International Journal of Information Technology and Knowledge Management
July-December 2012, Volume 5, No. 2, pp. 401-407

GRAPH PATTERN MINING: A SURVEY OF ISSUES AND APPROACHES

B. Bhargavi[1] and K.P. Supreethi[2]

Abstract: Most of the internet data that is available in public that are analyzed/archived is graph structured in nature. Graphs form a powerful modeling tool in many areas that include chemistry, biology, www etc. Hence there is a demand for efficiently querying such large graph data. Graph pattern matching problem is to find all the patterns from a large data graph that match the given graph pattern. The survey paper discusses tree pattern matching and graph pattern matching techniques and efficient computation of compressed transitive closure using 2-hop labeling. Join based algorithms are proposed which are two step filter(R-semijoin) and fetch(R-join) steps that are implemented using cluster-based index in relational database context. Optimization techniques like R-join order selection with R-semijoin enhancement and interleaving R-joins and R-semijoins are proposed which make graph pattern matching efficient.

Keywords: graph pattern, graph matching, 2-hop labeling, reachability join

1. INTRODUCTION

Due to rapid growth of Internet, most of the data is archived and analysed which is available in public is graph-structured in nature. For example, RDF (Resource Description Framework) is a directed labeled graph used to represent resource/property/value triples for describing semantic resources on web. Graphs form a powerful modelling tool to represent various networks in different areas like chemistry, biology, social networks, etc. In biology, complex protein-protein interaction networks can be conveniently expressed using graphs. According to Scifinder (a scientific website about chemical compounds) report, 4000 new chemical compound structures are added each day which can be efficiently represented as large graphs. Twitter, an online social networking site, initially stored its data and maintained relationships of users' data using an open fault-tolerant distributed graph database called FlockDB. Thus, there is a demand for efficiently querying the graph data.

Graph Database is a large labeled directed graph or a collection of labeled directed graphs. A graph pattern is a graph query which is constructed by connecting nodes based on links/relationships required by user. Given a graph database and a graph pattern, find all the patterns that match a user given graph pattern is the graph pattern matching problem. But, the graph pattern matching problem is challenging as graph data can be large and graph patterns can be large and complex.

Graph pattern matching applications include finding research collaboration information like citation links analysis from bibliographic data, relationships and their proximity between persons in a social network and finding patterns of interest by scientists in biological networks like protein-protein interaction networks, source code analysis etc. Extensive research has been done and many algorithms are proposed and implemented in following Graph pattern mining areas: frequent subgraph mining using sub-graph isomorphism, exact pattern matching and pattern matching with wildcards based on distance.

1.1 Subgraph Isomorphism

There are several algorithms defined that implement sub-graph isomorphism for frequent subgraph mining. In frequent subgraph mining, given a graph pattern and a large graph, the problem is to find all the isomorphic subgraphs that match a given graph pattern. Many algorithms are proposed for frequent subgraph mining on certain data and also currently, on uncertain data, *i.e.* data that is incomplete and inaccurate due to noise. But, subgraph isomorphic problem is an NP-complete problem.

1.2 Graph Pattern Matching

The other type of graph pattern matching is given an *n*-node user-defined graph pattern, find all the patterns from the graph database that match the user-given graph pattern exactly, *i.e.*, every label of each node from graph database should match the labels of user defined graph pattern and also every node reachable from/to every node *i.e.* edge should match to that of user graph pattern.

Different techniques have been proposed for graph pattern matching based on distance threshold described by Zou *et al.* [10]. The problem is to find all the patterns that may match exactly or inexactly with a limited distance between nodes based on threshold.

[1,2] Department of Computer Science and Engineering, JNTU College of Engineering, Hyderabad, India, [1]*E-mail: bhargavi.bbv1@gmail.com, supreethi.pujari@gmail.com*

The contributions of this paper include:

- Overview of tree-pattern matching techniques.
- Overview of various techniques for computing transitive closure and storage of the compressed form efficiently. Survey on 2-hop labeling computation for directed graphs and thus discovering a faster and efficient hierarchical geometry-based approach for 2-hop labeling computation.
- Suggesting efficient join-based algorithms and optimization techniques for graph pattern matching.

In this paper, section 2 covers tree pattern matching techniques. Section 3 describes briefly different datasets that will be tested upon for efficiency and scalability. Section 4 clearly describes about the graph pattern. Section 5 covers internal representation of graph data. Section 6 covers transitive closure storage and computation techniques. Section 7 covers the existing approach and section 8 includes join-based algorithms and optimization techniques and finally in section 9 we give concluding remarks.

2. TREE-PATTERN MATCHING TECHNIQUES

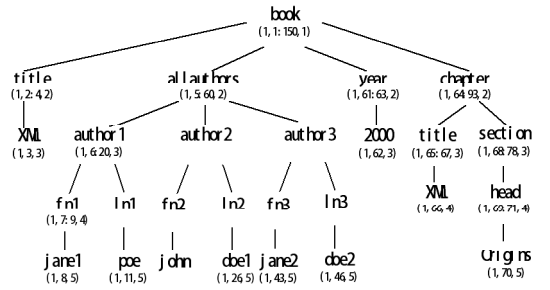


Figure 1: A Sample XML Tree Representation (Bruno et al. 2002)

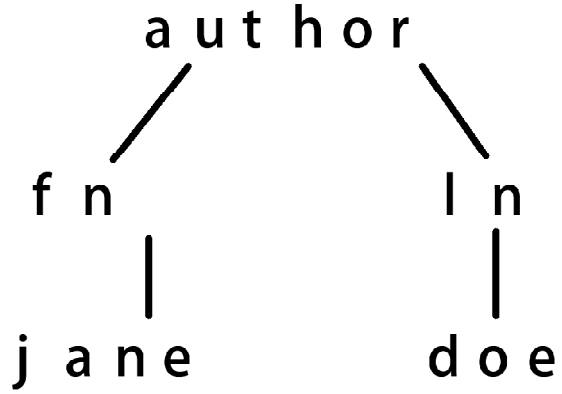


Figure 2: A Tree Pattern

Tree pattern matching is to find all the patterns in an xml document represented as a tree model that match user-given tree pattern. For instance, consider xml tree model shown in Figure 1. The query is to find all the authors who have their firstname "jane" and lastname "doe" which is represented as a tree-pattern in Figure 2. The tree pattern matching problem is to find all the set of tuples that match the user-defined tree pattern. In Figure 1, the tuple <author3, fn3, ln3> matches the given tree-pattern. There are different tree pattern matching techniques like binary structural join-based approach, tree pattern matching with stack encoding [8] and tree pattern matching with hierarchical stack encoding [9].

2.1 Binary Structural Join-based Approach

This is the basic approach that evaluates the tree pattern by decomposing the tree pattern into binary structural relationships, using structural join algorithms to match the binary relationships against XML database and then combining the results for finding the set of tuples that match the tree-pattern. But, this approach produces large number of intermediate results. This problem can be overcome by Stack encoding technique described by Bruno et al. [8].

2.2 Tree Pattern Matching with Stack Encoding

Tree pattern matching with stack encoding involves initializing stacks for each user-required node, finding intermediate results of twigs (i.e. a twig is a sub-query that ends at a leaf node), maintaining them in stacks for faster processing of intermediate results and combining the results of twigs efficiently to give the final set of matching tuples.

An improvement of the above approach is hierarchical stack encoding described by Chen et al. [9] in which for each node the results are arranged in stack hierarchical to their order in the database and results are enumerated efficiently by hybrid bottom-up and top-down approach. The hierarchical stack encoding approach is applied on Generalized Tree Pattern queries which are generalized form of XQuery. XQuery is a query language for XML documents just like SQL is to relational databases. XPath expressions and operators are used in XQuery that can traverse through elements and attributes of XML data forward or backward. Using FLWOR (for-let-where-order-by) expressions of XQuery, solution to given query is evaluated and results of query are retrieved. The queries will be tree-structured as xml data is modelled as a tree and results are retrieved in a result set tag in XQuery. But these techniques cannot be directly applied on graphs as graphs don't have the good property of trees *i.e.* the acyclic property. Hence, different approaches are applied on graph pattern matching.

3. XML DATA AS GRAPH

Graph database represents data as a graph. The data can be a large real xml document like DBLP data or synthetic XML

data like XMark which are to be parsed using SAX parser to derive the directed data graph. DBLP XML document maintains bibliographic details of research collaborations of various authors published at various conferences/journals and citation information that can be modelled as a large directed data graph and experimented to evaluate and test for efficiency and validity of various proposed algorithms that work on graph data.

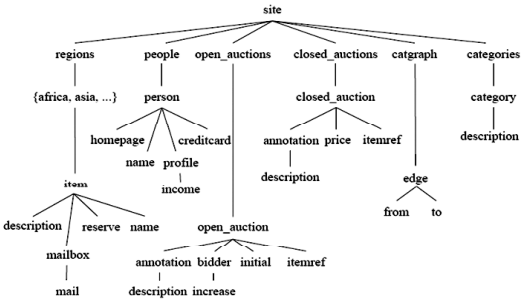


Figure 3: Elements and their Relationship in an XMark xml Document

XMark is a synthetic XML benchmark known for its irregular schema. It maintains auction site details like when the auction is open, when it is closed, who participated in it, etc. There are elements that internally refer to other elements in the document (For instance, in Figure 3, "itemref" element refers to "item" element) which can be used to encode the XMark XML document as a graph. The xmlgen tool of XMark constitutes a scaling factor which can be adjusted for generating XML documents with huge size of upto 10GB for testing scalability issues.

4. GRAPH PATTERN

A graph pattern is a pattern interested by user which contains set of vertex labels and edges representing the user-defined query. A graph pattern can take nodes as element tags, attribute -values or comparisons and edges as parent-child relationships or referencing relationships. Vertex labels usually represent element tags while edges represent relationships.

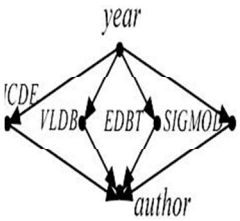


Figure 4: Graph Pattern

Consider the instance of graph pattern of Figure 5 which represents the query of finding all the authors who have published papers at ICDE, VLDB, EDBT and SIGMOD at the same year.

Each edge is defined as reachability joins(R-join) as it represents whether a destination edge label can be reachable from the source edge label. The graph pattern is evaluated as a sequence of R-joins by Cheng et al. [1]. The graph pattern matching problem finds all the set of tuples that match the user-given graph pattern from a large directed data graph. It is to be noted that if there exist R-joins $X \rightarrow Y$, $Y \rightarrow Z$ in the graph pattern where X,Y,Z represent labels, it implies that the graph pattern satisfies $X \rightarrow Z$ which is not present in the case of subgraph isomorphism making frequent subgraph mining intractable. For instance, in Figure 3, there exists R-joins, year→ICDE & ICDE→author, it implies that the given graph pattern satisfies year→author.

5. INTERNAL REPRESENTATIONS OF DIRECTED DATA GRAPH

A directed data graph can be internally represented in general using either the adjacency matrix or adjacency list implementation. In the adjacency matrix implementation, an $n*n$ matrix is maintained for *'n'* nodes of directed graph G. Each ijth element in matrix contains one of the values $\{1, 0, -1\}$ that represent if the ith node is adjacent to the jth node or not. If it is adjacent, the value of ijth element is 1 (the negative sign implies that jth node is the source node for the incoming edge to ith node) else it is 0. Adjacency matrix is generally used to represent dense graphs. Adjacency list is used to internally represent sparse graphs. In this representation, for each node, a list is maintained which contains the list of all the target nodes that are adjacent to the current node.

There are several libraries defined to internally represent graphs like in C++, the Boost graph library which supports various representations and user-defined interfaces for graphs, by default, Boost offers its own representation class adjacency_list. In java, JDSL offers rich support for graphs in jdsl.graph. It has a clear separation between interfaces, algorithms and representation. It offers an adjacency list representation of graphs that supports directed and undirected edges. Jgrapht is a java software library which contains classes and interfaces useful to work on different graph algorithms. Using jgraph library, the graphs can be visualized.

6. TRANSITIVE CLOSURE IMPLEMENTATION AND STORAGE TECHNIQUES

Transitive closure represents set of paths that satisfy transitivity property. If there is a path from nodes u to w of a directed graph G defined by (u, w) and similarly a path (w, v) then by transitivity there exists a path (u, v). By pre-computing transitive closure, we can access shortest paths faster and determine the existence of paths between two nodes.

The following are different techniques of computing and storing Transitive closure:

- Transitive closure computation by Warshall's algorithm
- Multi Interval encoding
- 2-hop labeling

Transitive closure computed by Warshall's algorithm is a naïve approach which traverses each node and computes the reachable paths from every node to other node of directed graph with time complexity $O(V^3)$.

6.1 Multi-interval Encoding

Chen et al. [11] used stack-based multi-interval encoding Scheme on directed acyclic graphs(DAGs) where for each node, set of intervals and a postid are assigned that represent the position of the node in graph using algorithms of [12]. Each interval of node [*s*, *e*] constitutes *'e'*, a post-order number of the node (postid) derived from post-order traversal of the optimal tree cover of DAG as described in [12] and *'s'* a post-order number of the lowest descendant node assigned to the current node that are used to represent the ancestor-descendant relationship between two nodes. Multiple intervals for each node of the graph come into existence if there are back edges (*i.e.*, a back edge is an edge not present in optimal tree cover but present in the DAG) to the node.

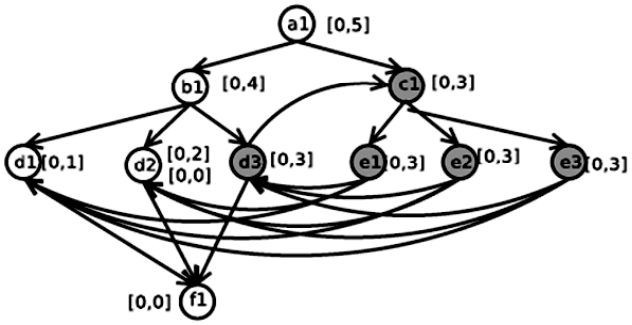


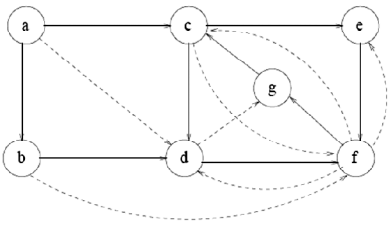
Figure 5: Multi-interval Encoding for a Graph

Consider an example of figure 5 in which multi-interval coding is assigned for a graph. First, the graph is condensed to DAG and then intervals are assigned for each node which represents the ancestor-descendant relationship between the nodes. For instance, consider the node "*b*1" which is encoded by set of intervals {[0,4]}. A node is said to be descendant of "*b*1" if postid of the node lies between [0, 4]. In Figure 6, the nodes "*d*1", "*d*2", "*d*3", "*c*1", "*e*1", "*e*2", "e3", "*f*1" form the descendants of the node "*b*1". In general, there exists a path *a*~*b* if and only if $\exists i\ (0 \le i \le n)$ such that $ai.x \le$ postidb $\le ai.y$. Thus, multi-interval encoding represents the compressed transitive closure of a graph.

6.2 2-Hop Labeling

Cohen et al. [3] defined theories and proposed a solution to compute 2-hop labeling on general graphs. A hop *h* is defined as a pair (*p*, *v*) where *p* is a path in a graph *G* & *v* is one of the end vertices of path *p* of graph. For each node *'v'* in directed graph *G*, a label $L(v)$ is assigned which has $L_{in}(v)$ that represents all the nodes *'u'* in *G* that can reach *v* and $L_{out}(v)$ that represents all nodes *'w'* in *G* that are reachable from *v*, (hence the name 2-hop, one hop from *u* to *v* and other hop from *v* to *w*) which define the 2-hop reachability labeling.

2-hop cover is defined as collection of hops which cover all connections in *G*. 2-hop labeling is assigned to the nodes of the graph such that all connections of the graph *G* are covered.



v	$L_{in}(v)$	$L_{out}(v)$
a		b, c, d
b		f, d
c	f	d, f
d	f, c	g, f
e	c, f	f
f	d, c	g, c, e
g	d, f	c

Figure 6: 2-hop Labeling for a Graph

For instance, consider the graph in Figure 6 where solid edges are the edges of the graph while dashed edges are hops that are not edges of the graph. The dashed edge, for instance $a \rightarrow d$ represents the hop $(a \rightarrow b \rightarrow d, d)$. The table to the right of graph of Figure 6 shows 2-hop labels assigned for each node. For instance for node "*g*", $L_{in}(g) = \{d, f\}$ and $L_{out}(g) = \{c\}$ which implies that the nodes *"d"*, *"f"* can reach *"g"* while *"g"* can reach *"c"*. Thus, by assigning 2-hop labels, all connections are covered.

The problem of finding 2-hop cover/2-hop labeling is found to be an NP-hard problem as it can be reducible to minimum set-cover problem which has no optimal solution. Each node in minimum set-cover problem is represented by $S(U_w, w, V_w)$ and centre graph is constructed where *w* is a node centre of the bipartite graph.

The centre with maximum cost is selected. The cost is assigned to the centre nodes based on the criterion of maximum number of paths that the node can cover and this set-cover problem is reduced to densest subgraph problem and all the centre nodes are traversed in decreasing order of cost and nodes are assigned to L_{in} and L_{out} based on whether the node is reachable from or to the current node and covered connections, nodes are pruned from pre-computed transitive closure and it is repeated until all connections are covered. The computation of 2-hop labelling is $O(\log|V|)$ times the optimal solution and takes $O(|V|\log|E|)$ space where $|V|$ and $|E|$ represent the total number of vertices and edges of directed graph respectively.

Schenkel et al. [4] used the 2-hop labeling technique for creating connection index for heterogeneous xml

document collection. They improved the 2-hop labelling computation algorithm by implementing it in a divide and conquer approach. It includes the following steps:

1. Partition the original graph with a considerable memory bound size for each partition.
2. Compute transitive closure and 2-hop cover for each partition and store the 2-hop cover on disk.
3. Merge the 2-hop covers for partitions that have one or more cross-partition edges, yielding 2-hop cover.

The advantage of this approach is efficient usage of memory and pre-computation of the transitive closure of the partition is required than that of the whole directed graph which makes it faster. Schenkel et al. [5] further made following improvements to compute faster and efficient 2-hop labelling:

- Partitioning is done such that transitive closure for each partition fits into memory.
- While building partition covers, cross-link targets, i.e. the nodes of one xml document that are linked to the nodes of other document, are selected as center nodes.

Skeleton graph that represents cross-link source and target edges is used while building the initial partitioning. They also worked on how to update 2-hop labels when xml elements /documents are inserted or deleted. Insertion is trivial which is done by adding new labels for each inserted edge. Deletion of node/ edge involves forming good documents that separate the graph which is trivial & bad documents are formed that don't separate the graph.

Cheng et al. [6] used geometry-based approach to compute 2-hop labelling. The main advantage over previous approaches is that in this approach there is no need to compute transitive closure. First, for a directed graph, construct DAG by condensing each maximal strongly connected component into node. Compute 2-hop cover for DAG. Compute 2-hop cover of G using 2-hop cover computed for DAG.

To compute 2-hop cover of DAG, a set of intervals $[s, e]$ are assigned to each node as in multi-interval encoding scheme for the DAG. Similarly, the DAG with reverse edges is constructed and multi-intervals are computed and stored in reachability table. Then a reachability map is constructed based on post-order numbers assigned to nodes of DAG on x-axis and post-order numbers of nodes in reverse DAG on Y-axis. Each point (x, y) in map is assigned a rectangle on the map. R-tree, a height balanced data structure is used to store and retrieve the rectangles of reachability map efficiently. 2-hop cover is computed by mapping 2-hop cover problem onto a 2-dimensional grid point and by using operations on rectangles with the help of R-tree, i.e. bipartite graph is mapped with a virtual center in rectangles, and densest bipartite graph is computed. For the nodes in densest bipartite graph, 2-hop labels are assigned and the process is repeated for the next denser bipartite graph until all connections are covered. This approach focused on finding minimum number of large subsets rather than minimum number of overlapping among subsets which is found to be efficient.

Cheng et al. [7] further improved the algorithm. First, the directed graph is converted to DAG and then partitioned into two subgraphs based on bisection. The edges connecting two subgraphs are stored in an induced subgraph and 2-hop cover is computed for it using either R-tree based node-oriented approach or edge oriented approach. Then, all the nodes and edges of induced subgraph are removed from the two partitioned subgraphs, to form two independent subgraphs and 2-hop cover of the resulting subgraphs can be computed independently without need of merging. The subgraphs can be further partitioned if the number of vertices are greater than a threshold other wise 2-hop cover of the subgraph is computed based on their earlier approach. This approach is found to be efficient in computing 2-hop labelling for the directed data graph.

7. AN EXISTING APPROACH OF GRAPH PATTERN MATCHING

The approach of graph pattern matching used by Wang et al. [2] on XML documents use the multi interval encoding scheme on directed graphs defined by [12]. For finding possible tuples of a query edge $A \rightarrow D$, two lists AList and DList, ancestor list and descendant list are maintained with Alist containing set of intervals $[x_i, y_i]$ of nodes with label A sorted in descending order of x value and Dlist contains post-ids sorted in ascending order that are sorted based on increasing post-order number. The lists are merged using range search tree for finding the required matching edges. Wang et al. [2] also worked on subgraph query patterns that are cyclic and devised algorithms based on interval-stack data structure.

But the drawback of using multi-interval encoding scheme in graph pattern matching is sorting is to be done while joining and interval encoding is lengthy and computations over them are time-consuming during graph pattern matching which are not present in the following proposed approach.

8. A JOIN BASED APPROACH OF GRAPH PATTERN MATCHING

Cheng et al. in [1] used the fast hierarchical geometry-based algorithm defined in [7] for efficiently computing 2-hop labeling in the Join/Semijoin approach of graph pattern matching. Each reachability condition/edge that appears in user-given graph pattern is defined as R-join. Each node of directed graph contains a label X and is uniquely identified

by name *xi* with *'i'* in [1, *n*] which implies that there are *'n'* nodes with same label *X* which is defined as extent (*X*). A table is maintained for each such label that represents its extent and 2-hop labels for each identifier node of the label which is termed as base relation.

8.1 Cluster-based R-join Index

Cluster-based index is constructed using these base relations which are used for indexing tuples that can join between two relations. The cluster-based index includes B+ tree construction of centre nodes that are formed during 2-hop labeling computation. B+ tree is a widely used indexing mechanism in relational databases. The leaf node of B+ tree points to two sets termed F-cluster and T-cluster. A node in F-cluster can reach a node in T-cluster via the centre node. These clusters are further grouped into labeled subclusters based on labels assigned to nodes. W-table is constructed for edges, that is, for all possible label pairs, set of centers are included. A center is included in $W(X, Y)$ if the center node has X-labeled F-subcluster and Y-labeled T-subcluster. For computing an R-join X'Y, W-table gives the center for the edge. Using the center in R-join index , resultant set of tuples is the cartesian product of X-labeled F-subcluster and Y-labeled T-subcluster.

8.2 Join Based Algorithms

Using the following algorithms, the result of matching set of tuples is obtained for a user-given graph pattern:

Algorithm1 is implemented initially for an edge which gives the result of join between two base relations using cluster-based R-join index.Algorithm2 which contains filter step that filters/prunes unwanted tuples from a temporal relation and fetch step that gives the resultant join of temporal relation and a base relation.

Algorithm1:

Input: an edge $E(X \rightarrow Y)$, *Tx*, *Ty*

Output: Set of binary tuples<*xi*, *yi*>

Get centers from W-table for *E*.

For each center *'w'*, get X-labeled F-subcluster nodes and Y-labeled T-subcluster nodes from R-join index.

Compute the cartesian product of two subclusters.

Algorithm2:

Input: an edge $X \rightarrow Y$, a temporal relation T_r containing X, T_y

Output: Set of tuples.

Filter() step: Prune the tuples from T_r that contain x_i that cannot reach y_i and store the remaining tuples with centers as $< r_i, w_i >$ in T_w.

Fetch() step: For each w_i, get set of Y-labeled T-subcluster nodes $\{y_i\}$ and compute cartesian product of $\{r_i\}$ and $\{y_i\}$ which gives resultant set of tuples.

Algorithm1 can be computed by just accessing cluster -based index while Algorithm2 requires access to base relations.

8.3 R-semijoin

The Filter step of Algorithm2 is considered to be R-semijoin. A unique feature of the proposed R-semijoin is that the R-join algorithms must first process R-semijoin to complete R-join. R-semijoin processing includes sequence of R-semijoins that together involves one scan of the temporal relation if in the sequence there is no R-join and processing cost of R-semijoin is estimated to be small.

8.4 Order Selection and Optimization

R-join/R-semijoin order selection is important for optimizing the join based algorithms. For R-join order selection, dynamic programming technique is used to optimize the cost of join processing. The cost of joins is the estimation of joins/semijoins sizes based on "status" and "moves". A "status" defines the cost of subquery in the directed data graph. A "move" defines adding an edge to the current "status" to get new "status". The better approach in R-join order selection is the greedy approach which is proposed for R-join order selection [1].

Optimization is achieved by following techniques:

R-join order selection followed by R-semijoin enhancement

In this technique, optimal R-join order is identified from the proposed approach. R-semijoins are added to same sequence of R-joins based on transitive closure of R-semijoins. By Join-Semijoin reorder, semijoins that are movable are reordered by moving them to left in the sequence. Additional R-joins can be computed from the edge transitive closure of the given graph pattern and added as R-semijoins.

Interleaving R-joins with R-semijoins

Using "status" and "move", R-joins can be interleaved with R-semijoins. There are 3 possible moves. Filter move through which semijoins are added to the current sequence. Fetch move by which self R-join can be processed and an R-join move which is only allowed to move from initial status to final status. Using these moves R-joins are interleaved with R-semijoins for optimization which is observed to perform better than the former optimization approach of R-join order selection with R-semijoin enhancement.

9. CONCLUDING REMARKS

A new join-based approach is proposed for implementing graph pattern matching efficiently. Graph pattern matching

problem is to find the matching patterns to the user-defined graph pattern from a large directed data graph. The user-given n-node graph pattern is processed for finding a set of n-ary tuples as a sequence of reachability joins(R-joins) where an edge represents an R-join. First, from xml document that is used for graph data representation, a large directed graph is constructed. Extensive survey has been done for faster computation of 2-hop labeling techniques for transitive closure representation. A fast and efficient hierarchical geometry-based algorithm is used to implement compressed transitive closure algorithm that assigns 2-hop labels to each node. Then, base relations are constructed for each label that contains 2-hop labelling information. A cluster-based index is constructed from 2-hop labels for faster access of tuples. Efficient algorithms are designed that use base relations and cluster-based index for finding the edge reachability joins. A unique feature of this approach is that R-semijoins are to be processed first before processing R-joins. Additionally, optimization techniques are used that include join-order selection with semijoin enhancement or interleaving R-joins with R-semijoins. The proposed approach computes faster than the existing approach as it does not require sorting and computations over multi-intervals during graph pattern matching.

References

[1] J. Cheng, Jeffrey Xu Yu and Phillip S.Yu , "Graph Pattern Matching: A Join/Semijoin Approach", *IEEE Transactions on Knowledge and Data Engineering*, **23(7)**, pp. 1006-1021.

[2] H. Wang, W. Wang, X. Lin, and J. Li , "Coding-based Join Algorithms for Structural Queries on Graph-Structured XML Document", Springer Publications 2008.

[3] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick, "Reachability and Distance Queries via 2-Hop Labels", *Proc. Ann. ACM-SIAM Symp. Discrete Algorithms* (SODA '02).

[4] R. Schenkel et al., HOPI: "An Efficient Connection Index for Complex XML Document Collections", *Proc. Int'l Conf. Extending Database Technology: Advances in Database Technology* (EDBT '04).

[5] R. Schenkel, A. Theobald, and G. Weikum, "Efficient Creation and Incremental Maintenance of the HOPI Index for Complex XML Document Collections", *Proc. 21st Int'l Conf. Data Eng.* (ICDE '05).

[6] J. Cheng, J.X. Yu, X. Lin, H. Wang, and P.S. Yu, "Fast Computation of Reachability Labeling for Large Graphs", *Proc. Int'l Conf. Extending Database Technology: Advances in Database Technology (EDBT '06).*

[7] J. Cheng, J.X. Yu, X. Lin, H. Wang, and P.S. Yu, "Fast Computing Reachability Labelings for Large Graphs with High Compression Rate", *Proc. Int'l Conf. Extending Database Technology: Advances in Database Technology* (EDBT '08).

[8] N. Bruno, N. Koudas, and D. Srivastava, "Holistic Twig Joins: Optimal XML Pattern Matching", *Proc. ACM SIGMOD.*

[9] S. Chen, H.G. Li, J. Tatemura, W.P. Hsiung, D. Agrawal, and K.S. Candan, "Twig2stack: Bottom-Up Processing of Generalized-Tree-Pattern Queries over XML Documents", *Proc. 32nd Int'l Conf. Very Large Data Bases (VLDB '06).*

[10] L. Zou, L. Chen and M.T. Ozsu, "Distancejoin: Pattern Match Query in a Large Graph Database", *In Proceedings of 35th International Conference on Very Large Data Bases* (VLDB '09).

[11] L. Chen, A. Gupta, and M.E. Kurul, "Stack-Based Algorithms for Pattern Matching on DAGs", *In Proceedings of 31st International Conference of Very Large Data Bases* (VLDB 05).

[12] R. Agrawal, A. Borgida, and H.V. Jagadish, "Efficient Management of Transitive Relationships in Large Data and Knowledge Bases", *Proc. ACM SIGMOD*, 1989.