Twig Query Processing over Graph-Structured XML Data

Zografoula Vagena
University of California
Computer Science & Engineering
Riverside, CA 92521, USA
foula@cs.ucr.edu

Mirella M. Moro
University of California
Computer Science & Engineering
Riverside, CA 92521, USA
mirella@cs.ucr.edu

Vassilis J. Tsotras
University of California
Computer Science & Engineering
Riverside, CA 92521, USA
tsotras@cs.ucr.edu

ABSTRACT

The emergence of XML as the *lingua franca* for e-commerce and other data exchange applications has created the need to explore query processing techniques for XML data. Belonging to the semi-structured category, XML data is usually modeled using graph structures. *Structural summaries*, proposed to speedup XML query processing, have also graph forms. Relevant query languages provide facilities to query both on value as well as structure.

Several proposals already exist for the efficient evaluation of queries over tree structured data, i.e. data whose underlying structure is a tree. Those techniques are not directly applicable when the data is modeled as a random graph. Furthermore, they cannot be applied when *structural summaries* are employed and, to the best of our knowledge, no analogous techniques have been reported for this case either. As a result, the potential of those *structural summaries* is not fully exploited.

In this paper, we investigate query evaluation techniques applicable for graph-structured data. We propose efficient algorithms for the case of directed acyclic graphs, which appear in many real world situations. We then tailor our approaches to handle other directed graphs as well. Our experimental evaluation reveals the advantages of our approaches over existing methods for graph-structured data.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]

1. INTRODUCTION

The widespread acceptance and employment of XML in B2B applications as the standard for data exchange among heterogeneous sources has necessitated the efficient storage and retrieval of XML data. Contrary to relational data, XML data is self-describing and irregular. Hence, it belongs to the category of *semi-structured* data and, as a result, inherits its graph-structure model.

Seventh International Workshop on the Web and Databases (WebDB 2004), June 17-18, 2004, Paris, France .

The absence of schema in XML has led to the employment of *structural summaries* [14, 24, 13, 20, 18, 9, 7, 28], derived directly from the data, to facilitate tasks that would benefit from the existence of a schema, such as query formulation. Those *structural summaries* can play an important role in query evaluation too, by enabling answering queries directly from them, instead of the original data, which has potentially larger size. They are, in general, graph structures too, even when derived from tree structures.

Query languages that have been proposed for XML data, such as XQuery [5] and XPath [4], take into consideration the inherent graph-structure and lack of schema and enable querying both on the structure, as well as on simple values. Structural selection is performed with a navigational approach, where data is explored and elements are located starting from specific entry points. *Tree Pattern Queries* (*TPQ*) [3], also known as *twigs*, have been defined as a subclass of navigation-enabling queries that are useful in many real world situations. They involve element selections that have specified tree structures.

Consider for example the bibliography database of Figure 1. Figure 1a illustrates the graph representation of the XML database, Figure 1b shows its *structural summary*, namely the A(k)-Index (where k is ∞) [18]. In Figure 1a, solid lines represent edges between elements and subelements, while dashed lines represent *idref* edges. We issue the query: `//bib[.//author[@name='Chen']]/article[@title='t1']` that retrieves articles with title 't1' if there exists an author with name 'Chen'. The structural part of the query can be represented as a *twig* and is shown in Figure 1c.

While value-based queries can be evaluated by adopting traditional indexing schemes, e.g. B^+-*trees*, efficient support for the structural part is a unique and challenging problem. Previous efforts to process the structural part of the query directly from the data [29, 2, 6, 12, 21, 27, 26] have assumed the tree structured model of XPath and cannot be applied when the structure is a general directed graph. Research on *structural summaries* [14, 24, 13, 20, 18, 9, 7, 28] has focused on the construction and maintenance of effective structures, i.e. structures that are (a) space efficient, (b) optimized with regard to a specified query workload, and (c) gracefully adaptable in the presence of updates. The query processing task has received little attention.

In this paper, we propose techniques for the evaluation of *twig* queries over graph-structured data. We investigate new ways to adapt ideas that have been proved successful for the case of tree-structured data, namely mapping of the structural conditions to join conditions. We start with the

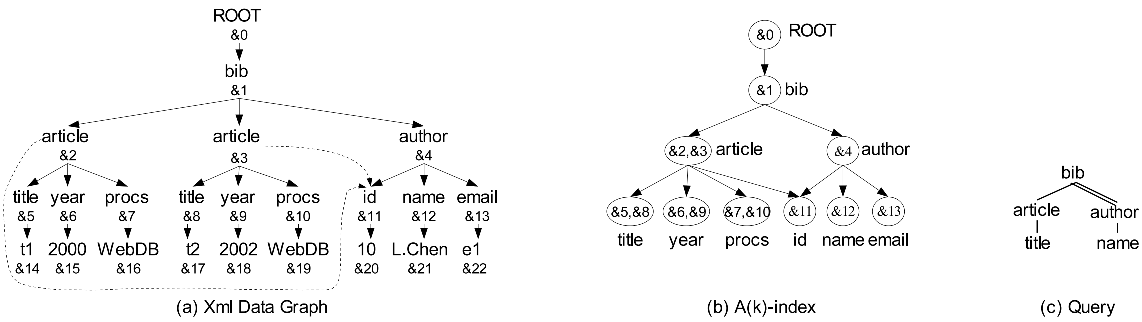

Figure 1: (a) Bibliography XML database, (b) A(k)-Index , (c) Sample Query

particular class of directed and acyclic graphs (which we show being the structure of the data in many cases) and then adapt them for the case of general directed graphs. Our techniques can be used to answer queries either over the original data or through *structural summaries*.

The rest of the paper is organized as follows. Section 2 presents background and related work. In Section 3, we describe and analyze our techniques, and in Section 4 we experimentally investigate their effectiveness. Section 5 provides our conclusions and directions for further work.

2. BACKGROUND AND RELATED WORK

We consider a model of XML data in which we ignore comments, processing instructions and namespaces. Then an XML database can be modeled as a directed, node-labeled graph $G = (V, E, E_{Ref}, root)$, where V is the set of nodes (indicating elements, values or attributes), E is the set of edges, which indicate an object-subobject relation, and E_{Ref} is the set of reference edges (*idref*). Each node in V is assigned a string literal label and has a unique identifier. The single *root* element is labeled `ROOT`. In Figure 1a, the graph structure of a bibliography database is presented.

Previous work has considered in detail the case where no reference edges exist. In those cases, the data structure is a tree. In order to efficiently capture the structural relationships between nodes, a region-based numbering scheme [11, 29, 2, 6] is usually embedded at the document nodes. In that scheme, each node in the document tree is assigned a region: (*left*,*right*). Given a node pair (u, v), node v is a *descendant* of node u (which is then an *ancestor* of v), if and only if, $u.left < v.left < v.right < u.right$.

Using that numbering scheme, set-based techniques [29, 22, 2, 6] regard the input as sequences of elements, and perform a join in a sort-merge fashion. They use the containment condition described in the previous paragraph as the join condition. Indexes have been proposed to skip elements that do not participate in any of the results [8, 6, 16]. Navigation-based techniques [12, 15], which use the input to guide the computation, have also been proposed. Those techniques answer the queries with a single pass over the input. The input is traversed in document order (i.e. in pre-order) and, using an FSM, the query pattern is matched with particular path instances as those instances become available. Work in [21] also utilizes navigation based techniques to solve a wider range of queries. More recently, similar numbering schemes have been employed in query-driven input-probing methods [27, 26], where the structure of the query defines the part of the input data to be examined next. Methods in this category construct the result by matching parts of the query incrementally.

While the above techniques operate at the granularity of the original data, query processing can employ *structural summaries* that have been proposed [20, 18, 19, 9, 7]. Using those structures, the data graph is summarized with a graph of a smaller size that maintains the structural characteristics of the original data. Each index node identifies a group of nodes in the original document. The concept of bisimilarity [25] is utilized to form the groups. Those proposals are mainly concerned with creating effective indexes and little is said about the processing of the twig queries. In summary, they regard the twig query as a path query (primary path) with structural constraints on the nodes of that path. They employ navigation-based techniques to identify nodes that match the primary path and recursively check the validity of structural constraints. However, such an approach may suffer from unnecessary access over the input in order to check the structural constraints. It would be desirable to devise techniques with similar characteristics as the ones in the previous paragraph for the case of *structural summaries* too. Nevertheless, the graph structure of those summaries hinders the direct adaptation of the techniques described in the previous paragraph.

3. PROCESSING TWIGS IN GRAPHS

In this section, we describe our solution to efficiently process twig queries over graph structured data. We focus on the descendant axis and consider the child axis as a special case. In a directed graph environment, the ancestor-descendant relationship of a tree pattern edge is satisfied if there is a path from the ancestor node to the descendant node. We first describe node labeling schemes that identify the structural relationship between two graph nodes (as the numbering scheme described in the previous section does for tree structures). We then describe our matching algorithms that utilize those schemes to efficiently compute twig queries and analyze their behavior.

3.1 Labeling Scheme for Digraphs

As mentioned earlier, in a directed graph environment, the ancestor-descendant relationship is satisfied if there is a path from the ancestor node to the descendant node. We would like to be able to answer the question of the existence of such a path within reasonable time. In graph theory, this is the well know *reachability* problem and can be handled by computing the transitive closure of the graph structure. Then, for each pair of nodes, the reachability question can be answered in constant time. However such an approach is space consuming, as it requires space $O(n^2)$ in the number of nodes. The question that is posed is whether one can get the same functionality using less space, possibly trading some of the time to check reachability. The answer is yes, by employing the minimum 2-hop cover of a graph. However, it has been shown [10] that the problem of identifying the minimum 2-hop cover is a NP-hard problem. A practical solution that identifies a 2-hop cover close to the smallest cover is presented in [10] and is employed in our techniques. The notion of 2-hop labeling is defined as follows:

Definition 1. Let $G = (V, E)$ be a directed graph. A 2-hop reachability labeling of G assigns to each vertex $v \in V$ a label $L(v) = (L_{in}(v), L_{out}(v))$, such that $L_{in}(v), L_{out}(v) \subseteq V$ and there is a path from every $x \in L_{in}$ to v and from v to every $x \in L_{out}$. Furthermore, for any two vertices $u, v \in V$, we should have:

$$v \leadsto u \ iff \ L_{out}(v) \bigcap L_{in}(u) \neq \emptyset \qquad (1)$$

The labeling size is defined to be $\sum_{v \in V} |L_{in}(v)| + |L_{in}(v)|$.

To obtain 2-hop labels, for each node, the 2-hop cover is defined as:

Definition 2. Let $G = (V, E)$ be a directed graph. For every $u, v \in V$, let P_{uv} be a collection of paths from u to v and $P = \{P_{uv}\}$. A hop is defined to be a pair (h, u), where h is a path in G and $u \in V$ is one of the endpoints of h (u is called the handle of the hop). A collection of hops H is said to be a 2-hop cover of P if for every $u, v \in V$, such that $P_{uv} \neq \emptyset$, there is a path $p \in P_{uv}$ and two hops $(h_1, u) \in H$ and $(h_2, v) \in H$, such that $p = h_1 h_2$, i.e. p is the concatenation of h_1 and h_2. The size of the cover H, is equal to the number of hops in H.

In the above definition, the set of paths P can be the set of all shortest paths between each pair of labels in the graph. Having produced a 2-hop cover for a graph, the 2-hop labels for the nodes in the graph can be created by mapping a hop with handle v to an item in the label of the node v.

In [10], a polynomial time algorithm is provided that finds a 2-hop cover close to the smallest such cover (larger by a factor of $O(logn)$). Their experiments show that 2-hop covers produced are compact, and each label size is a very small portion of the total number of nodes in the graph.

Having two nodes v and u with labels $(L_{in}(v), L_{out}(v))$ and $(L_{in}(u), L_{out}(u))$, equation 1 can be checked efficiently either in a hash or in a sort based fashion.

3.2 Labeling Scheme for DAGs

In the previous section we discussed a labeling scheme that can be used to answer the reachability question on any directed graph. However, although the time to answer is reasonably small, it is not constant anymore. In this section, we attempt to fix that for special graphs, namely directed acyclic ones. We first argue that many useful structures in our environment fall into this category and then we describe an alternative scheme, which achieves to answer the reachability question in constant time for those graphs.

As mentioned in Section 1, in general, the structural summaries are graph structures, even when derived from tree structures. Nevertheless, in that case, one can prove the following theorem:

THEOREM 1. *The A(k)-Index derived from a tree structure with k lower bounded by the height of the tree, is acyclic.*

Proof Sketch. The proof is based on the observation that if two nodes do not have the same level, then they cannot be k-bisimilar for k larger than the height of the tree (proof is omitted for lack of space).

Moreover, documents where subelements cannot have the same label as their superelements also produce directed and acyclic graphs. From the above discussion, it is obvious that the class of directed acyclic graphs is very important in many real world situations. As one would expect, there are opportunities to further optimize the computation of twig queries over such structures.

In [17], a labeling scheme is discussed to handle the reachability problem in planar directed graphs with one source and one sink. We argue that the directed acyclic graphs of our environment can be mapped to this framework, by introducing, if necessary, a dummy node to play the role of a sink. A similar method (dummy nodes introduction) allows to use techniques for planar graphs into non-planar ones. Moreover, because the reachability property between two nodes is not affected by the introduction of those dummy nodes, the latters can be used to create the labeling and be discarded afterwards. The role of the source can be played by the *root* element in the graph model described in Section 2. In such graphs the following theorem holds [17]:

THEOREM 2. *Let $G = (V, E)$ be a planar, directed and acyclic graph with one source and one sink. For each node $v \in V$, a label L consisting of two numbers a_v and b_v is assigned to it. Then for every pair of nodes u and $v \in V$ with labels (a_u,b_u) and (a_v,b_v) the following holds:*

$$u \leadsto v \ iff \ a_u < a_v \ and \ b_u < b_v \qquad (2)$$

One can always find such a label.

The algorithm to embed the labeling is very simple. Two depth-first searches are performed. A counter assigns the labels to the nodes and is initialized to $n + 1$, where n is the number of vertices in the graph. First, a *left/depth-first* search is performed. A stack is used such that a node is pushed when it is first reached, and popped when all the edges directed from it have been examined. The value that the counter has when a node is popped is assigned to the node as the first number of each label, and the value of the counter is decreased by one. At the end of the traversal, the counter is reinitialized and a second, *right/depth-first*, search is performed. During this search, the second number is assigned to the nodes in the same way as before.

3.3 Twig Processing in XML Graphs

We proceed with first describing our algorithm to handle twig queries when the input is a directed acyclic graph. Subsequently, we explain how the technique can be adapted when the structure is a general directed graph.

3.3.1 Twig Processing in DAGs

The algorithm proceeds in a navigational manner and employs an FSA (Finite State Machine) to identify matching nodes. It adopts ideas that have appeared in [6, 12, 15] to

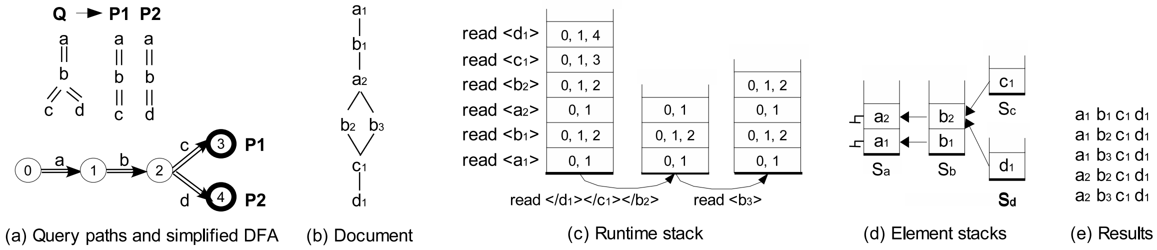

Figure 2: Twig processing example

guide the computation of the FSA, to encode partial results and to output the final results. We assume that the whole twig instance is required as output, however, XPath semantics can also be supported with minimal modifications (e.g. by filtering out the elements that do not participate in the result in a post-processing step that is pipelined with the matching algorithm).

In [12], a technique to answer multiple path queries, by sharing common prefixes over tree structures, is proposed. We describe an adaptation of this technique to efficiently answer twig queries too. Then, we point out necessary modifications to support directed acyclic graph structures.

The twig query is first divided into its constituent path queries, which are then simultaneously handled as in [12]. The algorithm utilizes an FSM, which is derived by the twig query to guide pattern matching. Either an NFA or DFA can be used, and each choice has its advantages and disadvantages. An NFA will possibly have a smaller number of states, while a DFA will avoid the backtracking that the programmatic simulation of an NFA incurs. We decided to adopt the DFA solution as it has been shown to provide performance advantages [23, 15].

The query-to-automaton mapping algorithm is an extension of the one provided in [12], so that the NFA is converted to the minimal equivalent DFA, and is omitted for lack of space. A runtime stack maintaining the DFA states is utilized to buffer previous states of the machine and to allow backtracking to those states, when necessary. Besides the runtime stack, a stack is associated with each query node and is called "elements stack" from now on. The role of elements stacks is to buffer document nodes that compose intermediate results, until the final results that contain them are formulated. The set of those stacks creates a compact representation of partial and total answers as in [6].

The input is accessed in document order, and only elements with the same tag as those of the query nodes are accessed. To achieve that, the document is preprocessed, and the elements are tag divided into sequences sorted in document order. Moreover, each element is augmented with the region-based numbering scheme described in Section 2. That way, the access in document order can be performed by sequentially traversing each sequence and picking, among the current elements, the one with the smallest *left* number. Visiting the elements in such a way guarantees that: (a) descendant nodes will be accessed after their ancestor counterparts and (b) when an ancestor node is found not to be joined with a descendant node, it can be discarded, as it is not going to be joined with any of the subsequent elements.

The algorithm proceeds as follows:

- The DFA is set to its initial state. That state is pushed into the runtime stack and becomes the active state.
- When a new element arrives, the DFA execution follows all matching transitions from all current active states. Moreover, the new element is pushed into its corresponding element stack, possibly triggering the popping of elements that will not participate into new results.
- When the subtree rooted at the element that pushed the current active states into the stack has been processed, the top set of active states is popped and the automaton backtracks.
- When an element with the same tag as a query leaf is about to be pushed into the stack, the path instances being formulated are created in a recursive manner, in a way similar to [6].

As described above, the algorithm produces all the path instances matching the path patterns that constitute the twig query. To construct the twig instances, those path instances need to be combined. An efficient way is by merging them on their common prefixes. This requires the path instances to be sorted in root-to-leaf order. However, the algorithm produces the results in leaf-to-root order. We adapt the blocking technique described in [6] to achieve that. To implement blocking, two linked lists are associated with each node q in the stack: the first, `(S)elf-list`, holds all blocked descendant extensions with root node q, and the second, `(I)nherit-list`, holds all blocked extensions with root elements, descendants of q. When a node is to be popped from its stack, its self- and inherit- lists (in that order) are appended to the inherit lists of the node below it, if one exists; or to the self list of all the associated elements of the stack that corresponds to the parent node in the query. If the element is the root element of the stack corresponding to the root node of the query, then the lists are returned.

When the input structure is a directed acyclic graph, one node may have multiple parents. In this case, region based numbering schemes cannot be used to identify the relative position of two elements, and the scheme described in Section 3.2 is employed instead. Moreover, the document order is not defined anymore. However, one can still get the advantages of that order as follows: if the nodes are accessed in the same order as in the *left/depth-first* search described in 3.2, it still holds that when an ancestor node is found not to be joined with a descendant node, it can be discarded, as it is not going to be joined with any of the subsequent elements. The computation proceeds similarly as in the case of the tree data structure. Furthermore, using the numbering scheme of Section 3.2 and by tag grouping the elements,

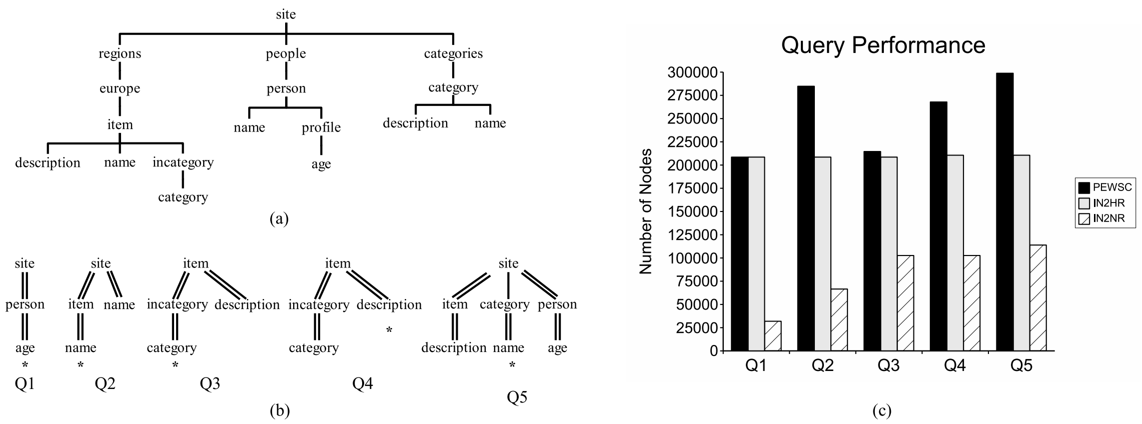

Figure 3: (a) Document DTD, (b) Queries, (c) Performance Evaluation

sorted on the first number of the label, we need only access elements having the same tags as the query nodes, instead of the whole document.

Paths rooted at nodes with multiple parents will be accessed multiple times (once for each parent), i.e. backtracking needs to be performed in the element sequences.

In several cases we can further optimize the performance of the algorithm, if we have a bound on the number of parents of each node. In those cases, and possibly performing a breadth first like access of the graph, we can re-use portions of paths that have already been accessed, and reside in the stacks, without accessing the same nodes again.

EXAMPLE 1. *Figure 2 illustrates a simple example. The query to be evaluated is* `//a//b[.//c]//d`*, which is decomposed in two path queries,* `P1` *and* `P2`*, as showed in Figure 2a. This figure also illustrates the DFA that is generated by grouping the path queries in common shared states (0, 1, and 2). In the DFA: a circle denotes a state; a bold circle denotes a final (accepting) state, which is marked by the IDs of accepted queries; a directed edge is a transition; the symbol on an edge is the input that triggers the transition. This is just a simplified illustration, since the actual DFA has more transitions (one per symbol in the query in each state), not showed here for clarity purposes.*

Consider the document fragment in Figure 2b. While the elements from a_1 to d_1 are read through the left path, the list of current states is pushed to the run-time stack as shown in Figure 2c, and each element is pushed into its corresponding element stack, as shown in Figure 2d. When c_1 is read, the query path `P1` *is matched, and when d_1 is read,* `P2` *is matched. When the subtree with root b_2 finishes processing, the backtracking is performed by popping the run-time stack once for each of its elements, as the second stack in 2c shows, while the elements b_2, c_1, d_1 are popped from the elements stacks. When b_3 is read, its parent is a_2, and it behaves the same way as b_2, such that it is combined with c_1 and d_1 to form new partial results as well.*

3.3.2 Twig Processing in general digraphs

The same algorithm can be utilized for the case of general directed graphs. However, in this case the relative position of two nodes is determined by 2-hop covers, as described in Section 3.1, and access is performed by following the actual edges of the graph. A table similar to the one described in [20] is employed to avoid unnecessary cycles and the performance characteristics are the same as in [20] (where only path queries are discussed).

4. EXPERIMENTAL EVALUATION

In order to investigate the effectiveness of our techniques, we performed a group of experiments over benchmark data. We begin by describing our evaluation methodology and then provide the results of our study.

4.1 Experimental Setup

We compared the performance of our techniques with the technique described in [18] for the evaluation of branch expressions, where twig queries are treated as having a primary path with structural constraints on the nodes that belong to that path. We call this technique *PEWSC* (Path Evaluation With Structural Constraints) from now on. For graph structured data, we used the XMark [1] generator to create an 100Mb database, and we only took into consideration the part of the document described by the DTD graphically illustrated in Figure 3a. For this part of the document, the node *category* residing under the node *incategory* is an *IDREF* referencing the node *category* residing under the node *categories*. Hence, the document is a DAG. To abstract between XML documents and *structural summaries*, we treat *IDREF* edges as any other edge. We run the queries shown in Figure 3b. In this figure, the primary path is marked with a $*$ symbol. We compare the *PEWSC* technique with our algorithm when 2-hop covers are used to determine reachability (we call the technique *IN2HR* - Input Navigation with 2 Hop-cover for Reachability - from now on) and when the labeling described in Section 3.2 is employed (we call this technique *IN2NR* - Input Navigation with 2 Numbers for Reachability - from now on). In this last case we could access only the sequences with the same tags as the query nodes (as already described in Section 3.3).

The performance measure is the total number of nodes visited to answer the query. This choice is justified as in the general case of graph structures it is difficult to make any guarantees about clustering for the index nodes [20]. Consequently, the access to the index nodes is, in general, an I/O operation per index node accessed.

4.2 Performance Results

The results of the experiments described in the previous paragraph are shown in Figure 3c. When the query is a path query (Q1), *PEWSC* and *IN2HR* access the same number of nodes. However, when structural constraints are added (Q2-Q5), *PEWSC* has to rescan parts of the documents to check for their validity, and as a result its performance degrades. Algorithm *IN2NR* performs always much better than the other algorithms as it manages to discard the sequences of elements that do not participate in the results. On the other hand, when the tag of query node exists under different contexts (i.e. nodes with tag *name*), then the algorithm will need to access a possibly large number of elements that do not participate in the result. However, in those cases, indexing as in the case of region-based numbering schemes ([8, 6, 16]) could further improve performance. The encouraging news is that the labeling described in Section 3.2 enables such indexing. We plan to investigate the indexing possibility in the future.

5. CONCLUSION

In this paper, we proposed techniques for evaluating *twig* queries over graph-structured data. We motivated our work both by the graph structure of XML documents, and by the existence of index graphs, namely *structural summaries*, which are graph structures too. We identified an important class of graphs that emerges in many real world situations, namely DAGs, for which we further tailored our approaches. Our preliminary evaluation shows the technique to be a viable solution to the aforementioned problem.

We plan to further evaluate our techniques with a variety of documents as well as *structural summaries*. Moreover, the investigation of the properties of the graph structures that emerge in related applications is an interesting path for future research.

6. REFERENCES

[1] The xml benchmark project. In *Available from http://www.xml-benchmark.org*.

[2] S. Al-Khalifa, H. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient xml query pattern matching. In *Proc. of ICDE*, 2002.

[3] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Minimization of tree pattern queries. In *Proc. of SIGMOD*, 2001.

[4] A. Berglund, S. Boag, D. Chamberlin, M. Fernandez, M. Key, J. Robie, and J. Simeon. Xml path language (xpath) 2.0. Nov 2003.

[5] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. Xquery 1.0: An xml query language. In *W3C Working Draft. Available from http://www.w3.org/TR/xquery*, Nov 2003.

[6] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal xml pattern matching. In *Proc. of ACM SIGMOD*, 2002.

[7] Q. Chen, A. Lim, and K. W. Ong. D(k)-index: An adaptive structural summary for graph-structured data. In *Proc. of SIGMOD*, 2003.

[8] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient structural joins on indexed xml documents. In *Proc. of VLDB*, 2002.

[9] C.-W. Chung, J.-K. Min, and K. Shim. Apex: An adaptive path index for xml data. In *Proc. of SIGMOD*, 2002.

[10] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *Proc. of ACM-SIAM SODA*, 2002.

[11] M. P. Consens and T. Milo. Optimizing queries on files. In *Proc. of SIGMOD*, 1994.

[12] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. M. Ficher. Path sharing and predicate evaluation for high-performance xml filtering. *ACM TODS*, 28(4), Dec 2003.

[13] M. N. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. Xtract: A system for extracting document type descriptors from xml documents. In *Proc. of SIGMOD*, 2000.

[14] R. Goldman and J. Widom. Dataguides: Enabling formulation and optimization in semistructured databases. In *Proc. of VLDB*, 1997.

[15] A. Halverson, J. Burger, L. Galanis, A. Kini, R. Krishnamurthy, A. N. Rao, F. Tian, S. Viglas, Y. Wang, J. F. Naughton, and D. J. DeWitt. Mixed mode xml query processing. In *Proc. of VLDB*, 2003.

[16] H. Jiang, W. Wang, H. Lu, and J. X. Yu. Holistic twig joins on indexed xml documents. In *Proc. of VLDB*, 2003.

[17] T. Kameda. On the vector representation of the reachability in planar directed graphs. *Information Processing Letters*, 3(3), January 1975.

[18] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In *Proc. of SIGMOD*, 2002.

[19] R. Kaushik, P. Bohannon, J. F. Naughton, and P. Shenoy. Updates for structure indexes. In *Proc. of VLDB*, 2002.

[20] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *Proc. of ICDE*, 2002.

[21] C. Koch. Efficient processing of expressive node-selecting queries on xml data in secondary storage: A tree automata-based approach. In *Proc. of VLDB*, 2003.

[22] Q. Li and B. Moon. Indexing and querying xml data for regular path expressions. In *Proc. of VLDB*, 2001.

[23] H. Liefke. Hotizontal query optimization on ordered semistructured data. In *Proc. of WEDDB*, 1999.

[24] T. Milo and D. Suciu. Index structures for path expressions. In *Proc. of ICDT*, 1999.

[25] R. Paige and R. Tarjan. Three partition refinement algorithms. 16(6).

[26] P. R. Rao and B. Moon. Prix: Indexing and querying xml using prufer sequences. In *Proc. of ICDE*, 2004.

[27] H. Wang, S. Park, W. Fan, and P. S. Yu. Vist: A dynamic index method for querying xml data by tree structures. In *Proc. of ACM SIGMOD*, 2003.

[28] K. Yi, H. He, I. Stanoi, and J. Yang. Incremental maintainance of xml structural indexes. In *Proc. of SIGMOD*, 2004.

[29] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. In *Proc. of SIGMOD*, 2001.