QuickStack: A Fast Algorithm for XML Query Matching

Iyad Batal
Department of Computer Science
University of Pittsburgh
iyad@cs.pitt.edu

Alexandros Labrinidis
Department of Computer Science
University of Pittsburgh
labrinid@cs.pitt.edu

June 6, 2008

Abstract

With the increasing popularity of XML for data representation and exchange, much research has been done for providing an efficient way to evaluate twig patterns in an XML database. As a result, many holistic join algorithms have been developed, most of which are derivatives of the well-known *TwigStack* algorithm. However, these algorithms still apply a two phase processing scheme: first identify all root-to-leaf path solutions and then join these intermediate solutions to form the twig results. In this paper, we first propose a novel algorithm, *QuickStack*, for matching single path queries. The proposed algorithm extensively optimizes the *PathStack* algorithm by effectively skipping the ancestor and descendant elements that do not participate in the results. Secondly, we generalize *QuickStack* to answer twig pattern queries. Unlike the previous algorithms, *QuickStack* joins the intermediate path solutions incrementally while evaluating the root-to-leaf paths of the twig query. Our extensive performance study, over a range of synthetic and real world datasets, shows that *QuickStack* provides a drastic improvement gain over *TwigStack* for a wide variety of both single path and twig queries. Finally, we compare our algorithm with *YFilter* for answering multiple XML queries.

1 Introduction

XML is emerging as the de facto standard for data representation and exchange over the Internet. Hence, indexing and querying XML documents efficiently has been among the major research issues in the database community. XML documents are semi-structured by nature and can be modeled as trees. To retrieve such tree-shaped data, several XML query languages have been proposed in the literature, for example XPath [3] and XQuery [4]. XML queries are typically formed as a twig (small tree) pattern and predicates may be additionally imposed on the contents or attribute values of the tree nodes. The edges of the twig are either parent-child or ancestor-descendant relationships. Finding all the occurrences of a twig pattern in an XML document with all associated predicates satisfied is a core operation in XML query processing, and is the focus of our work.

Early work on XML twig pattern processing usually involved decomposing the twig pattern into a set of binary structural relationships, matching these relationships, and stitching the matches to form the final result. In particular, Al Khalifa et al. [1] proposed two such binary structural join algorithms: *Tree-Merge* and *Stack-Tree*. The stack representation of *Stack-Tree* has been used in most follow-up works. However, the main drawback of binary structural join algorithms is that they may generate a large number of intermediate results that do not appear in the final results.

To address this problem, Bruno et al. [6] proposed two holistic join algorithms, namely *PathStack* and *TwigStack*. These algorithms use a chain of linked stacks to compactly represent partial results of individual root-to-leaf paths in the twig pattern. Both algorithms operate in two phases: first, compute all the relevant root-to-leaf path solutions, then join-merge these partial solutions to form the answers for the entire twig.

Currently, holistic join algorithms [19, 16, 9, 20, 8] employ the same two phase processing technique originally proposed in [6]. In this work, we propose another approach: *doing the solution-merging incrementally, while evaluating the other paths of a twig*. This approach allows the algorithm to quickly skip processing a lot of the elements that are not part of the solutions for the previously evaluated paths, which speeds up the algorithm.

Another group of XML query processing techniques is the *navigation-based* approach, which computes results by analyzing the input document one tag at a time, and is commonly used in information dissemination systems. *YFilter* [13], the current state-of-the-art for navigation-based algorithms, combines all path expressions into a single non-deterministic finite automaton (NFA), which enables highly efficient, shared processing for a

large number of XPath queries. Although *YFilter* is designed to optimize across XML queries, it has to examine all the elements of the document. In some cases, this could be more expensive than applying a holistic algorithm able to avoid processing large portions of the document that are not going to be in any matches. To illustrate this tradeoff, we compared our algorithm with *YFilter* for the case of multiple queries.

Contributions This work makes the following contributions:

- We develop *QuickStack*, an efficient algorithm to match single path XML queries. *QuickStack* is based on the *PathStack* algorithm, but aggressively discards ancestors and descendants that do not participate in any matches.
- We modify the stop condition used in *TwigStack* (and many of its extensions) in order to make the algorithm more efficient, without missing any results.
- We develop the *TQS* algorithm, an extension to *QuickStack*, to efficiently answer complex twig queries.
- We propose two general techniques that can be applied while building the element streams in order to reduce the processing time of the algorithm.
- We present experimental results on a range of real and synthetic data which shows that *QuickStack*/*TQS* significantly outperforms *TwigStack* for both twig queries and single path queries.
- We also experimentally compare *QuickStack* against *YFilter*, when answering multiple queries. Our results establish that *QuickStack* is more efficient than *YFilter* when the number of queries is small and the XML document is large.

The remainder of the paper is organized as follows. Section 2 is dedicated to background materials and the *PathStack* and *TwigStack* algorithms. Section 3 describes *QuickStack*, our proposed algorithm for matching single path queries. In Section 4, we generalize *QuickStack* to answer twig queries efficiently. In Section 5, we introduce our framework for answering multiple XML queries. Section 6 describes how to reduce the number of elements that participate in the algorithm. In Section 7, we report the experimental results. Lastly, we conclude in Section 8.

2 Background and related work

2.1 The region encoding scheme

Most existing XML query processing algorithms rely on a positional representation of the XML elements (see [12, 23]), where each element is assigned a triplet of numbers $(start, end, depth)$, based on its position in the data tree. This scheme is called *region encoding*. An element x is an ancestor of element y if the region of y is fully contained within the region of x. Since the regions cannot partially overlap (according to the strictly nested property of XML), we say that x is an ancestor of y iff $x.start < y.start < x.end$. Element x is a parent of element y if x is an ancestor of y and $x.depth = y.depth - 1$. The important property of this numbering scheme is that it allows determining the structural relationship between two elements in the XML document in constant time. Figure 1 shows a fictitious XML document with the region encoding for each element (we explain the use of the subscripts later).

Note that only the region encoding is needed to determine that element $c_2(12, 13, 4)$ is a descendant of $a_2(10, 15, 2)$; No examination of the tree is necessary.

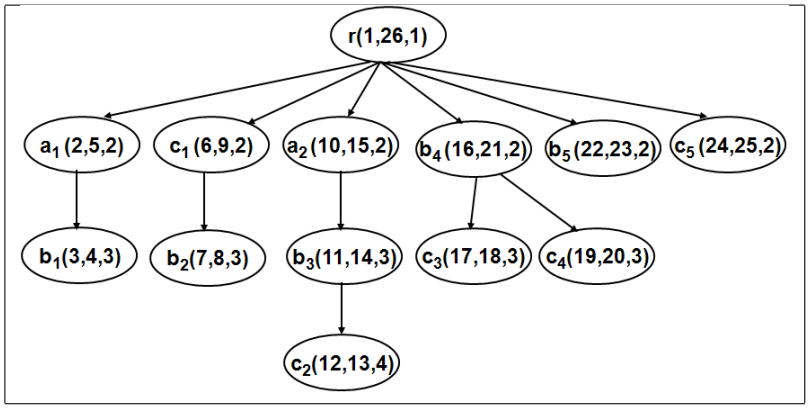

Figure 1: A sample XML document

2.2 Preliminaries

Since our main algorithm deals with single path queries, we represent each query as a chain or a unary tree. Let q denote a node in the query. The self-explanatory functions *isLeaf(q)* and *isRoot(q)* examine whether q is a leaf or a root node. The functions *child(q)* and *parent(q)* return the child and the parent of q, respectively. The function *subTree(q)* returns q and all

its descendants in the query tree. *q.level* gives the depth of the node in the query tree. In the rest of the paper, "*node*" refers to a node in the query tree, whereas "*element*" refers to an element in the XML document.

Each node q in the query is associated with a stream of the XML elements that match q from the document. So before the query can be evaluated by the algorithm, the document should be scanned to build the element streams of the query nodes. the A cursor T_q points to the current element of q's stream. *advance(T_q)* forwards T_q to the next element in the stream, while *eos(T_q)* tests whether T_q has reached the end of q's stream. The elements in the stream are encoded using the positional representation, *(start, end, depth)*, and sorted by the *start* attribute. The positional representation of the element pointed to by T_q can be accessed using $T_q.start$, $T_q.end$ and $T_q.depth$.

In addition to the stream, each node q is also associated with a stack S_q. Initially, all the stacks are empty and all the cursors point to the first element of the corresponding stream. During the evaluation of the query, the cursors advance sequentially and each stack S_q may cache some elements from the stream that appear before the cursor T_q.

The elements in the stacks should always be strictly nested from bottom to top, i.e, each element is a descendant of the element below it. In addition, the stack element e in S_q is also associated with a pointer to its lowest ancestor in $S_{parent(q)}$. Using this pointer, we can easily access all of e's ancestors in $S_{parent(q)}$. These cached elements in the stacks represent partial results that are potentially extended to full results as the algorithm goes on. The operations for the stacks are the usual *push* and *pop* and *empty* operations.

2.3 PathStack algorithm

As *QuickStack* is partially inspired by the *PathStack* algorithm, we briefly describe the way *PathStack* works, and illustrate using an example.

PathStack processes the stream elements in order of their *start* attribute. In other words, the elements are examined in order of their appearances in the pre-order traversal of the document tree. Therefore, the query path pattern is matched from the query root down to the query leaf. When e is the current element, *PathStack* removes from the stacks all the elements that end before $e.start$ since they can no longer be part of any match. Whenever an element of the leaf node is pushed into the stack , the algorithm outputs the current query path answers encoded in the stacks.

Example 1: Consider the path query //a//b//c on the XML document of

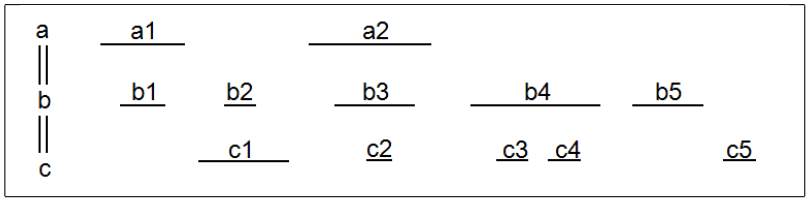

Figure 2: Query a//b//c and the element streams from the XML document in Figure 1

Figure 1. The streams of the XML elements associated with each query node are visualized in Figure 2. Each line segment represents the $(start, end)$ interval of the element in the document. A subscript is added to each element in the order of their *start* values for easy reference. It can be easily seen from Figure 2 that $(a2, b3, c2)$ is the only match of the query in this document.

***PathStack* on Example 1**

Initially, the three cursors are at (a_1, b_1, c_1). Element a_1 is pushed first into stack S_a because it is the element with the smallest *start* value among the current cursors and T_a advances to a_2. In the next iteration, b_1 is pushed in S_b. Then c_1 is pushed and it pops out elements a_1 and b_1 because $c_1.start > a_1.end$ and $c_1.start > b_1.end$. Since c_1 is a leaf node in the query tree, the algorithm tries to output the solutions encoded in the stacks, but the stacks are empty at this moment. Later on, when *PathStack* pushes c_2 in S_c, it outputs the solution (a_2, b_3, c_2). After that, *PathStack* continues to push and pop all the remaining elements until it reaches c_5. At that time, the algorithm terminates because the stream of the leaf node has ended.

As we see, *PathStack* had to examine all the elements in the streams in order to find the single solution for the query.

2.4 TwigStack algorithm

In this section, we describe the *TwigStack* algorithm and how it evaluates the query in Example 1 (Figure 2).

The *TwigStack* algorithm is used to evaluate twig patterns and operates in two phases. In the first phase, some (but not all) solutions to individual query root-to-leaf paths are computed. In the second phase, these solutions are merge-joined to compute the answers to the twig query. A key difference between *PathStack* and *TwigStack* is that when the twig pattern has only ancestor-descendant edges, *TwigStack* ensures that each root-to-leaf solution is merge-joinable with at least one solution to each of the other root-to-leaf query paths. Thus, none of the intermediate path solutions is superfluous.

TwigStack works by repeatedly calling a routine called *getNext*, which returns the next node for processing. *getNext(q)* only returns a node q' if it has a *descendant extension*, which means that there is a solution for the subquery rooted at q' composed entirely of the current elements of the nodes in that tree[1]. To do this, *getNext* first traverses down to the left-most leaf node of the query tree (by self recursive calls). Then, starting from that node, it tries to find the highest possible query node with a descendant extension.

In the case where the twig pattern contains a parent-child edge between two nodes, the algorithm might produce some extraneous intermediate solutions. However, we observed that ***TwigStack* can still be optimal, in terms of the size of the intermediate solutions, when the document does not have a recursive definition**[2].

Another important observation is that *TwigStack* is not only more efficient than *PathStack* for twig queries, but also for single path queries. The following example illustrates the behavior of *TwigStack* for a single path query.

***TwigStack* on Example1 (Figure 2)**
Initially, the cursors point to (a_1, b_1, c_1). The first call to *getNext* advances T_b to b_2 (since b_1 does not have a c child) and returns the element c_1 (trivially, the leaf node c has a descendant extension at all elements in its stream). The second *getNext* call advances T_b to b_3, T_a to a_2 and returns a_2 as the next element to process. After that, the elements b_3 and c_2 are returned (in this order) and the solution is output from the stacks. Next, the elements b_4, c_3, c_4 and c_5 are all returned by consecutive calls to *getNext* and the algorithm stops after processing c_5.

From this example, we can see that *TwigStack* has skipped some elements that *PathStack* had to process, namely: a_1, b_1, b_2 and b_5. However, it had to examine some extra elements that do not participate in the solution, namely: c_1, b_4, c_3, c_4 and c_5.

We will return to this example again at the end of Section 3 to see how *QuickStack*, our proposed algorithm, evaluates this query.

2.5 Other related algorithms

J. Lu et al. [19] presented the *TwigStackList* algorithm, which generates fewer intermediate results than *TwigStack* when there are parent-child edges

[1] More specifically, *getNext(q)*=q' if the current element e', pointed to by $T_{q'}$, has a descendant e_i, pointed to by T_{q_i}, for all $q_i \in children(q')$.

[2] The document has a recursive definition if an element can have a descendant with the same label.

in the query. [16] proposed the *TSGeneric+* algorithm that can utilize available indices, such as the *XR-tree* (XML Region Tree) index [15], to accelerate the running time of *TwigStack*. But their algorithm imposes the additional overhead of building the index over the element streams. Chen et al. [9] proposed a method to perform holistic twig pattern matching using different data partition strategies. Lu et al. [20] proposed *TJFast* that uses the *extended Dewey* labeling scheme to answer a twig query by accessing the labels of its leaf nodes. Finally, Chen et al. [8] proposed *Twig2Stack*, a bottom-up algorithm based on the hierarchical stack encoding scheme. This algorithm is also capable of processing the more complex *GTP* queries. However, the memory requirement for *Twig2Stack* is higher than *TwigStack* and it may keep the entire document in memory.

3 QuickStack for single path queries

In this section, we describe our proposed algorithm, *QuickStack*, for finding all matches of a single path query against an XML document. The algorithm is based on the *PathStack* algorithm, but adds a lot of features to effectively skip ancestors and descendants that do not participate in any match.

The algorithm is outlined in Figure 3. The argument q is the root of the path query. Lines 3-4 identify the nodes q_{min} and q_{max} that have the minimal and maximal *start* position among the current cursors. Lines 5-6 pop out all the elements that end before $q_{min}.start$ because they cannot contribute to the solutions any more. Lines 7-8 terminate the algorithm if the condition of *QuickEnd* is satisfied. In lines 9-13: we call *skipAncestors* on q_{max}'s parent node if q_{max} is lower in the query tree, i.e. closer to the leaf; otherwise, we call *skipDescendants* on q_{min} if its parent stack is empty. Both *skipAncestors* and *skipDescendants* return true if they skipped any elements from the streams.

If no skipping happened, q_{min} is pushed to its stack. Finally, when q_{min} is a leaf node, we call *outputSolutions* to output the current solutions from the stacks (Lines 16-18).

Algorithm QuickStack(q)

```
01: repeat (forever)
02:    skipping=false
03:    q_min=getMinSource(q)
04:    q_max=getMaxSource(q)
05:    for q_i in subTree(q)
06:       cleanStack(S_{q_i},T_{q_min}.start)
07:    if(QuickEnd(q))
08:       break
09:    if(q_max.level > q_min.level)
10:       skipping=skipAncestors(parent(q_max), T_{q_max}.start)
11:    else
12:       if(empty(S_{parent(q_min)}))
13:          skipping=skipDescendants(q_min, T_{parent(q_min)}.start))
14:    if ( ¬ skipping)
15:       moveStreamToStack(q_min)
16:       if (isLeaf(q_min))
17:          outputSolutions(S_{q_min})
18:          pop(S_{q_min})
```

Function getMinSource(q)
return $q_i \in subTree(q)$ s.t. $T_{q_i}.start$ is minimal

Function getMaxSource(q)
return $q_i \in subTree(q)$ s.t. $T_{q_i}.start$ is maximal

Procedure cleanStack(S_q, $start$)
pop all the elements from S_q that end before $start$.

Procedure moveStreamToStack(q)
1: push the element pointed by T_q in S_q, assign
its pointer to point to the top of $S_{parent(q)}$.
2: advance(T_q).

Procedure outputSolutions(S_q)
output all the solutions contained in the stacks.

Figure 3: The QuickStack algorithm

In order to fully understand our algorithm, we should first introduce the three main functions: *QuickEnd*, *skipAncestors* and *skipDescendants*.

3.1 The QuickEnd function

The *end* function proposed for *PathStack* and *TwigStack*, which was adopted in most of the follow-up works [16, 19, 9, 8], stops the algorithm when the stream of any of the query's leaf nodes ends. In this section, we propose a different stop criterion, *QuickEnd*, to terminate the algorithm earlier without missing any results.

Theorem 1 *No more solutions can be found for an XML path query if the stream of any node ends and its stack is empty.*

To show the correctness of Theorem 1, notice that the elements of the streams are pushed into the stacks in an increasing order of their *start* position. So, if the stream of a node n has finished, we know there are no more n elements to participate in future solutions. Furthermore, if the stack S_n is empty, then none of the remaining elements from n's descendants can be part of the results. Thus, the algorithm can terminate safely without missing any solutions.

Function QuickEnd(q): boolean

1: if ($\exists\ q_i \in$ subTree(q): eos(T_{q_i}) $\wedge$ empty(S_{q_i}))
2: return true
3: return false

Figure 4: The QuickEnd function

Note that the use of *QuickEnd* is not restricted to *QuickStack*; *it can also be used with other similar algorithms like PathStack and TwigStack.*

Example 2: Consider query //a//b//c and the element sets in Figure 5; the only match is (a_1, b_1, c_1).

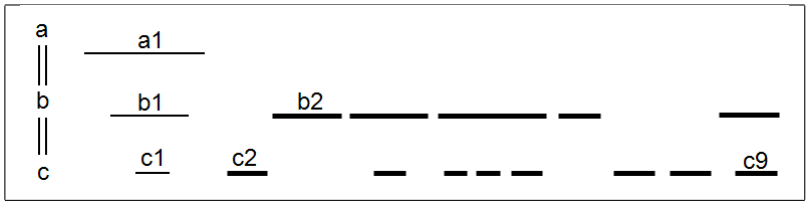

Figure 5: The early termination of *QuickStack* using the *QuickEnd* function. The skipped elements appear as the thick line segments

Right after finding the solution (a_1, b_1, c_1): the cursors are $(null, b_2, c_2)$. In the next iteration, c_2 is pushed to S_c and it empties all the stacks. At this point, the condition of *QuickEnd* is satisfied because a's stream has ended and its stack is empty. Therefore, the algorithm terminates without having to examine all the remaining elements. In comparison, if we use the original *end* function, then we have to examine all the elements up to c_9.

3.2 The skipAncestors function

The key idea for skipping ancestors is that if an element e from node q does not contain an element e' from $child(q)$, then there is no need to examine e and all its ancestors.

Function skipAncestors(q, *childStart*): boolean

```
1: skipping=false
2: while(¬ eos(T_q) ∧ T_q.end < childStart)
3:    skipping=true
4:    advance(T_q)
5: if(isRoot(q))
6:    return skipping
7: else
8:    return skipAncestors(parent(q), max(T_q.start, childStart))
            ∨ skipping
```

Figure 6: The skipAncestors function

The *skipAncestors* function takes two arguments: a query node q and the *start* value of the current element of *child(q)* (Figure 3: line 10). The function is called recursively on all the parents of q (up to the root) and it returns true if any element is skipped along the whole path. In line 2, all the elements that end before *childStart* are skipped. In line 8, the function is called on q's parent and the second parameter gets the maximum $T_q.start$ and *childStart*. The reason for taking the maximum is that the element with the biggest start is the first element that could participate in a solution. Therefore, we can skip to the further element, while maintaining the correctness of the results.

Example 3: Consider query a//b//c//d and the elements in Figure 7 ;this query does not have any matches.

Initially, the cursors point to (a_1, b_1, c_1, d_1). *QuickStack* identifies node d as q_{max} (since d_1 has the highest start value among the current elements) and it calls *skipAncestors* on node c to skip all the elements from c, b and a that end before $d_1.start$. As a result, the cursors advance directly to c_5, b_6

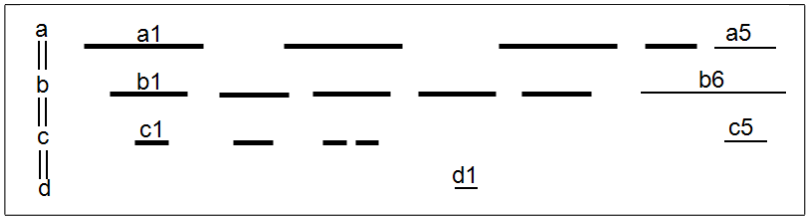

Figure 7: The application of the *skipAncestors* function. The skipped elements appear as the thick line segments

and a_5 and skip processing all of the elements before.

3.3 The skipDescendants function

The *skipDescendants* function applies a similar technique as *skipAncestors*: if an element e from node q is not contained within an element e' from $parent(q)$, then there is no need to examine e and all its descendants.

Function skipDescendants(q, $parentStart$): boolean

```
1: skipping=false
2: while(¬ eos(T_q) ∧ T_q.start < parentStart)
3:    skipping=true
4:    advance(T_q)
5: if(isLeaf(q))
6:    return skipping
7: else
8:    return skipDescendants(child(q), T_q.start) ∨ skipping
```

Figure 8: The skipDescendants function

The *skipDescendants* function takes two arguments: the query node q and the *start* value of the current element of *parent(q)* (Figure 3: line 13). Like *skipAncestors*, the function returns true if skipping happened. In line 2, all the elements that start before *parentStart* are skipped because they do not have a parent element (remember that the intervals could not partially overlap). In line 8, the function is recursively called on the child node of q[3].

Before we call *skipDescendants* on the node q_{min} in the *QuickStack* algorithm, we check that the stack of $parent(q_{min})$ is empty (Figure 3: line 12). The reason for this is that if there are some elements in $S_{parent(q_{min})}$, then these would be legitimate parents of $T_{q_{min}}$ (otherwise, they would have

[3]Here, there is no need to take the maximum as in *skipAncestors* because $T_q.start$ is always bigger than *parentStart*.

been popped out when q_{min} was pushed). Consequently, we cannot skip the descendants of q_{min}.

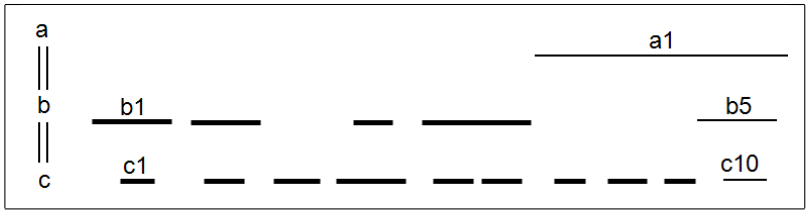

Figure 9: The application of *skipDescendants* function. The skipped elements appear as the thick line segments

Example 4: Consider query a//b//c and the elements in Figure 9; the query has one match: (a_1,b_5,c_{10}).
In this example, *QuickStack* identifies node b as q_{min} and calls *skipDescendants* on b to skip all the elements from b and c that start before $a_1.start$. As a result, T_b and T_a advance directly to b_5 and c_{10}.

***QuickStack* on Example 1 (Figure 2)**
Consider again the example of Figure 2. First, the algorithm calls *skipAncestors* on node b[4] and skips elements b_1 and a_1. Next, it calls *skipDescendants* to skip c_1 and *skipAncestors* to skip b_2. The algorithm pushes the elements a_2, b_3 and c_2 in the stacks and calls *outputSolutions* to output this result. Then, when b_4 is pushed in S_b, it pops out a_2, b_3 and c_2 from their stacks. At this point, the condition of *QuickEnd* is satisfied since a's stream is finished and its stack is empty, thus the algorithm terminates safely. From this example, we can see that *QuickStack* had to examine fewer elements than *TwigStack*.

4 QuickStack for twig queries (TQS)

In this section, we propose our *TQS* (Twig QuickStack) algorithm to answer twig queries efficiently. Recall that the solutions of a twig pattern can be seen as an intersection of all the solutions for its individual root-to-leaf paths. A naïve approach would be to use *QuickStack* to evaluate each path separately, then join all of the intermediate solutions afterward. A much more efficient approach (in terms of execution time and intermediate solution sizes) is our proposed *TQS* algorithm: each time the algorithm evaluates a path, it outputs the solutions that satisfy both the current path and all of the

[4] c_1 is $T_{q_{max}}$ in this case, so *skipAncestors* is called on its parent node b.

previously evaluated paths, i.e., it merges the solutions incrementally while matching the individual paths.

In order to explain our proposed *TQS* algorithm, we should first introduce the *QuickStack2* algorithm, which evaluates the paths of the twig and is at the crux of *TQS*. *QuickStack2* is similar to the original *QuickStack* algorithm (Figure 3), but has the ability to incorporate the solutions of the previously evaluated paths, while evaluating the current path q from the twig. Let us refer to the part of the twig that has already been evaluated as the *processed sub-twig*. Each time *QuickStack2* evaluates a path q, it outputs only the twig solutions that satisfy both q and the *processed sub-twig*.

Algorithm QuickStack2(q, bn, bn_sol)

```
01: repeat (forever)
02:   skipping=false
03:   q_min=getMinSource(q)
04:   if(q_min=bn)
05:     if(synchronize(T_{q_min}, T_{bn_sol})
06:       continue
07:   q_max=getMaxSource(q)
08:   for q_i in subTree(q)
09:     cleanStack(S_{q_i},T_{q_min}.start)
10:   if(QuickEnd(q))
11:     break
12:   if(q_max.level > q_min.level)
13:     skipping=skipAncestors(parent(q_max),T_{q_max}.start)
14:   else
15:     if(empty(S_{parent(q_min)}))
16:       skipping=skipDescendants(q_min,T_{parent(q_min)}.start))
17:   if ( ¬ skipping)
18:     moveStreamToStack(q_min)
19:     if (isLeaf(q_min))
20:       outputSolutions(S_{q_min})
21:       pop(S_{q_min})
```

Function synchronize(T_q,T_{bn_sol}): boolean

```
1: advance T_q and T_{bn_sol} to the next common element
   to both streams
2: if eos(T_{bn_sol}) then
     set T_q to the end of q's stream (so eos(T_q)=true)
3: return true only if any elements are skipped from q's stream
```

Figure 10: The QuickStack2 algorithm

The *QuickStack2* algorithm is outlined in Figure 10. In addition to the

path q, the algorithm takes two more arguments: bn, the lowest branching node[5] shared between q and the *processed sub-twig*, and bn_sol, the elements of the bn node that appear in the solutions of the *processed sub-twig*. The elements in bn_sol form an ordered stream and we define T_{bn_sol} to be its cursor. The code in Figure 10 in bold is the extra code needed to upgrade *QuickStack* to *QuickStack2*. In line 4, if q_{min} is the branching node, we call the *synchronize* function to skip all the elements of q_{min} that do not match any elements from bn_sol. If any elements were skipped from q_{min}, we start over at line 1 to find a new q_{min}. Therefore, *QuickStack2* can skip the elements that are not part of the *processed sub-twig* solutions, even if they are part of q's solutions.

To minimize the number of processed elements by *TQS*, we need to evaluate the paths that are very selective (expected to return the smallest number of matches) before the paths that have lower selectivity. Our heuristic for ordering the paths by selectivity is based on the following observation.

Proposition 1 *The number of stream elements of the leaf node in an XML path query provides an upper bound on the total number of solutions for that query.*

We can verify that this claim is true because any descendant element in the XML document cannot belong to more than one ancestor element according to the strictly nested property of the XML document.

Motivated by Proposition 1, we evaluate the paths of the twig in an ascending order according to the number of elements associated with their leaf nodes.

Putting it all together, *TQS* works as follows:

1. Decompose the twig into root-to-leaf paths.
2. Order the paths p_i according the number of elements in their leaf nodes (ascending order).
3. Call *QuickStack* on the first path p_1 and output sol_1, the matches of p_1
4. For $i = 2$ to $numPaths$:
   (a) Identify the lowest branching node, bn, between p_i (the current path) and the *processed sub-twig* (currently contains all the paths from p_0 to p_{i-1}).

[5]A branching node is a query node with more than one child.

(b) Extract bn_sol, the elements of bn from sol_{i-1}.

(c) Call *QuickStack2* on the current path p_i, given the elements in bn_sol. The algorithm outputs sol_i, the solutions that satisfy both p_i and the *processed sub-twig*.

(d) Update the *processed sub-twig* to include path p_i.

5. Finally, output $sol_{numPaths}$, the solutions for the full twig.

We can see that *TQS* may produce more intermediate solutions than *TwigStack* because, when *TQS* evaluates path i, some of the solutions in sol_i may not be part of the final twig results. However, *TQS* can skip a lot of the elements that *TwigStack* has to examine, as the following example illustrates.

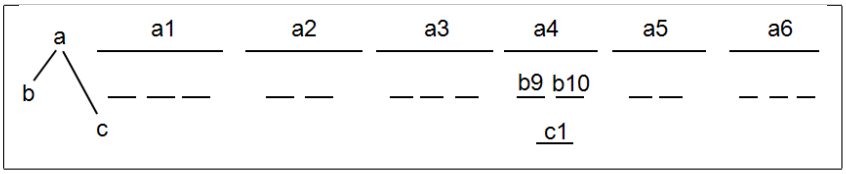

Figure 11: An example showing how *TQS* Skips the elements that are not in the twig Solutions

Example 4: Consider the twig query $/a[b][c]$ and the elements in Figure 11. This query has only two solutions a4[b9][c1] and a4[b10][c1].

***TQS* and *TwigStack* on Example 4**

To evaluate this query, *TQS* first calls *QuickStack* on the path $/a/c$ (because c has fewer elements than b) and it outputs the solution a_4/c_1. Next, *TQS* calls *QuickStack2* to evaluate $/a/b$. At this point, the *synchronize* function advances T_a all the way to a_4 (since a_4 is the only a element in the solutions for $/a/c$) and then *skipDescendents* advances T_b directly to b_9. After finding the two relevant solutions a_4/b_9 and a_4/b_{10}, *synchronize* jumps T_a to the end of a's stream. Consequently, *QuickEnd* ends the algorithm since $eos(T_a) \wedge empty(S_a)$.

In comparison, when *TwigStack* evaluates the path a/b, it has to examine all of the irrelevant elements b_1 to b_9 because b is a descendant extension for all of them.

5 Multiquery QuickStack (MQS)

In this section, we consider the scenario of matching multiple queries against an XML document, which usually occurs in XML filtering systems.

The problem definition is: *Given an XML document D and a set of queries $Q = \{q_1, ..., q_n\}$, find the set $R = \{r_1, ..., r_n\}$, where r_i is the answer for query q_i on D.*

We saw that the first step to evaluate a query is to build the streams of the elements from the XML document that match the query nodes. In this multiquery framework, we use the concept of the *generalized index*, which is a structure that contains all the elements that could participate in any of the input queries. This structure is represented by a tree that contains all the nodes from the input queries. The predicate on each node is the union of all the predicates on the corresponding nodes from the queries. For example, consider the following three queries:

- Q1=/bookstore[num=1]/book[price<50]/title
- Q2=/bookstore[num=50]/book[price<30]/chapter/title
- Q3=/bookstore[num>10 and num<20]/book[title="XML tutorial" and price <20 and price >80]/chapter/title.

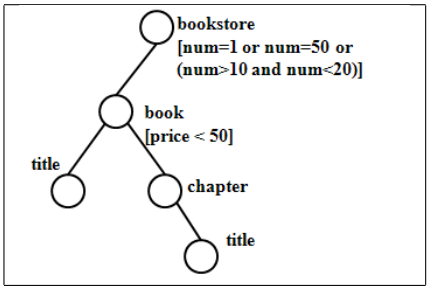

Figure 12: the structure of the generalized index for queries Q1, Q2, and Q3

The *generalized index* from these three queries contains all the stream elements that the algorithm needs to evaluate any of them, and is shown in Figure 12. Note that the predicate on the *book* element (price<50) is a general predicate that allows the stream of *book* to contain all the *book* elements that are needed in queries Q1, Q2 and Q3. The two *title* leaves in the *generalized index* will have different elements since we can check the direct parent of the *title* element in the document to determine which node it belongs to.

When using *QuickStack/TQS* to evaluate multiple queries against an XML document, we have several options to consider:

1. Option 1 (Naïve way): Parse the document for each query to build its element streams and execute the algorithm.

2. Option 2: Parse the document once and construct the *generalized index*. Then run the algorithm for each query on this common structure and check the query's predicates during the execution of the algorithm.
3. Option 3: Build the *generalized index* as in option 2, but then for each query: run the algorithm on the query's own element streams after extracting them from this common index. We extract the element streams of a query Q_i simply by applying Q_i's predicates on the corresponding nodes from the *generalized index*.

Clearly, the naïve way (option 1) is very wasteful because parsing the document is an expensive operation, and we do not want to do it for every input query. Option 2 is also not a good idea, because running *QuickStack*/*TQS* on the *generalized index* restricts its ability to skip the elements, and thus hurts its performance. For instance, when we execute *QuickStack* on Q3's own element streams, the algorithm skips a lot of *bookstore*, *chapter*, and *title* elements because the predicate on *book* is very selective (*book*'s stream is very short). But, if we evaluate Q3 on the *generalized index* in Figure 12, the stream of the *book* node will be much longer, and the algorithm will spend a lot of time processing extra elements that cannot be part of Q3's result. Option 3 is more efficient than the others and it will be considered in our experiments.

6 Pre-processing Techniques

In this section we explain some techniques we can apply while scanning the document to reduce the number of elements in the streams, and thus speed up the evaluation time of the algorithm. These techniques are *general and can be applied on any XML processing algorithm that uses the same intermediate structure.*

First, it is more efficient to evaluate the predicates while building the element streams, instead of doing it during the course of the algorithm. So if the predicate on a node is selective, the stream of this node will be short and the algorithm will skip a lot of irrelevant elements.

In addition, when a path in the twig has only one direct child of the branching node, we can evaluate this path as a predicate on the branching node. For example, consider the three paths query: $a[b][c]//d$. This query can be rewritten as $a[b \text{ and } c]//d$, and the two paths a/b and a/c can be evaluated as a predicate on a while building the element streams. In other words, we only add the a elements that have two children b and c to a's

stream while scanning the document, then we evaluate the path query $a//d$ using *QuickStack* to get the answers for the full twig query.

The second technique we introduce is the *depth evaluation* technique. Recall that each edge from a node p to a node n in the query tree imposes a constraint on the number of elements that can appear between n and p in the results. We call this number the *relative depth* of n from its parent p and denote it as $RD(n)$. $RD(n)$ is zero for a parent-child edge and is greater or equal to zero for an ancestor-descendant edge. With the *relative depth* concept introduced, we are able to represent queries that contain *wild cards*, since *wild cards* may impose new conditions on the *relative depth*, which cannot be represented by a parent-child edge nor an ancestor-descendant edge. For example, the query $/x/*/*/y/*//z$ specifies that $RD(x) = 0$, $RD(y) = 2$ and $RD(z) >= 1$.

When we build the element streams, we examine the XML elements one after the other and we add only the elements that satisfy the relative depth condition. Specifically, we define a boolean flag F_n and an integer variable D_n for each node n in the query. We set F_n to *true* when we encounter an open tag $< n >$ and set it to *false* when the tag is closed $< /n >$. If F_n is true, we increment D_n for each open tag and decrement it for each close tag in the document. Consequently, when F_n is true, D_n measures the depth of the current XML element from its ancestor n. Thus, before adding an element to the stream of $child(n)$, we first check that D_n=$RD(child(n))$.

Note that the depth evaluation for wild cards is more complicated for documents with recursive definitions. However, it is still easy to evaluate the basic parent-child and ancestor-descendant edges.

7 Experiments

We implemented our proposed *QuickStack* algorithm, as well as the *TwigStack* and *PathStack* algorithms in JAVA using JDK 1.5, sharing as much code and as many data structures as possible for a fair comparison. We used the *YFilter* system (version 1.0) [22] developed at the University of California at Berkeley, which is also implemented in Java.

We conducted our experiments on both synthetic and real-world data. For synthetic data, we used a bookstores dataset and another randomized dataset. We also used two real-world datasets: DBLP [18] and NASA [21]. We chose different types of queries over the datasets in order to give a comprehensive comparison between the algorithms.

The machine we used in our experiments is a Dell PowerEdge 2950 server

with two Dual-Core Intel Xeon 3GHz CPU processors and 4MB L2 cache. It is equipped with 16GB RAM, running Redhat Linux Version 3.4.6-3.

In section 7.1, we present experimental results on processing single path queries using *QuickStack*, *TwigStack* and *PathStack*. In section 7.2, we compare the performance of *QuickStack* with *TwigStack* on more complex twig queries. In section 7.3, we show the results of applying our *MQS* approach to process multiple queries and compare it against *YFilter*.

7.1 Single path queries

7.1.1 The bookstores dataset

Dataset description (Figure 13 and Table 1)
This dataset contains information about bookstores and the books they

```
<!ELEMENT BOOKSTORE (NAME, NUM, BOOK+)>
<!ELEMENT BOOK (TITLE, PRICE, CHAPTER+)>
<!ELEMENT CHAPTER (TITLE, NUM_OF_PAGES)>
<!ATTLIST BOOKSTORE STATE CDATA #REQUIRED>
<!ELEMENT NUM (#PCDATA)>
<!ELEMENT PRICE (#PCDATA)>
<!ELEMENT TITLE (#PCDATA)>
<!ELEMENT NUM_OF_PAGES (#PCDATA)>
```

Figure 13: The DTD of the bookstores dataset

have. The DTD of bookstores is shown in Figure 13. The bookstores are numbered sequentially according to their appearance in the XML file (the *num* element) and are randomly distributed among 7 different states (the *state* attribute). The books are given sequential titles (book1, book2, ...), so that each title is unique. The book prices vary from 10 to 100. Each bookstore has between 50 and 250 books and each book contains 5 to 20 chapters. The dataset file size is **121MB**, and the frequency of each element is shown in Table 1.

Element's name	**Number of occurrences**
bookstore (num)	1,000
book (price)	147,680
chapter (num_of_pages)	1,846,217
title	1,993,897

Table 1: Element frequencies of bookstores

Results (Table 3 and Figure 14)
We used the single root-to-leaf queries in Table 2. The last column of this table characterizes the selectivity for each query by showing the number of results the query has in the file.

Query	XPath expression	Number of matches
Q1	/*/bookstore [num=1] /book/price	120
Q2	//bookstore [num > 100 and num < 105] /book/chapter/title	7,148
Q3	//bookstore [num = 10 or num =120] /book/chapter/num_of_pages	5,209
Q4	//bookstore [num = 200] /book [price ≥ 20 and price ≤30] /chapter/title	374
Q5	//bookstore/book [title="book6985"] /chapter/title	20
Q6	//bookstore [@state="PA"] /book [price < 30] /chapter [title="chapter4"] /num_of_pages	4,464
Q7	//bookstore /book/chapter/title	1,846,217

Table 2: Queries over the bookstores dataset

Table 3 gives the execution times of the three algorithms in milliseconds. The first column is the query and the second column is the time taken to parse the XML document and build the element streams for the query nodes (similar to Figure 2). This cost is mainly IO cost and it is the same for each of the three algorithms. The last three columns show the execution time of *PathStack*, *TwigStack* and *QuickStack*. We can see that *QuickStack* consistently outperforms *PathStack* and *TwigStack* (except for Q7 as we explain later).

Query	scanning cost	PathStack	TwigStack	QuickStack
Q1	2,558	1,540	128	16
Q2	3,264	20,079	1,872	192
Q3	3,456	22,184	1,350	204
Q4	3,560	17,742	1,320	84
Q5	3,539	16,924	1,050	22
Q6	3,500	8,956	716	370
Q7	3,105	25,614	26,894	26,772

Table 3: Bookstores dataset: the evaluation time (in ms) of the three algorithms and the time to build the element streams for the queries in Table 2. The evaluation times for *TwigStack* and *QuickStack* are plotted in Figure 14

Figure 14 shows the execution time of the algorithms. We omit *PathStack* from all the figures because it is much slower than both *TwigStack* and *QuickStack*. We also omit Q7 since the time to process this query is much higher than the other queries, for both *TwigStack* and *QuickStack*.

To understand the reason behind the superior behavior of *QuickStack*

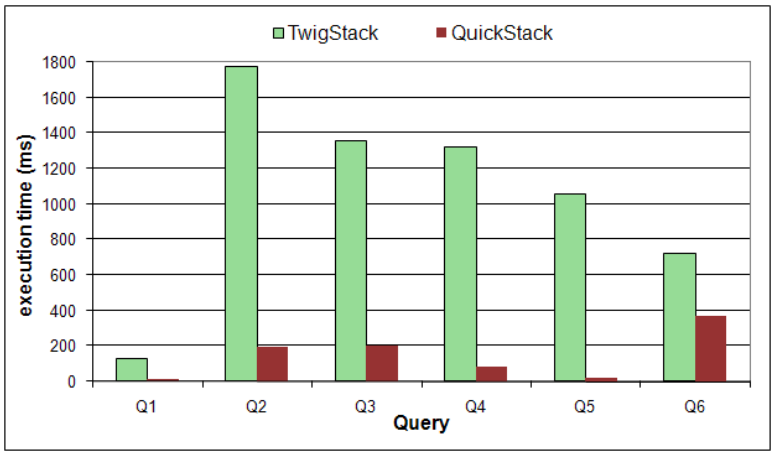

Figure 14: Bookstores dataset: the evaluation time (in ms) of *QuickStack* and *TwigStack* for the queries in Table 2 (excluding Q7, since it is out of range)

over *TwigStack*, consider the case of query Q3[6]. First, *QuickStack* calls *skipDescendants* to skip all the *book*, *chapter*, and *num_of_pages* elements that are before the 10th *bookstore* (remember that the bookstores are sorted in the file according to the *num* element). Then *skipDescendants* also skips the elements between the two bookstores in the predicate[7]. Finally, after processing the descendants of the 120th bookstore, *QuickEnd* terminates the algorithm. On the other hand, *TwigStack* has to examine all the elements in the streams since they all have descendant extension solutions.

Notice that Q7 is an extreme case because all the elements in the streams participate in the result (since it does not impose any predicate), and *PathStack* performs slightly better than both algorithms for this special case. The execution time of *PathStack* for this query is similar to the other queries since it processes all elements in the streams regardless of whether they participate in the results or not.

Query Q1 is a good example of the benefit of using the *QuickEnd* function since it allows the algorithm to stop immediately after processing the books of the first bookstore.

As we can see, one of the highest performance gain occurs for query Q5, where *QuickStack* is about 50 times faster than *TwigStack*. The reason is that Q5 is very selective: it has only one element in the stream of the *book* node (the book title is unique in the file) and has long streams for the

[6] Q3 asks about the number of pages for the chapters of all the books in the 10th or the 120th bookstore.

[7] *skipDescendants* will be activated on these elements since they do not have a parent.

chapter and *title* elements. Therefore the skipping abilities of *QuickStack* are fully exploited. However, the difference is less significant for Q6 because it returns a lot of results and skipping the elements in the streams becomes less effective. In this case, *QuickStack* is about twice as fast as *TwigStack*.

7.1.2 The DBLP dataset

Query	XPath expression	Number of matches
Q1	//inproceedings [author="Michael Stonebraker" and year=2003] /title	5
Q2	//inproceedings [title="Ratio Rules: A New Paradigm for Fast, Quantifiable Data Mining."] /author	4
Q3	//inproceedings [(author="Christos Faloutsos" or author="Rajeev Agrawal" or author="Soumen Chakrabarti") and year=2000] /author	46
Q4	//inproceedings [title="Spatial Join Selectivity Using Power Laws."] /cite	35
Q5	//article [author="Michael Stonebraker"] /cite	332
Q6	//article [journal="VLDB J."] /title	241

Table 4: Queries over DBLP dataset

Query	scanning cost	PathStack	TwigStack	QuickStack
Q1	4,692	2,110	128	22
Q2	4,596	3,614	214	26
Q3	5,015	3,006	230	32
Q4	4,117	1,188	88	22
Q5	4,036	960	80	28
Q6	4,080	2,290	144	33

Table 5: DBLP dataset: the evaluation time (in ms) of the three algorithms and the time to build the element streams for the queries in Table 4. The evaluation times for *TwigStack* and *QuickStack* are plotted in Figure 15

Dataset description
The DBLP (Digital Bibliography Library Project) file provides bibliographic information on major computer science journals and proceedings. The DBLP file that we used has size of **210 MB** and 4,884,836 elements in total. It contains information about 1,075,088 authors, 298,413 inproceedings entries and 173,630 articles. The file has a maximum depth of 6 and an average depth of about 2.9.
Results (Table 5 and Figure 15)
The execution times for the queries in Table 4 appear in Table 5 and Fig-

ure 15. For the given queries, *QuickStack* performs from 60% to 90% faster than *TwigStack* and from 97% to 99% faster than *PathStack*.

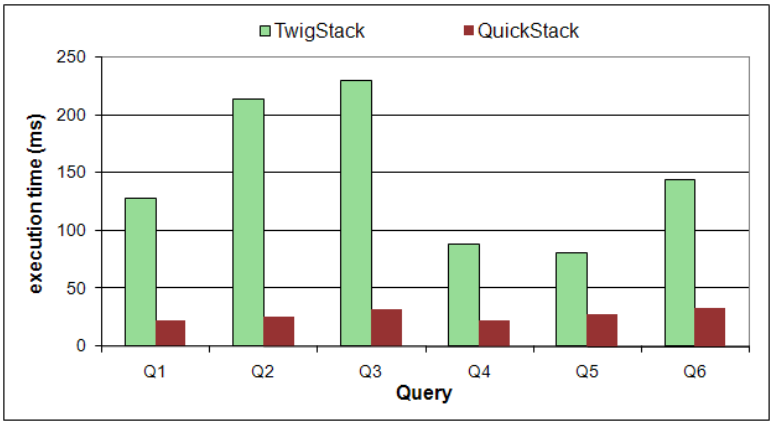

Figure 15: DBLP dataset: the evaluation time (in ms) of *QuickStack* and *TwigStack* for the queries in Table 4

7.1.3 The NASA dataset

Dataset description
This dataset is converted from a legacy flat-file format into XML format and it represents astronomical data from NASA. The file size is **23 MB**. It has 476,646 elements with a max depth of 8 and an average depth of about 5.58.
Results (Table 7 and Figure 16)
Table 7 and Figure 16 show the execution time of the algorithms for the queries in Table 6. For the given queries, *QuickStack* performs from 65% to 96% faster than *TwigStack* and from 97% to 98% faster than *PathStack*.

Section Summary: *QuickStack* consistently outperforms *TwigStack* for single-path queries and the gap increases for queries with very selective predicates.

7.2 Twig queries

7.2.1 The Random dataset

Dataset description (Figure 17 and Table 8)
To test the performance for twig queries, we chose to generate another synthetic dataset that allows us to better control the relationship between the

Query	XPath expression	Number of matches
Q1	//dataset [title="Astrographic Catalogue"] /reference//author/lastName	4
Q2	//dataset//fields/field [name="DE"] /definition	10
Q3	//dataset//reference/source//author [lastName="Mermilliod"] /initial	40
Q4	//dataset//fields/field [name="L"] /definition/footnote/para	9

Table 6: Queries over NASA dataset

Query	scanning cost	PathStack	TwigStack	QuickStack
Q1	410	948	54	2
Q2	373	1,076	72	18
Q3	360	860	52	18
Q4	394	1,316	106	26

Table 7: NASA dataset: the evaluation time (in ms) of the three algorithms and the time to build the element streams for the queries in Table 6. The evaluation times for *TwigStack* and *QuickStack* are plotted in Figure 16

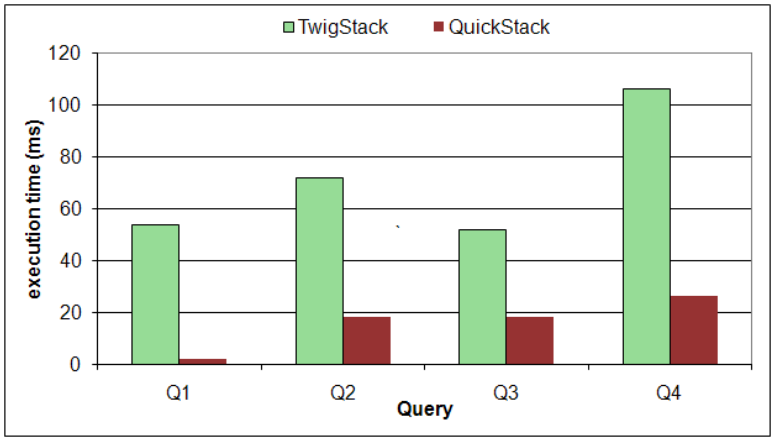

Figure 16: NASA dataset: the evaluation time (in ms) of *QuickStack* and *TwigStack* for the queries in Table 6

algorithms and the characteristics of the queries over the dataset. The structure of this randomized dataset is shown in Figure 17. We label the edges in the figure by their selectivity. For instance, there is a 60% probability

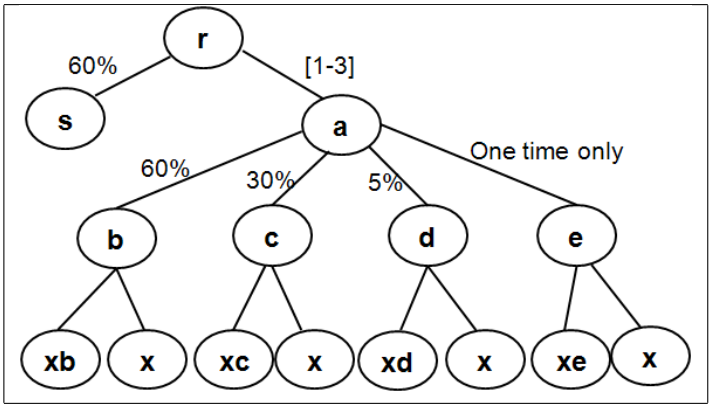

Figure 17: The structure of the random dataset

that a specific element r will have a child with label s. Also, each element r has between 1 and 3 children with label a. We omit the labels on the edges connected to the leaves because they have a 100% selectivity.

The dataset size is **15,747KB** and it has 2,187,515 elements in total. Table 8 shows the frequencies of the elements.

Element's name	Number of occurrences
r	300,000
s	179,813
a	599,131
b (xb)	359,736
c (xc)	179,524
d (xd)	30,019
e (xe)	1
x	569,280

Table 8: Element frequencies of the random dataset

Results (Table 10 and Figure 18)
We tested the algorithms for the twig queries in Table 9. For these experiments, we evaluate the queries as twig patterns without applying the techniques in Section 6 to evaluate simple paths as predicates on the branching node. For instance, Q2 is evaluated as a twig with two paths: $//a/b$ and $//a/c$. However, we still do the depth evaluation to reduce the number of elements in the streams.

Table 10 shows the execution time for the three algorithms, as well as the number of intermediate solutions they produced. *Naïve QuickStack* denotes the algorithm that executes *QuickStack* on each path independently.

Query	XPath expression	Number of matches
Q1	//a [b/x] /e	1
Q2	//a [b] /c	107,658
Q3	//a [b] [c] /d	5,473
Q4	//a [b] [c/x] /d	5,473
Q5	//a [b/xb] [c/xc] /d/xd	5,473
Q6	//r [s] /a [b] /d/x	10,672

Table 9: Twig queries over the random dataset

Query	**Naïve QuickStack**		**TwigStack**		**TQS**	
	Intermediate path solutions	evaluation time (ms)	Intermediate path solutions	evaluation time (ms)	Intermediate path solutions	evaluation time (ms)
Q1	359,737	1,555	2	302	2	4
Q2	539,260	1,629	215,316	1,297	287,182	693
Q3	569,279	1,575	16,419	532	44,601	287
Q4	569,279	2,036	16,419	1,391	44,601	319
Q5	569,279	2,234	16,419	1,710	44,601	368
Q6	569,568	2,498	32,016	1,567	58,736	606

Table 10: Random dataset: the evaluation time (in ms) and the number of intermediate solutions of the twig queries in Table 9. The evaluation times are plotted in Figure 18

Let us take a look at how *TQS* evaluates query Q3. First, *QuickStack* evaluates the most selective path (//a/d) since the number of elements in d is the least among the leaf nodes. Next, *QuickStack2* evaluates //a/c utilizing the a elements from the solutions of the first path. Lastly, *QuickStack2* evaluates //a/b and it skips the elements that do not match any solution for the *processed sub-twig* //a[c]/d.

We can see that *TQS* is about 75 times faster than *TwigStack* for Q1. The reason is that Q1 has only one match in the document (because we have only one e element). Therefore, *TQS* can directly pinpoint the only solution to //a/b/x that is relevant. On the other hand, *Naïve QuickStack* is very slow in this case because it generates enormous number of intermediate solutions (equal to the number of b elements plus 1 for element e).

Finally, consider the complex query Q6. This query has two branching nodes: r and a. Thanks to our depth evaluation techniques, the number of elements in x's stream will be equal to the number of d elements. Thus, *TQS* identifies path $/r//a/d/x$ as the most selective and evaluates it first using *QuickStack*. After that, it evaluates $//r/s$ and $/r/a/b$ (in this order)

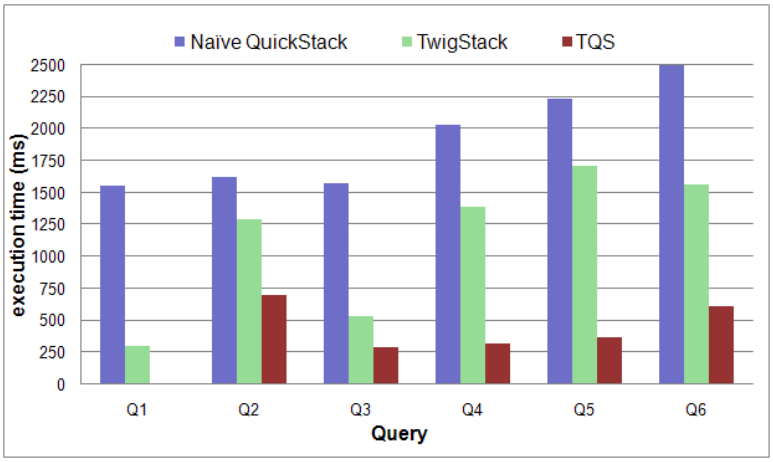

Figure 18: Random dataset: the evaluation time (in ms) of *Naïve QuickStack*, *TwigStack*, and *TQS*

using *QuickStack2*.

7.2.2 The bookstores, DBLP and NASA datasets

Query	XPath expression	Number of matches
BS_Q1	/*/bookstore [@state="MA"] [book [price=10]] /book [price=90]	2,006
BS_Q2	//bookstore [book [title="book77555"]]/book [price=50] /chapter/title	252
BS_Q3	//bookstore[book [title="book98000"]] [book [title="book98010"]] /book/title	50
DBLP_Q1	//inproceedings [author="Michael Stonebraker"] [year=2003] /title	5
DBLP_Q2	//article [journal="VLDB J."] /title	241
DBLP_Q3	//inproceedings [author="Nicolas Bruno"] [author="Nick Koudas"] [year=2002] /title	1
NASA_Q1	//journal[author [lastName="pereira"] /initial] /title	1
NASA_Q2	//fields[field [name="H1-36"]] [field[name="lambda"]] /field [name="He2-38"] /definition	1
NASA_Q3	//fields[field [definition="Distance"]] /field/units	859

Table 11: Twig queries over bookstores, DBLP, and NASA datasets

Results (Table 12 and Figure 19)
We also tested the evaluation of the twig queries in Table 11 over the bookstores, DBLP and NASA datasets. The results appear in Table 12 and Figure 19. We can observe some cases, like BS_Q2, NASA_Q1, NASA_Q2, where even *Naïve QuickStack* can outperform *TwigStack*. To explain the reason, consider NASA_Q2 for instance. This query is not only very selective as a whole, but also each of its individual paths are very selective. Hence, the number of the excess intermediate solutions that *Naïve Quick-*

Query	Naïve QuickStack		TwigStack		TQS	
	Intermediate path solutions	evaluation time (ms)	Intermediate path solutions	evaluation time (ms)	Intermediate path solutions	evaluation time (ms)
BS_Q1	4,656	86	4,012	182	4,326	91
BS_Q2	202,001	1,589	253	2,200	253	33
BS_Q3	147,680	1,124	52	248	52	10
DBLP_Q1	326,955	2,848	15	606	120	34
DBLP_Q2	173,871	656	482	495	482	68
DBLP_Q3	29,018	1,052	3	750	9	65
NASA_Q1	2,383	40	3	56	3	6
NASA_Q2	204	16	3	270	3	8
NASA_Q3	60,663	424	917	230	917	51

Table 12: Bookstores, DBLP, NASA datasets: the evaluation time (in ms) and the number of intermediate solutions for the twig queries in Table 11. The evaluation times are plotted in Figure 19

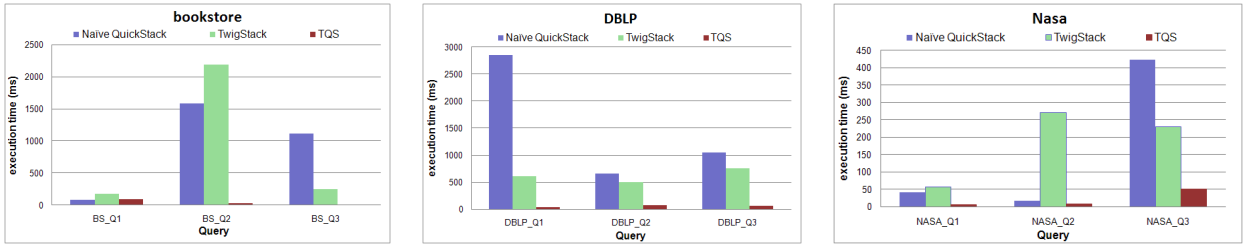

Figure 19: Bookstores, DBLP, NASA datasets: the evaluation time (in ms) of *Naïve QuickStack*, *TwigStack*, and *TQS*

Stack produces is small and it can still outperform *TwigStack* in such cases.

Section Summary: Although the number of intermediate solutions that *TwigStack* generates lower-bounds that of *TQS*, *TQS* can be several times faster in practice. The performance gap increases when there is a very selective path in the twig pattern.

7.3 Multiple queries

In this section, we compare the performance of *QuickStack*, *TwigStack*, and *YFilter*, under a varying number of input queries. We used a program that generates random single-path queries over the bookstores dataset. These queries vary in their selectivity and are similar to the ones presented in Table 2 (with different values for the predicates).

We used the same approach in Section 7.3 to evaluate the queries using *TwigStack*, which we denote as *MTQ*. We do the processing (for both *MTS* and *MQS*) in two phases: the first phase builds the element streams for all

the input queries, while the second phase evaluates the queries and output their results. So the time of the first phase is the time needed to parse the document, build the *generalized index* and extract from it the element streams of all the individual queries. We use the term *evaluation time* to denote the time of the second phase, while *total time* to denote the time for both phases.

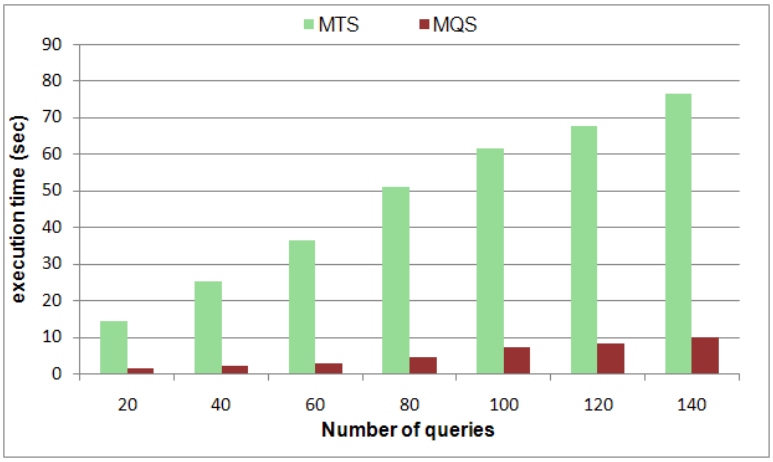

Figure 20: the Evaluation time (in sec.) of *MQS* and *MTS* for multiple input queries over the bookstores dataset

Figure 20 compares the evaluation time of *MTS* and *MQS*. Here, we exclude the time of the first phase because it is the same for both algorithms. Notice that as the number of queries increases, the gap in the evaluation time between *MQS* and *MTS* becomes bigger. For instance, *MQS* is more than 7 times faster than *MTS* for 140 input queries ($\simeq 90\%$ reduction in evaluation time).

Figure 21 compares both algorithms with *YFilter*. In order to have a fair comparison, we should use the total time of *MTS* and *MQS* since *YFilter* evaluates the queries while scanning the document. Note that *MQS* clearly outperforms *YFilter* when the number of queries is relatively small. For instance, for 60 queries, *MQS* reduces the execution time by 61% compared with *YFilter*. The reason for this is that *MQS* can avoid processing large portions of the elements, while *YFilter* has to consume all the elements in the document. However, these gains tend to diminish as we increase the number of queries since the performance of *YFilter* is almost independent on the number of input queries.

These results suggest that it would be best to use a hybrid approach that

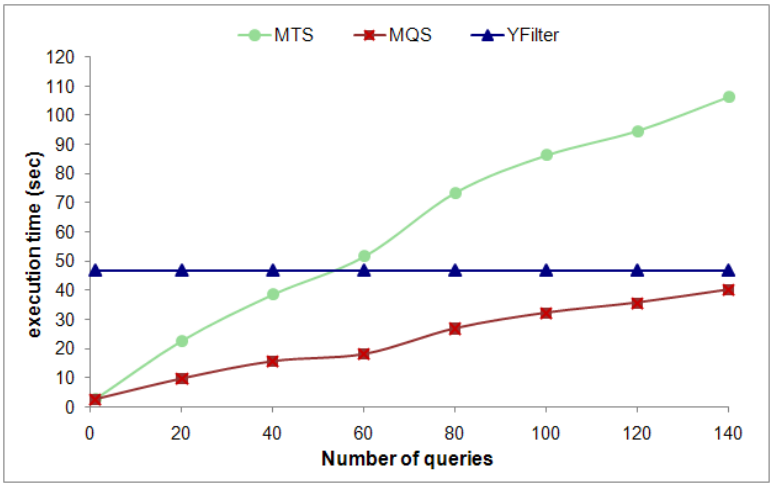

Figure 21: The total time in sec. of *MQS* and *MTS* against *YFilter* for multiple input queries over the bookstores dataset

switches automatically between *MQS* and *YFilter* based on the number of input queries and the size of the document. So for a big document and relatively small number of queries, using *MQS* is definitely better than scanning the whole document with YFilter. One can draw an analogy between XML and traditional databases by seeing *YFilter* as doing sequential scan, while *MQS* as accessing the database through an index.

Section Summary: *MQS* is more efficient than *YFilter* when the number of queries is small and the XML document is large.

8 Conclusions and Future work

In this paper we presented *QuickStack*, an enhanced holistic join algorithm for matching XML query patterns. *QuickStack* can effectively avoid processing unnecessary elements by skipping both ancestors and descendants that do not have a match in the document. The algorithm is extended to process twig queries by joining the query path solutions while matching the paths, instead of doing the join as a separate phase after the matching. Experimental results showed that our method is more efficient than *TwigStack* for both simple and complex XPath queries.

Regarding our future work, we plan to try augmenting the element streams with *XR-tree* [15]. We believe that *XR-tree* index will work perfectly well with our *skipAncestors* and *skipDescendants* functions and can

accelerate the evaluation when the node streams are very long.

Another interesting avenue to explore, encouraged by the experimental results, is to design a new parallel framework similar to the *IndexFilter* algorithm proposed in [5], but which relies on *QuickStack* instead of *PathStack*.

References

[1] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proceedings of ICDE*, 2002.

[2] M. Altinel and M. J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proceedings of the 26th VLDB*, 2000.

[3] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Simon. XML Path Language (XPath) 2.0 W3C working draft 16. Aug. 2002.

[4] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simon. XQuery 1.0: An XML Query Language W3C working draft 16. Aug. 2002.

[5] N. Bruno, L. Gravano, N. Koudas, and D. Srivastava. Navigation -vs. Index-Based XML Multi-Query Processing. In *Proceedings of ICDE*, 2003.

[6] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proceedings of SIGMOD*, 2002.

[7] B. Chen, T. W. Ling, M. T. Ozsu, and Z. Zhu. On Label Stream Partition for Efficient Holistic Twig Join. In *proceedings of DASFAA*, 2007.

[8] 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 *proceedings of VLDB*, 2006.

[9] T. Chen, J. Lu, and T. W. Ling. On Boosting Holism in XML Twig Pattern Matching Using Structural Indexing Techniques. In *Proceedings of SIGMOD*, 2005.

[10] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient Structural Joins on Indexed XML Documents. In *Proceedings of VLDB*, 2002.

[11] B. Choi, M. Mahoui, and D. Wood. On the Optimality of Holistic Algorithms for Twig Queries. In *Proceedings of DEXA*, 2003.

[12] M. P. Consens and T. Milo. Algebras for Querying Text Regions. In *Proceedings of PODS*, 1995.

[13] Y. Diao and M. J. Franklin. High-Performance XML Filtering: An Overview of YFilter. In *Proceedings of ICDE*, 2003.

[14] H. Jiang, H. Lu, and W. Wang. Efficient Processing of XML Twig Queries with OR-Predicates. In *Proceedings of SIGMOD*, 2004.

[15] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-Tree: Indexing XML Data for Efficient Structural Joins. In *Proceedings of ICDE*, 2003.

[16] H. Jiang, H. Lu, W. Wang, and J. X. Yu. Holistic Twig Joins on Indexed XML Documents. In *Proceedings of VLDB*, 2003.

[17] L. V. S. Lakshmanan and S. Parthasarathy. On Efficient Matching of Streaming XML Documents and Queries. In *proceedings of EDBT*, 2002.

[18] M. Ley. Dblp. computer science bibliography. http://www.informatik.uni-trier.de/~ley/db.

[19] J. Lu, T. Chen, and T.W. Ling. Efficient Processing of XML Twig Patterns with Parent Child Edges: A Look-ahead Approach. In *Proceedings of CIKM*, 2004.

[20] J. Lu, T. W. Ling, C.-Y. Chan, and T. Chen. From Region Encoding to Extended Dewey: On Efficient Processing of XML Twig Pattern Matching. In *proceedings of VLDB*, 2005.

[21] University of Washington XML Repository. http://www.cs.washington.edu/research/xmldatasets/.

[22] YFilter 1.0 release. http://yfilter.cs.umass.edu.

[23] C. Zhang, J. Naughton, D. Dewitt, Q. Luo, and G. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *proceedings of SIGMOD*, 2001.