Efficient Evaluation of Generalized Path Pattern Queries on XML Data

Xiaoying Wu
NJIT, USA
xw43@njit.edu

Stefanos Souldatos
NTUA, Greece
stef@dblab.ece.ntua.gr

Dimitri Theodoratos
NJIT, USA
dth@cs.njit.edu

Theodore Dalamagas
NTUA, Greece
dalamag@dblab.ece.ntua.gr

Timos Sellis
NTUA, Greece
timos@dblab.ntua.gr

ABSTRACT

Finding the occurrences of structural patterns in XML data is a key operation in XML query processing. Existing algorithms for this operation focus almost exclusively on path-patterns or tree-patterns. Requirements in flexible querying of XML data have motivated recently the introduction of query languages that allow a partial specification of path-patterns in a query. In this paper, we focus on the efficient evaluation of partial path queries, a generalization of path pattern queries. Our approach explicitly deals with repeated labels (that is, multiple occurrences of the same label in a query).

We show that partial path queries can be represented as rooted dags for which a topological ordering of the nodes exists. We present three algorithms for the efficient evaluation of these queries under the indexed streaming evaluation model. The first one exploits a structural summary of data to generate a set of path-patterns that together are equivalent to a partial path query. To evaluate these path-patterns, we extend PathStack so that it can work on path-patterns with repeated labels. The second one extracts a spanning tree from the query dag, uses a stack-based algorithm to find the matches of the root-to-leaf paths in the tree, and merge-joins the matches to compute the answer. Finally, the third one exploits multiple pointers of stack entries and a topological ordering of the query dag to apply a stack-based holistic technique. An analysis of the algorithms and extensive experimental evaluation shows that the holistic algorithm outperforms the other ones.

Categories and Subject Descriptors: H.2.4 [Database Management]: Systems−*query processing, textual databases*

General Terms: Algorithms.

Keywords: XPath query evaluation, XML

1. INTRODUCTION

Since XML gained wide acceptance as the standard for representing web data, it became increasingly important to have a language that provides flexible query capabilities for extracting patterns from XML documents. Traditionally, a structured query language, such as XPath [1] and XQuery [2], is used to specify path-patterns or tree-patterns against XML data. A distinguishing characteristic of these patterns is that they require a total order for the elements in every path of the pattern. Recent applications of XML require querying data where the structure is not fully known to the user, or integrating XML data sources with different structures [11, 15, 19, 27]. In order to deal with these problems, query languages are adopted that relax the structure of a path in a tree pattern. Popular solutions are keyword-based languages for XML [11, 15] or structured query languages for XML extended with keyword search capabilities [4, 19]. However, the first solution has important limitations: (1) structural restrictions cannot be specified in keyword-based queries, and (2) keyword-based queries return many meaningless answers [29, 20]. The drawback of the second solution is that it cannot avoid having a syntax which is complex for the simple user [15, 11, 23]. In any case, in order for all these languages to be useful in practice, they have to be complemented with efficient evaluation techniques.

In this paper we consider a query language that allows a partial specification of path patterns. The queries of this language are not restricted by a total order for the nodes in the path pattern. This language is flexible enough to allow on the one side keyword-style queries with no path structure, and on the other side fully specified path pattern queries. The language is general enough since in the general case, queries can be represented only as graphs and not mere trees. It is important to know that partial path queries *can not* be expressed by path-patterns or tree-patterns. Leaving apart the query answer form question, these queries can be expressed in XPath with reverse axes (such as *parent* axis and *ancestor* axis).

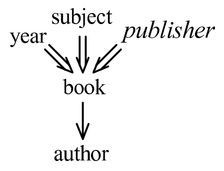


Figure 1: A partial path query

EXAMPLE 1.1. *Consider an XML bibliography which contains several book datasets. These book datasets organize books differently, grouped either by* publisher*, or by* year*, or by* author*, or by* subject*. Suppose that we want to find the authors of a book on the subject XML, published by O'Reilly in 2007. In addition, we have the following structural restrictions: (1)* author *is the child of* book*; and (2)* book *has* subject, year*, and* publisher *as its ancestors. Such query can be easily specified by a partial path query and it is shown as a directed graph in Fig. 1. For simplicity, we omit value predicates in the query. Using reverse axes, we can specify the query in XPath as:* $//book[ancestor :: publisher\ and\ ancestor :: subject\ and\ ancestor :: year]/author$. *However, it is not difficult to see that this query* can not *be expressed by a tree-pattern query.*

Partial path queries constitute an important subclass of XPath queries, where previous evaluation algorithms for tree-pattern queries cannot be applied. Note that Olteanu et al. [24, 23] showed that XPath queries with reverse axes can be equivalently rewritten as sets of tree-pattern queries. However, this transformation may lead to a number of tree-pattern queries which is exponential on the query size. Clearly, it is inefficient to evaluate a partial path query by evaluating an exponential number of tree-pattern queries.

Finding all the occurrences of structural patterns in an XML tree is a key operation in XML query processing. A recent approach for evaluating queries on XML data assumes that the data is preprocessed and the position of every node in the XML tree is encoded [32, 3, 5, 18]. Further, the nodes are partitioned, and an index of inverted lists called *streams* is built on this partition. In order to evaluate a query, the nodes of the relevant streams are read in the pre-order of their appearance in the XML tree. Every node in a stream can be read only once. We refer to this evaluation model as *indexed streaming* model. Algorithms in this model [32, 3, 5, 9, 17, 18, 21, 31, 6, 7] are based on stacks that allow encoding an exponential number of pattern matches in polynomial space. However, these existing algorithms focus almost exclusively on path-patterns or tree-patterns, therefore they cannot be used directly for partial path queries.

The problem. We address the problem of evaluating partial path queries that may contain repeated labels. This problem is complex because the queries in the general case can be directed acyclic graphs (dags). A straightforward approach for dealing with this problem would be to produce for a given partial path query Q, a set of path queries that together compute Q. Such an attempt faces two obstacles: (a) as mentioned above the number of path queries may be exponential on the number of query nodes, and (b) the best known algorithm for evaluating path queries under the indexed streaming model (*PathStack* [5]) does not account for repeated labels in a path query. To the best of our knowledge, there are no previous algorithms for this class of queries in the indexed streaming model.

Contribution. The main contributions of this paper are the following:

- We develop an approach, called *IndexPaths-R*, for evaluating a partial path query Q. *IndexPaths-R* generates a set of path queries P which is equivalent to Q. In order to minimize the number of queries in P, we exploit a structural summary of data, called *Index Tree*, which is similar to a 1-index [22] without extents. This approach guarantees that every path query in P corresponds to an existing pattern in the XML tree and therefore, returns a non-empty answer when evaluated on the XML tree.
- The path queries produced by the previous approach may contain repeated labels. To account for repeated labels in the partial query, we design a new stack-based algorithm, called *PathStack-R* for path queries with repeated labels.
- We also develop another algorithm for partial path queries, called *PartialMJ-R*. *PartialMJ-R* extracts a spanning tree from the query dag. It populates the stacks for all the nodes in the query. However, it evaluates the path queries defined by every root-to-leaf path of the query separately. The results are joined together to produce the final answer. *PartialMJ-R* in general, has lower complexity than *IndexPaths-R*. However, it may produce intermediate solutions, that is, partial solutions that do not make it to the final answer.
- We present a holistic stack-based algorithm for partial path queries, called *PartialPathStack-R*. *PartialPathStack-R* exploits multiple pointers of stack entries and a topological ordering of the nodes in the query dag to match dag patterns directly to the XML tree. *PartialPathStack-R* avoids the problem of intermediate solutions. We provide an analysis of *PartialPathStack-R*, and identify cases when it is asymptotically optimal.
- We implemented all three algorithms, and conducted experiments in benchmark and synthetic data to compare their performance. The experimental results showed that for partial path queries that are not mere path patterns, *PartialPathStack-R* always outperforms the other two, while for path queries, its performance is comparable to *PathStack-R*

Paper outline. The next section discusses related work. Section 2 overviews the XML data model and the partial path query language and its properties. In Section 3, we present our three evaluation algorithms. Section 4 presents and analyses our experimental results. We conclude and discuss future work in the last section.

2. RELATED WORK

In this paper, we focus on the indexed streaming model for the evaluation of queries. Next, we provide an overview on the relevant evaluation techniques on XML data.

Previous papers focused on finding matches of binary structural relationships (a.k.a. structural joins). In [32], the authors presented the Multi-Predicate Merge Join algorithm (MPMGJN) for finding such matches. Al-Khalifa et al. [3] introduced a family of stack-based join algorithms. These algorithms are more efficient compared to MPMGJN, as they do not scan the XML data multiple times. Algorithms for structural join order optimization were introduced in [30]. Structural join techniques can be further improved using various types of indexes [9, 17, 31].

The techniques above can be exploited to evaluate a path-pattern or a tree-pattern query. This task involves the following phases: (a) decomposing the query into binary structural relationships, (b) finding their matches, and (c) stitching together these matches. This approach is inefficient because it generates a large number of intermediate solutions (that is, solutions do not make it to the answer). To deal with this problem, [5] presented two stack-based join algorithms (*PathStack* and *TwigStack*) for the evaluation of path-pattern queries and tree-pattern queries without multiple occurrences of the same node label in the same query path. *PathStack* is shown optimal for path-pattern queries, while *TwigStack* is shown optimal for tree-pattern queries without child relationships. Further, [10] shows that without relaxing the indexed streaming model, it is not possible to develop optimal algorithms for the evaluation of tree-pattern queries.

Several papers focused on extending *TwigStack*. For example, in [21], algorithm *TwigStackList* evaluates efficiently tree-pattern queries in the presence of child relationships. Chen et al. [6] proposed algorithms that handle queries over graph-structured data. Evaluation methods of tree-pattern queries with OR predicates are developed in [16]. In [18], the XR-tree index [17] is used to avoid processing input that does not participate in the answer of the query.

Recently, algorithm *Twig2Stack* [7] is suggested to evaluate tree-pattern queries on XML data without decomposing a query into root-to-leaf path patterns, and it can handle queries with repeated labels. Nevertheless, *Twig2Stack* requires multiple visits to stream nodes, thus, it does not comply with the index streaming model requirements.

Considerable work has been done on the processing of XPath queries when the XML data is not encoded and indexed. For example, [13] suggested polynomial main memory algorithms for answering full XPath queries. Under the XML streaming model, evaluation algorithms for different fragments of XPath, which essentially represent tree-pattern queries, are presented among others in [25, 8, 14].

Partial tree-pattern queries were initially introduced in [27]. Their containment problem was addressed in [28] and semantic issues were studied in [29]. These papers did not focus on the evaluation of these queries. Evaluation algorithms are provided in [26] but for a restricted class of partial path queries that do not allow multiple occurrences of nodes with the same label. In this paper, we show how to efficiently evaluate partial path queries with repeated labels under the index streaming model.

3. PARTIAL PATH QUERY LANGUAGE

In this section, we briefly present the XML data model and the partial path query language.

3.1 Data Model

An XML database is commonly modelled by a tree structure. Tree nodes represent and are labelled by elements, attributes, or values. Tree edges represent element-subelement, element-attribute, and element-value relationships. We denote by $\mathcal{L}$ the set of XML tree node labels. Without loss of generality, we assume that only the root node of every XML tree is labeled by $r \in \mathcal{L}$. Figure 2(a) shows an XML tree. We denote XML tree labels by lower case letters. We use subscripts to distinguish nodes with the same label. The triplets by the nodes in the figure will be explained below. For XML trees, we adopt the region encoding widely used for XML query processing [32, 3, 5, 18]. The region encoding associates with every node a triplet (*begin*, *end*, *level*). The *begin* and *end* values of a node are integers which can be determined through a depth-first traversal of the XML tree, by sequentially assigning numbers to the first and the last visit of the node. The *level* value represents the level of the node in the XML tree. The utility of the region encoding is that it allows efficiently checking structural relationships between two nodes in the XML tree. For instance, given two nodes n_1 and n_2, n_1 is an ancestor of n_2 iff $n_1.begin < n_2.begin$, and $n_2.end < n_1.end$. Node n_1 is the parent of n_2 iff $n_1.begin < n_2.begin, n_2.end < n_1.end$, and $n_1.level = n_2.level - 1$.

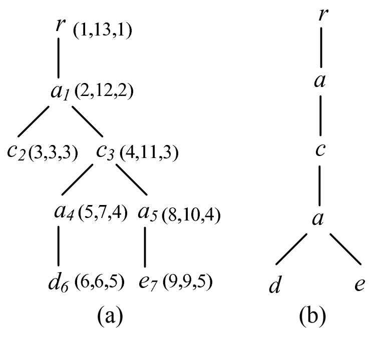


Figure 2: (a) An XML tree T, (b) the index tree of T

3.2 Partial Path Queries

A partial path query specifies a path pattern where the structure (an order among the nodes) may not be fully defined.

Syntax. In order to define these queries, paths or even trees are not sufficient, and we need to employ directed graphs.

DEFINITION 3.1. *A* partial path query *is a directed graph whose nodes are labeled by labels in $\mathcal{L}$, and every node is incident to at least one edge. There is at most one node labeled by r and this node does not have incoming edges. Edges between nodes can be of two types:* child *and* descendant. □

In the rest of the paper, unless stated differently, "query" refers to "partial path query". Query nodes denote XML tree nodes but we use capital letters for their labels. Therefore, a query node labeled by A denotes XML tree nodes labeled by a. In order to distinguish between distinct query nodes with the same label, we use subscripts. For instance, A_3 and A_4 denote two distinct nodes labeled by A. If Q is a query, and X and Y are nodes in Q, the expressions X/Y and $X//Y$ are called *structural relationships* and denote respectively a child and descendant edge from X to Y in Q.

Figure 3 shows four queries. Child (resp. descendant) edges are shown with single (resp. double) arrows. Query Q_1 is a partial path query which is also a path query since the structural relationships in the query induce a total order for the query nodes. Notice that a query graph can be disconnected, e.g. query Q_4 in Figure 3(d). Notice also that no order may be defined between two nodes in a query, e.g. between nodes A and C in Q_3, or between nodes A_1 and A_2 in Q_4.

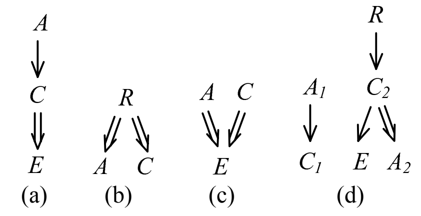


Figure 3: Queries (a) Q_1, (b) Q_2, (c) Q_3, (d) Q_4

Semantics. The answer of a partial path query on an XML tree is a set of tuples. Each tuple consists of tree nodes that lie *on the same path* and preserve the child and descendant relationships of the query. More formally:

An *embedding* of a partial path query Q into an XML tree T is a mapping M from the nodes of Q to nodes of T such that: (a) a node in Q labeled by A is mapped by M to a node of T labeled by a; (b) the nodes of Q are mapped by M to nodes that lie on the same path in T; (c) $\forall$ X/Y (resp. $X//Y$) in Q, $M(Y)$ is a child (resp. descendant) of $M(X)$ in T.

We call *image* of Q under an embedding M, denoted $M(Q)$, a tuple that comprises all the images of the nodes of Q under M. Such a tuple is also called *solution* of Q on T. The *answer* of Q on T is the set of solutions of Q under all possible embeddings of Q to T.

Consider query Q_2 of Figure 3. Notice that Q_2 is syntactically similar to a tree-pattern query (twig). However, the semantics of partial path queries is different: when query Q_2 is a partial path query, the images of the query nodes R, A_1 and C_1 should lie on the same path on the XML tree.

Clearly, we can add a descendant edge from node R to every node that does not have incoming edges in a query without altering its meaning. Therefore, without loss of generality, we assume that a query is a connected directed graph rooted at R. Figure 4 shows the queries of Figure 3 in that form.

Obviously, if a query has a cycle, it is unsatisfiable (that is, it does not have a non-empty answer on any database). Detecting the existence of cycle in a directed graph can be done in linear time on the size of the graph. In the following, we assume that a query is a dag rooted at node R.

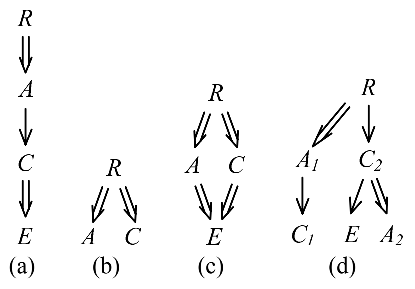


Figure 4: Queries (a) Q_1, (b) Q_2, (c) Q_3, (d) Q_4

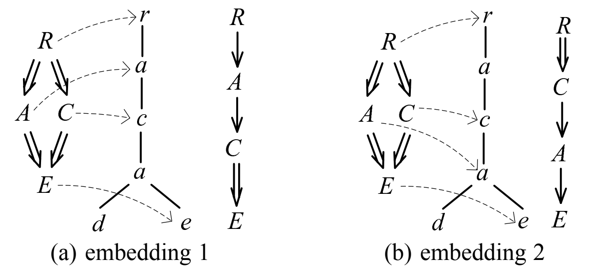


Figure 5: Two embeddings of query Q_3 of Fig. 4(c) on the index tree of Fig. 2(b) and the corresponding path queries

4. QUERY EVALUATION ALGORITHMS

In this section, we present three difference approaches for evaluating partial path queries.

4.1 IndexPaths-R: Leveraging Structural Indexes and Path Query Algorithms

Our first approach, called *IndexPaths-R*, endeavors to leverage existing algorithms for *path* queries [5]. It exploits a structural summary of data, called *index tree*, to generate a set of path queries that together are equivalent to a partial path query. In order to evaluate these queries, it extends Algorithm *PathStack* of [5] so that it can work on path queries with repeated labels.

4.1.1 Generating Path Queries from Index Trees

Given an XML tree T, an *index tree* I for T is an *1-index* [1] [22] without pointers to the XML data (i.e., without extents). Because I has no extents, its size is usually insignicant compared to the data size. Fig. 2(b) shows the index tree for the XML tree of Fig. 2(a).

Given a query Q and an index tree I, we can generate a set P of path queries that is equivalent to Q by finding all the embeddings of Q into I. For example, Fig. 5 shows all the possible mappings of the query Q_3 of Fig. 4(c) on the index tree of Fig. 2(b) (there are two of them) and the corresponding path queries. There is a one to one correspondence between the nodes of a path query and the nodes of Q.

PROPOSITION 4.1. *Let T be an XML tree and I be its index tree. Let also Q be a partial path query and $P = \{P_1, \ldots, P_n\}$ be the set of path queries generated for Q on I. Then, the answer of Q on T is the union of the answers of all the P_is on T.*

The proof is straightforward. Any of the two algorithms presented later in this paper can be used to find the embeddings of a query to an index tree. However, even a naive approach would be satisfactory given the size of an index tree.

In practice, the number of the path queries for the query Q is expected to be small. However, in extreme cases, it can be exponential on the number of nodes in Q. Nevertheless, any one of the path queries represents a pattern that appears in T. Therefore, it will return a non-empty answer when evaluated on T.

4.1.2 An Algorithm for Path Queries with Repeated Labels

Algorithm *PathStack* [5] optimally computes answers for path pattern queries under the indexed streaming model (cf. Section 1). However, the class of queries considered is restricted in that nodes in a query are assumed to have unique labels. The *PathStack* algorithm associates every query node with one stack and one stream of tree nodes. For a path query with repeated labels, extending *PathStack* with multiple streams for each query label (one for each query node with this label) would violate the indexed streaming model requirements, since one stream node might be visited multiple times during the evaluation. Therefore, we designed Algorithm *PathStack-R*, which extends *PathStack* by allowing nodes with the same label to share the same stream.

[1] 1-indexes coincide with *strong DataGuides* [12] when the data is a tree.

Notation. Let Q be a path query, q be a node in Q, and l be a label in Q. Function *nodes(Q)* returns all nodes of Q; *label(q)* returns the label of q in Q. Boolean function *isLeaf(q)* returns true iff q is a leaf node in Q. Function *parent(q)* returns the parent of q in Q; *occur(l)* returns all nodes in Q labeled by l; *label(Q)* returns the set of node labels in Q.

With every distinct node label l in Q, we associate a stream T_l of all nodes (positional representation) labeled by l in the XML tree. The nodes in the stream are ordered by the their *begin* field (cf. Section 3.1). To access sequentially the nodes in T_l, we maintain a cursor C_l, initially pointing to the first node in T_l. For simplicity, C_l may alternatively refer to the node pointed by cursor C_l in T_l. Operation *advance(C_l)* moves C_l to the next node in T_l. Function *eos(C_l)* returns true if C_l has reached the end of T_l. *C_l.begin* denotes the *begin* field in the positional representation of node C_l.

We associate a stack S_q with every query node q in Q. A stack entry in S_q consists of a pair: (positional representation of node from $T_{label(q)}$, pointer to an entry in the stack of q's parent). The expression $S_q.k$ denotes the entry at position k of stack S_q. We use the following stack operations: *push(S_q,entry)* pushes *entry* on S_q; *pop(S_q)* pops the top entry from S_q; and *top(S_q)* returns the position of the top entry of S_q.

PathStack-R. Algorithm *PathStack-R* is presented in Algorithm 1. *PathStack-R* gradually constructs matchings to Q and compactly encodes them in stacks, by iterating through stream nodes in ascending order of the *begin* values. Thus, the query nodes are matched from the query root to the query leaf.

In line 2, *PathStack-R* identifies the stream node to be processed. Line 3 calls *cleanStacks* to remove partial matchings that cannot become final solutions.

Line 5 (*moveStreamToStack*) is an important step, which is also the key difference from *PathStack*. It determines whether the identified stream node C_l qualifies for being pushed on stack S_q, where q is a query node labeled is L. The stream node C_l can be pushed on stack S_q iff (1) q is the root, or (2) q is not the root and the structural relationship between C_l and the top stack entry of q's parent p satisfies the structural relationship between p and q in the query. This ensures that stream nodes that do not contribute to answers will not be stored in the stacks and processed. Since Q can contain repeated labels, it is possible that l matches more than one node in Q and thus C_l can be pushed to more than one stack. Note that the order of pushing C_l to stacks is important. As discussed, in order

Algorithm 1 PathStack-R

```
1 while ¬end() do
2     $l$ = getNextQueryLabel()
3     cleanStacks($C_l$)
4     for every $q \in$ occur($l$) in the descending order do
5         moveStreamToStack($l$, $q$)
6         if isLeaf($q$) then
7             showSolutions($S_q$)
8             pop($S_q$)
9     advance($C_l$)
Function end()
1 return $\forall\ q \in$ nodes($Q$): isLeaf($q$) $\Rightarrow$ eos($C_{label(q)}$)
Function getNextQueryLabel()
1 return $l \in$ labels($Q$) such that $C_l$.begin is minimal
Procedure cleanStacks($C_l$)
1 for ($q$ in nodes($Q$)) do
2     pop all entries in $S_q$ whose nodes are not ancestors of $C_l$ in the XML tree
Procedure moveStreamToStack($l$, $q$)
1 $p$ = parent($q$)
2 if ($q$ is not the query root and empty($S_p$)) then
3     return
4 if ($q$ is the query root) or ($p//q \in Q$ or $S_p$.top($S_p$).level = $C_l$.level-1) then
5     push($S_q$, ($C_l$, pointer to $S_p$.top($S_p$)))
```

to determine if C_l can be pushed on a stack, the parent stack will be checked. In order to prevent C_l from 'seeing' its own copy in the parent stack, *moveStreamToStack* should be called on the matched nodes in their descending order in Q (line 4). We illustrate this by an example below.

Whenever an incoming stream node C_l is pushed onto the stack of the leaf node, procedure *showSolutions* iteratively produces solutions encoded in stacks (line 6-8). The details are omitted here and can be found in [5].

EXAMPLE 4.1. *Consider a path query Q_5 on the data path shown in Fig. 6. The input streams for a and b are: T_a=$\{a_1, a_2\}$, T_b = $\{b_1, b_2, b_3\}$. Initially, the two cursors C_a and C_b point to a_1 and b_1 respectively. When a_1 is read, it is not pushed on S_{A2}, since the push condition (line 2 in* moveStreamToStack*) is not satisfied: the stack S_{B2} is empty. When b_2 is read, it is pushed first on stack S_{B2} and then on S_{B1}. If we do not follow this sequence and push b_2 on S_{B_1} first, then b_2 will not be pushed on S_{B2} anymore (line 4 in* moveStreamToStack*). This would result in missing one solution for Q_5. Finally, after a_2 is read and pushed on S_{A2}, procedure* showSolutions *is invoked to output the query answer $\{a_1b_1b_2a_2, a_1b_2b_3a_2\}$.*

Analysis of IndexPaths-R. Given a node q in a path query Q, we call the path from the root of Q to q the $prefix$ path of q. Given a stream node x of an XML tree T, we say that x *matches* q iff x is the image of q under an embedding of the prefix path of q to T.

PROPOSITION 4.2. *Let q be a query node in Q and x be a stream node with the same label. x is pushed on stack S_q by Algorithm* PathStack-R *iff x matches q.*

As a result of Proposition 4.2, Algorithm *PathStack-R* will find and encode in stacks all the partial or complete (q is a leaf node in Q) solutions involving x. When at least one complete solution is encoded in the stacks, procedure $showSolutions$ is invoked to output them. Therefore, Algorithm *PathStack-R* correcly finds all the solutions to Q.

Given a path query Q and an XML tree T, let $input$ denote the sum of sizes of the input streams, $output$ denote the size of the answers of Q on T, $height$ denote the height of T, and $maxOccur$ denote the maximum number of occurrences of a query label in Q.

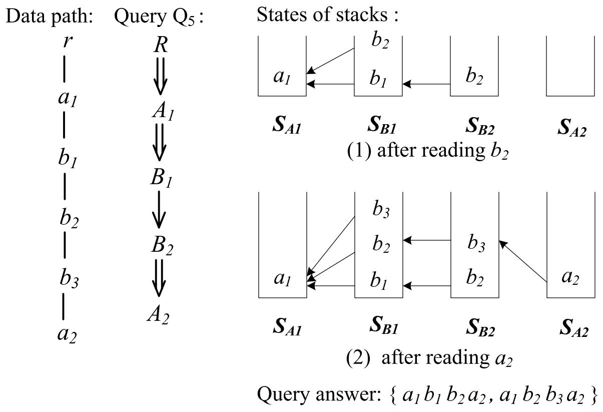


Figure 6: PathStack-R running example

The time complexity of *PathStack-R* depends mainly on the number of calls to *moveStreamToStack*, and the time to produce answers. For each stream node, procedure *moveStreamToStack* is invoked at most $maxOccur$ times, and each invocation takes O(1). As Algorithm *PathStack-R* does not generate any intermediate solutions, the time it spends on producing all the answers is proportional to $output$. Therefore, *PathStack-R* has time complexity $O(input \times maxOccur + output)$.

The space complexity of *PathStack-R* depends mainly on the number of stack entries at any given point in time. Since the worst-case size of any stack in *PathStack-R* is bounded by min($height$, $input$), *PathStack-R* has space complexity $O(min(height, input) \times |Q|)$.

THEOREM 4.1. Algorithm *PathStack-R* correctly evaluates path queries with repeated labels. It has time complexity $O(input \times maxOccur + output)$ and space complexity $O(min(height, input) \times |Q|)$.

The time complexity of *IndexPaths-R* is the product of the time complexity of *PathStack-R* and the number of path queries in P.

4.2 PartialMJ-R: a Partial Path Merge Join Algorithm

Given a partial path query Q, Algorithm *PartialMJ-R* extracts a spanning tree of Q. Then, it finds matches for all root-to-leaf paths of the spanning tree against the XML tree by using Algorithm *PathStack-R*.

The solutions for each path of the spanning tree are merge-joined by guaranteeing that (a) they lie on the same path in the XML tree, and (b) they satisfy the structural relationships that appear in the query graph and not in the spanning tree.

Fig. 7(b) shows the graph of a query Q_6, and Fig. 7(c) shows a spanning tree Q_s of Q_6. Edge $B_5//A_6$ of Q_6 is missing from Q_s. Any two solutions from the two root-to-leaf paths of Q_s that are on the same path of the XML tree can be merged to produce a solution for Q_6, if they satisfy the identity conditions on R and A_1 and the structural condition $B_5//A_6$.

PartialMJ-R is shown in Algorithm 2. Compared to *PathStack-R*, *PartialMJ-R* has two additional important steps: (1) Line 1 produces a spanning tree for the given partial path query and records the set of structural relationships presented in the query but are missing in the spanning tree; and (2) whenever a set of solutions for a root-to-leaf path in the spanning tree is produced, they are merge-joined with solutions produced earlier using Procedure *join-*

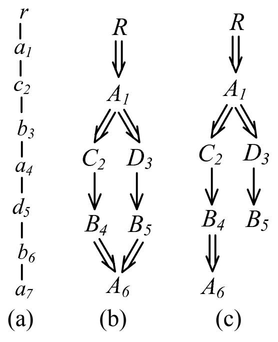


Figure 7: (a) data path (b) query Q_6 (c) Q_6's spanning tree Q_s

Algorithm 2 PartialMJ-R

```
1  create a spanning tree $Q_s$ of $Q$. $E$ denotes the set of edges in $Q$ which
   do not appear in $Q_s$
2  while ¬end() do
3      $l$ = getNextQueryLabel()
4      cleanStacks($C_l$)
5      for every $q \in$ occur($l$) in the descending order do
6          moveStreamToStack($l$, $q$)
7          if isLeaf($q$) then
8              showSolutionsWithBlocking($S_q$)
9              joinPathSolutions()
10             pop($S_q$)
11     advance($C_l$)
```

Procedure joinPathSolutions()

```
1  merge-join the solutions of the root-to-leaf paths of $Q_s$ that are pro-
   duced in the current loop (lie on the same path of the XML tree) and
   satisfy the structural relationships in $E$
```

PathSolutions (line 9). When Q is a path, Algorithm *PartialMJ-R* reduces to *PathStack-R*.

Compared to the approach *IndexPath-R*, Algorithm *PartialMJ-R* also evaluates the query by populating query stacks in one single pass of input streams. Nevertheless, this approach may generate many intermediate solutions that are not part of any final answer. A solution is called *intermediate*, if either it cannot be merged with any other solutions on a same data path or it does not satisfy structural conditions in Q that are not present in Q_s.

Considering evaluating the query Q_6 of Fig. 7(b) on the data of Fig. 7(a), four partial solutions are produced: $\{ra_1c_2b_3a_4, ra_1d_5b_6, ra_4d_5b_6, ra_1c_2b_3a_7\}$ in order of their construction. Among them, the first and fourth are solutions for the path $RA_1C_2B_4A_6$ and the rest are solutions for the path $RA_1D_3B_5$. The third solution cannot be merged with the first and fourth solutions, since the identity condition on R and A_1 is not satisfied. The first cannot be merged with the second, since the structural relationship between b_6 and a_4 does not satisfy $B_5//A_6$. Therefore $ra_1c_2b_3a_4$ and $ra_4d_5b_6$ are intermediate solutions. The answer of Q_6 is $\{ra_1c_2d_5b_3b_6a_7\}$.

Clearly, the intermediate solutions affect negatively on the worst case time and space complexity for *PartialMJ-R*.

4.3 PartialPathStack-R: a Holistic Algorithm

To overcome the problem of intermediate solutions of Algorithm *PartialMJ-R*, we developed a holistic stack-based algorithm called *PartialPathStack-R* for the evaluation of partial path queries. In contrast to *PartialMJ-R*, *PartialPathStack-R* does not decompose a query into paths, but tries to match the query graph to an XML tree as a whole. Also, unlike *PathStack-R*, *PartialPathStack-R* exploits multiple pointers per stack entry to avoid redundantly storing multiple copies of stream nodes in different stacks.

4.3.1 Preliminaries

A partial path query Q can be represented as a dag rooted at R. Let q be a node and l be a label in Q. Since now we are dealing with graphs, we replace the functions *isLeaf(q)* and *parent(q)* of Section 4.1.2 with the functions *isSink(q)* and *parents(q)* respectively. An additional function *children(q)* returns the set of child nodes of q in Q. The rest of the functions are similar to those defined for Algorithm *PathStack-R*.

As with *PathStack-R*, we associate every distinct node label l with a stream T_l and maintain a cursor C_l for that stream. Unlike *PathStack-R*, we now associate a stack S_l with every distinct node label l. The structures of a stack entry are now more complex in order to record additional information.

Before describing the data structures used by stack entries, we define an important concept, which is key to the understanding of the *PartialPathStack-R* algorithm.

DEFINITION 4.1. *Let Q be a partial path query, q be a node in Q, and T be an XML tree. The sub-dag of Q that comprises q and all its ancestor nodes is called* prefix *query of q and is denoted as Q_q. We say that a node x in T* plays the role *of q if x is the image of q under an embedding of Q_q to T.*

For every role that a stream node can play, we maintain a chain of pointers and record in the chain the nodes that play this role.

Let $p_1, \ldots, p_k$ be the parent nodes of all nodes labeled by l in Q. Note that it is possible that for some or all of these p_is, *label(p_i)* = l. An entry e in stack S_l has the following three fields:
(1) n_l: the positional representation of a node from T_l.
(2) $ptrs$: a set of k pointers labeled by $p_1, \ldots p_k$. Each pointer denotes a position in a stack. A pointer labeled by p_i points to an entry in stack $S_{label(p_i)}$. It is possible that a pointer is *null*. It is also possible that an entry has multiple pointers to the same stack. In this case, these pointers are labeled by different query nodes.
(3) $prevPos$: an array of size $|occur(l)|$. Given a node $q \in occur(l)$, $prevPos[q]$ records the position of the highest entry in S_l below e that plays the role of q (i.e., $prevPos[q]$ points to the previous entry in the chain). If q is the only node labeled with l, $prevPos[q]$ refers to the entry just below e.

The bottom of each stack has position 1. The expression $S_l.k$ denotes the entry at position k of stack S_l. The expression $S_l.k.p_i$ refers to the position of the entry in stack $S_{label(p_i)}$ pointed to by the pointer labeled p_i in the entry $S_l.k$.

With every stack S_l, we associate an array $lastPos_l$ of size $|occur(l)|$. The field $lastPos_l[q]$ records the position of the highest stack entry in S_l that plays the role of node q (the beginning of the chain). If q is the only node labeled with l, $lastPos_l[q]$ refers to the top entry in stack S_l.

4.3.2 PartialPathStack-R

Algorithm *PartialPathStack-R* is presented in Algorithm 3. A key feature of *PartialPathStack-R* is that it employs a topological order of the query nodes, i.e., a linear ordering of nodes which respects the partial order induced by the structural relationships of the query. This order is exploited when producing query answers. In the algorithm, query nodes of Q are indexed from 1 to $|Q|$ based on their position in the topological order.

Algorithm *PartialPathStack-R* calls in line 5 procedure *getRoles(l)*. For a stream node C_l under consideration, *getRoles(l)* finds all the roles that C_l can play and records the information in an object which is finally returned to the algorithm. More specifically, for each query node q in $occur(l)$, C_l plays the role of q iff for each q's parent p, there exists an entry e in the stack $S_{label(p)}$ such that the structural relationship between e and C_l satisfies the structural relationship between p and q in the query (line 8-17). Note that we can

Algorithm 3 PartialPathStack-R

```
1  create a topological order 1..$n$ of the query nodes in $Q$, and identify
   each node by its topological order.
2  while ¬end() do
3     $l$ = getNextQueryLabel()
4     cleanStacks($C_l$)
5     $e$ = getRoles($l$)
6     if ($R == l$ or $e.ptrs \neq \emptyset$) then
7        push($S_l$, $e$)
8        $O = \emptyset$
9        for ($q \in$ occur($l$)) do
10          if (isSink($q$) and lastPos$_l[q] \neq 0$) then
11             $O$ += $\{q\}$
12       if ($O \neq \emptyset$ and $\forall\ q \in$ nodes($Q$): isSink($q$) $\Rightarrow$ lastPos$_{label(q)}[q]$
         $\neq 0$) then
13          if ($n \in O$) then
14             outputSolutions($O$, $n$, lastPos$_{label(n)}[n]$)
15          else
16             $i$ = lastPos$_{label(n)}[n]$
17             repeat
18                outputSolutions($O$, $n$, $i$)
19                $i = S_{label(n)}$.i.prevPos[$n$]
20             until ($i$==0)
21    advance($C_l$)
```

Function 4 getRoles(l)

```
1  $e$ = newStackObject($l$)
2  if ($R == l$ ) then
3     lastPos$_R$[1]=1 /*we are at the root node*/
4  else
5     for ($q \in$ occur($l$)) do
6        $pptrs = \emptyset$
7        $hasRole = true$
8        for ( $p \in$ parents($q$)) do
9           $i$ = lastPos$_{label(p)}[p]$
10          $entry = S_{label(p)}.i$
11          if ($i \neq 0$) then
12             if ($p$/$q \in Q$) and ($entry.level \neq C_l.level$+1) then
13                $hasRole = false$
14          else
15             $hasRole = false$
16          if ($hasRole$) then
17             $pptrs$ += {pointer labeled by $p$ to $entry$}
18       if ($hasRole$) then
19          $e.ptrs$ += $pptrs$
20          $e$.prevPos[$q$] = lastPos$_l[q]$
21          $lastPos_l[q]$ = top($S_l$)+1
22 return $e$
```

efficiently find the entry e through the value of lastPos$_{label(p)}[p]$ (line 10) without exhaustively visting the stack entries. If C_l plays the role of q, a set of labeled pointers to all q's parents is generated and recorded (line 19). The values of $lastPos_l[q]$ and $prevPos[q]$ are accordingly updated as well (line 20-21).

PartialPathStack-R uses the information returned by $getRoles(l)$ to determine if the stream node C_l is qualified for being pushed on its stack S_l. The stream node C_l can be pushed on its stack iff it plays at least one role w.r.t a query node in $occur(l)$ (line 6-7). By populating stacks in this way, we ensure that at any given point in time, the nodes in stacks represent partial solutions that could be further extended to final solutions as the algorithm goes on.

The timing for producing solutions is important in order to avoid generating duplicate solutions. Whenever node C_l that plays the role of a sink node in the query (lines 9-11) is pushed on a stack, and for every sink node in the query there is an entry in the stacks that plays this role (line 12), it is guaranteed that the stacks contain at least one solution to the query. Subsequently, procedure

Procedure 5 outputSolutions($outputSinkNodes$, $curNode$, $stackPos$)

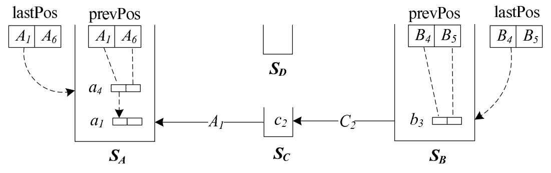
```
1  solution[$curNode$] = $stackPos$
2  $m = curNode$-1
3  if ($curNode$ = 1) then
4     output($S_{label(1)}$.solution[1],...,$S_{label(n)}$.solution[$n$])
5  else if ($m \in outputSinkNodes$) then
6     outputSolutions($outputSinkNodes$, $m$, $lastPos_{label(m)}[m]$)
7  else if (isSink($m$)) then
8     $i$ = lastPos$_{label(m)}[m]$
9     repeat
10       outputSolutions($outputSinkNodes$, $m$, $i$)
11       $i = S_{label(m)}$.i.prevPos[$m$]
12    until ($i$==0)
13 else
14    $i$ = minarg$_c$ $\{S_{label(c)}$.solution[$c$].$m\}$, $c \in$ children($m$)
15    repeat
16       outputSolutions($outputSinkNodes$, $m$, $i$)
17       $i = S_{label(m)}$.i.prevPos[$m$]
18    until ($i$==0)
```

outputSolutions (Procedure 5) is invoked to output all the solutions that involve C_l (lines 14 and 18). Note that since every time solutions are produced, they involve the newly pushed node C_l, *PartialPathStack-R* does not generate duplicate solutions.

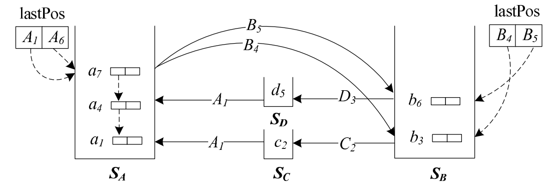
Procedure *outputSolutions* gradually produces the nodes in each solution in an order that corresponds to the reverse topological order of the query nodes. This way, the image of a query node is produced in a solution after the images of all its descendant nodes in the query are produced. Procedure *outputSolutions* takes three parameters: *outputSinkNodes*, *curNode*, and *stackPos*. A solution under construction by *outputSolutions* is recorded in an array $solution$ indexed by the query nodes. The image of *curNode* recorded in $solution[curNode]$ is the position of an entry in stack $S_{label(curNode)}$. WHen recursively processing the next query node *curNode-1*, denoted as m, we consider three cases:
(1) If m is in *outputSinkNodes* (which implies m is a sink node), we output solutions that involve the last qualified entry in stack $S_{label(m)}$ (line 6).
(2) If m is a sink node not in *outputSinkNodes*, we output solutions that involve all the qualified entries in stack $S_{label(m)}$ (line 7-12).
(3) If node m is an internal query node, the highest entry e in stack $S_{label(m)}$ that can be used in a solution as an image of m is the lowest ancestor in the XML tree of the images of the child nodes of m in the query. Since the child nodes of m have already been processed, their images are recorded in $solution$. Entry e is identified by the lowest position in stack $S_{label(m)}$ pointed to by pointers from stack entries that are images of the child nodes of m in the solution under construction (line 14). The chain of entries that play the role of m in stack $S_{label(m)}$ starting with e are used as images of m for constructing solutions (lines 15-18).

Note that if there is a child relationship from m to another node, only a single recursive call to *outputSolutions* needs to be invoked. We omit the details in the algorithm for the interest of space.

EXAMPLE 4.2. *Fig. 8 shows a running example for* Partial-PathStack-R*, where the stack for the query root R is not shown for simplicity. We do not show the $lastPos$ and $prevPos$ of query nodes with a single occurrence in the query. We use for Q_6 the topological order: R, A_1, C_2, D_3, B_4, B_5, A_6. The input streams are T_a: $\{a_1, a_4, a_7\}$, T_b: $\{b_3, b_6\}$, T_c: $\{c_2\}$, and T_d: $\{d_5\}$. The initial value for the input stream cursors C_a, C_b, C_c, and C_d in that order is a_1, b_3, c_2, d_5. Fig. 6(a) shows the state of the stacks after a_4 is read. At this point, there is no entry in stack S_a that plays the role of A_6. Therefore $lastPos_a[A_6]$ is 0. Similarly, $lastPos_b[B_5]$*



(a) the state of stacks after reading a_4



(b) the state of stacks after reading a_7

Figure 8: PartialPathStack-R on Q_6 and data in Fig. 7(a)

for stack S_b is 0, since the entry b_3 can only play the role of B_4. Since a_4 plays the role of A_1, its $prevPos[A_1]$ field points to the lower entry a_1 in S_a that plays the role of A_1. Fig. 6(b) shows the state of stacks after a_7 is read. As a_7 plays the role of A_6, the entry for a_7 has two outgoing pointers which respectively point to b_3 and b_4 in stack S_b, and its stack position is recorded in $lastPos_a[A_6]$. Since A_6 is a sink node of Q_6, a_7 triggers the generation of solutions. Note that when A_1 is processed by outputSolutions*, a_1 is chosen as a value for A_1 in the solution under construction, since a_1 has lower position than a_4 in stack S_a (line 13 in* outputSolutions*). The final answer for Q_6 is $\{ra_1c_2d_5b_3b_6a_7\}$.*

4.3.3 Analysis of PartialPathStack-R.

PROPOSITION 4.3. *A stream node x is pushed on stack S_l iff x plays the role of a query node labeled by L.*

It is easy to see that for each stream node x stored in stack S_l, the above proposition along with Definition 4.1 ensures that Algorithm *partialPathStack-R* will find all solutions in which x matches one of its roles.

Given a query Q and an XML tree T, let $indegree$ denote the maximum number of incoming edges to a query node and $outdegree$ denote the maximum number of outgoing edges from a query node. Other parameters are the same for the analysis of *IndexPaths-R*.

Since *PartialPathStack-R* does not generate any intermediate solutions, the time complexity of *PartialPathStack-R* depends mainly on the number of calls to *getRoles* and *outputSolutions*. For each stream node, *getRoles* takes time $O(indegree \times maxOccur)$. Procedure *outputSolutions* spends O($outdegree$) on each query node, since in line 13 it computes the lowest stack position among the entries of the children of the query node. Thus it takes $O(outdegree \times output)$ to produce all the solutions. Therefore, *PartialPathStack-R* has time complexity $O(input \times indegree \times maxOccur + outdegree \times output)$.

The space complexity depends mainly on how many entries are stored in stacks at a given point in time and the number of pointers associated with these entries. The total number of stack entries at any time is $O(min(height, input))$. For each stack entry, the maximum number of outgoing pointers is $O(indegree \times maxOccur)$. Therefore, the number of pointers in stacks is bounded by $min(height, input) \times indegree \times maxOccur$.

THEOREM 4.2. Algorithm *PartialPathStack-R* correctly evaluates a partial path query Q on an XML tree T. The algorithm uses $O(min(height, input) \times indegree \times maxOccur)$ space and $O(input \times indegree \times maxOccur + outdegree \times output)$ time.

Based on the above theorem, *PartialPathStack-R* is asymptotically optimal if the $indegree$, $outdegree$, and $maxOccur$ of the query are bounded by a constant. Clearly, for the case of a query whose dag is a tree, only the $outdegree$ and $maxOccur$ need to be bounded by some constants for *PartialPathStack-R* to be asymptotically optimal.

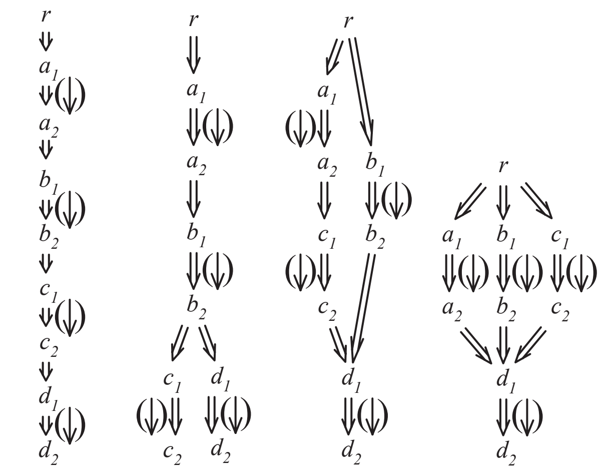
5. EXPERIMENTAL EVALUATION

We ran a comprehensive set of experiments to measure the performance of *IndexPaths-R*, *PartialMJ-R* and *PartialPathStack-R*. In this section, we report on their experimental evaluation.

Setup. We evaluated the performance of the algorithms on both benchmark and synthetic data. For benchmark data, we used the Treebank XML document[2]. The XML tree of Treebank consists of around 2.5 million nodes having 250 distinct element tags, and its depth is 36. For synthetic data, we generated random XML trees, using IBM's AlphaWorks XML generator[3]. The number of distinct element tags used in all synthetic trees was fixed to 5. For each measurement on synthetic data, $five$ different XML trees with the same number of nodes were used. Each value displayed in the plots is averaged over these $five$ measurements.

Fig. 9 shows the queries used in our experiments. Queries Q_1 to Q_4 include only descendant relationships, while queries Q_5 to Q_8 include child relationships as well. Our query set comprises a full spectrum of partial path queries, from simple path-pattern queries to complex dag queries. The queries are appropriately modified for the Treebank dataset, so that they can all produce solutions. Thus, node d_2 is removed, and node labels r, a, b, c and d are changed to $File$, S, VP, NP and NN, respectively.

We implemented all algorithms in C++, and ran our experiments on a dedicated Linux PC (AMD Sempron 2600+) with $2GB$ of RAM.



(a) $Q_1(Q_5)$ (b) $Q_2(Q_6)$ (c) $Q_3(Q_7)$ (d) $Q_4(Q_8)$

Figure 9: Partial path queries.

Execution time on fixed datasets. We measured the execution time of *IndexPaths-R*, *PartialMJ-R* and *PartialPathStack-R* for evaluating all queries in Fig. 9 on Treebank and on two synthetic

[2]http://www.cis.upenn.edu/ treebank

[3]www.alphaworks.ibm.com/tech/xmlgenerator

datasets, $SD1$ and $SD2$. All trees in $SD1$ have depth 12 and consist of 1.5 million nodes. All trees in $SD2$ have depth 20 and consist of 1 million nodes. For path-pattern queries Q_1 and Q_5, we also measured the execution time of *PathStack-R*.

Fig. 10(a), 10(b) and 10(c) present the evaluation results. Fig. 10(d) shows the number of results obtained per query in each dataset. Algorithm *PartialPathStack-R* is efficient for all types of queries.

As expected, *IndexPaths-R*, *PartialMJ-R* and *PartialPathStack-R* perform almost as fast as *PathStack-R* in the case of the path-pattern queries Q_1 and Q_5.

The execution time of algorithm *IndexPaths-R* is high for queries with a large number of path queries generated from the index tree, that is, for queries Q_3, Q_4, Q_7 and Q_8.

The performance of both *PartialMJ-R* and *PartialPathStack-R* in all datasets is affected by the number of solutions. This confirms our complexity results that show dependency of the execution time on the input and output size. In the case of queries Q_2 and Q_4 on $SD2$ (Fig. 10(c)), where the number of solutions is high (Fig. 10(d)), the execution time of *PartialMJ-R* strongly increases.

Execution time varying the input size. We measured the execution time of *IndexPaths-R*, *PartialMJ-R* and *PartialPathStack-R* for evaluating queries Q_2, Q_4 and Q_8 of Fig. 9 on random XML trees of various sizes. Fig. 11 presents the execution time and the results obtained on XML trees whose node stream sizes vary from 0.5 to 2.5 million nodes. The depth of the XML trees is 12.

PartialPathStack-R clearly outperforms *IndexPaths-R* and *PartialMJ-R* regarding queries Q_2 and Q_4 (*IndexPaths-R* is out of range in Fig. 11(b)). For query Q_8, the performance of *PartialPathStack-R* is similar to that of *PartialMJ-R*. This is due to the small number of results produced (Fig. 11(f)).

An increase in the input size results in an increase in the output size (Figures 11(d), 11(e) and 11(f)). Also, when the input and the output size go up, the execution time of all algorithms increases (Figures 11(a), 11(b) and 11(c)). This confirms the complexity results that show dependency of the execution time on the input and output size.

In the experimental evaluation of query Q_4, the output size (Fig. 11(e)) increases sharper than in the evaluation of query Q_2 (Fig. 11(d)). The execution time of *PartialPathStack-R* is only slightly higher in the evaluation of Q_4 (Fig. 11(b)) than in the evaluation of Q_2 (Fig. 11(a)). In contrast, the execution time of *PartialMJ-R* is strongly affected.

6. CONCLUSION

In this paper, we addressed the problem of evaluating partial path queries with repeated labels under the indexed streaming model. Partial path queries generalize path-pattern queries and are useful for integrating XML data sources with different structures and for querying XML documents when the structure is not fully known to the user. Partial path queries are expressed as dags and can be specified in XPath with reverse axes.

We designed three algorithms for evaluating partial path queries on XML data. The first algorithm, *IndexPaths-R*, exploits a structural summary of data to generate an equivalent set of path patterns of a partial path query and then uses a stack-based algorithm, *PathStack-R*, for evaluating path-pattern queries with repeated labels. The second algorithm, *PartialMJ-R*, extracts a spanning tree from the query dag and uses *PathStack-R* to find the matches of the root-to-leaf paths in the tree. These matches are progressively merge-joined to compute the answer. Finally, the third algorithm, *PartialPathStack-R*, exploits multiple pointers of stack entries and a topological ordering of the nodes in the query dag to apply a stack-based holistic technique. To the best of our knowledge, *PartialPathStack-R* is the first holistic algorithm that evaluates partial path queries with repeated labels. We analyzed to the three algorithms and conducted extensive experimental evaluations to compare their performance. Our results showed that *PartialPathStack-R* has the best theoretical value and has considerable performance superiority over the other two algorithms.

We are currently working on extending our approaches for evaluating partial tree-pattern queries under the indexed streaming model.

7. REFERENCES

[1] XML Path Language (XPath). World Wide Web Consortium site, W3C. http://www.w3.org/TR/xpath20.

[2] XML Query Language (XQuery). World Wide Web Consortium site, W3C. http://www.w3.org/XML/Query.

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

[4] S. Amer-Yahia, C. Botev, and J. Shanmugasundaram. Texquery: a full-text search extension to xquery. In *WWW*, 2004.

[5] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD*, 2002.

[6] L. Chen, A. Gupta, and M. E. Kurul. Stack-based algorithms for pattern matching on DAGs. In *VLDB*, 2005.

[7] 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. In *VLDB*, 2006.

[8] Y. Chen, S. B. Davidson, and Y. Zheng. An efficient XPath query processor for XML streams. In *ICDE*, 2006.

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

[10] B. Choi, M. Mahoui, and D. Wood. On the optimality of holistic algorithms for twig queries. In *DEXA*, 2003.

[11] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSearch: A semantic search engine for XML. In *VLDB*, 2003.

[12] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, 1997.

[13] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. *ACM Trans. Database Syst.*, 2005.

[14] G. Gou and R. Chirkova. Efficient algorithms for evaluating XPath over streams. In *SIGMOD*, 2007.

[15] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword proximity search on XML graphs. In *ICDE*, 2003.

[16] H. Jiang, H. Lu, and W. Wang. Efficient processing of XML twig queries with or-predicates. In *SIGMOD*, 2004.

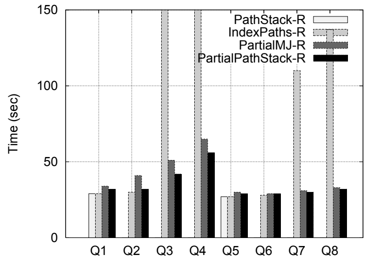
[17] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-Tree: Indexing XML data for efficient structural joins. In *ICDE*, 2003.

[18] H. Jiang, W. Wang, H. Lu, and J. X. Yu. Holistic twig joins on indexed XML documents. In *VLDB*, 2003.

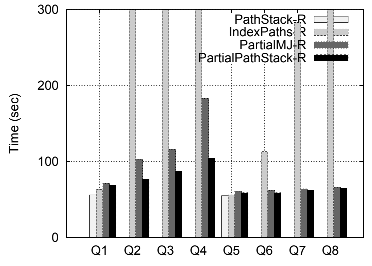
[19] Y. Li, C. Yu, and H. V. Jagadish. Schema-Free XQuery. In *VLDB*, 2004.

[20] Z. Liu and Y. Chen. Identifying meaningful return information for XML keyword search. In *SIGMOD*, 2007.

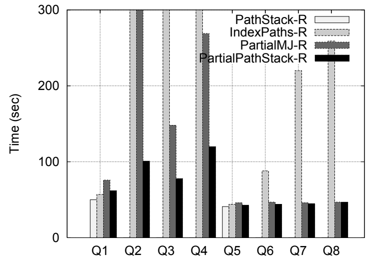
[21] J. Lu, T. Chen, and T. W. Ling. Efficient processing of XML twig patterns with parent child edges: a look-ahead approach. In *CIKM*, 2004.



(a) Execution time (Treebank)



(b) Execution time ($SD1$)

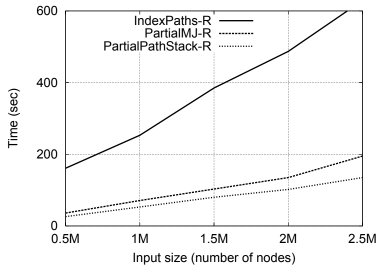


(c) Execution time ($SD2$)

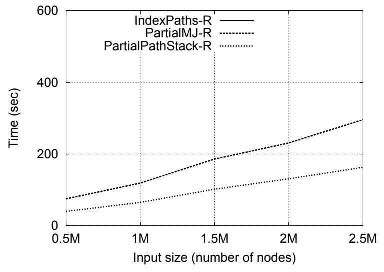
Query	**Treebank**	$SD1$	$SD2$	**Query**	**Treebank**	$SD1$	$SD2$
Q_1	94859	99576	884899	Q_5	1136	3527	39392
Q_2	94859	439725	2991754	Q_6	1136	3715	51058
Q_3	678011	855090	1542299	Q_7	1239	5510	45944
Q_4	1565355	1742980	3779766	Q_8	1258	6081	51583

(d) Number of results

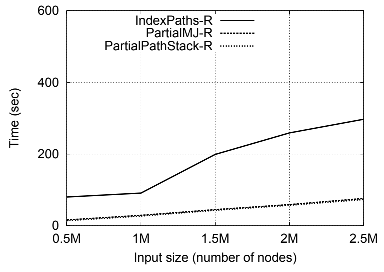
Figure 10: Evaluation of various types of queries for fixed datasets.



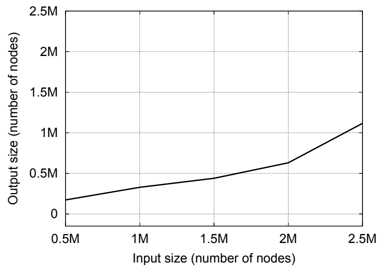
(a) Q_2 execution time



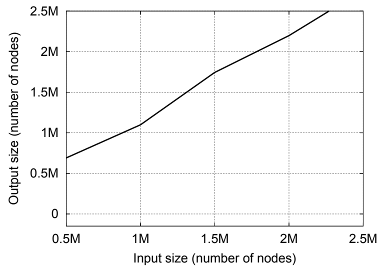
(b) Q_4 execution time



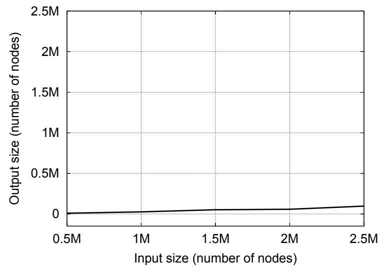
(c) Q_8 execution time



(d) Q_2 results



(e) Q_4 results



(f) Q_8 results

Figure 11: Evaluation of queries varying the input size.

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

[23] D. Olteanu. Forward node-selecting queries over trees. *ACM Trans. Database Syst.*, 2007.

[24] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In *EDBT Worshops*, 2002.

[25] F. Peng and S. S. Chawathe. XPath queries on streaming data. In *SIGMOD*, 2003.

[26] S. Souldatos, X. Wu, D. Theodoratos, T. Dalamagas, and T. Sell. Evaluation of partial path queries on XML data. In *CIKM*, 2007.

[27] D. Theodoratos, T. Dalamagas, A. Koufopoulos, and N. Gehani. Semantic querying of tree-structured data sources using partially specified tree patterns. In *CIKM*, 2005.

[28] D. Theodoratos, S. Souldatos, T. Dalamagas, P. Placek, and T. Sellis. Heuristic containment check of partial tree-pattern queries in the presence of index graphs. In *CIKM*, 2006.

[29] D. Theodoratos and X. Wu. Assigning semantics to partial tree-pattern queries. *Data Knowl. Eng.*, 2007.

[30] Y. Wu, J. M. Patel, and H. V. Jagadish. Structural join order selection for XML query optimization. In *ICDE*, 2003.

[31] B. Yang, M. Fontoura, E. Shekita, S. Rajagopalan, and K. Beyer. Virtual cursors for XML joins. In *CIKM*, 2004.

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