On the Evaluation of Pattern Match Queries in Large Graph Databases

by

Bin Guo

A thesis submitted to the
Department of Applied Computer Science
in conformity with the requirements for
the degree of Master of Science

University of Winnipeg
Winnipeg, Manitoba, Canada
January 2018

Abstract

Recently, graph databases have been received much attention in the research community due to their extensive applications in practice, such as social networks, biological networks and World Wide Web, which bring forth a lot of challenging data management problems including subgraph search, shortest-path query, reachability verification, pattern matching, and so on. Among them, the graph pattern matching is to find all matches in a data graph G for a given pattern graph Q and is more general and flexible than other problems mentioned above. In this thesis, we address a kind of graph matching, the so-called *pattern matching with* δ, by which an edge in Q is allowed to match a path of length $\leq \delta$ in G. In order to reduce the search space when exploring G to find matches, we propose a novel pruning algorithm to eliminate all unqualified vertices. We also propose a strategy to speed up the distance-based join over two lists of vertices. Extensive experiments have been conducted, which show that our approach makes great improvements in running time compared to existing ones.

Acknowledgments

First and foremost, I am heartily thankful to my supervisor, Dr. Yangjun Chen, for his patience and knowledge and support from the initial to the final level enabling me to develop an understanding of this thesis work. Thanks for giving me the opportunity to work with him, leading me going through each phase of this research, encouraging me when I felt frustrated.

My deepest gratitude goes to my wife, Alice Chen, for his emotional support and encouragement. I dedicate this thesis to her. Without their support, I do not think I can finish this research.

Last but not least, I offer my regards and blessing to all of my teachers and friends who help me during this thesis work.

Content

CHAPTER 1 INTRODUCTION

1.1 What is Pattern Matching?

Nowadays, in numerous applications, including social networks, biological networks, and WWW networks, as well as geographical networks, data are normally organized into directed graphs with vertices for objects and edges for their relationships. The burgeoning size and heterogeneity of networks have inspired extensive interests in querying a graph in different ways, such as *subgraph search* [1][2][3][4][5][6][7][8][9][10][11][12][13][14] [15], *shortest-path queries* [16][17][18], *reachability queries* [16][19][20] [21], and *pattern matching queries* [22][23][24][25][26]. Among them, the pattern matching is most difficult, by which we are asked to look for all matches of a certain pattern graph Q in a data graph G, each of which is isomorphic to Q or satisfies certain conditions related to Q. As a key ingredient of many advanced applications in large networks, the graph matching is conducted in many different domains: (1) in the traditional relational database research, a schema is often represented as a graph. By matching of data instances we are required to map a schema graph to part of a data graph to check any updating of data for consistency [27]; (2) in a large metabolic network, it is desirable to find all protein substructures that contain an α-β-barrel motif, specified as a cycle of β strands embraced by an α-helix cycle [23]; (3) in the computer vision, a scene is naturally represented as a graph G, where a feature is a vertex in G and an edge stands for a geographical adjacency of two features[28]. Then, a scene recognition is just a matching of a graph standing for a scene to another stored in databases.

The first two applications mentioned above are typically exact matching, called subgraph matching, by which the graph isomorphism checking or subgraph isomorphism is required. In other words, the mapping between two graphs must be both vertex-label preserving and edge preserving in the sense that if two vertices in the first graph are linked by an edge, they are mapped to two vertices in the second by an edge as well. It is well-known that the subgraph isomorphism checking is *NP*-complete.

The third application is a kind of inexact matching, called pattern matching, which is our mainly focused part in this work. First, two matching features from two graphs may disagree in some way due to different observation of a same object. Secondly, between two adjacent features in a graph may there be some more features in another graph [28] figured out by a different observer. This leads to a new kind of queries, called *pattern matching with* δ, by which an edge in a query graph is allowed to match a path in a data graph. More specifically, two adjacent vertices v and v' in a query graph Q can match two vertices u and u' in a data graph G with $label(v) = label(u)$ and $label(v') = label(u')$ if the distance between u and u' is less than δ. Here, the distance of two vertices is defined to be the length of the shortest path connecting these two vertices.

In addition, assuming that the data graphs are unweighted graphs, the above parameter δ means the fewest steps or hops between two specific vertices u and u'. It is easy to know that when $\delta = 1$, this pattern matching problem is reduced to traditional subgraph exact matching problem.

In a word, pattern matching queries are more flexible, informative and challenging compared to traditional subgraph queries. In this work, we will mainly focus on pattern matching queries, for which we propose a novel and efficient new method.

1.2 Current Research

A lot of work has been done on subgraph matching, but most of them are for special kind of graphs, such as [29] for planar graphs and [30] for valence graphs, or by establishing indexes [23][5][6][7][8][9][13][14][15], or indexing frequent queried subgraphs [31][32]. The problem is that all the mentioned above are only suitable for subgraph matching, none of which can be applied to pattern matching queries, since all the pruning techniques are based on the necessary condition of subgraph isomorphism.

However, there is little work related to pattern matching. In [26], Tong et al. discussed the first pattern matching algorithm, called *G-Ray*. It can do pattern matching only when $\delta \leq 2$, but it is useless when $\delta > 2$. In [25], Cheng et al. proposed a total different algorithm *R-join* for pattern matching queries but with reachability limitation between two vertices in G rather than δ defined above. In *R-join*, an index structure is introduced, called *2-hop labeling* discussed in [16][17], which is used to facilitate the calculation of reachability between each pair of vertices in G. It is easy to know that such index technique can be extended to compute the shortest-path distance between each pair of vertices in G. Given two lists , R_i and R_j, where $R_i(R_j)$ G with the same label as $v_i(v_j)$ ((v_i, v_j) is a query edge in Q.), the join operation between R_i and R_j, namely constructing the matched relations by δ defined above, is costly with time complexity $O(\sqrt{|E(G)|}|R_i||R_j|)$. This algorithm has been greatly improved by the *MD-join* of Zou et al. [24]. They map all vertices in G into the points in a vector space of *k*-dimensions by using the so-called *LLR-embedding* technique discussed in [12][33], where k is selected to be $O(\log^2 |V(G)|)$ to save space. Based on this index technique, most of shortest-path distance calculation between each pair of vertices in G can be filtered. Then, only small number of tuples need to be checked by using *2-hop labelin*g [16][17]. However, the optimal index size of 2-hop labeling is $O(|V(G)|\sqrt{|E(G)|})$ and index time complexity is $O(|V(G)|^4)$ in the worst case, which is forbiddingly high for large data graphs.

Both in *R-join* [25] and *MD-join* [24], it is one important issue that the matching relations R_{ij} are not the final matching results after join operation between each pair of lists, R_i and R_j (corresponding to a query edge (v_i, v_j) in Q). The classical natural join is a necessary operation for generating all matches. The problem is that their natural join is NP-complete with time complexity $O(\prod_{ij} |R_{ij}|)$, which the running time increases dramatically with the query edge number of Q and also the size of each R_{ij}.

1.3 Our Method

In this work, we propose a new framework for pattern matching queries by improving *R-join* [25] and *MD-join* [24], which mainly contains two main parts:

- *Relation Construction*. Given a query Q with n vertices, for each vertex v_i in Q, we first find a list R_i that include all those vertex u_i in data graph G with $label(u_i) = label(v_i)$. Then, for each edge (v_i, v_j) in Q, we need to find all matching pairs (u, u') where $u \in R_i, u' \in R_j$ in G whose shortest-path distances are $\leq \delta$. These matching tuples can be stored in a relation R_{ij}. In other words, a relation R_{ij} include all vertex pairs (u, u') whose shortest-path distances are bounded by δ.

- *Matching Construction*. Here, R_{ij} is the final matching results if Q only a single query edge (v_i, v_j). However, normally query Q is a graph and includes more than one query edge. Thus, finally, we need to perform a classical natural join [34] to extract all matching results from all relations R_{ij}, each of which corresponds to a query edge in Q.

By *Relation Construction*, we have to perform a shortest-path distance computation between two lists, R_i and R_j, online. One naive solution to reduce the cost is to pre-compute and store all pair-wise shortest-path distance. This method is fast but prohibitively high in space usage ($O(|V(G)|^2)$), where $|V(G)|$ is the number of vertices in G. The *D-join* algorithm in [24], as a better solution, is still not efficient enough because of the huge searching space.

In order to speed up the process of *Relation Construction*, we propose a notion of Δ-*Transitive Closure* for data graph G denoted as G^{Δ}. The idea is simple. Since pre-computing and storing all pair-wise shortest-path distance has space usage $O(|V(G)|^2)$, we can only pre-compute part of pair-wise shortest-path distances within Δ, where Δ is maximum value of possible parameter δ. Normally, δ tend to be small for the pattern matching problems. By doing this, the space usage is reduced to $O(|V(G)| \cdot d^{\Delta})$, where d is the average degree of data graph G. It is true that in the worst case the space usage is still $O(|V(G)|^2)$. However, given the average degree d and Δ tend to be small in practice and the index size of G^{Δ} is in general acceptable. For example, a typical directed data graph Citeseer, as shown in our experiment section, has 0.38 million vertices and 1.7 million edges with $d = 4.6$ (calculated roughly by $|E(G)|/V(G)$ for directed graphs). Normally, we prefer select δ ranging from 1 to 4, that is, $\Delta = 4$. In this example, the index space requirement estimate for G^{Δ} would be 1.9GB, which is more competitive than the 8GB index size of 2-hop labeling algorithm. Note that, the Δ could not be too large, for the pattern matching query is to find subgraphs that is similar to the pattern and adopting big Δ is meaningless to measure such similarity. Furthermore, when receiving a query Q, the time complexity of online relation construction is much faster than other compared methods like D-join by an order of magnitude.

By *Matching Construction*, it is true that the classical natural join over all relations R_{ij} is inescapable for exacting all the matching results, but we can do better. That is, in all relations R_{ij}, we can previously remove the redundant tuples that are not necessary to participate the final natural joins. Based on this observation, in order to prune all relations R_{ij} we propose a two-level filtering strategy:

1) (*Domain Filtering*) Let $e = (v_i, v_j)$ be an edge in Q. Let u, u' be two vertices in G where $u \in R_i$ and $u' \in R_j$ (each list R_i called $Domain$ as well). If the shortest-path distance between u and u' is $\leq \delta$, then we say, u and u' supports each other. Assume that $e' = (v_j, v_k)$ is another edge in Q, but joining e at v_j. If u' does not have any support from R_k, then u' definitely does not belong to any answer and can be removed from R_j. More importantly, u gets one less support from R_j now. If the number of supports of u from R_j becomes 0, u should also be removed

from R_i, which may lead to the elimination of some more vertices from some other domains. This propagation process repeats until each R_i cannot be changed anymore. Lists R_i and relations R_{ij} after *Domain Filtering* are denoted as R'_i and R'_{ij}, respectively.

2) (*Relation Filtering*) Based on *Domain Filtering*, let $\langle u, u' \rangle$ be a tuple in R'_{ij}. Assume that (v_i, v_k) and (v_j, v_k) are two edges in Q, incident to (v_i, v_j) respectively at v_i and v_j. Then, $\langle u, u' \rangle$ should have at least one support from R_k, i.e., there exists at least one vertex u'' in R_k such that $\langle u, u'' \rangle \in R_{ik}$ and $\langle u', u'' \rangle \in R_{jk}$. Otherwise, $\langle u, u' \rangle$ should be removed from R_{ij}, which may lead to the elimination of u and u' if their supports from a certain domain are all removed in this way. Again, such elimination of vertices and tuples can also be propagated as described above. Each list R_i and relation R'_{ij} after *Relation Filtering* are denoted asR''_i and R''_{ij}, respectively.

Our theoretical analysis shows that, in the worst case, the cost of the *Domain Filtering* is bounded by $O(|E(Q)|D^2)$ and the cost of *Relation Filtering* is bounded by by $O(|V(Q)|^3 D'^3)$, where D is the maximum size of each list R_i, and D' is the maximum size of each list R'_i. Nevertheless, the average speed of *Domain Filtering* and *Relation Filtering* are really fast. After above filtering algorithms, in most of case, the natural join space would be reduced efficiently, and thus its running time could be greatly decreased. Again take, for example, roadNetPA, which is a typical data graph. For a typical graph pattern query Q with 5 edges and $\delta = 4$, originally the total number of tuples is more than 0.125 million and the classical natural join would need 23.2 second to pickup all the matching results. But after the Domain Filtering and Relation

Filtering, the total number of tuples decrease to 5400 and the natural join only need about 279 milliseconds (see section 8.5), which speeds up to 83 times. It is amazing that the Domain Filtering and Relation Filtering, however, spend only 115.6 and 27.9 milliseconds, respectively.

1.4 Contributions

The main contributions of this thesis can be summarised as follows:

1) For handling pattern matching queries over a large data graph G, we propose a general framework which includes two parts, *Relation Construction* and *Matching Construction*.
2) In order to enable shortest-path distance computation efficiently in *Relation Construction*, we propose an index method based on the notion of Δ -Transitive Closure, G^{Δ} , considering the necessary condition of pattern matching queries on real data graphs. Such approach is simple and fast to get all the matching pairs between two list, R_i and R_j, and with reasonable indexing time and size.
3) In order to speed up the process of *Matching Construction*, we propose a two-level filtering strategy, *Domain Filtering* and *Relation Filtering*, which will greatly reduce the relations participating the Natural Joins. Since the classical Natural Joins used by *Matching Construction* are well-know NP-complete, this pruning strategy is meaningful to facilitate its processing.
4) Finally, extensive experiments are conducted with real synthetic data graphs in order to evaluate our proposed algorithms.

1.5 Structure of the Thesis

The reminder of the thesis is organized as follows: In Chapter 2, we discuss the related work. In Chapter 3, we give the formal definition for pattern matching queries. Chapter 4 is devoted to describe the general framework of our method. In Chapter 5 - 7, the details in the framework will be fully discussed. A variety of experiments are reported in Chapter 8. Finally, a short conclusion and future work is set forth in last Chapter 9.

CHAPTER 2 RELATED WORK

The graph matching problem has always been one of the main focuses in graph database. A great number of methods have been proposed. Roughly speaking, all of them can be divided into two categories: subgraph matching (exact matching) and pattern matching (inexact matching). By the former, for a data graph G and a query Q, all the subgraphs of G, which are isomorphic to Q, are obtained. By the latter, a parameter δ is introduced as the vertex pairs' shortest-path distance tolerance. Compared with the subgraph matching, the pattern matching is more flexible and informative.

2.1 Subgraph Matching

For subgraph matching, some early algorithms, such as [35][36], are proposed without using index, which have super-linear time complexity and only can work for "toy" graphs with about 1K vertices. Later, many approaches, such as BitMat [13] and RDF-3X [14], are proposed by creating index on distinct edges. The problem with these approaches are the excessive use of costly join operations. To avoid excessive joins, other approaches, like recent work SpiderMine [31] and [32], are proposed by creating index on frequent subgraphs or frequently queried subgraphs. The problem with these approaches is that finding and saving frequent subgraphs is very costly in both time and space, and queries that do not contain frequent subgraphs are not well supported. Another index method is proposed in GraphQL [15], which indexes the subgraph within distance radius r for each vertices. In the same spirit, [9] encode the labels of vertices within distance radius r into a signature, and then index the signature. This approaches are not efficient, as it requires high indexing time and space complexity.

Unfortunately, all the methods mentioned above are limited to exact subgraph matching and not applicable to our mainly focused pattern matching problem, since all the pruning techniques are based on the necessary condition of subgraph isomorphism.

2.2 Pattern Matching

2.2.1 G-Ray

There is little related work existing for inexact pattern matching query. The first algorithm was discussed in [26], in which Tong et al. proposed a method called *G-Ray* (or *Graph X-ray*) to find subgraphs that match a query pattern Q either exactly or inexactly. If the matching is inexact, an edge (v_i, v_j) in Q is allowed to match a shortest-path of length 2. That is, v_i and v_j can match respectively two vertices u and u', which are separated by an intermediate vertex. This algorithm is based on a basic graph searching, but with two heuristics being used:

- *Seed selection*. Each time to search a data graph G, a set of starting points will be determined. Normally, they are some vertices having the same label as a vertex v in Q, which has the largest degree.
- A *goodness* score function $g(u)$ ($u \in G$) is used to guide the searching of G such that only the subgraphs with good measurements will be explored.

Although the *G-Ray* can efficiently find the *best-effort* subgraphs that qualify for Q, it is not as general and flexible as the graph pattern matching with δ defined in the previous section.

2.2.2 2-hop labeling

In *R-join* [25] and *D-join* [24], it is necessary to first compute the reachability and shortest-path distances between each pair of vertices u

and u' in data graph G. One straightforward solution to reduce the cost is to pre-compute and store all pair-wise shortest-path distance. This method is fast but requires much space ($O(|V(G)|^2)$), where $|V(G)|$ is the number of vertices in G. The other straightforward method is to use classical Dijkstra's Algorithm [37], which has time complexity $O(|V(G)|^2 log|V(G)|)$ to get all pairs-wise shortest-path distances. Although this method has no extra space cost, it is not efficient especially for large data graph G.

In order to optimize the computation of shortest-path distance between each pair of vertices u and u' in G, the *2-hop labeling* algorithm is used in [24]. Specifically, each vertex u in G will be assigned a label $L(u) = (L_{in}(u), L_{out}(u))$, where $L_{in}(u), L_{out}(u) \subseteq V(G)$. Vertices in $L_{in}(u)$ and $L_{out}(u)$ are called *centers*. There are two kinds of 2-hop labeling: reachability labeling and distance labeling. By the former, given two vertices $u, u' \in G$, there is a path from u to u' (denoted as $u \rightsquigarrow u'$) if and only if $L_{out}(u) \cap L_{in}(u') \neq \Phi$. By the latter, For distance labeling, the shortest-path distance between two vertices $u, u' \in G$ (denoted as $Dist_{sp}(u, u')$) is computed by using the following equation:

$$Dist_{sp}(u,u') = min\{Dist_{sp}(u,w) + Dist_{sp}(w,u')|w \in (L_{out}(u) \cap L_{in}(u'))\} \tag{1}$$

The shortest-path distances between vertices and centers such as $Dist_{sp}(u, w)$ and $Dist_{sp}(w, u')$ are pre-computed and stored. The size of 2-hop labeling can be defined as $\sum_{u \in V(G)}(|L_{in}(u)| + |L_{out}(u)|)$, while the size of 2-hop distance labeling is $O(|V(G)|\sqrt{|E(G)|})$. Thus, according to Equation 1, it need only $O(\sqrt{E(G)})$ time to compute the shortest-path distance by distance labeling, since the average size of distance labeling for each vertex is $O(\sqrt{E(G)})$.

Note that finding the minimum size of a 2-hop for each u in G is proved to be *NP*-hard [16]. Therefore, in practice, only a nearly optimal solution is used [25][24]. In [16], Cohen et al. proposed a heuristic algorithm based on the minimal set-cover problem. Initially, all pairwise shortest-path distance in G are computed, denoted by D_G. Then, in each iteration, one vertex w in $V(G)$ is selected as a 2-hop center to maximize the following equation:

$$\frac{D_{(G,w)} \cap D_G}{|A_w| + |D_w|} \tag{2}$$

where $D_{(G,w)}$ is the shortest-paths which are covered by w, A_w contains all vertices that can reach w and D_w contains all vertices that are reachable from w. Then all paths in $D_{(G,w)}$ are removed from D_G. This process is iterated until $D_G = \varPhi$, and all selected 2-hop centers are returned. However, pre-computing all-pairs shortest-paths is prohibitively high in space usage ($O(|V(G)|^2)$). This heuristic algorithm itself also requires high running time, which means it is impossible to get all 2-hop labeling in reasonable time.

In [17], Cheng and Yu improve the above method only for directed data graphs. Specifically, a large directed graph data G is first converted into a directed acyclic graph (DAG) $G^{\downarrow}$ by removing some vertices in each strong connected components (SCC) of G. These removed vertices are selected as 2-hop centers. Obviously, all shortest-paths that pass through these removed vertices are covered by these selected 2-hop centers. Then $G^{\downarrow}$ is partitioned into two subgraphs, $G^{\perp}$ and $G_{\top}$, by a set of vertices as separators, which are also selected as 2-hop centers and must cover all shortest-paths across $G^{\perp}$ and $G_{\top}$. This process will continue until these subgraphs are small enough to compute the 2-hop labeling directly by method in [16]. However, there are many redundant vertices in the above 2-hop labeling. Thus, some pruning strategies are proposed for reducing redundancy based on the previously identified 2-hop labeling which makes [17] much faster than [16]. However, there still three problems in [17]. Firstly, if G is an undirected graph, it is impossible to generate a DAG by removing some vertices in G. Secondly, if G is not a sparse directed graph, there may exit a large number of SCC in G, so that a large number of vertices need to be removed from G to generate a DAG. Thus, the size of 2-hop labeling in G tends to be very large. Finally, the pruning strategies are based on all previously identified 2-hop labeling, which need to be cached in memory; otherwise, the frequent swap between memory and space will affect the performance dramatically. The running time of redundancy checking is also very high.

In order to adjust method in [17] for undirected data graph G, Zhou et al. proposed a "betweenness" based method to compute 2-hop distance labeling for G in [38]. The "betweenness" measures the relative importance of a vertex that is needed by others when connecting along shortest paths, where vertices that occur on more shortest-paths between other vertices have higher betweenness value. Based on this notation, we select some vertices with high betweenness value in G as 2-hop centers. Since computing betweenness is expensive in running time, a simple random sampling approach is proposed to estimate betweenness. Specifically, the *top-k* vertices with the highest estimated betweenness value are selected as 2-hop centers. Then, at each step, some of these selected vertices are removed in order to separate G into subgraphs, which is similar to method in [17]. This process continue until the subgraphs of G is small enough to be compute 2-hop labeling directly by method in [16].

2.2.3 Extend Reachability Join

According to Equation (1), an reachability join (*R-join*) algorithm was discussed in [25]. The main idea of this algorithm is as follows: based on the 2-hop labeling method [17][16], for each center w, two clusters $F(w)$ and $T(w)$ of vertices are defined, where via w every vertex u in $F(w)$ can reach every vertex u' in $T(w)$. Then, an index structure is built based on these clusters, by which for each vertex label pair (l, l'), all those centers w will be stored in a *W*-Table if w is in $F(w)$ and labeled l or in $T(w)$ and labeled l'. Thus, when a query Q is submitted, for each edge (v, v') labeled, for example, with (A, B) in Q, all those centers w will be searched such that in the *W*-table there exists at least a vertex u labeled A in $F(w)$, and there exists at least a vertex u' labeled B in $T(w)$. The Cartesian Product of vertices labeled A in $F(w)$ and vertices labeled B in $T(w)$ will form the matches of (v, v') in Q. This operation is called an *R-join*. When the number of edges in Q is larger than one, a series of *R-joins* (called *MR-join*) need to be conducted. By doing this, the classical natural join with join order selection [25] is used to generate the final pattern matching results. The worst time complexity of this method is bounded by $O(\prod_i |R_i|)$, where R_i is a subset of vertices in G with the same label as $label(v_i)$ ($v_i \in V(Q), i = 1, \dots, n$).

We can simply extend *R-join* to distance pattern match by using 2-hop distance labeling instead of reachability labeling. Given one query edge

$e = (v_i, v_j)$, we first obtain two lists $R_i = R(label(v_i))$ and $R_j = R\left(label(v_j)\right)$. In the second step, for each vertex pair (u, u') where $u \in R_i$ and $u' \in R_j$ in the Cartesian product, we need to compute the $d = Dist_{sp}(u, u')$ by Equation 1. If $d \leq \delta$, (u, u') is a matching result. Since the large number of shortest distance computation is at least $|R_i| \cdot |R_j|$, the running time of this naïve extended *R-join* method (called *ER-join*) is bounded by $O(|R_i||R_j|\sqrt{E(G)})$, which is really very time consuming.

2.2.4 Distance-based Join

In [24], the authors extended the idea of [18] by proposing a Distance-based join (*D-join*) for handling pattern matching queries with δ. By this method, for each edge $e = (v_i, v_j)$ in Q with label (A, B), a *D-join* algorithm is conducted to get all the matches in *G*, according to Equation 3 given below, where R_i and R_j are two lists respectively corresponding to v_i and v_j in *Q*, where u and u' are two vertices respectively in the two lists, and R_{ij} is the a relation that contains all the matches got from join operation.

$$R_{ij} = R_i \underset{Dist_{sp}(u,\, u') \leq \delta}{\bowtie} R_j \tag{3}$$

In order to reduce the cost of this join, the so-called *LLR Embedding* technique discussed in [12][33] is utilized to map all vertices in *G* into the points of a *k*-dimensional vector space. Here *k* is selected to be $O(\log|V(G)|)$ to save space. Then, the *Chebyshev* distance [28] between each pair of points *u* and *v* in the vector space, referred to as $L_\infty(u, v)$, is computed. In comparison with the approach discussed in [18], this method is more efficient since the Chebyshev distance is easy to calculate. Furthermore, the *k-medoids* algorithm [29] is used to divide each R_i (more exactly, the points corresponding to R_i) into different clusters $C_{ik}(k = 1, \ldots, l \textit{ for some } l)$. For each cluster C_{ik}, a center c_{ik} is determined and then the radius of C_{ik}, denoted as $r(C_{ik})$, is defined to be the maximal L_∞-distance between c_{ik} and a point in C_{ik}. During a *D-join* process between lists R_i and R_j, such clusters can be used to reduce computation by checking whether $L_\infty(c_{ik}, c_{jk'}) > r(C_{ik}) + r(C_{jk'}) + \delta$. If it is the case, the corresponding join (i.e., $C_{ik} \bowtie C_{ik'}$) need not be carried out since the L_∞-distance between any two points $u \in C_{ik}$ and $u' \in C_{jk'}$ must be larger than δ. By using the above main pruning method along with *Neighbor Area Pruning* and *Triangle Inequality Pruning*, all candidate matching results are evaluated, which will be further checked by a *2-hop labeling* technique in order to get the final results.

Although above *D-join* algorithm is much better than Extended *R-join* algorithm to obtain all pair-wised shortest-path distances, it still need much indexing time and space by using *2-hop labeling* and *LLR-embedding*.

2.2.5 Natural Join and Join Order Selection

Note that the above two important algorithms, *R-join* and *D-join*, are limited to obtain all relations R_{ij} corresponding to all the edges in query Q, which can be described by Equation 2. However, since the query Q is normally a graph, such relations R_{ij} are not final matching results and a classical natural join operation should be applied for this task. Although such operation is given different names, such as *Interleave R-join* and *Multi D-join* (*MD-join*), separately, they are essentially using a same method, the classical natural join, but with the different strategies of optimization.

Concretely, before natural joins, we assume that the join order, which is a traversal order in query Q, is specified. We will discuses specifically how to choose a better join order later. According to a traversal order in Q, we visit one edge $e = (v_i, v_j)$ in Q from vertex v_i in each step. If v_j is new encountered, such e is called a *forward edge*; if v_j has already been visited, such a e is called a *backward edge*. Essentially, only the forward edges need perform *R-join* or *D-join* algorithm, while a backward edge is a select operation. Given a query Q, a subgraph Q' induced by all visited edges in Q is called status. All the matches of Q' obtained by the natural joins are stored in a table $MR(Q')$, in which columns correspond to vertices v_i in Q'. Initially, Q' is status NULL and then in each step $Q' := Q' + e$, where e is the query edge selected by the join order. this process continue until $Q' = Q$, which means all the query edges have been visited and $MR(Q)$ is the final matches of query Q.

The above natural join is used in both Interleave R-join and MD-join, and the different part is in the join order selection. For the former, in order to get a perfect join order, a traditional dynamic programming algorithm [24] is adopted based on a cost model analysis. However, the time complexity of this solution is up to $O(n^2 \cdot 2^n)$, where $n = V(Q)$, which is inefficient due to the large solution space, especially when $|E(Q)|$ is large; for the latter, a simple yet efficient greedy solution is proposed to find a good join order, which is performing backward edge as early as possible if there is one or more backward edge in each natural join step.

However, no matter what join order we select, the time complexity of natural join is still $O(\prod_i |R_i|)$ in the worst case, which is, in fact, NP-complete. The improvement after using a good join order is not obvious and even can be ignored in the case that the large amount time is used for natural joins. We can do much better, in this work, besides the join order selection we propose a novel two-level filtering algorithm to remove all the redundant data in the large searching space before Natural Joins are conducted, which can significantly speed up the Natural Join process. In addition, the above greedy solution to the join order selection can also be applied for a little improvement during Natural Joins.

CHAPTER 3 PRELIMINARIES

In this chapter, we give the formal definition of the pattern matching queries over a large data graph G. Firstly, in this thesis, unless otherwise specified, G is a vertex-labeled graph. Secondly, we will use the shortest- path distance to measure the distance between each pair of two vertices in G.

3.1 Definitions

Definition 3.1 (*Data Graph G*) A data graph $G = (V(G), E(G), \Sigma)$ is a vertex-labeled graph. Here, $V(G)$ is a set of labeled vertices, $E(G)$ is a set of edges (ordered pairs) each with a nonnegative weight, and Σ is a set of vertex labels. Each vertex $u \in V(G)$ is assigned a label $l \in \Sigma$, denoted as $label(u) = l$.

Definition 3.2 (*Query Q*) A query Q is a vertex-labeled graph, $Q = (V(Q), E(Q))$. Here, $V(Q)$ is a set of labeled vertices, and $E(Q)$ is a set of edges. Each vertex $v \in V(Q)$ is also assigned a label $l \in \Sigma$, denoted as $label(v) = l$.

Definition 3.3 (List $R(l)$ and R_i) Given a data graph G, we use $R(l)$ to represent a list that includes all those vertices u in G whose labels are $l \in \Sigma$, i.e., $label(u) = l$ for each $u \in R(l)$. Let $v_i \in V(Q)$, we also use R_i to represent a list $R(l = label(v_i))$.

Definition 3.4 (Edge Query with δ) Given a data graph G, an edge $e = (v_i, v_j)$ in a query graph Q and a parameter δ, the evaluation of e reports all matching pairs $\langle u_i, u_j \rangle$ in G if the following conditions hold:

1) $label(u_i) = label(v_i)$ and $label(u_j) = label(v_j)$;

2) The distance from u_i to u_j in G is not larger than δ. That is, $Dist_{sp}(u_i\ , u_j) \le \delta$.

Definition 3.5 (*Pattern Matching Query with* δ) Given a data graph *G*, a query graph *Q* with *n* vertices $\{v_1, \dots, v_n\}$ and a parameter δ, the evaluation of *Q* reports all matching results $\langle u_1, \dots\ u_n \rangle$ in G if the following conditions hold:

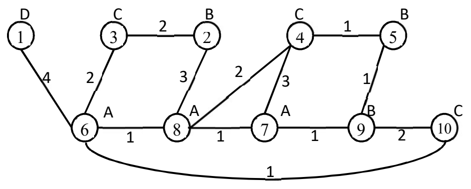
1) $label(u_i) = label(v_i)$ for $1 \le i \le n$;
2) For any edge $(v_i, v_j) \in Q$, the shortest path distance between u_i and u_j in G is no larger than δ, i.e., $Dist_{sp}(u_i\ , u_j) \le \delta$ $(1 \le i \le n)$.

The common symbols used in this thesis are summarized in Table 1.

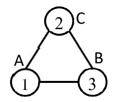
data graph G		query graph Q	
$N = V(G)$	the vertex set of G	$n = V(Q)$	the vertex set of Q
$M = E(G)$	the edge set of G	$m = E(Q)$	the edge set of Q
u_i	a vertex in G	v_i	a vertex in Q
$label(u_i)$	the label of u_i	$label(v_i)$	the label of v_i

Table 1: Meaning of used symbols.

3.2 Example



(a) Data Graph G



(b) Query Q

Figure 1: An examples of a data graph *G* and a query *Q*.

Example 1: In Figure 1, we have an example, an undirected and weighted data graph G, in which the numbers inside the vertices, $\{u_1, \dots, u_{10}\}$, are their IDs and the letters attached to them are their labels, and the numbers besides the edges are their weights. There are altogether 4 labels, $\Sigma = \{A, B, C, D\}$. In Figure 1(b), we show a simple query, an undirected and unweighted graph Q, which contains three vertices, $\{v_1, v_2, v_3\}$ labeled with A, B and C, respectively, and three query edges $\{(v_1, v_2), (v_2, v_3),\ (v_3, v_1)\}$. According to Definition 3.3, three lists $R_1 = R(A)$, $R_2 = R(B)$ and $R_3 = R(C)$ will be constructed. There are $R_1 = \{u_6, u_7, u_8\}$, $R_2 = \{u_3, u_5, u_9\}$ and $R_3 = \{u_2, u_{10}, u_4\}$. According to Definition 3.4 and 3.5, if we choose $\delta = 1$, it has no any matching result; if we choose $\delta = 2$, the matching results are $\{\langle u_8, u_9, u_{10}\rangle, \langle u_7, u_9, u_4\rangle\}$; if we choose $\delta = 3$, it is easy to know that there should be more matching results then before. Obviously, the parameter δ cannot be too large for measuring patter matching properly.

CHAPTER 4 FRAMEWORK

In this chapter, we describe the general framework of our new method for inexact pattern matching queries over a data graph G. We first need to give two definitions, Δ-Transitive Closures and Relations.

4.1 The Δ-Transitive Closure Definition

Definition 4.1 (Δ-Transitive Closure G^{Δ}) Given a data graph $G = (V(G), E(G), \Sigma)$ with vertex set $V(G) = \{u_1, u_2, \dots, u_n\}$, we define the Δ-transitive closure of G as the graph $G^{\Delta} = (V(G), E^{\Delta}(G), \Sigma)$ where $E^{\Delta}(G) = \{(u_i, u_j, w): w = Dist_{sp}(u_i, u_j)\ and\ w \le \Delta\}$.

In Definition 4.1, we introduce the notation of the Δ-Transitive Closure for data graph G denoted as G^{Δ}. Intuitively, all vertex pairs (u_i, u_j) in G (except for own edges (u_i, u_j)) where $Dist_{sp}(u_i, u_j) \le \Delta$ are new edges added into the original data graph G, forming G^{Δ}. We can perform the pattern matching query in G^{Δ} instead of G when receiving query Q with δ. By doing this, we transform an inexact matching to an exact matching, since all the shorted-path distances between vertex pairs in G are stored in G^{Δ} as its edges.

One important issue is how to decide the value of Δ for G^{Δ}. In the worst case the size of G^{Δ} is $O(|V(G)|^2)$ if Δ is very large, which is forbiddingly high for a large data graph G. By contrast, if Δ is too small, the size of G^{Δ} will also be small, but it needs to reconstruct G^{Δ} if the parameter δ for Q is $> \Delta$. However, Δ could not be too big, for the pattern matching query is to find subgraphs that is similar to the pattern. Adopting big Δ is meaningless to measure such similarity. This is well proved by experiments in Chapter 8, from which we can see that the number of matching results will increase dramatically with the growth of δ and such large scale of matching results tend to be meaningless in reality. Therefore, we prefer a proper value for Δ, which has not only reasonable indexing time and size but also big enough value for general patter matching queries.

4.2 The Relation Definition

Definition 4.2 ($R(l,l')$) Given a data graph G, we use relation $R(l,l')$ to represent all edges $\{(u,u',w)\} \subseteq E^{\Delta}(G^{\Delta})$ where $u \in R(l)$, $u' \in R(l')$. Each $R(l,l')$ corresponds to a label pair (l,l') where $l,l' \in \Sigma$ and $l \neq l'$. Thus, the total number of $R(l,l')$ is $\frac{k(k-1)}{2}$ where $k = |\Sigma|$, since $R(l,l')$ equal to $R(l',l)$. Actually, G^{Δ} is stored as relation $R(l,l')$.

Definition 4.3 (Relation R_{ij}) When receiving a query Q with δ, let $(v_i,v_j) \in E(Q)$, we use notation R_{ij} to represent $\{(u,u',w) \in E^{\Delta}: u \in R_i \ and \ u' \in R_j \ and \ w \leq \delta\}$ or $\{(u,u',w) \in R\left(label(v_i), label(v_j)\right) : w \leq \delta\}$. Each edge (v_i,v_j) corresponds to a relation R_{ij}. Considering the query edge (v_i,v_j) is equivalent to (v_j,v_i), R_{ij} and R_{ij} are actually one relation but different order of columns.

Definition 4.4 (Reduced Relation R'_{ij} and Further Reduced Relation R''_{ij}) Giving a query Q with δ, each relation R_{ij} is generated according to Definition 4.2. In relation R_{ij}, most redundant tuples can be filtered by using *Domain Filtering* algorithm, which is denoted as R'_{ij}; the rest of redundant tuples in each relation R'_{ij} can be further filtered by using *Relation Filtering* algorithm, which is denoted as R''_{ij}. Here, the redundant tuples are one kind of tuples that are not necessary to participate the Natural Joins, and removing such tuples will not influence our matching results.

Definition 4.5 (Tuples) Give all relations R_{ij}, the tuples are all their items $(u, u', w) \in R_{ij}$.

4.3 Framework

Now, we begin to discuss the general framework of our method, which is illustrated by Figure 2. As discussed in Chapter 1, there are two main parts: Relation Construction and Matching Construction. By the former, we first generate G^{Δ} (see Definition 4.2) and then construct all relations $R(l, l')$. Note that this can be pre-computed before receiving a query Q, which is *offline* or an *index* operation. After that, when receiving a query Q with δ, the online operation gets started. All relations R_{ij} can be extract from $R(l, l')$ efficiently. The algorithms related to G^{Δ} are showed in Chapter 5. By the latter, based on relation R_{ij} obtained in last part, it can be reduced to R'_{ij} by *DomainFiltering* algorithm and further reduced to R''_{ij} by *RelationFiltering* algorithm, which is called two-level filtering and will be discussed in Chapter 6. Finally, the classical natural join (see Chapter 7) is performed on such fully filtered relations R''_{ij} in order to obtain all the matching results.

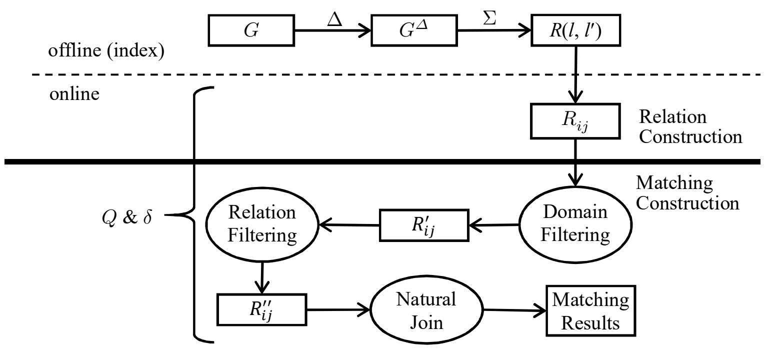


Figure 2: The framework of our method.

CHAPTER 5 G^{Δ} CONSTRUCTION

As we described in the framework of our method, in order to generate all relations R_{ij} (Definition 4.3) for the data graph G, we should first construct the Δ-Transitive ClosureG^{Δ} (Definition 4.1). In other words, in G all pair-wised shortest-path distances within Δ should be pre-calculated. This is a well-researched area and many algorithms have been developed. For unweighted data graphs the classical Breadth-First Search (BFS) [37], as a best choice, is simple and efficient, while for weighted data graphs we choose Dijkstra's Algorithm [37]. In this chapter, we will discuss these two algorithms.

5.1 G^{Δ}-BFS

```
Algorithm 1: $G^{\Delta}$-BFS($G$, Δ)
Input: data graph $G$ and $\Delta$.
Output: $E^{\Delta}$.
1.  $E^{\Delta} = \Phi$;
2.  for each $u \in V(G)$ do
3.      $u.color = WHITE, u.dist = \infty$;
4.  for each $s \in V(G)$ do
5.      call Δ-BFS($G$, $s$, $\Delta$, $E^{\Delta}$);
6.  return $E^{\Delta}$;
Δ-BFS($G$, $s$, $\Delta$, $E^{\Delta}$)
7.  $s.color = GRAY, s.dist = 0$;
8.  $LIST := \Phi$, $LIST$.append($s$);
9.  $QUEUE := \Phi$ , ENQUEUE ($QUEUE$, $s$);
10. while $QUEUE \neq \Phi$ do
11.     $u =$ DEQUEUE($QUEUE$);
12.     if $u.dist \geq \Delta$ do break;
13.     $E^{\Delta} = E^{\Delta} \cup (s, u, u.dist)$;
14.     for each $u' \in G.Adj[u]$ do
15.         if $u'.color == WHITE$ then
16.             $u'.color = GRAY, u'.dist = u.dist + 1$;
17.             $QUEUE$.enqeue($u'$), $LIST$.append($u'$);
```

```
18.     $u.color = BLACK$;
19.for each $u \in LIST$ do
20.     $u.color = WHITE, u.dist = \infty$;
```

Algorithm 1 lists the steps to construct G^{Δ} for unweighted data graphs G. In line 5, we can see the BFS is performed for each vertex $s \in V(G)$ as a starter. Here, the BFS algorithm we use is generally the same as classical one except in line 12-13. Specifically, in line 12, since the searching after $u.dist \geq \Delta$ is unnecessary, the loop will stop; and in line 13, the items $(s, u, u.dist)$ are added to E^{Δ} as new edge in each step. In other words, for each starter $s \in V(G)$, instead of searching the whole graph, our searching space is limited by δ, which can reduce the running time and space effectively. In addition, in line 8 and 17 we use the $LIST$ in order to record all visited vertices that are reset in line 19-20.

It is easy to know the total running time of the classical BFS procedure is $O(N + M)$, where $N = |V(G)|$ and $M = |E(G)|$ (See Table 1) in the worst case. Since the BFS procedure is executed on each starter $s \in V(G)$, the time complexity of our Algorithm 1 is $O\big(N(N + M)\big) = O(NM)$ at the worst case. However, on average, the time complexity of Algorithm 1 should be $O(Nd^{\Delta})$, where d is the average degree of vertices in G. Actually, in real data graphs, especially sparse graphs, we have $d^{\Delta} \ll M$ if Δ is relatively small, which leads to a better performance.

By the space complexity, it is $O(N^2)$ in the worst case, which is prohibitively high in large data graphs. Nevertheless, on average, it needs only $O(Nd^{\Delta})$ to store E^{Δ}.

5.2 G^{Δ}-DIJKSTRA

Algorithm 2: G^{Δ}-DIJKSTRA(G, Δ)

```
Input: data graph $G$ and $\Delta$.
Output: $E^{\Delta}$.
1.  $E^{\Delta} = \Phi$;
2.  for each $u \in V(G)$ do
3.      $u.color = WHITE, u.dist = \infty$;
4.  for each $s \in V(G)$ do
5.      call Δ-DIJKSTRA ($G$, $s$, $\Delta$, $E^{\Delta}$);
6.  return $E^{\Delta}$;
Δ-DIJKSTRA ($G$, $s$, $\Delta$, $E^{\Delta}$)
7.  $s.color = GRAY$, $s.dist = 0$;
8.  $LIST := \Phi$, $LIST$.appedn($s$);
9.  $QUEUE := \Phi$, $QUEUE$.enqueue($s$);
10. while $QUEUE \neq \Phi$ do
11.     $u = QUEUE$.extract-min();
12.     if $u.dist \geq \Delta$ do break;
13.     $E^{\Delta} = E^{\Delta} \cup (s, u, u.dist)$;
14.     for each $u' \in G.Adj[u]$ do
15.         if $u'.color \neq BLACK$ then
16.             if $u'.dist > u.dist + w(u, u')$ do
17.                 $u'.dist = u.dist + w(u, u')$;
18.         if $u'.color == WHITE$ do
19.             $u'.color = GRAY$;
20.             $QUEUE$.enqueue($u'$), $LIST$.append($u'$);
21.     $u.color = BLACK$;
22. for each $u \in LIST$ do
23.     $u.color = WHITE, u.dist = \infty$;
```

We adopt Dijkstra's algorithm to deal with the weighted data graphs, as shown in Algorithm 2. Here the Dijkstra's algorithm we use is generally the same as classical one except in line 12-13, which is analogy to algorithm 1. So, here its discussion is omitted.

It is well-known that total running time of the Dijkstra's procedure is bounded by $O((N+M)logN)$, where $N = |V(G)|$ and $M = |E(G)|$ (see Table 1), by implementing the min-priority queue as a Fibonacci heap. Since the BFS procedure is executed on each starter $s \in V(G)$, the time complexity of Algorithm 2 is $O(N(N+M)logN) = O(NMlogN)$ in the worst case. Compared with the Algorithms 1, whose time complexity is $O(NM)$ in the worst case, the Algorithm 2 is little slower. This is why we adopt Algorithm 1 for unweighted graph rather than Algorithm 2, even Algorithm 2 can compatible unweighted graph as well. Similarly, on average, the time complexity of Algorithm 2 should be $O(Nd^{\Delta}logd^{\Delta})$, where d is the average degree of vertices in G, and, like in Algorithm 1, $d^{\Delta} \ll M$. The space complexity of Algorithm 2 is exactly the same as Algorithm 1.

5.3 Example

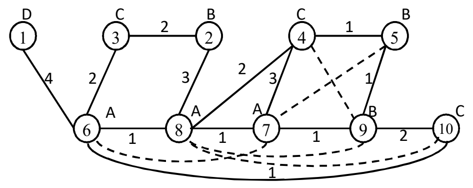


Figure 3: The Δ-Transitive Closure of data graph in Figure 1(a) when choosing $\Delta = 2$.

Example 2: Continue with Example 1. In Figure 3, we create an Δ-Transitive Closure as an example for the data graphs showed in Figure 1(a) when choosing $\varDelta = 2$, in which the dash lines are the new added edges. It is clear to see that the size of Δ-Transitive Closure is much smaller than the traditional Transitive Closure of G.

CHAPTER 6 Two-Level Filtering

After getting all relations R_{ij} (see Definition 4.3), we can use classical natural joins to extract all the matching results. Before a natural join procedure, we can use the *Domain Filtering* algorithm to obtain reduced relations R'_{ij} and the *Relation Filtering* algorithm to obtain further reduced relations R''_{ij} when receiving a query Q with δ. It is true that doing natural joins on R''_{ij} is much faster than directly on R_{ij}. In this chapter, we will discuss these two filtering algorithms in great details.

6.1 Domain Filtering

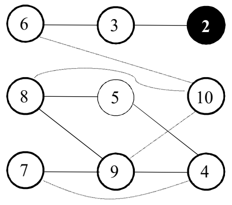
6.1.1 Algorithm Description

Definition 6.1 (*supports*) Let (v_i, v_j) be an edge in query Q. Let u and u' be two vertices in G belonging to $R_i = R(label(v_i))$ and $R_j = R(label(v_j))$, respectively. If $Dist_{sp}(u, u') \leq \delta$, then we say, u and u' support each other.

Definition 6.2 (*fully-supported*) Let v_i be a vertex in query Q. Let u be a vertex in G belonging to $R_i = R(label(v_i))$. If for any edge (v_i, v_j) (or (v_j, v_i))) incident to v_i there exists at least one vertex u' in $R_j = R(label(v_j))$ such that $Dist_{sp}(u, u') \leq \delta$ (or $Dist_{sp}(u', u) \leq \delta$), we say, u is *fully-supported*.

By the *Domain Filtering*, what we want is to remove all those vertices which are not fully-supported since any of such vertices cannot appear in any pattern matching answer to query Q. For this purpose, we will associate each vertex u in G with two data structures: a *support list*, denoted as $u.S$, which contains all the vertices supporting u; and a *counter*, denoted as $u.C$, which is a list of n entries, $u.C[1], \ldots, u.C[n]$, with each $u.C[i](i = 1, \dots, n)$ recording how many supports u has from R_i. Each R_i correspond to a vertex $v_i(i = 1, \dots, n)$ in query Q. In $u.S$, each element is of the form $\langle j, u' \rangle$ indicating that u' is a vertex from R_j.

In Figure 4(a), we show all the supports of the vertices shown in Figure 1(a) when δ is set to 2, where an edge connecting two vertices u and u' indicates that they support each other. In Figure 4(b), we show the data structures of supports and counters. For example, for u_6 in $R_1 = R(A)$ shown in Figure 4(a), its support list $u_6.S$ will contain $\langle 2, u_3 \rangle$ from $R_2 = R(B)$ and $\langle 3, u_3 \rangle$ from $R_3 = R(C)$) and not any support from R_1. Accordingly, $u_6.C[2]$ and $u_6.C[3]$ are both set to 1 and $u_6.C[1]$ is meaningless as $u_6 \in R_1$. All the counters associated with u_6 can be represented by an array [-, 1, 1].



(a)

vertex	support list S	counter C
u_2	$\{\langle 2, u_3 \rangle\}$	[0, 1, -]
u_3	$\{\langle 1, u_6 \rangle, \langle 3, u_2 \rangle\}$	[1, -, 1]
u_4	$\{\langle 1, u_7 \rangle, \langle 2, u_5 \rangle, \langle 2, u_9 \rangle\}$	[1, 2, -]
u_5	$\{\langle 1, u_8 \rangle, \langle 3, u_4 \rangle\}$	[1, -, 1]
u_6	$\{\langle 2, u_3 \rangle, \langle 3, u_{10} \rangle\}$	[-, 1, 1]
u_7	$\{\langle 2, u_9 \rangle, \langle 3, u_4 \rangle\}$	[-, 1, 1]
u_8	$\{\langle 2, u_5 \rangle, \langle 2, u_9 \rangle, \langle 3, u_{10} \rangle\}$	[-, 2, 1]
u_9	$\{\langle 1, u_8 \rangle, \langle 1, u_7 \rangle, \langle 3, u_4 \rangle, \langle 3, u_{10} \rangle\}$	[2, -, 2]
u_{10}	$\{\langle 1, u_6 \rangle, \langle 1, u_8 \rangle\}$	[2, 1, -]

(b)

Figure 4: The data structures of vertices for the domain filtering algorithm.

Based on support lists and counters, the domain filtering can be done very efficiently in two phases by using a *STACK* to control the propagation working process:

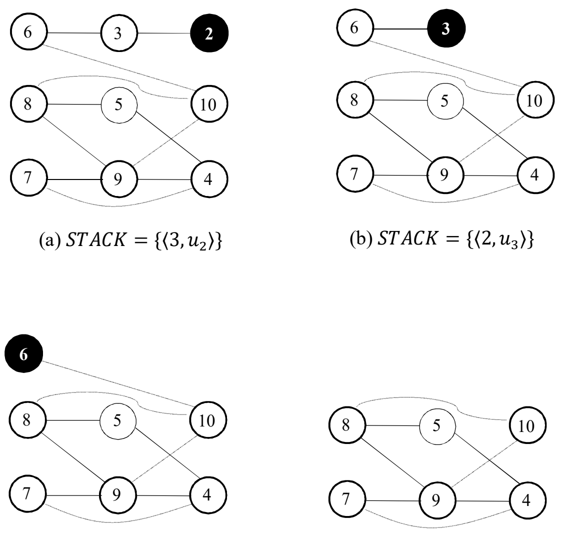
1) In the first phase, when receiving a query Q with n vertices, we will process each query edge $e = (v_i, v_j)$ in $E(Q)$ one by one. Specifically, for each query edge e, we first construct the supports and counters for vertices $u \in R_i, R_j$ according to relation R_{ij} generated by G^{Δ} (Algorithm 1 or Algorithm 2). Then, we will check each vertex $u \in R_i (i = 1, \ldots, n)$ to see whether its counter $u.C[j] = 0$. If it is the case, we know that u does not have any supports from R_j and u should be removed from R_i and, at the same time, pushed $\langle i, u \rangle$ into $STACK$. Also, the same operation should be performed on each vertex $u' \in R_j$,
2) In the second phase, we will pop out the elements from the $STACK$. Let $\langle i, u \rangle$ be the element currently popped out of the *STACK*. Then, for each element $\langle j, u' \rangle$ in $u.S$, we will decrease $u'.C[i]$ by one. It is because u is a deleted vertex from R_i and thus u' has lost one support from list R_i. More importantly, if $u'.C[i]$ becomes 0, u' should be removed from R_j, too; and $\langle j, u' \rangle$ should be pushed into $STACK$. This process repeats until $STACK$ becomes empty and all the remained vertices $u \in R_i (i = 1, \ldots, n)$ must be fully supported now. In this way, R_{ij} will be reduced to $R'_{ij} (i, j = 1, \ldots, n)$.

```
Algorithm 3: DomainFiltering($Q$, $\delta$, all $R_{ij}$)
Input: query $Q$, $\delta$ and all relations $R_{ij}$.
Output: reduced relations $R'_{ij}$.
//phase 1: construct the support lists, counters and STACK.
1. $STACK := \Phi$;
2. for each $(v_i, v_j) \in E(Q)$ do
3.     get $R_i, R_j, R_{ij}$
4.     for each $(u, u') \in R_{ij}$ do
5.         $u.S$.append($\langle j, u' \rangle$), $u.C[j]++$;
```

```
6.          $u'.S$.append($\langle i,u\rangle$), $u'.C[i]++$;
7.      for each $u \in R_i$ do
8.          if $u.C[j] == 0$ then
9.              $R_i$.remove($u$), $STACK$.push($\langle i,u\rangle$);
10.     for each $u' \in R_j$ do
11.         if $u'.C[i] == 0$ then
12.             $R_j$.remove($u'$), $STACK$.push($\langle j,u'\rangle$);
//phase 2: process the STACK.
13. while $STACK \neq \Phi$ do
14.     $\langle i,u\rangle = STACK$.pop();
15.     for each $\langle j,u'\rangle \in u.S$ do
16.         if $(--u'.C[i]) == 0$ then
17.             $R_j$.remove($u'$), $STACK$.push($\langle j,u'\rangle$);
18. return $R'_{ij} = R_{ij} - \{(u,u') : u \notin R_i \ or \ u' \notin R_j\}$;
```

The Algorithm 3 *DomainFiltering*( ) is a formal description of the above working process. In this algorithm, lines 1 – 12 make up the first phase, in which we construct support lists and counters, and initialize *STACK* by appending all removed vertices. In the second phase (lines 13 – 17), *STACK* is utilized to accommodate and propagate any vertex which becomes non-supported after some related vertices have been eliminated. In line 18, the pruned relations R'_{ij} is returned.

6.1.2 Example


(a) $STACK = \{\langle 3, u_2 \rangle\}$ (b) $STACK = \{\langle 2, u_3 \rangle\}$

(c) $STACK = \{\langle 1, u_6 \rangle\}$ (d) $STACK = \Phi$

Figure 5: The Execution of Algorithm 3.

Example 3:

- Consider Figure 3 in Example 2 once again. We apply the Algorithm 3 *DomainFiltering*( ) to the three domains $R_1 = R(A)$, $R_2 = R(B)$, and $R_3 = R(C)$ shown in the figure. After the first phase, only one vertex u_2 is removed from R_3 and thus only $\langle 3, u_2 \rangle$ is pushed into $STACK$, because among all the counters only $u_2.C[1]$ is 0. We illustrate this by the black node in Figure 5(a).
- In the second phase, we will first pop the top element $\langle 3, u_2 \rangle$ out of $STACK$, and then check the only element in $u_2.S = \{\langle 2, u_3 \rangle\}$. Doing this, $u_3.C[3] = 1$ will be decreased by 1, becoming 0. So u_3 will be removed from R_2 and, at the same time, $\langle 2, u_3 \rangle$ will be pushed into *STACK*, which is illustrate by the black node in Figure 5(b).
- In a next step, we will pop $\langle 2, u_3 \rangle$ out of *STACK*. Again, we will check all elements in $u_3.S = \{\langle 1, u_6 \rangle, \langle 3, u_2 \rangle\}$, but u_2 is removed

from R_3 in the above step. Doing this, $u_6.C[2] == 1$ will be decreased by 1, becoming 0. So, $\langle 2, u_3 \rangle$ will be pushed into *STACK* (see Figure 5(c) for illustration.)

- When popping $\langle 1, u_6 \rangle$ out of *STACK*, we will do the same operations: checking $u_6.S = \{\langle 2, u_3 \rangle, \langle 3, u_{10} \rangle\}$, in which u_3 was removed in last step. We decrease $u_{10}.C[1] = 2$ by 1. Since $u_{10}.C[1] = 1$ is still larger than 0, it will not be removed from R_3 and no vertex will be pushed into $STACK$. Now, $STACK$ becomes *empty* and the second phase stops. What is left is shown in Figure 5(d), from which we can see that all the domains R_1, R_2, and R_3 are reduced. In other words, all the vertices in domains which are not fully supported are removed.

6.1.3 Computational Complexity and Correctness

In this subsection, we analyze the running time of *DomainFiltering*( ) and prove its correctness.

6.1.3.1 Computational complexity

To know the time complexity of the algorithm, we first analyze the first phase. Considering the inner for-loop at lines 3-12, its running time is bounded by

$$\sum_{(v_i, v_j) \in Q} (|R_{ij}| + |R_i| + |R_j|) \leq O(\sum_{(v_i, v_j) \in Q} (|R_{ij}|).$$

Now we analyze the running time of the second phase. For this, we consider the bounds on the **while**-loop at line 13 and the **for**-loop at line 15. We first notice that each different $\langle i, u \rangle$ can be pushed into *STACK* at most once. Given that $\langle i, u \rangle$ has been removed from R_i, the only elements that can be impacted are those in an R_j such that v_j is connected to v_i in query Q. Let d_i be the degree of v_i. Since i can appear at most D times at line 15 and there are at most $d_i D$ elements of $u.S$ for a given i, line 16 can be executed at most

$$\sum_{i=1}^{n} Dd_i D = D^2 \sum_{i=1}^{n} d_i = mD^2$$

times, where $m = |E(Q)|$ and $D = \max\{|R_1|, \dots, |R_n|\}$. Thus, the time complexity of *DomainFiltering*( ) is bounded by $O(mD^2)$.

The space overhead is made up of two parts: the space for storing support lists and the space for counters. Since in the worst case for a vertex in R_i its support list may contain all vertices in R_j for each j such that $(v_i, v_j) \in Q$, the first part is bounded by

$$\sum_{(v_i, v_j) \in Q} |R_i||R_j| \leq mD^2$$

And the number of counters is obviously bounded by $O(n^2 D)$, where $n = |V(Q)|$.

6.1.3.2 Correctness

To prove the correctness of *DomainFiltering*( ), we need to explain that any remaining vertex in R_i is *fully-supported* when *STACK* becomes *empty*.

Proposition 1 Let Q be a query graph containing n vertices $v_1, \dots, v_n$. Let $R_1 = R(label(v_1)), \dots, R_n = R(label(v_n))$ be n lists constructed from a data graph G according to Q. When *STACK* becomes *empty* during the execution *DomainFiltering*( ) algorithm, each vertex in the remaining lists must be *fully-supported*.

Proof. Assume that when *STACK* becomes *empty*, there is still a vertex u in some R_i, which is not *fully-supported*. Then, there must be an integer j such that no vertex in R_j supporting u. Thus, we have $u.C[j] == 0$ and by executing lines 16-17, $\langle i, u \rangle$ will definitely be pushed into *STACK*. It is a contradiction. □

6.2 Relation Filtering

6.2.1 Algorithm Description

Let query Q has n vertices $v_i \in V(Q)(i = 1, \dots, n)$, and m edges $(v_i, v_j) \in E(Q)$. After running *DomainFiltering*( ) algorithm, we obtain n reduced lists R'_i $(i = 1, \dots, n)$, and m reduced relations $R'_{ij}((v_i, v_j) \in Q)$, in which most of redundant tuples are eliminated.

However, we can do better. That is, we can use the *RelationFiltering*( ) algorithm to further reduce R'_{ij} to R''_{ij}. Specifically, we will view Q as a "complete" graph by adding a virtual bi-directed edge connecting v_i and v_j if $(v_i, v_j) \notin Q$, and the corresponding R'_{ij} is understood as an always *true* relation. Note that such virtual edges can greatly simplify the description but are only in our imagination. For each pair $\langle u, u' \rangle \in R'_{ij}$, which is not virtual, the following condition must be satisfied: if there exist two edges $(v_i, v_k), (v_k, v_j) \in Q$, there exists $u'' \in R'_k$ such that $\langle u, u'' \rangle \in R'_{ik}$ and $\langle u', u'' \rangle \in R'_{jk}$. Any pair $\langle u, u' \rangle \in R'_{ij}$ not satisfy such condition should be eliminated from R'_{ij}. We refer to this checking as a *triangle-check*, denoted as $tri[\langle i, j, k \rangle, \langle u, u', u'' \rangle]$. This tri operation will return true or false to indicate the above condition is satisfied or not. This is the basic idea of the *RelationFiltering*( ) algorithm.

Note that the above *triangle-check* needs to iterate all vertices $u'' \in R_k$ for each pair $\langle u, u' \rangle \in R'_{ij}$, which is not efficient. This procedure could be well speeded up by using multi-map which is a data structure like a mapping and can store elements formed by a combination of a key and multi-mapped values. By doing this, instead of checking all vertices $u'' \in R_k$ for each pair $\langle u, u' \rangle \in R'_{ij}$, we can obtain qualified vertices $u'' \in R_k$ at one step. We refer to this check as a *batch-triangle-check*, denoted as $btri[\langle i,j,k \rangle, \langle u,u' \rangle]$. Its procedure is simple. This $btri$ operation will return a set including all qualified vertices in R_k.

Obviously, if $\langle u, u' \rangle$ is removed from R'_{ij}, u and u' each will lose a support from R'_i and R'_j, respectively. So, in this way, we may also be able to remove a lot of vertices when they have lost all their supports from some other list.

Above process works in a way similar to the *DomainFiltering*( ) algorithm, but with the data structures for supports and counters somehow changed. Concretely, each vertex $u'' \in R'_k (k = 1, \dots, n)$ will be associated with a set of supports $u.S$ of the form $\langle i,j; u,u' \rangle$ where $\langle u, u' \rangle \in R'_{ij}$, indicating that the triangle-check holds: $tri[\langle i,j,k \rangle, \langle u,u',u'' \rangle]$. In addition, each pair $\langle u,u' \rangle$ in a relation R'_{ij}, will be associated with a list of n counters, denoted $\langle u,u' \rangle.C$. Each counter $\langle u,u' \rangle.C[k]$ in it records how many supports from R'_k.

Algorithm 4: *RelationFiltering*(Q, all R'_i, all R'_{ij})

Input: query Q that has n vertices $v_1, \dots, v_n$; All filtered lists R'_i.
Output: Further filtered relations $R''_{ij} (i, j = 1, \dots, n)$.

```
//phase 1: construct the supports, counters and STACK.
1. for each $e = (v_i, v_j) \in E(Q)$ do
2.     for each $k (k = 1, \dots n \ and \ k \neq i, j)$ do
3.         get $R'_i, R'_j, R'_k, R'_{ik}, R'_{jk}, R'_{ij}$;
4.         $MMAP_1 := \Phi, MMAP_2 := \Phi, SET := \Phi$;
5.         for each $\langle u, u'' \rangle \in R_{ik}$ do $MMAP_1[u]$.append($u''$);
6.         for each $\langle u', u'' \rangle \in R_{jk}$ do $MMAP_2[u']$.append($u''$);
```

```
7.        for each $\langle u, u' \rangle \in R'_{ij}$ do
8.            $SET := btri[\langle i, j, k \rangle, \langle u, u' \rangle] = MMAP_1[u] \cap MMAP_2[u']$;
9.            if $SET == \Phi$ then
10.               $R'_{ij}$.remove($\langle u, u' \rangle$);
11.               if $(--u.C[j]) == 0$ then
12.                   $R'_i$.remove($u$), STACK.push($\langle i, u \rangle$);
13.               if $(--u'.C[i]) == 0$ then
14.                   $R'_j$.remove($u'$), STACK.push($\langle j, u' \rangle$);
15.           else then
16.               $\langle u, u' \rangle.\boldsymbol{C}[k] = SET.size$;
17.               for each $u'' \in SET$ do $u''.S$.append($\langle i, j; u, u' \rangle$);
//phase 2: process the STACK.
18.while $STACK \neq \Phi$ do
19.    $\langle k, u'' \rangle$ = STACK.pop();
20.    for each $\langle i, j; u, u' \rangle \in u''.S$ do
21.        if $(--\langle u, u' \rangle.C[k]) == 0$ then
22.            $R'_{ij}$.remove($\langle u, u' \rangle$);
23.        if $(--u.C[j]) == 0$ then
24.            $R'_i$.remove($u$), STACK.push($\langle i, u \rangle$);
25.        if $(--u'.C[i]) == 0$ then
26.            $R'_j$.remove($u'$), STACK.push($\langle j, u' \rangle$);
27.return $R''_{ij} = R'_{ij} - \{(u, u'): u \notin R'_i \; or \; u' \notin R'_j\}$;
```

The Algorithm 4 *RelationFiltering*( ) is the formal description of above working process. In the first phase (lines 1–17), we create support lists for vertices and counters for edges as described above. In addition, a stack *STACK* is initialized with all the elements $\langle i, u \rangle$ such that u has been removed from the corresponding domain. In line 8, the *btri*( ) (*batch-triangle-check*) is used to speed up the *tri*( ) (*triangle-check*) by using multi-mapping data structure. In line 4-6, two multi-maps, $MMAP_1$ and $MMAP_2$, are initialized, and in line 8 the result set of *btri* is efficiently obtained by an operation of intersection. In the second phase (lines 18 – 26), we propagate the elimination of vertices which have no supports from at least a different list. In line 27, the pruned relations R''_{ij} is returned.

6.2.2 Example

Example 4: Continue with Example 3. When applying the algorithm *RelationFiltering*( ) to R'_1, R'_2 and R'_3, as well as R'_{12}, R'_{23}, and R'_{31} shown in Figure 5(d), the first phase will work as follows:

1) for $(v_1, v_2) \in E(Q)$ and $k = 3$:

   a. for $\langle u_8, u_5 \rangle$:

      $tri[\langle 1,2,3 \rangle, \langle u_8, u_5, u_{10} \rangle] = false$;

      $tri[\langle 1,2,3 \rangle, \langle u_8, u_5, u_4 \rangle] = false$;

      or $btri[\langle 1,2,3 \rangle, \langle u_8, u_5 \rangle] = \Phi$;

      remove $\langle u_8, u_5 \rangle$ from R_{12};

      $u_8.C[2] := u_8.C[2] - 1 = 2 - 1 = 1$;

      $u_5.C[1] := u_5.C[1] - 1 = 1 - 1 = 0$;

      remove u_5 from R_2;

      $STACK = \{\langle 2, u_5 \rangle\}$;

   b. for $\langle u_8, u_9 \rangle$:

      $tri[\langle 1,2,3 \rangle, \langle u_8, u_9, u_{10} \rangle] = true$;

      $tri[\langle 1,2,3 \rangle, \langle u_8, u_9, u_4 \rangle] = false$;

      or $btri[\langle 1,2,3 \rangle, \langle u_8, u_9 \rangle] = \{u_{10}\}$;

      $u_{10}.S = \{\langle 1,2; u_8, u_9 \rangle\}$;

      $\langle u_8, u_9 \rangle.C[3] := 1$;

   c. for $\langle u_7, u_9 \rangle$:

      $tri[\langle 1,2,3 \rangle, \langle u_7, u_9, u_{10} \rangle] = false$;

      $tri[\langle 1,2,3 \rangle, \langle u_7, u_9, u_4 \rangle] = true$;

      or $btri[\langle 1,2,3 \rangle, \langle u_7, u_9 \rangle] = \{u_4\}$;

$u_4.S = \{\langle 1,2; u_7, u_9\rangle\}$;

$\langle u_7, u_9\rangle.C[3] \coloneqq 1$;

2) for $(v_2, v_3) \in E(Q)$ and $k = 1$:

   a. Note that in last step u_5 has been removed from R_2', so that $\langle u_5, u_4\rangle$ must also be removed from R_{23}.

   b. for $\langle u_9, u_{10}\rangle$:

   $tri[\langle 2,3,1\rangle, \langle u_9, u_{10}, u_8\rangle] = true$;

   $tri[\langle 2,3,1; u_9, u_{10}, u_7\rangle] = false$;

   or $btri[\langle 2,3,1; u_9, u_{10}\rangle] = \{u_8\}$;

   $u_8.S = \{\langle 2,3; u_9, u_{10}\rangle\}$;

   $\langle u_9, u_{10}\rangle.C[1] \coloneqq 1$;

   c. for $\langle u_9, u_4\rangle$:

   $tri[\langle 2,3,1\rangle, \langle u_9, u_4, u_8\rangle] = false$;

   $tri[\langle 2,3,1\rangle, \langle u_9, u_4, u_7\rangle] = true$;

   or $tri[\langle 2,3,1\rangle, \langle u_9, u_4\rangle] = \{u_7\}$;

   $u_7.S = \{\langle 2,3; u_9, u_4\rangle\}$;

   $\langle u_9, u_4\rangle.C[1] \coloneqq 1$;

3) for $(v_3, v_1) \in E(Q)$ and $k = 2$:

   a. for $\langle u_{10}, u_8\rangle$:

   $tri[\langle 3,1,2\rangle, \langle u_{10}, u_8, u_9\rangle] = true$;

   or $btri[\langle 3,1,2\rangle, \langle u_{10}, u_8\rangle] = \{u_9\}$;

   $u_9.S = \{\langle 3,1; u_{10}, u_8\rangle\}$;

   $\langle u_{10}, u_8\rangle.C[2] \coloneqq 1$;

   b. for $\langle u_4, u_7\rangle$:

   $tri[\langle 3,1,2\rangle, \langle u_4, u_7, u_9\rangle] = true$;

   or $btri[\langle 3,1,2\rangle, \langle u_4, u_7\rangle] = \{u_9\}$;

$u_9.S = u_9.S \cup \{\langle 3, 1; u_4, u_7\rangle\} = \{\langle 3, 1; u_{10}, u_8\rangle, \langle 3, 1; u_4, u_7\rangle\};$

$\langle u_4, u_7\rangle.C[2] := 1;$

In the second phase, we first pop out the only one item $\langle 2, u_5\rangle$ from *STACK*. Since $u_5.S$ is *empty*, the second phase terminates immediately. The final results can be stored as a graph shown in Figure 6, in which all the redundant elements are removed from R'_1, R'_2 and R'_3 as well as R'_{12}, R'_{23}, and R'_{31}. By exploring a path, we can get an answer to our query.

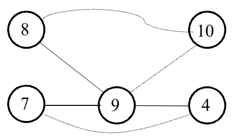


Figure 6: The result after *DomainFiltering*( ) and *RelationFiltering*( ).

6.2.3 Computational Complexity

The time complexity of the algorithm can be easily analyzed. We need only to count the number of times lines 21-26 are executed, which is bounded by $O(n^3D'^3)$, where $n = V(Q)$ and $D' = max\{|R'_1|, ..., |R'_n|\}$.

To know the space overhead, we first analyze the space used by all counters associated with the pairs in all R_{ij}'s: $\langle u, u'\rangle.\boldsymbol{C}$, which is bounded by

$$\sum_{(v_i, v_j)\in Q} |R'_i||R'_j||C| \leq O(n^3D'^2)$$

For storing support lists, we need even a larger space:

$$\sum_{v_k\in Q} |R'_k| \sum_{(v_i, v_j)\in Q} (|R'_i||R'_j|) \leq O(n^3D'^3)$$

Finally, concerning the correctness of the algorithm, we have the following proposition.

Proposition 2 Let Q be a directed query graph containing n vertices $v_1, \ldots, v_n$. Let $R'_1, \ldots, R'_n$ be n reduced lists constructed by executing *DomainFiltering*( ). When $STACK$ becomes *empty*, any pair $\langle u, u' \rangle$ in a remaining $R'_{ij} (i \neq j)$ must have at least a support from any other list $R'_k (k \neq i, j)$.

Proof. The proposition can be proven in a way similar to Proposition 1. □

CHAPTER 7 NATURAL JOIN AND JOIN ORDER SELECTION

After running the *DomainFiltering*( ) and *RelationFiltering*( ) algorithms when receiving a query Q, we obtain m reduced relations R''_{ij}, in which all redundant items are removed. However, this is not the final answer to a pattern match query by Definition 3.5; and a series of join operations, called classical *Natural Joins*, need to be performed.

7.1 Natural join

Given a query Q, we visit each edge $e = (v_i, v_j) \in E(Q)$ in each step according to a specific join order (we will discuses this in the next subsection). During the traversal of Q, a subgraph Q' (called status) induced by all visited edges in Q is maintained. Initially, Q' is set to be NULL. Then, in each step, a new edge $e = (v_i, v_j)$ is added to Q', $Q' := Q' \cup (v_i, v_j)$. This process will continue until $Q' == Q$. For such a new edge $e = (v_i, v_j)$, if its vertex v_i or v_j not have been visited before, it is called a forward edge; otherwise, it is called a backward edge. The matching results can be recorded in a table $MR(Q')$, in which each result is denoted as $\langle v_i, \dots, v_j; u_i, \dots, u_j \rangle$, where $u_i \in R_i, u_j \in R_j$ and$V(Q') = \{v_i, \dots, v_j\}$and $i, j \leq n$. Obviously, the forward edge and backward edge should be handle in different ways.

- *Forward edge processing*. Let $e = (v_i, v_k)$ be a forward edge. Assume that $v_i (i < k)$ is a vertex not having been visited while $v_k (i < k)$ have been visited. An equal-join $MR(Q') \bowtie R''_{ik}$ needs to be performed. The matching results is then augmented from $MR(Q') =$

$\langle v_i, \dots, v_j; u_i, \dots, u_j \rangle$ to $MR(Q' + e) = \langle v_i, \dots, v_j, v_k; u_i, \dots, u_j, u_k \rangle$, where $u_k \in R_k$ and $k \leq n$.

- *Backward edge*. Let $e = (v_i, v_k)$ be a backward edge. We will can scan the intermediate table $MR(Q')$ to filter out all those tuples that do not match pairs $(u_i, u_k) \in R_{ik}$. After filtering, we obtain $MR(Q' + e)$ from $MR(Q')$. Essentially, this is a selection operation based on the Definition 3.5.

It is easy to see that the time complexity of this equal join is $O(\prod_{ij} |R''_{ij}|)$. Considering R''_{ij} are effectively filtered relations from R_{ij}, the natural join on relations R''_{ij} should be much faster than on original relations R_{ij}.

7.2 Example

Example 5: Continue with Example 4. When applying the *Natural Join* to R''_{12}, R''_{23}, and R''_{31} shown in Figure 6, the process will work as follow 2 steps:

1) For $E(Q'_1) = \{(v_1, v_2)\}$, (v_1, v_2) is a new visited forward edge. The $MR(Q'_1)$ is:
   $\langle 1, 2; u_8, u_9 \rangle$
   $\langle 1, 2; u_7, u_9 \rangle$
2) For $E(Q'_2) = \{(v_1, v_2), (v_2, v_3)\}$, (v_2, v_3) is a new visited forward edge. The $MR(Q'_2) = MR(Q'_1) \bowtie R''_{23}$:
   $\langle 1, 2, 3; u_8, u_9, u_{10} \rangle$
   $\langle 1, 2, 3; u_8, u_9, u_4 \rangle$
   $\langle 1, 2, 3; u_7, u_9, u_{10} \rangle$
   $\langle 1, 2, 3; u_7, u_9, u_4 \rangle$

3) For $E(Q'_3) = \{(v_1, v_2), (v_2, v_3), (v_1, v_3)\}$, (v_1, v_3) is a backward edge.

The $MR(Q'_3) = MR(Q'_2) \bowtie R''_{13}$:

$\langle 1, 2, 3; u_8, u_9, u_{10} \rangle$

$\langle 1, 2, 3; u_7, u_9, u_4 \rangle$

Example 5 shows all the steps of a natural join. In the first step, (v_1, v_2) is a first encountered forward edge, and the matching result table MR is simply equal to R''_{12}. In the second step, (v_2, v_3) is a forward edge, and there are 4 intermediate records in MR by joining with R''_{23}. In the last step, (v_1, v_3) is a backward edge, and only 2 records are left after the MR being filtered by R''_{13}. Therefor, the final matching results of the example in Figure 5 are $\langle u_8, u_9, u_{10} \rangle$ and $\langle u_7, u_9, u_4 \rangle$. By comparing with the Natural Joins directly on Figure 5(a), which is neither filtered by the *DomainFiltering*( ) nor by the *RelationFiltering*( ), the number of intermediate records in MR is up to 6.

7.3 Join Order Selection

Before we do a natural join, the join order should be defined. A join order corresponds to a traversal order in query Q, which may somehow affect the performance of the Natural Joins. This is a well researched area and many strategies for finding a good nesting order have been proposed, which can be described as follow [39]:

- *Deterministic Algorithms.* This kind of algorithms, such as dynamic programming algorithm, constructs a solution step by step in a deterministic manner, either by applying a heuristic or by an exhaustive search.
- *Randomized Algorithms*. This kind of algorithms first define a set of movers. Each of the algorithms performs a random walk along the

edges according to certain rules, terminating as soon as no more applicable moves exist or a time limit is exceeded. The best order obtained so far is used as the join order.

- *Genetic Algorithms.* This kind of algorithms makes use of a randomized strategy very similar to the biological evolution in a search for good problem solutions. The basic idea is to start with a random population and generate offspring by random crossover and mutation. The "fittest" members of the population survive the subsequent selection; the next generation is based on these. The algorithm terminates as soon as there is no further improvement or after a predetermined number of generations. The fittest member of the last population is the solution.
- *Hybrid algorithms.* This kind of algorithms combine the strategies of pure deterministic and pure randomized algorithms: solutions obtained by deterministic algorithms are used as starting points for randomized algorithms or as initial population members for genetic algorithms.

However, all the above strategies of speeding up the join operation are specifically designed for relational database, and not quite efficient and suitable for graph pattern matching queries since our patterns mostly are graphs with many edges. For instance, a dynamic programming algorithm, as a exhaustive search of *Deterministic Algorithms*, works well only for few relations, and can be prohibitively expensive for more than five or six relations.

In our implementation, We adopt a simple yet efficient greedy solution proposed in [24] in order to find a good join order. The heuristic rule of this greedy solution is simple: given a status Q', if there is a backward edge e attached to Q', the next status is $Q' = Q' \cup e$; otherwise, a forward edge e is chosen. In other words, the backward edges have higher priority than backward edges during each traversal step. In this thesis, all the experiments are under such a greedy join order selection except in section 8.8, where we test the impact of different join order selections.

CHAPTER 8 EXPERIMENT

In this chapter, we evaluate our methods and some existing approaches over a variety of real and synthetic data graphs. We divided the experiments in five pars. In the first part, we use all real data graphs (See Table 3) to compare all methods against each other and to study how well our method perform on real data graphs. In the second part, we vary the number of edges and the value of γ for two types of synthetic data graphs, ER and SF, respectively, in order to compare all methods against each other and study how well our method perform on synthetic data graphs. In the last three parts, we study in depth of our method. In the third part for synthetic data graphs, we study the impact of label set size since the less labels imply bigger average size of list R_i. In the forth part for real data graphs, we fix the value of δ but vary the edge number of query Q in order to see its effect. The last part is also for real data graphs, by which we fix both the δ and $|Q(E)|$, but vary the shape of query Q in order to study how it affects the performance.

8.1 Settings

All methods are implemented in C++, compiled by Visual Studio 2013 with optimization of O2 (maximize speed). All of our experiments are performed on a desktop computer with 64-bit Windows 10 operating system, Intel I7-7700 3.6GHz CPU and 14G RAM. Since the physical memory is limited to 14G, the frequent swap between memory and disk will affect the performance if all needed data cannot be cashed into memory.

8.2 Tested Methods

We have tested altogether three methods listed below. Each of them includes two main steps, Relation Construction (RC for short) and Matching Results Construction (MC for short):

- Extend Reachability Join (ER-join for short) [25].
  RC: 2-hop labeling (2HL for short) [16][17][38].
  MC: Classical Natural Join (N-join for short) [34].
- Multi Distance-based Join (MD-join for short) [24].
  RC: LLR-Embedding (LE for short) [33] and 2-hop labeling (2HL for short) [16][17].
  MC: Classical Natural Join (N-join for short) [34].
- Our Method (discussed in this work).
  RC: Δ-Transitive Closure of data graph G (G^{Δ} for short) (ours).
  MC: DomainFiltering (DF for short) (ours), Relation Filtering (RF for short) (ours), Classical Natural Join (N-join for short) [34] and Join Order Selection (JOS for short) [24][39].

	index (offline)			query (online)	
methods	algorithms	time	space	algorithms	time
ER-join	2-hop labeling	$O(N^4)$	$O(N\sqrt{M})$	ER-join (RC)	$O(\sum_{ij}\sqrt{M}\|R_i\|\|R_j\|)$
				Natural Join (MC)	$O(\prod_{ij}\|R_{ij}\|)$
MD-join	2-hop labeling	$O(N^4)$	$O(N\sqrt{M})$	D-join (RC)	$O(\sum_{ij}\sqrt{M}\|R_i\|\|R_j\|)$
	LLE-Embedding	$O(N^2logN)$	$O(Nlog^2N)$	Natural Join (MC)	$O(\prod_{ij}\|R_{ij}\|)$
ours	G^{Δ} by Dijkstra	$O(Nd^{\Delta}logN$	$O(Nd^{\Delta})$	G^{Δ} (RC)	$O(\sum_{ij}\|R_{ij}\|)$
				DF (MC)	$O(mD^2)$
				RF (MC)	$O(n^3D'^3)$
				Natural Join (MC)	$O(\prod_{ij}\|R_{ij}\|)$

Table 2: The summary of all tested methods.

In Table 2, we compare the above methods in terms of theoretical time and space complexity for both index (offline) and query(online) stages. In the query stages, each of the above methods has two steps, relation construction (RC) and matching results construction (MC). Each of them will be extensively tested.

8.3 Tested Data Graphs

In the experiments, we use both real and synthetic data graphs. Table 3 and Table 4 summarize all the important parameters of the tested graphs. In both tables, for each graph, we show:

- whether a graph is directed or undirected;
- number of vertices;
- number of edges;
- whether a graph is labeled. (If a graph is not labeled, we will randomly assign its vertices the labels out of an alphabet Σ.)
- the size of Σ; (If a graph is not labeled, we will randomly assign its vertices the labels out of an alphabet Σ.)
- whether a graph is weighted; (If a graph is not weighted, we will randomly assign its edges the weight from 1 to 1000.)
- average degree of vertex calculated by $\frac{|E(G)|}{|V(G)|}$ for directed graphs and $\frac{2|E(G)|}{|V(G)|}$ for undirected graphs;
- the density of graphs calculated by $\frac{|E(G)|}{|V(G)|^2}$ for directed graphs and $\frac{2|E(G)|}{|V(G)|^2}$ for undirected graphs; and
- the average number of vertices with a same label calculated by $\frac{|V(G)|}{|\Sigma|}$.

Table 3 lists all the details of real data graphs. The Yeast, as a protein-to-protein interaction network in budding yeast, comes from vlado (http://vlado.fmf.uni-lj.si/pub/networks/datas/); the Citeseer is come from konect (http://konect.uni-koblenz.de/networks/citeseer); and all the other real graphs are come from SNAP (http://snap.stanford.edu/data/index.html). All the data graphs G are sorted by the number of vertices, indicating scales.

data graphs	directed	$N = \|V(G)\|$	$M = \|E(G)\|$	labeled	$\|\Sigma\|$	weighted	avg. degree	density	avg. $\|R(l)\|$
yeast	no	2,361	7,182	yes	13	no	6.1	2.58E-03	196.8
wikiVote	yes	7,115	103,689	no	100	no	14.6	2.05E-03	71.2
citeHepph	yes	34,546	421,578	yes	124	1-1000	12.2	5.33E-04	278.6
webStanford	yes	281,903	2,312,497	no	100	no	8.2	3.53E-04	2,919.0
comDBLP	no	317,080	1,049,866	no	500	no	6.6	2.67E-04	23.5
webNotreDame	yes	325,729	1,497,134	no	100	1-1000	4.5	2.91E-05	3,257.3
citeseer	yes	384,413	1,751,463	no	500	no	4.6	2.09E-05	768.8
webBerkStan	yes	685,230	7,600,595	no	500	no	11.1	1.41E-05	3,426.2
webGoogle	yes	875,713	5,105,039	no	500	1-1000	5.8	1.19E-05	1751.4
roadNetPA	no	1,088,092	1,541,898	no	50	no	2.8	1.62E-05	21761.8
roadNetTX	no	1,379,917	1,921,660	no	20	1-1000	2.8	6.66E-06	68995.8
citePatterns	yes	3,774,768	16,518,948	no	100	1-1000	4.4	2.60E-06	37747.7

Table 3: The summary of real date graphs.

data graph	directed	$\|V(G)\|$	parameter	synthetic label	$\|\Sigma\|$	weighted	avg. degree	avg. $\|R(l)\|$
ER	no	100,000	$\|E(G)\| = [100k, 500k]$	yes	100 - 500	no	2-10	1000 - 200
SF	yes	100,000	$\gamma = [2.1, 2.9]$	yes	100 - 500	no		1000 - 200

Table 4: The summary of synthetic data graphs.

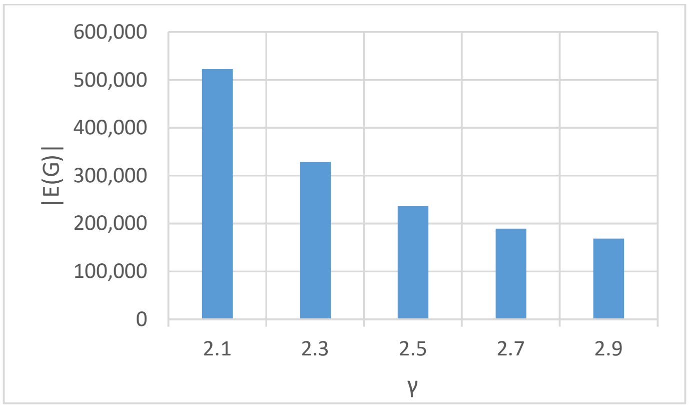


Figure 7: The edge number of SF graphs varying by γ.

Table 4 list two different synthetic data graphs:

1) *ER* (*Erdos Renyi Model*). This kind of graphs is a classical random graph model. It defines a random graph as N vertices connected by M

edges, chosen randomly from the $N(N-1)/2$ possible edges. We set $N = 100K$ and vary M from $100K$ to $500K$.

2) *SF* (*Scale-Free Model*). This kind of graphs is created by using the graph generator gengraph-win (http://fabien.viger.free.fr/liafa/genera tion/), by which the power-law distribution ($p(d) = \alpha d^{-\gamma}$) is followed when generating vertices and edges. Usually, $2 < \gamma < 3$ is chosen [40]. We vary γ from 2.1 to 2.9 and their corresponding number of edges are shown in Figure 7.

For both ER and SF graphs, we fix $|V(G)| = 100K$ but vary the number of labels, which are randomly assigned to all vertices, from 100 to 500 for special experiments below. For easy testing, we also define the ER as undirected graphs but the SF as directed graphs.

8.4 Tested Query Graphs

We have used 14 different pattern queries in Figure 8 for the tests, which are sorted by the number of edges.

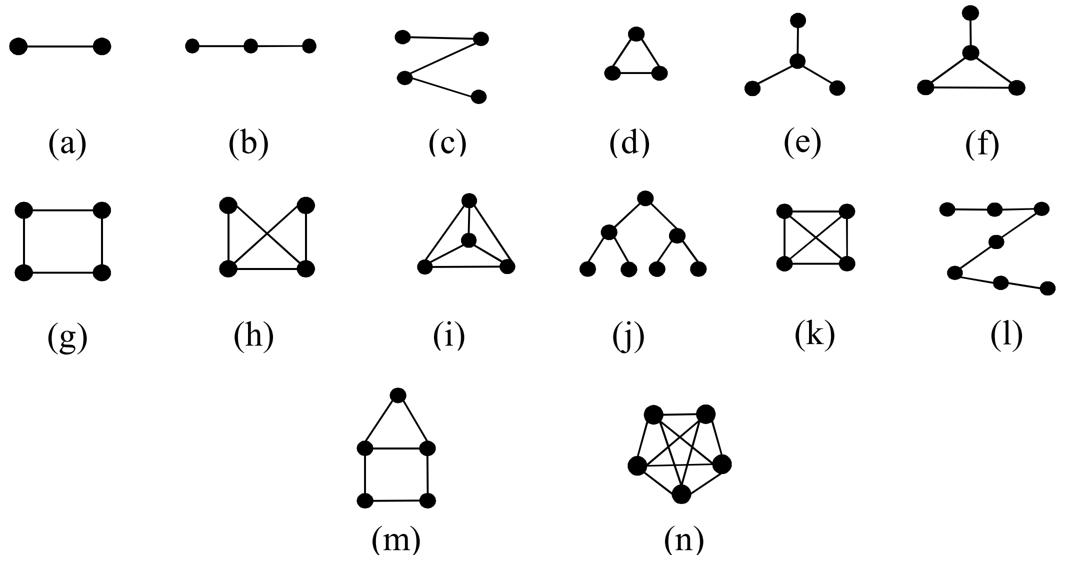


Figure 8: The examples of queries with different patterns.

Such pattern queries could be roughly divided into four categories according to their shapes: a line-pattern (LP for short) has a linear structure as shown in Figure 7(a), (b), (c), and (l); a tree-pattern (TP for short) has a tree structure as shown in Figure 4(e), and (j); a graph-patterns (GP for short) has a graph structure as shown in Figure 7(f), (g), (h), (i), and (m), which have approximately equal number of forward edges and backward edges; a complete-graph-pattern (CP for short) has a complete graph structure as shown in Figure 7(d), (k), (n), in which every pair of distinct vertices is connected by a unique edge.

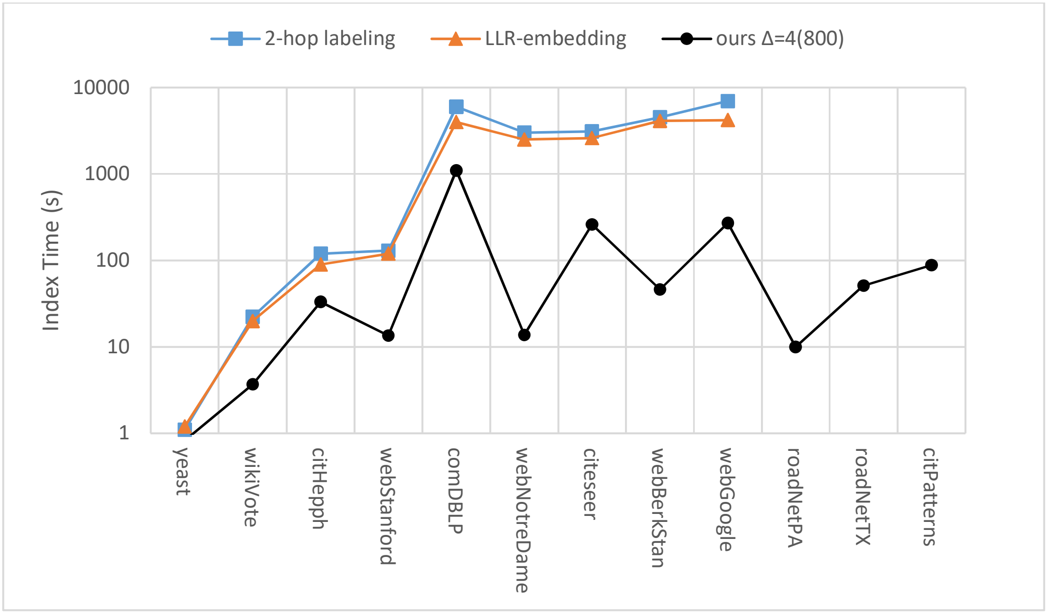
In the first to third experiments, we test a GP pattern shown in Figure 7(h), which has 5 query edges (3 forward edges and 2 backward edge) and 4 vertices. In the fourth experiment, we still use this graph pattern but varying the number of edges from 3 to 9. In the final experiment, we show how the four patterns affect our query performance.

8.5 First Experiment: Real Graphs Performance

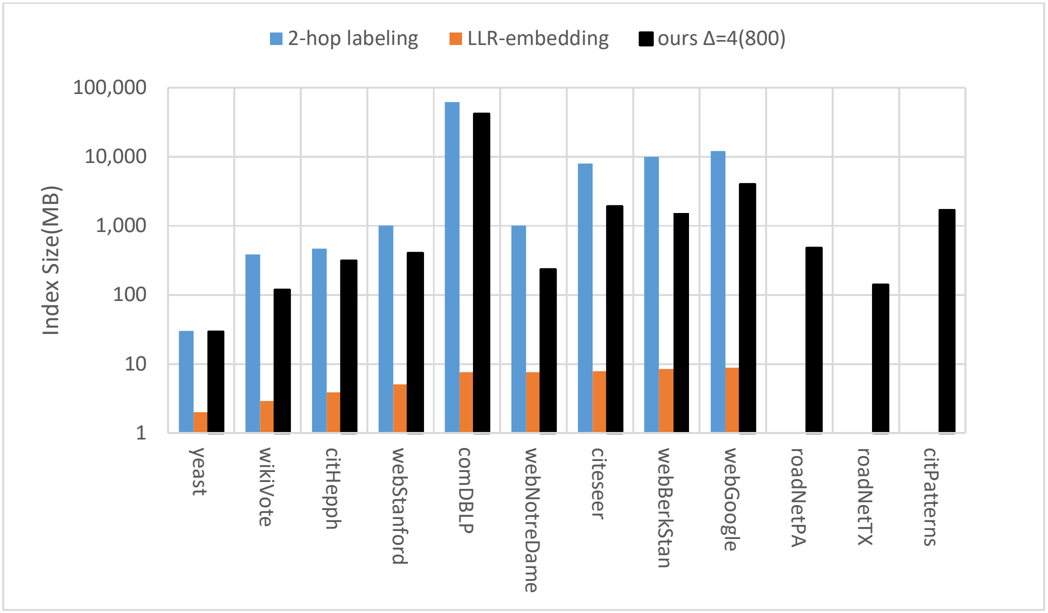
In our first experiment, we compare the performance of our method with the ER-join and MD-join on the real data graphs of Table 3. The major performance criteria here are the indexing time and sizes (offline), as well as the query time (online). We expect to see a tradeoff between indexing costs and query performance.

8.5.1 The Indexing Times and Sizes for Different Methods

Note that, as show in Table 2, for the index procedure the ER-join only use the 2-hop labeling and MD-join incudes both the 2-hop labeling and LLR-embedding method. The indexing time and sizes of the ER-join and MD-join are summarized in Figure 9. It clearly shows that both the ER-join and MD-join do not scale well on large graphs, since the 2-hop labeling both they use have large indexing time and size. In particular, MD-join needs some more indexing time and size for the LLR-Embedding. However, our method has much less running time and space usage for establishing indices by fixing $\Delta = 4$ for unweighted graphs and $\Delta = 800$ for weighted graphs.



(a) Index times

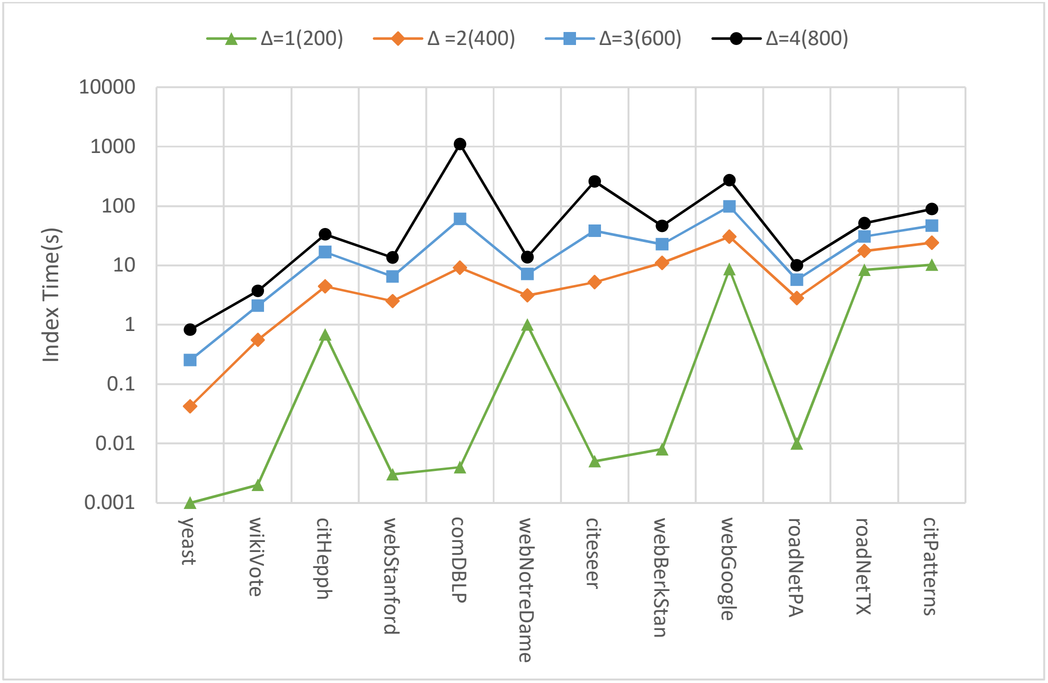


(b) Index sizes

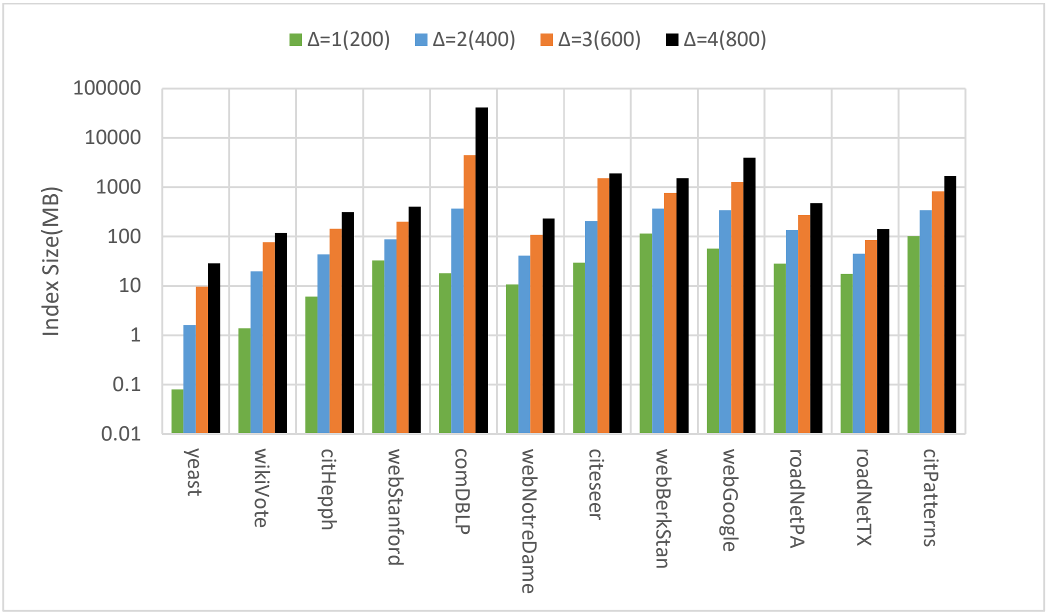
Figure 9: The indexing times (seconds) and sizes (MB) of real data graphs for different methods. Methods with indexing time over 2 hours are not showed.

8.5.2 The Index Times and Sizes for Our Method

The indexing times and sizes of our method are summarized in Figure 10, from which we can see that the index times and sizes increase dramatically with the value of Δ, especially for the data graphs with bigger degrees of vertices. However, the value of Δ is normally small for the pattern queries in practice. In our experiments, we only tested the case of $\Delta = \{1, 2, 3, 4\}$ for unweighted and $\Delta = \{200, 400, 600, 800\}$ for weighted graphs.



(a) Index time



(a) Index sizes

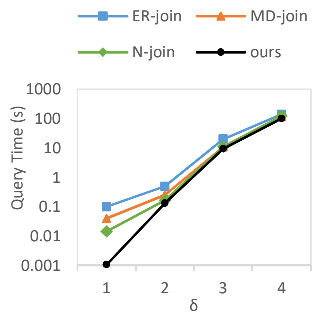
Figure 10: The indexing times (Seconds) and size (MB) of real data graphs for our method by varying Δ.

8.5.3 The Query Times for Different Methods

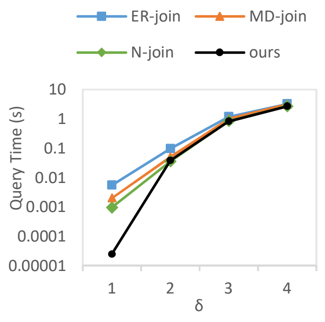
After indexing, we begin to test the query times of different methods. In this test, we fix the query Q shown in Figure 7(h), a five edges graph pattern, but vary the δ in order to see the effects of δ. We vary δ from 1 to 4 for unweighted graph and 200 to 800 for weighted graph since it is meaningless to choose too large δ for tests.

8.5.3.1 The query time for each real data graph

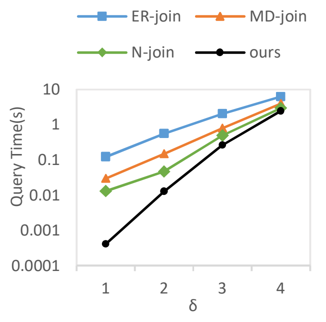
As shown in Table 2, we divide the whole running time into two parts: Relation Construction (RC) and Matching Results Construction (MC). Figure 11 summarizes the query time of different methods for each real data graph. We observe that both ER-join and MD-join have much more query time than N-join since only the N-join is in the RC part for both ER-join and MD-join. For the RC part we can see that MD-join is much faster than ER-join but still not very efficient, especially for larger graphs. However, our method is almost an order of magnitude faster than all of them. This is mainly because our method does not have running time for RC part since all relations have been constructed offline. Furthermore, our method can speed up the N-join process in varying ways, especially, for small δ. Such a huge improvement makes our method suitable for pattern matching queries on large data graphs.



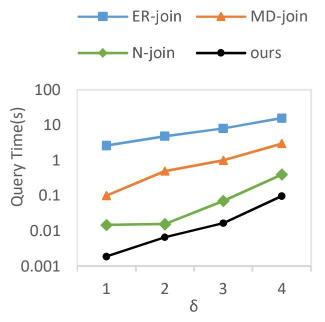
(a) yeast



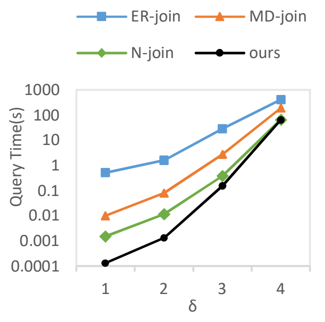
(b) wikiVote



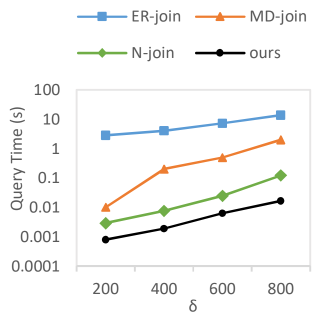
(c) citeHepph



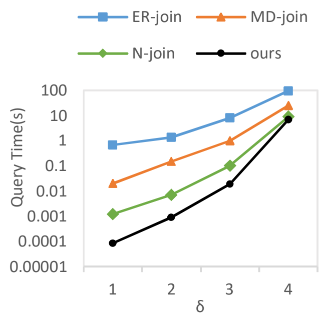
(e) webstanford



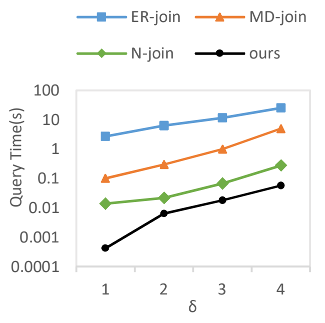
(f) comDBLP



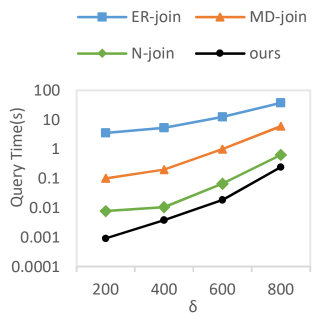
(g) webNotreDame



(e) citeseer



(f) webBerkStan



(g) webGoogle

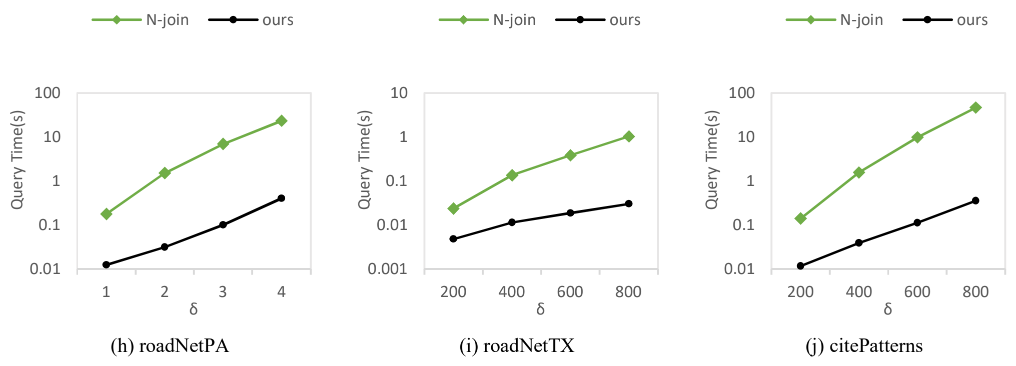
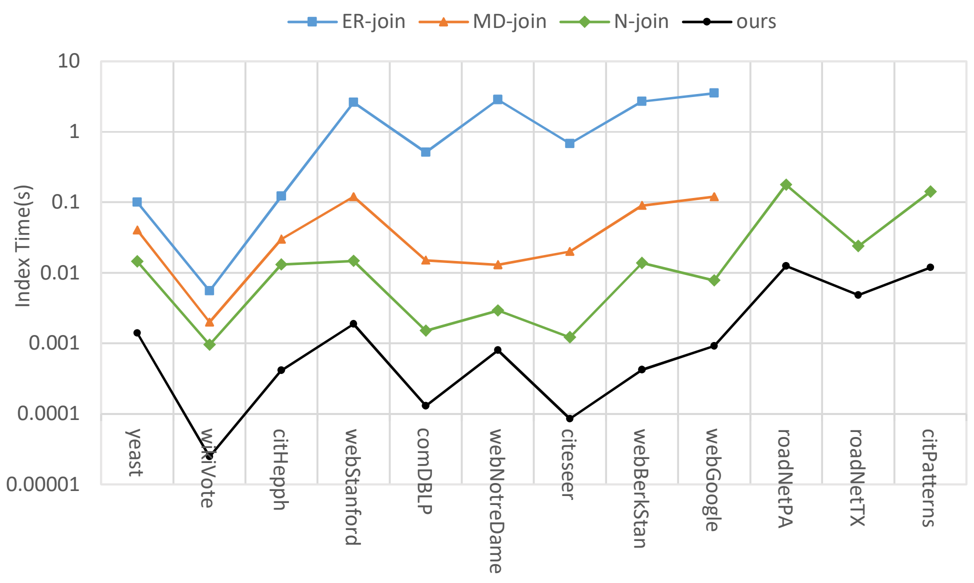


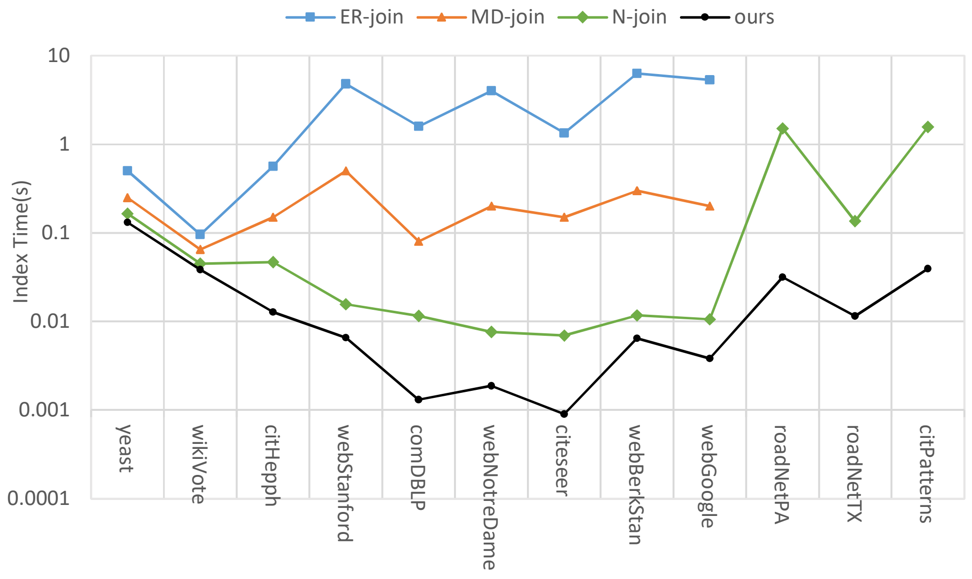
Figure 11: The query times (seconds) of each real data graph for different methods by fixing the query (as Fig. 7(h)) but varying δ.

8.5.3.2 The query times for each δ

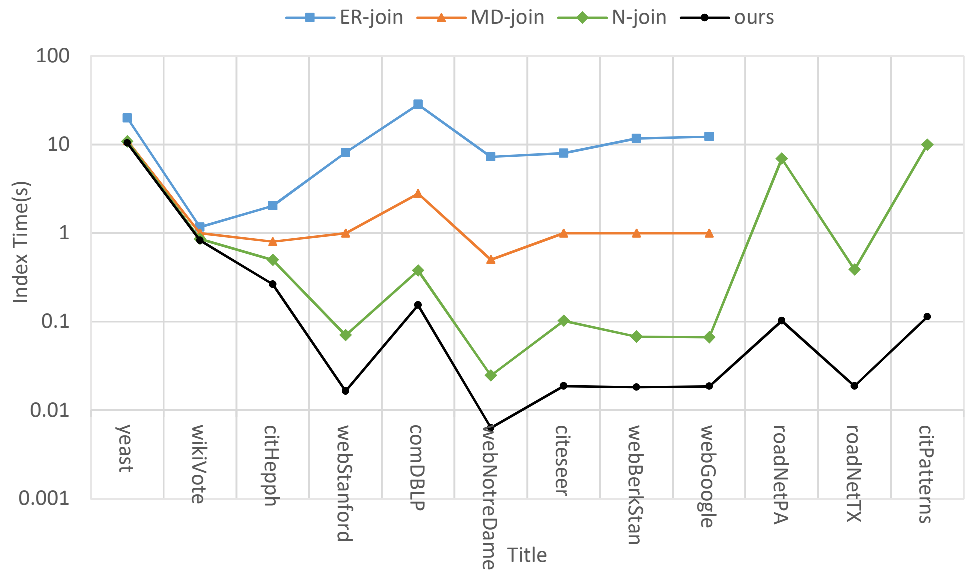
In Figure 12, we summarize the query times for each δ. It is easy to see that the MD-join improves more for the running time of MC than the ER-join when δ is small. For our method, it performs better when δ is small as well. However, when $\delta = 4(800)$ we can see that our method nearly has no speeding-up to N-join for some data graphs such as yeast, comDBLP and citeseer. This is explained by that a larger δ implies less limitations and less tuples can be filtered by our DF&RF algorithms.



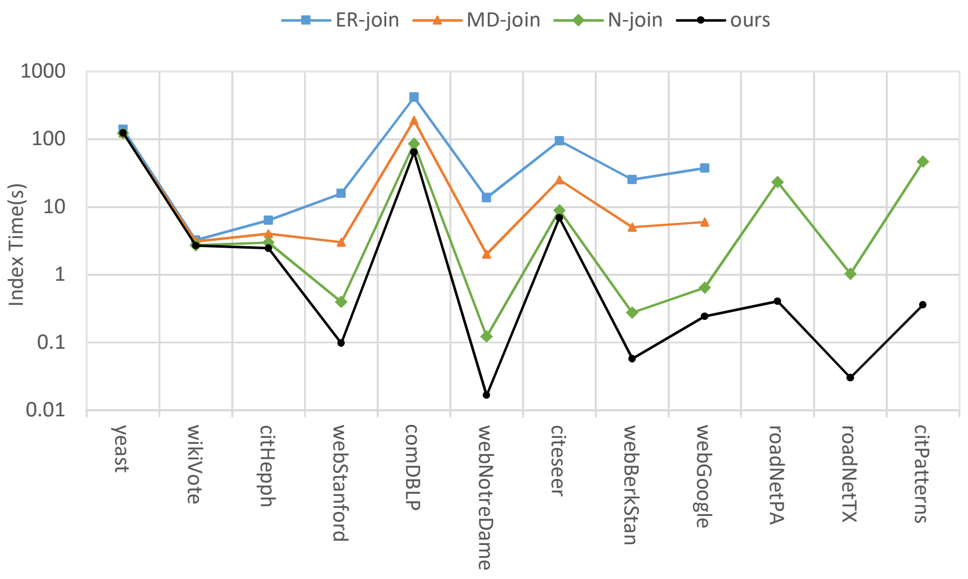
(a) δ = 1 (200)



(b) δ = 2 (400)



(c) δ = 3 (600)

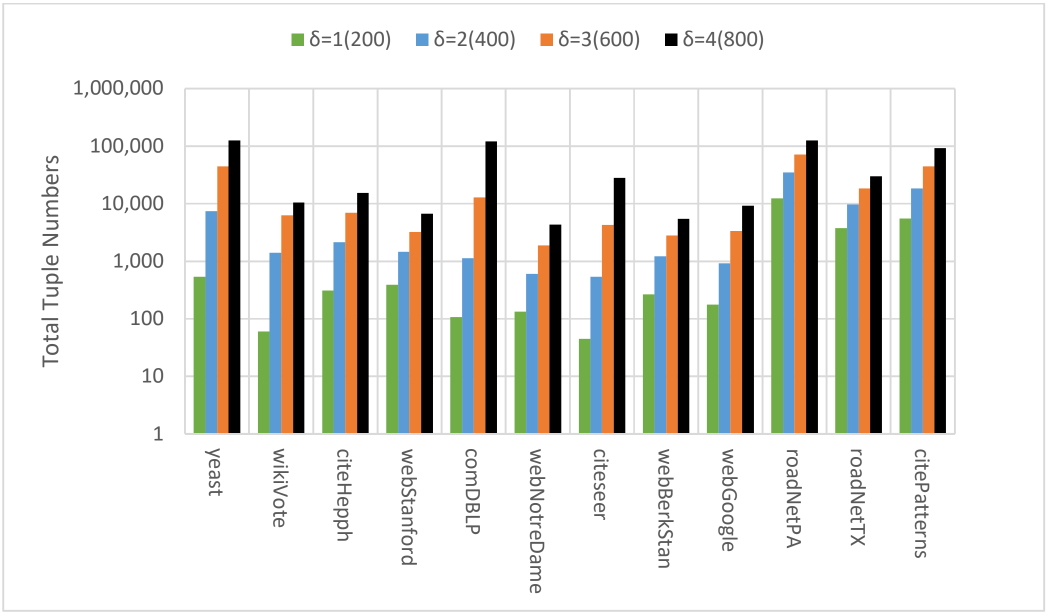


(d) δ = 4 (800)

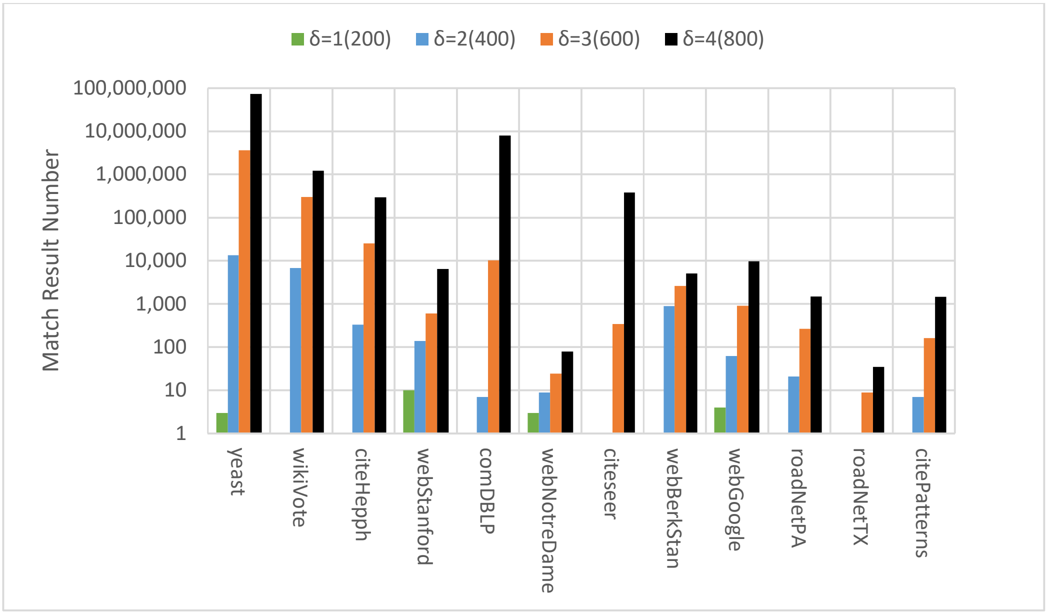
Figure 12: The query times (seconds) of each δ for different methods by fixing the query (as Fig. 7(h)).

8.5.3.3 The tuple numbers and matching result numbers

In Figure 13, we report tuple numbers and matching results numbers. We can see that as δ increase, the tuples numbers dramatically increase, as so does the numbers of matching results. This is why all methods need more query time for bigger δ. In the opposite, when δ is small, some of data graphs have no or quite few matching results, which explains why the queries are very fast since the N-join process is almost not necessary.



(a) Total tuple numbers



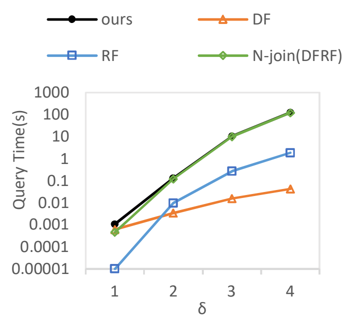
(b) Matching result numbers

Figure 13: The total tuple numbers and matching result numbers of all real data graphs (for all methods) by fixing the query (as Fig. 7(h)) but varying δ.

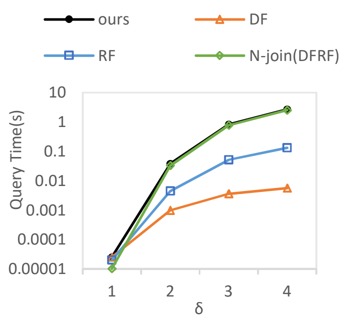
8.5.4 The Query Times for Our Method

8.5.4.1 The query times for each graphs

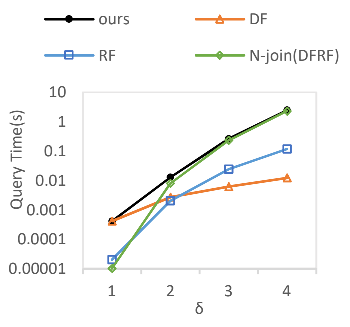
Figure 14 reports the query times of our method in detail. We can see that the running times of the DF and RF filtering algorithms are very fast, but nearly linearly increasing with δ. In general, if the many tuples are left after DF&RF, the query time of our method is mainly spent on final N-join after the DF and RF. However, by using efficient DF and RF filtering algorithm, our method perform much better than the classical N-join.



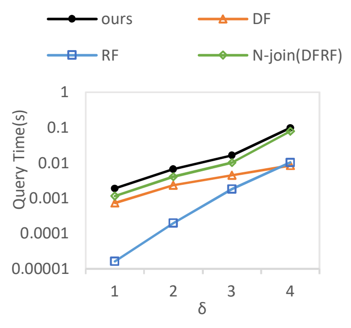
(a) yeast



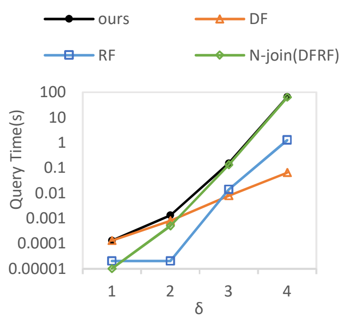
(b) wikiVote



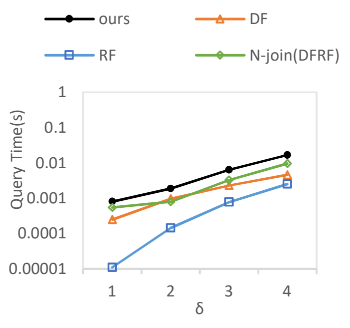
(c) citeHepph



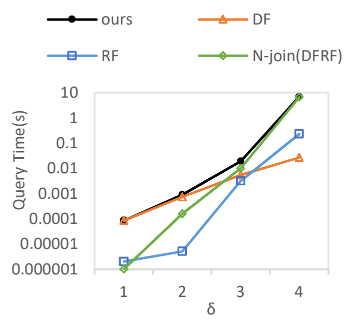
(e) webstanford



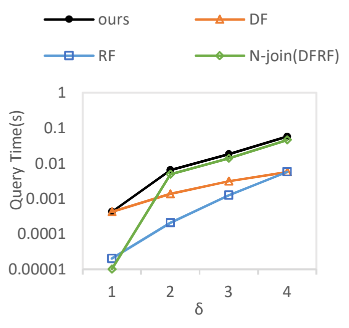
(f) comDBLP



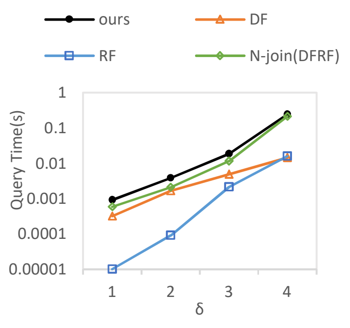
(g) webNotreDame



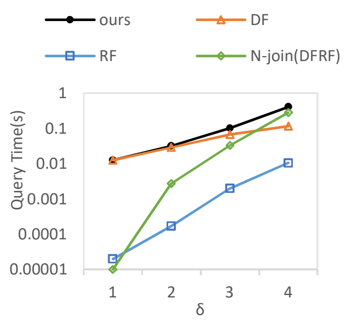
(e) citeseer



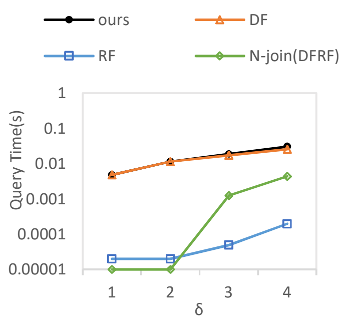
(f) webBerkStan



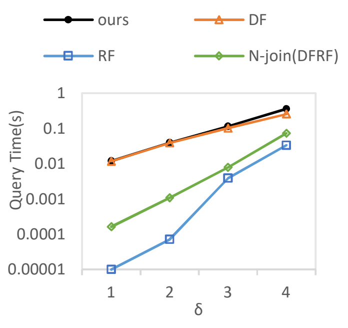
(g) webGoogle



(h) roadNetPA



(i) roadNetTX



(j) citePatterns

Figure 14: The detail query times (seconds) of real data graphs for our method by fixing the query (as Fig. 7(h)) but varying δ.

8.5.4.2 The tuple numbers filtered by DF&RF

Table 5 reports the tuples numbers of the real data graphs filtered by DF&RF algorithms for different δ. From this, it can be seen that our DF algorithm is very effective by removing most useless tuples from relations; and our RF algorithm can further remove the some tuples from relations. When δ is small a few tuples are left, which can greatly speed up the classical N-join process. Note that for some smaller data graphs but with higher density, like yeast and wikiVote graphs, our DF&RF algorithms cannot remove too many tuples since only a small proportion of tuples is useless.

graphs	δ = 1(200)			δ = 2(400)			δ = 3(600)			δ = 4(800)		
	total	after DF	after RF	total	after DF	after RF	total	after DF	after RF	total	after DF	after RF
yeast	540	9	9	7440	3220	2822	44442	22113	22049	125000	62470	62470
wikiVote	60	0	0	1407	1266	1145	6233	5794	5774	10461	9794	9794
citeHepph	308	0	0	2146	1386	392	7011	6347	4356	15417	14893	12954
webStanford	388	18	18	1466	127	127	3214	1092	632	6628	3819	2407
comDBLP	108	0	0	1140	14	14	12926	4832	3257	120904	59709	58617
webNotreDame	134	12	12	600	267	25	1886	1196	66	4296	3165	170
citeseer	45	0	0	536	5	5	4268	2626	504	28259	27217	22774
webBerkStan	264	0	0	1215	159	150	2782	693	529	5449	2090	1342
webGoogle	175	9	9	929	68	68	3359	1168	843	9161	5924	4397
roadNetPA	12380	0	0	35144	98	98	71500	1114	1113	125004	5488	5405
roadNetTX	3776	0	0	9624	0	0	18200	42	42	29860	131	131
citePatterns	5538	5	5	18378	40	30	44625	3515	446	92373	24361	2818

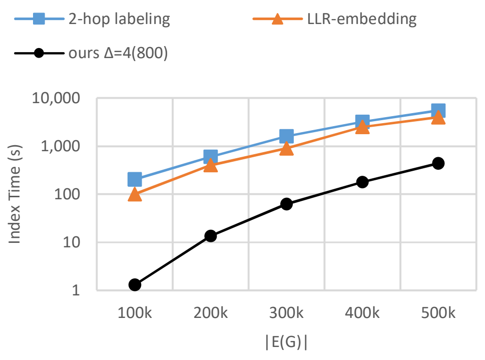
Table 5: The tuple numbers of real data graphs after DF&RF by fixing the query (as Fig. 7(h)) varying δ.

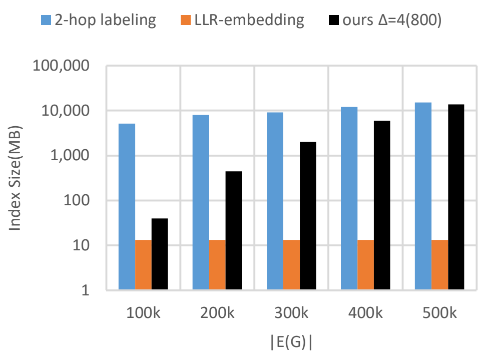
8.6 Second Experiment: Synthetic Graphs Performance

In our second experiment, we compare the performance of our method against the ER-join and the MD-join on the synthetic data graphs described in Table 4. We choose the vertex number $n = 100{,}000$ and fix the label size $|\Sigma| = 200$. Also, for the undirected ER graphs we vary the number of edges from 100,000 up to 500,000, thereby increasing the density and the average degree of vertices increase from 2 to 10; for the directed SF graphs we vary the parameter γ from 2.1 to 2.9 by which the number of edges decrease as in Figure 7, thereby decreasing the density. We expect to see a tradeoff between indexing costs and query performance.

8.6.1 The Index Times and Sizes for Different Methods

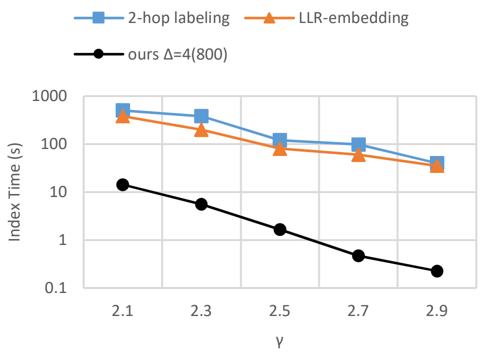
Figure 15 and 16 summarize the index times and sizes of the ER and SF graphs for the different methods, respectively. For establishing index, the results clearly shows that both ER-join (by using 2 hop labeling) and MD-join (by using 2-hop labeling and LLR embedding) do not scale well on ER graphs with more edges and SF graphs with smaller γ. However, our method has much less indexing time and sizes both in ER and SF graphs.

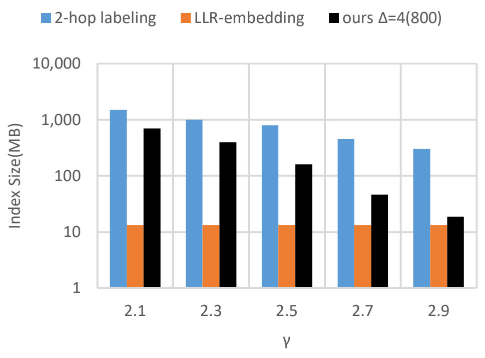




(a) Index time (b) Index sizes

Figure 15: The index times (seconds) and sizes (MB) of ER data graphs for different methods by fixing label size = 200 but varying edge number.



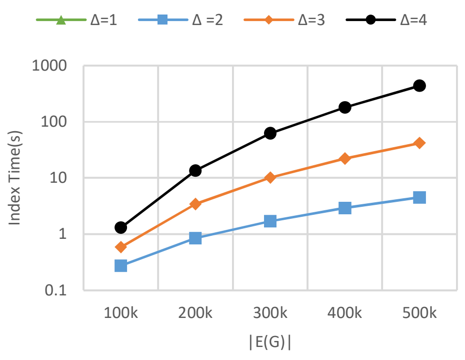


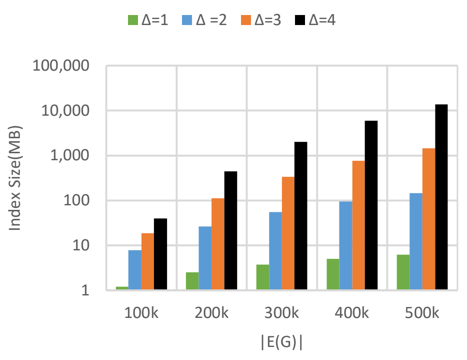
(a) Index time (b) Index sizes

Figure 16: The index times (seconds) and sizes (MB) of SF data graphs for different methods by fixing label size = 200 but varying γ.

8.6.2 The Index Times and Sizes for Our Method

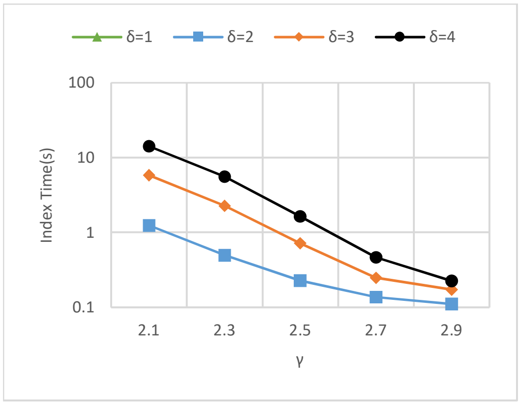
The index time and sizes of ER and SF graphs for our method are summarized in Figure 17 and 18, respectively. For ER graphs the index times and sizes increase dramatically with the δ and edge number; for SF graphs the index times and sizes decrease dramatically with γ. In other words, our method perform much better on sparse graphs to establish indices.

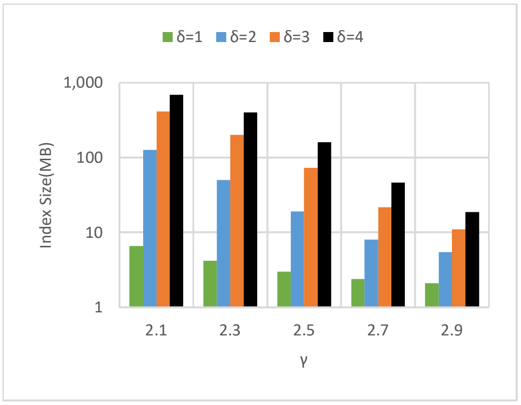




(a) index time (b) index size

Figure 17: Indexing times (seconds) and sizes (MB) of the ER graphs (label size = 200) for our method by varying edge number.





(a) index time (b) index size

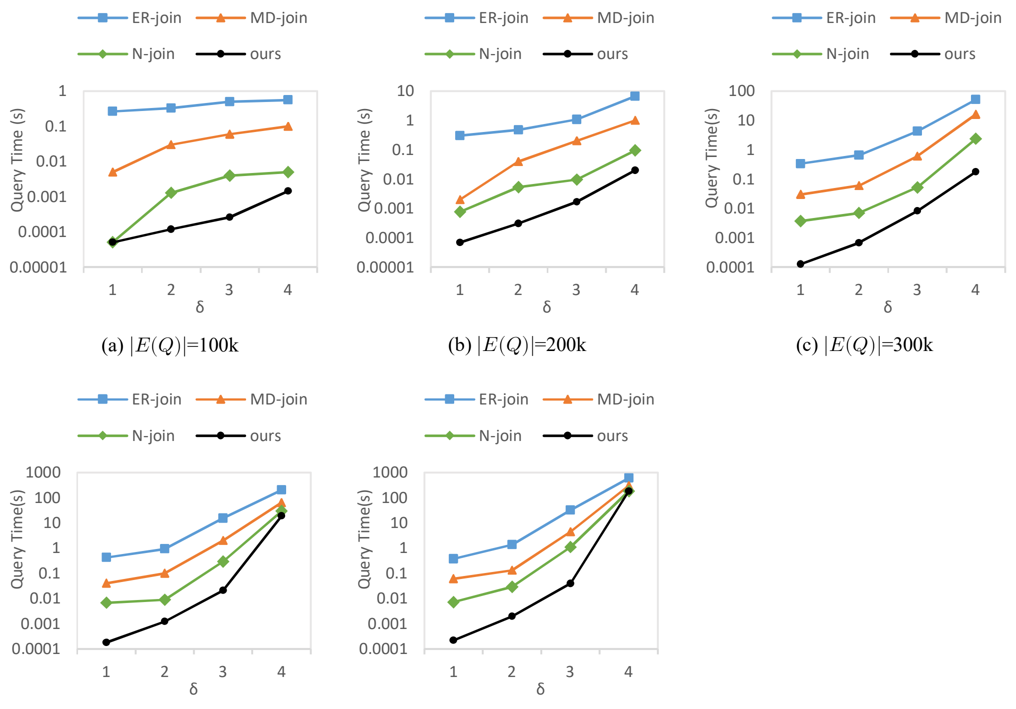
Figure 18: Indexing times (seconds) and sizes (MB) of the SF graphs (label size = 200) for our method by varying γ.

8.6.3 The Query Times for All Methods Comparing

After indexing, we begin to test the query times of different methods. In this test, we still fix the query Q shown in Figure 7(h) but vary δ from 1 to 4, since it is meaningless to choose too large δ for tests. In addition, for ER graphs we generate 5 graphs with different edge number ranging from $100K$ to $500K$; for SF graphs we generate 5 graphs with different value of γ ranging from 2.1 to 2.9.

8.6.3.1 The Query Times for Each Graph

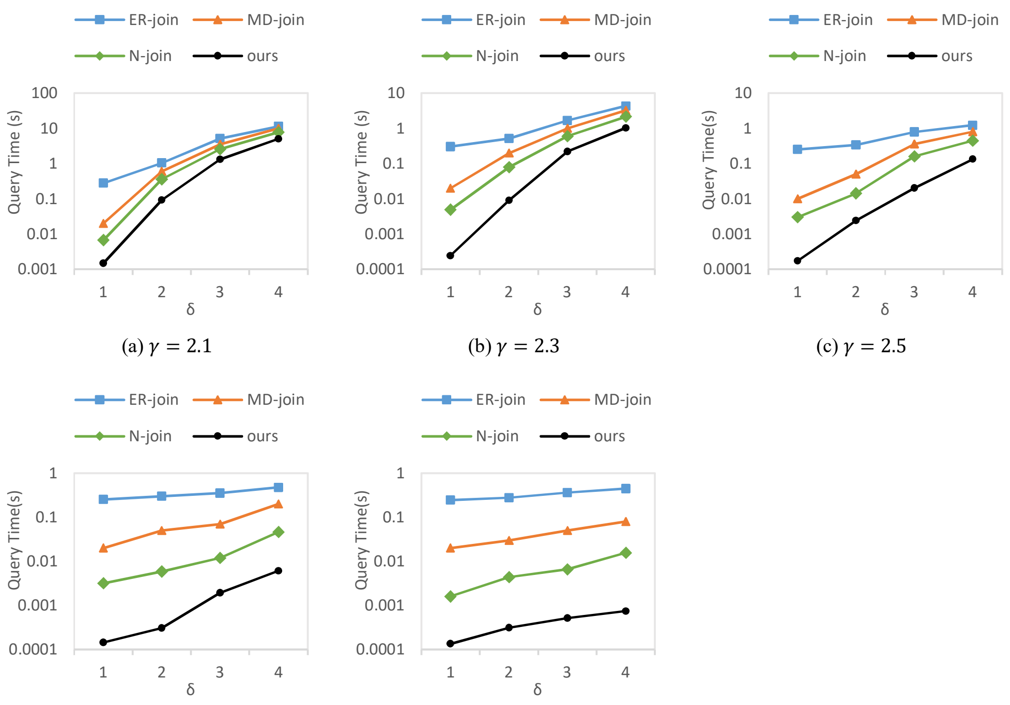
Figure 19 and 20 summarize the query times of ER and SF graphs for different methods, respectively. We can see that our method have much less query time than other methods and the query time increase dramatically with δ. However, for ER graphs with $500K$ edges, $\delta = 4$ and for SF graphs with $\gamma = 2.1, \delta = 4$, our method has less speedup for N-join. This is mainly because the proportion of redundant tuples tend to small in dense graphs.



(a) $|E(Q)|$=100k (b) $|E(Q)|$=200k (c) $|E(Q)|$=300k

(e) $|E(Q)|$=400k (f) $|E(Q)|$=500k

Figure 19: The query times of ER graphs (label size = 200) with different edge numbers for different methods by varying δ.



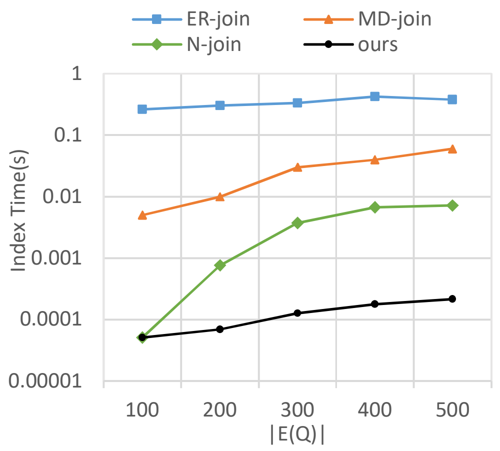
(a) $\gamma = 2.1$ (b) $\gamma = 2.3$ (c) $\gamma = 2.5$

(e) $\gamma = 2.7$ (f) $\gamma = 2.9$

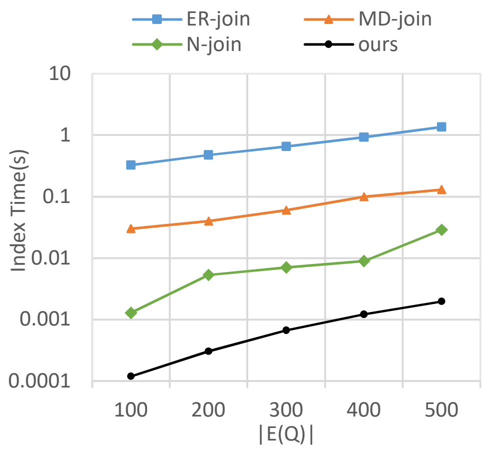
Figure 20: The query times of SF graphs (label size = 200) with different γ for different methods by varying δ.

8.6.3.2 The Query Times for Each δ

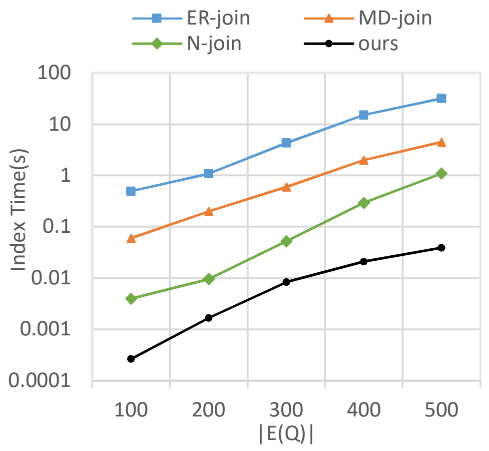
In Figure 21 and 22, we summarize the query times in ER and SF graphs with each δ for different methods by varying the number of edges and the value of γ, respectively. It is easy to see that both in ER and SF graphs our method has the much better performance than other methods especially when the δ is small.



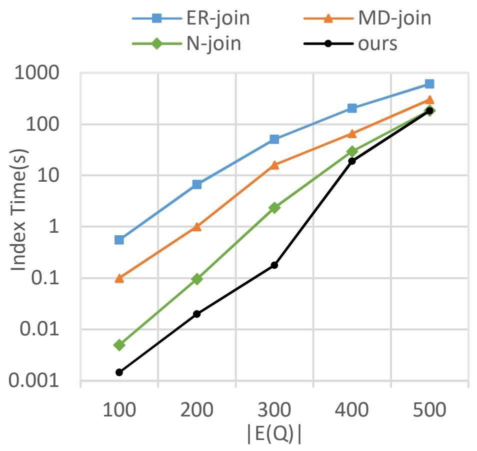
(a) δ =1



(b) δ = 2

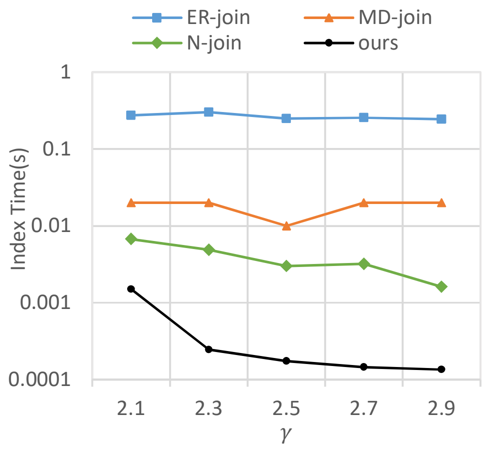


(a) δ =3

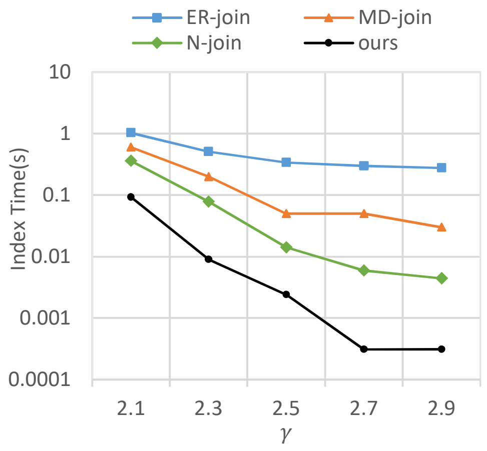


(b) δ = 4

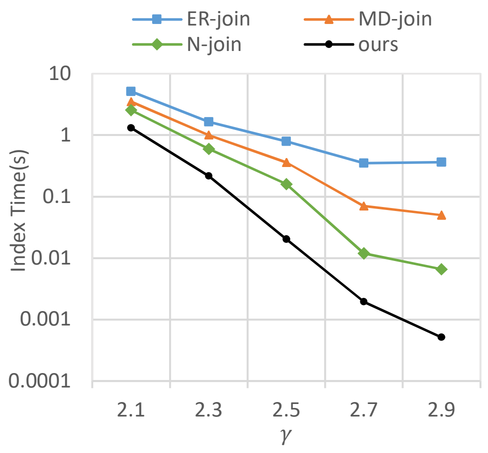
Figure 21: The query times of ER graphs (label size = 200) with each δ for different methods by varying edge numbers.



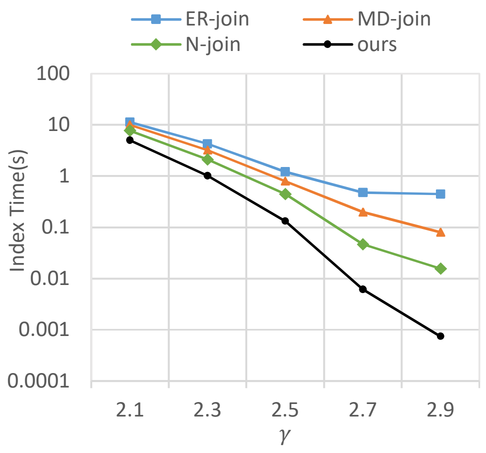
(a) δ =1



(b) δ = 2



(a) δ =3

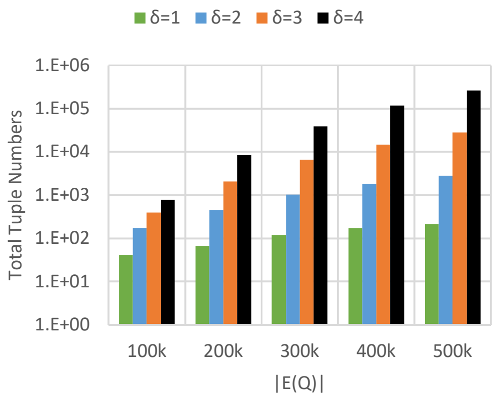


(b) δ = 4

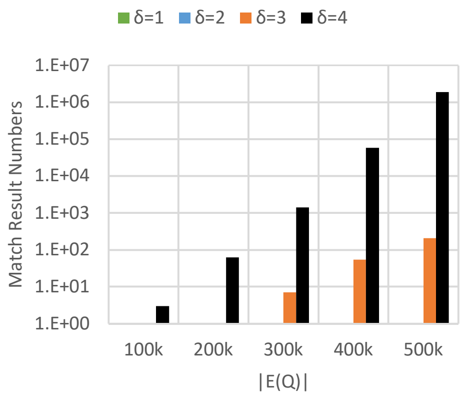
Figure 22: The query times of SF graphs (label size = 200) each δ for different methods by varying γ.

8.6.3.3 The Tuple Numbers and Matching Result Numbers

In Figure 23 and 24, we report the tuple numbers and match result numbers for ER and SF graphs, respectively. We can see that with the increasing of δ, the tuple numbers increase dramatically, which lead to much more match result numbers. This is why all methods need more query time for bigger δ. For both ER and SF graphs, the tuple numbers and matching result numbers increase sharply with the their edge numbers and γ, respectively, since their graph densities increase.

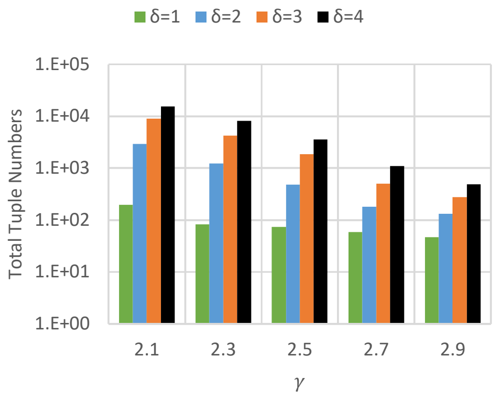


(a) Total tuple numbers

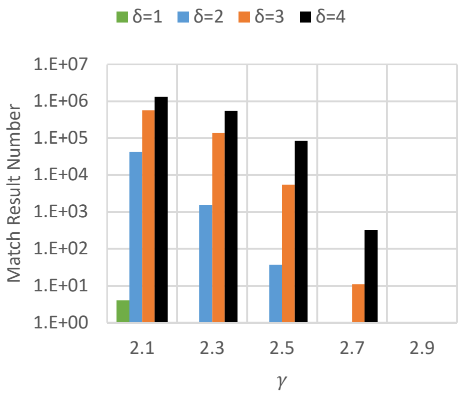


(b) Match result numbers

Figure 23: The total tuple numbers and match result numbers of ER graphs for our method by varying δ and the edge number.



(a) Total tuple numbers



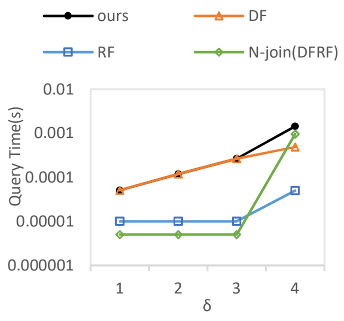
(b) Match result numbers

Figure 24: The total tuple numbers and match result numbers of SF graphs for our by varying δ and γ.

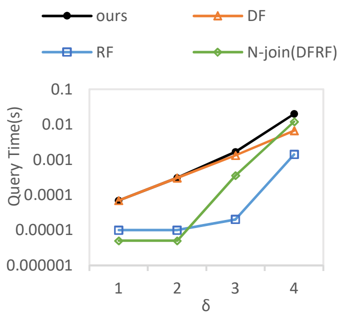
8.6.4 The Query Time for Our Method

8.6.4.1 The query time for each graph

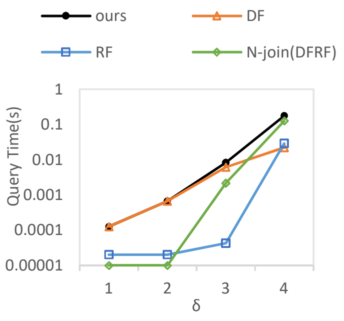
Figure 25 and 26 reports the query time of our method with more details for the ER and SF graphs, respectively. We can see that the running time of DF and RF filtering algorithms are so fast, nearly linearly increasing with δ. The query time of our method is mainly spent on final N-join after DF and RF filtering. By using efficient DF and RF filtering algorithm, our method perform much better than classical N-join.



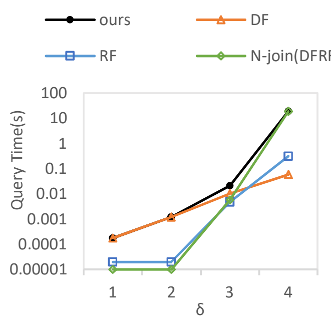
(a) $|E(Q)|$=100k



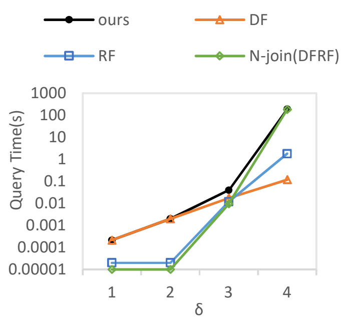
(b) $|E(Q)|$=200k



(c) $|E(Q)|$=300k

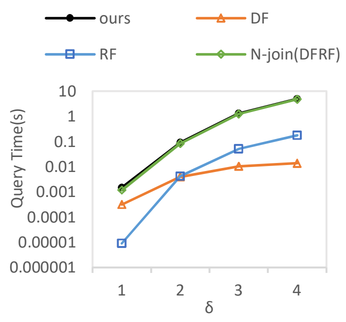


(d) $|E(Q)|$=400k

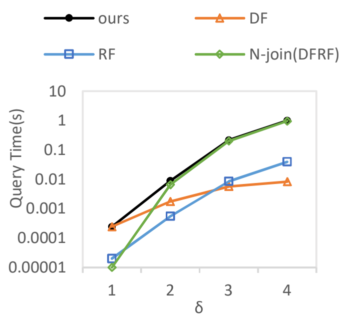


(e) $|E(Q)|$=500k

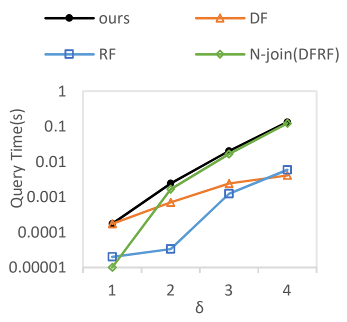
Figure 25: The detail query times of ER graphs for our method by varying δ.



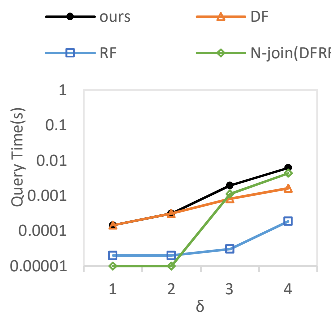
(a) $\gamma = 2.1$



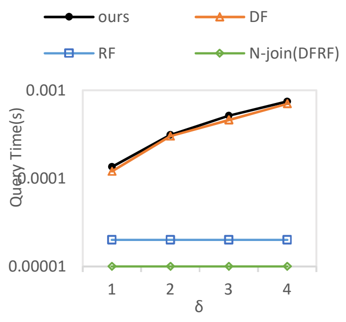
(b) $\gamma = 2.3$



(c) $\gamma = 2.5$



(d) $\gamma = 2.7$



(e) $\gamma = 2.9$

Figure 26: The detail query times of SF graphs for our method by varying δ.

8.6.4.2 The Tuple Numbers Filtered by DF&RF

Table 6 and 7 reports the tuple numbers of ER and SF graphs filtered by DF and RF for different δ, respectively. It is easy to see that our DF filtering algorithm is very effective, which can remove most of the redundant tuples. However, our RF filtering algorithm can further remove a few redundant tuples. When δ is small even no tuples is left, which means there is no matching results. With the increase of graph density, the proportions of redundant tuples will decrease. This well explain why our method has better performance in sparse graphs.

	δ = 1			δ = 2			δ = 3			δ = 4		
$\lvert E(Q)\rvert$	total	after DF	after RF	total	after DF	after RF	total	after DF	after RF	total	after DF	after RF
100K	42	0	0	174	0	0	394	0	0	790	9	9
200K	68	0	0	456	0	0	2072	5	5	8454	1461	183
300K	120	0	0	1030	0	0	6596	36	35	38812	19123	3733
400K	172	0	0	1816	0	0	14856	5954	211	115864	57932	44735
500K	216	0	0	2768	0	0	27886	13554	706	262850	131425	130665

Table 6: The tuple numbers of ER graphs with different edge numbers after DF&RF by varying δ.

	δ = 1			δ = 2			δ = 3			δ = 4		
γ	total	after DF	after RF	total	after DF	after RF	total	after DF	after RF	total	after DF	after RF
2.1	196	11	11	2934	1517	1471	8956	6226	6093	15471	12283	11893
2.3	83	0	0	1230	313	312	4274	2524	2474	8188	5390	5284
2.5	74	0	0	483	32	32	1845	619	607	3574	1990	1966
2.7	59	0	0	181	0	0	500	30	30	1103	124	124
2.9	47	0	0	132	0	0	278	0	0	491	0	0

Table 7: The tuple numbers of SF graphs with different γ after DF&RF by varying δ.

8.7 Third Experiment: Impact of Label Numbers

We next analyze the performance of our method while varying the number of labels using synthetic graphs. Here, all the numbers of vertices for tested synthetic graph are set to $|V(G)| = 100K$; we fix $|E(Q)| = 300K$ for ER graphs and $\gamma = 2.5$ for SF graphs; and for both ER and SF graphs we vary the number of labels from 100 to 500 which are assigned to vertices of data graph randomly; and, at the same time, we vary the value of δ from 1 to 4. Obviously, as the number of labels increase the average list size $|R(l)|$ will decrease. Our aim here is to better understand the impact of both the number of labels and δ on the query performance of our method. We expect that the more labels will lead to the less query time.

8.7.1 The Query Times for Each δ

Figure 27 and 28 show the query time of the ER and SF graphs for our method by fixing $|E(G)| = 300K$ and $\gamma = 2.5$, but varying label numbers and δ, respectively. We observe that in the ER graphs the query time decreases with the increase of labels for all δ value. However, in the SF graphs the query time generally decrease with the increase of labels for $\delta = 1, 2$, but some fluctuations occur for $\delta = 3, 4$. This is mainly because the ER kind of graphs are random graphs whose edges are randomly chosen from all possible edges, but the SF kind of graphs are not random graphs whose edges are generated according to power-low distribution (see Section 8.4). Such non-average distributions of edges for the SF graphs will lead to some different query performances compared with the ER graphs.

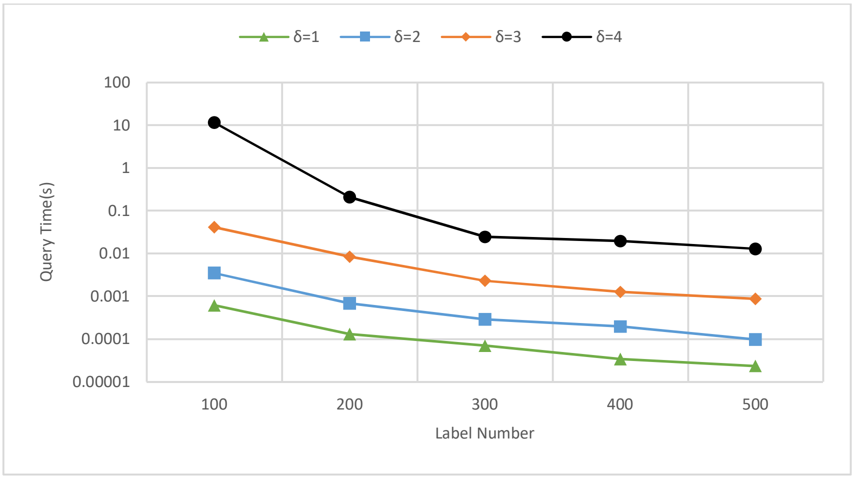


Figure 27 : The query times of ER graphs ($|E(G)| = 300K$) for our method by varying label numbers and δ.

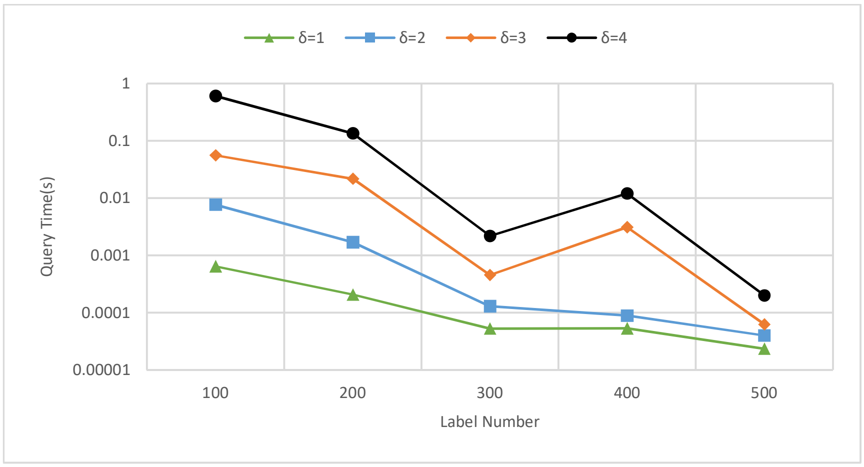
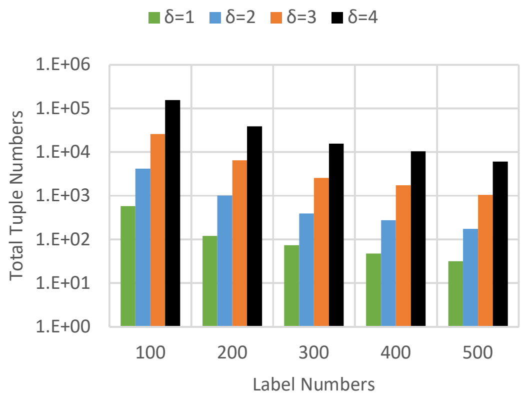


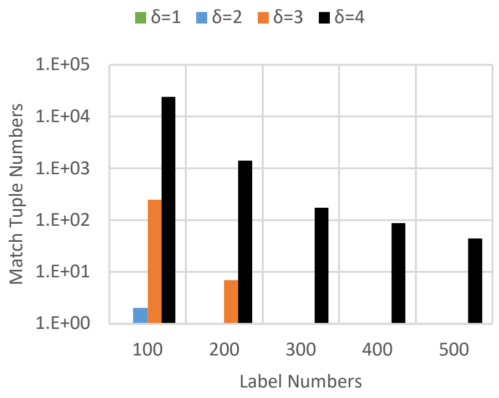
Figure 28 : The query times of SF graphs ($\gamma = 2.5$) for our method by varying label numbers and δ.

8.7.2 The Tuple Numbers and Matching Result Numbers

Figure 29 and 30 shows the tuple numbers and matching result numbers of the ER and SF graphs for our method, respectively. We observe that in the ER graphs the tuple number and matching number decrease with the increase of labels for all δ, but in the SF graphs this trend has some fluctuations. This well explains the query performances shown above.

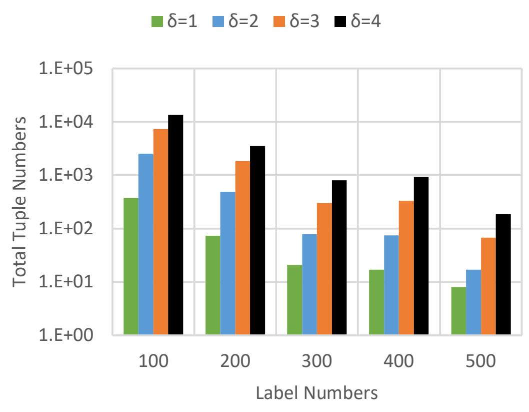


(a) Total tuple numbers

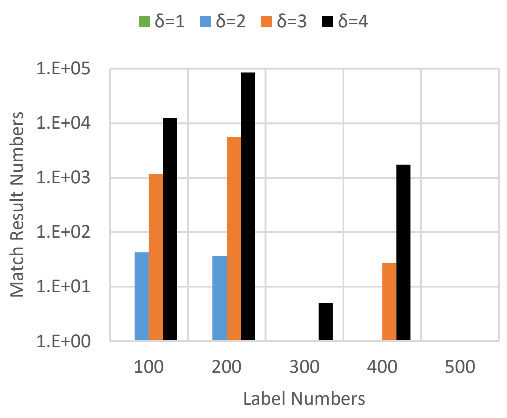


(b) Matching result numbers

Figure 29: The tuple numbers and matching result numbers of the ER graphs (edge number = 300) for our method by varying label numbers and δ.



(a) Total tuple numbers



(b) Matching result number

Figure 30: The tuple numbers and match result numbers of the SF graphs ($\gamma = 2.5$) for our method by varying label numbers and δ.

8.7.3 The Tuple Numbers Filtered by DF&RF

Table 8 and 9 shows the tuple numbers filtered by DF and RF algorithms, respectively. We observe that our DF and RF algorithms are powerful to remove redundant tuples especially when δ is small. This is why our method is efficient for evaluating pattern matching queries. In addition, with the increase of labels the tuple numbers decrease and the proportions of redundant tuples are decrease.

	δ = 1			δ = 2			δ = 3			δ = 4		
|Σ|	total	after DF	after RF	total	after DF	after RF	total	after DF	after RF	total	after DF	after RF
100	574	0	0	4228	10	10	26140	9045	768	156302	78017	35685
200	120	0	0	1030	0	0	6596	36	35	38812	19123	3733
300	74	0	0	394	0	0	2558	5	5	15694	7477	591
400	48	0	0	276	0	0	1758	0	0	10398	4711	310
500	32	0	0	176	0	0	1060	5	5	6052	2448	141

Table 8: The tuple numbers of ER graphs (edge number = 300) filtered by DF&RF by varying label number and δ.

	δ = 1			δ = 2			δ = 3			δ = 4		
|Σ|	total	after DF	after RF	total	after DF	after RF	total	after DF	after RF	total	after DF	after RF
100	373	0	0	2544	146	117	7379	2422	1808	13473	8465	6238
200	74	0	0	483	32	32	1845	619	607	3574	1990	1966
300	21	0	0	79	0	0	299	0	0	803	33	13
400	17	0	0	75	0	0	334	32	32	943	287	256

500	8	0	0	17	0	0	67	0	0	185	0	0

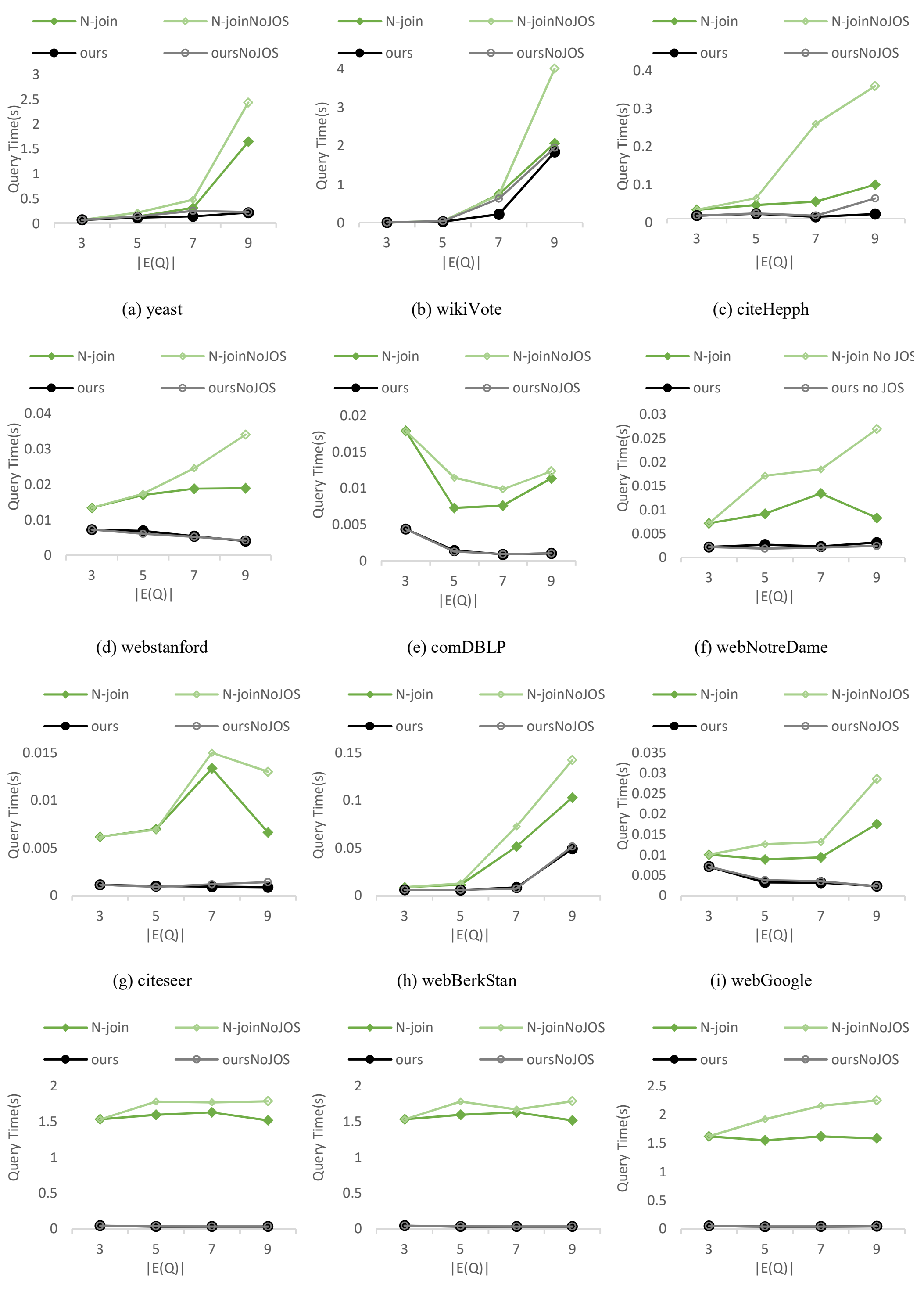
Table 9: The tuple numbers of SF graphs (edge number = 300) filtered by DF&RF by varying label number and δ.

8.8 Forth Experiment: Impact of Query Edge Numbers And Join Order Selections

In this subsection, we show the performance of our method as we vary the number of query edges, $|E(Q)|$. The query graphs are taken form those shown in Figure 7 and have approximately equal numbers of forward edges and backward edges. For such a query pattern, we vary the number of query edges from 3 to 9. We expect that as the query edges grow a higher speeding-up over the classical N-join will be obtained for our method.

8.8.1 The Query Time for Each Graph

Figure 31 shows that as the query edges grow the difference between our method and the classical N-join increase. The reason for this is that more edges in a query means a higher restriction and therefore more percentage of useless tuples can be filtered. As discussed in Section 7.2, the join order can also affect the performance of the N-join. Thus, Figure 31 also compares the N-join and our method with and without join order selections. We can see that a good join order can significantly speed up the classical N-join procedure. However, the good join order nearly does not have improvement for our method. This is because that our DF&RF algorithms remove almost all the useless tuples, which greatly reduces the searching space. Obviously, when $|Q(E)| \leq 3$ the join order selection has no effects to both the classical N-join and our method, since fewer query edges mean fewer joins and in this case the order of joins become less important.



(a) yeast (b) wikiVote (c) citeHepph

(d) webstanford (e) comDBLP (f) webNotreDame

(g) citeseer (h) webBerkStan (i) webGoogle

(j) roadNetPA (k) roadNetTX (l) citePatterns

Figure 31: The query times for each real data graphs by varying the number of query edges.

8.8.2 The Query Time for Each Query Edge Number

Figure 32 shows the query time of our method for all the real data graphs by varying the number of query edges. We can not observe a clear trend of query times for the different number of query edges. This is mainly because more query edges produce more tuples which increase the searching space, but at the same time, more query edges impose more restrictions so that the DF&RF algorithms are more powerful in reducing the relations participating the N-join.

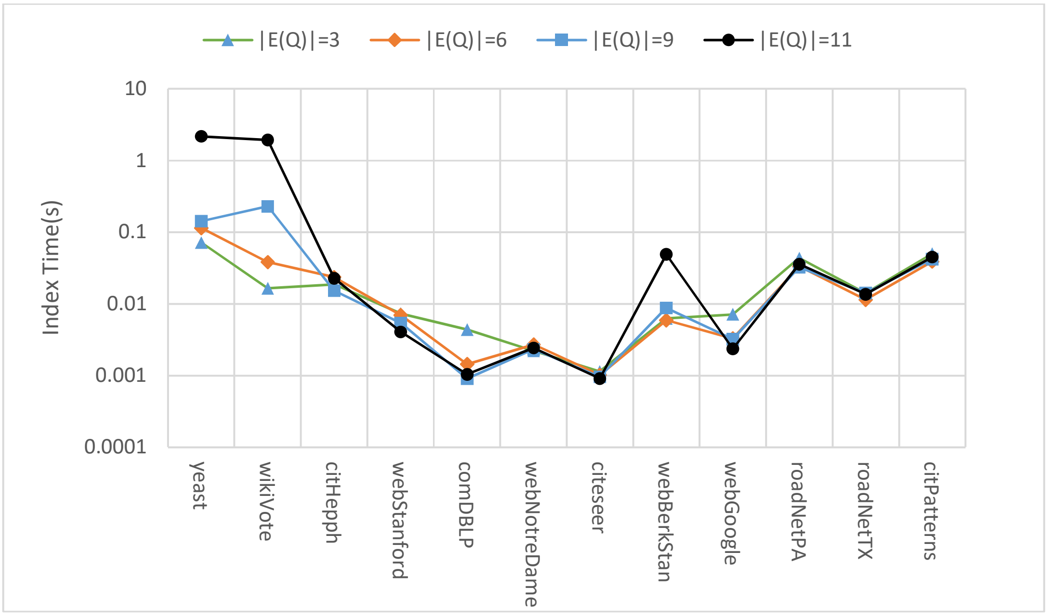
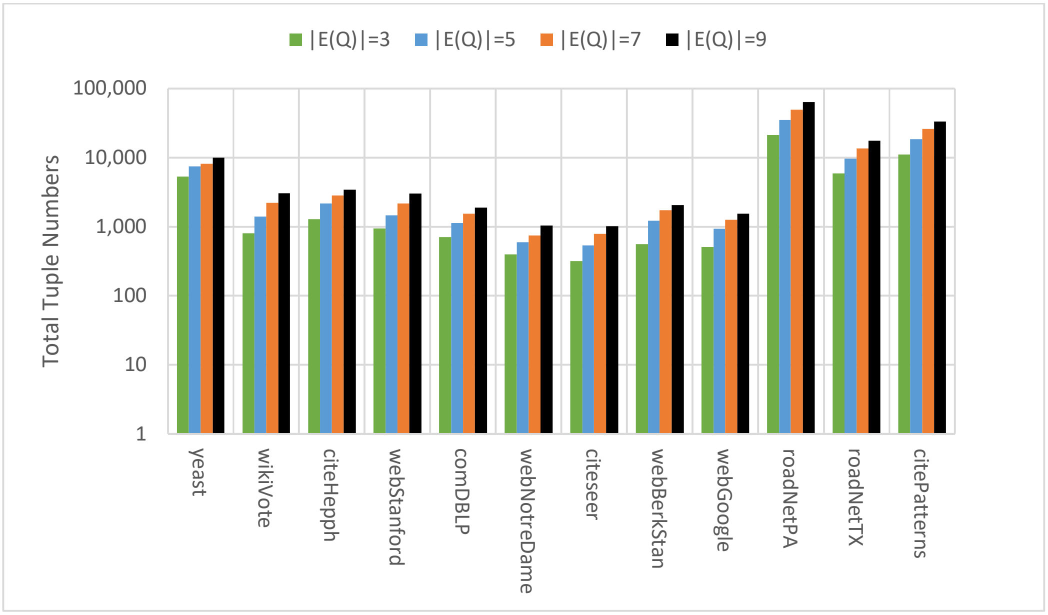


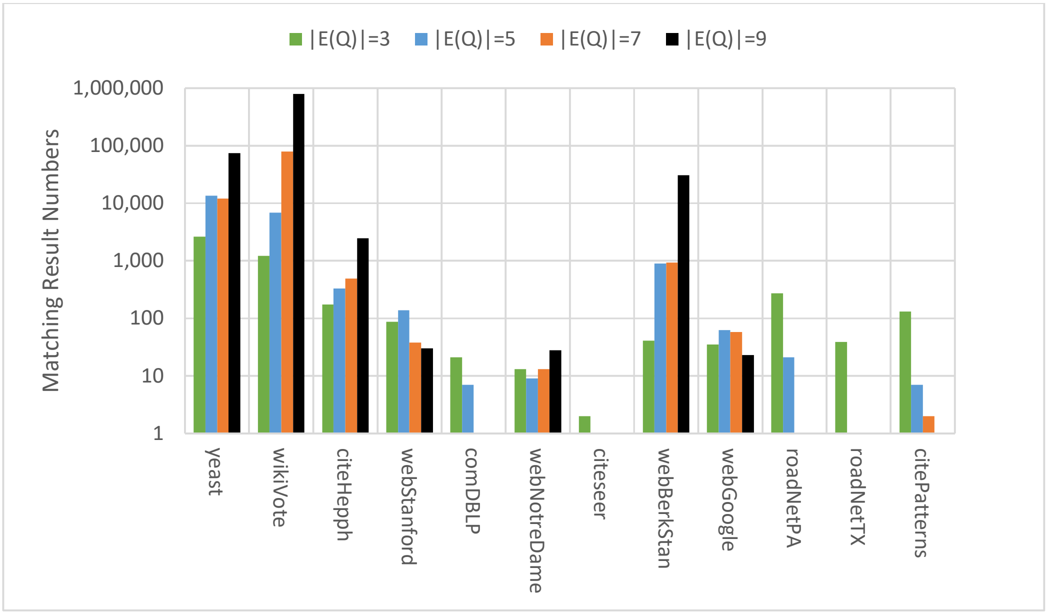
Figure 32: The query times of our method for all data graphs by varying the numbers of query edges.

8.8.3 The Tuple Numbers and Matching Result Numbers

Figure 33 shows the tuple numbers and matching result numbers for the different number of query edges. We can see that the total tuples increase linearly with query edges. However, the number of matching results do not have a clear trend, which explains the variation of query time shown above.



(a) Total tuple numbers



(b) Matching result numbers

Figure 33: The total tuple numbers and matching result numbers for all data graphs by varying the number of query edges.

8.8.4 The Tuple Numbers Filtered By DF&RF

Table 10 shows the tuple numbers of for each real data graphs after DF&RF algorithms by varying the number of query edges. We can observe that our DF algorithm can filter a large number of tuples. In comparison, the number of the tuples removed by the RF is relatively small. More importantly, with the increase of query edges more tuples can be filtered by the DF&RF, which significantly expedite the classical N-join process.

graphs	$\|E(Q)\| = 3$			$\|E(Q)\| = 5$			$\|E(Q)\| = 7$			$\|E(Q)\| = 9$		
	total	after DF	after RF	total	after DF	after RF	total	after DF	after RF	total	after DF	after RF
yeast	5300	2364	2115	7440	3220	2822	8062	3049	2506	9980	3624	2962
wikiVote	808	741	692	1407	1266	1145	2223	1978	1833	3067	2714	2532
citeHepph	1280	931	336	2146	1386	392	2821	1676	423	3435	1885	479
webStanford	945	197	194	1466	127	127	2171	92	91	3011	44	44
comDBLP	708	59	56	1140	14	14	1532	0	0	1884	0	0
webNotreDame	398	208	33	600	267	25	746	35	34	1037	47	46
citeseer	319	7	6	536	5	5	786	0	0	1017	0	0
webBerkStan	560	109	91	1215	159	150	1736	223	214	2058	227	227
webGoogle	511	88	76	929	68	68	1254	62	62	1549	34	34
roadNetPA	21148	789	788	35144	98	98	49246	7	7	63686	0	0
roadNetTX	5914	114	114	9624	0	0	13588	0	0	17534	0	0
citePatterns	11015	458	362	18378	40	30	25810	10	10	33262	0	0

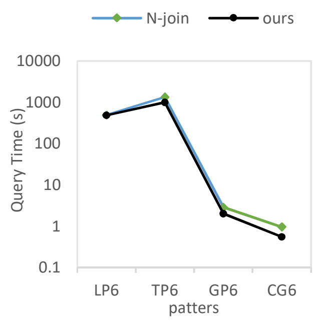
Table 10: the tuple numbers for each real data graphs after DF &RF algorithms by varying the number of query edges.

8.9 Fifth Experiment: Impact of Query Patterns

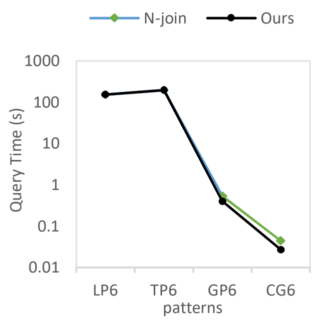
Finally, we analyze the performance of our method by varying the query patterns but fixing the number of query edges to 6. Here we test 4 kinds of queries LP6, TP6, GP6 and CG6 corresponding to 4 patters (see Section 8.4) with 6 query edges against 12 real data graphs (see Table 3). We expect to see the different query patterns can affect the query performance.

8.9.1 The Query Time for Each Graphs

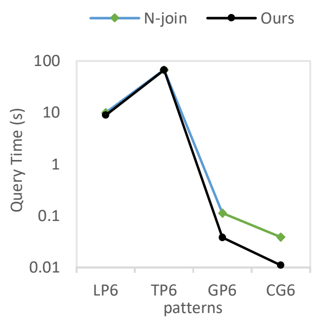
From Figure 34, we can see that LP6 and TP6 tend to require much more query time than GP6 and CG6 for both the N-join and our methods, especially for small graphs such as yeast and wikiVote, as they are of complicated structures and have fewer matching results. We also observe that for GP6 and CG6 our method have a larger speeding-up to the classical N-join for LP6 and TP6. The reason is that for the same number of query edges both GP6 and CG6 generally have much more limitations than LP6 and TP6, which lead to a higher percentage of redundant tuples eliminated by our DF&RF algorithms. Obviously, the CG6, as a complete graph pattern, has the most limitation and the percentage of tuples removed by our DF&RF algorithms is the highest among all these query patterns. So, for this kind of query, a big difference between the N-join and ours can be observed.



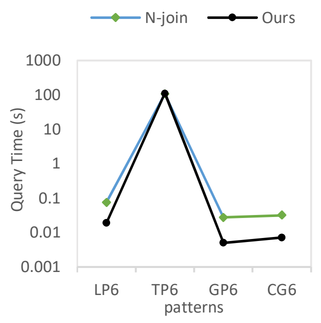
(a) yeast



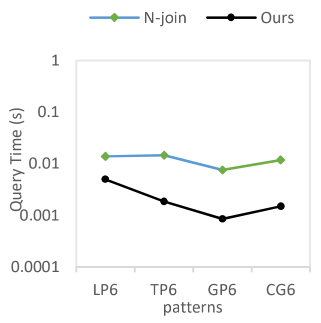
(b) wikiVote



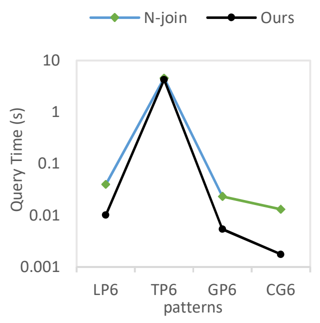
(c) citeHepph



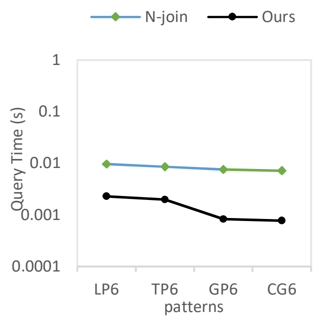
(e) webstanford



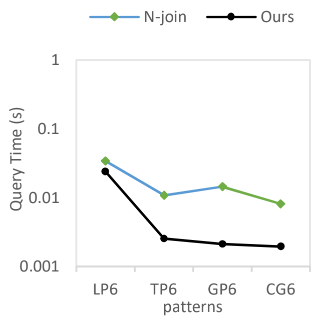
(f) comDBLP



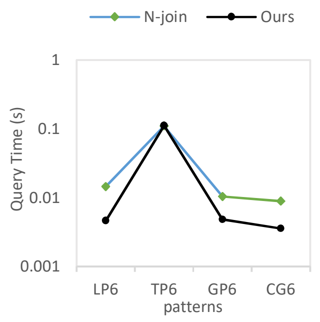
(g) webNotreDame



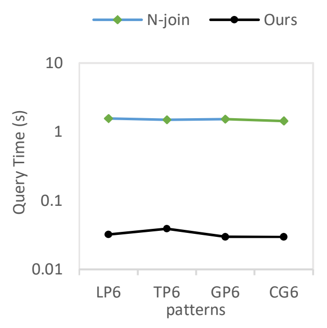
(e) citeseer



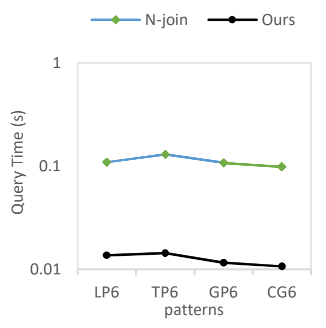
(f) webBerkStan



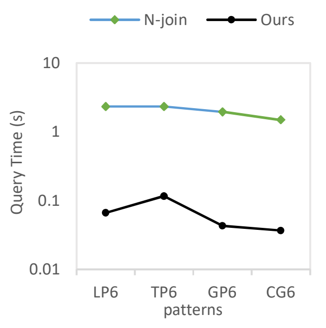
(g) webGoogle



(h) roadNetPA



(i) roadNetTX



(j) citePatterns

Figure 34: The query times of each real graph for N-join and our method by fixing $|E(Q)| = 6$ but varying the query patterns.

8.9.2 The Query Times for Each Pattern Query

From Figure 35, it is clear to see that our methods have mostly a better performance for GP6 and CG6 than LP6 and TP6. However, for larger data graphs, this running time differences becomes small. This can be explained by their matching result numbers (see the next section).

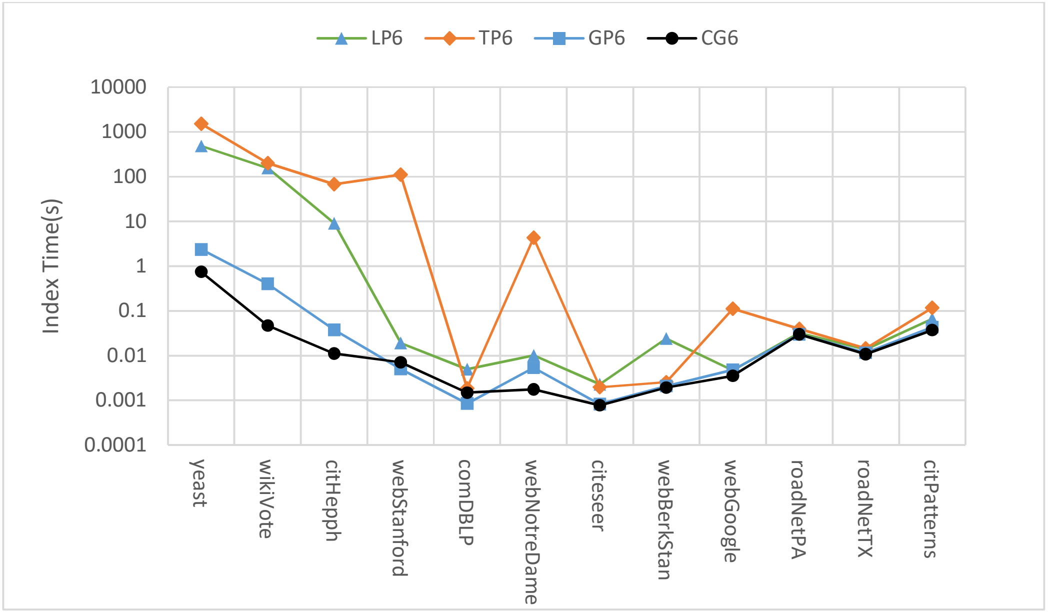
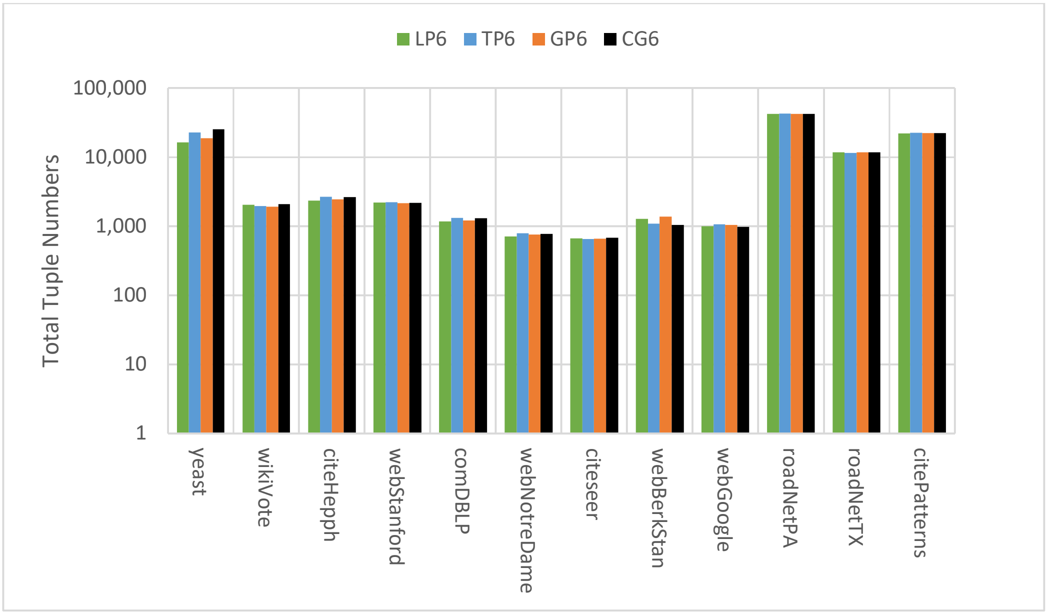


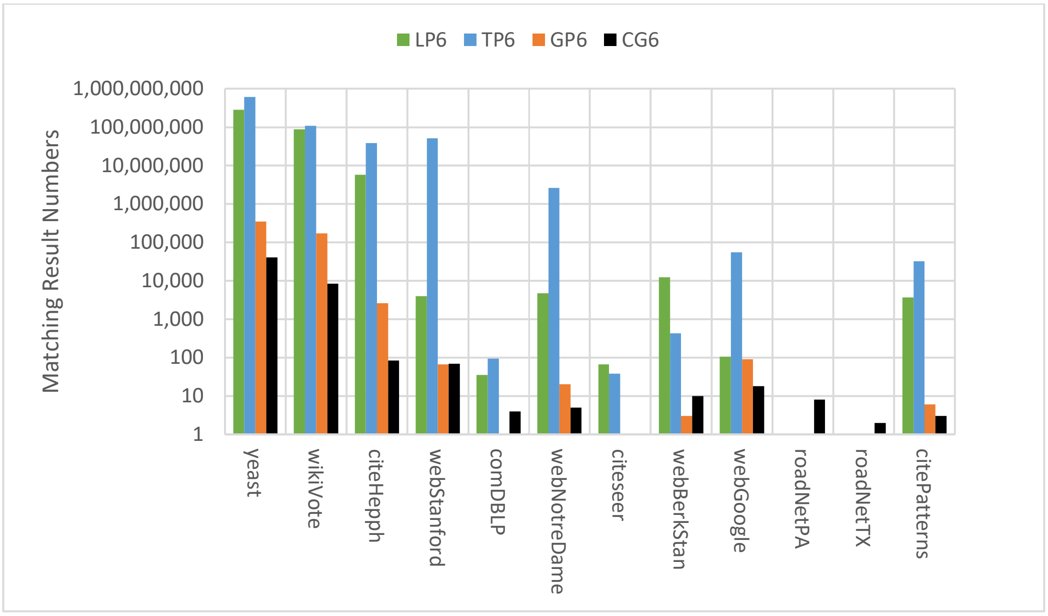
Figure 35: The query times of each query pattern for our method by fixing $|E(Q)| = 6$.

8.9.3 The Tuple Numbers and Matching Result Numbers

From Figure 36(a), we observe that for the different patterns of queries it has nearly the same number of total tuples. However, Figure 36(b) shows that their matching results have a big differences. In general, LP6 and TP6 have much more matching results than GP6 and CG6.



(a) Total tuple numbers



(b) Matching result numbers

Figure 36: The tuple numbers and matching result number of each query pattern for our method by fixing the query number = 6.

8.9.4 The Tuple Numbers Filtered by DF&RF

Table 11 shows the tuple numbers after filtered by DF and RF algorithms for the different queries of patterns. For LP6 and TP6, we observe that the DF algorithm can filter a large percentage of tuples especially for large data graphs, but the RF algorithm has almost no effect since both LP and TP contain no triangles. Only for GP6 and CG6, the RF algorithm is able to remove some unqualified tuples. Generally, our DF and RF algorithms are more powerful to remove redundant tuples in GP6 and CG6 than in LP6 and TP6. This explains why our method performs better in GP6 and CG6.

graphs	LP6			TP6			GP6			CG6		
	total	after DF	after RF	total	after DF	after RF	total	after DF	after RF	total	after DF	after RF
yeast	16258	7577	7577	22702	10127	10127	18752	8332	8095	25364	11388	10318
wikiVote	2048	1850	1850	1968	1804	1804	1932	1672	1649	2095	1839	1631
citeHepph	2353	1837	1837	2685	2179	2179	2454	1708	1056	2667	1649	271
webStanford	2202	381	381	2246	525	525	2142	105	105	2169	236	232
comDBLP	1180	60	60	1316	49	49	1214	0	0	1310	22	21
webNotreDame	718	135	135	792	257	257	762	291	95	771	306	18
citeseer	663	28	28	651	23	23	660	0	0	681	0	0
webBerkStan	1276	186	186	1090	29	29	1390	12	12	1050	43	43
webGoogle	1002	63	63	1070	261	261	1045	83	83	988	64	58
roadNetPA	42454	0	0	42674	0	0	42236	6	6	42168	45	45
roadNetTX	11730	0	0	11524	0	0	11668	0	0	11722	12	12
citePatterns	22062	775	775	22490	999	999	22197	29	29	22250	18	18

Table 11: The tuple numbers of each real data graphs after DF&RF algorithms by fixing $|E(Q)| = 6$ but varying the query patterns.

CHAPTER 9 CONCLUSION AND FUTURE WORK

9.1 Conclusion

In this thesis, we present and analyze a new general framework for performing patter matching queries over large data graphs. We divide the problems of pattern matching queries into two part, RC (Relation Construction) and MC (Matching Results Construction). For RC part, considering that the value of δ tends to be small in pattern matching queries, we propose the notion of Δ-Transitive Closure, G^{Δ}, where Δ is the maximum value of received δ, instead of the traditional transitive closure. By doing this, we reduce the running time to $O(Nd^{\Delta})$ for unweighted data graphs and $O(Nd^{\Delta}logd^{\Delta})$ for weighted data graphs on average, respectively, where $N = |V(G)|$ and d is the average degree of vertices in data graphs. We also reduce the space usage to $O(Nd^{\Delta})$ in both unweighted and weighted data graphs on average. Most importantly, all of this can be done offline as indexes and do not need online running time when receiving a query with δ.

For the MC part, the classical Nature Joins are adopted to construct all the matching results. However, this kind of Natural Joins is NP-complete, taking $O(\prod_{ij}|R_{ij}|)$ running time. To speed up the MC part, we propose two-level filtering strategy, DF (Domain Filtering) and RF (Relation Filtering), in order to remove the redundant tuples that are not necessary to participate the Natural Joins in all relations R_{ij}. The running time DF and RF are both bounded by $O(mD^2)$ and $O(n^3D'^3)$, respectively, where $m = |E(Q)|$, $D = \max\{|R_1|, \dots, |R_n|\}$ and D' is the D firstly filtered by DF. From the experiments we can see our two filtering algorithms are very efficient to speed up the Natural Joins and with reasonable running time.

9.2 Future Work

As a future work, we will optimize the indexing time and size in order to well handle large data graphs since currently the real-life graphs tend to be larger and larger. For example, “Facebook” has 2.2 billion monthly active users (January 2018), which is a huge social network graph. The straightforward method is to use the parallel computing to speed up the indexing process. In this way, the related algorithms should be redesigned by the parallel programming and one or more GPU are required. Furthermore, in order to reduce the index size, we could choose to calculate the shortest-path distances online when receiving a query with δ. This may require reasonable running time if it is sped up by the parallel computing.

The other topic is to extend the shortest-path distance limitation used in this thesis to other kind of limitations. For example, as we mentioned in the related work, in [25], the reachability limitation is used to handle the graph pattern matching problems. In addition, in [41], a new limitation, called label-constrained reachability, is proposed. Specifically, for a directed edge-labeled graph, we want to know if there is a path for a given source vertex to a given target vertex using only edges with labels from a restricted subset. This is a fundamental query in some social networks like a citation network. Besides, if our data graphs are geographic graphs (such as our tested graphs roadNetPA and roadNetTx shown in Table 5), the shorted-path distance can also be easily extended to Euclidean distance or Chebyshev distance.

REFERENCE

[1] D. Shasha, J. T. L. Wang, and R. Giugno, “Algorithmics and applications of tree and graph searching,” *ACM SIGMOD-SIGACT-SIGART Symp. Princ. Database Syst.*, p. 39, 2002.

[2] J. Cheng, Y. Ke, W. Ng, and A. Lu, “Fg-index: towards verification-free query processing on graph databases,” *Proc. 2007 ACM SIGMOD Int. Conf. Manag. data*, pp. 857–872, 2007.

[3] H. Jiang, H. Wang, P. S. Yu, and S. Zhou, “Gstring: A novel approach for efficient search in graph databases,” *Data Eng. 2007. ICDE 2007. IEEE 23rd Int. Conf.*, pp. 566–575, 2007.

[4] Y. Tian, R. C. McEachin, C. Santos, D. J. States, and J. M. Patel, “SAGA: A subgraph matching tool for biological graphs,” *Bioinformatics*, vol. 23, no. 2, pp. 232–239, 2007.

[5] X. Yan, P. S. Yu, and J. Han, “Graph indexing: a frequent structure-based approach,” *Proc. 2004 ACM SIGMOD ...*, pp. 335–346, 2004.

[6] S. Zhang, M. Hu, and J. Yang, “TreePi: A novel graph indexing method,” *Proc. - Int. Conf. Data Eng.*, pp. 966–975, 2007.

[7] D. W. Williams, J. Huan, and W. Wang, “Graph Database Indexing Using Structured Graph Decomposition Department of Computer Science,” *Data Eng. 2007.*, pp. 976–985, 2007.

[8] P. Zhao, J. X. Yu, and P. S. Yu, “Graph indexing: tree+ delta<= graph,” *Proc. 33rd Int. Conf. Very large data bases*, no. October 2016, pp. 938–949, 2007.

[9] P. Zhao, “On graph query optimization in large networks,” *Proc. VLDB Endow.*, no. 1, pp. 340–351, 2010.

[10] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, “Fast graph matching for detecting CAD image components,” *Proceedings 15th International Conference on Pattern Recognition. ICPR-2000*, vol. 2. pp. 1034–1037, 2000.

[11] L. P. Cordella, P. Foggia, C. Sansone, F. Tortorella, and M. Vento, “Graph Matching : A Fast Algorithm and its Evaluation,” *Fourteenth Int. Conf. Pattern Recognit.*, pp. 1582–1584, 1998.

[12] N. Linial, E. London, and Y. Rabinovich, “The geometry of graphs and some of its algorithmic applications.,” *Combinatorica*, no. 215–245, 1995.

[13] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler, “Matrix ‘Bit’ loaded: a scalable lightweight join query processor for RDF data,” *Proc. 19th Int. Conf. World wide web*, pp. 41–50, 2010.

[14] T. Neumann and G. Weikum, “The RDF-3X engine for scalable management of RDF data,” *VLDB J. Int. J. Very Large Data Bases*, vol. 19, no. 1, pp. 91–113, 2010.

[15] H. He and A. K. Singh, “Graphs-at-a-time : Query Language and Access Methods for Graph Databases Categories and Subject Descriptors,” *Sigmod*, pp. 405–418, 2008.

[16] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick, “Reachability and Distance Queries via 2-Hop Labels,” *SIAM J. Comput.*, vol. 32, pp. 1338–1355, 2003.

[17] J. Cheng and J. X. Yu, “On-line exact shortest distance query processing,” *Proc. 12th Int. Conf. Extending Database Technol. Adv. Database Technol. EDBT 09*, p. 481, 2009.

[18] N. Jing, Y. W. Huang, and E. A. Rundensteiner, “Hierarchical encoded path views for path query processing: An optimal model

and its performance evaluation," *IEEE Trans. Knowl. Data Eng.*, vol. 10, no. 3, pp. 409–432, 1998.

[19] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu, "Dual labeling: Answering graph reachability queries in constant time," *Proc. - Int. Conf. Data Eng.*, vol. 2006, p. 75, 2006.

[20] S. Tribl and U. Leser, "Fast and Practical Indexing and Querying of Very Large Graphs," 2007.

[21] Y. Chen and Y. Chen, "An efficient algorithm for answering graph reachability queries," *Proc. - Int. Conf. Data Eng.*, no. May, pp. 893–902, 2008.

[22] W. E. Moustafa, A. Kimmig, A. Deshpande, and L. Getoor, "Subgraph pattern matching over uncertain graphs with identity linkage uncertainty," *Proc. - Int. Conf. Data Eng.*, pp. 904–915, 2014.

[23] H. He and A. K. Singh, "Closure - Tree : An Index Structure for Graph Queries," 2006.

[24] L. Chen, "Distance-Join : Pattern Match Query In a Large Graph," *Vldb*, vol. 2, no. 1, pp. 886–897, 2009.

[25] J. Cheng, J. X. Yu, B. Ding, P. S. Yu, and H. Wang, "Fast graph pattern matching," *Proc. - Int. Conf. Data Eng.*, pp. 913–922, 2008.

[26] H. Tong, B. Gallagher, C. Faloutsos, and T. Eliassi-Rad, "Fast best-effort pattern matching in large attributed graphs," *Proc. 13th ACM SIGKDD Int. Conf. Knowl. Discov. data Min.*, p. 737, 2007.

[27] S. Melnik, H. Garcia-Molina, and E. Rahm, "Similarity flooding: A versatile graph matching algorithm and its application to schema matching," *Proc. - Int. Conf. Data Eng.*, pp. 117–128, 2002.

[28] S. Toresen, “An efficient solution to inexact graph matching with applications to computer vision,” Norwegian University of Science and Technology, 2007.

[29] J. E. Hopcroft and J.-K. Wong, “Linear time algorithm for isomorphism of planar graphs (preliminary report),” in *Proceedings of the sixth annual ACM symposium on Theory of computing*, 1974, pp. 172–184.

[30] E. M. Luks, “Isomorphism of graphs of bounded valence can be tested in polynomial time,” *J. Comput. Syst. Sci.*, vol. 25, no. 1, pp. 42–65, 1982.

[31] F. Zhu, Q. Qu, D. Lo, X. Yan, J. Han, and P. S. Yu, “Mining Top-K Large Structural Patterns in a Massive Network,” *37th Int. Conf. Very Large Data Bases*, p. Vol. 4, No. 11, 2011.

[32] X. Yan, P. S. Yu, and J. Han, “Substructure similarity search in graph databases,” in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, 2005, pp. 766–777.

[33] C. Shahabi, M. R. Kolahdouzan, and M. Sharifzadeh, “A road network embedding technique for K-nearest neighbor search in moving object databases,” *Geoinformatica*, vol. 7, no. 3, pp. 255–273, 2003.

[34] R. Elmasri and S. B. Navathe, *Fundamentals of database systems*. Pearson, 2015.

[35] J. R. Ullmann, “An Algorithm for Subgraph Isomorphism,” *J. ACM*, vol. 23, no. 1, pp. 31–42, 1976.

[36] L. Cordella, P. Foggia, C. Sansone, and M. Vento, “A (Sub)Graph Isomoprhism Algorithm for Matching Large Graphs,” *IEEE Trans.*

Pattern Anal. Mach. Intell., vol. 26, no. 10, pp. 1367–1372, 2004.

[37] Thomas H Cormen; Charles Eric Leiserson; Ronald L Rivest; Clifford, *Introduction to Algorithms*, Third edit. The MIT Press, 2014.

[38] L. Zou, L. Chen, M. T. ??zsu, and D. Zhao, “Answering pattern match queries in large graph databases via graph embedding,” *VLDB J.*, vol. 21, no. 1, pp. 97–120, 2012.

[39] M. Steinbrunn, G. Moerkotte, and A. Kemper, “Optimizing Join Orders,” pp. 1–55.

[40] R. Albert and A.-L. Barabasi, “Statistical mechanics of complex networks,” 2001.

[41] L. D. J. Valstar and G. H. L. Fletcher, “Landmark Indexing for Evaluation of Label-Constrained Reachability Queries,” pp. 345–358, 2017.