpowl:postalCode ?f. ?d geo:lat ?g.
   ?d geo:long ?h. ?b dbpowl:location ?d. ?b foaf:homepage ?i.
   ?j dbpprop:clubs ?a.
```

```
Q2 ?a rdf:type dbpowl:Person. ?a dbpprop:name ?c. ?e rdfs:label ?f.
   ?a dbpprop:placeOfBirth ?d. ?e dbpprop:isbn ?g.
   ?e dbpprop:author ?a. ?j dbpprop:author ?k. ?k rdfs:label ?b.
   ?e dbpprop:precededBy ?j. ?k dbpprop:name ?c.
   ?k dbpprop:placeOfBirth ?d.
```

```
Q3 ?a dbpprop:nationality ?b. ?a rdfs:label ?c. ?a rdf:type ?e .
   ?b rdfs:label ?d. ?b rdf:type ?e. ?b dbpprop:name ?f.
```

```
Q4 ?a foaf:name ?b. ?a rdfs:comment ?c. ?a rdf:type ?d.
   ?a dbpprop:series ?e. ?e dbpowl:starring ?f. ?f rdf:type ?i.
   ?g dbpowl:starring ?f. ?h dbpowl:previousWork ?g.
```

LUBM

```
Q1 ?a rdf:type lubm:GraduateStudent. ?b rdf:type lubm:University.
   ?c rdf:type lubm:Department. ?c lubm:subOrganizationOf ?b.
   ?a lubm:memberOf ?c. ?a lubm:undergraduateDegreeFrom ?b.
```

```
Q2 ?a rdf:type lubm:FullProfessor.
   ?a lubm:headOf ?b. ?e lubm:undergraduateDegreeFrom ?c.
   ?a lubm:teacherOf ?d. ?e rdf:type lubm:GraduateStudent.
   ?b lubm:subOrganizationOf ?c. ?e lubm:teachingAssistantOf ?d.
```

```
Q3 ?a rdf:type lubm:GraduateStudent. ?b lubm:headOf ?c.
   ?b rdf:type lubm:FullProfessor. ?a lubm:advisor ?b.
   ?d lubm:publicationAuthor ?a. ?d lubm:publicationAuthor ?b.
```

```
Q4 ?a rdf:type lubm:UndergraduateStudent.  ?b lubm:headOf ?d.
   ?b rdf:type lubm:FullProfessor.  ?a lubm:advisor ?b.
   ?c rdf:type lubm:Course.  ?a lubm:takesCourse ?c.
   ?b lubm:teacherOf ?c.
```

SNIB

```
Q1 ?a foaf:knows ?b. ?a sibv:Engaged_with ?c.
   ?c sioc:moderator_of ?d. ?b foaf:knows ?c.
   ?d sioc:container_of ?e. ?e sib:like ?a.
```

```
Q2 ?a sib:initiator ?b. ?a sib:memb ?b. ?a sib:memb ?c.
   ?b foaf:knows ?e. ?g sib:tag ?b. ?g a sib:Photo.
   ?a sib:declined ?d. ?e sibv:Married_with ?c.
   ?c sioc:creator_of ?f. ?f sioc:container_of ?g.
   ?f a sioct:ImageGallery.
```

```
Q3 ?a sib:tag ?b. ?b foaf:knows ?c.
   ?c sibv:Married_with ?d. ?e sioc:container_of ?a.
   ?d sioc:creator_of ?e.
```

```
Q4 ?a foaf:knows ?b. ?b foaf:knows ?c. ?c foaf:knows ?d.
   ?d foaf:knows ?a. ?b sibv:Married_with ?d.
```

YAGO2

```
Q1 ?a yago2:isCitizenOf ?b.  ?a yago2:hasPreferredName ?c.
   ?a yago2:hasAcademicAdvisor ?d.  ?b yago2:isLocatedIn ?e.
   ?d yago2:isCitizenOf ?f.  ?d yago2:hasPreferredName ?g.
   ?f yago2:isLocatedIn ?e.
```

```
Q2 ?a yago2:wasBornIn ?b.  ?a yago2:isCalled ?c.
   ?a yago2:isMarriedTo ?b.  ?b yago2:isLocatedIn ?d.
   ?a yago2:isCalled ?e.  ?a yago2:livesIn ?f.
   ?f yago2:isLocatedIn ?d.
```

```
Q3 ?a yago2:hasFamilyName ?b.  ?a yago2:directed ?c.
   ?d yago2:hasFamilyName ?e.  ?d yago2:actedIn ?c.
   ?d yago2:isMarriedTo ?a.    ?c yago2:isCalled ?e.
   ?c yago2:hasPreferredName ?f.  ?c rdf:type ?g.
```

```
Q4 ?a yago2:isKnownFor ?b.  ?a yago2:directed ?c.
   ?a yago2:wasBornIn ?d.   ?c yago2:wasCreatedOnDate ?e.
   ?c yago2:isCalled ?f.    ?c rdf:type ?g.
   ?b rdf:type ?h.  ?d yago2:isLocatedIn ?i.
```

SP2B

```
Q1
   ?a dcterms:references ?b.  ?a a bench:Inproceedings.
   ?b rdf:_1 ?c.  ?b rdf:_2 ?d.
   ?c dcterms:references ?e.
   ?e rdf:_1 ?f.  ?e rdf:_2 ?g.
   ?d dcterms:references ?h.
   ?h rdf:_1 ?i.  ?h rdf:_2 ?j.
```

```
Q2 ?a swrc:editor ?b.  ?c dc:creator ?b.
   ?c dcterms:partOf ?a.
```

```
Q3 ?a dc:creator ?b.  ?b foaf:name ?c.
   ?a dc:title ?d.    ?a bench:abstract ?e.
   ?a dcterms:references ?f.  ?f rdf:_50 ?g.
```

```
Q4 ?a swrc:editor ?b.  ?c swrc:editor ?b.
   ?b foaf:name ?d.    ?a dc:creator ?b.
   ?a dc:title ?e.     ?a dcterms:references ?f.
   ?f rdf:_10 ?g.
```

References

1. Abadi, D.J., Marcus, A., Madden, S., Hollenbach, K.: SW-Store: a vertically partitioned DBMS for semantic web data management. VLDB J. **18**(2), 385–406 (2009)
2. Atre, M., Chaoji, V., Zaki, M.J., Hendler, J.A.: Matrix "bit" loaded: a scalable lightweight join query processor for RDF data. In: Proceedings of the 19th International Conference on World Wide Web (WWW 2010) (2010)

3. Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.D.: Magic sets and other strange ways to implement logic programs. In: Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS 1986) (1986)
4. Bernstein, P.A., Chiu, D.M.W.: Using semi-joins to solve relational queries. J. ACM **28**(1), 25–40 (1981)
5. Bizer, C., Lehmann, J., Kobilarov, G., Auer, S., Becker, C., Cyganiak, R., Hellmann, S.: DBpedia—a crystallization point for the web of data. J. Web Sem. **7**(3), 154–165 (2009)
6. Bröcheler, M., Pugliese, A., Subrahmanian, V.S.: DOGMA: a disk-oriented graph matching algorithm for RDF databases. In: Proceedings of the 8th International Semantic Web Conference (ISWC 2009) (2009)
7. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: a generic architecture for storing and querying RDF and RDF schema. In: Proceedings of the First International Semantic Web Conference (ISWC 2002) (2002)
8. Carroll, J.J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., Wilkinson, K.: Jena: implementing the semantic web recommendations. In: Proceedings of the 13th International Conference on World Wide Web—Alternate Track Papers & Posters (WWW 2004) (2004)
9. Chebotko, A., Lu, S., Fotouhi, F.: Semantics preserving SPARQL-to-SQL translation. Data Knowl. Eng. **68**(10), 973–1000 (2009)
10. Chen, M.S., Hsiao, H.I., Yu, P.S.: On applying hash filters to improving the execution of multi-join queries. VLDB J. **6**(2), 121–131 (1997)
11. Erling, O., Mikhailov, I.: RDF support in the Virtuoso DBMS. In: Proceedings of the 1st Conference on Social Semantic Web (CSSW 2007) (2007)
12. Fellbaum, C. (ed.): WordNet An Electronic Lexical Database. The MIT Press (1998)
13. Goldman, R., Widom, J.: DataGuides: enabling query formulation and optimization in semistructured databases. In: Proceedings of 23rd International Conference on Very Large Data Bases (VLDB 1997) (1997)
14. Gou, G., Chirkova, R.: Efficiently querying large XML data repositories: a survey. IEEE Trans. Knowl. Data Eng. **19**(10), 1381 –1403 (2007)
15. Graefe, G.: Query evaluation techniques for large databases. ACM Comput. Surv. **25**(2), 73–170 (1993)
16. Guo, Y., Pan, Z., Heflin, J.: LUBM: a benchmark for OWL knowledge base systems. J. Web Sem. **3**(2–3), 158–182 (2005)
17. He, H., Singh, A.K.: 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) (2008)
18. He, H., Yang, J.: Multiresolution indexing of XML for frequent queries. In: Proceedings of the 20th International Conference on Data Engineering (ICDE 2004) (2004)
19. Hoffart, J., Suchanek, F.M., Berberich, K., Weikum, G.: YAGO2: a spatially and temporally enhanced knowledge base from Wikipedia. Artif. Intell. **194**, 28–61 (2013)
20. Huang, H., Liu, C., Zhou, X.: Approximating query answering on RDF databases. World Wide Web **15**(1), 89–114 (2012)
21. Kaushik, R., Shenoy, P., Bohannon, P., Gudes, E.: Exploiting local similarity for indexing paths in graph-structured data. In: Proceedings of the 18th International Conference on Data Engineering (ICDE 2002) (2002)
22. Kim, K., Moon, B., Kim, H.J.: 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) (2011)
23. Klyne, G., Carroll, J.J.: Resource description framework (RDF): concepts and abstract syntax. W3c recommendation, World Wide Web Consortium (2004)
24. Köhler, H.: Estimating set intersection using small samples. In: Proceedings of the Thirty-Third Australasian Computer Science Conference (ACSC 2010) (2010)
25. Kuramochi, M., Karypis, G.: Finding frequent patterns in a large sparse graph. In: Proceedings of the Fourth SIAM International Conference on Data Mining (SDM 2004) (2004)
26. Maduko, A., Anyanwu, K., Sheth, A.P., Schliekelman, P.: Graph summaries for subgraph frequency estimation. In: Proceedings the 5th European Semantic Web Conference (ESWC 2008) (2008)
27. Milo, T., Suciu, D.: Index structures for path expressions. In: Proceedings of the 7th International Conference on Database Theory (ICDT 1999) (1999)
28. Moerkotte, G., Neumann, T., Steidl, G.: Preventing bad plans by bounding the impact of cardinality estimation errors. PVLDB **2**(1), 982–993 (2009)

29. Morsey, M., Lehmann, J., Auer, S., Ngomo, A.C.N.: DBpedia SPARQL benchmark—performance assessment with real queries on real data. In: Proceedings of the 10th International Semantic Web Conference (ISWC 2011) (2011)
30. Neumann, T., Moerkotte, G.: Characteristic sets: accurate cardinality estimation for RDF queries with multiple joins. In: Proceedings of the 27th International Conference on Data Engineering (ICDE 2011) (2011)
31. Neumann, T., Weikum, G.: RDF-3X: a RISC-style engine for RDF. PVLDB **1**(1), 647–659 (2008)
32. Neumann, T., Weikum, G.: Scalable join processing on very large RDF graphs. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2009) (2009)
33. Owens, A., Seaborne, A., Gibbins, N.: Clustered TDB: a clustered triple store for Jena. Tech. rep., University of Southampton (2008)
34. Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF. W3c recommendation, W3C Recommendation (2008)
35. Qun, C., Lim, A., Ong, K.W.: D(k)-index: an adaptive structural summary for graph-structured data. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD 2003) (2003)
36. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: SP2Bench: a SPARQL performance benchmark. In: Proceedings of the 25th International Conference on Data Engineering (ICDE 2009) (2009)
37. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database management system. In: Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data (SIGMOD 1979) (1979)
38. Shasha, D., Wang, J.T.L., Giugno, R.: 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) (2002)
39. Sidirourgos, L., Goncalves, R., Kersten, M.L., Nes, N., Manegold, S.: Column-store support for RDF data management: not all swans are white. PVLDB **1**(2), 1553–1563 (2008)
40. Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., Reynolds, D.: SPARQL basic graph pattern optimization using selectivity estimation. In: Proceedings of the 17th International Conference on World Wide Web (WWW 2008) (2008)
41. Sun, Z., Wang, H., Wang, H., Shao, B., Li, J.: Efficient subgraph matching on billion node graphs. PVLDB **5**(9), 788–799 (2012)
42. Tian, Y., McEachin, R.C., Santos, C., States, D.J., Patel, J.M.: SAGA: a subgraph matching tool for biological graphs. Bioinformatics **23**(2), 232–239 (2007)
43. Tran, T., Ladwig, G.: Structure index for RDF data. In: Workshop on Semantic Data Management (SemData@VLDB2010) (2010)
44. Udrea, O., Pugliese, A., Subrahmanian, V.S.: GRIN: a graph based RDF index. In: Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI 2007) (2007)
45. Virgilio, R.D., Nostro, P.D., Gianforme, G., Paolozzi, S.: A scalable and extensible framework for query answering over RDF. World Wide Web **14**(5–6), 599–622 (2011)
46. Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. PVLDB **1**(1), 1008–1019 (2008)
47. Wong, K.F., Yu, J., Tang, N.: Answering XML queries using path-based indexes: a survey. World Wide Web **9**(3), 277–299 (2006)
48. Yan, X., Yu, P.S., Han, J.: Graph indexing based on discriminative frequent structure analysis. ACM Trans. Database Syst. **30**(4), 960–993 (2005)
49. Zhang, S., Li, S., Yang, J.: GADDI: distance index based subgraph matching in biological networks. In: Proceedings of the 12th International Conference on Extending Database Technology (EDBT 2009) (2009)
50. Zhao, P., Han, J.: On graph query optimization in large networks. PVLDB **3**(1), 340–351 (2010)
51. Zou, L., Mo, J., Chen, L., Özsu, M.T., Zhao, D.: gStore: answering SPARQL queries via subgraph matching. PVLDB **4**(8), 482–493 (2011)