저작자표시-비영리-변경금지 2.0 대한민국

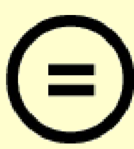
이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

Ph.D. DISSERTATION

A Query Optimization Technique using Graph-Structural Information in Relational RDF Stores

관계형 RDF 저장소에서 그래프 구조적 정보를 사용한 질의 최적화 기법

FEBRUARY 2014

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Kisung Kim

Ph.D. DISSERTATION

A Query Optimization Technique using Graph-Structural Information in Relational RDF Stores

관계형 RDF 저장소에서 그래프 구조적 정보를 사용한 질의 최적화 기법

FEBRUARY 2014

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Kisung Kim

A Query Optimization Technique using Graph-Structural Information in Relational RDF Stores

관계형 RDF 저장소에서 그래프 구조적 정보를 사용한 질의 최적화 기법

지도교수 김 형 주

이 논문을 공학박사 학위논문으로 제출함

2013 년 11 월

서울대학교 대학원

전기 · 컴퓨터 공학부

김 기 성

김기성의 공학박사 학위논문을 인준함

2014 년 1 월

위 원 장	이 상 구	(인)
부위원장	김 형 주	(인)
위 원	문 봉 기	(인)
위 원	김 선	(인)
위 원	임 동 혁	(인)

Abstract

As the size of Resource Description Framework (RDF) graphs has grown rapidly, SPARQL query processing on the large-scale RDF graph has become a more challenging problem. For efficient SPARQL query processing, the handling of the intermediate results is the most crucial element because it generally involves many join operators. In order to address this problem, we propose the triple filtering method that exploits the graph-structural information of RDF data. We design the RDF Path index (RP-index) and the RDF Graph index (RG-index) for the triple filtering. These two indices uses the path information and the graph information of the RDF graph, respectively. However, these indices have the size problem due to the exponential number of the indexed patterns. We address the size problem by indexing only effective the path and graph patterns for the triple filtering. The triple filtering is performed very efficiently by a relational operator called the RDF Filter (RFLT) with little overhead compared to the original query processing. Through comprehensive experiments on large-scale RDF datasets, we demonstrate that our approaches 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

Student Number: 2003-23569

Contents

Abstract i

Contents iii

List of Figures vii

List of Tables ix

Chapter 1 Introduction 1

1.1 Research Motivation 3

1.2 Our Contributions 6

1.3 Outline 11

Chapter 2 Related Work 13

2.1 RDF Stores 13

2.1.1 Summary of Existing Methods of Relation-based RDF Stores 16

2.1.2 Overview of RDF-3X 18

2.2 Handling the Intermediate Results 20

2.3 Path-based and Graph Indices 21

2.4 Frequent Graph Pattern Mining 23

Chapter 3 Preliminaries **25**

3.1 RDF and SPARQL 25

3.2 Path and Graph Pattern 29

3.2.1 Incoming Predicate Path 29

3.2.2 k-neighborhood Subgraph 30

3.3 Candidate Vertex Set 31

Chapter 4 R3F: RDF Triple Filtering Framework using RP-index **35**

4.1 Motivating Example 35

4.2 Overall Process of R3F 37

4.3 RP-index Definition 38

4.3.1 Physical Structure of RP-index 39

4.3.2 Discriminative and Frequent Predicate Paths 40

4.3.3 Reverse Predicate 42

4.3.4 Handling Other Types of Queries 45

4.3.5 Determining RP-index Parameters 46

4.4 Processing Triple Filtering 47

4.4.1 RFLT Operator 47

4.5 Generating an Execution Plan with RFLT Operators 52

4.5.1 Filtering Effect of Vlists 54

4.5.2 Cardinality of RFLT Operator 55

4.5.3 Generating an Execution Plan 57

4.6 RP-index Building 59

4.6.1 Complexity of building RP-index 63

4.6.2 Parallel Building Methods 63
4.6.3 Incremental Maintenance 65
4.7 Experimental Results 68
4.7.1 RP-index Size 70
4.7.2 Query Evaluation Performance 73
4.7.3 Incremental Maintenance of RP-index 78

Chapter 5 RG-index: RDF Triple Filtering using the Graph Index 87
5.1 Motivating Example 87
5.2 Design of RG-index 90
5.2.1 Physical Structure of RG-index 92
5.3 Handling the Size Problem of RG-index 96
5.3.1 Discriminative Patterns 96
5.3.2 Frequent Patterns 97
5.4 Building RG-index 98
5.4.1 Overview of gSpan 98
5.4.2 RDF Graph Pattern Mining using gSpan 99
5.4.3 Complexity of building RG-index 106
5.5 Triple Filtering using RG-index 106
5.5.1 Generating an Execution Plan with RFLT Operators 107
5.6 Experimental Results 109
5.6.1 RG-index Size 111
5.6.2 Query Evaluation Performance 112
5.6.3 Index Building Time 116

Chapter 6 Conclusion and Future Work 119

6.1 Future Work 120

Appendices **125**

Chapter A Related Open Source Projects **125**

A.1 RDF-3X 125

A.2 gSpan 129

Chapter B Data Structure of RP-index and RG-index **133**

B.1 RP-index 133

B.2 RG-index 134

Chapter C Query Sets **137**

요약 **151**

Acknowledgements **153**

List of Figures

Figure 1.1 Linked Open Data Cloud Diagram by Richard Cyganiak and Anja Jentzsch. http://lod-cloud.net/ 2

Figure 1.2 Execution Plan 5

Figure 1.3 Overview of RDF Triple Filtering (R3F) 7

Figure 2.1 Frequent Pattern Mining 24

Figure 3.1 RDF Graph 28

Figure 3.2 RDF Graph and k-Neighborhood Subgraph 30

Figure 4.1 SPARQL Query Graph and Execution Plan 36

Figure 4.2 Vlists in RP-index$(D, 3)$ 39

Figure 4.3 A Trie for Predicate Paths 41

Figure 4.4 Extended SPARQL Query 43

Figure 4.5 A SPARQL Query with a Predicate Variable 45

Figure 4.6 Execution Plan using RFLT 49

Figure 4.7 RFLT Operator 50

Figure 4.8 RFLT Operator and U-SIP 53

Figure 4.9 Filtering Effect . . . 55
Figure 4.10 Parallel Building of RP-index . . . 64
Figure 4.11 Query Execution Time . . . 80
Figure 4.12 Intermediate Results . . . 81
Figure 4.13 Path Query (YAGO2) . . . 82
Figure 4.14 Effects of Discriminative Ratio (YAGO2) . . . 82
Figure 4.15 Effects of Frequency Function (YAGO2) . . . 83
Figure 4.16 Effects of *maxL* (YAGO2) . . . 83
Figure 4.17 Update Time (Predicates) . . . 84
Figure 4.18 Update Time (Update Size) . . . 85

Figure 5.1 RDF Graph and SPARQL Query Graph . . . 88
Figure 5.2 RG-index ($maxL = 3$) . . . 91
Figure 5.3 DFS Subscriptions . . . 94
Figure 5.4 Redundant Graph Pattern . . . 101
Figure 5.5 Non-redundant Graph Pattern . . . 102
Figure 5.6 Rightmost Extension . . . 103
Figure 5.7 Backward Extension using the Results of the Forward Extension . . . 104
Figure 5.8 RG-index Size (YAGO2) . . . 111
Figure 5.9 Query Execution Time (YAGO2) . . . 116

Figure 6.1 Infrequent Pattern . . . 121
Figure 6.2 Object Value Distribution of predicate 'isCitizenOf' (YAGO2) 122

List of Tables

Table 1.1 Cardinalities of Intermediate Results 5

Table 2.1 Summary of Existing Approaches and Our Approaches . . . 17

Table 4.1 Incremental Update Method of *Vlist*(*ppath*) 67

Table 4.2 Statistics about Datasets 69

Table 4.3 Graph Densities of Datasets 70

Table 4.4 RP-index Parameter Settings 70

Table 4.5 Number of Vlists in RP-indices 71

Table 4.6 Total Size of RP-indices (GB) 72

Table 4.7 RP-index with the Predicate Variables (Setting 3) 72

Table 4.8 Filter Usage (SNIB) 75

Table 4.9 Cardinality Estimation Errors (using upper bound) 78

Table 4.10 Cardinality Estimation Errors (using exact intersection) . . 78

Table 5.1 DFS Codes of the Graph Pattern 94

Table 5.2 Statistics about Datasets 110

Table 5.3 Graph Densities of Datasets 110

Table 5.4 Index Statistics 113
Table 5.5 Query Statistics 114
Table 5.6 Query Execution Time (ms) 115
Table 5.7 Database Loading Time of RDF-3X 117
Table 5.8 Index Building Time (ms) 117

Chapter 1

Introduction

The Resource Description Framework (RDF) [1] is the core data model for the Semantic Web, and SPARQL [2] is the standard query language for RDF data. RDF data is a set of triples(subject, predicate, object) which describe the relationship between two resources(subject and object). The RDF data forms a graph called RDF graph which consists of the resources and their relationships. 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. RDF features flexibility with little schema restriction and expressive power which can represent graph-structured data. By virtue of these features, it has been utilized in various areas, such as bioinformatics [3, 4], media data [5], Wikipedia [6], social networks [7], and government [8]. Like this, RDF is widely used to represent and integrate data from various domains.

As an real-life example of large-scale RDF data, there is the LOD, Linking Open Data, project [9]. It was initiated and led by W3C, and as its name implies,

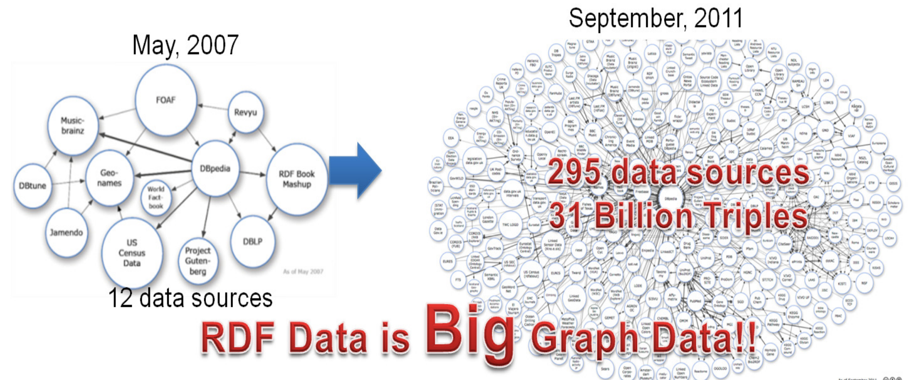


Figure 1.1: Linked Open Data Cloud Diagram by Richard Cyganiak and Anja Jentzsch. http://lod-cloud.net/

the project's goals are to publish various open data sets on the Web as RDF, and to link the data items from different data sources using RDF links. Figure 1.1 is of LOD Cloud diagram. This diagram shows the data sources converted into RDF in LOD project and there relationships. This diagram can show how RDF data has grown rapidly. At May, 2007, there existed only 12 data sources in LOD project. And it kept growing, and at September, 2011, the data sources had increased up to 295 data sources. And the total number of triples amounts to 31 billions. As we can see in this diagram, RDF data becomes real-life Big Graph Data.

The main reasons of the rapid increase of RDF data and its use for integrating data from various data sources are its flexibility and inherent graph structure. Although these benefits give a strong expressive power and flexibility to RDF, it also poses significant challenges for the processing of large-scale RDF data, especially for processing complex SPARQL queries.

In this thesis, we aim to propose an efficient SPARQL query processing tech-

nique which can evaluate SPARQL queries over large-scale RDF graphs. More specifically, we propose a novel filtering method called *RDF Triple Filtering* to address the problem of redundant intermediate results. The triple filtering can accelerate query evaluation by reducing these unwanted intermediate results. It filtering out irrelevant triples retrieved from the scan operators before they are passed to the join operators using graph-structural information of RDF graphs. In this section, we explain our research motivation with concrete examples and provide the comparisons with the privious approaches. Also we present summary of our contributions and the outline of the thesis.

1.1 Research Motivation

There exist numerous bodies of literature for efficient SPARQL query processing. In order to store large-scale RDF and process SPARQL queries, most state-of-the-art RDF systems employ the relational model. Examples of this relation-based RDF stores are Jena [10], Sesame [11], SW-Store [12], Virtuoso [13], and RDF-3X [14]. Relation-based RDF stores use the relation tables to store RDF data and translate SPARQL queries into relational algebraic expressions [15]. They decompose the RDF graph into triples, which consist of Subject, Predicate and Object, and store these triples in a relation table with three columns (Subject, Predicate ,Object). This table is called the triple table. The SPARQL queries are evaluated through a sequence of joins on the triple table. Let us consider the following SPARQL query.

```
SELECT ?n1 ?n2 ?n3 ?n4
WHERE {
    ?n1 p1 ?n2.
    ?n2 p2 ?n3.
    ?n3 p3 ?n4.
}
```

The above SPARQL query consists of three triple patterns, which form a graph pattern, and the evaluation of this query is to find all subgraphs in the RDF graph matching with the query graph pattern. This SPARQL query requires three scan operations which retrieve the matching triples for each triple pattern from the triple table, and two join operations which combine the retrieved triples.

The main problem of relation-based RDF stores is that they need too many join operators to process SPARQL queries. In general, a SPARQL query with N triple patterns requires $N - 1$ join operations. There has been a lot of research on storing and querying of RDF data [10–14]. However they have limitations that they do not use the graph-structural information of the RDF graph.

Let us consider the previous example SPARQL query. This query is processed normally as follows (Although the physical structures and detailed implementations are different for each RDF engine, they share a common framework for processing RDF data). We assume that the RDF graph is stored in the form of relations (for example, relational tables in Jena [10] and Sesame [11], or clustered B+tree indices in RDF-3X [14]). Then, 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

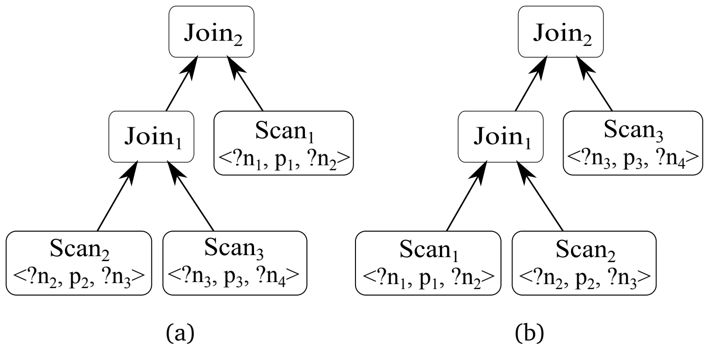


Figure 1.2: Execution Plan

Table 1.1: Cardinalities of Intermediate Results

Graph Pattern	Cardinality
$?n_1 \xrightarrow{p_1} ?n_2 \xrightarrow{p_2} ?n_3 \xrightarrow{p_3} ?n_4$	1,000
$?n_1 \xrightarrow{p_1} ?n_2 \xrightarrow{p_2} ?n_3$	1,000,000
$?n_2 \xrightarrow{p_2} ?n_3 \xrightarrow{p_3} ?n_4$	500,000

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 [14].

Figure 1.2 shows two possible execution plans for the previous SPARQL query, which have three scan operators (one for each triple pattern) and two join operators. Each operator in the execution plans makes the partially matching fragments for the query graph pattern. For example, $Join_1$ in Figure 1.2a produces all the matching fragments for the graph pattern which consists of the second and the third triple pattern of the SPARQL query. Also, let use as-

sume that the numbers in Table 1.1 are the result cardinalities for the subgraph patterns included in the query graph pattern. Because $Join_1$ in Figure 1.2 (a) generates all the matching fragments for the graph pattern in the third row of Table 1.1 and the number of the result rows would be 500,000. However, the number of the final results(the first row in Table 1.1) is only 1,000. Consequently, at least 499,000 rows of 500,000 rows become the redundant intermediate results. The cost which are consumed for generating and processing these useless intermediate results is wasted because they do not contribute to the final query results. And in large-scale RDF dataset, the size of the intermediate results intends to increase and the overhead due to them becomes more serious.

Most RDF engines try to reduce these intermediate results by choosing an execution plan with the optimal join order when compiling the query. For example, Figure 1.2b shows another execution plan whose results are the same with the Figure 1.2a but whose join order is different from that of the execution plan inFigure 1.2a. The query optimizer prefers the execution plan in Figure 1.2a to the execution plan in Figure 1.2b because the latter would generate 500,000 more rows than the former plan. However, as we can see in this example, the execution plan with the optimal join order could not remove all the useless intermediate results.

1.2 Our Contributions

RDF Triple Filtering Framework. In this paper, we propose a novel triple filtering framework called *R3F (RDF Triple Filtering)* [17] to reduce the useless intermediate results effectively and efficiently using graph-structural informa-

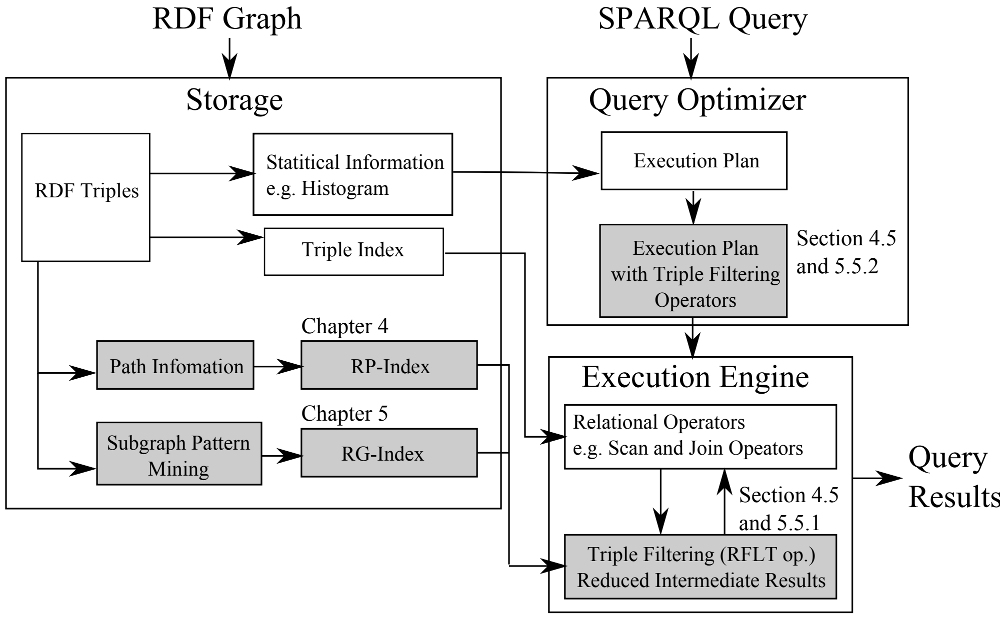


Figure 1.3: Overview of RDF Triple Filtering (R3F)

tion of RDF data. R3F adds several filtering operators to an execution plan. These operators filter out irrelevant triples using the necessary condition for becoming the final query results. This information is provide as vertex lists, which is used for the filter data for the triple filtering. This information can be provided from any information sources as long as it is a sorted list of vertices. Therefore, we call our method as a framework. In this these we provide the filter data from two indices; a path-based index and a graph-based index.

We deal with the entire process for applying the triple filtering for SPARQL query processing including (1) the building and maintaining indices, (2) query optimization and (3) query execution operators. Figure 1.3 shows the overview of the RDF store using the triple filtering. The grey boxes represent the modules for the triple filtering. The contributions of this paper can be summarized as

follows.

RP-index and RG-index. In order to provide the filter data for R3F, we propose two types of indices, *RDF Path index* RP-index [16, 17] and *RDF Graph index* RG-index. RP-index uses the path information and RG-index uses the graph-structural information. Each index provides the list of the nodes in the RDF graph which are reached by paths with a specific path pattern or are included in a specific subgraph pattern. 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.

For example, in the previous example, we can obtain the list of vertices which can be reached by paths which are matching for a path pattern $\{(?n_1, p_1, ?n_2), (?n_2, p_2, ?n_3)\}$ by using RP-index. This vertex list can be used as filter data to filter the result triples of $Scan_2$ or $Scan_3$ in Figure 1.2a and then we can prune the triples which would not be joined in $Join_2$ in advance. As a result, we can reduce the number of intermediate results using the path pattern information. Using these node lists, we can reduce the useless intermediate results effectively for complex SPARQL queries.

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 [18], 1-index [19], A(k)-index [20], D(k)-index [21], and M(k)-index [22]. Although these indices can be used for the triple filtering, 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 RP-index using the discriminative fragment concept applied in gIndex [23].

We also propose a graph index called *RG-index* in order to improve the filtering power of RP-index. RG-index indexes the graph patterns in the RDF graph rather than the path information, and therefore, it can enhance the filtering effects compared to RP-index. For building RG-index we adapt the gSpan [24] algorithm, one of the most well known algorithms for mining frequent graph patterns. Originally, gSpan was developed for treating a transactional graph database, which comprises many small-size graphs. Thus, in order to apply the gSpan to the RDF graph, which is a single large graph, the gSpan algorithm has to be modified. Further, to reduce the duplicate computations that occur during graph pattern mining, we propose a mechanism for caching the intermediate results.

In addition, we propose an efficient building and maintaining algorithms for the and RP-index and RG-index.

RFLT Operator and Query Optimization. We propose a new relational operator called the *RDF Filter* (RFLT) that conducts triple filtering very efficiently for its child operators using vertex lists from RP-index or RG-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 RFLT operator.

We implement R3F on top of RDF-3X [14], the fastest RDF engine accord-

ing to the published numbers (we discuss RDF-3X in Section **??**). 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 [12,14,25]. For efficient triple filtering, the triple filtering 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, RP-index and RG-index is designed to supply the filter data. RP-index and RG-index are a sort of supplementary index for pruning irrelevant triples retrieved from the triple indices (or aggregated indices) using the incoming predicate path information. Hence, RP-index, RG-index and the indices in RDF-3X are in a complementary relationship. Figure 1.3 shows the overview of the RDF stores using the triple filtering. We focus on the graph pattern matching component of SPARQL query processing, especially the basic graph pattern [2]. However, we also discuss how to apply our approach to other types of queries.

RDFS [26] and OWL [27] provide semantic information for RDF data, and this information can create additional triples that are not explicitly stated in the RDF data. We assume that these inferred triples materialize in the RDF database in advance using the forward chaining strategy, as in Jena [10] and Sesame [11].

Our contributions can be summarized as follows.

1. At first, we propose a novel triple filtering method for efficient SPARQL query processing. We provide the framework for processing the triple filtering.

2. For efficient and effective triple filtering, we design a path-based index called *RP-index*. Additionally, we deal with the size problem of RP-index using the discriminative and frequent fragment concept from gIndex [23], and also consider maintenance issues.

3. We also describe the design of RG-index and propose an efficient building algorithm adapted from the gSpan algorithm.

4. We present RFLT operator, which conducts triple filtering efficiently. In addition, we develop the cost model and the cardinality estimation method for RFLT operator. And we integrate this operator into the cost-based query optimizer.

5. We implement RP-index and RG-index on RDF-3X [14] and present comprehensive performance evaluation results using very large-scale real-life and synthetic RDF datasets, which demonstrate that the performance of our methods is superior to that of the existing methods.

1.3 Outline

The remainder of the thesis is organized as follows. Chapter 2 reviews the related work. In this chapter, an overview of the target RDF-3X system is also presented. Chapter 3 gives some preliminary notations and discusses the data model related to our work. Chapter 4 describes the overall process of R3F, RDF triple filtering, framework. In this chapter, we present the concept of the triple filtering and propose the triple filtering method using RP-index. We presents the design of RP-index and its building and incremental update method. We also introduce RFLT operator and discusses the generation of execution plans

using this operator. Chapter 5 proposes RG-index which can provide stronger filtering power than RP-index. In Chapter 6 concludes our work and discusses the future work.

Chapter 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. We also introduce frequent graph pattern mining techniques.

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 [15]. 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 [10] and Sesame [11] use

relational databases as their underlying stores (currently, they also provide native RDF stores [28]). 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 [12] 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 [29]. Hexastore [25] 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 [30] 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 [14] is another relation-based RDF store, that we discuss in more detail in Section 2.1.2. SWIM (Semantic Web Information Management) [31] 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 [32].

Recently, a few *graph-based* RDF stores have also been proposed. In the GRIN index [33], an RDF graph is partitioned into several subgraphs. Those relevant to a query can then be chosen by the GRIN index. DOGMA [34] 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 [35] constructs an index that summarizes the structure of an RDF graph, and processes queries using the structure index. gStore [36] 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.

These graph-based system (GRIN index [33], DOGMA [34], PIG [35], and gStore [36]) use graph-traversal approaches and graph indexing. They focus on reducing the search space of the graph traversing algorithms using the graph indices. While we also use a graph index (RG-index), our approach is different from these systems in that we focus on reducing the input size of joins in relation-based RDF stores.

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 [37]. 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 triple filtering is designed for relation-based RDF stores, and also uses a kind of graph index, RP-index. Therefore, our approaches 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.

Recently, RDF stores based on a clustered environment, such as MapReduce, have also been proposed, i.e., HadoopRDF [38], SHARD [39], multi-node extension of RDF-3X [40], and Rya [41]. In these distributed RDF systems, reducing the join inputs can improve the query performance more than do the

single-node RDF stores, because it can reduce the network overhead for transporting intermediate results. RG-index can be applied in these systems.

2.1.1 Summary of Existing Methods of Relation-based RDF Stores

As we already mentioned, the main problem of SPARQL query processing is that it involves many join operators. Several approaches have been proposed for resolving this problem, which can be summarized as: (1) Reducing the number of joins; (2) making the join operators themselves efficient; and (3) reducing the inputs of join operators. Jena [10] and Oracle [42] proposed the property table. They reduces the number of joins by clustering several properties accessed together in a single property table. Because it stores the join results in a single table, it can reduce the number of joins. However, the property table approach has several problems in that it requires the users' clustering decisions and the previous knowledge about the query workload [12]. In addition, it incurs many null values or multi-values, which are hard to process, because it is created by denormalizing the triple table [12].

In order to process the joins efficiently, SW-Store [12] proposed the vertical partitioning, in which the triple table is partitioned vertically according to the predicate values. Since it uses a column-oriented store as its underlying store, triples are stored as sorted by the subject column. Therefore, the subject-subject joins can be processed efficiently using the fast merge join in SW-Store. However, the merge joins can be used only for the subject-subject joins in SW-Store. To extend the possibilities of using merge joins, in Hexastore [25] and RDF-3X [14], the multiple indexing approach is applied. They index triples by all six possible orderings of (subject, predicate, object). Triples can be retrieved

Table 2.1: Summary of Existing Approaches and Our Approaches

Approaches	System	Pros	Cons
Property Table	Jena, Sesame, Oracle	Reduce the joins	Need user's decision
Vertical Partitioning	SW-Store	Fast merge joins	Applied only Subject-Subject join
Multiple Indexing	RDF-3X	Handle any type of triple patterns	Space overhead
Triple Filtering	RP-index, RG-index	Reduce the redundant intermediate results	Overhead for maintaining the indices

for any orderings, and merge joins can be used for joins other than the subject-subject join.

U-SIP (Ubiquitous Sideways Information Passing) [43] is proposed for reducing the inputs of join operators. U-SIP dynamically builds filters to provide information about the subject IDs or object IDs to be read next (we call this the next information). RDF-3X uses this next information to skip reading unnecessary disk blocks. While scanning the leaf blocks sequentially, if it determines that the next block can be skipped, it performs the B+tree index look-up to skip unnecessary blocks. In these ways, U-SIP can prune the triples that are irrelevant for the given query and reduce the input size of joins.

In short, most relation-based RDF stores mainly use join operations, and

they proposed several techniques for processing the join operations efficiently. However, these previous approach do not use graph-structural information of the RDF graph. Table 2.1 shows the summary of the existing approaches and our approaches.

2.1.2 Overview of RDF-3X

RDF-3X [14] 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 5.1b. $Scan_1$ 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 [14].

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) [44–46]. 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 [47]. Following this approach, several selectivity estimation techniques for SPARQL query processing have also been proposed [48,49]. In RDF-3X, several specialized histograms for RDF are used [14, 43, 50]. 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 [44–46], 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 triple filtering method exploits the graph-structural information, and can there-

fore 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 [18], 1-index [19], A(k)-index [20], D(k)-index [21], and M(k)-index [22] (cf. [51, 52] 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 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, 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 [23]. Thus, 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) [53]. 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 [53]. 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 [54], gIndex [23]). 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 [55], GraphQL [56], GADDI [57], and SPath [58]. 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. For example, SPath index provides vertex list which are adjacent to the currently traversing vertex and at the same time have specified features. So it is specialized to the graph traversing algorithm. Also, to our best knowledge, it is first attempt to index the graph pattern directly and provide the vertex lists

for all vertices in the graph pattern.

2.4 Frequent Graph Pattern Mining

There exist numerous bodies of literature focused on frequent graph pattern mining (cf. [59] for detailed surveys). There are also two problem formulations for graph mining [53] like the graph indexing, which is described in the previous section: the graph-transaction setting and the single-graph setting. The graph-transaction setting has drawn more attention than the single-graph setting.

A frequent graph pattern mining algorithm first generates the candidate graph patterns, and then checks that its support is larger than the minimum support. If this condition is satisfied, the pattern is included in the results. The main focuses of the designers of frequent graph pattern mining algorithms are how to generate candidate graph patterns without generating duplicate patterns and how to prune infrequent patterns efficiently. To achieve these goals, they exploit the a-priori principle [24, 53, 59], and canonical labeling mechanisms for representing the graph patterns are proposed.

We adapt the gSpan [24] algorithm to build RG-index. It uses the DFS codes [24] as the canonical representation of the candidate graph patterns and the depth-first manner of pattern generation. We discuss gSpan in more detail in Section 5.4.1. Figure 2.1 shows the overall process of the graph mining algorithm.

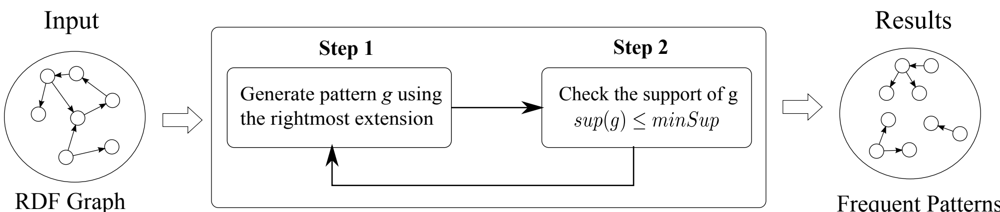


Figure 2.1: Frequent Pattern Mining

Chapter 3

Preliminaries

In this chapter, we formally define the RDF and SPARQL model, and present some notations.

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 [2]. A basic graph pattern is a set of conjunctive triple patterns, which means its results should be matched to all triple patterns [2]. It should be noted that in our model the joins that have predicate variables are not considered, because this join type is rarely used. In addition, various features of the RDF and SPARQL are omitted for simplicity. For example, some features of the RDF, such as blank

nodes and data types, are not considered. We focus on SPARQL queries with basic graph patterns. A basic graph pattern is a set of triple patterns [2]. Optional graph patterns and union graph patterns are not considered. However, our approaches can be applied to queries having these features with minor modifications. (we will discuss this issue in Section 4.3).

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 $tp(s,p,o) \in (U \cup \mathit{VAR}) \times U \times (U \cup L \cup \mathit{VAR})$ (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. Figure 5.1a and Figure 3.1 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. An RDF graph is defined as follows.

Definition 3.1.1 [RDF Graph] We define an RDF graph for the RDF database D as $G_D = (V_D, E_D, L_D)$, where V_D is a set of vertices corresponding to the subjects and objects of all triples in D ($V_D \subseteq (U \cup L)$), E_D is a set of directed edges corresponding to all triples that are from the subjects to the objects, and L_D is an edge-label mapping, $L_D : E_D \rightarrow P_D$, such that $t(s,p,o) \in D$, $L_D(s,o) = p$.

The vertices in an RDF graph correspond to URIs or literals. It should be noted that URIs or literals are not considered vertex labels; rather, they are unique identifiers for vertices. As in the RDF graph, the vertices in the query graph are identified by the variable names, URIs or literals. Therefore, both G_D and G_Q are edge-labeled directed graphs. We define a query graph as follows.

Definition 3.1.2 [Query Graph] A query graph for a SPARQL query Q is defined as $G_Q = (V_Q, E_Q, L_Q)$, where V_Q is a set of vertices corresponding to the subjects and objects of all triple patterns in Q ($V_Q \subseteq (U \cup L \cup VAR)$), E_Q is a set of directed edges corresponding to all triples that are from the subjects to the objects, and L_Q is an edge-label mapping, $L_Q : E_Q \rightarrow P_D$, such that $tp(s,p,o) \in Q$, $L_Q(s,o) = p$.

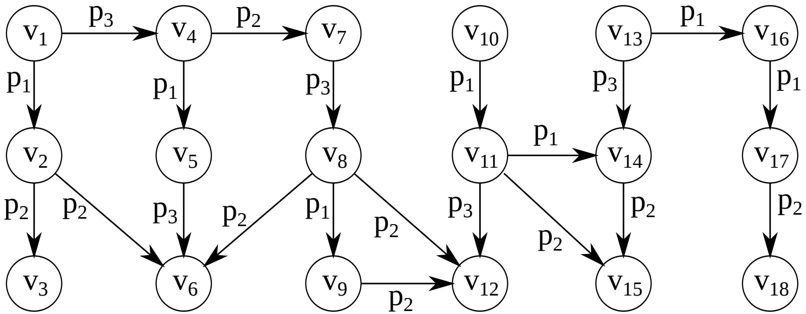


Figure 3.1: RDF Graph

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 3.1.3 [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 3.1.4 [SPARQL Query Answer] For the RDF graph in Figure 3.1, the answer set of the SPARQL query in Figure 5.1a 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 Path and Graph Pattern

In this section, we present two kinds of patterns: incoming predicate path and k- neightborhood subgraph. These two types of patterns form a base patterns of RP-index and RG-index, and are used the necessary condition for the triple filtering.

3.2.1 Incoming Predicate Path

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

Definition 3.2.1 [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 3.2.2 [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 $\mathit{InPPath}(v)$. When the maximal path length *maxL* is given, a variant of the notation, $\mathit{InPPath}(v, \mathit{maxL})$, is used to denote a subset of $\mathit{InPPath}(v)$, such that $\mathit{InPPath}(v, \mathit{maxL}) = \{\mathit{ppath} \mid \mathit{ppath} \in \mathit{InPPath}(v) \wedge |\mathit{ppath}| \leq \mathit{maxL}\}$.

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

Example 3.2.3 [Incoming Predicate Path] For the RDF graph in Figure 3.1, the incoming path set of v_{12} with maximum length 3 is $\mathit{InPPath}(v_{12}, 3) = \{\langle p_2 \rangle,$

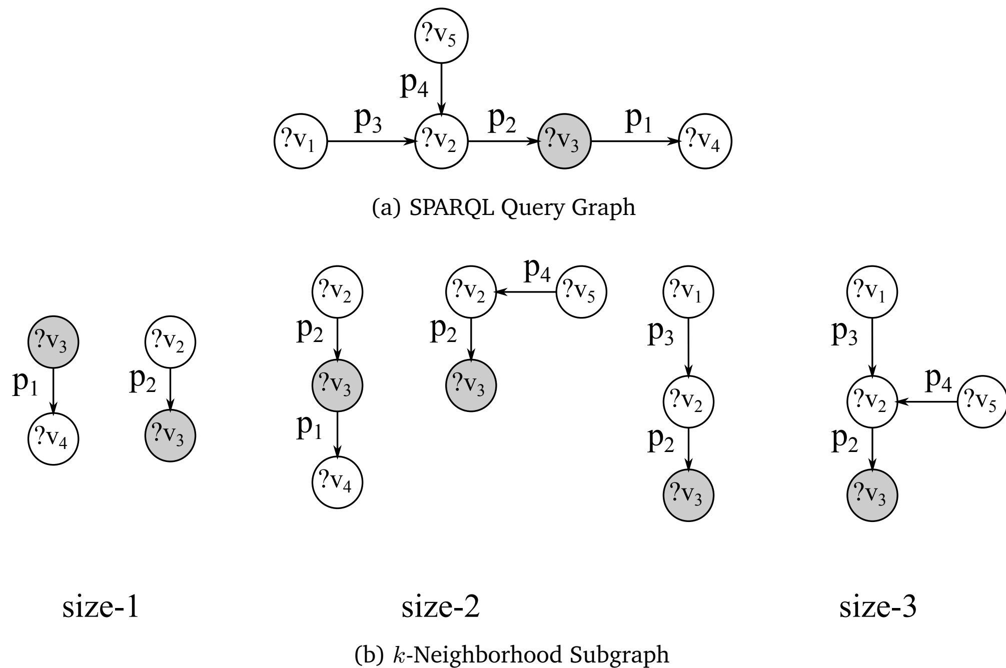


Figure 3.2: RDF Graph and k-Neighborhood Subgraph

$\langle p_3 \rangle$, $\langle p_1, p_2 \rangle$, $\langle p_3, p_2 \rangle$, $\langle p_1, p_3 \rangle$, $\langle p_2, p_3, p_2 \rangle$, $\langle p_3, p_1, p_2 \rangle\}$. For the SPARQL query graph in Figure 5.1a, *InPPath*$(?v_3, 3) = \{\langle p_1, p_2 \rangle, \langle p_3, p_2 \rangle\}$.

3.2.2 k-neighborhood Subgraph

We define the k-neighborhood subgraph as follows.

Definition 3.2.4 [k-Neighborhood Subgraph] Given a vertex v in a graph G, the k-neighborhood subgraph, denoted by $N(v, k)$, is a set of subgraphs that contain v and whose size is no more than k.

The k-neighborhood subgraph is applied to both the RDF graph and the query graphs. Let us consider a query graph in Figure 3.2a. Then, the graphs in

Figure 3.2b are $N(?v3, 3)$.

3.3 Candidate Vertex Set

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

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 $\mathit{Ans}(Q, v)$. At first, we define the candidate vertex set using the incoming predicate path as follows.

Definition 3.3.1 [Candidate Vertex Set using Incoming Predicate Paths] 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 $C_{\mathit{InPPath}}(v, \mathit{maxL}) = \{v_g \mid v_g \in V_D \wedge \mathit{InPPath}(v, \mathit{maxL}) \subseteq \mathit{InPPath}(v_g, \mathit{maxL})\}$.

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

Proof: We prove that if vertex $v_D \in G_D$ is in $\mathit{Ans}(Q, v)$, v_D must have all incoming predicate paths of v. That is, $\forall v_D \in \mathit{Ans}(Q, v)$, $\mathit{InPPath}(v, \mathit{maxL}) \subseteq \mathit{InPPath}(v_D, \mathit{maxL})$. If $v_D \in \mathit{Ans}(Q, v)$, there exists a substitution $\theta \in \mathit{Ans}(Q)$ 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, $\langle e_1, \ldots, e_n \rangle$ ($n \leq \mathit{maxL}$) 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, $\forall v_D \in \mathit{Ans}(Q, v), \mathit{InPPath}(v, \mathit{maxL}) \subseteq \mathit{InPPath}(v_D, \mathit{maxL})$; that is, all $v_D \in \mathit{Ans}(Q, v)$ contain all incoming predicate paths of v, and $\mathit{Ans}(Q, v) \subseteq C_{\mathit{InPPath}}(v, \mathit{maxL})$. ∎

Example 3.3.3 [Candidate Vertex Set using Incoming Predicate Path] The candidate vertex for $?v_3$ in Figure 5.1a 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 3.1, there are three vertices that have these incoming predicate paths, so $C_{\mathit{InPPath}}(?v_3, 2) = \{v_6, v_{12}, v_{15}\}$. We can see that $\mathit{Ans}(Q, ?v_3) = \{v_6, v_{12}, v_{15}\} \subset C_{\mathit{InPPath}}(?v_3, 2)$ (i.e., satisfying the condition for the candidate vertex set).

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

Definition 3.3.4 [Candidate Vertex Set using k-neighborhood] Given a vertex v in a query graph G_Q and *maxL*, the candidate vertex set using the subgraph patterns, denoted by $CV(v, \mathit{maxL})$, is a set of data vertices whose k-neighborhood

subgraphs are the same as $N(v, \mathit{maxL})$.

We can also prove that $CV(v, \mathit{maxL})$ satisfies the condition for the candidate vertex set, i.e $\mathit{Ans}(Q, v) \subseteq CV(v, \mathit{maxL})$. However, for simplicity, we omit the proof. RP-index and RG-index filter out irrelevant triples using C_{InPPath} and $CV(v, \mathit{maxL})$, respectively.

Chapter 4

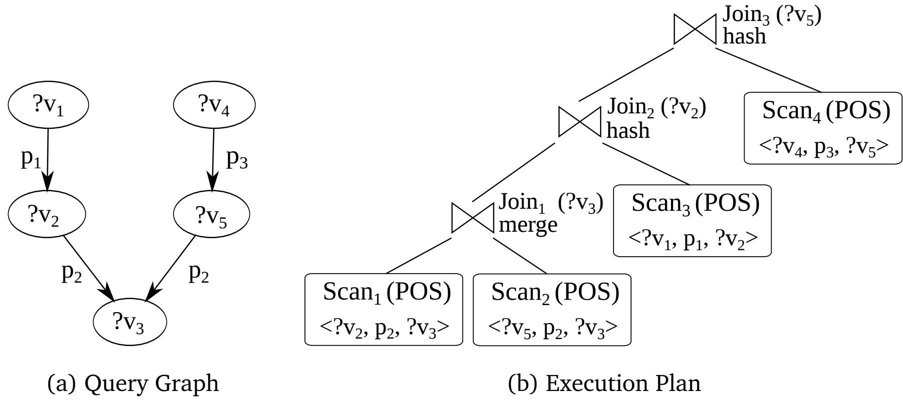
R3F: RDF Triple Filtering Framework using RP-index

In this section, we present an overview of the triple filtering framework, R3F, and discuss the design of RP-index. We present a logical description of RP-index in Section 4.3, and discuss its physical implementation in Section 4.3.1.

4.1 Motivating Example

Let us consider another example of a SPARQL query and its execution plan in RDF-3X, as shown in Figure 4.1 (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 $\langle ?v_5, p_2, ?v_3\rangle$.

However, this execution plan has a problem that it can generate redundant intermediate results. That is, the operators in this execution plan can generate



(a) Query Graph (b) Execution Plan

Figure 4.1: SPARQL Query Graph and Execution Plan

useless subgraphs which are not included in the final results. In this example, not all 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 chapter, we propose the triple filtering method called *RDF Triple Filtering framework* (R3F). R3F can use any pattern information for the triple filetering. In this chapter, we propose the triple filtering using the path information. We check the relevance of triples for a particular query using *incoming predicate path* information. Consider, for example, vertex $?v_3$ in Figure 5.1a,

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, we can filter out irrelevant triples, and consequently reduce redundant intermediate results.

4.2 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 $C_{\mathit{InPPath}}(?v_S, \mathit{maxL})$ or $C_{\mathit{InPPath}}(?v_O, \mathit{maxL})$, 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 RP-index and RFLT operator. RP-index is designed to provide C_{InPPath} efficiently, and is presented in Section 4.3. 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 RFLT operators. We will discuss the *RFLT* operator and the query optimization method in Section 4.4.

4.3 RP-index Definition

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 4.3.1 [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., $Vlist(ppath) = \{v \in V_D \mid ppath \in InPPath(v)\}$.

RP-index for D is defined as follows.

Definition 4.3.2 [RP-index] Given the RDF database D, 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 4.3.3 [RP-index] Figure 4.2 shows the Vlists in $RP\text{-}index(D, 3)$ for D in Figure 3.1 with $maxL = 3$. There are 15 Vlists in $RP\text{-}index(D, 3)$.

We introduce *maxL* to limit the size of RP-index. As *maxL* increases, the number of predicate paths in RP-index increases and, as a result, the quality of the triple filtering can be improved. However, the space overhead of RP-index also increases. In other words, there is a tradeoff between the quality of the triple filtering and the space overhead of RP-index. This tradeoff can be adjusted by *maxL* (we also use another method to address the size problem of RP-index, discussed in Section 5.3.1).

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

Figure 4.2: Vlists in RP-index$(D, 3)$

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

4.3.1 Physical Structure of 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 2.1.2). 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 [14] (see Section 2.1.2). 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 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 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 4.3 shows the trie for RP-index($D, 3$) in Figure 4.2. 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 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.

4.3.2 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 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

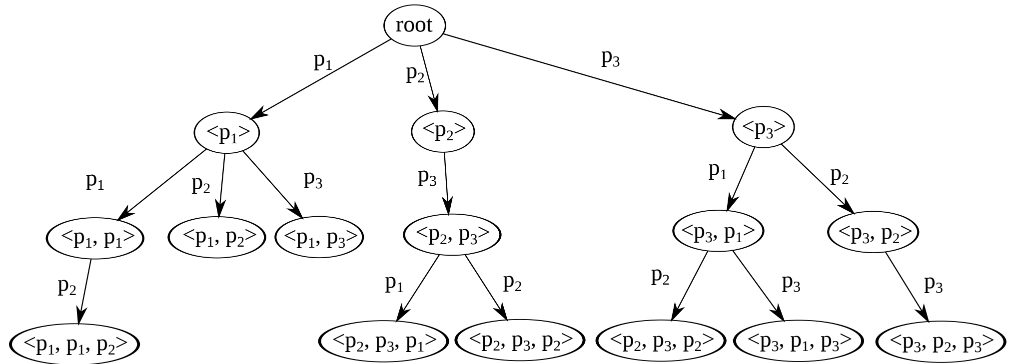


Figure 4.3: A Trie for Predicate Paths

concept in gIndex [23].

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 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 4.3.4 [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 4.3.5 [Discriminative Predicate Path] For RP-index in Figure 4.2, 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 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 4.3.6 [Frequent Predicate Path] Given a size-increasing function $\psi(l)$, predicate path *ppath* is frequent if and only if $|Vlist(ppath)| \geq \psi(|ppath|)$.

4.3.3 Reverse Predicate

Because the triple filtering 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 5.6 has no incoming predicate path, and so

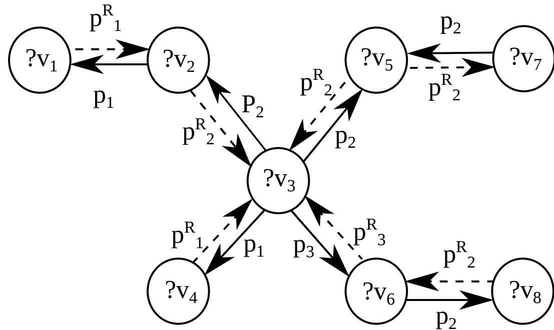


Figure 4.4: Extended SPARQL Query

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 4.3.7 [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 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 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 (*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 5.6 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 5.6 the dashed edges denote those with reverse predicates. Considering the reverse predicates, *InPPath*($?v_3$) $= \{\langle p_2^R \rangle, \langle p_1^R, p_2^R \rangle, \langle p_1^R \rangle, \langle p_3^R \rangle, \langle p_2^R, p_3^R \rangle\}$.

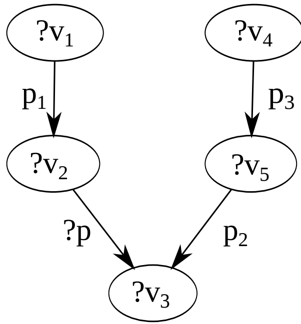


Figure 4.5: A SPARQL Query with a Predicate Variable

4.3.4 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, the triple filtering 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 4.5. This query has one edge with a predicate variable $?p$. If we exclude this edge when 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 RP-index, the predicate paths containing this variable predicate also need to be indexed. When generating incoming predi-

cate 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 the triple filtering 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.

4.3.5 Determining RP-index Parameters

Until now, we have only discussed the design of RP-index. In this section, we discuss its tuning issues. 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 RP-index. It is important to make RP-index as small as possible while maintaining its filtering power. The size of 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 5.6. From our experience, *maxL* $= 3$ is sufficient in most cases.

Although we use small *maxL*, it is still possible for 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 5.6. 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 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 5.3.1. They can reduce the size of 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 RP-index.

When RP-index does not have some necessary predicate paths that users can identify, it is possible to add such paths to 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 RP-index can be effectively controlled while retaining its filtering power. We see the effects of the parameters in the experimental results (Section 4.7.2).

4.4 Processing Triple Filtering

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

4.4.1 RFLT Operator

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 effi-

ciently 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 4.4.1 [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 4.4.2 [Sortkey] $Scan_1$ in Figure 5.1b 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, RFLT operator conducts triple filtering for its child scan operator using their sortkey vertices. The query optimizer indicates to RFLT operator which predicate paths it should use for triple filtering as follows. 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 $Scan_i$. 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

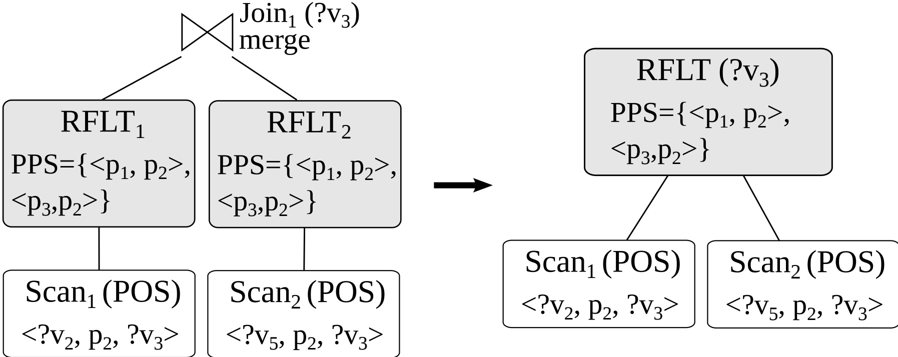


Figure 4.6: Execution Plan using RFLT

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 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 RFLT operator to process merge join operations and triple filtering at the same time.

Figure 4.6 shows part of the execution plan using RFLT operators for the query in Figure 5.1a. The predicate path set (PPS) in 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 $Scan_1$

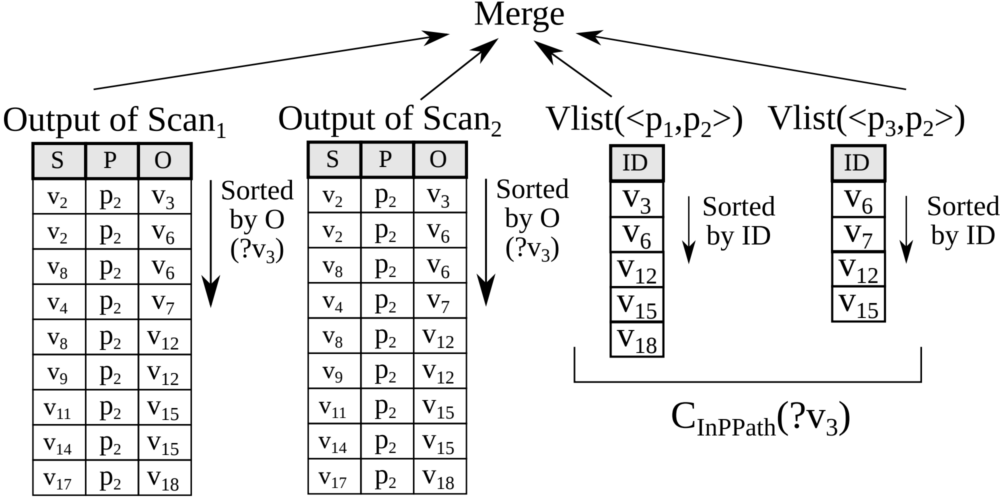


Figure 4.7: RFLT Operator

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 4.7 illustrates the filtering process of 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.

$$\text{I/O cost: } O\left(\sum_{p \in PPS} \|Vlist(p)\|\right) \tag{4.1}$$

$$\text{CPU cost: } O\left(\sum_{scan \in ChildOP} |scan| + \sum_{p \in PPS} |Vlist(p)|\right) \tag{4.2}$$

where $\|vlist\|$ is the number of blocks of *vlist*, PPS is a set of predicate paths assigned for 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 RFLT operator is very efficient and lightweight. In Section 4.7.2, we compare the size of Vlist and the input triples.

Implementation of RFLT Operator

We have implemented our RFLT operator in RDF-3X. RDF-3X adapts the iterator model of the query execution [60], 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 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 RFLT operator are the joined results of child input operators that pass the triple filtering. In order to conduct triple filtering, 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 RFLT operator could generate the results and conduct the triple filtering simultaneously by performing only the N-way merge joins.

RFLT Operator and U-SIP

RDF-3X exploits a type of SIP technique called *U-SIP* (see Section 2.1.2). In U-SIP, a scan operator can skip the reading of irrelevant blocks by utilizing the next information provided by other scan operators. With the triple filtering, 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 4.8. This figure illustrates the POS index for $Scan_1$ to read, and the filter data of 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 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, the triple filtering can provide the next information for scan operators. Consequently, the triple filtering and U-SIP can utilize synergy effects.

4.5 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 [14]. In order to make a query optimizer that considers triple filtering, we need to provide the query optimizer with (1) the cost function of RFLT operator, which was given in the

Figure 4.8: RFLT Operator and U-SIP

previous section, and (2) the estimated cardinalities of 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 RP-index. In addition, we form another statistic similar to the characteristics set [50]. 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 3.1, $S_C(v_{14}) = \{p_1, p_3, p_2^R\}$. The number of vertices which have the characteristics set S is called the occurrence count [50] 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 [50].

4.5.1 Filtering Effect of Vlists

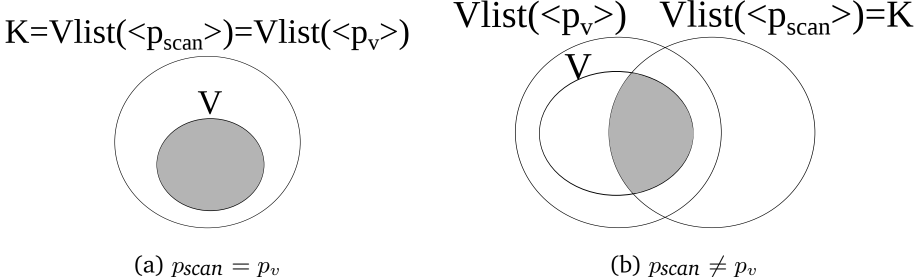
We define the filtering effect of Vlist V for $Scan_i$, $E(Scan_i, V)$, as the fraction of the remaining values of the sortkey column after filtering. Let us denote the sortkey column of $Scan_i$ and the set of its distinct values as K, interchangeably. Then, $E(Scan_i, V)$ can be represented as follows:

$$E(Scan_i, V) = |V \cap K|/|K|. \tag{4.3}$$

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 Eq. (4.3) can also be estimated using Vlists. Figure 4.9 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 $Vlist(\langle p_v \rangle) \cap$



(a) $pscan = p_v$

(b) $pscan \neq p_v$

Figure 4.9: Filtering Effect

$Vlist(\langle p_{scan}\rangle)$ 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, $count(\{p_v, p_{scan}\})$.

4.5.2 Cardinality of 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 RFLT operator can be estimated as follows.

$$|RFLT| = |Scan_i| \times |C \cap K|/|K| \tag{4.4}$$

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 [61] for estimating this set, they require some additional operations, such as sampling. In this case, we take a rather simple ap-

proach by using the upper bound of $|\mathit{RFLT}|$ as the estimated value. This means that we conservatively underestimate the effect of triple filtering. The upper bound can be estimated as $|C \cap K| \leq \min\left(\min_{\mathit{path} \in \mathit{PPS}} |\mathit{Vlist}(\mathit{ppath})|, |K|\right)$, 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 the filtered triples, and assume that the values are distributed uniformly, the output cardinality can be estimated as follows:

$$|\mathit{RFLT}| = |J| \times \prod_{\mathit{Scan}_i \in \mathit{ChildOP}} |\mathit{Scan}_i| / |K_i| \tag{4.5}$$

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 $J = \left(\bigcap_{p \in P_s} \mathit{Vlist}(\langle p \rangle)\right) \cap \left(\bigcap_{\mathit{ppath} \in \mathit{PPS}} \mathit{Vlist}(\mathit{ppath})\right)$, 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 $|\bigcap_{p \in P_s} \mathit{Vlist}(\langle p \rangle)|$ from the characteristics set $U_1 = \mathit{count}(P_s)$. Also, we define $U_2 = \min_{\mathit{ppath} \in \mathit{PPS}} |\mathit{Vlist}(p)|$. 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 4.7.2.

Our method is very similar to the Characteristic Set [50], 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

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]
```

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.

4.5.3 Generating an Execution Plan

The query optimization of RDF-3X is based on the bottom-up dynamic-programming (DP) framework [14]. 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 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 Eq. (4.3). 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 4.6). 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 information for estimating the output cardinalities which is resident in memory (in getCost function).

4.6 RP-index Building

In this section, we present the method of building RP-index.

Building 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

Algorithm 2 Operator Build Functions

procedure buildScan (tp)
1: $P \leftarrow$ a set of all possible scan operators for tp
2: **for** $\forall p \in P$ **do**
3: $p \leftarrow$ buildRFLT(p)
4: **end for**
5: **return** P

procedure buildJoin (p_1, p_2)
1: $P \leftarrow$ a set of all possible join plans for p_1 and p_2
2: **for** $\forall p \in P$ **do**
3: $p \leftarrow$ buildRFLT(p)
4: **end for**
5: **return** P

procedure buildRFLT (p)
1: $op \leftarrow$ the root operator of p
2: **if** op is scan operator **then**
3: $v \leftarrow$ the sortkey vertex of op;
4: rootOp.ChildOP $\leftarrow$ {op};
5: rootOp.PPS $\leftarrow$ getEffectivePPath(InPPath($v, maxL$), scan.predicate);
6: rootOp.Cost $\leftarrow$ getCost(rootOp);
7: **return** rootOp
8: **else if** op is merge join operator **then**
9: $v \leftarrow$ the sortkey vertex of op;
10: rootOp.ChildOP $\leftarrow \bigcup_{c \in p.ChildOP}$ c.ChildOP;
11: rootOp.PPS $\leftarrow \bigcap_{c \in p.ChildOP}$ c.PPS;
12: rootOp.Cost $\leftarrow$ getCost(rootOp);
13: **return** rootOp
14: **end if**

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:

$$Vlist(ppath) = \rho_{ID}\left(\Pi_O\left(Vlist(ppath_{pre}) \rtimes_{ID=S} D(p)\right)\right) \quad (4.6)$$

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 5.3.1, we only store Vlists for discriminative and frequent predicate paths in an attempt to address the size problem of RP-index. Due to this, there are some cases where it is impossible to build Vlists using Eq. (4.6). For example, if $Vlist(\langle p_1, p_2\rangle)$ is infrequent, then we cannot use Eq. (4.6) to build $Vlist(\langle p_1, p_2, p_3\rangle)$ because $Vlist(\langle p_1, p_2\rangle)$ is not stored in 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:

$$|Vlist(ppath)| \leq |Vlist(ppath_{suf})| \quad (4.7)$$

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 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

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($\langle\rangle$, *PQ*) /* enqueue an empty predicate path in *PQ* */
8: **while** $PQ \neq \emptyset$ **do**
9:     $ppath_{pre} \leftarrow$ dequeue(*PQ*)
10:     **for each** $p \in P_D$ **do**
11:         *ppath* $\leftarrow$ append p to $ppath_{pre}$
12:         **if** *Vlist*$(ppath)$ can be skipped (Eq. (4.7)) **then**
13:             continue;
14:         **end if**
15:         **if** isUpdate **then**
16:             *vlist* $\leftarrow$ UpdateVlist($ppath$); /* Incremental update (Table 4.1) */
17:         **else**
18:             *vlist* $\leftarrow$ CreateVlist($ppath$); /* Build using Eq. (4.6) */
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**

a size-$(l-1)$ predicate path in PQ. For each generated predicate path $ppath$, the pruning condition (Eq. (4.7)) 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 Eq. (4.6). 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 RP-index (lines 20–28).

4.6.1 Complexity of building RP-index

At first, Vlists for 1-length predicate paths are created by reading the whole database. The number of all triples are represented by $|D|$. And then, Vlists for n-length predicate paths are built using Vlists $n-1$-length predicate paths by joining them with the triples with all predicates, which amortized to all triples in the database. The maximum size of the Vlists is $|R|$, which is the number of the resources in D. So we can represent the complexity of the building n-length predicate paths by $|P|^n \times |R| \times |D|$, where $|P|$ is the number of the predicates. Hence, the complexity of the building RP-index can be represented by $O\left(|D| + \sum_{n=1}^{maxL-1} |P|^n \times |R| \times |D|\right)$. This is a worst case complexity for building RP-index. In practice, the cost can be reduced by using several parameters and using the parallel building method described in the next section.

4.6.2 Parallel Building Methods

Previously, we presented algorithms for building RP-index. However, these algorithms can be very time-consuming because there can be a large number of pattern in the RDF graph. Even though we limit *MaxL* and the other parameters, it can take too long time especially for large-scale RDF graphs. Therefore, in this section we present the parallel algorithm for building RP-index.

The basic idea of parallelization is to decompose a job into a number of small pieces of the job which can be performed independently and simultane-

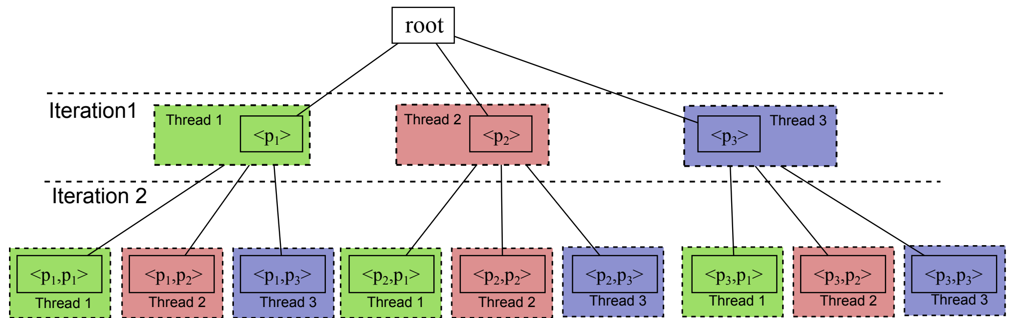


Figure 4.10: Parallel Building of RP-index

ously. Let us look at the example in Figure 4.10, which illustrate the parallel building of RP-index. In this example, the set of predicates is $\{p_1, p_2, p_3\}$. And it shows two iterations for building 1-length predicate paths and 2-length predicate paths. Note that the root node in RP-index does not have a predicate path and a Vlist. As we already mentioned, the building process is performed in a breadth-first fashion. The idea is that in a iteration, building each Vlists is independent for each other; that is, building *Vlist*$(\langle p_1, p_1\rangle)$ has nothing to do with building *Vlist*$(\langle p_1, p_2\rangle)$ as long as there is already *Vlist*$(\langle p_1)$. Hence, we can build *Vlist*$(\langle p_1, p_1\rangle)$ and *Vlist*$(\langle p_1, p_2\rangle)$ simultaneously by using two threads (actually, we use pthread APIs to implement builder of RP-index). There is no contentions between these building threads and they read *Vlist*$(\langle p_1)$ at the same time. In this way, we can parallelize the building process of RP-index.

We can further optimize the building algorithm. If we make *Vlist*$(\langle p_1, p_2\rangle)$, *Vlist*$(\langle p_2, p_2\rangle)$ and *Vlist*$(\langle p_3, p_2\rangle)$ separately, the triples whose predicate p_2 should be read three times. It wastes disk I/O because the same contents are read several times. This could be more serious when there exist more predicate paths. In order to avoid these wasteful reading of triples, we build several Vlists which

have the same last predicate at same time. For example, we can read the triples of the predicate p_2 only one time, while we build three Vlists at the same time; *Vlist*$(\langle p_1, p_2\rangle)$, *Vlist*$(\langle p_2, p_2\rangle)$ and *Vlist*$(\langle p_3, p_2\rangle)$. The colored box in Figure 4.10 shows this idea. Each node represents one Vlist for the specified predicate path. And the dashed boxes represent the threads which build the Vlists and the color of boxes shows the Vlist which a thread builds at the same time. For example *Vlist*$(\langle p_1, p_2\rangle)$, *Vlist*$(\langle p_2, p_2\rangle)$ and *Vlist*$(\langle p_3, p_2\rangle)$ is colored as red and assigned to 'Thread 2'. Of course, the building process does not handle several iteration. The parallelization is performed at the level of each iteration. In this way, we can reduce disk I/O and accelerate the building process by using parallelization.

4.6.3 Incremental Maintenance

In order to ensure the correctness of query results, RP-index should be consistent with the RDF database and updated concurrently. The easiest way to obtain the newest version of 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 RP-index.

We assume that RDF applications have read-mostly workloads in which the updates for RDF stores are usually batched [14]. 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 from Eq. (4.6), *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 4.1 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 4.1, '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 according to Table 4.1. 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 RP-index. For example, RP-index can be updated in the background, while accepting user-

Table 4.1: 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*$(ppath_{pre})$

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 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.

Recently, x-RDF-3X [62] proposed the update method for high speed updates. It uses differential indices in main memory which process online update fast, and when the storage for them is full, it is merged to the triple indices resident in disks. We can use this architecture in order to support the update of

RP-index. We can defer the update of RP-index and do not apply the triple filtering for scan operators for the differential indices. In this way, we can support the OLTP workload.

4.7 Experimental Results

We have implemented the triple filtering on top of the open-source RDF-3X system (version 0.3.6)[1]. The triple filtering was written in C++ and compiled with g++ with the -O3 flag for the experiments. Implementation includes RFLT operator, extension of the query optimizer, and 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) [63], Lehigh University Benchmark (LUBM) [64], Social Network Intelligence Benchmark[2] (SNIB), Yet Another Great Ontology 2 (YAGO2) [6], and SPARQL Performance Benchmark (SP2B) [65]. DBSPB is a synthetic dataset, but it simulates the data distribution of DBpedia [66] and has the characteristics of a real-world dataset [63]. LUBM is a benchmark dataset whose domain is the university, and SNIB is another synthetic dataset whose domain is a social network site. YAGO2 is a knowledge-base derived from Wikipedia[3], WordNet [67], and GeoNames[4], and SP2B is a benchmark that simulates the DBLP scenario[5].

---

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

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

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

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

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

Table 4.2: Statistics about Datasets

	Predicates	URIs	Literals	Triples	RDF-3X Size
DBSPB	39,675	38,402,797	46,195,618	278,913,738	25 GB
LUBM	18	217,007,404	111,618,881	1,334,681,192	77 GB
SNIB	44	35,199,091	12,508,290	387,606,173	17 GB
YAGO2	93	6,872,931	22,452,390	195,048,649	9 GB
SP2B	77	177,272,798	348,388,613	931,696,802	123 GB

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 5.2. 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.

Table 4.3 show the graph density for each datasets. Because literals are terminal vertices and it does not have multiple parents, we are interested in the graphs with resources. So, we calculated the density with all resources and literals, and the densities with all resources, respectively. The densities calculated as $|E|/|V| \times |V - 1|$, where E is the set of all edges and V is the set of all vertices. The densities show that the graphs are very sparse and the densities with resources are more denser.

Table 4.3: Graph Densities of Datasets

	Graph Density	Graph Density (without Literals)
DBSPB	$3.89 \times e - 08$	$9.64 \times e - 08$
LUBM	$1.23 \times e - 08$	$1.89 \times e - 08$
SNIB	$1.70 \times e - 07$	$1.64 \times e - 07$
YAGO2	$2.26 \times e - 07$	$1.76 \times e - 06$
SP2B	$3.37 \times e - 09$	$1.46 \times e - 08$

Table 4.4: 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

4.7.1 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 4.4 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 1000 for DBSPB, SNIB and SP2B, and 10000 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.

Table 4.5: 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 4.5 and Table 4.6 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 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 RP-index effectively by storing only Vlists for the discriminative and frequent predicate paths (Setting 3).

RP-index with the Predicate Variable

We propose some methods for handling queries with predicate variables in Section 4.3.4, one of which is to index the predicate variables when building RP-index. In this section, we discuss the effects of indexing the predicate variable

Table 4.6: 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 4.7: RP-index with the Predicate Variables (Setting 3)

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

on the size of RP-index. We built RP-index with the predicate variables for the LUBM and YAGO2 datasets, using Setting 3 for the building parameters.

Table 4.7 shows the size of 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 RP-index and the number of Vlists. However, if we adjust the parameters of RP-index appropriately, we expect to 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.

4.7.2 Query Evaluation Performance

In this section, we present the query performance of the triple filtering 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 4.11 shows the execution times (for DBSPB, Setting 2 is not included because it grows too large that it could not be built). In this figure, we can see that the triple filtering reduces the execution time of most queries. In particular, there are some queries for which the triple filtering 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 the triple filtering 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 the triple filtering is not very effective. These queries do not have the selective path patterns that the triple filtering 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 5.3.1 do not harm the filtering power of 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.

Also note that there are some cases which the performance of Setting 3 is slightly better than that of Setting 2 (e.g. Q3 and Q4 of LUBM, Q1 and Q2 of SNIB, Q2 of YAGO2, Q2 of SP2B). This is because Setting 3 removes some Vlists, the queries are applied less Vlists. For example, Q4 of LUBM uses Vlists whose size is total 142 MB in Setting 2 and 16 MB in Setting 3. If the filtering power does not degrades, the reduced size of Vlists can improve the overall query performance.

Figure 4.12 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 the triple filtering is effective.

Filter Usage

Table 4.8 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 the triple filtering 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

Table 4.8: Filter Usage (SNIB)

Setting	Query	Path Length			Reverse	Vlist Size (MB)	Data Size (MB)
		1	2	3			
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

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

Path Query

In order to evaluate the triple filtering 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 also evaluated the effects of user-defined parameters of

RP-index (*maxL*, γ, and $\psi(l)$) using these queries.

Figure 4.13 shows the average execution time for each path length. We can see that path queries are processed more efficiently using the triple filtering. The results are similar to those in the previous experiments using the test queries. RP-index under Setting 2 is most effective, and 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. 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 the triple filtering 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. 4.14, 4.15, and 4.16 show the effect of the three RP-index parameters on its performance and size. In Figure 4.14, we decrease the discriminative ratio γ with fixed *maxL* $= 3$ and $\psi(l) = 0$. From Figure 4.14a, 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 4.14b). In Figure 4.15, 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 RP-index. Figure 4.16 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 RP-index increases exponentially. Therefore, as we discussed in Section 4.3.5, we do not need a large *maxL* value.

Accuracy of Cardinality Estimation

In this section, we study the accuracy of the cardinality estimation technique discussed in Section 4.5. We calculate the q-error $\max(c/\hat{c}, \hat{c}/c)$ [68], where c is the real cardinality and $\hat{c}$ is the estimated cardinality. This is the method used in [50] to evaluate estimation techniques. In Section 4.5, we need to estimate the intersection of Vlists and the input sortkey columns, and for this, we use the upper bound of the intersection. Table 4.9 and Table 4.10 show the q-errors for the experimental queries. The q-errors in Table 4.9 are calculated using the upper bound, as in Section 4.5, and those in Table 4.10 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 4.10, 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.

Table 4.9: 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

Table 4.10: 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

4.7.3 Incremental Maintenance of RP-index

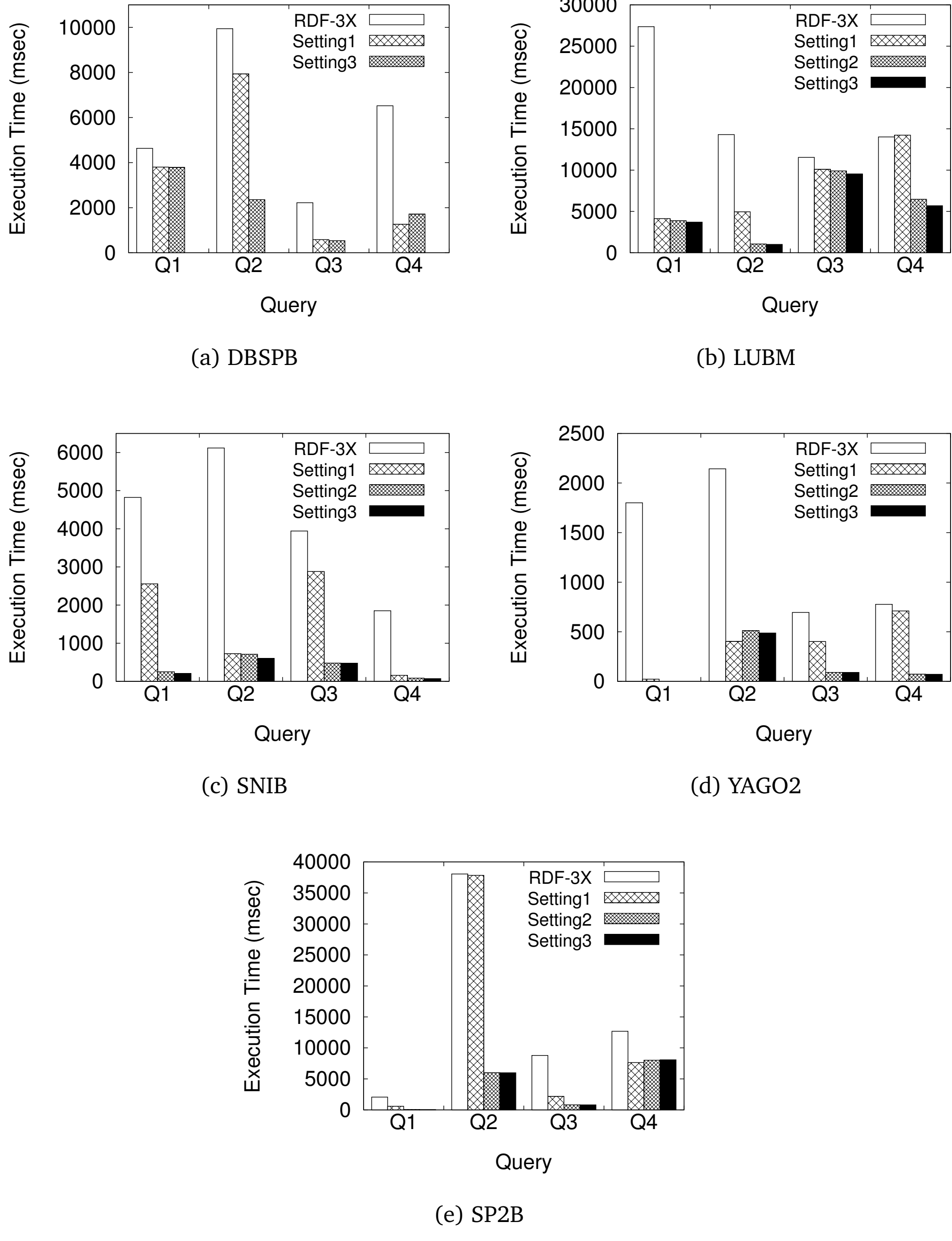
In this section, we present experimental results for the incremental update of RP-index. We measured the incremental update times of RP-index and compared them to the total rebuilding times.

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

Figure 4.17 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 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 4.18 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 4.6.3.



(a) DBSPB

(b) LUBM

(c) SNIB

(d) YAGO2

(e) SP2B

Figure 4.11: Query Execution Time

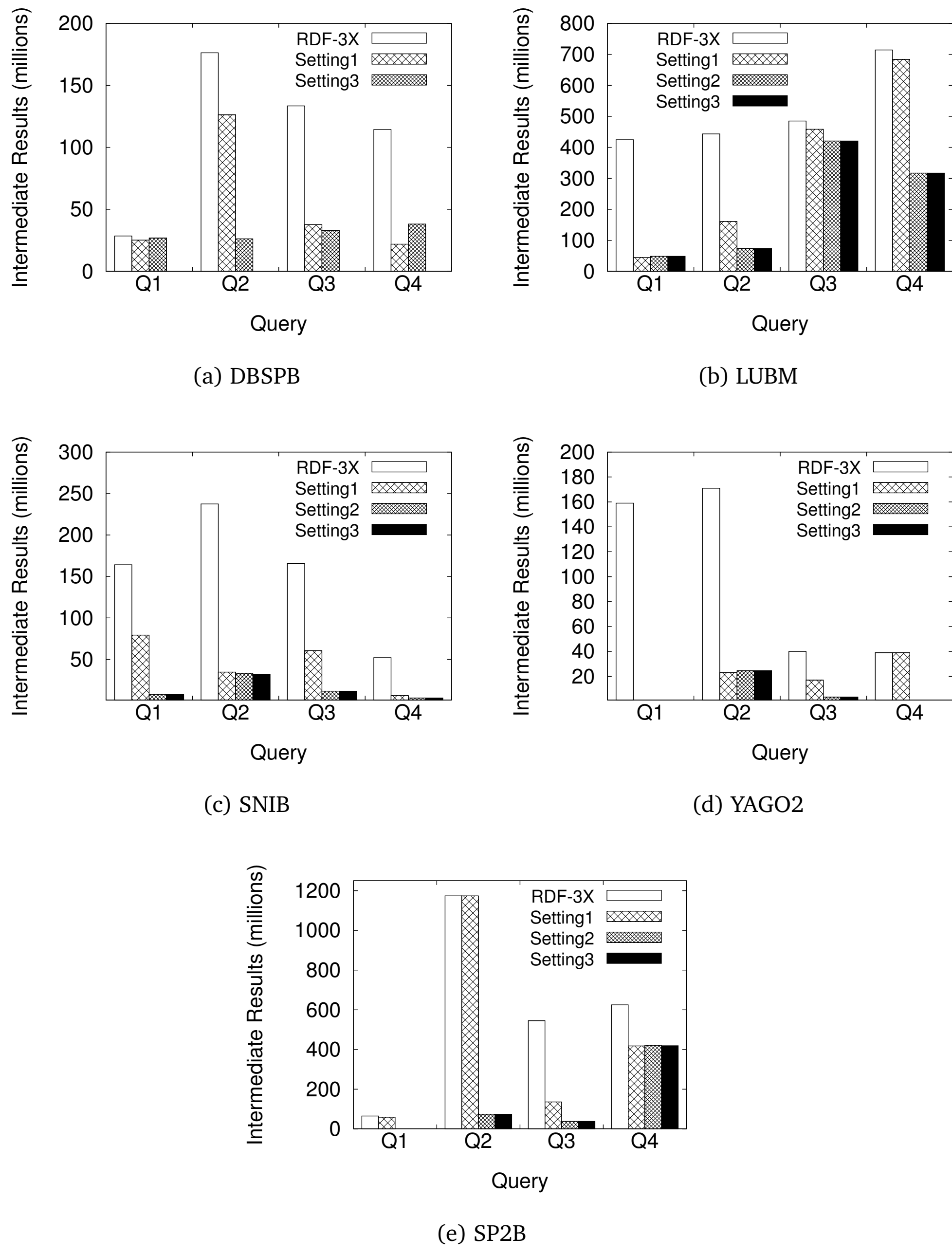


Figure 4.12: Intermediate Results

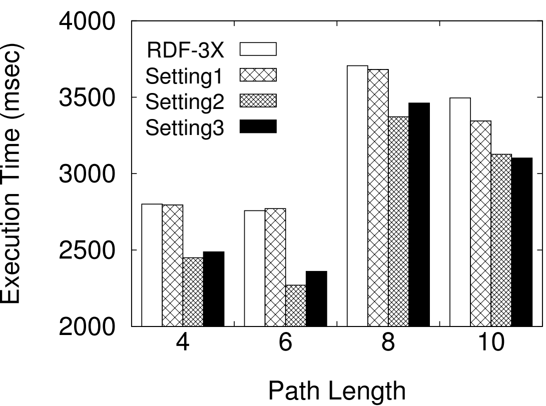
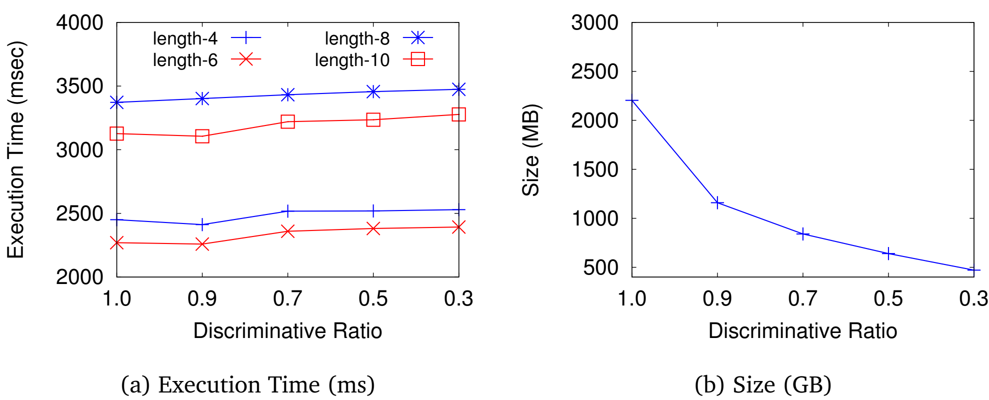
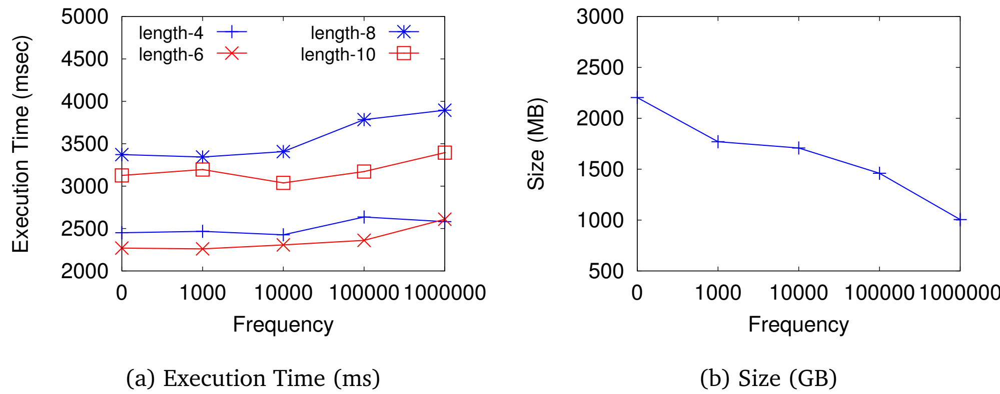


Figure 4.13: Path Query (YAGO2)



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

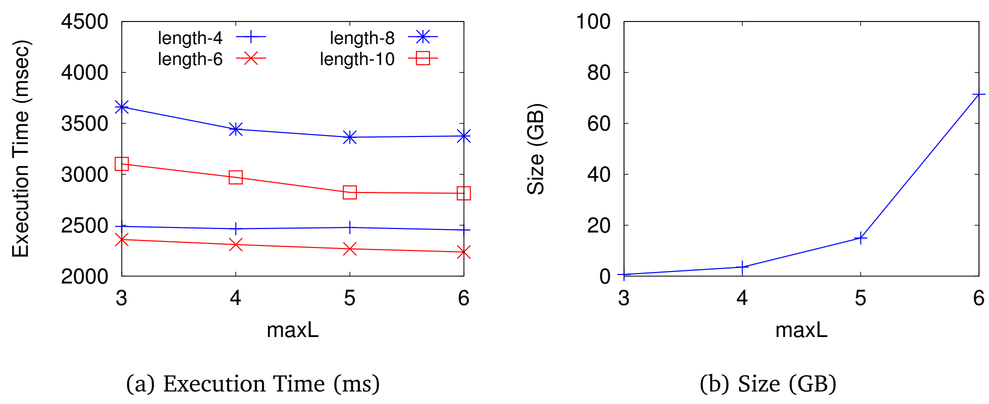
Figure 4.14: Effects of Discriminative Ratio (YAGO2)



(a) Execution Time (ms)

(b) Size (GB)

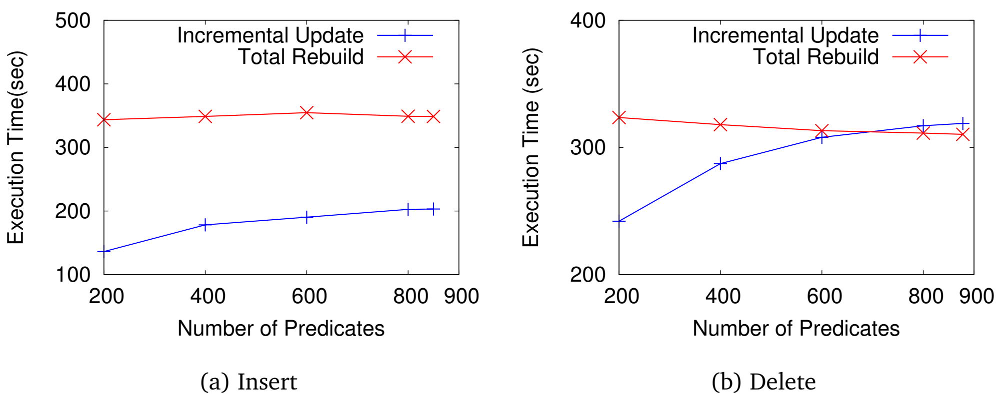
Figure 4.15: Effects of Frequency Function (YAGO2)



(a) Execution Time (ms)

(b) Size (GB)

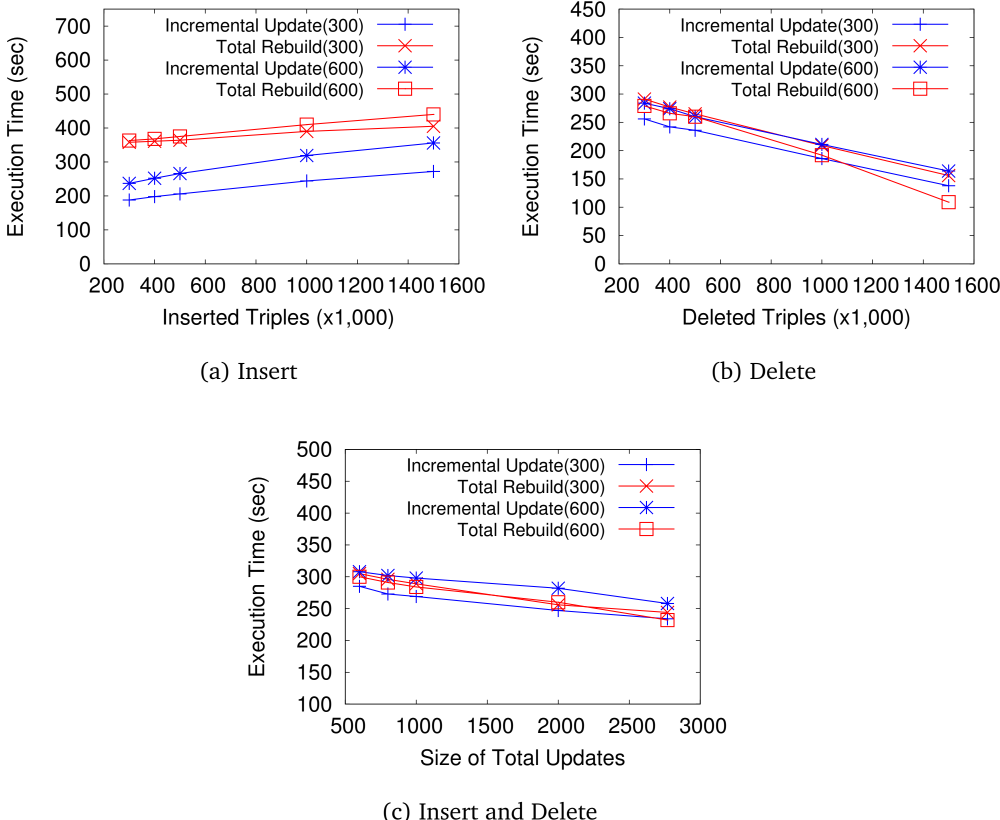
Figure 4.16: Effects of *maxL* (YAGO2)



(a) Insert

(b) Delete

Figure 4.17: Update Time (Predicates)



(a) Insert

(b) Delete

(c) Insert and Delete

Figure 4.18: Update Time (Update Size)

Chapter 5

RG-index: RDF Triple Filtering using the Graph Index

RP-index proposed in the previous chapter is to improve the query evaluation by reducing redundant intermediate results. It exploits the incoming path information in order to determine the irrelevance of a triple, and uses additional filtering operators in the execution plan to filter out irrelevant triples among the input triples. However, its filtering power is limited, because it uses only the incoming path information and cannot use the graph-structural information of the RDF graph. In this section, we present RG-index which uses the graph-structural information of the RDF graph.

5.1 Motivating Example

Let us consider the example in Figure 5.1. It shows a SPARQL query graph, its execution plan, and four fragments of an RDF graph: R_1, R_2, R_3, and R_4. In

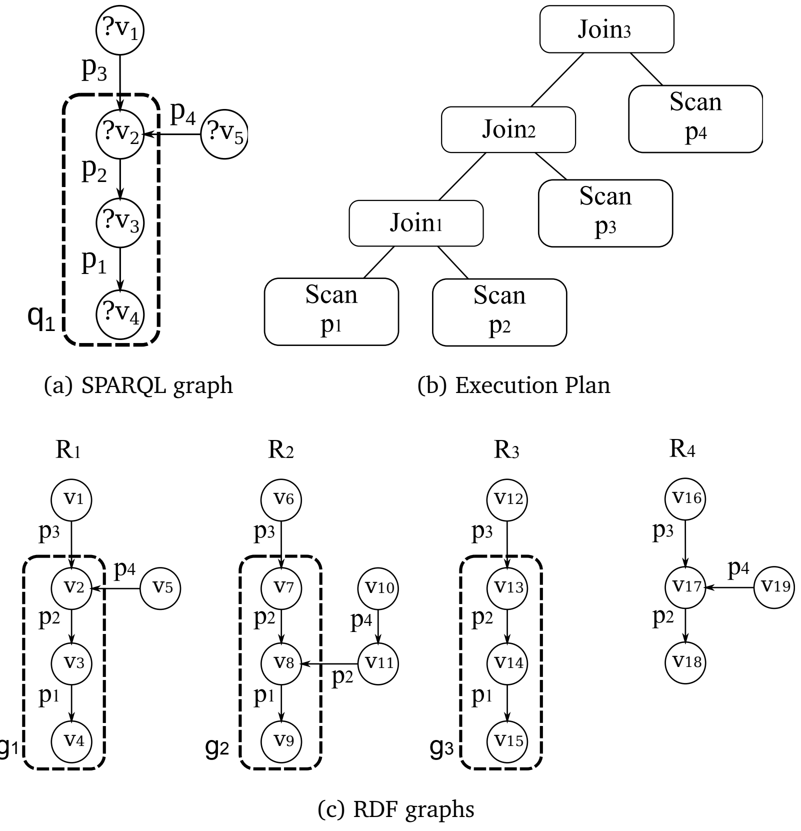


Figure 5.1: RDF Graph and SPARQL Query Graph

the execution plan, $Join_1$ produces the intermediate results matching the subgraph q_1 in the query graph: g_1, g_2, and g_3 in the RDF graph. However, because only R_1 is matched to the query graph, it is the final result for the query, and g_2 and g_3 become redundant intermediate results. RP-index can reduce these intermediate results using the necessary condition for the final results that the matching vertices for $?v_3$ should have two incoming predicate paths: $\langle p_3, p_2 \rangle$ and $\langle p_4, p_2 \rangle$. Using this necessary condition, RP-index can avoid producing g_3 in $Join_1$, because v_{14} does not have the incoming predicate path $\langle p_4, p_2 \rangle$. How-

ever, it should be noted that g_2 is still produced because v_6 has both incoming predicate paths and satisfies the necessary condition. In order to remove these intermediate results, we should be able to consider the graph-structural information.

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

In addition, we propose an efficient building algorithm for RG-index. In order to build RG-index efficiently, we adapt the gSpan [24] algorithm, one of the most well known algorithms for mining frequent graph patterns. Originally, gSpan was developed for treating a transactional graph database, which comprises many small-size graphs. Thus, in order to apply the gSpan to the RDF graph, which is a single large graph, the gSpan algorithm has to be modified. Further, to reduce the duplicate computations that occur during graph pattern mining, we propose a mechanism for caching the intermediate results.

5.2 Design of RG-index

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

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

Definition 5.2.1 [Graph Pattern] A graph pattern is a connected graph whose vertices are all variables and the labels of edges are all URIs. A set of triple patterns gp is called a graph pattern iff its mapping graph is a connected graph and $\forall tp(s,p,o) \in gp$, $s \in \mathit{VAR} \wedge o \in \mathit{VAR} \wedge p \in P_D$.

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

Definition 5.2.2 [Vertex List] Given a graph pattern $G(V,E,L)$ and a vertex $v \in V$, a vertex list $\mathit{Vlist}(G,v)$ is $\mathit{Ans}(G,v)$, the projection over v for the answer set of G. A set of all vertex lists for G is denoted by $\mathit{VS}(G) = \{\mathit{Vlist}(v,G) \mid \forall v \in V\}$

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

Size	Graph Pattern	Vlist
Size 1	gp_1 $?v_1$ p_1 $?v_2$	$Vlist(gp_1, ?v_1) = \{v_3, v_8, v_{14}\}$ $Vlist(gp_1, ?v_2) = \{v_4, v_9, v_{15}\}$
	gp_2 $?v_1$ p_2 $?v_2$	$Vlist(gp_2, ?v_1) = \{v_2, v_7, v_{13}, v_{17}\}$ $Vlist(gp_2, ?v_2) = \{v_3, v_8, v_{14}, v_{18}\}$
Size 2	gp_3 $?v_1$ p_2 $?v_2$ p_1 $?v_3$	$Vlist(gp_3, ?v_1) = \{v_2, v_7, v_{13}\}$ $Vlist(gp_3, ?v_2) = \{v_3, v_8, v_{14}\}$ $Vlist(gp_3, ?v_3) = \{v_4, v_9, v_{15}\}$
	gp_4 $?v_1$ p_3 $?v_2$ p_4 $?v_3$	$Vlist(gp_4, ?v_1) = \{v_1, v_{16}\}$ $Vlist(gp_4, ?v_2) = \{v_2, v_{17}\}$ $Vlist(gp_4, ?v_3) = \{v_5, v_{19}\}$
Size 3	gp_5 $?v_1$ p_3 $?v_2$ p_4 $?v_3$ p_2 $?v_4$	$Vlist(gp_5, ?v_1) = \{v_1, v_{16}\}$ $Vlist(gp_5, ?v_2) = \{v_2, v_{17}\}$ $Vlist(gp_5, ?v_3) = \{v_5, v_{19}\}$ $Vlist(gp_5, ?v_4) = \{v_3, v_{18}\}$

Figure 5.2: RG-index ($maxL = 3$)

Definition 5.2.3 [RG-index] RG-index for RDF database D with the maximum length *maxL* is a set of pairs $\langle G, VS(G) \rangle$, where G is a graph pattern in D whose size is less than or equal to *maxL*.

Example 5.2.4 [RG-index] Figure 5.2 shows an example of RG-index for the RDF graph in Figure 5.1c. This RG-index indexes five graph patterns for the RDF graph and has fourteen Vlists.

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

Lemma 5.2.5 *[Candidate Vertex Set] Given a vertex v in a query graph G_Q and maxL, we can obtain a superset of $CV(v, maxL)$ by intersecting Vlists for k-neighborhood subgraphs of v.*

$$\bigcap_{gp \in N(v, maxL)} Vlist(gp, v) \subseteq CV(v, maxL) \tag{5.1}$$

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

We generate graph patterns up to size *maxL*. Because the number of graph patterns grows exponentially for the its size, we have to limit the size of graph patterns to be indexed. There exists a tradeoff between the filtering power and the space overhead of RG-index. As *maxL* increases, more graph patterns are indexed in RG-index and therefore, its filtering effects for queries can be improved. However, the space overhead of RG-index also increases. This tradeoff can be adjusted by *maxL*.

5.2.1 Physical Structure of RG-index

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

DFS Code Representation

We use the minimal DFS code proposed for gSpan [24] as the canonical representation of graph patterns. The minimal DFS code for a graph pattern gp is defined as follows. First, all nodes in gp are given DFS subscripts while they are traversed by a depth-first search. If two nodes are subscripted as v_i and v_j, and $i < j$, then v_i is traversed before v_j. It should be noted that, for a graph pattern gp, many different subscripts can be made, because there can exist several DFS trees for gp.

By using this subscription, each edge in gp is represented by a DFS edge. Originally, gSpan was designed to treat undirected graphs [24], and DFS edge representation for undirected graphs was proposed. However, we are treating directed edge-labeled graphs. Therefore, the edge representation $\langle i, j, l_{(i,j)}, d_{(i,j)} \rangle$, where i and j are DFS subscripts, $l_{(i,j)}$ is the edge label, and $d_{(i,j)}$ is the edge direction, is used. If the edge is from v_i to v_j, $d_{(i,j)} = \rightarrow$; otherwise, $d_{(i,j)} = \leftarrow$. A DFS edge with $i < j$ is called a forward edge, and a DFS edge with $i > j$ is called a backward edge. Forward edges are edges that are visited during the DFS search, and edges that are not visited become backward edges.

Using this DFS subscription and the DFS edge representation, a graph pattern can be mapped into a DFS code, which is a sequence of DFS edges. In the DFS code, DFS edges for edges of the graph patterns are sequenced as follows. Forward edges are ordered as they are discovered. Given a vertex v, all of its backward edges should appear after the forward edge pointing to v.Among the backward edges from the same vertex, say (v_i, v_j), (v_i, v_k), if $j < k$, then (v_i, v_j) should appear before (v_i, v_k).

gSpan defines a lexicographic order among DFS codes [24]. For two given

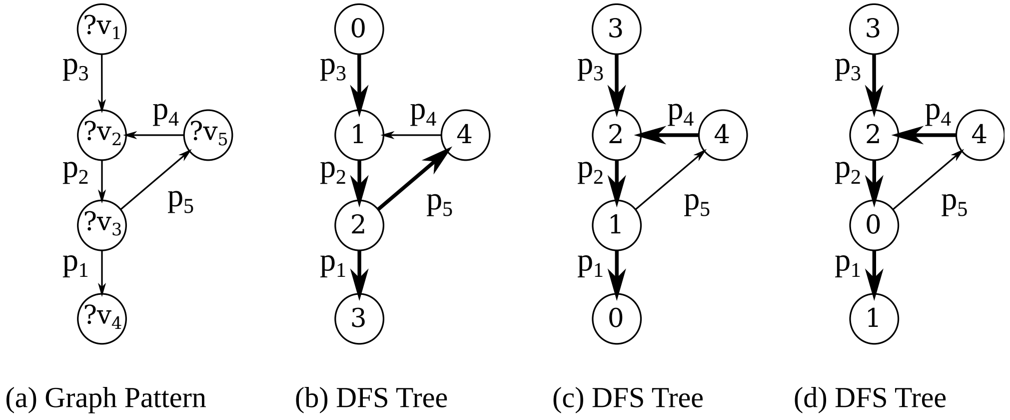


Figure 5.3: DFS Subscriptions

Table 5.1: DFS Codes of the Graph Pattern

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

DFS edges, the order is determined first by their two subscripts, then by edge labels, and finally by directions. We define the order between directions such that $\rightarrow$ is smaller that $\leftarrow$. gSpan defines the canonical label of gp as its lexicographically minimum DFS code.

Example 5.2.6 [DFS Code] Figure 5.3 shows a graph pattern (Figure 5.3 (a)) and its three DFS subscriptions (Figure 5.3 (b)–(d)). Each vertex is annotated

by its subscription. Forward edges are represented by thick edges, while backward edges are represented by thin edges. Table 5.1 shows the DFS codes for three subscriptions. The order among the DFS codes is (d) $<$ (c) $<$ (d); (d) is the minimum DFS code for the graph pattern.

Storage of RG-index

Each vertex in the RDF database is assigned a four-byte integer ID. Physically, Vlists are stored as the sorted lists of these vertex IDs. Vlists are stored in a disk as sorted by vertex IDs so that the Vlist can be read from the disk as sorted. The reason for storing Vlists as sorted is to allow the filter data to be obtained by simply merging the relevant Vlists. Another benefit of sorting is that sorted Vlists can be compressed by the delta-based byte-level compression scheme similarly to compressed triples in RDF-3X [14]. The delta between two vertex IDs is encoded with one header byte and the minimum number of bytes for the delta (1 bytes $\sim$ 4 bytes). If the delta is smaller than 128, it is stored directly in the header byte, consuming only one byte. Otherwise, the header byte stores the byte length of the delta with its most significant bit set as 1 to indicate the delta is not small. This compression scheme alleviates the overall size overhead of Vlists and reduces disk I/O overhead for reading the Vlists.

The DFS codes of the graph patterns in RG-index are organized in a trie (or prefix tree) data structure. Each node in level l in the trie has a pointer to the Vlist for its associated length-l DFS code. The trie provides compact storage for the DFS codes, because the duplicated parts of the DFS codes can be shared. In addition, it provides an efficient way to access the Vlist for a given predicate path. The location in the disk of the Vlist for a graph pattern can be found by

traversing the trie using the DFS code. The number of the nodes in the trie is equal to the number of the DFS codes in RG-index. For real-life datasets and a small $maxL$ value, the trie is relatively small and can be resident in the main memory.

5.3 Handling the Size Problem of RG-index

Even if the graph patterns are limited to size $maxL$, it is still infeasible to index all possible subgraph patterns in the RDF database D, due to the exponential number of the possible graph patterns. Because RG-index is designed to provide the filter data for the triple filtering, it does not have to index all possible graph patterns in D. Instead, by choosing and indexing only graph patterns with effective filtering power, its size can be reduced while its filtering power is retained. We discuss how to choose the graph patterns in Section 5.3.1.

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

5.3.1 Discriminative Patterns

The first criterion is to store only Vlists with enough filtering power as compared to other replaceable Vlists. If $Vlist_i \supset Vlist_j$, $Vlist_i$ can be used in place of $Vlist_j$, because $Vlist_i$ has all vertices in $Vlist_j$. Therefore, it is possible to store only $Vlist_i$ and remove $Vlist_j$ from RP-index. However, this replacement can degrade the filtering power because the replacing filter is prone to produce more false positives than the replaced filter. Therefore, it is important to choose predicate paths that do not degrade the filtering power significantly. A discriminative

predicate path is one whose Vlist cannot be replaced by another Vlist without significantly degrading the filtering power. We define the discriminative Vlist as follows.

Definition 5.3.1 [Discriminative Vlist] Given discriminative ratio γ $(0 < \gamma \leq 1)$ and a set of Vlists V, *vlist* is discriminative w.r.t V iff $\forall vlist_s \in V \wedge vlist_s \in vlist$, $|vlist| < \gamma \times |vlist_s|$.

5.3.2 Frequent Patterns

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

Definition 5.3.2 [Frequent Graph Pattern] Given size-increasing function $\psi(l)$, a graph pattern G is frequent if and only if $sup(G) \geq \psi(|G|)$.

5.4 Building RG-index

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

5.4.1 Overview of gSpan

gSpan [24] generates graph patterns in a depth-first fashion. That is, it starts from a 1-edge pattern and grows the pattern into larger patterns by adding one edge to the pattern. The most important issue in gSpan is minimizing the generation of the same graph patterns. Because a graph pattern can be generated in several ways, for efficient mining it is essential not to generate the patterns in duplicate. To achieve this, the pattern generation of gSpan is limited to the minimum DFS codes; otherwise, it is possible that the same patterns can be generated several times. If gSpan can ensure that all minimum DFS codes are generated, the generations of the non-minimum DFS codes are redundant because any graph pattern can be represented by the minimum DFS code. Therefore, for each generated DFS code, gSpan checks whether it is the minimum DFS code for the generated pattern, and if not, the DFS code is pruned and not extended further.

In addition, to reduce the generation of non-minimum DFS codes, gSpan uses the rightmost extension when adding an edge to a graph pattern. The rightmost extension restricts the pattern growth as follows. For a DFS code, the first and the last vertices of the DFS traversal are called the *root* and the

rightmost vertex, respectively. The path from the root to the rightmost vertex is called the *rightmost path*. The patterns can be grown such that a forward edge can be added to vertices in the rightmost path, and a backward edge can be added only to the rightmost vertex. If g is extended by adding e according to the rightmost extension, the extended pattern is denoted by $g \diamond_r e$.

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

5.4.2 RDF Graph Pattern Mining using gSpan

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

Support for the RDF graph

First, in order to apply the frequent pattern mining algorithm for the RDF graph, we need to measure the support of the generated patterns. gSpan was proposed for use in the context of the transactional setting, and the support for the transactional setting is easily defined as the number of graphs in the database matched for a graph pattern. This definition has the anti-monotonicity property, which is essential for efficient mining. However, it is not easy to define the support that satisfies this property in the single large graph setting [53]. Several

support definitions for the single large graph setting have been proposed. We use the definition of graph pattern frequency in [69].

Definition 5.4.1 [Support of Graph Pattern] Given a graph pattern $G(V, E, L)$, the support of G is $sup(G) = \min_{v \in V}(|Vlist(G, v)|)$.

This definition uses the minimum number of vertices of the graph pattern as the support, and is computationally efficient as compared to other support definitions for the single graph [70]. In addition, it ensures the anti-monotonicity of the support. Using this support, only the size of Vlists for the graph pattern is required.

Avoiding Redundant Patterns

We restrict the pattern generation of gSpan such that it does not generate all possible patterns in the RDF graph. There exist graph patterns that become redundant due to the semantics of SPARQL. In fact, evaluating the basic graph patterns of SPARQL queries is not exactly the same as subgraph isomorphism. This is because pattern mapping is not bijective; that is, the different vertices in a query graph can be matched to a same vertex in the RDF graph. Let us take a look at the example in Figure 5.4. In this figure, there are an RDF graph and three graph patterns: G_1, G_2, and G_3. These three graph patterns are all matched to the RDF graph, although G_2 and G_3 have more edges than the RDF graph. v_2 in the RDF graph is matched to several vertices in these patterns; i.e., $?v_2$ and $?v_3$ in G_2 are matched to v_2. Therefore, the Vlists for these vertices are identical; *Vlist*$(G_1, ?v_2) =$ *Vlist*$(G_2, ?v_2) =$ *Vlist*$(G_2, ?v_3) =$ *Vlist*$(G_3, ?v_2) =$

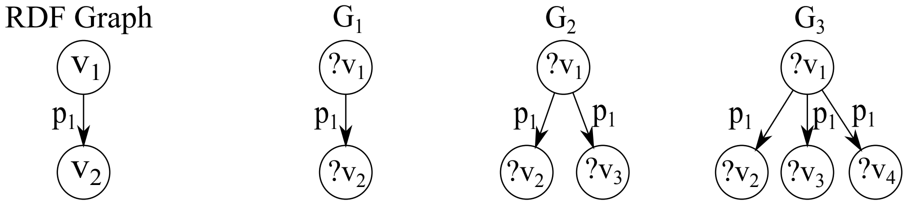


Figure 5.4: Redundant Graph Pattern

Vlist$(G_3, ?v_3)$ = *Vlist*$(G_3, ?v_4) = \{v_2\}$. G_2 and G_3 are redundant because they have the same filtering power as G_1.

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

Lemma 5.4.2 *If a graph pattern G has a non-trivial automorphism θ, then $\forall v \in G \wedge \theta(v) \neq v$, $s.t.$Vlist(G, v) = Vlist$(G, \theta(v))$, where $\theta(v)$ is the matching vertex by θ.*

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

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

In order not to generate graph patterns having automorphisms, automorphism checking should be performed for each generated graph pattern that is known to be NP-complete. Since this is too costly, we take an approximate approach instead. Patterns whose vertices have edges of the same type, i.e., edges with the same label and the same direction, are not generated. However, this method can remove graph patterns that are not redundant. For example, the

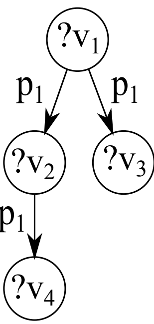


Figure 5.5: Non-redundant Graph Pattern

graph pattern in Figure 5.4 is removed because v_1 has two edges of the same type. However, it does not have non-trivial automorphism. Although the exclusion of these types of patterns can degrade the filtering power of RG-index, in order to achieve efficient construction, these patterns are not considered.

Caching the Intermediate Results

The support of each generated pattern should be calculated and Vlists built for it. However, this process is very time-consuming because it requires finding all occurrences of the pattern in the RDF graph. The easiest way to perform this is to make and execute a SPARQL query for the generated pattern. However, this incurs many duplicate computations. Let us take a look at the example in Figure 5.6. This figure illustrates a forward extension in which G_1 is extended to G_2. If the occurrences of these two patterns are calculated separately using two SPARQL queries generated for them, the subgraph of G_2, which corresponds to G_1, is calculated twice. This is because G_2 contains G_1.

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

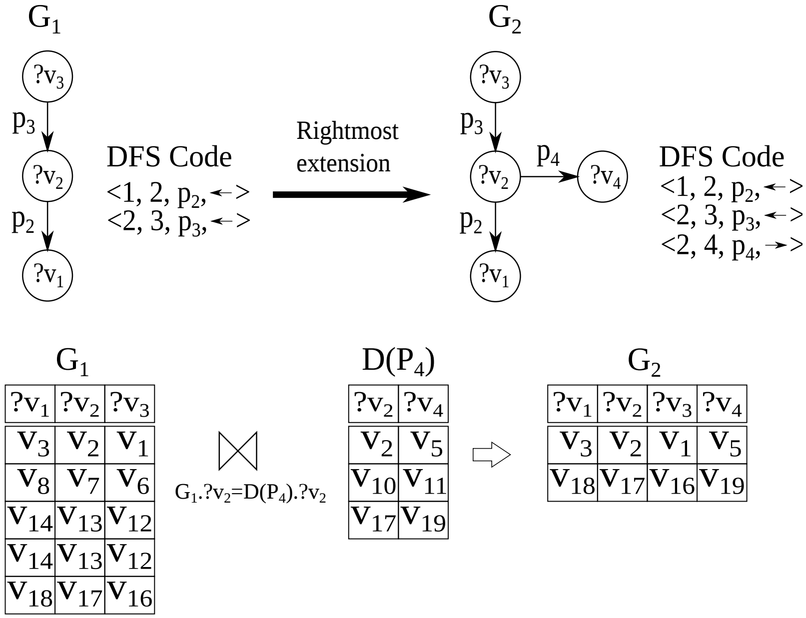


Figure 5.6: Rightmost Extension

Figure 5.6 shows the entire process. The occurrences of the graph pattern are stored as a table whose columns correspond to each vertex in the graph pattern. Then, the occurrences of the child patterns can be obtained by a join operation for the table. In this example, the table G_1, which contains the occurrence results of G_1, is joined with the triple whose predicate is p_4, and the results become the occurrences of G_2.

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

$$G_2 = \begin{cases} G_1 \bowtie_{v_i=S} D(p) & \text{if add}\langle v_i, v_j, p, \leftarrow\rangle \\ G_1 \bowtie_{v_i=O} D(p) & \text{if add}\langle v_i, v_j, p, \rightarrow\rangle \end{cases} \tag{5.2}$$

For a backward extension, the results are calculated as

$$G_2 = \begin{cases} \sigma_{v_j=O}\left(G_1 \bowtie_{v_i=S} D(p)\right) & \text{if add}\langle v_i, v_j, p, \leftarrow\rangle \\ \sigma_{v_j=O}\left(G_1 \bowtie_{v_i=O} D(p)\right) & \text{if add}\langle v_i, v_j, p, \rightarrow\rangle \end{cases} \tag{5.3}$$

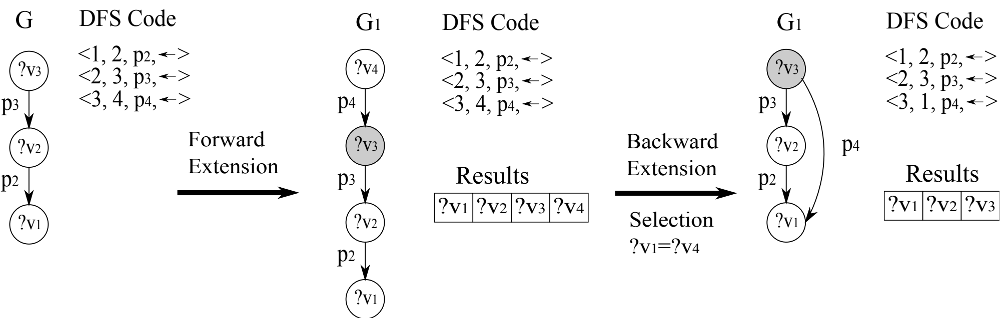


Figure 5.7: Backward Extension using the Results of the Forward Extension

For a backward extension, a selection operation is needed in addition to the join operation. The results of a backward extension can be obtained from the selection operation for the results of a forward extension. Figure 5.7 shows an example of the rightmost extension for the rightmost vertex, $?v_3$. First, the results of the forward extension are calculated. Then, if the extension is for the rightmost vertex, it can be used for the backward extension (the backward extension is possible only for the rightmost vertex). If the backward extension is to add $\langle 3, 1, p_4 \rangle$, then the results can be obtained by performing the selection operation with the condition $?v_4 = ?v_1$ for the results of the forward extension. This is efficient, because only the selection operation is required, and the results of the forward extension are reused.

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

Algorithm 4 shows the overall process of building RG-index. The function

Algorithm 4 gSpanRDF (s, D, minSup, RGindex)

Input: an RDF database D and *minSup*

```
1: n ← |V|;
2: /* rightmost extension */
3: for each v ∈ the rightmost path of s do
4:    for each p ∈ P and d ∈ {←, →}  do
5:       /* forward extension */
6:       e ← ⟨v, v_{n+1}, p, d⟩
7:       if s ◇_r e is not minimal then
8:          continue;
9:       end if
10:      G′ ←getOccurrenceForward(G, p, e);
11:      if s ◇_r e is frequent and discriminative then
12:         Insert into RGindex
13:      end if
14:      if v is the rightmost vertex then
15:         /* backward extension */
16:         for each v_j ∈ V ∧ v_j ≠ v  do
17:            e_b ← ⟨v, v_j, p, d⟩
18:            if s ◇_r e_b is not minimal then
19:               continue;
20:            end if
21:            G_b ← getOccurrenceBackward(G′, p, e);
22:            if s ◇_r e is frequent and discriminative then
23:               Insert into RGindex
24:            end if
25:         end for
26:      end if
27:   end for
28: end for
29: return
```

gSpanRDF is called recursively to generate graph patterns from 1-size to *maxL*-size. It adds an edge to the input DFS code. First, it performs the forward extension for every vertex in the rightmost path. It adds edges, varying the label with the predicates in the RDF database and its direction. Then, it checks that

the generated DFS code is the minimal DFS code of its corresponding graph pattern. If not, the DFS code is pruned. Then, it calculates the occurrences of the graph patterns using Eq. 5.2. If the graph pattern is frequent and discriminative, the pattern and Vlists are inserted into RG-index. Then, if the extension is for the rightmost vertex, it performs the backward extension. The edge for the backward extension is from the rightmost vertex and to the other vertex in the graph pattern. The DFS code should be also checked as to whether it is minimal. Then, the occurrences of the DFS code are calculated by performing the selection operation for the results of the forward extension, as previously explained.

5.4.3 Complexity of building RG-index

First of all, Vlists for 1-size graph patterns are created. This costs reading the whole database, which can represented by $|D|$ (the number of all triples). Then, for each occurrence of $n-1$-size subgraphs, n-size subgraphs are built. We can represent the maximum number of occurrences of n-size subgraph pattern by $|D|^n$. So we can represent the complexity of the building RG-index by $O\left(|D| + \sum_{n=1}^{maxL-1} |D|^{n-1} \times |D|\right)$. Note that this is a worst case complexity, and the cost in practice are much lower and we can adjust it by using several parameters.

5.5 Triple Filtering using RG-index

In this section, we describe how the triple filtering is processed. The triple filtering using RG-index is similar to the case using RP-index. We use RFLT operator, which is described in Section 4.4. In this section, we focus on the difference

between using RG-index and using RP-index.

5.5.1 Generating an Execution Plan with RFLT Operators

Cost Function of RFLT operator

First, we define the cost function of RFLT operator using RG-index. It is very similar to the previous cost function. However, we need to redefine it considering the graph patterns.

$$\text{I/O cost: } O\left(\sum_{g \in GS} \|Vlist(g,v)\|\right) \tag{5.4}$$

$$\text{CPU cost: } O\left(\sum_{scan \in ChildOP} |scan| + \sum_{g \in GS} |Vlist(g,v)|\right) \tag{5.5}$$

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

Output Cardinality Estimation of RFLT Operator

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

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

We first consider RFLT operator having one scan operator. The set of distinct values for the sortkey column of the scan operator is denoted by S. The

intersection of S and Vlists of RFLT is denoted by $C = \cap_{g \in GS} \mathit{Vlist(g,v)}$. Then, the output cardinality of RFLT operator can be estimated as

$$|\text{RFLT}| = |\text{Scan}| \times \frac{C}{S} \tag{5.6}$$

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

$$|\text{RFLT}| = |J| \times \prod_{scan \in Childs} \frac{|\text{scan}|}{\text{S}} \tag{5.7}$$

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

Our method is very similar to the Characteristic Set [50], which was proposed for estimating the cardinalities of star-join queries. However, our method does not aim to replace the Characteristic Set, but rather to reflect the filtering effect in the cardinality estimation. We expect that exploiting the Characteristic Set in our estimation method will improve the estimation accuracy. Therefore, our method and the Characteristic Set have a complementary relationship.

Adding RFLT Operators

We use the query optimization of RDF-3X, which is based on the bottom-up dynamic-programming (DP) framework [14]. The query compiler maintains

the DP table, in which the optimal plans for the subproblems of the query are stored. First, the optimizer seeds its DP table with scan operators for the triple patterns as solutions of the 1-size subproblems. For each scan operator created in the seeding phase, an RFLT operator is added as its parent operator. The query optimizer should assign to RFLT operators the Vlists for the triple filtering. It assigns to an RFLT operator the Vlists for the graph patterns, which are k-neighborhood subgraphs of the query graph for the sortkey of the child scan operators. However, it is not necessary to assign all k-neighborhood subgraphs. If there are two subgraphs, $s.t.\ g_i \subset g_j$, $\mathit{Vlists}(g_i, v)$ does not need to be assigned because $\mathit{Vlists}(g_j, v) \subset \mathit{Vlists}(g_i, v)$. For an RFLT operator, $\mathit{RFLT.Vlist} = \{\mathit{Vlist}(g, v) \mid g \in N(v, \mathit{maxL}) \wedge \nexists g' \in N(v, \mathit{maxL})\ s.t.\ g \subset g'\}$.

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

5.6 Experimental Results

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

All the experiments were conducted on a hardware platform with eight 3.0 GHz Intel Xeon processors, 16 GB of memory, and running the 64-bit 2.6.31-23 Linux Kernel. We conducted the experiments using three datasets: Lehigh Uni-

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

Table 5.2: Statistics about Datasets

	Predicates	URIs	Literals	Triples	RDF-3X Size
LUBM	18	217,007,404	111,618,881	1,334,681,192	77 GB
YAGO2	93	6,872,931	22,452,390	195,048,649	9 GB
SP2B	77	177,272,798	348,388,613	931,696,802	123 GB

Table 5.3: Graph Densities of Datasets

	Graph Density	Graph Density (without Literals)
LUBM	$1.23 \times e - 08$	$1.89 \times e - 08$
YAGO2	$2.26 \times e - 07$	$1.76 \times e - 06$
SP2B	$3.37 \times e - 09$	$1.46 \times e - 08$

versity Benchmark (LUBM) [64], Yet Another Great Ontology 2 (YAGO2) [6], and SPARQL Performance Benchmark (SP2B) [65]. LUBM is a benchmark dataset whose domain is the university, YAGO2 is a knowledge-base derived from Wikipedia[2], WordNet [67], and GeoNames[3], and SP2B is a benchmark that simulates the DBLP scenario[4].

The benchmark datasets (LUBM and SP2B) have their own scale factors. We generated 10,000 universities for LUBM, and 96 GB triples for SP2B. These datasets have different characteristics, as shown in Table 5.2.

[2] `http://www.wikipedia.org`

[3] `http://www.geonames.org`

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

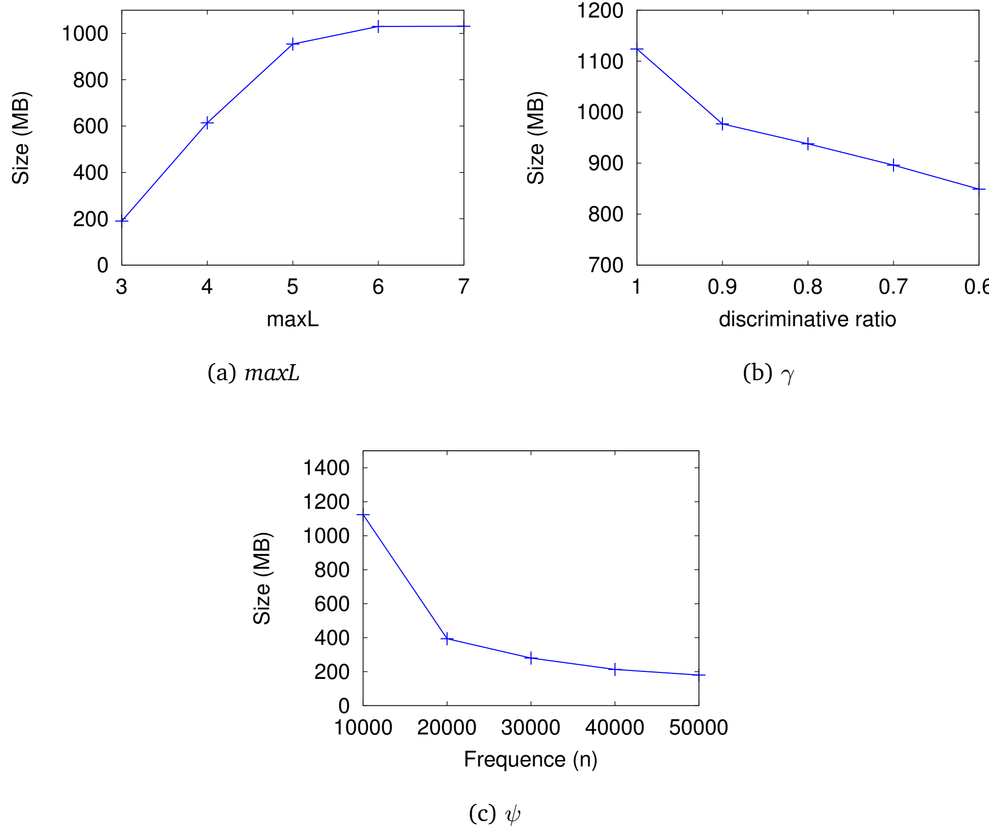


Figure 5.8: RG-index Size (YAGO2)

5.6.1 RG-index Size

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

Figure 5.9 shows the effects of three parameters: *maxL*, γ, and ψ (actually n in the function). As can be seen in this figure, the size of RG-index grows exponentially for the size of the graph patterns. However, by varying the two parameters, γ and ψ, we can adjust the size adequately for our purpose. In addition, the size of RG-index using γ and ψ is small as compared to that of the RDF database. We describe the effects of the parameters for the query performance in the next section.

5.6.2 Query Evaluation Performance

In this section, we present the effects of RG-index for the query evaluation performance. First, we compared the query evaluation performance of RDF-3X, RP-index, and RG-index. We built the RG-indices and the RP-indices for the three datasets. RP-indices index all incoming path patterns whose length is up to 7 (i.e., $maxL = 7$). We also build RP-indices including reverse predicates and we denote these as RP-index (R). We build RG-indices and use the same *maxL* for RG-index. However, the RG-indices index a subset of graph patterns by using $\gamma = 0.7$ and $\psi(l) = ((l-1)/maxL)^2 \times n$, where n is different for each dataset due to its size problem. Table 5.4 shows the statistics of the RG-indices and the RP-indexs.

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

In order to measure the query performance, we extracted graph patterns from each dataset and used them as the test queries. Table 5.6 shows the aver-

Table 5.4: Index Statistics

(a) YAGO2

	Size	Number of Patterns	Number of Vlists
RP-index	341 MB	486,508	486,508
RP-index (R)	2.3 G	852,676	852,676
RG-index	880 MB	82,534	416,497

(b) LUBM

	Size	Number of Patterns	Number of Vlists
RP-index	1.4 G	77	77
RP-index (R)	1.7 G	102	102
RG-index	1.1 G	103	535

(c) SP2B

	Size	Number of Patterns	Number of Vlists
RP-index	1.3 G	68,277	68,277
RP-index (R)	3.1 G	122,117	122,117
RG-index	1.3 G	32,436	149,812

aged query execution time. In this table, the queries are divided according to their execution times. YAGO2 has many queries with short execution times because it is small and has many predicates. However, LUBM has a small number of predicates and a large number of triples. Therefore, its queries take a long time to evaluate. SP2B has intermediate characteristics between YAGO2 and

Table 5.5: Query Statistics

Group	A	B	C	D	
Execution Time (ms)	$0 \sim 10$	$10 \sim 100$	$100 \sim 1000$	$1000 \sim$	Total
Count (YAGO2)	824	143	41	19	1027
Count (LUBM)	0	7	14	45	67
Count (SP2B)	178	203	189	7	577

LUBM.

Table 5.6 shows the averaged execution times. Both the RP-index and RG-index improve the query performance by more than about 30%. In addition, it can be seen that RG-index is more effective than the RP-index for YAGO2 and SP2B. In LUBM, the RP-index and RG-index show similar effects. This is because LUBM has a relatively structured data model, and therefore, there exists a small amount graph pattern that is effective for triple filtering. In addition, it should be noted that RG-index is more effective for queries with longer execution times. This is because the queries with long execution times have more intermediate results, which RG-index can reduce effectively.

Next, we measured the query performance varying RG-index parameters. We used RG-index, which was presented in Section 5.6.1, in order to present the effect of the parameters on the size of RG-index. Figure 5.9 shows the results. We can improve the query performance by increasing the *maxL* value. However, the improvement decreases as the *maxL* value increases. Therefore, we should choose an adequate *maxL* value for the query workload. The discriminative ratio rarely affects the query performance. This is because the effective graph

Table 5.6: Query Execution Time (ms)

(a) YAGO2

Group	A	B	C	D	Total
RDF-3X	3.53	34.18	240.43	16671.26l	325.62
RP-index	2.75 (22%)	11.83 (65%)	94.73 (60%)	9194.21 (44%)	177.73 (45%)
RP-index (R)	3.00 (15%)	17.82 (47%)	79.78 (66%)	4747.26 (71%)	95.90 (70%)
RG-index	2.32 (34%)	8.65 (74%)	27.60 (88%)	581.36 (96%)	14.92 (95%)

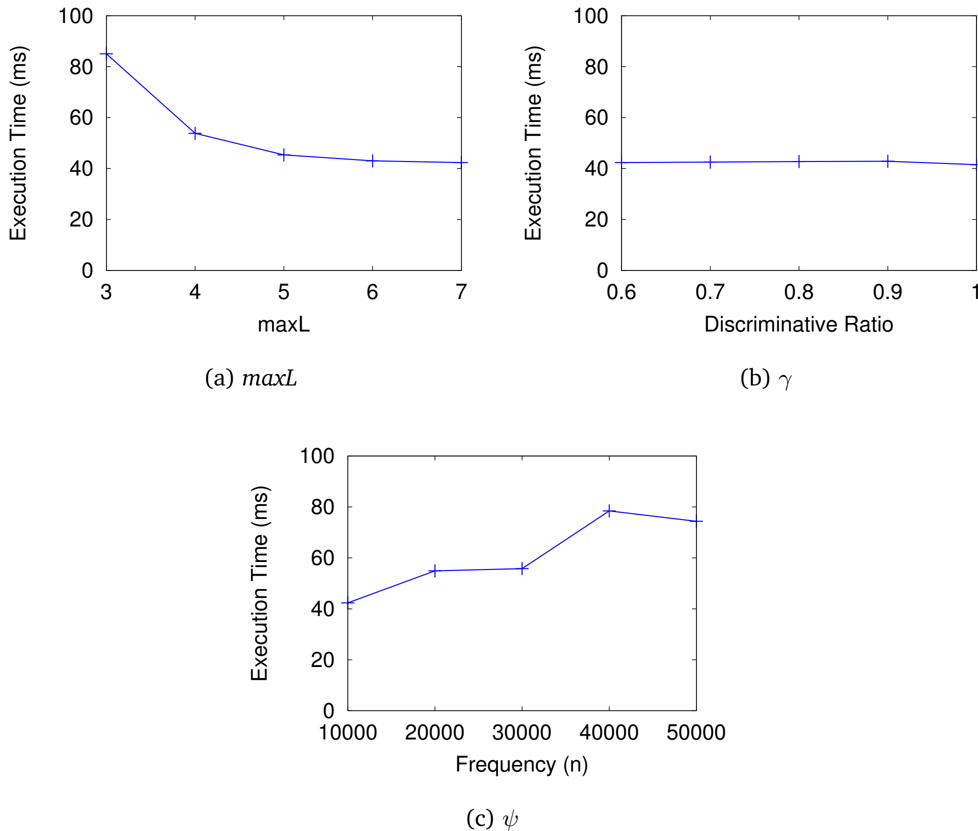
(b) LUBM

Group	A	B	C	D	Total
RDF-3X	N/A	53	540.8	134,490	114,385
RP-index	N/A	50 (5%)	479.6 (11%)	90,290 (32%)	76,808 (32%)
RP-index (R)	N/A	50 (5%)	479.6 (11%)	90,290 (32%)	76,808 (32%)
RG-index	N/A	50 (5%)	477.2 (11%)	89,587 (33%)	76,209 (33%)

(c) SP2B

Group	A	B	C	D	Total
RDF-3X	2.76	29.02	244.62	1383.42	108.65
RP-index	2.38 (13%)	25.2 (13%)	182.72 (25%)	555.42 (59%)	76.08 (30%)
RP-index (R)	2.39 (13%)	25.2 (13%)	153.92 (43%)	127 (91%)	61.06 (44%)
RG-index	2.33 (15%)	16.39 (43%)	122.8 (49%)	106.85 (92%)	44.34 (59%)

patterns remain for the small discriminative ratio. The frequency affects the query performance. Therefore, we should adapt the frequency considering the trade-off between the size and the query performance.



(a) *maxL* (b) γ

(c) ψ

Figure 5.9: Query Execution Time (YAGO2)

5.6.3 Index Building Time

In this section, we measure the index building time of RP-index and RG-index and compare these times to the data loading time of RDF-3X. For this experiments, we use YAGO2 dataset. The data loading time includes the parsing and loading of triples, and building triple indices and computing statistics. We measure the index building time of RP-index and RG-index varying the frequency threshold using n in the frequency function, $\psi(l) = ((l-1)/\mathit{maxL})^2 \times n$. Also, in

Table 5.7: Database Loading Time of RDF-3X

	RDF-3X
Loading Time (secs)	4,264
Query Time (msecs)	409.4

Table 5.8: Index Building Time (ms)

(a) RP-index

	Without Reverse Predicate	n = 1000	n = 2000	n = 4000
Building Time (secs)	93.33	449.33	299.79	164.88
Query Time (msecs)	368.19	254.0	254.01	258.3

(b) RG-index

	n = 1000	n = 2000	n = 4000
Building Time (secs)	5776.25	3290.53	1381.61
Query Time (msec)	171.25	169.46	187.34

order to give the impact to the query performance, we measure the query performance for each case with the entire query set which we used in the previous section.

Table **??** shows the data loading time of RDF-3X and the averaged query time. And table **??** shows the index building time and the query performance of RP-index and RG-index. First of all, we can see that the index building time of RP-index is affordable compared to the data loading time. The building time of RG-index is longer than the data loading time. However, we can see that

it decreases below the data loading time as the frequency threshold increases. Also, we can observe that RP-index takes shorter time to build than RG-index, the query performance of RG-index is superior to those of RP-index.

Chapter 6

Conclusion and Future Work

In this thesis, we proposed a novel triple filtering framework in order to reduce the number of redundant intermediate results in SPARQL query processing. The triple filtering filters out redundant triples using a necessary condition for results based on the graph-structural information of the RDF graph. To organize the filter data for the triple filtering, we designed RP-index and RG-index. RP-index uses the path information and has limited filtering power. In order to increase the filtering power, we also proposed RG-index which uses the graph-structural information. RG-index indexes the graph patterns in the RDF graph, and therefore, it can improve the query performance more than a triple filtering method that uses a path-based index. However, the size of RG-index grows exponentially for the pattern size. In order to address the size problem of RG-index, we proposed indexing only the discriminative and frequent patterns. In addition, we proposed an efficient algorithm for building RG-index, which is an adaptation of the frequent graph pattern mining algorithm, gSpan. And we

also considered the size problem and maintenance issues of each index. In addition, we presented 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 the triple filtering is very effective in reducing the number of redundant intermediate results, and improved query performance for complex SPARQL queries.

6.1 Future Work

Indexing Patterns Considering Query Workload

We propose path-based index and sub-graph index for the triple filtering. These indices extract the existing patterns in the RDF graphs. However, as mentioned before, due to the size problem we could not index all existing patterns in the RDF-graphs. We should limit the maximum size of indexed patterns or adapt several parameters in order to make the indices have affordable size. However, this could limit the performance of the triple filtering. For example, for the cases that the large size queries are general, both RP-index and RG-index with small *maxL* would not filter effectively.

In addition, because we do not include infrequent patterns in the indices in the hypotheses that infrequent patterns are not liable to be queried, some effective infrequent patterns can be removed. Let us take the example in Figure 6.1. In this RDF graph, there is a infrequent pattern. If the pattern is frequently used in the query workload, it will be better to decide to include this infrequent pattern in the index.

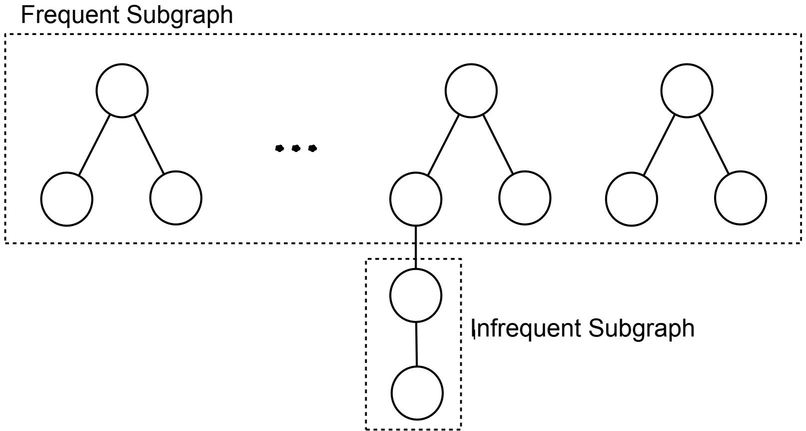


Figure 6.1: Infrequent Pattern

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

More Accurate Estimation of Cardinality

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

In addition, The RFLT operator which conducts the triple filtering changes the output cardinalities of the target scan operators. Although we propose a cardinality estimation method for RFLT operator, it has a limitation that it assumes the uniform distribution. We also observed some cases that the assumption does

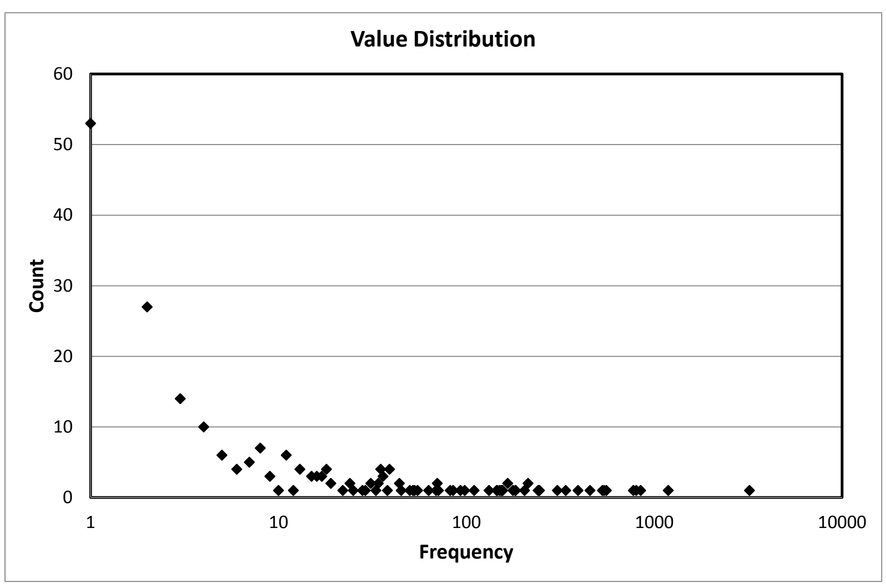


Figure 6.2: Object Value Distribution of predicate 'isCitizenOf' (YAGO2)

not hold, and therefore the estimation results are very poor. Figure 6.2 shows the object value distribution of the predicate 'isCitizenOf' of YAGO2 dataset. We can see that the uniform distribution assumption does not hold for this predicate.

In our estimation method, we use the set intersection calculation. In this thesis, although we use the upper bound for estimating the set intersection, there are other set intersection estimation methods. We plan to applying other set intersection methods in order to make the cardinality estimation more accurate.

Applying Distributed Environment

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

Appendix A

Related Open Source Projects

We use two open source project, RDF-3X and gSpan. In the following sections, we describe how we used these projects in our research.

A.1 RDF-3X

We use RDF-3X for implementing our triple filtering method and conduct our experiments by comparing RDF-3X. RDF-3X provides the basic RDF store functionalities like storage and indices for RDF data, the cost-based query optimizer and the query executor. RDF-3X stores the RDF graphs in the six triple indices, which are implemented as the clustered B+ tree index. RDF-3X is managed as an open source project and available for non-commercial usage. We can download RDF-3X from `https://code.google.com/p/rdf3x/`.

RDF-3X has several relational operators like RDBMS; i.e example Scan, Join, Aggregation, et. al. Each operator extends Operator class. This class is the base

class of all operator and provides the iterator model interface, which consists of first(), next() method. The following is the skeleton of Operator class. We omit some irrelevant methods or fields.

```
/// Base class for all operators of the runtime system
class Operator
{
   public:

   /// Constructor
   explicit Operator(double expectedOutputCardinality);
   /// Destructor
   virtual ~Operator();

   /// Produce the first tuple
   virtual unsigned first() = 0;
   /// Produce the next tuple
   virtual unsigned next() = 0;
};
```

We also implement two operator classes for the triple filtering by extending this Operator class; RFLT operator and RFLTM operator. These class are similar to the Operator class except that they have fields for the triple filtering. RFLT operator conducts the triple filtering for a single scan operator, and RFLTM operator conducts the triple filtering for several scan operators and also conducts merge-join the all child operators. Because RFLTM performs the join, it has more fields and methods than RFLT operator.

```
class RFLT : public Operator
{
   std::vector<RGindex::Node*>* fltInfos;
   Operator* input;
   Register* inputValue;
   RPathFilter *rpathFLT;
   Register* fltValue;
```

```
   unsigned count;

   public:
   RFLT(Operator* input,
        std::vector<RGindex::Node*>* fltInfos,
        std::vector<unsigned> nodeIds,
        std::vector<GSPAN::DFSCode> &dfscodes)
   ~RFLT();
   /// Produce the first tuple
   unsigned first();
   /// Produce the next tuple
   unsigned next();
};
```

```
class RFLTM : public Operator
{
   /// The input
   std::vector<Operator*> childOp;
   /// The join attributes
   std::vector<Register*> childValue;
   /// The non-join attributes
   std::vector<std::vector<Register*>*> childTail;
   std::vector<unsigned> childTailSize;

   /// Buffers for cartesian product
   unsigned **buf_mat;
   unsigned *buf_tail_offset;
   unsigned *buf_next_offset;
   unsigned *buf_max_size;
   unsigned buf_width;

   unsigned *buf_size;
   unsigned *buf_offset;

   /// Buffers for next values
   unsigned **shadowValue;
   unsigned *cnts;
   unsigned resultCnt;
   unsigned *currentCnts;
```

```
   unsigned numOfChildren;
   int *childStatus;

   /// Filter info
   std::vector<RPathTreeIndex::Node*>* fltInfos;
   Register* inputValue;
   RPathFilter *rpathFLT;
   unsigned* fltValue;
   unsigned joinedValue;
   bool hasPrimaryChild;
   unsigned primaryChild;
   unsigned primaryChildrowCnt;

   public:
   RFLT_M(std::vector<RGindex::Node*>* fltInfos,
          std::vector<unsigned> &nodeIds,
          std::vector<GSPAN::DFSCode> &dfscodes_i,
          double expectedOutputCardinality);
   ~RFLT_M();
   /// Produce the first tuple
   unsigned first();
   /// Produce the next tuple
   unsigned next();

   void addChild(Operator *child, Register *value,
                 std::vector<Register*> *tail);
   unsigned getNextMergedValue(unsigned fltNextVal,
                               unsigned &joinedNextVal);
};
```

The filter information is provided for RFLT and RFLTM operators using RPathFilter class. It reads the segment for the specified Vlists and provides the vertex IDs. It has an interface similar to the iterator model.

```
class RPathFilter
{
   std::vector<RPathSegment *> rpathSgmts;
   std::vector<GSPAN::DFSCode> dfscodes;
```

```
    std::vector<double> cards;
    std::vector<unsigned> sizes;
    std::vector<unsigned> nodeIDs;
    unsigned *values;//, curVal;
    unsigned size;
    RPathSegment::Scan *sgmtScans;
    bool GetNextValue(unsigned value);

    public:
    RPathFilter(std::vector<RGindex::Node*>* fltInfo,
                std::vector<unsigned> &nodeIds,
                std::vector<GSPAN::DFSCode> &dfscodes);
    ~RPathFilter();
    bool CheckFilter(unsigned value);
    bool first();
    bool next(unsigned value);
    Register fltValue;
};
```

A.2 gSpan

The authors of gSpan provide the gSpan binary code at http://www.cs.ucsb.edu/~xyan/software/gSpan.htm. And the C++ source code of gSpan can be downloaded at http://www.nowozin.net/sebastian/gboost/. The code is included in the gBoost which is the software package for classification of graphs. It includes following functionalities (from its web site).

1. Discriminative Subgraph Mining
2. Frequent Subgraph Mining (gSpan)
3. Subgraph-Graph isomorphism test (through VFlib)
4. nu-LPBoost 2-class classifier

5. nu-LPBoost 1.5-class classifier

6. simple wrappers to easily train a classifier for graphs

The gSpan code is written by Taku Kudo. The software is dual-licensed under both the GNU General Public License, version 2 and the Mozilla Public License, version 1.1.

We do not use the entire gSpan code. Rather, we use the DFS code class and the generation code of the minimum DFS sequence of a graph pattern. We adapt the codes to be able to handle the directed labeled graph. Following is the classes of the DFS code and the Graph.

```
class DFS {
public:
    int src;
    int dest;
    int elabel;
    edge_type_t type;

    friend bool operator < (const DFS &d1, const DFS &d2) {
        if (d1.src<d2.src ||
            (d1.src==d2.src && d1.dest<d2.dest) ||
            (d1.src==d2.src && d1.dest==d2.dest &&
             d1.elabel<d2.elabel) ||
            (d1.src==d2.src && d1.dest==d2.dest &&
             d1.elabel==d2.elabel && d1.type<d2.type))
           return true;
        return false;
    }
};

struct DFSCode: public std::vector<DFS> {
public:
    bool is_min(void);

    /* Convert current DFS code into a graph. */
```

```
    void toGraph(Graph &g);

    void push(int src, int dest, int elabel, edge_type_t type) {
        resize(size() + 1);
        DFS &d = (*this)[size()-1];

        d.src = src;
        d.dest = dest;
        d.elabel = elabel;
        d.type = type;
    }
    bool hasSameDFS(int src, int elabel, edge_type_t type) {
       for(std::vector<DFS>::iterator iter=begin(), limit=end();
           iter!=limit; iter++) {
          DFS dfs=(*iter);
          if(dfs.src==src && dfs.elabel==elabel &&
             dfs.type==type)
             return true;
          if(dfs.dest==src && dfs.elabel==elabel &&
             dfs.type!=type)
             return true;
       }
       return false;
    }
};
```

```
class Vertex {
public:
    std::vector<Edge> edge;

    void push(int from, int to, int elabel, edge_type_t type) {
        edge.resize(edge.size()+1);
        edge[edge.size()-1].from = from;
        edge[edge.size()-1].to = to;
        edge[edge.size()-1].elabel = elabel;
        edge[edge.size()-1].type = type;
        return;
    }

    void push(int from, int to, int elabel,
```

```
            edge_type_t type, unsigned int id) {
        edge.resize(edge.size()+1);
        edge[edge.size()-1].from = from;
        edge[edge.size()-1].to = to;
        edge[edge.size()-1].elabel = elabel;
        edge[edge.size()-1].type = type;
        edge[edge.size()-1].id = id;
        return;
    }
};
```

Appendix B

Data Structure of RP-index and RG-index

In this section, we describe the data structure of RP-index and RG-index.

B.1 RP-index

RP-index is a data structure which has a trie structure. The predicate path is implemented as PPath class, which has a list of predicates. RPathTreeIndex class is a tree data structure whose node is implemented as the Node structure. It has methods for inserting nodes and searching nodes by predicate paths.

```
class PPath {
   public:
   PPath() {};
   PPath(PPath& pp, unsigned predicate);
   void Add(unsigned predicate);
   std::vector<unsigned> path;
};
```

```
class RPathTreeIndex {
   public:
   struct Node {
      unsigned predicate;
      unsigned startPage;
      unsigned startIndexPage;
      unsigned cardinality;
      RPathSegment *rpathSgm;
      std::map<unsigned, Node *> *children;
   };
   RPathTreeIndex(char *dataset, char *path,
                  unsigned maxL);
   Node* SearchNode(PPath& ppath);
   Node* InsertIntoIdx(unsigned startPage,
                       unsigned startIdxPage,
                       unsigned cardinality, unsigned byte,
                       const char *ppathStr);
};
```

B.2 RG-index

RG-index also has a structure similar to RG-index. All DFS codes for mined subgraph patterns are also organized in a trie. RG-index class is the base class for RG-index, and it also has methods for inserting and searching nodes. In addition, it has a method for building RG-index, which actually conduct the subgraph mining algorithm.

```
class RGindex {
public:
   struct Node {
      vector<unsigned> offsets;
      vector<unsigned> blks;
      vector<unsigned> cardinalities;
      vector<RPathSegment *> segments;
      std::map<GSPAN::DFS, Node *> children;
   };
```

```
private:
   unsigned minSup, maxL;
   Database *db;
   DictionarySegment *dictionary;
   map<unsigned, unsigned> suppMap;

   void subgraphMining(unsigned maxL, GSPAN::DFSCode dfscode,
                       InterResultTuple &results);
   void InsertIntoTree(Node* root, GSPAN::DFSCode dfscode,
                       unsigned level, Node* newNode);
   void expansion(set<unsigned> &preds, GSPAN::DFSCode &dfscode,
                  unsigned source, unsigned newVertexID,
                  GSPAN::edge_type_t edgetype,
                  InterResultTuple *old_results,
                  hash_tbl_t &hash_tbl,
                  bool storeResults, unsigned maxL,
                  bool rightmost, unsigned curMinSup);
public:
   PredMap predMap_new, predMap_old;
   Node root;
   MemoryMappedFile file;
   RGindex(char *dataset, char *path);
   /* insert the graph corresponding to the dfscode
      into RG-index and materialize the Vlists */
   void insert(GSPAN::DFSCode &dfscode,
               InterResultTuple &results);
   void build(Database &db, unsigned maxL, unsigned minSup);
   RGindex::Node* SearchNode(Node& root,
                             GSPAN::DFSCode& dfscode,
                             unsigned level);
};
```

Appendix C

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.
```

Bibliography

[1] G. Klyne, J. J. Carroll, Resource description framework (RDF): Concepts and abstract syntax, W3c recommendation, World Wide Web Consortium (2004).

[2] E. Prud'hommeaux, A. Seaborne, SPARQL query language for RDF, W3c recommendation, W3C Recommendation (2008).

[3] F. Belleau, M.-A. Nolin, N. Tourigny, P. Rigault, J. Morissette, Bio2RDF: towards a mashup to build bioinformatics knowledge systems., Journal of biomedical informatics 41 (5) (2008) 706–16.

[4] N. Redaschi, U. Consortium, UniProt in RDF: Tackling Data Integration and Distributed Annotation with the Semantic Web, Nature Precedings.

[5] G. Kobilarov, T. Scott, Y. Raimond, S. Oliver, C. Sizemore, M. Smethurst, C. Bizer, R. Lee, Media meets semantic web — how the bbc uses dbpedia and linked data to make connections, in: Proceedings of the 6th European Semantic Web Conference on The Semantic Web(ESWC'09), 2009, pp. 723–737.

[6] J. Hoffart, F. M. Suchanek, K. Berberich, G. Weikum, YAGO2: A spatially and temporally enhanced knowledge base from Wikipedia, Artif. Intell. 194 (2013) 28–61.

[7] P. Mika, Social Networks and the Semantic Web, in: Proceedings of International Conference on Web Intelligence (WI'04), 2004, pp. 285–291.

[8] J. Sheridan, Linking UK government data, in: WWW Workshop on Linked Data, 2010, pp. 1–4.

[9] C. Bizer, T. Heath, T. Berners-Lee, Linked data - the story so far, Int. J. Semantic Web Inf. Syst. 5 (3) (2009) 1–22.

[10] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, K. Wilkinson, Jena: implementing the semantic web recommendations, in: Proceedings of the 13th international conference on World Wide Web - Alternate Track Papers & Posters (WWW 2004), 2004.

[11] J. Broekstra, A. Kampman, F. van Harmelen, Sesame: A generic architecture for storing and querying RDF and RDF Schema, in: Proceedings of the First International Semantic Web Conference (ISWC 2002), 2002.

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

[13] O. Erling, I. Mikhailov, RDF Support in the Virtuoso DBMS, in: Proceedings of the 1st Conference on Social Semantic Web (CSSW 2007), 2007.

[14] T. Neumann, G. Weikum, RDF-3X: a RISC-style engine for RDF, PVLDB 1 (1) (2008) 647–659.

[15] A. Chebotko, S. Lu, F. Fotouhi, Semantics preserving SPARQL-to-SQL translation, Data Knowl. Eng. 68 (10) (2009) 973–1000.

[16] K. Kim, B. Moon, H.-J. Kim, 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.

[17] K. Kim, B. Moon, H.-J. Kim, R3f: Rdf triple filtering method for efficient sparql query processing, World Wide Web (2013) 1–41.

[18] R. Goldman, J. Widom, DataGuides: Enabling query formulation and optimization in semistructured databases, in: Proceedings of 23rd International Conference on Very Large Data Bases (VLDB 1997), 1997.

[19] T. Milo, D. Suciu, Index structures for path expressions, in: Proceedings of the 7th International Conference on Database Theory (ICDT 1999), 1999.

[20] R. Kaushik, P. Shenoy, P. Bohannon, E. Gudes, Exploiting local similarity for indexing paths in graph-structured data, in: Proceedings of the 18th International Conference on Data Engineering (ICDE 2002), 2002.

[21] C. Qun, A. Lim, K. W. Ong, 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.

[22] H. He, J. Yang, Multiresolution indexing of XML for frequent queries, in: Proceedings of the 20th International Conference on Data Engineering (ICDE 2004), 2004.

[23] X. Yan, P. S. Yu, J. Han, Graph indexing based on discriminative frequent structure analysis, ACM Trans. Database Syst. 30 (4) (2005) 960–993.

[24] X. Yan, J. Han, gspan: Graph-based substructure pattern mining, in: Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002), 2002.

[25] C. Weiss, P. Karras, A. Bernstein, Hexastore: sextuple indexing for semantic web data management, PVLDB 1 (1) (2008) 1008–1019.

[26] P. Hayes, B. McBride, RDF Semantics, W3c recommendation, World Wide Web Consortium (2004).

[27] D. L. McGuinness, F. van Harmelen, OWL Web Ontology Language overview, W3c recommendation, World Wide Web Consortium (2004).

[28] A. Owens, A. Seaborne, N. Gibbins, Clustered TDB: A clustered triple store for Jena, Tech. rep., University of Southampton (2008).

[29] L. Sidirourgos, R. Goncalves, M. L. Kersten, N. Nes, S. Manegold, Column-store support for RDF data management: not all swans are white, PVLDB 1 (2) (2008) 1553–1563.

[30] M. Atre, V. Chaoji, M. J. Zaki, J. A. Hendler, 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.

[31] H. Huang, C. Liu, X. Zhou, Approximating query answering on RDF databases, World Wide Web 15 (1) (2012) 89–114.

[32] R. D. Virgilio, P. D. Nostro, G. Gianforme, S. Paolozzi, A Scalable and Extensible Framework for Query Answering over RDF, World Wide Web 14 (5-6) (2011) 599–622.

[33] O. Udrea, A. Pugliese, V. S. Subrahmanian, GRIN: A graph based RDF index, in: Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI 2007), 2007.

[34] M. Bröcheler, A. Pugliese, V. S. Subrahmanian, DOGMA: A disk-oriented graph matching algorithm for RDF databases, in: Proceedings of the 8th International Semantic Web Conference (ISWC 2009), 2009.

[35] T. Tran, G. Ladwig, Structure index for RDF data, in: Workshop on Semantic Data Management (SemData@VLDB2010), 2010.

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

[37] Z. Sun, H. Wang, H. Wang, B. Shao, J. Li, Efficient subgraph matching on billion node graphs, PVLDB 5 (9) (2012) 788–799.

[38] M. F. Husain, J. P. McGlothlin, M. M. Masud, L. R. Khan, B. M. Thuraisingham, Heuristics-based query processing for large RDF graphs using cloud computing, IEEE Trans. Knowl. Data Eng. 23 (9) (2011) 1312–1327.

[39] K. Rohloff, R. E. Schantz, High-performance, massively scalable distributed systems using the mapreduce software framework: the shard triple-store, in: SPLASH Workshop on Programming Support Innovations for Emerging Distributed Applications, ACM, 2010.

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

[41] R. Punnoose, A. Crainiceanu, D. Rapp, Rya: a scalable RDF triple store for the clouds, in: 1st International Workshop on Cloud Intelligence (colocated with VLDB 2012) (Cloud-I 2012), ACM, 2012.

[42] E. I. Chong, S. Das, G. Eadon, J. Srinivasan, An efficient SQL-based RDF querying scheme, in: Proceedings of the 31st International Conference on Very Large Data Bases (VLDB 2005),, ACM, 2005.

[43] T. Neumann, G. Weikum, Scalable join processing on very large RDF graphs, in: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2009), 2009.

[44] P. A. Bernstein, D.-M. W. Chiu, Using semi-joins to solve relational queries, J. ACM 28 (1) (1981) 25–40.

[45] F. Bancilhon, D. Maier, Y. Sagiv, J. D. Ullman, 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.

[46] M.-S. Chen, H.-I. Hsiao, P. S. Yu, On applying hash filters to improving the execution of multi-join queries, VLDB J. 6 (2) (1997) 121–131.

[47] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, T. G. Price, 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.

[48] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, D. Reynolds, SPARQL basic graph pattern optimization using selectivity estimation, in: Proceed-

ings of the 17th International Conference on World Wide Web (WWW 2008), 2008.

[49] A. Maduko, K. Anyanwu, A. P. Sheth, P. Schliekelman, Graph summaries for subgraph frequency estimation, in: Proceedings the 5th European Semantic Web Conference (ESWC 2008), 2008.

[50] T. Neumann, G. Moerkotte, Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins, in: Proceedings of the 27th International Conference on Data Engineering (ICDE 2011), 2011.

[51] K.-F. Wong, J. Yu, N. Tang, Answering XML queries using path-based indexes: A survey, World Wide Web 9 (3) (2006) 277–299.

[52] G. Gou, R. Chirkova, Efficiently querying large XML data repositories: A survey, Knowledge and Data Engineering, IEEE Transactions on 19 (10) (2007) 1381 –1403.

[53] M. Kuramochi, G. Karypis, Finding frequent patterns in a large sparse graph, in: Proceedings of the Fourth SIAM International Conference on Data Mining (SDM 2004), 2004.

[54] D. Shasha, J. T.-L. Wang, R. Giugno, 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.

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

[56] H. He, A. K. Singh, 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.

[57] S. Zhang, S. Li, J. Yang, GADDI: distance index based subgraph matching in biological networks, in: Proceedings of the 12th International Conference on Extending Database Technology (EDBT 2009), 2009.

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

[59] H. Cheng, X. Yan, J. Han, Mining graph patterns, in: C. C. Aggarwal, H. Wang (Eds.), Managing and Mining Graph Data, Vol. 40 of Advances in Database Systems, Springer, 2010, pp. 365–392.

[60] G. Graefe, Query evaluation techniques for large databases, ACM Comput. Surv. 25 (2) (1993) 73–170.

[61] H. Köhler, Estimating set intersection using small samples, in: Proceedings of the Thirty-Third Australasian Computer Science Conference (ACSC 2010), 2010.

[62] T. Neumann, G. Weikum, x-RDF-3X: Fast Querying, High Update Rates, and Consistency for RDF Databases, PVLDB 3 (1) (2010) 256–263.

[63] M. Morsey, J. Lehmann, S. Auer, A.-C. N. Ngomo, DBpedia SPARQL benchmark - performance assessment with real queries on real data, in: Proceedings of the 10th International Semantic Web Conference (ISWC 2011), 2011.

[64] Y. Guo, Z. Pan, J. Heflin, LUBM: A benchmark for OWL knowledge base systems, J. Web Sem. 3 (2-3) (2005) 158–182.

[65] M. Schmidt, T. Hornung, G. Lausen, C. Pinkel, SP2Bench: A SPARQL Performance Benchmark, in: Proceedings of the 25th International Conference on Data Engineering (ICDE 2009), 2009.

[66] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, S. Hellmann, DBpedia - a crystallization point for the Web of Data, J. Web Sem. 7 (3) (2009) 154–165.

[67] C. Fellbaum (Ed.), WordNet An Electronic Lexical Database, The MIT Press, 1998.

[68] G. Moerkotte, T. Neumann, G. Steidl, Preventing bad plans by bounding the impact of cardinality estimation errors, PVLDB 2 (1) (2009) 982–993.

[69] B. Bringmann, S. Nijssen, What is frequent in a single graph?, in: Advances in Knowledge Discovery and Data Mining, 12th Pacific-Asia Conference, PAKDD 2008, 2008, pp. 858–863.

[70] M. Fiedler, C. Borgelt, Subgraph support in a single large graph, in: Workshops Proceedings of the 7th IEEE International Conference on Data Mining (ICDM 2007), 2007.

요약

최근 RDF 그래프 데이터가 증가하며 대용량 RDF 그래프 데이터에 대한 질의 처리 방법이 큰 관심을 받고 있다. 관계형 RDF 저장소는 RDF 그래프를 관계형 테이블에 저장하고 조인 연산을 통해 RDF 그래프 질의를 처리한다. 하지만 이런 질의 처리 방법은 여러 조인 연산이 포함되기 때문에, 조인 연산 간에 발생하는 중간 결과를 효과적으로 처리하는 방법이 필수적이다.

이런 문제를 해결하기 위해 본 학위 논문에서는 관계형 RDF 저장소에서 그래프 구조 정보를 활용해 효과적으로 불필요한 중간 결과를 제거할 수 있는 트리플 필터링 기법을 제안하였다. 기존의 방법과는 달리 본 논문에서 제안한 기법은 관계형 RDF 저장소에 그래프 인덱스 기법을 효율적으로 적용할 수 있는 방안을 제안했다.

RDF 그래프의 구조 정보를 효율적으로 인덱싱하고 저장할 수 있는 자료 구조로 RDF 경로 인덱스(RP-index) 와 RDF 그래프 인덱스(RG-index)를 제안했다. 이 두 인덱스는 각각 경로 정보와 그래프 정보를 활용해 효율적으로 필터링 데이터를 제공하며, 이를 이용해 질의 처리 과정에서 최종 결과에 포함되지 않을 트리플을 미리 제거할 수 있다. 또한, 각각의 인덱스를 만드는 과정에서 불필요한 중간 계산을 줄이는 방안과 점진적으로 인덱스를 갱신할 수 있는 기법도 제안했다. 특히 RG-index에 대해서는 서브그래프 패턴 마이닝 기법을 응용해 효율적으로 인덱스를 생성하는 방법을 제공했다.

하지만 RDF 그래프에 존재하는 경로 패턴과 그래프 패턴의 개수가 지수적으로 증가하기 때문에 이런 인덱스를 만들고 유지하는데 있어서는 인덱스 크기를 효과적으로 제어하는 방법이 필요하다. 본 논문에서는 필터링 효과를

유지하며 인덱스의 크기를 줄일 수 있는 방법을 제안하였다. 또한, 트리플 필터링을 효율적으로 수행하기 위한 RDF 필터 연산자도 제안하였다. 이 연산자는 머지 프로세스를 통해 필터링에 수반되는 오버헤드를 크게 줄여 기존 질의 성능에 영향을 최소화한다.

마지막으로 여러 가지 벤치마크 데이터와 실제 데이터를 활용한 다양한 실험을 통해 제안된 방법이 질의 처리 성능을 크게 향상 시킬 수 있음을 보였다.

주요어: RDF, SPARQL, Query Optimization, RDF Store, Triple Filtering, Intermediate Results

학번: 2003-23569

Acknowledgements

연구실에서 힘들 때마다 선배님들의 박사 논문을 펼쳐 감사의 글을 읽고는 했습니다. 그 글들을 읽으며 지금 힘든 과정이 결코 나만 그렇지 않다는 위로를 받을 수 있었습니다. 또한, 선배님들이 그랬듯 저 역시 이 길을 혼자 가는 것이 아니라 많은 분의 도움과 사랑이 있기에 가능하다는 것을 깨닫고 힘을 낼 수 있었습니다. 짧은 지면이지만 이 글을 통해 제가 여기까지 올 수 있도록 함께 해주신 분들께 감사의 마음을 전하고자 합니다.

우선 여러 가지로 부족했던 저를 학문의 세계로 받아주시고 이끌어 주신 김형주 교수님께 진심으로 감사드립니다. 그리고 멀리 미국에서 저의 연구를 도와주시고 저의 학위 심사에도 참여해 주신 문봉기 교수님께 깊은 감사를 드립니다. 또한, 귀중한 시간을 내어 논문 심사를 해주신 이상구 교수님, 김선 교수님, 임동혁 교수님께도 감사드립니다. 그리고 힘들 때마다 항상 방향을 잃지 않도록 도와주신 이상원 교수님께도 감사드립니다.

IDB 연구실에서 맺은 귀중한 인연은 저에게 큰 힘이 됐습니다. 학위 과정이 길어지며 힘든 점도 많았었지만, 그만큼 많은 선후배님을 만날 수 있어 좋았습니다. 힘들 때 서로 돕고 격려해주고 때로는 술잔도 같이 기울이며 많은 추억을 만들 수 있었습니다. 일일이 나열하기 힘들 정도로 많은 선후배님과 동기들에게 깊은 감사를 드립니다. 특히 같이 학위 심사를 하며 여러 가지로 도움을 많이 준

태휘와 현우에게 감사드립니다. 지금 힘든 과정을 걷고 있을 후배들도 힘내고 서로 격려하며 즐겁게 학위 과정을 하길 바랍니다.

그리고 힘든 시기마다 힘이 되어준 죽마고우인 친구들의 우정에 감사드립니다. 저는 이십 대와 삼십 대의 많은 부분을 박사 과정에서 보냈습니다. 그러며 여러 가지 변화도 겪고 인생의 문제도 만났었지만 이런 과정을 함께 겪고 얘기할 수 있었던 친구들이 있었기에 힘들지 않았습니다. 우리의 우정이 변함없이 계속되기를 바랍니다.

또한, 티베로의 박상영 본부장님과 박헌영 팀장님 그리고 소중한 팀원분들께 감사드립니다. 많은 것을 배울 수 있었고 박사 심사 동안 여러 가지로 바쁜 시간에 이해해 주신 점 깊이 감사드립니다. 항상 즐겁게 일할 수 있어서 행운으로 생각하고 있습니다.

마지막으로 항상 저를 응원해주시고 사랑해주신 부모님과 아내, 장인, 장모님과 가족 친지 분들께 감사드립니다. 계속 길어지고 끝이 나지 않을 것 같은 학위 과정을 묵묵히 기다려주신 이분들의 사랑이 없었다면 절대로 가능하지 않았을 것입니다. 그리고 항상 듬직한 형과 큰형, 작은형, 형수 님들, 은영이 누나와 원이 그리고 조카들에게도 감사드립니다.