PathStack¬: A Holistic Path Join Algorithm for Path Query with Not-predicates on XML Data

Enhua Jiao, Tok Wang Ling, Chee-Yong Chan

School of Computing, National University of Singapore
{jiaoenhu,lingtw,chancy}@comp.nus.edu.sg

Abstract. The evaluation of path queries forms the basis of complex XML query processing which has attracted a lot of research attention. However, none of these works have examined the processing of more complex queries that contain not-predicates. In this paper, we present the first study on evaluating path queries with not-predicates. We propose an efficient holistic path join algorithm, PathStack¬, which has the following advantages: (1) it requires only one scan of the relevant data to evaluate path queries with not-predicates; (2) it does not generate any intermediate results; and (3) its memory space requirement is bounded by the longest path in the input XML document. We also present an improved variant of PathStack¬ that further minimizes unnecessary computations.

1 Introduction

Finding all root-to-leaf paths in tree-structured XML documents that satisfy certain selection predicates is the basis of complex XML query processing. Such selection predicates are called path queries (i.e., twig queries without branches), and there has been a lot of research on the efficient evaluation of path queries (as well as the more general twig queries) [1, 4, 7, 9, 10, 11]. However, none of these works have considered the processing of more general queries that involved *not-predicates*, which are very common and useful in many applications.

As an example of a path query with a not-predicate, consider the XPath query: //supplier[not(./part/color='red')], which finds suppliers who do not supply any red color parts. A naïve approach to evaluate such path queries is to decompose it into multiple simple path queries (without not-predicates) and evaluate each of the decomposed path queries individually using an existing approach (e.g., PathStack [1]); the final result is then derived by combining the individual results. Thus, the example query can be computed by the set difference of two simple path queries: p1 – p2, where p1 = //supplier and p2 = //supplier[./part/color='red']. Clearly, this approach can be extended to process complex path queries with nested not-predicates by applying the decomposition recursively. However, such a naïve approach is obviously inefficient as it not only incurs high I/O cost for the repetitive scans of the data and the generation of intermediate results, but also incurs computational overhead to combine the intermediate results to derive the final result.

In this paper, we study the problem of evaluating path queries with not-predicates and make the following contributions:

1. We define both the representation of path queries with not-predicates as well as the semantics of matching such queries.
2. We develop two novel algorithms, PathStack$^{\neg}$ and imp-PathStack$^{\neg}$, to efficiently evaluate path queries with not-predicates. Our approach is a generalization of the PathStack algorithm [1], which is based on using a collection of stacks to store partial/complete matching answers.
3. We demonstrate the effectiveness of the proposed algorithms over a naïve approach with an experimental performance study.

To the best of our knowledge, this is the first paper that addresses this problem.

The rest of the paper is organized as follows. Section 2 defines the representation and semantics of path queries with not-predicates. In Section 3, we present our first algorithm for evaluating path queries with not-predicates called PathStack$^{\neg}$. In Section 4, we present an improved variant of PathStack$^{\neg}$ called imp-PathStack$^{\neg}$ that incorporates two optimizations to reduce unnecessary computations. We present a performance study in Section 5. Section 6 discusses related work. Finally, we conclude our paper in section 7 with some future research plans. Due to space constraint, proofs of correctness and other details are given in [6].

2 Preliminaries

2.1 Data Model

For simplicity and without loss of generality, we model an XML document as a rooted, ordered labeled tree, where each node corresponds to an element and each edge represents a (direct) element-subelement. As an example, Fig.1 (b) shows the tree representation for the simple XML document in Fig.1 (a).

Similar to [1], our work does not impose any specific physical organization on the document nodes, and it suffices that there is some efficient access method that returns a stream of document nodes (sorted in document order) for each distinct document element. We also further assume that each stream of returned nodes can be filtered to support any value predicate matching in the path queries; thus, for simplicity, we ignore value predicates in our path queries.

Finally, our work also assumes an efficient method to determine the structural relationship between a given pair of document nodes (e.g., determine whether one node is an ancestor or a parent of another node). Several positional encoding schemes for document nodes have been proposed for this purpose (e.g., [1]), and our proposed algorithms can work with any of these schemes.

2.2 Representation of Path Queries with Not-predicates

A path query with not-predicates is represented as a labeled path $<n_1, n_2, \ldots, n_m>$, where each node n_i (with level number i) is assigned a label, denoted by label(n_i), that is an element name. Each pair of adjacent nodes n_i and n_{i+1} is connected by an edge, denoted by edge(n_i, n_{i+1}), which is classified into one of the following four types: (1) ancestor/descendant edge, represented as "||"; (2) parent/child edge, represented as "|"; (3) negative ancestor/descendant edge, represented as "||¬"; (4) negative parent/child edge, represented as "|¬". A negative edge corresponds to a not-predicate in XPath expression. Two examples of path queries are shown in Fig.1 (c) and (d), where each node is depicted as n_i:label(n_i).

For convenience, we abbreviate the terms parent/child and ancestor/descendant by P/C and A/D, respectively. Given a node n_i, we use parentEdge(n_i) to denote edge(n_{i-1}, n_i) if $i > 1$, and use childEdge(n_i) to denote edge(n_i, n_{i+1}) if $i < m$.

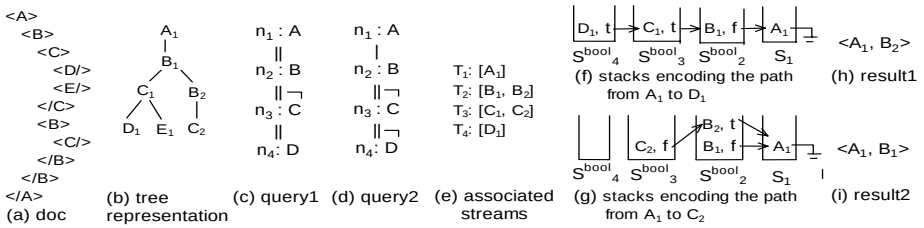

Fig. 1. (a) An XML document consisting of elements only; (b) the tree representation of the document in (a) (integer subscript here is for easy reference to nodes with the same element name); (c) representation of path query: //A//B[not(.//C//D)]; (d) representation of path query: //A/B[not(.//C[not(.//D)])]; (e) the associated streams used in PathStack¬ algorithm; (f) and (g) two examples of stack encoding for root-to-leaf paths in doc tree; (h) result for (c) on (b); (i) result for (d) on (b).

2.3 Matching of Path Queries with Not-predicates

Definition 1. (output node, non-output node, output leaf node, leaf node) A node n_i in a path query is classified as an output node if n_i does not appear below any negative edge; otherwise, it is a non-output node. The output node with the maximum level number is also called the output leaf node. The last node n_m in a path query is also referred to as the leaf node.

For example, $\{n_1, n_2\}$ and $\{n_3, n_4\}$ are the sets of output nodes and non-output nodes in Figs. 1(c) and (d), respectively. Note that n_2 is the output leaf node and n_4 is the leaf node.

We use subquery(n_i, n_j) ($1 \leqslant i \leqslant j \leqslant m$) to refer to a sub-path of a path query that starts from node n_i to node n_j. For example, subquery(n_2, n_4) in Fig.1 (c) refers

to the sub-path consisting of the set of nodes $\{n_2, n_3, n_4\}$ and the two edges edge(n_2, n_3) and edge(n_3, n_4).

Definition 2. (Satisfaction of subquery(n_i, n_j)) Given a subquery(n_i, n_j) and a node e_i in an XML document D, we say that e_i satisfies subquery(n_i, n_j) if (1) the element name of e_i is label(n_i); and (2) exactly one of the following three conditions holds:
(a) i = j; or
(b) edge(n_i, n_{i+1}) is an A/D (respectively, P/C) edge, and there exists a descendant (respectively, child) node e_{i+1} of e_i in D that satisfies subquery(n_{i+1}, n_j) ; or
(c) edge(n_i, n_{i+1}) is a negative A/D (respectively, P/C) edge, and there does not exist a descendant (respectively, child) node e_{i+1} of e_i in D that satisfies subquery(n_{i+1}, n_j).

We say that e_i fails subquery(n_i, n_j) if e_i does not satisfy subquery(n_i, n_j). For notional convenience, we use the notation e_i to represent a document node that has an element name equal to label(n_i).

Example 1. Consider the XML document and query in Figs.1 (b) and (c), respectively. Observe that (1) D_1 satisfies subquery(n_4, n_4) (condition a); (2) C_1 satisfies subquery(n_3, n_4) because of (1) and D_1 is a descendant of C_1 (condition b); and (3) B_1 fails subquery(n_2, n_4) because of (2) and C_1 is a descendant of B_1 (condition c).

Definition 3. (Matching of Path Queries with Not-predicates) Given an XML document D and a path query $<n_1, n_2, \ldots, n_m>$ with n_k as the output leaf node, a tuple $<e_1, \ldots, e_k>$ is defined to be a matching answer for the query iff (1) for each adjacent pair of nodes e_i and e_{i+1} in the tuple, e_{i+1} is a child (respectively, descendant) node of e_i in D if edge(n_i, n_{i+1}) is a P/C (respectively, A/D) edge; and (2) e_k satisfies subquery(n_k, n_m). We refer to a prefix of a matching answer $<e_1, \ldots, e_k>$ as a partial matching answer.

Example 2. Consider the document in Fig.1 (b). For query1 in Fig.1 (c), $<A_1, B_1>$ is not a matching answer for it since C_1 satisfies subquery(n_3, n_4) and therefore B_1 fails subquery(n_2, n_4); hence $<A_1, B_1>$ fails condition (2) of Definition 3. However, $<A_1, B_2>$ is a matching answer for it because there does not exist a C_i node in Fig.1 (b) which is a descendant of B_2 and satisfies subquery(n_3, n_4); therefore B_2 satisfies subquery(n_2, n_4). Clearly, $<A_1, B_2>$ satisfies condition (2) of Definition 3. Similarly, for query2 in Fig.1 (d), $<A_1, B_1>$ is a matching answer for it since B_1 satisfies subquery(n_2, n_4) and $<A_1, B_1>$ satisfies condition (2) in Definition 3. However, $<A_1, B_2>$ is not a matching answer for query2 because B_2 fails subquery(n_2, n_4).

3 PathStack¬ Algorithm

In this section, we describe our first algorithm, called PathStack¬, for evaluating path queries that contain not-predicates. As the name implies, our approach is based on the stack encoding technique of the PathStack approach [1] for evaluating path queries without not-predicates.

3.1 Notations and Data Structures

Each query node n_i is associated with a data stream T_i, where each T_i contains all document nodes for element label(n_i) sorted in document order. Each stream T_i maintains a pointer that points to the next node in T_i to be returned. The following operations are supported for each stream: (1) eof(T_i) tests if the end of the stream T_i is reached; (2) advance(T_i) advances the pointer of T_i; and (3) next(T_i) returns the node pointed to by the pointer of T_i.

Each query node n_i is also associated with a stack S_i which is either a regular stack or a boolean stack. In a *regular stack*, each item in S_i consists of a pair <e_i, pointer to an item in S_{i-1}>, where e_i is a document node with the element name of e_i equal to label(n_i). In a *boolean stack,* each item in S_i consists of a triple <e_i, pointer to an item in S_{i-1}, satisfy>, where satisfy is a boolean variable indicating whether e_i satisfies subquery(n_i, n_m) w.r.t. all the nodes in the data streams that have been visited so far during the evaluation. Note that the pointer to an item in S_{i-1} is null iff i=1. The stack S_i associated with n_i is a boolean stack if n_i is a non-output node or the output leaf node; otherwise, S_i is a regular stack. If S_i is a boolean stack, we can also denote it explicitly by S^{bool}_i. Note that only regular stacks are used in the PathStack algorithm [1].

The following operations are defined for stacks: (1) empty(S_i) tests if S_i is empty; (2) pop(S_i)/ top(S_i) pops/returns the top item in S_i; and (3) push(S_i, item) pushes item into S_i. For an input XML document D, the stacks are maintained such that they satisfy the following three stack properties:

1. At every point during the evaluation, the nodes stored in the set of stacks must lie on a root-to-leaf path in the input XML document D.
2. If e_i and e'_i are two nodes in S_i, then e_i appears below e'_i in S_i iff e_i is an ancestor of e'_i in D.
3. Let m_i=<e_i, $pointer_i$> and m_{i-1}=<e_{i-1}, $pointer_{i-1}$> be two items in stacks S_i and S_{i-1}, respectively. If $pointer_i$=m_{i-1} (i.e., e_i is linked to e_{i-1}), then e_{i-1} must be an ancestor of e_i in D such that there is no other node (with the same element name as e_{i-1}) in D that lies along the path from e_{i-1} to e_i in D.

3.2 Algorithm

The main algorithm of PathStack$^{\neg}$ (shown in Fig.2) evaluates an input path query q by iteratively accessing the data nodes from the streams in sorted document order, and extending partial matching answers stored in the stacks. Each iteration consists of three main parts. The first part (step 2) calls the function getMinSource to determine the next node from the data streams to be processed in document order. Before using the selected next node to extend existing partial matching answers, the algorithm first needs to pop off nodes from the stacks that will not form a partial matching with the next node (i.e., preserve stack property1). This "stack cleaning" operation is done by the second part (steps 3 to 9). Each time an item <e_i, $pointer_i$, *satisfy*> is popped from a boolean stack S^{bool}_i, the algorithm will output all the matching answers that end with e_i (by calling showSolutions) if n_i is the output leaf node and *satisfy* is true. Other-

wise, if n_i is a non-output node, then S_{i-1} must necessarily be a boolean stack, and updateSatisfy is called to update the *satisfy* values of the appropriate nodes in S_{i-1}. Finally, the third part (step 11) calls the function moveStreamToStack to extend the partial answer currently stored in stacks by pushing the next node into the stack S_{min}.

```
Algorithm PathStack¬(q)
01 while (¬end(q))
02   n_min = getMinSource(q)  // find the next node
03   for query node n_i of q in descending i order // clean stack
04     while((¬empty(S_i)∧(top(S_i) is not an ancestor of next(T_min))
05       e_i = pop(S_i)
06       if (n_i is the output leaf node ∧ e_i.satisfy=true)
07         showSolutions(e_i)  // output solution
08       else if (n_i is a non-output node)
09         updateSatisfy(n_i, e_i)
10   //push the next node
11   moveStreamToStack(n_min, T_min, S_min, pointer to top(S_min-1))
12 repeat steps 03 to 09 for the remaining nodes in the stacks

Function getMinSource(q)
  Return query node n_i of q such that next(T_i) has the minimal document
order among all unvisited nodes.

Function end(q)
  Return ∀ n_i in q ⇒ eof(T_i) is true.
```

Fig. 2. PathStack¬ Main Algorithm

```
Procedure moveStreamToStack(n_i, T_i, S_i, pointer)
01 if S_i is a regular stack // regular stack, no Boolean value
02   push(S_i, <next(T_i), pointer>)
03 else if n_i is the leaf node
04   push(S_i, <next(T_i), pointer, true>)
05 else if childEdge(n_i)is negative
06   push(S_i, <next(T_i), pointer, true>)
07 else if childEdge(n_i)is positive
08   push(S_i, <next(T_i), pointer, false>)
09 advance(T_i)

Procedure updateSatisfy(n_i, e_i)
01 if e_i.satisfy = true
02   e_i-1 = e_i.pointer
03   if parentEdge(n_i) is a negative edge
04      newvalue = false
05   else
06      newvalue = true
07   if parentEdge(n_i) is an A/D edge
08      for all e'_i-1 in S^pool_i-1 that are below e_i-1 (inclusive of e_i-1)
09          e'_i-1.satisfy = newvalue
10   else // parentEdge(n_i) is an P/C edge
11      if e_i-1 is a parent of e_i
12          e_i-1.satisfy = newvalue
```

Fig. 3. Procedures moveStreamToStack and updateSatisfy

The details of the procedures moveStreamToStack and updateSatisfy are shown in Fig.3. In moveStreamToStack, if the input stack S_i is a boolean stack, then the *satisfy* value of the data node e_i to be pushed into S_i is initialized as follows. If n_i is the leaf node in the query (step 3), then e_i trivially satisfies subquery(n_i, n_m) and

satisfy is set to true. Otherwise, *satisfy* is set to false (respectively, true) if childEdge(n_i) is a positive (respectively, negative) edge since e_i satisfies (respectively, fails) subquery(n_i, n_m) w.r.t. all the nodes that have been accessed so far.

Procedure updateSatisfy maintains the satisfy values of stack entries such that when a data node e_i is eventually popped from its stack S_i, its satisfy value is true iff e_i satisfies subquery(n_i, n_m), i.e, w.r.t. the whole input XML document. The correctness of updateSatisfy is based on the property that once an initialized *satisfy* value is complemented by an update, its value will not be complemented back to the initialized value again.

The details of procedure showSolutions can be found in [1].

Example 3. This example illustrates the evaluation of query1 in Fig.1 (c) on the XML document in Fig.1 (b) using algorithm PathStack$^{\neg}$.

(1) The nodes A_1, B_1, C_1, and D_1 are accessed and pushed into their corresponding stacks; the resultant stack encoding is shown in Fig.1 (f).
(2) B_2 is the next node to be accessed (E_1 is not accessed as it is irrelevant to the query), and nodes C_1 and D_1 need to be first popped off from their stacks to preserve the stack properties. When node D_1 is popped, it is detected to satisfy subquery(n_4, n_4), and therefore C_1.satisfy is updated to *true*. When C_1 is popped, it is determined to satisfy subquery(n_3, n_4). Consequently, B_1.satisfy is updated to *false*.
(3) B_2 is accessed and pushed into S^{bool}_2
(4) C_2 is accessed and pushed into S^{bool}_3; the resultant stack encoding is shown in Fig.1 (g).
(5) Since all the relevant data nodes have been accessed, the algorithm now pops off the remaining nodes in the stacks in the order of C_2, B_2, B_1, and A_1. When C_2 is popped, it is detected to fail subquery(n_3, n_4) and so no update takes place. When B_2 is popped, it is detected to satisfy subquery(n_2, n_4). Since B_2 is a leaf output node, the matching answer <A_1, B_2> is generated. When B_1 is popped, it is detected to fail subquery(n_2, n_4) and so no matching answer is produced. Finally, A_1 is popped without triggering any operations.
(6) Since all the stacks are empty, the algorithm terminates with exactly one matching answer <A_1, B_2>.

3.3 Performance Analysis

In this section, we present an analysis of the time and space complexity of algorithm PathStack$^{\neg}$. Let $Size_i$ denote the total number of nodes in the accessed data streams, $Size_\ell$ denote the length of the longest path in the input XML document, and $Size_o$ denote the size of the matching answers.

Since the number of iterations in the outer while loop (steps 1 to 11) is bounded by the number of nodes in the input streams, and both the inner for loop (steps 3 to 9) and step 12 are bounded by the longest path in the input XML document, the CPU complexity of PathStack$^{\neg}$ is given by O($Size_i$ * $Size_\ell$ + $Size_\ell$). The I/O complexity is O($Size_i$ + $Size_o$) since the input data streams are scanned once only and the only out-

puts are the matching answers. The space complexity is given by O($Size_t$) since at any point during the evaluation, the data nodes that are stored in the stacks must lie on a root-to-leaf path in the input XML document.

4 Improved PathStack¬ Algorithm

In this section, we present an improved variant of PathStack¬, denoted by imp-PathStack¬, that is based on two optimizations to reduce unnecessary computations. Due to space constraint, the details of the optimizations are omitted in this paper but can be found in [6].

4.1 Reducing the number of boolean stacks

One key extension introduced by our PathStack¬ algorithm to handle not-predicates is the use of boolean stacks for output leaf and non-output query nodes. Boolean stacks are, however, more costly to maintain than regular stacks due to the additional *satisfy* variable in each stack entry. In this section, we present an optimization to minimize the number of boolean stacks used in the algorithm.

Our optimization is based on the observation that boolean stacks are actually only necessary for query nodes that have negative child edges. To understand this optimization, note that a non-output node n_i can be classified into one of three cases: (1) n_i is also the leaf node; or (2) n_i has a positive child edge; or (3) n_i has a negative child edge. For case (1), since each data node e_i in S_i trivially satisfies subquery(n_i, n_m), e_i .satisfy is always true and therefore S_i can be simplified to a regular stack (i.e., S_i can be viewed as a virtual boolean stack). For case (2), the satisfy value of each node in S_i can effectively be determined from the nodes in S_j, where n_j is the nearest descendant query node of n_i that is associated with a (real or virtual) boolean stack. Details of this are given in [6]. Thus, S_i again can be simplified to a regular stack. Consequently, only non-output nodes belonging to case (3) need to be associated with boolean stacks.

4.2 Nodes Skipping

Our second optimization aims to exploit the semantics of not-predicates to minimize the pushing of "useless" data nodes into stacks that do not affect the input query's result. In the following, we explain and illustrate the intuition for this optimization; more details are given in [6].

Consider a stack S^{bool}_i that corresponds to a query node n_i with a negative child edge. Suppose e_i, the topmost node in S^{bool}_i, has a *false* value for *satisfy*. Then there are two cases to consider for the optimization depending on whether childEdge(n_i) is an A/D or P/C edge.

Case 1: If childEdge(n_i) is an A/D edge, then it follows that every data node below e_i in S^{bool}_i also has a *false* value for *satisfy*. Therefore, for each $j > i$, the nodes in T_j that precede next(T_i) in document order can be skipped as they will not contribute to any matching answers. For example, consider the query query1 on document doc1 in Fig.4. Note that after the path of nodes from A_1 to C_1 have been accessed, the satisfy values for both A_1 and A_2 are determined to be *false*. Thus, the stream of nodes $\{B_2,\ldots,B_5\}$ and $\{C_2,\ldots,C_4\}$ can be skipped as they will not affect the *satisfy* value of A_3.

Case 2: If childEdge(n_i) is a P/C edge, then let e'_i be the lowest node in S^{bool}_i with a *false* value for *satisfy*. It follows that for each $j > i$, the data nodes in T_j that are descendants of e'_i and precede next(T_i) in document order will not contribute to any matching answers and can therefore be skipped. For example, consider the query query2 on document doc2 in Fig.4. Note that after the path of nodes from A_1 to C_1 have been accessed, the *satisfy* values for both A_1 and A_2 are determined to be *false*, and the stream of nodes $\{B_3, B_4\}$ and $\{C_2,\ldots,C_4\}$ can be skipped. As another example, consider query query2 on document doc1 in Fig.4. After the path of nodes from A_1 to C_1 have been accessed, the *satisfy* value for A_2 is determined to be *false*, the stream of nodes $\{B_2, B_3\}$ and $\{C_2\}$ can be skipped. Note that B_4 and C_3 can not be skipped in this case as they will affect A_1's *satisfy* value which is yet to be determined.

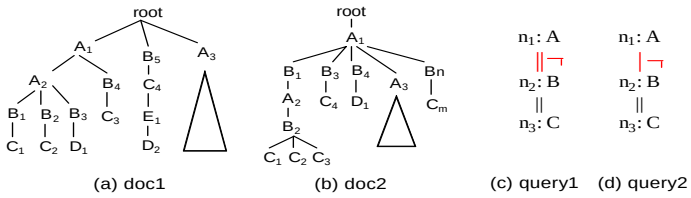

Fig. 4. XML documents and path queries

The node skipping optimization becomes even more effective when combined with the boolean stack reduction optimization since it enables the nodes' satisfy values to be updated earlier and a more aggressive node skipping. For example, consider the query query1 on document doc2 in Fig.4. When the boolean stack optimization is used, there is only one boolean stack S^{bool}_2, and the *satisfy* values of A_1 and A_2 are both determined to be *false* once the path of nodes from A_1 to C_1 have been accessed. In contrast, without the first optimization, all the stacks are boolean, and the *satisfy* values of A_1 and A_2 are determined to be *false* only after B_2 is popped off from its stack when B_3 is accessed; consequently, the nodes C_2 and C_3 can not be skipped.

5 Experimental Evaluation

This section presents experimental results to compare the performance of our proposed algorithms, PathStack$^{\neg}$ and imp-PathStack$^{\neg}$, as well as the decomposition-based naïve approach described in Section 1 (referred to as Naïve).

We used the synthetic data set Treebank.xml [14] with about half a million of nodes, an average path length of 8 levels, and a maximum path length of 35 levels. We generated three sets of path queries (denoted by Q1, Q2, and Q3), where each query in Qi contains exactly i number of not-predicates and has 7 levels. About 30% of the data nodes are accessed for each query, and the matching answers are formed from about 0.4% of the data nodes. For each query and approach, we measured both the total execution time as well as the disk I/O (in terms of the total number of data nodes that are read/written to disk). Our experiments were conducted on a 750MHz Ultra Sparc III machine with 512MB of main memory.

5.1 Naïve vs. PathStack¬

Fig.5 compares the execution time of Naïve and PathStack¬, and the results show that PathStack¬ is much more efficient than Naïve. In particular, observe that while the execution time of Naïve increases almost linearly as the number of not-predicates increases, the execution time of PathStack¬ remains constant. This can be explained by the results in Fig.6 (which compares their I/O performance) and Fig.7 (which gives the detailed breakdown).

Fig.6 shows that the I/O cost of PathStack¬ is independent of the number of not-predicates since each data stream is scanned exactly once (without any intermediate results generated), and the final matching answers written to disk have about the same size. On the other hand, Fig.7 reveals that as the number of not-predicates increases, Naïve incurs more disk I/O as it needs to access the data streams multiple times and output intermediate results to disk.

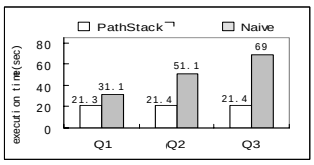

Fig. 5. Execution time comparison between PathStack¬ and Naïve.

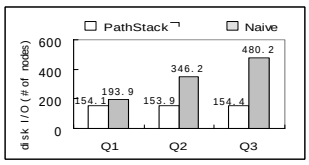

Fig. 6. Disk I/O comparison between Path-Stack¬ and Naïve.

		Streams		Intermediate Result		Final Results		Total
		# of nodes	% of total	# of nodes	% of total	# of nodes	% of total	# of nodes
Q1	PathStack¬	152.1 k	98.7%	0 k	0%	2 k	1.3%	154.1 k
	Naive	185.7 k	95.8%	6.2 k	3.2%	2 k	1%	193.9 k
Q2	PathStack¬	152.0 k	98.8%	0 k	0%	1.9 k	1.2%	153.9 k
	Naive	337.1 k	97.4%	7.2 k	2.1%	1.9 k	0.5%	346.2 k
Q3	PathStack¬	152.3 k	98.6%	0 k	0%	2.1 k	1.4%	154.4 k
	Naive	466.8 k	97.2%	11.3 k	2.4%	2.1 k	0.4%	480.2 k

Fig. 7. Breakdowns of disk I/O in PathStack¬ and Naïve.

5.2 PathStack¬ vs. imp-PathStack¬

Fig.8 compares the execution time of PathStack¬ and imp-PathStack¬; the amount of time spent only on scanning the accessed data streams is also shown (labeled as

"sequential scan") for comparison. Our results show that imp-PathStack¬ is only slightly faster than PathStack¬, with about 90% of the total execution time being dominated by the scanning of the data streams. Note that our implementation of imp-PathStack¬ did not utilize any indexes for accessing the data streams. We expect that if the data streams were indexed, the performance improvement of imp-PathStack¬ over PathStack¬ (due to the additional reduction of I/O cost in node skipping) would become more significant.

Fig.9 compares the number of skipped nodes for various queries using imp-PathStack¬. Our results did not reveal any interesting relationship between the number of not-predicates and the percentage of skipped nodes (which is between 2.4% and 18.5%); we expect this percentage to be higher for an XML document that has a higher fan-out (note that the fan-out of treebank.xml is only around 2-3). More analysis can be found in [6].

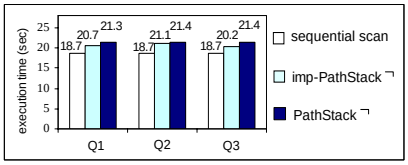

Fig.8. Execution time comparison between Sequential Scan, imp-PathStack¬ and PathStack¬.

	Stream Size (# of nodes)	Nodes Skipped (# of nodes)	% of skipping
Q1	152.1 k	10.2 k	6.7 %
Q2	152.0 k	3.6 k	2.4 %
Q3	152.3 k	28.1 k	18.5 %

Fig.9. Percentage (%) of nodes skipped for each query set in imp-PathStack¬.

6 Related Work

XML query processing and optimization for XML databases have attracted a lot of research interests. Particularly, path query matching has been identified as a core operation in querying XML data. While there is a lot of work on path and twig query matching, none of these works addressed the evaluation of queries with not-predicates. Below, we review the existing work on path/twig query evaluation, all of which do not address not-predicates.

Earlier works [3, 5, 9, 10, 12, 13, 14] have focused on a decomposition-based approach in which a path query is decomposed into a set of binary (parent-child and ancestor-descendant) relationships between pairs of query nodes. The query is then matched by (1) matching each of the binary structural relationships against the XML data, and (2) "stitching" together these basic matches. The major problem with the decomposition-based approach is that the intermediate results can get very large even when the inputs and final results are small.

The work in [1, 2] are more closely related to ours. The algorithms PathStack and PathMPMJ were proposed to evaluate path queries without not-predicates. These algorithms process a path query in a holistic manner, which do not generate large intermediate results and also avoid costly binary structural joins. PathStack has been shown to more efficient than PathMPMJ as it does not require repetitive data scans.

7 Conclusions and Future Work

In this paper, we have proposed two novel algorithms PathStack¬ and imp-PathStack¬ (which is an improved variant of PathStack¬ to further minimize unnecessary computation) for the efficient processing of path queries with not-predicates. We have defined the representation and matching of path queries with not-predicates, and proposed the simple but effective idea of using boolean stacks to support efficient query evaluation. Our proposed algorithms require only a single scan of each data stream associated with the input query without generating any intermediate results. To the best of our knowledge, this is the first work that addresses the evaluation of path queries with not-predicates.

While our proposed algorithms can be easily extended to handle twig queries with at most one path containing not-predicates, we are currently extending our work to process more general twig queries that have not-predicates in multiple branches.

References

1. N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML pattern matching. In *Proc. of the SIGMOD, 2002*.
2. N. Bruno, N. Koudas, D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. *Technical Report. Columbia University. March 2002.*
3. D. Florescu and D. Kossman. Storing and querying XML data using an RDMBS. IEEE Data Engineering Bulletin, 22(3): 27-34, 1999.
4. H. Jiang, H. Lu, W. Wang, Efficient Processing of XML Twig Queries with OR-Predicates, In *Proc. of the SIGMOD 2004*.
5. H. Jiang, W. Wang, H. Lu, and J. X. Yu. Holistic twig joins on indexed XML documents. In *Proc. of the VLDB, pages 273-284, 2003*.
6. E. Jiao, Efficient processing of XML path queries with not-predicates, M.Sc. Thesis, National University of Singapore, 2004.
7. Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. *In Proc. of the VLDB, pages 361-370, 2001.*
8. R. Riebig and G.Moerkotte. Evaluating queries on structure with access support relations. In *Proc. of the WebDB'00, 2000.*
9. J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. *In Proc. of VLDB, 1999*.
10. D. Srivastava, S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. *In Proc. of the ICDE, pages 141-152, 2002.*
11. 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 the SIGMOD, pages 110-121, 2003*.
12. Y. Wu, J. M. Patel, and H. V. Jagadish. Structural join order selection for XML query optimization. *In Proc. of the ICDE, pages 443-454, 2003.*
13. C. Zhang, J. Naughton, D. Dewitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. *In Proc. of the SIGMOD, 2001.*
14. Treebank.xml: http://www.cis.upenn.edu/~treebank/.