World Wide Web (2023) 26:661–681
https://doi.org/10.1007/s11280-022-01103-5

FPIRPQ: Accelerating regular path queries on knowledge graphs

Xin Wang[1] · Wenqi Hao[1] · Yuzhou Qin[1] · Baozhu Liu[1] · Pengkai Liu[1] · Yanyan Song[1] · Qingpeng Zhang[2] · Xiaofei Wang[1]

Received: 22 April 2022 / Revised: 13 August 2022 / Accepted: 8 September 2022 /
Published online: 7 October 2022

Abstract
With the growing popularity and application of knowledge-based artificial intelligence, the scale of knowledge graph data is dramatically increasing. As an essential type of query for RDF graphs, Regular Path Queries (RPQs) have attracted increasing research efforts, which explore RDF graphs in a navigational manner. Moreover, path indexes have proven successful for semi-structured data management. However, few techniques can be used effectively in practice for processing RPQ on large-scale knowledge graphs. In this paper, we propose a novel indexing solution named FPIRPQ (Frequent Path Index for Regular Path Queries) by leveraging Frequent Path Mining (FPM). Unlike the existing approaches to RPQs processing, FPIRPQ takes advantage of frequent paths, which are statistically derived from the data to accelerate RPQs. Furthermore, since there is no explicit benchmark targeted for RPQs over RDF graph yet, we design a micro-benchmark including 12 basic queries over synthetic and real-world datasets. The experimental results illustrate that FPIRPQ improves the query efficiency by up to orders of magnitude compared to the state-of-the-art RDF storage engine.

Keywords Knowledge graphs · Regular path queries · Path index

1 Introduction

With the proliferation of Knowledge Graphs (KG) in recent years, the applications of KGs have a rapid growth in diverse domains, such as biology [1–3], finance [4, 5], and education [6, 7]. In the Semantic Web community, the *Resource Description Framework* (RDF) [8] has been extensively applied and becomes a de-facto standard format for KGs. Moreover, as an essential type of query for RDF graphs, RPQs have attracted increasing research efforts. RPQs explore RDF graphs in a navigational manner, which is indispensable in

This article belongs to the Topical Collection: *APWeb-WAIM 2021*
Guest Editors: Yi Cai, Leong Hou U, Marc Spaniol, Yasushi Sakurai

✉ Xiaofei Wang
xiaofeiwang@tju.edu.cn

Extended author information available on the last page of the article

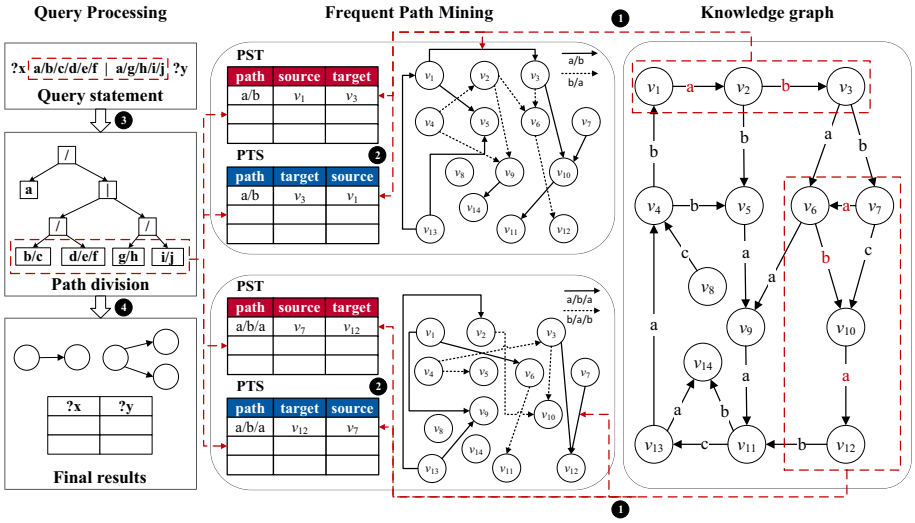


Fig. 1 The overflow of FPIRPQ

most graph query languages. Furthermore, as the standard query language on the RDF graphs, SPARQL 1.1 [9] provides the *property path* [10] feature, which is actually an implementation of RPQ semantics. In addition, the result of an RPQ $Q = (x,r,y)$ over an RDF graph G is a set of pairs of resources (v_0,v_n) such that there exists a path ρ in G from v_0 to v_n, where the label of ρ, denoted by $\bar{l}_\rho$, satisfies the regular expression r in Q.

Although theoretical aspects of RPQs have been well studied and several approaches are proposed to accelerate RPQs by leveraging prune and filter techniques [11], few methods can be used effectively in practice for RPQ evaluation and optimization. Moreover, the existing approaches focus more on optimizing query processing, rather than considering utilizing the statistical characteristics of data, resulting in the differences in query performance over different KGs. Compared with other methods with path index, FPIRPQ takes advantage of the frequent paths existing in the KGs to build the path index, which will alleviate the difference in query performance over datasets since the indexes are adaptive to the data.

In order to exploit and manage the statistical features of the data, we adopt the path indexing technique, which has been successfully employed in the field of semi-structured data management. The whole procedure of constructing the path index and further query processing by FPIRPQ is illustrated in Figure 1. The frequent paths that exhibit in the KG are utilized for constructing the index (@*IL*@❶ in Figure 1). It is worth noting that FPIRPQ only captures frequent paths without indexing rare paths, which will save the storage space required. Furthermore, for handling different kinds of queries, two tables, `PST` and `PTS`, are constructed to record the label strings of frequent paths, source vertices, and target vertices, the first two columns of which are indexed using B-tree (@*IL*@❷ in Figure 1). Moreover, the label strings of paths extracted from query statements are divided into several indexed substrings to improve the query efficiency in FPIRPQ, which is described in detail in Section 5 (@*IL*@❸ in Figure 1). Afterwards, the processed query statements are executed utilizing the path index to obtain the results which match the specified paths, and these results are combined further to generate the relational table recording the final results (@*IL*@❹ in Figure 1).

This paper is expanded on the PAIRPQ: An Efficient Path Index for Regular Path Queries on Knowledge Graphs [12]. Compared to the conference version, our contributions can be summarized as follows:

1. We propose a novel path index method over frequent paths to accelerate RPQs, i.e., FPIRPQ, which makes full use of statistics of the underlying KGs. With the path indexes built by FPIRPQ, the label string of RPQ is separated into several indexed subparts, which will contribute to reducing intermediate calculations and improving query efficiency.
2. In Related Work, we supplement a summary of existing algorithms of frequent graph pattern mining and analyze their respective advantages and disadvantages.
3. In Preliminaries, the definition of Path is added and we refine the original definition and its examples, which is described in more detail for better understanding.
4. For the FPM algorithm, we describe how the FPM algorithm we proposed was inspired by existing algorithms and how it can be adapted to KG. Moreover, the differences between our algorithm and existing algorithms are analyzed in detail.
5. For the method of query processing, we introduce a novel greedy algorithm and explain the detail of it combined with examples.
6. In Experiments, we have first extended the micro-benchmark query set that includes 12 queries and more experiments have been conducted on more datasets to verify the effectiveness and efficiency of the proposed approach. In addition, we have added a comparative description to provide a more detailed analysis of the experimental results.

The rest of this paper is organized as follows. We reviewed the related works in Section 2. Section 3 provides the fundamental definitions that form the background for this work. We describe the path index method, i.e., FPIRPQ, for RPQs on RDF graphs in Section 4 and introduce the greedy algorithm for query processing in detail in Section 5. Section 6 shows the experimental results, and we conclude in Section 7.

2 Related Work

In this section, we first review the related works on the algorithm of graph mining, which focus mainly on mining frequent patterns from graphs. Furthermore, the existing work on path index and RDF storage engine applied to knowledge graphs are concluded.

2.1 Frequent graph pattern mining

With the surge of graph data, graph mining has gained much attention in the last few decades. Moreover, it has long been recognized that frequent graph-based patterns can be applied effectively to many significant tasks in graph database management. To mine the frequent connected subgraphs efficiently, gSpan [13] adopts the depth-first search strategy, but it is designed to handle the mining issue in the graph-transaction setting, where the input is a graph dataset rather than a single graph. In contrast with gSpan, SUBDUE [14] is proposed to mine the frequent subgraphs in a single graph. By substituting patterns with a single vertex, the original graph will be compressed and the efficiency of the algorithm in SUBDUE will be improved further by exploiting approximations. However, the poor scalability of SUBDUE limits its performance on large-scale datasets. Ghazizadeh et al. [15]

propose an algorithm called SeuS, which utilizes a data structure summary to collapse all vertices of the same label together and prune infrequent candidates. While the algorithm shows good performance in the presence of a relatively small number of highly frequent subgraphs, it will be extremely expensive to compute when handling a large number of frequent subgraphs with low frequency. Inspired by the related works above, we propose a greedy algorithm to mine the frequent path on KG and support the construction of path index in this paper.

2.2 Path Index

There is a set of existing works on path indexing for graph query evaluation, which can be classified into the following categories: (1) DataGuide, (2) T-index, and (3) k-bisimilarity index.

DataGuide. For a given data graph G_d, a DataGuide [16, 17] is a summary graph G_s where each label path of G_d has exactly one corresponding data path instance in G_s, and every label path of G_s is a label path of G_d. Moreover, it has been proved that the process of creating a DataGuide over a source database is equivalent to the conversion of a non-deterministic finite automaton (NFA) to a deterministic finite automaton (DFA) [18]. Nevertheless, DataGuide requires a powerset construction over the underlying database, which in the worst case can be of exponential cost. Furthermore, DataGuides just follows the original structural information in the data graphs rather than further processing. However, the weights attached to different paths should be related to their frequency of occurrence in data graphs when constructing indexes, which is also one of the contributions in FPIRPQ.

T-Index. T-index [19] (*template index*) is presented to answer queries for specified path templates. 1-index [19] is the most simple T-index, which is also a labeled summary graph G_s like DataGuide. The nodes in G_s are equivalence classes of nodes in the data graph G_d such that for each edge in G_d there exists an edge in G_s, which results that the structure of 1-index are more compact than DataGuide. In the 2-index, every node represents the equivalence class for a 2-length path. T-index, which generalises 1-index and 2-index, only constructs indexes on paths that are queried frequently in order to reduce the space required for indexing. Therefore, in order to identify the paths which should be indexed, query logs are indispensable when building T-index, however, the query logs are not available in several datasets, which hinders the wider application of the T-index. Compared with T-index, the frequent paths required can be extracted from the data graphs in FPIRPQ, which increases the availability of FPIRPQ.

k*-*Bisimilarity Index. As noted above, both the DataGuide and T-index are designed to answer RPQs accurately, which results in the increased size and complexity with little added value. To overcome the limitations of DataGuide and T-index, A(k)-index [20], D(k) [21], and M(k)-index

[22] are proposed based on k-bisimilarity. In the A(k)-index, paths are grouped based on their length. For queries whose corresponding paths are no longer than k, we can get the exact answers by A(k)-index, but for the paths longer than k, only approximate results can be obtained using A(k)-index. Moreover, D(k)-index, which is an adaptive path index, exploits the query load and dynamically adjusts the structure of summary graph to reduce the size of the index and improve performance. Furthermore, to avoid excessive refinement on irrelevant indexes or data vertices, M(k)-index allows different k values for different index vertices having the same label. In contrast with the methods above, we not only consider the statistical characteristics of the data graphs but also reduce the reliance on query logs, ensuring the applicability of FPIRPQ.

2.3 RDF Storage Engine

With the widespread adoption of KGs in a diverse domain, a variety of RDF storage engines have emerged. On the top of the relational database, Virtuoso [23] is an RDF storage engine that implements the management of multi-model data. Moreover, Virtuoso supports the query languages of SQL and the property path feature defined in SPARQL 1.1. However, due to the shortcomings of its storage features, the query efficiency over Virtuoso is not satisfactory. Moreover, gStore [24] is a prototype system that supports the features of SPARQL 1.1 and accelerates query processing by utilizing VS*-tree. Nevertheless, gStore only supports a subset of features in SPARQL 1.1, particularly, the property path is beyond the capability of the gStore system. In addition, KGDB [25] is another prototype system that implements the efficient storage of both RDF graphs and property graphs. Furthermore, KGDB realizes semantic alignment of SPARQL and Cypher, which means basic queries of SPARQL and Cypher are all supported, and the property path is allowed in the system. However, no method of query optimization for RPQs is available in KGDB, which limits the query efficiency of RPQs. Considering the support for property paths, Virtuoso and KGDB are included in the experiments to verify the effectiveness and efficiency of FPIRPQ.

3 Preliminaries

In this section, we will define the concepts of relevant background knowledge and the main notations used throughout this paper are illustrated in Table 1.

Definition 1 (RDF Graph) Let U, B, and L denote Uniform Resource Identifiers (URI), blank nodes, and literals, respectively, which are three disjoint infinite sets. Then an RDF triple $(s,p,o) \in (U \cup B) \times U \times (U \cup B \cup L)$ is a statement of a fact, meaning that there exists a relation p between s and o or the value of property p for s is o, where s, p, and o represents the subject, predicate, and object, respectively, and an RDF graph is a finite set of RDF triples.

For a given RDF graph $G = (V,E,\Sigma)$, V, E, and Σ represent the set of vertices, edges, and edge labels in G, respectively. Formally, $V = \{s|(s,p,o) \in G\} \cup \{o|(s,p,o) \in G\}$, $E = \{(s,o)|(s,p,o) \in G\}$ and $\Sigma = \{p|(s,p,o) \in G\}$. Moreover, we define an infinite set of variables Var, which is disjoint from U and L. As shown in Figure 2, the example graph is an

Table 1 List of notations

Notation	Description
$t = (s,p,o)$	A triple in knowledge graph G
$\bar{l}_\rho$	The label string of a path ρ
S_ρ	The set of source vertices of path ρ
T_ρ	The set of target vertices of path ρ
$a^{-1}R$	The derivative of the regular expression R
minSup	Minimum support threshold for frequent path mining
P	The set of frequent paths
FL	The set of label strings of frequent paths

RDF graph consisting of 20 triples, which can be denoted as G. For example, (v_2, b, v_5) is an RDF triple as well as an edge labeled with b in G, where $V = \{v_i | 1 \leq i \leq 14\}$ and Σ_G = {a, b, c}.

Definition 2 (Path) Given an RDF graph $G = (V,E,\Sigma)$, a path represents a sequence $\rho = v_0 l_0 v_1 \cdots v_{n-1} l_{n-1} v_n$, where$(v_i,l_i,v_{i+1}) \in G$ for every $i \in \{0, \ldots, n-1\}$. Moreover, a path ρ can be seen as a set of triples in G, i.e., $\rho = \{(v_i, l_i, v_{i+1}) \in G \mid i \in \{0, \ldots, n-1\}\}$. In addition, we denote the label of a path ρ by $\bar{l}_\rho$, which is the string $l_0 \cdots l_{n-1} \in \Sigma^*$. For instance, $\rho = v_1 \mathtt{a} v_2 \mathtt{b} v_3$ is a path of the RDF graph in Figure 2, and $\bar{l}_\rho = \mathtt{ab}$.

Definition 3 (Regular Path Queries) For a given RDF graph $G = (V,E,\Sigma)$, consider $Q = (x,R,y)$ represents a regular path query where x and y are variables (i.e., $x,y \in Var$) and R is a regular expression over the alphabet Σ. Moreover, the regular expression R in G can be recursively defined as $R ::= \varepsilon | l | R/R | R|R | R^*$, where $l \in \Sigma$ and /, |, and * represents concatenation, alternation, and the Kleene 's closure, respectively. It is also allowed that the R/R^* and $\varepsilon|R$ can be denoted for short by R^+ and $R?$, respectively, and $L(R)$ denotes the language expressed by R. Therefore, the answer set of Q under the standard semantics is defined as $\{(x,y) \mid \exists$ *a path* ρ *in* G *from* x *to* y *s.t.* $\bar{l}_\rho \in L(R)\}$. In addition, the set of source vertices of path ρ is defined as $S_\rho = \{x \mid \exists$ *a path* ρ *in* G *from* x *to* y *s.t.* $\bar{l}_\rho \in L(R)\}$, while the set of target vertices is $T_\rho = \{y \mid \exists$ *a path* ρ *in* G *from* x *to* y *s.t.* $\bar{l}_\rho \in L(R)\}$.

As shown in the example graph in Figure 2, assume that $Q_1 = (x,\mathtt{a/b},y)$ is the regular path query, then (v_1,v_3) is in the result set of the query Q_1. Similarly, (v_1,v_2) and (v_2,v_3) are in the answer set corresponding to the query $Q_2 = (x,\mathtt{a|b},y)$, since both the path $\rho_1 = v_1 \mathtt{a}\ v_2$ and $\rho_2 = v_2 \mathtt{b}\ v_3$ satisfy the query. Moreover, as for Kleene 's closure (*), the query $Q_3 = (x, \mathtt{b}^*, y)$ can be utilized to denote that we want to query the set of vertex pairs (x,y), where there is a path whose labels are composed of one or more b, so that (v_2,v_3) and (v_2,v_7) are both results that satisfy the query.

Definition 4 (Frequent Path Mining) Assume that $G = (V,E,\Sigma)$ is an RDF graph and *minSup* represents a minimum support threshold, the aim of frequent path mining over G is to find a set of paths $P = \{\rho_1, \rho_2, \ldots, \rho_n\}$. P can be separated into m equivalence classes $C_1, C_2, \ldots, C_m$, within which each path has the same label strings, assuring that the number of paths in each equivalence class should be greater than *minSup*, i.e., $|C_1|, |C_2|, \ldots, |C_m| >$ *minSup*.

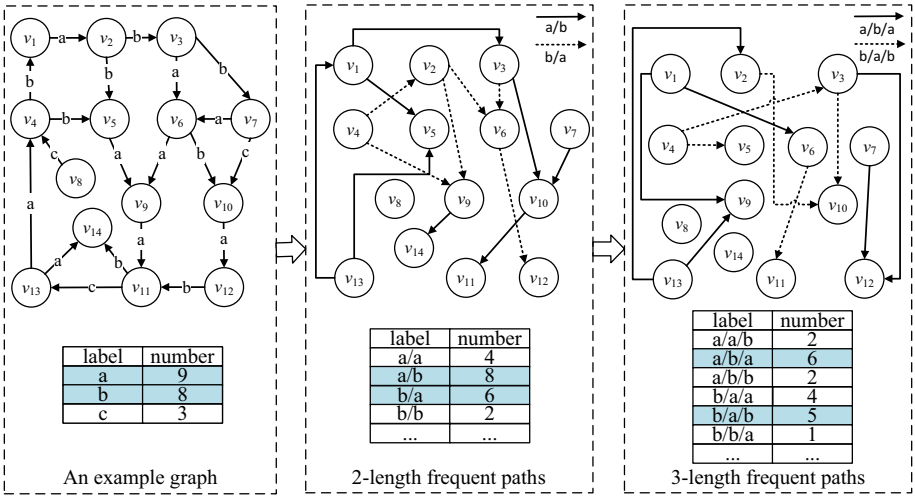


Fig. 2 Greedy FPM on KG

Definition 5 (Regular Expression Derivatives) The concept of the derivative of a regular expression is introduced to better analyze regular expressions with arbitrary logical connectives. For any given regular expression R and any string u, $u^{-1}R$ is used to denote the derivative of R with respect to u, which can be calculated recursively as follows:

$$(ua)^{-1}R = a^{-1}(u^{-1}R) \quad \text{for a symbol } a \text{ and a string } u$$
$$\varepsilon^{-1}R = R$$

Using the previous two rules, the derivative with respect to an arbitrary string is explained by the derivative with respect to a single-symbol string a. Meanwhile, considering several special forms of R (ε, ϕ and single-symbol string b), the derivative can be computed as follows:

$$a^{-1}\varepsilon = \phi$$
$$a^{-1}\phi = \varepsilon$$
$$a^{-1}b = \begin{cases} \varepsilon & \text{if } a = b \\ \phi & \text{otherwise} \end{cases}$$

For the more general case, the derivative of a regular expression with an arbitrary logical connective is calculated as shown below, the proof of which is described in more detail in [26].

$$a^{-1}(R)^* = (a^{-1}R)R^*$$
$$a^{-1}(R/S) = \begin{cases} (a^{-1}R)S + a^{-1}S & \text{if } R \text{ can be } \varepsilon \\ (a^{-1}R)S & \text{otherwise} \end{cases}$$
$$a^{-1}(R|S) = (a^{-1}R)|(a^{-1}S)$$

4 Frequent Path Index

In this section, we describe how to construct the path index that employs the frequent paths in graphs efficiently. For index construction, we have two steps: mining frequent paths on KGs and building the path index.

4.1 Frequent Path Mining

The idea of most FPM algorithms is to keep the possible substructures and return them if they are found to exceed a given limit during recursive traversal. Inspired by this, our algorithm also adheres to this algorithmic idea. However, as noted above, most FPM algorithms are designed to handle the issue of mining in the graph-transaction setting, where the input is a set of multiple data graphs. Meanwhile, given that our algorithm is required to be applicable to knowledge graphs of large scale, recursive traversal of all paths would significantly increase the execution time of the algorithm, which is why few single graph-based FPM algorithms [27] can be directly applied to KG. For instance, the performance of algorithm proposed by Vanetik et al. [28] has only been shown on data of very small scale (around 100 edges). Moreover, After analyzing a large number of query logs, it is concluded that SPARQL queries with a small number of triples (from 0 to 2) account for a significant share of the total number of queries per dataset generally [29]. Therefore, we propose a greedy FPM algorithm in this paper to extract the frequent paths from KGs and limit the depth of recursion in the algorithm to improve the scalability of the algorithm.

Algorithm 1 proposes the FPM method adopted in FPIRPQ. We compute the frequent path by a bottom-up approach, which computes first P^1, then P^2, and so forth up to P^k. To identify P^1, we first traverse all the triples in G to record the number of the appearance of each edge label (line 1-3), then filter paths whose labels occur less than or equal to *minSup* times (line 4). To obtain k-length frequent paths, we recursively join frequent paths from P^1 to P^k (line 6). To obtain P^i $(2 \leq i \leq k)$, the JOIN operation will be executed on each pair of label sequences $\bar{l}_1$ and $\bar{l}_2$, where $\bar{l}_1 \in FL^1$ and $\bar{l}_2 \in FL^{i-1}$ (line 9-10). If the number of occurrences of the candidate paths $P_{candidates}$ obtained after the JOIN operation is greater than *minSup* (line 11), the candidate paths $P_{candidates}$ would be included in the result, the related label sequence $\bar{l}$ and the appearance of the joined path is also recorded for the the next JOIN operation (line 12-14). In addition, due to the high proportion of SPARQL queries involving a small number of triples (from 0 to 2) [29], we set $k = 3$ in Algorithm 1 to minimize the execution time and complexity, which means that we only consider the path ρ where $|\bar{l}_\rho| \leq 3$, i.e., up to 3 labels involved in the path, and this will further contribute to improving the scalability of FPIRPQ.

Theorem 1 The time complexity of the greedy FPM algorithm on KG is bounded by $O\big(|E| + \Pi_{i=1}^{k} m_i\big)$, where $|E|$ is the number of edges in KG and m_i is the i-length of label sequences whose occur times are more than *minSup*.

Proof (Sketch) For the FPM algorithm above, the time complexity is composed of two parts: (1) The algorithm traverses the graph by edges and filters infrequent paths, whose complexity is $O(|E|)$. (2) To obtain the frequent paths with length less than or equal to k, we join frequent paths from P^1 to P^k recursively, with complexity $O\big(\Pi_{i=1}^{k} m_i\big)$, hence the overall time complexity of the proposed algorithm is $O\big(|E| + \Pi_{i=1}^{k} m_i\big)$.

Algorithm 1 Frequent path mining on **KG**.

Input: Knowledge graph $G = \{t \mid t = (s, p, o)\}$ and a minimum support threshold $minSup$

Output: frequent paths $P^{\leq k}$ and frequent label sequences $FL^{\leq k}$ with length less than or equal to k.

foreach $t = (s, p, o)$ **do**
  if $p \neq$ `rdf:type` $\wedge\, o \notin L$ **then**
    $count[p] \leftarrow count[p] + 1$; `// count the apperarance of each label`
$P^1 \leftarrow \{(s, p, o) \mid (s, p, o) \in G \wedge count[p] > minSup \wedge o \notin L\}$;
$FL^1 \leftarrow \{p \mid (s, p, o) \in P^1\}$; `// p is regarded as 1-length label string`
for $i = 2, ..., k$ **do** `// recursively join frequent paths from` P^1 `to` P^k
  foreach $\bar{l}_1 \in FL^1$ **do**
    foreach $\bar{l}_2 \in FL^{i-1}$ **do**
      $\bar{l} \leftarrow \bar{l}_1/\bar{l}_2$; `// join two label sequences` $\bar{l}_1$ `and` $\bar{l}_2$
      $P_{candidate} \leftarrow \{(s_1, \bar{l}, o_2) \mid \rho_1 = (s_1, \bar{l}_1, o_1) \in P^1, \rho_2 = (s_2, \bar{l}_2, o_2) \in P^{i-1} \wedge o_1 = s_2\}$;
      if $|P_{candidate}| > minSup$ **then**
        $P^i \leftarrow P^i \cup P_{candidate}$;
        $FL^i \leftarrow FL^i \cup \{\bar{l}\}$;
        $count[\bar{l}] \leftarrow |P_{candidate}|$; `// record the appearance of` $\bar{l}$
return $P^{\leq k}, FL^{\leq k}$;

Example 1 As shown in Figure 2, for a given graph G and $minSup = 4$, Algorithm 1 will first traverse G to count the number of occurrences of each edge in G, which are illustrated in the table on the left. Afterwards, the edges determined by *minSup* will be joined to generate paths with a 2-length label sequence in a greedy way. Furthermore, two frequent paths can be obtained in this process, i.e., paths whose labels are `a/b` and `b/a`, which can be employed to generate frequent paths further. Finally, after all the paths with 3-length label strings have been generated, two paths whose label strings are `a/b/a` and `b/a/b` are recorded as frequent paths. After the FPM procedure is completed, all frequent paths and their label strings counted from the tables in Figure 2 will be recorded in P and FL, respectively.

4.2 Index Scheme

To accelerate the RPQs by employing the frequent paths extracted from data, we propose a path index method, i.e., FPIRPQ. Assuming that the occurrence of patterns shows a consistent distribution in queries and the RDF graph, i.e., the patterns that have a significant noticeable share of the RDF graph will also occur frequently in the queries, then with the most frequent paths indexed, most RPQs will benefit from FPIRPQ.

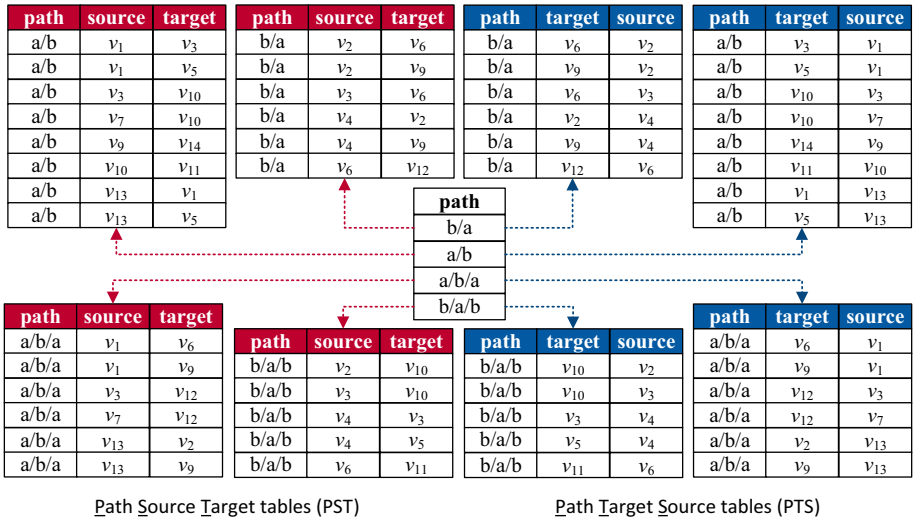


Fig. 3 The path index of FPIRPQ

When constructing frequent path index tables, the form of RPQs should be considered. Generally speaking, common regular path queries are usually given a subject or object of the query, which is employed as the basic part for matching the final result during query processing, so the subject or object of a path should be recorded in the index table. Two relation tables, `PST` and `PTS`, are created to handle queries with known subjects and objects, respectively. Combined with the structure of the index table, B-tree is adopted for the index construction to maximize the efficiency of the query.

Example 2 As illustrated in Figure 3, the frequent paths extracted from the RDF graph are recorded in both `PST` and `PTS`, so that our proposed frequent path index can acclimate to queries with known subjects or objects. The first two columns of `PST` or `PTS` relation tables are indexed using the B-tree, thus, we can quickly find those paths according to the label string and known source or target. The first column of `PST` and `PTS` records the label string $\bar{l}_\rho$ of the frequent path ρ mined from the RDF graphs, while the second column indicates S_ρ (or T_ρ), and the third column is about T_ρ (or S_ρ).

B-tree is typically employed to improve the efficiency of equivalent and range queries on sortable data, so it is particularly suitable for indexing ID columns that are stored in the numeric form. Therefore, a globally unique key will be assigned to each vertex or edge in `PST` and `PTS` when it is first processed during the traversal of KG, which guarantees the global uniqueness of the IDs of the vertices or edges and reduces the space cost. With these IDs, the path index in form of B-tree can be constructed easily and the storage space of FPIRPQ will be compressed further.

5 Query Processing

Based on the analytical research mentioned above, The majority of queries can be solved by the indexed items. But to improve the applicability of FPIRPQ, it is necessary to design the processing method for longer-length queries. In order to reduce the complexity of query processing and increase the query efficiency, we only consider the first few steps in Kleene's closure. Moreover, the regular expression derivatives defined in Definition 5 are used for processing the other types of operators to improve the query efficiency by prefix matching in the process of computing the final result. Furthermore, most paths in data graph should be indexed after index construction and the path label strings extracted from queries can be separated into subparts with lengths ranging from 1 to 3. On the top of frequent path index, we propose a greedy path division algorithm. The algorithm can fully utilize the information in the index to partition and assemble the paths and leverage histogram of the data to obtain the optimal path division results.

Algorithm 2 proposes the path division approach utilized in this paper. For the regular expression extracted from the query, if it contains alternation operator 'l', a prefix-matching procedure is first applied to extract the prefix by the Brzozowski's derivative to reduce intermediate computations and speed up query processing. If the prefix of all substrings is the same string s, the procedure will be called recursively for both the prefix s and the modified substrings (line 2-3). Moreover, for the regular expressions that do not contain alternation operators, the function FindBestPlan will be called and executed to generate the optimal path division plan utilizing the frequent label strings $FL^{\leq 3}$ which generated in Algorithm 1 to obtain the final division (line 6-7).

Theorem 2 The time complexity of the path division algorithm is bounded by $O\left(n^{n-3} + m \cdot n\left(n^{n-2}+1\right) + \log(m)\right)$, where n is the max length of the subparts of r separated by alternation operators and m is the size of the set of path division plans corresponded. According to the query analytical study in the paper [29], most queries are composed of a small number of triples, so as the length of the substring of the path label string, n is smaller in practice, which is acceptable.

Algorithm 2 Path division algorithm `Divide` $(r, FL^{\leq 3})$

Input: regular expression r, frequent label strings $FL^{\leq 3}$
Output: modified regular expression r'

1 **if** $r = r_1|r_2|\ldots|r_\alpha$ **then**
  `// r contains alternation operators '|' and can be divided into α parts that connect with '|'`
2 **if** *the prefix of* $r_1, r_2, \ldots, r_\alpha$ *can be the same string* s **then**
3 $r' \leftarrow$ `Divide` $\left(s, FL^{\leq 3}\right)$ / $\left(\texttt{Divide}\left(\left(s^{-1}r_1,\right)|\ldots|\left(s^{-1}r_\alpha\right)\right), FL^{\leq 3}\right)$;
4 **else**
5 $r' \leftarrow$ `Divide`$(r_1, FL^{\leq 3})$|`Divide`$(r_2, FL^{\leq 3})$|...|`Divide`$(r_\alpha, FL^{\leq 3})$;
6 **else**
7 $r' \leftarrow$ `FindBestPlan` $\left(r, FL^{\leq 3}\right)$; `// find best division plan for` r'
8 **return** r';

Input: label string $\bar{l}$ and frequent label strings $FL^{\leq 3}$ with length less than or equal to 3
Output: modified label string $\bar{l}_m$

1 $\mathcal{PLS}_2 \leftarrow \emptyset$, $PL_2 \leftarrow \emptyset$;
2 $\mathcal{PLS}_3 \leftarrow$ `GeneratePlan`$(\bar{l}, 3, FL^3)$; `// generate division plans by` FL^3
3 **foreach** $PL_3 \in \mathcal{PLS}_3$ **do**
4 $\quad PL_2 \leftarrow PL_3$;
5 $\quad \overline{L}' \leftarrow$ `GetOtherSubString`$(\bar{l}, PL_3)$;
$\quad$ `//` $\overline{L}'$ `represents the set of all substring that can be combined with all the string in` PL_3 `into string` $\bar{l}$
6 $\quad$ **foreach** $\bar{l}' \in \overline{L}'$ **do**
7 $\quad\quad \mathcal{PLS}'_2 \leftarrow$ `GeneratePlan`$(\bar{l}', 2, FL^2)$;
8 $\quad\quad$ **foreach** $PL'_2 \in \mathcal{PLS}'_2$ **do**
$\quad\quad\quad$ `// add 2-length label strings indexed in` FL^2 `into plans`
9 $\quad\quad\quad PL_2 \leftarrow PL_2 \cup PL'_2$;
10 $\quad \mathcal{PLS}_2 \leftarrow \mathcal{PLS}_2 \cup \{PL_2\}$;
11 $\mathcal{PLS} \leftarrow \emptyset$, $PL \leftarrow \emptyset$;
12 **foreach** $PL \in \mathcal{PLS}_2$ **do** `// add not-indexed label strings into plans`
13 $\quad \overline{L}'' \leftarrow$ `GetOtherSubString`$(\bar{l}, PL)$;
14 $\quad PL \leftarrow PL \cup \overline{L}''$;
15 $\quad \mathcal{PLS} \leftarrow \mathcal{PLS} \cup \{PL\}$;
16 $PL \leftarrow$ `minCost`$(\mathcal{PLS})$; `// find division plan with minimum join cost`
17 $\bar{l}_m \leftarrow$ `Combine`(PL); `// add parentheses to each string in` PL `and combine them into a whole string`
18 **return** $\bar{l}_m$;

Function FindBestPlan ($\bar{l}$, $FL^{\leq 3}$)

Proof (Sketch) The time complexity of Algorithm 2 consists of three parts: (1) If there exists alternation operator, the regular expression extracted from query is divided into subparts, where the time complexity is $O(n)$. (2) The algorithm captures the longest common substring of each part which are separated by alternation operators, where complexity is $O(n^2)$. (3) To generate the optimal path division plan for each part, the function `FindBestPlan` is called. If $n \geq 3$, the complexity is $O(n^{n-3} + m \cdot n(n^{n-2} + 1) + \log(m))$. Hence, the overall time complexity of the proposed algorithm is $O(n^{n-3} + m \cdot n(n^{n-2} + 1) + \log(m))$.

Function FindBestPlan presents a greedy procedure to generate the optimal path division plan utilizing the frequent label strings $FL^{\leq 3}$. Since the process of generating division schemes should follow the principle of selecting the longest index item possible, the initial set of division schemes, $\mathcal{PLS}_3$, should be generated using 3-length label strings of frequent paths first (line 2). Afterwards, in order to match the remaining substrings of each division plan in $\mathcal{PLS}_3$, the 2-length label strings of frequent paths will be utilized to generate the

Input: label string $\bar{l}$, length of substring k and frequent label strings FL^k with length equal to k
Output: The set of division plans $\mathcal{PLS}$

1 $\mathcal{PLS} \leftarrow \emptyset$, $PL \leftarrow \emptyset$;
2 **if** $|\bar{l}| = k$ **then**
3 **if** $\bar{l} \in FI^k$ **then** // the label string $\bar{l}$ can be found in FL^k
4 $PL \leftarrow \{\bar{l}\}$;
5 $\mathcal{PLS} \leftarrow \mathcal{PLS} \cup \{PL\}$;
6 **else if** $|\bar{l}| > k$ **then**
7 $\overline{L}^k \leftarrow$ FindAllSubstring$(\bar{l}, k, FL^k)$;
  // retrieve all the k-length substring of $\bar{l}$ in FL^k
8 **foreach** $\bar{l}^k \in \overline{L}^k$ **do**
9 $PL \leftarrow PL \cup \{\bar{l}^k\}$;
10 $\bar{l}_r \leftarrow$ getRightSubstring$(\bar{l}, \bar{l}^k)$;
  // $\bar{l}_r$ represents the label string that can be combined with $\bar{l}^k$ into $\bar{l}$ $(\bar{l} = \bar{l}^k/\bar{l}_r)$
11 $\mathcal{PLS}' \leftarrow$ GeneratePlan$(\bar{l}_r, k, FL^K)$;
  // call function GeneratePlan recursively on $\bar{l}_r$
12 **foreach** $PL' \in \mathcal{PLS}'$ **do**
13 $PL \leftarrow PL \cup PL'$;
14 $\mathcal{PLS} \leftarrow \mathcal{PLS} \cup \{PL\}$;
15 **return** $\mathcal{PLS}$;

Function GeneratePlan ($\bar{l}$, k, FL^k)

plan set $\mathcal{PLS}_2$, where strings included in each scheme are all label strings recorded in FL^k, and are also disjoint substrings of $\bar{l}$ (line 3-10). Compared to the original label string $\bar{l}$, the missing substrings of each division plan PL_2 in $\mathcal{PLS}_2$ are then complemented to generate the complete set of division schemes $\mathcal{PLS}$, where all strings in each scheme in $\mathcal{PLS}$ can be combined into the complete label string $\bar{l}$ (line 12-15). In the final step of the procedure, following the principle of reducing the size of the candidate set as much as possible, the less costly join order among them should be selected, which means the join operation should be performed as early as possible for the paths with small candidate sets, so as to improve the query efficiency (line 16-19).

In the whole procedure of function FindBestPlan, it is essential for obtaining the optimal path division plan to generate the set of division plans based on the given label string. Therefore, The Function GeneratePlan is designed and implemented to generate a set of division plans for a label string $\bar{l}$ with k-length substrings of frequent paths label strings. It is worth noting that the strings in a plan are all disjoint k-length substrings of the original label string $\bar{l}$. For the label string whose length is equal to k, $\bar{l}$ itself is a path division scheme if it can be found in the frequent path index (line 2-5). Moreover, if the length of $\bar{l}$ is greater than k, all k-length substrings corresponding to frequent paths contained in $\bar{l}$ need to be found first, which are also recorded in $\overline{L}^k$ (line 7). Then, for each string in $\overline{L}^k$, the

corresponding $\bar{l}_r$ is found, which represents a label string that can be combined with $\bar{l}^k$ into $\bar{l}$. Then the Function GeneratePlan is called recursively for $\bar{l}_r$, and the division scheme is finally generated (line 7-14).

Example 3 As shown in Figure 4, the original regular expression in the query statement will be transformed into the final division result through three phases. The first phase is the extraction of the common prefix: this part will recursively extract the prefix from this regular expression by the Brzozowski's derivatives to reduce intermediate computations. Next, the partitioning schemes are generated based on the frequent path index: based on the previously proposed greedy path division algorithm, there are two partitioning schemes for the regular expression b/c/d/e/f, and the first scheme, the red dashed rectangle in the top right of the figure, is chosen based on the principle of least join cost of the partitioning scheme. The final path division result can be obtained by assembling the path partitioning results for each subpart.

6 Experiments

To verify the effectiveness and efficiency of FPIRPQ, we implement our method and compare it with the baselines on several datasets in this section.

6.1 Experimental Settings

FPIRPQ was implemented on the top of KGDB [25]. The system was deployed on a server, which has a 16-core Intel Xeon Silver 4216@ 2.10 GHz CPU, with 512GB of RAM and 1920GB SSD, running a 64-bit CentOS 7.7 operating system.

Datasets. We evaluate our method over both benchmark and real-world datasets. Composed of repeatable synthetic data, LUBM [30] allows users to define the size of the dataset. To study the

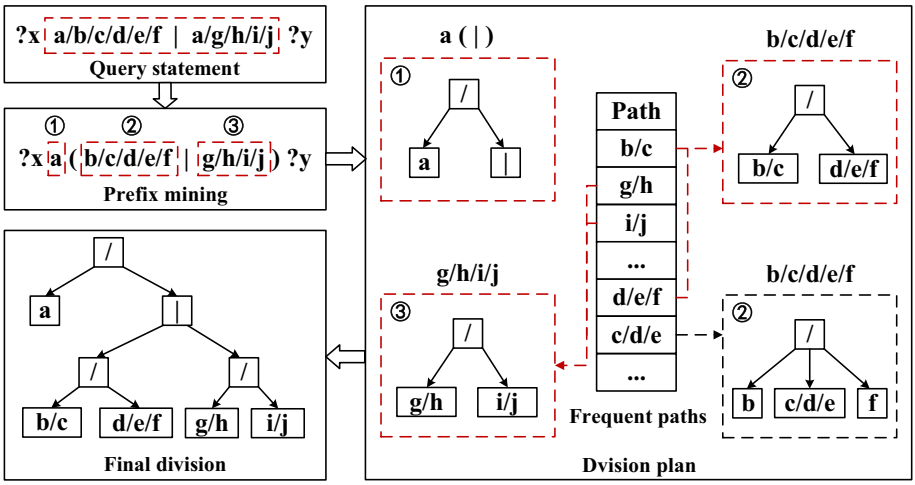


Fig. 4 Path division procedure

scalability of FPIRPQ, we adopted six LUBM datasets of different sizes in the experiments. DBpedia [31] is a real-world dataset extracted from Wikipedia, which maps information into different ontology classes with corresponding properties based on Wikipedia infoboxes. To verify the effectiveness of our approach over real data, experiments are also conducted on a subset of the DBpedia (containing entity classes such as person, organization and place). The statistics of the datasets are shown in Table 2.

Baselines. In order to verify the effectiveness and efficiency of our method, we compared FPIRPQ against two KG databases: Virtuoso [23] and KGDB [25]. As a hybrid database management system, Virtuoso supports various data models. Moreover, as a KG database, KGDB implements a unified storage scheme for accommodating RDF graphs and property graphs in KGs.

Benchmark Queries. To measure the performance of the proposed approach of query processing, we create 12 benchmark queries[1] over synthetic datasets (LUBM) and real-world datasets (DBpedia) as there is no explicit benchmark targeted for RPQs on RDF graphs yet. As illustrated in Table 3, considering the result of FPM mentioned in Section 4.1 and the feature of RPQ, the queries proposed are mainly composed of chain queries, while also including star queries and complex queries. Moreover, Based on the rules in SPARQL 1.1 [9], the precedence of all operators in the regular expression is maintained.

6.2 Experimental Results

Exp 1. Index construction. As shown in Figure 5, the FPM time and the storage space occupied by the indexes constructed by FPIRPQ both increase in a linear trend with the size of the dataset increasing. Moreover, since the index is constructed based on the results of FPM, the storage space required for the index constructed by FPIRPQ is also closely related to the

Table 2 Dataset overview

Dataset	#Triples	#Vertices	#Edges
LUBM10	1,316,700	207,429	630,757
LUBM20	2,782,126	437,558	1,332,030
LUBM30	4,109,002	645,957	1,967,309
LUBM40	5,495,742	864,225	2,630,657
LUBM50	6,890,640	1,082,821	3,298,814
LUBM100	13,824,536	2,179,780	8,952,366
DBpedia	23,445,441	2,257,499	6,876,041

[1] https://github.com/haowq0417/FPIRPQ

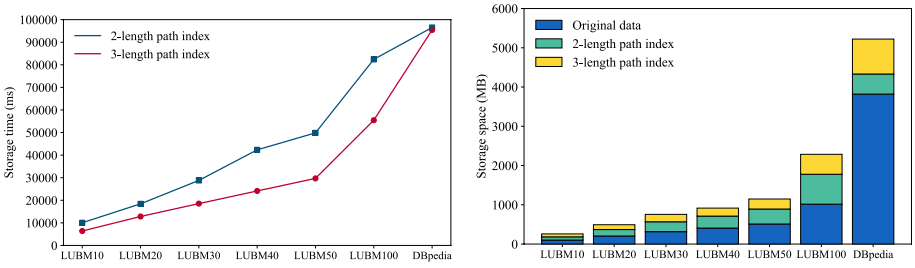
Table 3. Benchmark queries

Query	Type	Feature	Diagram*
$Q1$	chain	2-length index	
$Q2$	star	type-limited, 2-length index	
$Q3$	chain	without index	
$Q4$	chain	3-length index	
$Q5$	star	type-limited, 3-length index	
$Q6$	chain	2-length index, not-indexed item	
$Q7$	chain	2-length index, not-indexed item, literal variable	
$Q8$	chain	without index, closure operator	
$Q9$	complex	3-length index, alternation operator	
$Q10$	complex	3-length index, not-indexed item, alternation operator	
$Q11$	chain	2-length index, not-indexed item, closure operator	
$Q12$	chain	not-indexed item, 2-length index, closure operator	

* The solid circles represent the edge labels, the dotted circles represent the entity variables (dotted circle with *t* means the entity is type-limited), the dotted squares represent the literal variables, the rings represent the closures, and the lines with options represent the alternation operators

time of FPM. Furthermore, it can be concluded from Figure 5(a) that on the LUBM dataset, with the growth of the data scale, the trend of increasing time and space required to build a 3-length path index gradually decreases compared to the 2-length path index, which proves that focusing on shorter paths can reduce the extraction time of frequent path indexes effectively. In addition, for DBpedia, which is the real-world dataset, the time required to build a 2-length or 3-length path index is basically the same, which proves the effectiveness of the greedy FPM algorithm presented in Section 4.1.

As illustrated in Figure 5(b), for the selected datasets of LUBM, the final size of all the indexes required is essentially the same as the space occupied by the original data. Compared to the synthetic datasets, there is a greater variety of



(a) Frequent path mining time (b) Storage space

Fig. 5 The experimental results of time and storage space

edges in the real-world dataset and the presence of a large number of low frequency edges leads to a lower proportion of high frequency edges, which is the reason why the size of the index is significantly smaller than the original graph data. Although FPIRPQ does not predominate in terms of storage space, the space cost of the indexes is worth compared to the improved query efficiency.

Exp 2. Query efficiency. As shown in Figures 6 and 7, FPIRPQ can increase the query efficiency by two orders of magnitude for most benchmark queries on LUBM. Furthermore, compared to KGDB, where no constructed path indexes, FPIRPQ improves the query efficiency by one order of magnitude on average for most benchmark queries on LUBM, which illustrates the effectiveness of FPIRPQ further and explains the reason why FPIRPQ deals with $Q3$ and $Q8$ in the same execution time as KGDB. For DBpedia, FPIRPQ is at least twice as

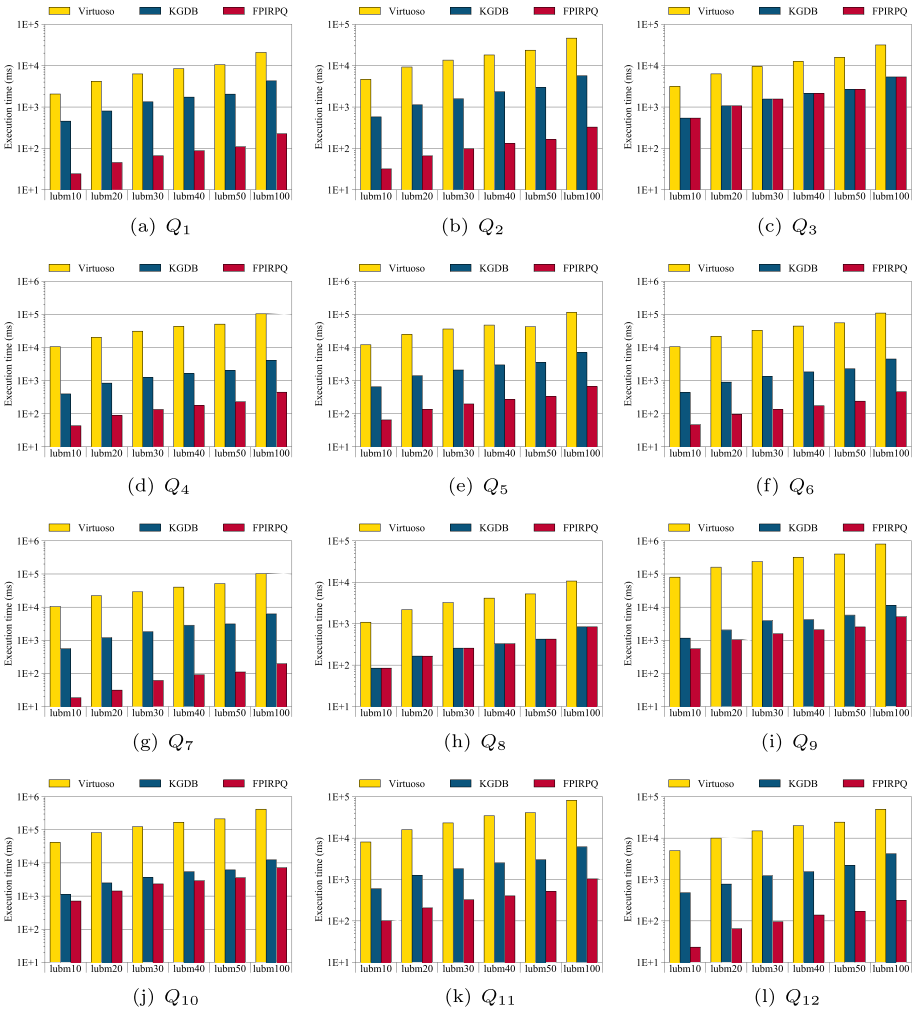


Fig. 6 Query execution time on LUBM (logarithmic scale)

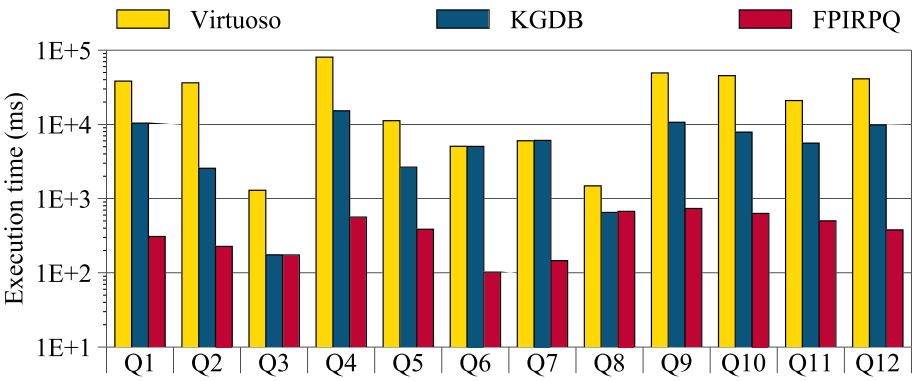


Fig. 7 Query execution time on DBpedia (logarithmic scale)

efficient at querying as KGDB and more than three times as efficient as Virtuoso.

The difference in query performance between LUBM and DBpedia stems from the fact that on DBpedia, the generated frequent paths account for a greater proportion of the results of high-frequency edge join operations, and there is a smaller variety of frequent paths compared to LUBM, which reduces the effect of FPIRPQ on query performance improvement.

The advantages of FPIRPQ can be fully demonstrated for longer regular path queries involving the `JOIN` operations of multiple tables and queries containing closure operators. For queries containing the operator of alternation, however, the need to perform the `UNION` operation of two intermediate resultant relational tables will influence the query efficiency of FPIRPQ, which is the reason why the advantage of FPIRPQ over KGDB is reduced for the queries of $Q9$ and $Q10$. Even so, we can observe from the overall experimental results that due to the presence of frequent path indexes, the reduction in query time increases with the growth of the data scale, which also proves the effectiveness of Algorithm 2.

In Virtuso, to accommodate the data of KG, three-column tables are built since all the data need to be stored in the form of triples. Furthermore, based on the tables, five indexes including `PSOG`, `POSG`, `SP`, `OP`, and `GS`(`S`, `P`, `O`, and `G` stand for subject, property, object, and graph, respectively.), are constructed in Virtuoso to accelerate query processing. Therefore, it is obvious that Virtuoso requires more time to handle the queries involving multiple edges, since more time-consuming `JOIN` operations of relation tables with more rows are required. Compared with Virtuoso, the significantly improved query efficiency of KGDB is closely related to the type-based storage scheme adopted, where all the data is classified and stored according to the types of vertices or edges. Moreover, FPIRPQ adopts B-tree for constructing the frequent path index on the top of KGDB, which further improved the query efficiency.

7 Conclusion

In this paper, FPIRPQ, a novel approach based on the frequent path index is proposed to accelerate RPQs on knowledge graphs. To make full use of the statistics of underlying KGs, we propose a greedy FPM algorithm for extracting frequent paths. Moreover, in order to improve the efficiency of query processing, the *Brzozowski's derivatives* is utilized to divide the label strings of paths into parts and a greedy algorithm employing the path index effectively for further division is proposed. Furthermore, we propose micro-benchmarks including 12 basic queries over synthetic and real-world datasets to measure the performance of FPIRPQ. The experimental results show that FPIRPQ improves the query efficiency by up to orders of magnitude compared to the state-of-the-art methods.

Acknowledgments This work is expanded on the PAIRPQ: An Efficient Path Index for Regular Path Queries on Knowledge Graphs [12], and is supported by National Key Research and Development Program of China (2019YFE0198600); the National Natural Science Foundation of China (61972275).

Author contributions Xin Wang and Wenqi Hao are the major contributors in writing the manuscript and preparing the pictures. Yuzhou Qin and Baozhu Liu participate in the experiments and analyze the results. All authors read and approve the final manuscript.

Funding This work is supported by National Key Research and Development Program of China (2019YFE0198600); the National Natural Science Foundation of China (61972275).

Data availability The queries ($Q1 \sim Q12$) designed on LUBM and DBpedia are are available in GitHub (https://github.com/haowq0417/FPIRPQ), and all the other data generated or analyzed during this study are included in this published article.

Declarations

Human and animal ethics Not applicable

Ethics approval and consent to participate Not applicable

Consent for publication Not applicable

Competing interests The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

1. Ernst, P., Meng, C., Siu, A., Weikum, G.: Knowlife: A knowledge graph for health and life sciences. IEEE Computer Society (2014)
2. Shi, L., Li, S., Yang, X., Qi, J., Pan, G., Zhou, B.: Semantic health knowledge graph: semantic integration of heterogeneous medical knowledge and services. BioMed research international 2017 (2017)
3. Rotmensch, M., Halpern, Y., Tlimat, A., Horng, S., Sontag, D.: Learning a health knowledge graph from electronic medical records. Scientific Reports **7**(1), 1–11 (2017)
4. Liu, J., Lu, Z., Du, W.: Combining enterprise knowledge graph and news sentiment analysis for stock price prediction. In: Proceedings of the 52nd Hawaii International Conference on System Sciences (2019)
5. Ulicny, B.: Constructing knowledge graphs with trust. In: 4Th International Workshop on Methods for Establishing Trust of (Open) Data, Bentlehem, USA (2015)
6. Chen, P., Lu, Y., Zheng, V.W., Chen, X., Yang, B.: Knowedu: a system to construct knowledge graph for education. Ieee Access **6**, 31553–31563 (2018)

7. Grévisse, C., Manrique, R., Mariño, O., Rothkugel, S.: Knowledge graph-based teacher support for learning material authoring. In: Colombian Conference on Computing, pp 177–191. Springer (2018)
8. Consortium, W.W.W., et al.: Rdf 1.1 concepts and abstract syntax (2014)
9. Consortium, W.W.W., et al.: Sparql 1.1 query language (2013)
10. Kostylev, E.V., Reutter, J.L., Romero, M., Vrgoč, D.: Sparql with property paths. In: International Semantic Web Conference, pp 3–18. Springer (2015)
11. Wang, X., Wang, S., Xin, Y., Yang, Y., Li, J., Wang, X.: Distributed pregel-based provenance-aware regular path query processing on rdf knowledge graphs. World Wide Web, 1–32 (2019)
12. Liu, B., Wang, X., Liu, P., Li, S., Wang, X.: Pairpq: An efficient path index for regular path queries on knowledge graphs. In: Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint International Conference on Web and Big Data, pp 106–120. Springer (2021)
13. Yan, X., Han, J.: gspan: Graph-based substructure pattern mining. In: 2002 IEEE International Conference on Data Mining, 2002. Proceedings, pp 721–724. IEEE (2002)
14. Holder, L.B., Cook, D.J., Djoko, S., et al.: Substucture discovery in the subdue system. In: KDD Workshop, pp. 169–180, Washington, DC, USA (1994)
15. Ghazizadeh, S., Chawathe, S.S.: Seus: Structure extraction using summaries. In: International Conference on Discovery Science, pp 71–85. Springer (2002)
16. Goldman, R., Widom, J.: Dataguides: Enabling Query Formulation and Optimization in Semistructured Databases. Technical report, Stanford (1997)
17. Goldman, R.: Approximate dataguides. workshop on query processing for semistructured data and non-standard data formats. http://www-db.stanford.edu/pub/papers/adg.ps (1999)
18. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to automata theory, languages, and computation. Acm Sigact News **32**(1), 60–65 (2001)
19. Milo, T., Suciu, D.: Index structures for path expressions. In: International Conference on Database Theory, pp 277–295. Springer (1999)
20. Kaushik, R., Shenoy, P., Bohannon, P., Gudes, E.: Exploiting local similarity for indexing paths in graph-structured data. In: Proceedings 18th International Conference on Data Engineering, pp 129–140. IEEE (2002)
21. Chen, Q., Lim, A., Ong, K.W.: D (k)-index: An adaptive structural summary for graph-structured data. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, pp 134–144 (2003)
22. He, H., Yang, J.: Multiresolution indexing of xml for frequent queries. In: Proceedings. 20th International Conference on Data Engineering, pp 683–694. IEEE (2004)
23. Erling, O., Mikhailov, I.: Rdf support in the virtuoso dbms. In: Networked Knowledge-Networked Media, pp 7–24. Springer (2009)
24. Das, S., Agrawal, D., El Abbadi, A.: G-store: A scalable data store for transactional multi key access in the cloud. In: Proceedings of the 1st ACM Symposium on Cloud Computing, pp 163–174 (2010)
25. Liu, B., Wang, X., Liu, P., Li, S., Zhang, X., Yang, Y.: Knowledge graph database system with unified model and query languages. Ruan Jian Xue Bao/Journal of Software (in Chinese) **32**(3), 781–804 (2021)
26. Brzozowski, J.A.: Derivatives of regular expressions. Journal of the ACM (JACM) **11**(4), 481–494 (1964)
27. Zhu, F., Qu, Q., Lo, D., Yan, X., Han, J., Yu, P.S.: Mining top-k large structural patterns in a massive network. Proceedings of the VLDB Endowment **4**(11), 807–818 (2011)
28. Vanetik, N., Gudes, E., Shimony, S.E.: Computing frequent graph patterns from semistructured data. In: 2002 IEEE International Conference on Data Mining, 2002. Proceedings., pp. 458–465. IEEE (2002)
29. Bonifati, A., Martens, W., Timm, T.: An analytical study of large sparql query logs. VLDB J. **29**(2), 655–679 (2020)
30. Guo, Y., Pan, Z., Heflin, J.: Lubm: a benchmark for owl knowledge base systems. Journal of Web Semantics **3**(2-3), 158–182 (2005)
31. Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P.N., Hellmann, S., Morsey, M., Van Kleef, P., Auer, S., et al.: Dbpedia–a large-scale, multilingual knowledge base extracted from wikipedia. Semantic Web **6**(2), 167–195 (2015)

Authors and Affiliations

Xin Wang[1] · Wenqi Hao[1] · Yuzhou Qin[1] · Baozhu Liu[1] · Pengkai Liu[1] · Yanyan Song[1] · Qingpeng Zhang[2] · Xiaofei Wang[1]

Xin Wang
wangx@tju.edu.cn

Wenqi Hao
haowenqi@tju.edu.cn

Yuzhou Qin
yuzhou_qin@tju.edu.cn

Baozhu Liu
liubaozhu@tju.edu.cn

Pengkai Liu
liupengkai@tju.edu.cn

Yanyan Song
songyanyan1895@tju.edu.cn

Qingpeng Zhang
qingpeng.zhang@cityu.edu.hk

[1] College of Intelligence and Computing, Tianjin University, Tianjin, China

[2] School of Data Science, City University of Hong Kong, Hong Kong, China