δ-Transitive closures and triangle consistency checking: a new way to evaluate graph pattern queries in large graph databases

Yangjun Chen, Bin Guo & Xingyue Huang

The Journal of Supercomputing
An International Journal of High-Performance Computer Design, Analysis, and Use

ISSN 0920-8542

J Supercomput
DOI 10.1007/s11227-019-02762-4

ONLINE FIRST

VOLUME 65, NUMBER 3
September 2013
ISSN 0920-8542

THE JOURNAL OF SUPERCOMPUTING

High Performance Computer Design, Analysis, and Use

Your article is protected by copyright and all rights are held exclusively by Springer Science+Business Media, LLC, part of Springer Nature. This e-offprint is for personal use only and shall not be self-archived in electronic repositories. If you wish to self-archive your article, please use the accepted manuscript version for posting on your own website. You may further deposit the accepted manuscript version in any repository, provided it is only made publicly available 12 months after official publication or later and provided acknowledgement is given to the original source of publication and a link is inserted to the published article on Springer's website. The link must be accompanied by the following text: "The final publication is available at link.springer.com".

Springer

Author's personal copy

The Journal of Supercomputing
https://doi.org/10.1007/s11227-019-02762-4

δ-Transitive closures and triangle consistency checking: a new way to evaluate graph pattern queries in large graph databases

Yangjun Chen[1] · Bin Guo[1] · Xingyue Huang[1]

© Springer Science+Business Media, LLC, part of Springer Nature 2019

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 queries, 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 compared with other problems mentioned above. In this paper, we address a kind of graph matching, the so-called *graph 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 new index structure and a novel pruning technique to eliminate a lot of unqualified vertices before join operations are carried out. Extensive experiments have been conducted, which show that our approach makes great improvements in running time compared to existing ones.

Keywords Graph matching · δ-Transitive closures · Triangle consistency · Join ordering

✉ Yangjun Chen
y.chen@uwinnipeg.ca

Bin Guo
guo-b75@webmail.uwinnipeg.ca

Xingyue Huang
huangxy19910321@163.com

[1] Department of Applied Computer Science, University of Winnipeg, Winnipeg, Canada

Published online: 01 February 2019

Author's personal copy

1 Introduction

Nowadays, in numerous applications, including social networks, biological networks, and WWW networks, as well as geographical networks, data are normally organized into 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* [1–4], *shortest path queries* [5], *reachability queries* [6–9], and *pattern matching queries* [10–15]. Among them, the pattern matching is very challenging, 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 [16]; (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 [17]; (3) in the computer vision, a scene is naturally represented as a graph $G(V, E)$, where a feature is a vertex in V and an edge in E stands for a geographical adjacency of two features [18]. Then, a scene recognition is just a matching of a graph standing for part of a scene to another stored in databases.

The first two applications mentioned above are typically exact 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 [19]. A lot of work has been done on this problem, but most of them are for special kind of graphs, such as [20] for planar graphs and [21] for valence graphs, or by establishing indexes [17, 22–27], or using some kind of heuristics [28, 29] to speed up the working process. The third application is a kind of inexact matching. First, two matching features from two graphs may disagree in some way due to different observations of a same object. Secondly, between two adjacent features in a graph may there be some more features in another graph [18] figured out by a more detailed description. This leads to a new kind of queries, called *pattern matching with* δ (or *graph matching with* δ), where δ is a number, by which an edge in a query graph is allowed to match a path of length $\leq \delta$ 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 $\leq \delta$. Here, the distance between u and u' is defined to be the weight of the shortest path connecting these two vertices, denoted as $dist(u, u')$. Note that when $\delta = 1$, the problem reduces to the normal subgraph isomorphism.

Tong et al. [11] discussed the first algorithm, based on *joins*, for evaluating such queries with $\delta = 2$. Its worst time complexity is bounded by $O(\prod_i |D_i|)$, where

Author's personal copy

D_i is a subset of vertices in G with the same label as $label(v_i)$ ($v_i \in Q$). Cheng et al. [12] extended this algorithm to evaluate the same kind of queries, but with δ unlimited. For such more general and more useful queries, an index structure is introduced, called a *2-hop cover* [30], which can be used to facilitate the construction of relations R_{ij} each corresponding to an edge $(v_i, v_j) \in Q$ such that for each $<u, u'> \in R_{ij}$ the shortest distance between u and u' is bounded by δ. But its running time is also bounded by $O(\prod_i |D_i|)$.

Cheng's algorithm has been greatly improved by Zou et al. [13]. They map all vertices in $G(V, E)$ into the points in a *vector space* of k-dimensions by using the so-called *LLR embedding* technique discussed in [31, 32], where k is selected to be $O(\log^2|V|)$ to save space. In this way, the computation of distance between each pair of vertices can be done a little bit faster. However, the worst time complexity remains unchanged.

The main disadvantage of [12, 13] is due to the usage of 2-hop covers as their indexes, which needs to first construct the whole *transitive closure* of $G(V, E)$ as a pre-data structure and requires $O(|V|^3)$ time and $O(|V|^2)$ space in the worst case, obviously not scaling to larger graphs which are common in contemporary applications. For example, in our experiments, to create the 2-hop cover of a real graph *roadNetPA* of 1,088,092 vertices and 1,541,898 edges on a desktop with a 2.40-GHz Intel Xeon E5-2630 processor and 32 GB RAM, we need 33.4 h. The space requirement is about 27.023G bytes. For another real graph *roadNetTX* of 1, 379, 917 vertices and 1,921,660 edges, we are not able to get the result within 3 days!

In this paper, we address this limitation and propose a new method to evaluate pattern matching with δ based on a new concept of δ-transitive closure of G, used as an index, as well as a filtering method to remove useless data before a join is conducted. Besides, the bit mapping technique is also integrated into our filtering method to speed up the computation.

1. (*δ-transitive closure*) For a weighted (directed or undirected) graph G, we will construct an index over it for a given constant δ, called the δ-transitive closure of G and denoted as G^δ, which is a graph with $V(G^\delta) = V(G)$ and has an edge between vertex u and vertex u' if and only if the shortest distance between these two vertices is $\leq \delta$.
2. (*relation filtering*) Let $e = (v_i, v_j)$ be an edge in Q. Let R_{ij} be a relation corresponding to $<label(v_i), label(v_j)>$ such that for each $<u, u'>$ in R_{ij} there is a path from u to u' with their shortest distance $\leq \delta$ in G, $label(u) = label(v_i)$ and $label(u') = label(v_j)$. A tuple $<u, u'>$ in R_{ij} is said to be *triangle consistent* if for any $v_k \in Q$ incident to v_i, v_j, or both of them there exists at least a vertex $u'' \in D_k$ such that $<u', u''>$ ($<u'', u'>$) $\in R_{jk}$ (resp. R_{kj}) and $<u, u''>$ ($<u'', u>$) $\in R_{ik}$ (resp. R_{ki}). If v_i or v_j is not incident to v_k, we consider that a *virtual* relation between v_i (or v_j) and v_k exists and for any $u \in D_i$ (or $u' \in D_j$) and any $u'' \in v_k$ we have $<u, u''>$ (or $<u', u''>$) belonging to this virtual relation. For example, if we do not have (v_j, v_k) in Q, then u' need not be checked against any vertex u'' in D_k as if $<u', u''>$ always exists in the corresponding virtual relation R_{jk}. Our goal is to

Author's personal copy

remove all those tuples which are not triangle consistent before join operations are actually conducted, which enables us to improve efficiency by one order of magnitude or more.

3. (*bit mapping*) The relation filtering works in a propagational way. That is, eliminating a tuple from an R_{ij} may lead to some more tuples from some other relations removed. To expedite this process, a kind of bit mapping techniques is used, which further decreases the time complexity.

The reminder of the paper is organized as follows: In Sect. 2, we give the basic concepts of the problem. In Sect. 3, we discuss the related work. Section 4 is devoted to the index construction. In Sects. 5 and 6, we discuss the relation filtering and the join ordering, respectively. In Sect. 7, we report the experiment results. Finally, a short conclusion is set forth in Sect. 8.

2 Problem statements

In this section, we give a formal definition of the pattern matching queries with δ over directed weighted graphs G. First of all, G should be a *Weakly Connected Component* (*WCC*) (i.e., the undirected version of G is connected); otherwise, we can decompose G into a collection of *WCC*s and perform pattern matching over each *WCC* in turn. Secondly, we will use the shortest path length to measure the distance between two vertices. However, our approach is not restricted to this distance function, and it can also be applied to other metrics without any difficulty.

Definition 2.1 (*Data Graph G*) A data graph $G = (V(G), E(G), \Sigma)$ is a vertex-labeled, directed, weighted, and weakly connected graph. Here, $V(G)$ is a set of labeled vertices, $E(G)$ is a set of edges (ordered pairs) each with a weight represented as a nonnegative number, and Σ is a set of vertex labels. Each vertex $u \in V(G)$ is assigned a label $l \in \Sigma$, denoted as $label(v) = l$.

Definition 2.2 (*Query Graph Q*) A query Q is a vertex-labeled and directed 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 $l \in \Sigma$.

Definition 2.3 (*List $\boldsymbol{D}[l]$*) Given a data graph G, we use $\boldsymbol{D}[l]$ 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 $u \in \boldsymbol{D}[l]$.

Sometimes we also use the notation $\boldsymbol{D}$, instead of $\boldsymbol{D}[l]$, if its label l is clear from the context. Let v be a vertex in Q with $label(v) = l$. We also call $\boldsymbol{D}[l]$ the domain of v.

Author's personal copy

Definition 2.4 (*Edge Query*) 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 $<u_i, u_j>$ 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, $Dist(u_i, u_j) \leq \delta$.

Definition 2.5 (*Pattern Matching Query with* δ) Given a data graph G, a query graph Q with n vertices $\{v_1, \ldots, v_n\}$ and a parameter δ, the evaluation of Q reports all matching results $<u_1, \ldots u_n>$ in G if the following conditions hold:

1. $label(u_i) = label(v_i)$ for $1 \leq i \leq n$;
2. For any edge $(v_i, v_j) \in Q$, the distance between u_i and u_j in G is no larger than δ, i.e., $Dist(u_i, u_j) \leq \delta$.

In Fig. 1a, we show a simple graph, in which the numbers inside the vertices are their IDs and the letters attached to them are their labels. There are altogether 4 labels: $\Sigma = \{A, B, C, D\}$. Each edge is also associated with a number, representing its weight. In Fig. 1b, we show a simple query Q.

Since Q contains three vertices labeled with $A, B, C \in \Sigma$, respectively, three lists: $\boldsymbol{D}[A]$, $\boldsymbol{D}[B]$, $\boldsymbol{D}[C]$ from G will be constructed. For example, $\boldsymbol{D}[A] = \{u_6, u_7, u_8\}$.

In addition, the query Q also has 3 query edges, $\{(v_1, v_2), (v_2, v_3), (v_3, v_1)\}$. If the parameter $\delta = 5$, the pairs matching a query edge (v_1, v_2) (with labels (A, B)) are $\{<u_6, u_3>, <u_7, u_9>, <u_8, u_9>, <u_8, u_5>\}$. The matching result of the whole query Q is $\{<u_8, u_9, u_{10}>, <u_7, u_9, u_4>\}$. If $\delta = 6$, the pairs matching (v_1, v_2) is the same as $\delta = 5$. But the matching result of the whole query Q will be augmented by adding $\{<u_8, u_9, u_4>\}$.

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

In a similar way, we can also define the problem for undirected, weighted graphs, by simply removing directions of edges when calculating the distance between two vertices.

Finally, we should notice that if G contains negative-weight cycles the shortest path from a vertex to another may not be well defined. It is because by a negative-weight

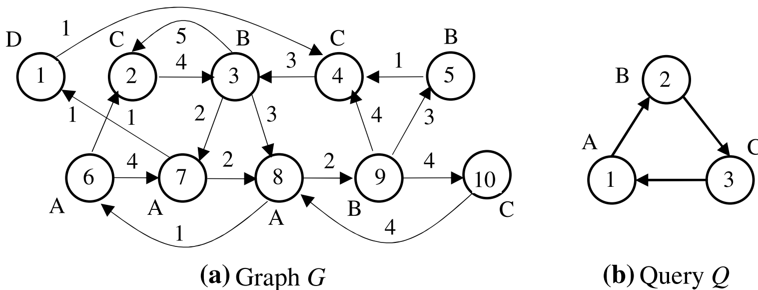


Fig. 1 Examples of a data graph G and a query Q

Author's personal copy

Table 1 Meaning of used symbols

Directed data graph G			Directed query graph Q
$V(G)$	Vertex set of G	$V(Q)$	Vertex set of Q
$E(G)$	Edge set of G	$E(Q)$	Edge set of Q
u_i	A vertex in G	v_i	A vertex in Q
$Label(u_i)$	Label of u_i	n	Number of vertices in Q
$Dist(u_i, u_j)$	Distance between u_i and u_j	m	Number of edges in Q

cycle C, we have some edges associated with negative weights and all the weights of the edges on C can sum up to a negative value. Thus, we can always find a path with lower weight by following such a cycle. In this paper, negative-weight cycles will not be considered.

3 Related work

The first algorithm for handling pattern matching with δ was discussed in [11], 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 path of length 2. That is, v_i, 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 graph G, a set of starting points will be determined. Normally, they are some vertices having the same label as a vertex v, but with the largest degree in Q.
- A *goodness* score function $g(u)$ $(u \in G)$ is used to guide the searching of G such that only those 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.

The method discussed in [12] is general, in which an index called a *2-hop cover* is used to speed up the computation of distances between each pair of vertices u and u' with $u \rightsquigarrow u'$, where $\rightsquigarrow$ represents that u is reachable to u' through a path. By the 2-hop labeling method, each vertex u in G will be assigned a label $L(u) = (L_{in}(u), L_{out}(u))$, where $L_{in}(u), L_{out}(u) \subseteq V(G)$ and $u \rightsquigarrow u'$ if and only if $Lout(u) \cap Lin(u') \neq \Phi$. To find the minimum size of a 2-hop cover for each u in G proves to be *NP*-hard [6]. Therefore, in practice, only a nearly optimal solution is used [6, 18].

The vertices in both $L_{in}(u)$ and $L_{out}(u)$ are called *centers*. By using the centers, $Dist(u, u')$ can be computed as follows [12]:

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

Author's personal copy

Clearly, the distances between a vertex u and a center w (i.e., $Dist(u, w)$ or $Dist(u, w)$) can be pre-computed. As shown in [6], for each vertex the average size of distance label is bounded by $O(|E(G)|^{1/2})$. Thus, by using Eq. (1), we need $O(|E(G)|^{1/2})$ time to compute distances for each pair of reachable vertices in the worst case.

According to (1), a join algorithm was discussed in [12]. The main idea of this algorithm is as follows: based on the 2-hop labeling method [6], for each center w, two clusters $F(w)$ and $T(w)$ of vertices are defined and 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 and labeled l if w is in $F(w)$ or labeled l' if it is in $T(w)$. 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 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* or *reachability-join*. When the number of edges in Q is larger than one, a series of *R-joins* need to be conducted. The worst time complexity of this method is bounded by $O(\prod_i |D_i|)$, where D_i is a subset of vertices in G with the same label as $label(v_i)$ with $v_i \in Q$ $(i = 1, \ldots, n)$.

Zou et al. [13] extended the idea of [12] by proposing a general framework for handling pattern matching queries with δ. By this method, for each edge (v_i, v_j) in Q with label (A, B), a *D-join* (distance-based join) algorithm is conducted to get all the matches in G, according to Eq. 2 given below, where D_i and D_j are two lists, respectively, for two vertices v_i and v_j in Q, while u and u' are two vertices, respectively, in the two lists.

$$\mathrm{RS} = D_i \bowtie D_j \tag{2}$$

In order to reduce the cost of this join, the so-called *LLR embedding* technique discussed in [31, 32] is utilized to map all vertices of G into the points of a k-dimensional vector space. Here k is selected to be $O(\log^2|V(G)|)$ to save space. Then, the *Chebyshev* distance [33] between each pair of points u and u' in the vector space, referred to as $L_\infty(u, u')$, is computed. In comparison with the approach discussed in [12], this method is more efficient since the Chebyshev distance is easy to calculate. Furthermore, the *k-medoids* algorithm [34] is used to divide each R_i (more exactly, the points corresponding to R_i) into different clusters $\boldsymbol{C}_{ik}$ $(k = 1, \ldots, l$ for some $l)$. For each cluster $\boldsymbol{C}_{ik}$, a center c_{ik} is determined and then the radius of $\boldsymbol{C}_{ik}$, denoted as $r(\boldsymbol{C}_{ik})$, is defined to be the maximal L_∞-distance between c_{ik} and a point in $\boldsymbol{C}_{ik}$. During a *D-join* process between lists D_i and D_j, such clusters can be used to reduce computation by checking whether $L_\infty(c_{ik}, c_{jk'}) > r(\boldsymbol{C}_{ik}) + r(\boldsymbol{C}_{jk'}) + \delta$. If it is the case, the corresponding join (i.e., $\boldsymbol{C}_{ik} \bowtie \boldsymbol{C}_{ik'}$) need not be carried out since the L_∞-distance between any two points $u \in \boldsymbol{C}_{ik}$ and $u' \in \boldsymbol{C}_{jk'}$ must be larger than δ. By using the above main pruning method along with *Neighbor Area Pruning* and *Triangle Inequality Pruning* [13], all candidate matching results are evaluated, which will be further checked by the 2-hop labeling technique to get the final results. In addition, in the case of multiple edges of Q, another procedure, called the *MD-join* (distance-based multi-way join), needs to be invoked to filter all the unmatched vertices each

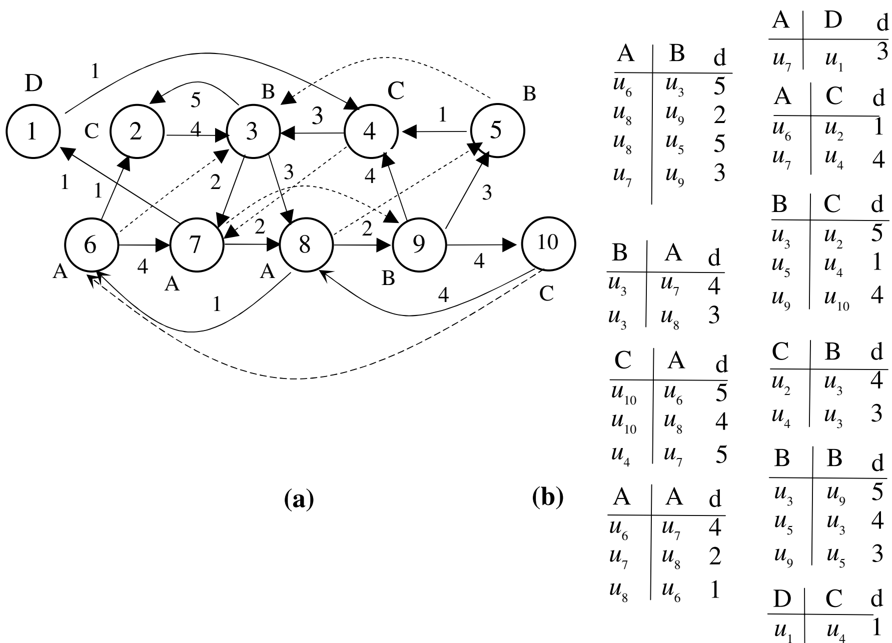
Author's personal copy

time a new query edge is visited. The worst time complexity of this method is the same as [12].

Although a lot of work has been done on subgraph search [1–4], shortest path queries [5], and reachability queries [6–8], none of them can be used or extended to evaluate pattern matching queries with δ since all the pruning techniques developed for them are based on an exact edge matching with δ set to be 1. In addition, the *graph simulation* is a kind of graph matching quite different from the graph isomorphism [35], by which a pattern vertex is allowed to map to multiple vertices in a data graph and therefore can be easily solved in polynomial time. More recently, the so-called bounded simulation has been discussed in [36, 37], by which some more constraints are imposed on a matching and even can be solved in a cubic time [37]. The techniques developed for both of them cannot be employed for our purpose.

4 Construction of indexes

In this section, we begin to describe our method. In a nutshell, our algorithm comprises three steps. In the first step, we will construct offline a δ-transitive closure for a certain δ-value as an index over G, by which a set of binary relations as illustrated in Fig. 2b will be constructed. (In general, how large δ is set is determined according to historical query logs.) Thus, when a query Q is submitted, for each edge $e = (v_i, v_j)$ in Q, a relation corresponding to <$label(v_i)$, $label(v_j)$>, referred to as R_{ij}, will be loaded from hard disk to main memory. In the second step, we will do a relation



A	B	d
u_6	u_3	5
u_8	u_9	2
u_8	u_5	5
u_7	u_9	3

B	A	d
u_3	u_7	4
u_3	u_8	3

C	A	d
u_{10}	u_6	5
u_{10}	u_8	4
u_4	u_7	5

A	A	d
u_6	u_7	4
u_7	u_8	2
u_8	u_6	1

A	D	d
u_7	u_1	3

A	C	d
u_6	u_2	1
u_7	u_4	4

B	C	d
u_3	u_2	5
u_5	u_4	1
u_9	u_{10}	4

C	B	d
u_2	u_3	4
u_4	u_3	3

B	B	d
u_3	u_9	5
u_5	u_3	4
u_9	u_5	3

D	C	d
u_1	u_4	1

Fig. 2 An example of δ-transitive closures

Author's personal copy

filtering. Finally, in the third step, a series of *equi-joins* on the reduced relations will be conducted to get the final results.

In the subsequent discussion, we mainly concentrate on the index construction. The second and third steps will be shifted to Sects. 5 and 6, respectively.

4.1 Constructing δ-transitive closures

As with [12], we will construct an index for graphs to speed up the computation. However, instead of 2-hop covers, we will construct *δ-transitive closures* for them.

Definition 4.1 (*δ-Transitive closures*) Let $G=(V, E, \Sigma)$ be a directed, weighted graph. Let $\delta>0$ be a positive number. A δ-transitive closure of G, denoted G^{δ}, is a graph such that $V(G^{\delta})=V$, and there is an edge $<u, u'> \in E(G^{\delta})$ if and only if $dist(u, u') \leq \delta$.

Example 1 Consider the graph shown in Fig. 1a again. For $\delta=5$, its δ-transitive closure G^5 is a graph as shown in Fig. 2a, which can be stored as a set of binary relations as shown in Fig. 2b, in which each tuple is a pair of vertices, but associated with the corresponding shortest distance ≤ 5 for computation convenience.

This concept is motivated by an observation that δ tends to be small in practice. For instance, in a social network where each vertex u represents a person and each edge represents a relationship like *parent-of*, *brother-of*, *sister-of*, *uncle-of*, and so on, if the weight of each edge is set to be 1, then to check whether a person is a relative of somebody else, setting $\delta=6$ or larger, is not so meaningful. As another example, we consider an application in forensics and assume that a detective may want to investigate an individual who is related to a known criminal through some relationships, such as money laundry and human trafficking. Then, setting $\delta>5$ may not be quite helpful for the detective to find some significant evidence.

Clearly, for different applications, we can adjust the values of δ to pre-compute G^{δ}. One choice for this task is to use an algorithm for finding all-pair shortest paths [38]. However, since such an algorithm needs to store the output in a matrix, it is not so suitable for very large graphs. In this case, we can utilize a single-source shortest path algorithm [39] and run it to generate G^{δ}. Specifically, the following two steps will be conducted:

1. Remove all those edges from $G(V, E)$, whose weight is $>\delta$.
2. Run the single-source shortest path algorithm $|V|$ times each with a different vertex as a source. (The algorithm is slightly modified in such a way that when every value in the heap used by the algorithm is larger than δ, the process stops.)

If all the edge weights are nonnegative, the best algorithm for this problem uses *Fibonacci heap* and needs only $O(|V|^2 \lg\delta + |V||E|)$ time [39].

In comparison with 2-hop covers, the main advantage of δ-transitive closures is threefold:

Author's personal copy

- Less space is required to store a δ-transitive closure than a 2-hop cover if δ is not set so large.
- Much less time is needed to create a δ-transitive closure of G than a 2-hop cover of G. It is because to generate a 2-hop cover the entire transitive closure of G has to be first created, which requires $O(|V|^3)$ time in the worst case, much higher than the time complexity for creating G^δ.
- No online time is needed to form R_{ij}'s while by the methods with 2-hop covers R_{ij}'s have to be generated on-the-fly when a query Q arrives. This is done by using the 2-hop data structures for the relevant edges in Q. Obviously, the response time to queries can be greatly delayed.

4.2 Relation signatures and vertex counters

For each relation in a δ-transitive closure, we can also establish two extra data structures for efficiency: one is a set of *bit string pairs* with each associated with an R_{ij}, called the *relation signature* of R_{ij}; and the other is a set of counters each for a single vertex in G.

4.2.1 Relation signatures

Consider $G(V, E, \Sigma)$. Let $\Sigma = \{l_1, \ldots, l_k\}$. We will divide all the vertices into $|\Sigma|$ disjoint lists, denoted as $\boldsymbol{D}[l_1], \ldots, \boldsymbol{D}[l_k]$ such that $\boldsymbol{D}[l_1] \cup \ldots, \cup \boldsymbol{D}[l_k] = V$, $\boldsymbol{D}[l_i] \cap \boldsymbol{D}[l_j] = \phi$ for $i \neq j$, and all the vertices in a $\boldsymbol{D}[l_i]$ have the same label l_i. Then, we sort all vertices in $\boldsymbol{D}[l_i]$ in ascending order by their vertex IDs and then refer to each vertex by its order number. For example, for the graph shown in Fig. 1a, we have $\boldsymbol{D}[A] = \{u_6, u_7, u_8\}$. Then, u_6 is the first vertex, u_7 the second, and u_8 the third in $\boldsymbol{D}[A]$. In this way, a vertex in G can be referred to as a pair (l, i), where l is a label, and i is the order number of the vertex in $\boldsymbol{D}[l]$. For example, u_6 can be represented as (A, 1). Let R be a relation in G^δ corresponding to a pair of vertex labels (l, l'). Denote by $R[1]$ and $R[2]$ all vertices in the first and the second columns, respectively. Then, all the vertices in $R[1]$ ($R[2]$) can be represented by a bit string s of length $|\boldsymbol{D}[l]|$ (resp. $|\boldsymbol{D}[l']|$) such that $s[i] = 1$ if the ith vertex of $\boldsymbol{D}[l]$ (resp. $\boldsymbol{D}[l']|$) appears in $R[1]$ (resp. $R[2]$). Otherwise, $s[i] = 0$. Let s, s' be the bit string for $R[1]$ and ($R[2]$), respectively. We call $S = [s \mid s']$ the signature of R, respectively, referred to as $R{\cdot}S[1] = s$ and $R{\cdot}S[2] = s'$.

Example 2 Continued with Example 1. In Fig. 3a, we show all the lists each associated with a label in G^δ. In Fig. 3b, we redraw the relations shown in Fig. 2b with each vertex replaced with its order number in the corresponding list. Their signatures are shown in Fig. 3c.

4.2.2 Counters

By using relation signatures, we are able to indicate whether a vertex appears in a column of a certain relation. However, the information on how many times it appears in

Author's personal copy

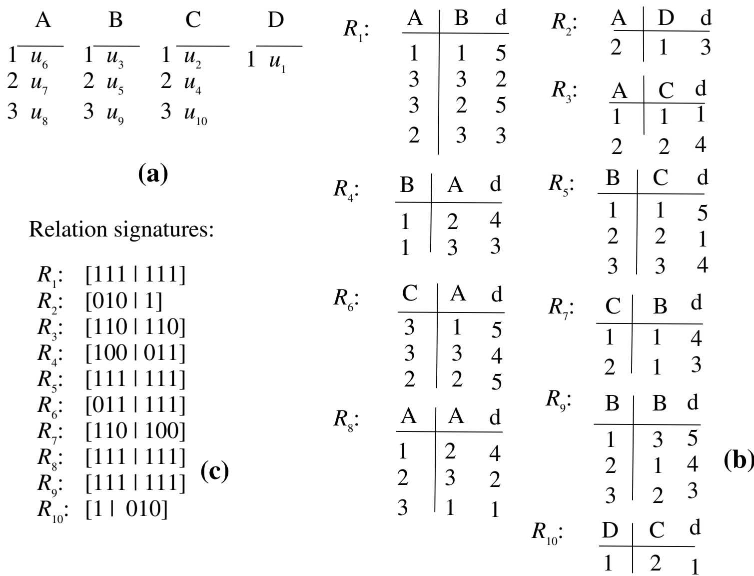


Fig. 3 Illustration for relation signatures

that column is missing. So, for each vertex u in a $\boldsymbol{D}[l]$, we will also associate it with a set of counters each for a column in a different relation, in which it occurs. Assume that u is a vertex appearing in the first column of some R_i. Then, it will have a counter, denoted as $u{\cdot}C_i$ [1], to record how many times it appears in that column of R_i. In general, if it appears in k columns (respectively, from different relations), it will be associated with k counters. For example, u_6 (represented by (A, 1)) appears in 6 columns: R_1 [1], R_3 [1], R_4 [2], R_6 [2], R_8 [1], and R_8 [2] (see Fig. 3b). Thus, u_6 will be associated with 6 counters: $u_6{\cdot}C_1$ [1] = 1, $u_6{\cdot}C_3$ [1] = 1, $u_6{\cdot}C_4$ [2] = 1, $u_6{\cdot}C_6$ [2] = 1, $u_6{\cdot}C_8$ [1] = 1, $u_6{\cdot}C_8$ [2] = 1. But u_8 (represented by (A, 3)) appears only in 5 columns. Thus, it has 5 counters: $u_8{\cdot}C_1$ [1] = 2, $u_8{\cdot}C_4$ [2] = 1, $u_8{\cdot}C_6$ [2] = 1, $u_8{\cdot}C_8$ [1] = 1, $u_8{\cdot}C_8$ [2] = 1.

Obviously, both relation signatures and counters for vertices can be established offline as part of indexes.

5 Relation filtering

In this section, we present our algorithm for relation filtering. First, we give the algorithm in 5.1. Then, we prove the correctness and analyze the computational complexity in 5.2.

5.1 Algorithm description

When a query Q with parameter δ' arrives, a simple way to evaluate it can be described as follows: First, we locate all the relevant relations (in G^{δ}). Then, for

Author's personal copy

each edge $(v_i, v_j) \in Q$, remove all those tuples $<u, u'>$ with $dist(u, u') > \delta'$ from the corresponding relation $R_{ij} = R(label(v_i), label(v_j))$. Next, we join such R_{ij}'s to form the final result.

However, we can do better by filtering all those tuples which cannot contribute to the final result before the joins are carried out. To this end, we need a new concept of *triangle consistency*.

Definition 5.1 (*Triangle consistency*) Let Q be a query with parameter δ'. Let (v_i, v_j) be an edge in Q. A tuple $t = <u, u'> \in R_{ij}$ in G^δ with $\delta \geq \delta'$ is said to be *triangle consistent* with respect to a vertex v_k $(k \neq i, j)$ in Q if for $R_{jk,}$ or R_{kj}, and R_{ik}, or R_{ki} in G^δ, one of the following conditions is satisfied:

1. if v_k is incident to v_j but not to v_i, then there exists u'' such that $<u', u''> \in R_{jk}$ if $(v_j, v_k) \in Q$, or $<u'', u'> \in R_{kj}$ if $(v_k, v_j) \in Q$;
2. if v_k is incident to v_i but not to v_j, then there exists u'' such that $<u', u''> \in R_{ik}$ if $(v_i, v_k) \in Q$, or $<u'', u'> \in R_{ki}$ if $(v_k, v_i) \in Q$;
3. if v_k is incident to both v_i and v_j, then there exists u'' such that

- $<u, u''> \in R_{ik}$ and $<u'', u'> \in R_{kj}$ if $(v_i, v_k), (v_k, v_j) \in Q$; or
- $<u, u''> \in R_{ik}$ and $<u', u''> \in R_{jk}$ if $(v_i, v_k), (v_j, v_k) \in Q$; or
- $<u'', u> \in R_{ki}$ and $<u'', u'> \in R_{jk}$ if $(v_k, v_i), (v_k, v_j) \in Q$; or
- $<u'', u> \in R_{ki}$ and $<u', u''> \in R_{kj}$ if $(v_k, v_i), (v_j, v_k) \in Q$.

In each of the above three cases, u'' is said to be consistent with t. The motivation of this concept is that if $<u, u'>$ in R_{ij} is not triangle consistent with some v_k $(k \neq i, j)$ in Q incident to v_i or v_j, it cannot be part of any answer to Q. Thus, only if $<u, u'>$ in R_{ij} is triangle consistent with any vertex in Q incident to v_i, v_j, or both, $<u, u'>$ can be possibly part of an answer. In this case, we say, $<u, u'>$ is triangle consistent. Notice that if neither v_i nor v_j is incident to any vertex in Q, all the tuples in R_{ij} are trivially considered to be triangle consistent.

The following example helps for illustration.

Example 3 Consider the query with parameter $\delta' = 5$ shown in Fig. 1b. To evaluate this query against the graph shown in Fig. 1a, we will first load three relations into main memory: R_1, R_5, and R_6 (shown in Fig. 3b) from G^δ. For illustration, we show the data in the three relations as a graph in Fig. 4a.

In this graph, the tuple represented by edge $(u_8, u_9) \in R_{12}$ $(= R_1$ shown in Fig. 3b) is triangle consistent with respect to v_3 in Q. But edge $(u_3, u_2) \in R_{23}$ $(= R_5$ shown in Fig. 3b) is not triangle consistent with v_1 since we have an edge (v_3, v_1) labeled with $<C, A>$ in Q, but we do not have an edge going from u_2 to a vertex in $\boldsymbol{D}(label(v_1)) = \boldsymbol{D}(\mathrm{A})$. (Note that edge (u_6, u_2) in Fig. 4a is just in the reverse direction of edge (v_3, v_1) in Q.)

Author's personal copy

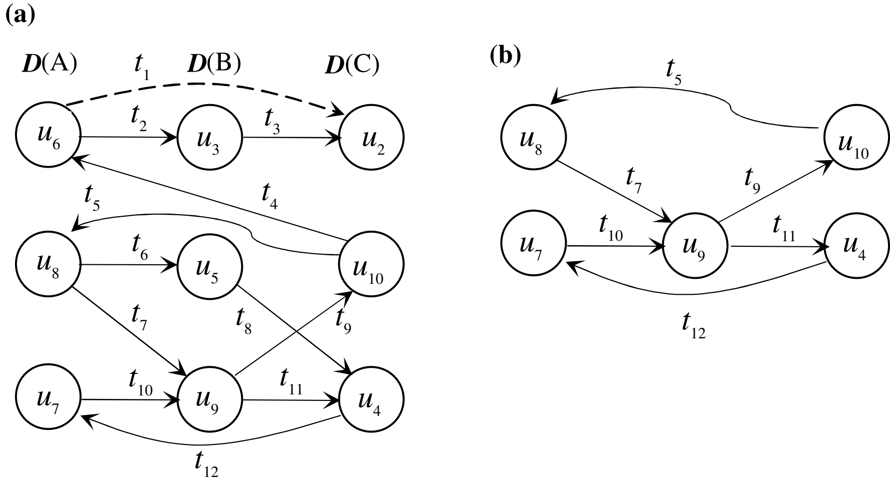


Fig. 4 Graph data stored in relations

In fact, only the tuples represented by the edges in the subgraph shown in Fig. 4b are triangle consistent, while all the other edges not in this subgraph are not. From this example, we can see that by a relation filtering the sizes of relations can be dramatically decreased. Our purpose is to find an efficient way to remove all the triangle inconsistent data from the relevant relations before the joins over them are actually performed.

In the following, we will first devise a procedure to check the triangle consistency of each tuple $t = <u, u'>$ in every R_{ij} with respect to a single vertex v_k ($k \neq i, j$) in Q. Then, a general algorithm for removing all the useless data will be presented.

Below is the formal description of these algorithms. Besides the data structures described in 4.2, each tuple t in every relation R_{ij} (corresponding to $(v_i, v_j) \in Q$) is additionally associated with two kinds of extra data structures for efficient computation:

α[t, k]—the number of vertices u in $\boldsymbol{D}[k]$ (for each $k \neq i, j$), each of which is triangle consistent with t. Initially, each α[t, k] = 0.

β[t]—a set containing all those vertices u in $\boldsymbol{D}[k]$ (for each $k \neq i, j$) such that each of them is triangle consistent with t. In β[t], each element is referred to as (k, u), indicating that it is a vertex in $\boldsymbol{D}[k]$. Initially, each β[t] = ϕ (empty).

In addition, a global variable L is used to store all the tuples which are found not triangle consistent with some v in Q, to facilitate the propagation of inconsistency.

Author's personal copy

Algorithm 1: *tcControl*(R_{ij}, v_k)

Input: R_{ij}, v_k.
Output: reduced R_{ij} such that each $<u, u'>$ in it is triangle consistent with respect to v_k.

1. **if** v_k is incident to v_j or v_j but not to both **then**
2. **if** $(v_j, v_k) \in Q$ **then** $R := R_{ij}$; $R' := R_{jk}$; $a := 2$; $b := 1$;
3. **if** $(v_k, v_j) \in Q$ **then** $R := R_{ij}$; $R' := R_{kj}$; $a := 2$; $b := 2$;
4. **if** $(v_i, v_k) \in Q$ **then** $R := R_{ij}$; $R' := R_{ik}$; $a := 1$; $b := 1$;
5. **if** $(v_k, v_i) \in Q$ **then** $R := R_{ij}$; $R' := R_{ki}$; $a := 1$; $b := 2$;
6. call *check*-1(R, R', a, b);
7. **if** v_k is incident to both v_i and v_j **then**
8. **if** $(v_i, v_k), (v_k, v_j) \in Q$ **then** *check*-2(i, j; i, k; k, j);
9. **if** $(v_i, v_k), (v_j, v_k) \in Q$ **then** *check*-2(i, j; i, k; j, k);
10. **if** $(v_k, v_i), (v_k, v_j) \in Q$ **then** *check*-2(i, j; k, i; k, j);
11. **if** $(v_k, v_i), (v_j, v_k) \in Q$ **then** *check*-2(i, j; k, i; j, k);
12. call Δ-*consistency*(i, j, k);

The above algorithm is a general control to check the triangle consistency for all the tuples in R_{ij} (relation for edge $(v_i, v_j) \in Q$) with respect to a certain vertex $v_k \in Q$. In general, we will distinguish between two cases: (1) v_k is incident only to one of the two vertices: v_i or v_j (see line 1), and (2) v_k is incident to both v_i and v_j (see line 7). For case 1, we further have four sub-cases (see lines 2, 3, 4, 5, respectively). For each of them, a sub-procedure *check*-1( ) is invoked (see line 6), in which three tasks will be performed:

- make a bit-wise *and* operation over the corresponding relation signatures,
- change the counters of the corresponding vertices according to the result of the bit-wise *and* operation, and
- change the relevant relation signatures themselves.

For case 2, we also distinguish among four sub-cases (see lines 8, 9, 10, 11, respectively). But for each of them, we will call a different sub-procedure *check*-2( ) to do the same tasks as *check*-1( ), but in different ways. Finally, in line 12, Δ-*consistency* will be invoked to check the triangle consistency involving v_i, v_j, and v_k.

Algorithm 2: *check*-1(R_1, R_2, a, b)

Input: R_1, R_2, a, b.
Output: reduced R_1, R_2 by removing un-consistent tuples.

1. $s := R_1.S[a] \wedge R_2.S[b]$;
2. $c := (a \bmod 2) + 1$; $d := (b \bmod 2) + 1$;
3. call *tuple-removing*(s, R_1, R_2, a, c, d);
4. call *tuple-removing*(s, R_2, R_1, b, d, c);

Author's personal copy

As mentioned above, in *check*-1( ), we will first calculate $s = R_1 \cdot S[b] \wedge R_2 \cdot S[a]$ (see line 1), where R_1 and R_2 represent two relations corresponding to two edges incident to a common vertex in Q. Then, in terms of s, we will make changes, respectively, with respect to R_1 and R_2 as described above by calling *tuple-removing*( ) given below.

Special attention should also be paid to the modulo operations given in line 2: $y = (x \bmod 2) + 1$, by which $y = 1$ if $x = 2$ and $y = 2$ if $x = 1$.

In the following algorithm, we use $R(l, a)$ to represent the value appearing in the l-th tuple, a-th column of relation R. For a tuple t, $t(i)$ $(i = 1, 2)$ stands for its i-th value.

Algorithm 3: *tuple-removing*(s, R, R', a, b, c)

Input: s, R, R', a, b, c.
Output: removing tuples from R according to R'.
1. assume that $R'[c]$ corresponds to v_k in Q;
2. **for** $l = 1$ to $|R|$ **do** $\{t := l$-th tuple of R;
3.   **if** $s[R(l, a)] = 1$ **then** $\{\alpha[t, k] :=$ number of tuples t' in R' with $t'(b) = R(l, a)$; append all such t's to $\beta[t]$;$\}$
4.   **else** $\{$
5. $L := L \cup \{t\}$; let $t = <x, y>$;
6.   **if** $y.C_R[b] > 0$ **then** $\{y.C_R[b]$ --; if $y.C_R[b] = 0$ then
7. $R.S[b][y] := 0;\}$
8.   **if** $x.C_R[a] > 0$ **then** $\{x.C_R[a]$ --; if $x.C_R[a] = 0$ then
9. $R.S[a][x] := 0;\}\}$

In *tuple-removing*(s, R, R', a, b, c), we will first eliminate inconsistent tuples from R according to s (lines 2–5). Then, for each removed $<x, y>$, we will change the counters associated with x, y, respectively (see lines 6 and 8). In particular, if the counter of a vertex becomes 0, the bit for that vertex in the corresponding relation signature will also be modified to 0 (see lines 7 and 9).

Algorithm 4: *check*-2$(i, j; r, l; p, q)$

Input: i, j, k, l, p, q.
Output: reduced R_{ij} according to R_{rl}, R_{pq}.
1. $R := R_{ij}$; $R' := R_{rl}$;
2. **if** $r = i$ **then** $\{a := 1; b := 1;\}$ **else** $\{a := 2; b := 1;\}$
3. call *check-1*(R, R', a, b);
4. $R := R_{ij}$; $R' := R_{pq}$;
5. **if** $p = j$ **then** $\{a := 1; b := 2;\}$ **else** $\{ a := 1; b := 2;\}$
6. call *check-1*(R, R', a, b);
7. $R := R_{rl}$; $R' := R_{pq}$;
8. **if** $r = i \wedge p = j$ **then** $\{a := 2; b := 2;\}$
9. **if** $r = i \wedge p \neq j$ **then** $\{a := 1; b := 2;\}$
10. **if** $r \neq i \wedge p = j$ **then** $\{a := 1; b := 1;\}$
11. **if** $r \neq i \wedge p \neq j$ **then** $\{a := 1; b := 1;\}$
12. call *check-1*(R, R', a, b);

Author's personal copy

In *check*-2( ), we need to do three bit-wise *and* operations, with each corresponding to a pair of joint edges within a triangle (see lines 1, 4, and 7). Each of them can be simply done by calling *check*-1( ) (see lines 3, 6, and 12). But we should notice the difference between the third case (line 7) and the first two cases (line 1 and line 4). For the third case, we need to distinguish among 4 sub-cases (lines 8–11), while for each of the first two cases only two sub-cases (see lines 2 and 5) need to be handled.

Now, we give the formal description of Δ-*consistency*(i, j, k), which is invoked in line 12 in *tcControl*( ). For simplicity, however, we only show the algorithm for the case (v_j, v_k), $(v_k, v_i) \in Q$. For this, we will check, for each tuple $<u', u''> \in R_{jk}$, whether there exists l with $s[l] = 1$ such that $<l, u'> \in R_{ij}$, $(u'', l) \in R_{ki}$, where $s = R_{ij} \cdot S[2] \wedge R_{jk} \cdot S[1]$. For the other three cases (i.e., (v_i, v_k), $(v_j, v_k) \in Q$, (v_i, v_k), $(v_j, v_k) \in Q$, (v_k, v_i), $(v_k, v_j) \in Q$), a similar process can be established for each of them.

Algorithm 5: Δ-*consistency*(i, j, k)

Input: i, j, k.
Output: reducing R_{ij}, R_{ki}, R_{ki} according to Δ-*consistency*.
1. $s := R_{ij}.S[2] \wedge R_{jk}.S[1]$.
2. **for** $l = 1$ to $|s|$ **do** {
3.   **if** $s[l] = 1$ **then**
4.   **for** each pair of tuples: $<u', l> \in R_{ij}$, $<l, u''> \in R_{jk}$ **do**
5.   **if** $<u'', u'> \in R_{ki}$ **then** {$t_1 := <u', l>$; $t_2 := <l, u''>$; $t_3 := <u'', u'>$;
6.   add (k, u'') to $\beta[t_1]$; (i, u') to $\beta[t_2]$; (j, l) to $\beta[t_3]$;
7. $\alpha[t_1, k]$ ++; $\alpha[t_2, i]$ ++; $\alpha[t_3, j]$ ++; }}
8. **for** each $t \in R_{ij} \cup R_{ki} \cup R_{ki}$ with $\alpha[t, x] = 0$ **do**
9.   {let $t = <u, u'>$; $L := L \cup \{t\}$;
10.   **if** $t \in R_{ij}$ **then** {remove t from R_{ij}; $R := R_{ij}$;}}
11.   **if** $t \in R_{jk}$ **then** remove t from R_{jk}; $R := R_{jk}$;}}
12.   **if** $t \in R_{ki}$ **then** {remove t from R_{ki}; $R := R_{ki}$;}
13. $u.C_R[1]$ --; if $u.C_R[1] = 0$, change $R.S[1][u]$ to 0;
14. $u'.C_R[2]$ --; if $u'.C_R[2] = 0$, change $R_R.S[2][u']$ to 0;
15.   }

The above algorithm mainly comprises two steps. In the first step (lines 1–7), we create $s := R_{ij} \cdot S[2] \wedge R_{jk} \cdot S[1]$ and then scan s bit by bit. For each $s[l] = 1$, we will check, for each pair of tuples: $<u', l> \in R_{ij}$ and $<l, u''> \in R_{ij}$, whether $<u'', u'>$ appears in R_{ij}. If such a tuple exist, α[] and β[] will be accordingly changed (lines 6–7). In the second step (lines 8–15), for each $\alpha[t, x] = 0$ (for some x) we remove t from the corresponding relation (see lines 10–12) and accordingly change relevant counters and relation signatures (see lines 13–14).

Example 4 Applying Δ-*consistency*(2, 3, 1) to the three edges of Q shown in Fig. 1b against the δ-transitive closure shown in Fig. 3b, three relations: R_5, R_6, and R_1 will be loaded into main memory. The following computation will be conducted. (Since

Author's personal copy

R_5, R_6, and R_1 correspond to the three edges in Q, they are also referred to as R_{23}, R_{31}, and R_{12}, respectively.)

1. $s{:}= R_{23}{\cdot}S[2] \wedge R_{31}{\cdot}S[1] = R_5{\cdot}S[2] \wedge R_6{\cdot}S[1] = 111 \wedge 011 = 011$.
2. $s[2] = 1$. For $t_8 = <2, 2> (<u_5, u_4>) \in R_{23} = R_5$ and $t_{12} = <2, 2> (<u_4, u_7>) \in R_{31} = R_6$, we will check whether $<2, 2> (<u_7, u_5>)$ is in $R_{12} = R_1$. Since $<2, 2> (<u_7, u_5>) \notin R_{12}$, lines 6 and 7 will not be executed and therefore $\beta[t_8] = \beta[t_{12}] = \phi$ and $\alpha[t_8, 1] = \alpha[t_{12}, 2] = 0$.

For $t_{11} = <3, 2> (<u_9, u_4>) \in R_{23} = R_5$ and $t_{12} = <2, 2> (<u_4, u_7>) \in R_{31} = R_6$, we will check whether $<2, 3> (<u_7, u_9>)$ is in $R_{12} = R_1$. Since $t_{10} = <2, 3> (<u_7, u_9>) \in R_{12}$, lines 6 and 7 will be executed and therefore $\beta[t_{11}] = \{(1, u_7)\}$, $\beta[t_{12}] = \{(2, u_9)\}$, $\beta[t_{10}] = \{(3, u_4)\}$; and $\alpha[t_{11}, 1] = \alpha[t_{12}, 2] = \alpha[t_{10}, 3] = 1$.

3. $s[3] = 1$. For $t_9 = <3, 3> (<u_9, u_{10}>) \in R_{23} = R_5$ and $t_5 = <3, 3> (<u_{10}, u_8>) \in R_{31} = R_6$, we will check whether $<3, 3> (<u_8, u_9>)$ is in $R_{12} = R_1$. Since $t_7 = <3, 3> (<u_8, u_9>) \in R_{12}$, lines 6 and 7 will be executed and therefore $\beta[t_9] = \{(1, u_8)\}$, $\beta[t_5] = \{(2, u_9)\}$, $\beta[t_7] = \{(3, u_{10})\}$; and $\alpha[t_9, 1] = \alpha[t_5, 2] = \alpha[t_7, 3] = 1$.

After the above steps, we must have:

$\alpha[t_5, 2] = 1$ $\beta[t_5] = \{(2, u_9)\}$ $\alpha[t_2, 3] = 0$ $\beta[t_2] = \phi$
$\alpha[t_7, 3] = 1$ $\beta[t_7] = \{(3, u_{10})\}$ $\alpha[t_3, 1] = 0$ $\beta[t_3] = \phi$
$\alpha[t_9, 1] = 1$ $\beta[t_9] = \{(1, u_8)\}$ $\alpha[t_4, 2] = 0$ $\beta[t_4] = \phi$
$\alpha[t_{10}, 3] = 1$ $\beta[t_{10}] = \{(3, u_4)\}$ $\alpha[t_6, 3] = 0$ $\beta[t_6] = \phi$
$\alpha[t_{11}, 1] = 1$ $\beta[t_{11}] = \{(1, u_7)\}$ $\alpha[t_8, 1] = 0$ $\beta[t_8] = \phi$
$\alpha[t_{12}, 2] = 1$ $\beta[t_{12}] = \{(2, u_9)\}$

Then, all those tuples, whose α-values equal 0, will be first stored in L (see line 9) and then checked to propagate inconsistency before they are finally removed (see lines 10–13). Accordingly, the counters of the corresponding vertices and also possibly relation signatures will be changed (see lines 13–14).

In this way, the graph shown in Fig. 4a will be reduced to the graph shown in Fig. 4b.

Based on *tcControl*(), a general algorithm for the relation filtering can be easily designed. It works in two phases.

Author's personal copy

Algorithm 6: *rFiltering*(Q, R_{ij}'s)

Input: Q.
Output: Δ-consistent R_{ij}'s.
1. **for** each $(v_i, v_j) \in Q$ **do** {
2.   **for** each $k \neq i, j$ **do** {
3.   call *tcControl*(R_{ij}, v_k);
4.   }
5. }
6. **while** L is not empty **do** {
7.   choose t from L and remove t from L;
8.   assume that $t = <x, y> \in R_{ij}$;
9.   **for** each $(k, z) \in \beta[t]$ **do** {
10.   assume that $t_1 = <y, z>$ and $t_2 = <z, x>$;
11. $\alpha[t_1, i]$ --; remove $(i, x) \in \beta[t_1]$;
12.   **if** $\alpha[t_1, i] = 0$ **then** $L := L \cup \{t_1\}$;
13. $\alpha[t_2, j]$ --; remove $(j, y) \in \beta[t_2]$;
14.   **if** $\alpha[t_2, j] = 0$ **then** $L := L \cup \{t_2\}$;
15. }

In the first phase (lines 1–5), we check the triangle consistency for each edge in Q. In the second phase (lines 6–14), we propagate inconsistency among all those triangles which share one edge. To see this, we consider a larger query shown in Fig. 5a. Obviously, by checking the consistency with respect to triangle $\mathbf{\Delta}_{143}$ tuple $t_5 \in R_{31}$ (=R_6 shown in Fig. 3c) will be removed. Then, (2, u_9) in $\beta[t_5]$ will be checked (see line 9), leading to the elimination of $t_7 = <u_8, u_9>$ from R_{12} (=R_1) and $t_9 = <u_9, u_{10}>$ from R_{23} (=R_5) see lines 11–14). For this query, only five edges $<u_7, u_9>$, $<u_9, u_4>$, $<u_4, u_7>$, $<u_7, u_1>$, and $<u_1, u_4>$ (in Fig. 2b) will survive the relation filtering. (See Fig. 5b for illustration.)

5.2 Correctness and computational complexity

In this section, we prove the correctness of the algorithm and analyze its computational complexity.

Lemma 1 *Let* (v_i, v_j) *be an edge* $\in Q$. *Assume that* v_j *is a vertex incident to* v_i *or* v_j, *but not to both of them. Let* $<u, u'> \in R_{ij}$ *be an tuple surviving the relation filtering. Then, there exists at least a tuple* $<u', u''>$ *in* R_{jk} *(or* $<u'', u'>$ *in* R_{kj}*); or tuple* $<u, u'>$ *in* R_{ik} *(or* $<u', u>$ *in* R_{ki}*).*

Proof The lemma holds in terms of Algorithm *check-1*(). □

Lemma 2 *Denote by* R_{ij} *the relation corresponding to* $(v_i, v_j) \in Q$. *Let* R_{ij}' *be the reduced relation of* R_{ij} *by applying rFiltering() to* Q *and all the relevant relations. Then, for each* $<u, u'> \in R_{ij}'$ *and each* $v_k \in Q$ $(k \neq i, j)$, *there exists at least a vertex* u'' *satisfying the conditions stated in Definition 5.1*

Author's personal copy

Proof We prove the lemma for the case (v_j, v_k), $(v_k, v_i) \in Q$. The correctness for the other cases can be established in a similar way. In that case, we will first call *check*-2($i, j; k, i; j, k$) (see line 11 in *tcControl*()) to remove all those tuples <u_1, u_2> from R_{jk} such that u_1 does not appear in $R_{ij}[2]$ or u_2 does not appear in $R_{ij}[1]$. The same claims apply to the tuples in both R_{ki} and R_{ij}. Then, by executing Δ-*consistency*(i, j, k), we further remove all those tuples from R_{ij}, R_{ki}, and R_{jk}, which are not triangle consistent. □

Proposition 1 *Let R_{ij} be a relation reduced by executing rFiltering(Q, R_{ij}'s). A tuple t* = *<u, u'> is in R_{ij} if and only if t is triangle consistent.*

Proof *If part*. If a tuple is triangle consistent, it will definitely not be removed according to Lemma 1, Lemma 2, and the inconsistency propagation process (the second phase, lines 6–14) in *rFiltering*().

Only-if part. Any tuple which is not triangle consistent will be removed by check-1( ), check-2( ), Δ-*consistency*( ), or through the inconsistency propagation done in the second phase in *rFiltering*( ). □

The time complexity of the algorithms *rFiltering*( ) can be easily analyzed.

We mainly estimate the cost of *tcControl*( ), which comprises three parts: the running time of *check*-1( ), *check*-2( ), and Δ-*consistency*, respectively, referred to as c_1, c_2, and c_3. Denote by r and s the largest size among all R_{ij}'s, and the maximum length of relation signatures, respectively. First, we have $c_1 = O(\max\{r, s\})$. It is because in *check*-1( ) we have only a bit-wise *and* operation over two relation signatures and one **for**-loop done in *tuple-removing*( ), which needs only $O(r)$ time.

Since in *check*-2( ) we have only three calls of *check*-1( ), c_2 is also bounded by $\max\{r, s\}$.

Let d be the largest degree of a vertex in G^δ. Then, the cost of line 1 in Δ-*consistency* is bounded by $O(s \cdot d^2)$. So, we have $c_3 = O(s \cdot d^2)$. Since in *rFiltering*() we will call *tcControl*( ) $O(n \cdot m)$ times. The total cost of the whole working process is bounded by

$$O(n \cdot m(c_1 + c_2 + c_3)) = O(n \cdot m \cdot s \cdot d^2).$$

The cost for the inconsistency propagation is also bounded by $O(n \cdot m \cdot s \cdot d^2)$ since we can put at most so many elements into L.

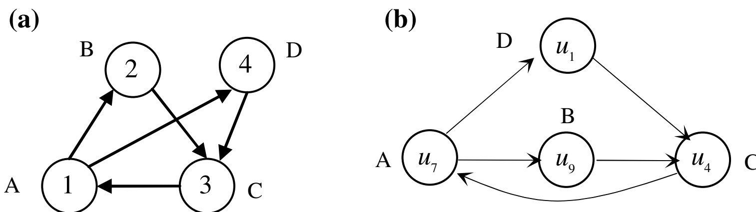


Fig. 5 A larger query and filtering results

Author's personal copy

Now we estimate the space complexity of the algorithm. First, we need a space to accommodate all the relevant relations from G^{δ}, and their signatures, as well as the vertex counters. This part of space is obviously bounded by $O(mr)$. The dominant space requirement is due to the storage of β[t]'s, which is bounded by:

$$\sum_{v_i \in Q} mr|D_i| \leq nmr|D|, \tag{3}$$

where D_i is $\boldsymbol{D}[l(v_i)]$ and $D = \max_i\{\boldsymbol{D}[l(v_i)] \mid v_i \in Q\}$. It is because for each t in a relation R_{ij} β[t] can have up to O(|D_k|) elements for each $v_k \in Q$. However, the space for storing α[t, k]'s is much less, which is bounded by

$$\sum_{v_i \in Q} mr \leq nmr, \tag{4}$$

since corresponding to each t we can have at most n α-values each for a vertex in Q. Therefore, the total space overhead is bounded by O(nmr|D|).

6 On join ordering

After the relation filtering, we will join all the reduced R_{ij}'s to get the final results. To do this efficiently, we need first to determine an optimal order of joins.

6.1 Cost model

It is well known that different orders will lead to different performances. But the join order selection is normally based on the cost estimate of every join involved, which needs to know both the cardinality of each R_{ij} and the selectivity of each join, where R_{ij} is the relation after the relation filtering is conducted. We assume that intermediate results should be stored in a temporary table on disk. We also use B_{ij} to represent the number of pages on disk that hold tuples of the relation R_{ij}. Thus, the cost of joining R_{ij} and R_{jk} can be estimated as follows [40]:

$$C = B_{ij} \cdot \left(\sigma_{ijk} \cdot |R_{jk}| + \frac{B_{jk}}{M-1} \right) + B_{jk} \cdot (\sigma_{ijk} \cdot |R_{ij}| + 1), \tag{5}$$

where M is the number of pages available in main memory, and σ_{ijk} is the selectivity of the join between R_{ij} and R_{jk}, calculated as follows:

$$\delta_{ijk} = \frac{\sum_{u \in D_j} u \cdot C_{ij}[2] \cdot u \cdot C_{ij}[1]}{\left|R_{ij}\right| \cdot \left|R_{jk}\right|}. \tag{6}$$

Here, $u{\cdot}C_{ij}[x]$ is a counter of u to record the number of its appearances in the x-th column in R_{ij}.

Author's personal copy

In (5), we assume that R_{jk} is the smaller relation and is handled as the inner operand while R_{ij} as the outer operand. In fact, this cost comprises two parts. The first part is for the join computation:

$$B_{ij} + \frac{B_{ik}}{M-1} + B_{jk}. \tag{7}$$

The second part is for storing the result:

$$B_{ij} \cdot \sigma_{ijk} \cdot \left|R_{jk}\right| + B_{jk} \cdot \sigma_{ijk} \cdot \left|R_{ij}\right|. \tag{8}$$

6.2 Join order selection

The join order selection can be performed by adopting the traditional dynamic programming algorithm [41] using the cost model described above. However, this solution is inefficient since a very large solution space has to be searched, especially when $|E(Q)|$ is relatively large. So, we will modify the *linearizing* algorithm proposed in [42] for this task, but with the above cost model being used.

We will handle Q as an undirected graph Q' and explore Q' in the depth-first manner to generate a spanning tree (or forest) T. With respect to T, all the edges in Q' will be classified into *tree edges* and *forward edges*. Any tree edge is an edge on T. Otherwise, it is a forward edge. The order of all joins between relations represented by the tree edges can be determined by using the *linearizing* algorithm proposed in [42]. But we need to incorporate the forward edges into the estimation of the costs and sizes of intermediate results.

Let v_i be a vertex in T. Let $v_{i1}, \ldots, v_{ik}$ be the children of v_i. Assume that we have a forward edge from v_i to a vertex v_l in $T[v_{ij}]$ (sub-tree rooted at v_{ij}, $1 \le j \le k$. See Fig. 6 for illustration).

Denote by s_{ij} the estimated size of the intermediate result by joining all the relations in $T[v_{ij}]$. Then, to estimate the cost and the size for $T[v_i]$ (in a way suggested in [42]), s_{ij} should be changed to $\gamma \cdot s_{ij}$, where

$$\gamma = \frac{\sum_{u \in D_l} u \cdot C_{il}[2]}{\left|D_i\right| \cdot \left|D_l\right|}. \tag{9}$$

In general, assume that there are r forward edges from v_i to r different vertices in $T[v_{ij}]$. Then, s_{ij} should be changed to $\gamma_1 \ldots \cdot \gamma_r \cdot s_{ij}$, where γ_p ($p = 1, \ldots, r$) is the γ-value for the p-th forward edge from v_i to a vertex in $T[v_{ij}]$. Each forward edge also corresponds to a join operation. Then, we will form a subsequence of joins for these forward edges, sorted increasingly by their γ-values and insert it just after the subsequence generated for $T[v_{ij}]$ in the whole join sequence generated for T.

Author's personal copy

7 Experiments

In our experiments, we have tested altogether three different methods:

- *2-hop-based* [12] (*2Hb* for short),
- *LLR-embedding-based* [13] (*LLRb* for short),
- *Our method* discussed in this paper.

All the three methods have been implemented in C++, compiled by GNU make utility with optimization of level 2. In addition, all of our experiments are performed on a 64-bit Ubuntu operating system, run on a single core of a 2.40-GHz Intel Xeon E5-2630 processor with 32 GB RAM.

We divide the experiments into two groups. In the first group, we test all the methods against real data, while the second group is on synthetic data. The first group is further divided into three parts. In the first part, we report the offline time for constructing indexes over all the real graphs for all the three methods. In the second part, we compare the query times of the three methods over the real graphs for different query patterns. In the third part, we study in depth the performance using a specific query pattern, but with different δ values.

The second group is also further divided into three parts. As with the first group, we will first measure the indexing time for two types of synthetic graphs: *Erdos–Renyi model* (*ER*) and *Scale-Free model* (*SF*). Then, in the second, we vary the number of vertices and δ values in order to study the effect of these two parameters. Finally, in the third part, we study the effect of the number of labels in a graph.

Datasets We used both real and synthetic datasets. Table 2 provides an overview of the real datasets used in the experiments. These datasets have been taken from various sources. Among them, **yeast** is a protein-to-protein interaction network in budding yeast, taken from vlado (http://vlado.fmf.uni-lj.si/pub/networks/d-atas/); **citeseer** is from KONECT (http://konect.uni-koblenz.de/ networks/citeseer); all the other real graphs are taken from SNAP (http://snap.stanford.edu/data/index.html). Six of the graphs, e.g., **webStanford**, already have natural labels for vertices. The last column indicates whether or not we synthetically generated labels. In addition,

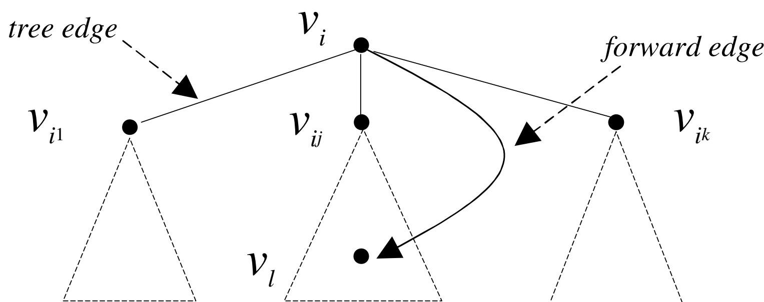


Fig. 6 Illustration for forward edges

Author's personal copy

six graphs are also undirected. For them, by the computation of the shortest distance between two vertices, the direction is not considered.

All these real graphs are not weighted. So, we choose six of them and assign each edge in them a weight = 1. In all the remaining graphs, each edge is randomly assigned a number from {1,…, 10} as its weight.

Two types of synthetic datasets are used in our experiments. One is the *ER*-model. It is a random graph of |*V*| vertices and |*E*| edges. To create a set of edges, we randomly select vertex *u* and vertex *v* from set *V*. Then, a directed edge is created from *u* to *v* and a random label from a label set Σ is assigned to each of them. When creating edges, it is guaranteed that an edge is not repeatedly generated, and the weight assigned to it is different from its parent edge if multiple weights are used for a graph. This method simulates many real-world problems and may contain many large *strongly connected components* (*SCC*s). For all the created *ER* graphs, |Σ| is set to 10. The second dataset is the *SF*-model. It is another random graph satisfying the power law distribution of vertex out-degrees *d*: $P(d) = \alpha d^{-l}$, where α and *l* are two constants. This dataset is created by the graph generator *gengraphwin* (http://fabien.viger.free.fr/liafa/generation/). For the SF graphs, we fix α to 1, but set *l* to different values (either 2.2, 2.4, 2.6, or 2.8) to change the distribution of vertices' out-degrees. We notice that the larger *l* is, the smaller the number of edges in a graph. To study the scalability in these two kinds of graphs, we vary graph size from 100 k to 400 K vertices. In addition, the density |*E*|/|*V*| of the ER graphs varies from 2 to 5, and the density of the SF graphs varies by changing the *l*-value.

Pattern queries In the experiments, we have used two kinds of pattern queries: trees and graphs as shown in Fig. 7. They do look somehow arbitrarily selected, but for our method tree pattern queries are quite different from graph patterns since for tree patterns only *check*-1( ) is useful while for graph patterns all the procedures *check*-1( ), *check*-2( ), and Δ-*consistency*( ) work together to eliminate useless

Table 2 Overview of all real datasets

Data graph	|V|	|E|	|Σ|	Directed	Weight	Syn. label
Yeast	2361	7182	12	No	1	No
wikiVote	7115	103,689	100	Yes	1	Yes
citeHepph	34,546	421,578	124	Yes	1–10	No
webStanford	281,903	2,312,497	100	Yes	1	Yes
comDBLP	317,080	1,049,866	13,477	No	1	No
webNotreDame	325,729	1,497,134	100	Yes	1–10	Yes
citeseer	384,413	1,751,463	500	Yes	1	Yes
webBerkStan	685,230	7,600,595	500	Yes	1	Yes
webGoogle	875,713	5,105,039	500	Yes	1–10	Yes
roadNetPA	1,088,092	1,541,898	500	No	1	Yes
roadNetTX	1,379,917	1,921,660	500	No	1–10	Yes
citePatterns	3,774,768	16,518,948	1,1319	Yes	1–10	No
wiki-topcats	1,791,489	28, 511,807	500	Yes	1–10	Yes

Author's personal copy

tuples. Therefore, we expect that more speedup can be obtained for graph pattern queries than for trees, due to more checked constraints.

7.1 Tests on real data

In this subsection, we report the test results on real data. First, we show the indexing time in Sect. 7.1.1. Then, we show the query time in Sect. 7.1.2. In Sect. 7.1.3, we study in depth the performance by testing a specific query pattern against all the real graphs, but with different δ values.

7.1.1 Indexing time and space

The indexing time and space of our method for all the real graphs are shown in Table 3. From this, we can see that for different δ values, both indexing time and space for storing indexes are different. In general, as the δ-value increases, both time and space requirements are gradually enlarged.

In Table 4, we show the indexing time and space of the 2-hop-based method [12] and the LLR-embedding-based method. From this, we can see that more time and space are needed by the LLR embedding than the 2-hop. It is because to transform 2-hop covers into vectors some extra time and space are required. In particular, after the vectors are created by the LLRb, the 2-hop covers cannot be simply discarded since they have to be kept for the verification of final join results.

Comparing Table 4 with Table 3, we can also see that in general much time and space are needed by the 2Hb (and the LLRb) than ours. In addition, for both *road-NetTX* and citePatterns, we have tried to establish their 2-hop covers. But within 3 days we are not able to get results.

7.1.2 Query time over tree patterns and graph patterns

In this subsection, we show the query time of all the tested methods over the real graphs. In Figs. 8 and 9, we show average times on all the four tree pattern queries shown in Fig. 8a and the four graph pattern queries shown in Fig. 8b, respectively. From these two figures, we can see that our method uniformly outperforms both the 2Hb and LLRb, for the following two reasons:

1. By the 2Hb and LLRb, relations have to be constructed on-the-fly from 2-hop covers, or from vectors which are built by mapping a 2-hop cover to a vector space

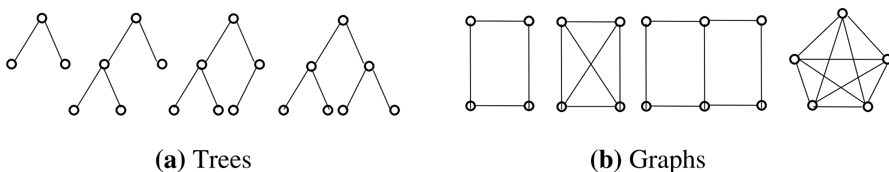


Fig. 7 Tested queries

Author's personal copy

Table 3 Indexing time and size of real data by ours

Graph	δ	Time (s)	Size (M)	δ	Time (s)	Size (Mb)	δ	Time (s)	Size (Mb)	δ	Time (s)	Size (Mb)
Yeast	2	0.01	0.1	4	0.05	1.8	6	0.21	10.8	8	0.825	29
wikiVote	2	0.25	1.4	4	031	22.1	6	1.6	85.7	8	3.7	118.4
citHepph	5	1.9	5.6	10	2.6	37.8	15	14.4	123	20	33.1	313.8
webStanford	2	2.0	32.9	4	2.8	222	6	5.1	258	8	13.5	402.9
comDBLP	2	53.4	18.1	4	69.8	38.1	6	185	452	8	299	77
webNotreDame	5	1.1	18.2	10	1.6	56.8	15	5.8	138	20	13.7	234.3
citeseer	2	56.7	29.6	4	66.5	251	6	137	1800	8	260.6	1910.6
webBerkStan	2	12.6	116.4	4	15.1	415	6	27.0	830	8	46.3	1525.6
webGoogle	5	27.8	75.0	10	39.6	374	15	117.2	1300	20	271	4001.2
roadNetPA	2	1.0	28.4	4	1.3	146.3	6	4.3	297.9	8	10	476.9
roadNetTX	5	4.2	17.7	10	6.5	44.8	15	18.2	84.6	20	51.1	140.8
citePatterns	5	9.56	109	10	13.4	369.6	15	23.7	893	20	88.4	1863.8
wiki-topcats	5	21	156.2	10	24.9	523.5	15	36.6	904.1	20	143.3	2341.3

Author's personal copy

Table 4 Indexing time and size of real graphs by 2Hb and LLRb

Graph	2Hb		LLRb	
	Time (s)	Size (Mb)	Time (s)	Size (Mb)
Yeast	1.1	30	1.1 + 1.2	30 + 2
wikiVote	22.2	386.8	22.2 + 19.8	386.8 + 2.9
citHepph	120	470.0	120 + 91.1	470 + 3.9
webStanford	130	1000.0	130 + 124.3	1000 + 5.1
comDBLP	600	61.7	600 + 412.5	61.7 + 7.565
webNotreDame	300	1000.0	300 + 254.1	1000 + 7.59
citeseer	3100	8000.0	3100 + 1613.2	8000 + 7.79
webBerkStan	4530	10,000.0	4530 + 2162.7	10,000 + 8.5
webGoogle	700	12,000.0	700 + 228.3	12,000 + 8.9
roadNetPA	2200.24	27,023.0	2200.24 + 16,797	27,023 +120.0
roadNetTX	F	F	F	F
citePatterns	F	F	F	F
wiki-topcats	F	F	F	F

F indiates that the tests on the corresponding graphs fail to produce the results

using the LLR embedding while by ours the relations are directly loaded from hard disk.

2. By our method, the relation filtering can greatly reduce the size of relations before they take part in joins.

In addition, we notice that the LLRb can be much better than the 2Hb since by using LLR embeddings not only a relation can be established more quickly than by using 2-hop covers, but also less time is spent to do joins by checking Chebyshev distances and clustering points of a vector space, which is constructed from a 2-hop cover by the LLR embedding.

Finally, by comparing Figs. 8 and 9, we can see that the time used for evaluating a tree pattern query is generally much larger than for evaluating a graph pattern query of the same vertices. It is because the number of matching subgraphs to a tree

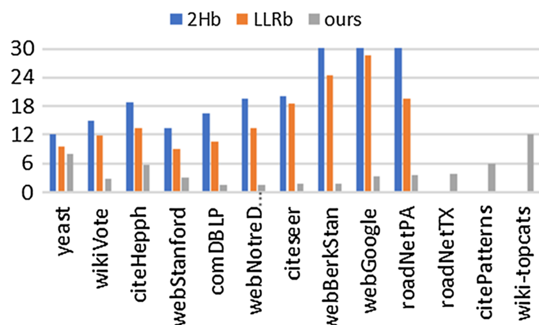


Fig. 8 Time(s) for evaluating tree pattern queries

Author's personal copy

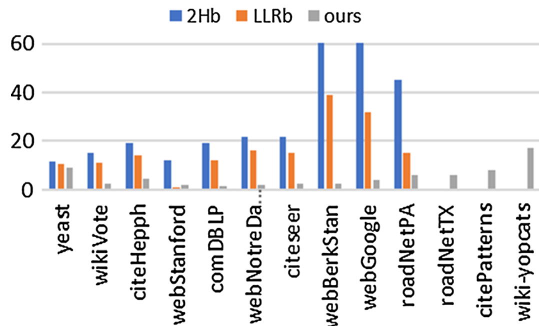


Fig. 9 Time(s) for evaluating graph pattern queries

pattern is normally much larger than the number of matching subgraphs to a graph pattern, as shown in Table 5.

From Table 5, we cannot only see that the space overhead for evaluating the two different kinds of queries, but also the power of tuple filtering based on the triangle consistency checking.

7.1.3 Query time over a specific query pattern

In order to deliver a deep insight into the performance of all the tested methods, we report here a breakdown of their query time for evaluating a specific query pattern, which is a complete graph containing five vertices shown in Fig. 7b. The test results are shown in Table 6 for small graphs and in Table 7 for large graphs. From these two tables, we can see that for both the 2Hb and LLRb, their query time each comprises two parts: RC time (for constructing relations from 2-hop covers or vectors) and NJ time (for doing natural joins to get final results.) For ours, the query time is made up of two parts: RF (for relation filtering) and NJ time. We see that the relation filtering itself does not need much time, but can dramatically reduce the time for the subsequent joins for almost all the tested graphs. In general, they can get 2 times to 10 times speedup.

In Table 7, we show the test results for large graphs.

7.1.4 Tests on large δ

For small δ, our method is obviously superior to the other strategies. However, for large δ, the performance of our method will definitely be degraded due to the large size of generated δ-transitive closures. To observe how fast the performance is degenerated as the value of δ increases, we have tested a specific graph *roadNetPA* by setting the weight of each edge being 1, but changing δ from 8 to 24. The result is shown in Table 8.

From Table 8, we can see that for large δ both the indexing time and index space by our method are significantly increased. However, they are still somehow better than the 2Hb and LLRb (see Table 4). The reason for this is that the generated δ-transitive closures are in general smaller than the whole transitive closures that

Author's personal copy

Table 5 Space used by our method

Graph	Size (number of tuples)			
	Used for graph patterns		Used for tree patterns	
	Before filtering	After filtering	Before filtering	After filtering
yeast	44,442	22,049	125,000	62,470
wikiVote	6223	5774	10,461	9794
citHepph	7011	4347	15,417	8911
webStanford	3214	632	6628	2407
comDBLP	12,926	3257	120,904	58,617
webNotreDame	1886	66	4296	170
citeseer	4268	504	28,259	18,021
webBerkStan	2782	529	5449	1342
webGoogle	3359	843	9161	4397
roadNetPA	71,500	1113	125,004	5405
roadNetTX	18,200	42	29,860	131
citePatterns	44,625	446	92,373	2818
wiki-topcats	56,741	2031	107,776	4023

have to be created by them. The query time is also incremented, but mitigated to some extent by using the tuple filtering.

7.2 Tests on synthetic data

In our second experiment, we compare the performance of our method with the 2Hb and LLRb on some synthetic data graphs. We choose $n = 100{,}000$ vertices for the ER graphs, but vary the number of edges m from 200,000 up to 1,000,000. Accordingly, the largest degree of a vertex in ER graphs increases from 2 to 10. For the SF graphs, we also choose 100,000 vertices, but set l to different values 2.2, 2.4, 2.6, or 2.8 with $\alpha = 1$ to generate different probabilistic distributions $P(d) = \alpha d^{-l}$ of out-degrees d of vertices. For simplicity, the weight of any edge in both ER graphs and SF graphs is set to 1.

Our aim here is to study the impact of graph density on performance using two significantly different synthetic graph models, as density is a basic property of graphs. We expect that building indexes on denser graphs will cost relatively more time and space for all methods, as the number of possible paths to explore increases with density.

In Fig. 10a and b, we show the indexing time and space for the ER graphs, where $G(i)$ represents a δ-transitive closure with δ-value set to i. From this, we can see that the LLRb needs a little bit more time to establish indexes than the 2Hb. But its index size is smaller than the 2Hb's. However, for $\delta \leq 8$, both the construction time and the sizes of δ-transitive closures are much smaller than both the 2Hb's and LLRb's.

Author's personal copy

Table 6 Query time for a specific query pattern over small graphs

Graphs	δ	2Hb		LLRb		Ours	
		RC (s)	NJ (s)	RC (s)	NJ (s)	RF (s)	NJ (s)
Yeast	2	0.098	0.0736	0.045	0.054	0.05	0.03
	4	1.0	1.6	0.21	1.31	0.047	0.082
	6	1.2	9.9	1.1	5.0	1.35	3.24
	8	1.6	10.5	2.3	7.2	2.8	6.3
wikiVote	2	0.121	0.0015	0.014	0.0015	0.001	0.01
	4	1.1	0.106	0.112	0.106	0.011	0.047
	6	1.5	9.94	1.38	8.9	0.071	1.157
	8	3.3	11.65	2.94	9.0	0.151	2.6
citHepph	5	1.2	18.7	0.6	18.7	0	0
	10	5.1	112	2.1	112	0.032	0.011
	15	8.2	2.4	3.9	1.4	0.056	0.252
	20	10.6	8.3	7.6	6.3	0.119	4.4
webStanford	2	2.2	3.0	1.9	3.0	0.0003	1.0
	4	4.7	30.7	3.5	30.7	0.00056	1.32
	6	7.1	5.2	14	4.2	0.00071	1.45
	8	7.7	6.0	3	5.0	0.00090	2.2
comDBLP	2	3.5	0.19	0.62	0.19	0.0069	0.043
	4	5.6	2.15	1.2	2.15	0.049	0.456
	6	5.9	9.13	1.92	9.1	0.232	1.13
	8	7.4	10.2	2.1	9.8	0.31	1.3
webNotreDame	5	2.1	0.017	0.32	0.017	0	0
	10	3.5	0.123	0.8	0.123	0.001	0.03
	15	7.9	0.92	1.5	0.62	0.0118	0.0495
	20	11.2	8.39	11.2	3.39	0.321	0.72

The same claim applies to the SF graphs, as shown in Fig. 11, in which we report the indexing time and space for the generated graphs.

In Figs. 12, 13, 14, and 15, we show the average time on evaluating all the tree patterns and graph patterns shown in Fig. 7 against both the ER graphs and SF graphs. From them, we have three findings: (1) Our method is much better than both the 2Hb and the LLRb in query time due to the direct loading of relations and the relation filtering; (2) as the values of δ increases, the query time of all the three methods dramatically grows up; (3) the LLRb generally works better than the 2Hb due to the efficient checking of Chebyshev distances and the clustering of vertices.

Finally, we show the impact of the join ordering in Fig. 16. In this test, we run two versions of our algorithm: with and without join ordering, to find matches of the four graph patterns shown in Fig. 7b (referred to as P1, P2, P3, and P4, respectively) in an SF graph with $|V| = 100{,}000$, $|E| = 500{,}000$, and l set to be 2.8. Although, after the relation filtering, the sizes of the relations have been greatly reduced, the difference of join ordering can still be observed.

Author's personal copy

Table 7 Query time for a specific query pattern over large graphs

Graphs	δ	2Hb		LLRb		Ours	
		RC (s)	NJ (s)	RC (s)	NJ (s)	RF (s)	NJ (s)
citeseer	2	10.2	0.6	3.2	0.6	0	0
	4	8.3	5.2	5.7	2.2	0.001	0.03
	6	8.1	9.2	7.6	7.8	0.0118	0.494
	8	8.8	11.6	8.1	10.6	0.321	1.5
webBerkStan	2	7.1	1.2	4.6	0.9	0	0
	4	10.41	16.3	7.4	8.3	1.0012	0.085
	6	9.8	20.3	8.1	13.1	0.0066	0.177
	8	39.2	32	8.4	16.7	0.0147	1.114
webGoogle	5	10.3	2.1	6.7	1.6	0.001	0.059
	10	15.4	20.4	7.85	11.4	0.001	0.011
	15	17.3	23.8	9.3	17.8	0.0077	0.065
	20	19.4	40.3	10.4	19.3	0.0285	3.307
roadNetPA	2	3.9	3.6	2.2	2.1	0	0
	4	7.2	20.4	4.4	7.4	0	0
	6	8.6	21.8	5.8	10.8	0.054	0.173
	8	10.3	32.1	6.9	13.1	0.3	3.314
roadNetTX	5	F	F	F	F	0.06	0.02
	10					0.1069	0.869
	15					0.2086	1.038
	20					0.3132	5.248
citPatterns	5	F	F	F	F	0.106	0.02
	10					0.2049	1.097
	15					0.6125	1.447
	20					2.0708	6.1437
wiki-topcats	5	F	F	F	F	0.21	0.024
	10					0.407	2.12
	15					0.9351	2.56
	20					4.022	7.001

F indiates that the tests on the corresponding graphs fail to produce the results

8 Conclusion

In this paper, we have proposed a novel algorithm for pattern match problem over large data graphs G. The main idea behind it is the concept of δ-transitive closures and a relation filtering method based on the concept of triangle consistency. As part of an index, a δ-transitive closure G^{δ} of G will be constructed offline for a certain δ-value. Then, for a query with $\delta' \leq \delta$, the relevant relations in G^{δ} can be directly loaded into main memory, instead of constructing them on-the-fly from some auxiliary data structures such as 2-hops [30] and LLR-embedding vectors [31]. In particular, the useless tuples in the relations will be filtered before they take part in joins, which enables us to achieve high performance. In addition, the bit mapping

Author's personal copy

Table 8 Test results on large δ

$\delta=8$			$\delta=16$			$\delta=24$		
Indexing time (s)	Index size (MB)	Query time (s)	Indexing time (s)	Index size (MB)	Query time (s)	Indexing time (s)	Indexing time (mb)	Query time (s)
10	476.9	3.5	709.30	10,888.01	9.75	1211.71	17,219.0	15.6

Author's personal copy

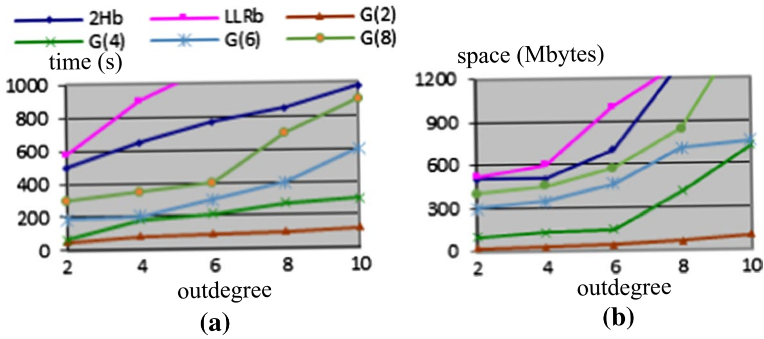


Fig. 10 Indexing time and space for ER graphs

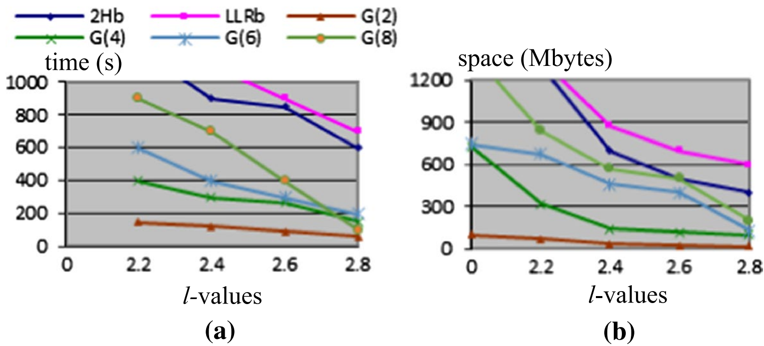


Fig. 11 Indexing time and space for SF graphs

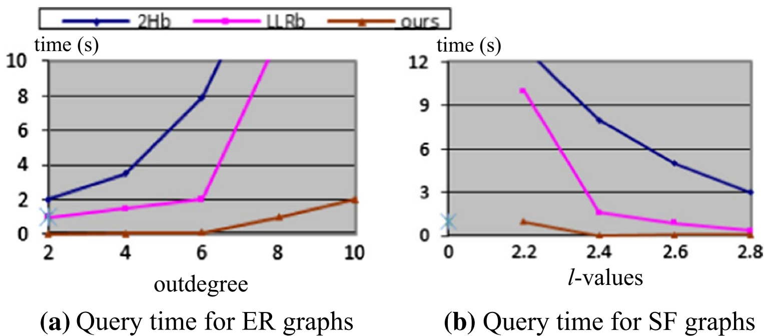


Fig. 12 Query time with $\delta = 2$

technique has been integrated into the relation filtering to expedite the working process. Also, extensive experiments have been conducted, which shows that our method is promising.

Author's personal copy

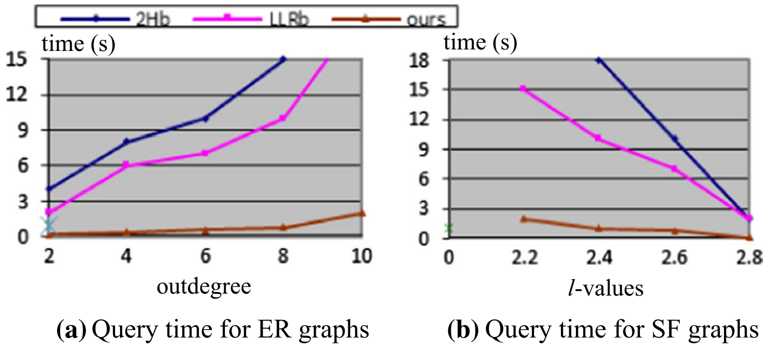


Fig. 13 Query time with $\delta = 4$

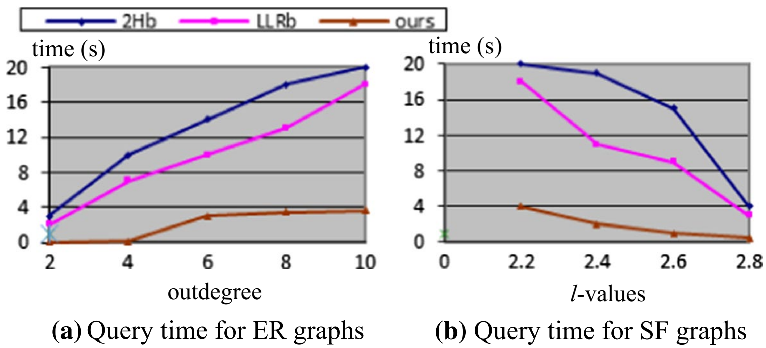


Fig. 14 Query time with $\delta = 6$

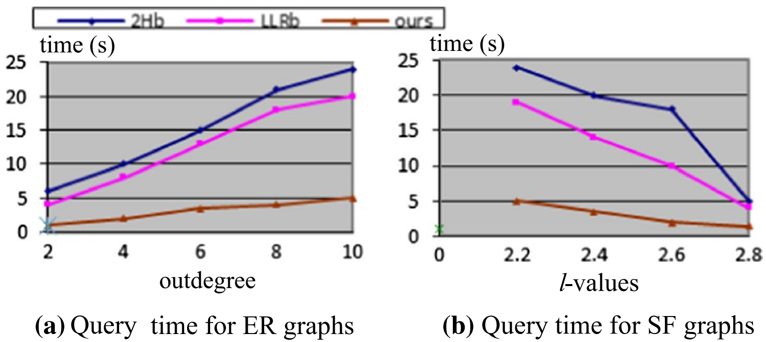


Fig. 15 Query time with $\delta = 8$

Author's personal copy

Fig. 16 Impact of join ordering on time(s)

References

1. Shasha D, Wang JTL, Giugno R (2002) Algorithmics and applications of tree and graph searching. In: ACM SIGMOD-SIGACT-SIGART Symposium Principles Database Systems, p 39
2. Cheng J, Ke Y, Ng W, Lu A (2007) Fg-index: towards verification-free query processing on graph databases. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp 857–872
3. Jiang H, Wang H, Yu PS, Zhou S (2007) GString: a novel approach for efficient search in graph databases. In: Proceedings of the 23rd International Conference on ICDE, pp 566–575. IEEE
4. Tian Y, McEachin RC, Santos C, States DJ, Patel JM (2007) SAGA: a subgraph matching tool for biological graphs. Bioinformatics 23(2):232–239
5. Cheng J, Yu JX (2009) On-line exact shortest distance query processing. In: Proceedings of the 12th International Conference on Extending Database Technology Advances Database Technology, EDBT 09, pp 481–492
6. Cohen E, Halperin E, Kaplan H, Zwick U (2003) Reachability and distance queries via 2-Hop labels. SIAM J Comput 32:1338–1355
7. Chen Y, Chen Y (2008) An efficient algorithm for answering graph reachability queries. In: Proceedings of the ICDE, pp 893–902
8. Wang H, He H, Yang J, Yu PS, Yu JX (2006) Dual labeling: answering graph reachability queries in constant time. In: Proceedings of the International Conference on ICDE, pp 75–86
9. Chen Y, Chen YB (2011) Decomposing DAGs into spanning trees: a new way to compress transitive closures. In: Proceedings of the 27th International Conference on Data Engineering (ICDE 2011), IEEE, April 2011, pp 1007–1018
10. Moustafa WE, Kimmig A, Deshpande A, Getoor L (2014) Subgraph pattern matching over uncertain graphs with identity linkage uncertainty. In: Proceedings of the International Conference on ICDE, pp 904–915
11. Tong H, Gallagher B, Faloutsos C, Eliassi-Rad T (2007) Fast best-effort pattern matching in large attributed graphs. In: Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery Data Mining, pp 737–746
12. Cheng J, Yu JX, Ding B, Yu PS, Wang H (2008) Fast graph pattern matching. In: Proceedings of the International Conference on ICDE, pp 913–922
13. Zou L, Chen L, Özsu M (2009) Distance-join: pattern match query in a large graph. VLDB 2(1):886–897
14. Tian Y, Patel JM (2008) TALE: a tool for approximate large graph matching. In: Proceedings of the International Conference on ICDE, pp 963–972
15. Conte D, Foggia P, Sansone C, Vento M (2004) Thirty years of graph matching in pattern recognition. Int J Pattern Recognit Artif Intell 18(3):265–298
16. Melnik S, Garcia-Molina H (2002) Similarity flooding: a versatile graph matching algorithm and its application to schema matching. In: Proceedings of the ICDE
17. He H, Singh AK (2008) Closure—tree: an index structure for graph queries. In: Proceedings of the ICDE, pp 405–418
18. Toresen S (2007) An efficient solution to inexact graph matching with applications to computer vision. Ph.D. thesis, Department of Computer and Information Science. Norwegian University of Science and Technology
19. Garey, Johnson DS (1990) Computers and intractability: a guide to the theory of Np-completeness. W.H. Freeman & Co, New York

Author's personal copy

20. Hopcroft JE, Wong J (1974) Linear time algorithm for isomorphism of planar graphs. In: Proceedings of the 6th Annual ACM Symposium Theory of Computing, pp 172–184
21. Luks EM (1982) Isomorphism of graphs of bounded valence graphs can be tested in polynomial time. J Comput Syst Sci 25:42–65
22. Yan X, Yu PS, Han J (2004) Graph indexing: a frequent structure-based approach. In: Proceedings of the ACM SIGMOD, pp 335–346
23. Zhang S, Hu M, Yang J (2007) TreePi: a novel graph indexing method. In: Proceedings of the International Conference on Data Engineering, pp 966–975
24. Williams DW, Huan J, Wang W (2007) Graph database indexing using structured graph decomposition Department of Computer Science. In: Proceedings of the 23rd International Conference on ICDE, pp 976–985
25. Zhao P, Yu JX, Yu PS (2007) Graph indexing: tree + delta >= graph. In: Proceedings of the International Conference on VLDB, October 2007, pp 938–949
26. Zhao P, Jiawei H (2010) On graph query optimization in large networks. In: Proceedings of the VLDB, pp. 340–351
27. Trißl S, Leser U (2007) Fast and practical indexing and querying of very large graphs. In: Proceedings of the SIGMOD'2007, pp. 845–856
28. Cordella LP, Foggia P, Sansone C, Vento M (2000) Fast graph matching for detecting CAD image components. In: Proceedings of the 15th International Conference Pattern Recognition, pp 1034–1037
29. Cordella LP, Foggia P, Sansone C, Tortorlla F, Vento M (1998) Graph matching: a fast algorihm and its evaluation. In: Proceedings of the 15th International Conference on Pattern Recognition, pp 1852–1854
30. Cohen E, Halperin E, Kaplan H, Zwick U (2003) Reachability and distance queries via 2-hop labels. SIAM J Comput 32(5):1338–1355
31. Linial N, London E, Rabinovich Y (1995) The geometry of graphs and some of its algorithmic applications. Combinatorica 15(2):215–245
32. Shahabi C, Kolahdouzan MR, Sharifzadeh M (2003) A road network embedding technique for K-nearest neighbor search in moving object databases. Geoinformatica 7(3):255–273
33. Abello JM, Pardalos PM, Resende MGC (eds) (2002) Handbook of massive data sets. Springer, Berlin
34. Jiawei H, Kamber M, Pei J (2012) Data mining: concepts and techniques. Elsevier/Morgan Kaufmann, Amsterdam
35. Henzinger MR, Henzinger T, Kopke P (1995) Computing simulations on finite and infinite graphs. In: Proceedings of the FOCS
36. Li J, Cao Y, Ma S (2017) Relaxing graph pattern matching with explanations. In: Proceedings of the International Conference CIKM'17, November 6–10, Singapore
37. Fan W, Wang X, Wu Y (2013) Incremental graph pattern matching. ACM Trans Database Syst 38(3):18.1–18.44
38. Fredman ML (1976) New bounds on the complexity of the shortest path problem. SIAM J Comput 5(1):83–89
39. Ahuja RK, Mehlhorn K, Orlin JB, Tarjan RE (1990) Faster algorithms for the shortest path problem. J ACM 37:213–223
40. Steinbrunn M, Moerkotte G, Kemper A (1997) Heuristic and randomized optimization for the join ordering problem. VLDB J 6(3):191–208
41. Wu Y, Patel JM, Jagadish HV (2003) Structural join order selection for xml query optimization. In: Proceedings of the ICDE
42. Krishnamurthy R, Boral H, Zaniolo C (1986) Optimization of non-recursive queries. In: Proceedings of the VLDB, Kyoto, Japan, pp 128–137

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.