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

Abstract

Finding all distinct matches of an XML path query is a core operation of XML query evaluation and has been widely studies in recent years. In this paper, we propose a novel holistic twig join algorithm, namely QuickStack, for matching an XML query pattern. The proposed QuickStack algorithm extensively optimizes over the PathStack algorithm by effectively skipping the elements that do not participate in the answers. QuickStack is guaranteed to outperform the previous algorithms for single root-to-leaf queries. 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 queries. We also propose a generalization of QuickStack to answer multiple XML path queries and we compare its performance with YFilter, the state of the art for navigational based algorithms.

1. Introduction

XML is emerging as a 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 considered semi-structured databases and can be modeled as trees. To retrieve such tree-shaped data, several XML query languages have been proposed in the literature: examples include XPath [3] and XQuery [4] .XML queries are typically formed as a twig (small tree) patterns with predicates 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 the XML document with all associated predicates satisfied is a core operation in XML queries processing.

Earlier work on XML twig pattern processing usually decomposes the twig pattern into a set of binary structural relationships, match these relationships, and finally stitch the matches to form the final result. In particular, Al Khalifa et al. [1] proposes two new families of *structural join* algorithms: *Tree-Merge* and *Stack-Tree*. The stack representation of *Stack-Tree* has been used in most of the follow up works. *Structural Joins using B+ trees* [8] builds B+ tree indexes on the start attribute of the joining element sets to skip processing the elements that are guaranteed not to participate in the result. Jain et al. [13] propose a new index called *XR-tree* (XML Region Tree) which is designed specifically for XML data. *XR-tree* is very efficient for skipping both ancestors and descendants during a structural join. The main drawback of all *structural join* algorithms is that they may generate large and possibly unnecessary intermediate results that do not appear in the final result.

To address this problem, Bruno et al [6] propose 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 query. Both algorithms operate in two phases: The first phase computes all the relevant (root-to-leaf) path solutions, while the second phase join-merges these partial solutions to form the answer for the entire twig. *TwigStack* is considered a refinement of *PathStack* because it ensures that, when the query has only Ancestor-Descendant edges, all intermediate solutions produced in the first phase will participate in the final solution. However, *TwigStack* is no longer guaranteed to be I/O and CPU optimal when the query contains a Parent-Child edge between two elements.

To overcome this shortcoming, J. Lu et al [17] present *TwigStackList* algorithm, which is more efficient (generate less intermediate results) than *TwigStack* with the presence of Parent-Child edges in the query. Their technique is to look-ahead some elements the input data streams and cache limited number of them to lists in main memory. [14] proposes a generic *TSGeneric+* algorithm which can utilize available indices, such as *XR-tree* index, to accelerate the running time of *TwigStack*.

Another group of XML path query processing technique is the *navigation-based* approach which computes results by analyzing the input document one tag at a time. Navigational techniques are commonly used in Information Dissemination systems, where many XML path queries have

been preprocessed, and a stream of XML documents is presented as input. *YFilter* [10] is the state-of-the-art algorithm for navigation-based algorithms. *YFilter* combines all path expressions into a single Nondeterministic Finite Automaton (NFA), which enables highly efficient, shared processing for large number of XPath queries. Bruno et al [5] introduced *Index-Filter* algorithm, which generalizes *PathStack* algorithm to answers multiple XML path queries against an XML document.

We summarize the contributions of this paper as follows:

- We develop an efficient holistic path join algorithm, namely *QuickStack*, to match an XML root-to-leaf query. *QuickStack* generalizes *PathStack* algorithm and it effectively skips ancestors and descendants that do not participate in the join.
- We present experimental results on a range of real and synthetic data which shows that *QuickStack* significantly outperforms *TwigStack* for single root-to-leaf queries.
- We present *MQS* algorithm, an extension of *QuickStack* to answer multiple queries and compare it experimentally against *YFilter*. Our results establish that *MQS* is more efficient than *YFilter* when the number of queries is small or the XML document is large. This result mainly owe to the focused processing of *MQS* achieved by the use of indices (especially for queries with high selectivity).

The remainder of the paper is organized as follows. Section 2 is dedicated to some background knowledge and related work on XML. Section 3 describes our proposed algorithm, *QuickStack*, for matching a single query against an XML document. In section 4, we generalize *QuickStack* to answer multiple queries. Section 5 presents our experimental results that compare the proposed algorithms with others. Section 6 discusses some extensions for *QuickStack*. Lastly, section 7 gives the conclusions and directions for future research.

2. Background and related work

2.1. XML numbering scheme

Most existing XML query processing algorithms rely on a positional representation of element nodes, where each element is assigned a triplet of numbers $(start, end, depth)$, based on its position in the data tree. Such a numbering scheme allows determining the structural relationship between two elements in the XML document in constant time. An important property of this numbering scheme is that for any two distinct elements x and y, one of the following 4 cases should hold: (i) the interval of x is completely before or (ii) completely after the interval of y, (iii) the interval of x contains or (iv) is contained in the interval of y. Hence intervals can never intersect partially. Formally, element x is an ancestor of element y iff $x.start < y.start < x.end$. Element x is a parent of element y iff x is an ancestor of y and $x.depth = y.depth - 1$. Figure 1 shows a fictitious XML document with the positional representation for each element. Note that it is easy to determine that element

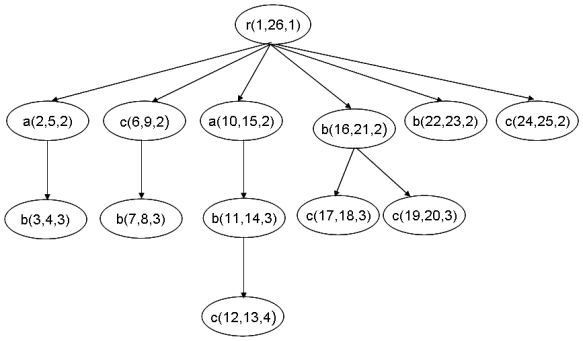

Figure 1. An example XML document

$c(12, 13, 4)$ is a descendant of $a(10, 15, 2)$ without having to examine any of the intermediate nodes.

2.2. Preliminaries

As we are dealing with single root-to-leaf queries, each query is represented by a chain or a unary tree. Let q denote a node in the query. The self-explaining 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 or the parent of q, respectively. *subTree(q)* returns q and all its descendents in the query tree. *q.level* gives the depth of the node in the query tree (note that *level* is different from the *depth* attribute of the element). 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 involved in the algorithm.

Each node q in the query is associated with a data stream of the elements that match the node predicate. Cursor T_q points to the current element of q's stream. T_q can be forwarded to the next element in the stream with the procedure *advance(T_q)*, while the function *eos(T_q)* tests whether T_q has reached the end q's stream. Elements within a stream are encoded using the numbering scheme: *(start, end, depth)* and sorted on 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, node q is associated with a stack S_q. Each element in S_q is a pair: (an element from T_q, a pointer to the element's lowest ancestor in the parent stack). The operations over stacks are the usual *empty*, *pop* and *push* operations.

Initially, all the stacks are empty and all the cursors point to the beginning of the data streams. During the evaluation of the query, the cursors advance sequentially and the stacks

cache the elements that may participate in the solution. An important property of the stacks is that every element in the stack is a descendant of all the elements below it.

2.3. PathStack algorithm

As *QuickStack* is partially inspired by *PathStack* algorithm, we briefly review how *PathStack* algorithm works through an example. Afterward, we explain *QuickStack* algorithm and demonstrate how it solves a lot of the performance problems related to *PathStack*.

Example: Consider the path query //a//b//c on the XML document of Figure 1. The streams of the XML elements associated with each query node are visualized in Figure 2. A subscript is added to each element in the order of their start values for easy reference.

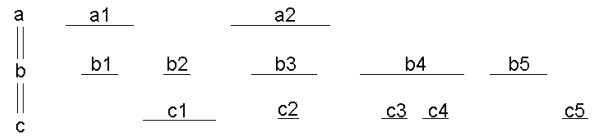

Figure 2. Query a//b//c and the data streams of the XML document in Figure 1

Initially, the three cursors are at (a_1, b_1, c_1). a_1 is pushed first in the stack because it is the element with the least *start* value, and the cursor advances to a_2. In the next iteration, b_1 is pushed in S_b. Then c_1 is pushed and it popes out the elements a_1 and b_1 because c_1 is after them, i.e. $c_1.start > a_1.end$ and $c_1.start > b_1.end$. Since c_1 is a leaf node in the query tree, the algorithm calls *showSolutions*, which will not find any solution because the stacks are empty at this moment. After several iterations, *PathStack* pushes c_2 in S_c and it outputs the solution (a_2, b_3, c_2). Later on, *PathStack* repeatedly pushes and popes all the remaining elements until it reaches c_5. At that time, the algorithm terminates because the stream of the leaf node ends.

3. QuickStack algorithm

In this section, we describe our proposed algorithm, *QuickStack*, for finding all matches of a single root-to-leaf path query against an XML document. The algorithm effectively evaluates the query by skipping ancestors and descendants that do not participate in the result.

We first describe a synthetic dataset that is used to illustrate the algorithm. This dataset contains information about bookstores and the books they have. We also use this bookstores dataset in the experiments.

the bookstores dataset: The DTD for this dataset is shown in Figure 3. Bookstores are numbered sequentially according to their appearance in the XML file and are randomly distributed among 7 different states (the *state* attribute). Books are given sequential titles (book1, book2, ...), so that each title is unique. The books prices vary from 10 to 100.

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

Figure 3. DTD of the bookstores dataset

Now we explain in details how *QuickStack* works. The algorithm is outlined in Figure 4.

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:       showSolutions(S_{q_min})
18:       pop(S_{q_min})
```

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 Tq in Sq, assign its pointer to point to the top of $S_{parent(q)}$.
2: advance(Tq).

Procedure showSolution(Sq)
output all the solutions contained in the stacks.

Figure 4. QuickStack algorithm

In lines 3-4, the algorithm identifies the nodes q_{min} and q_{max} that has the minimal and maximal *start* position among the current cursors. All the elements that are before q_{min} are removed from their stacks (Lines 5-6). Lines 7-8 terminate the algorithm if the condition of *QuickEnd* is satisfied. *QuickStack* chooses between calling *skipAncestors* or *skipDescendants* depending on the height of q_{min} and q_{max} in the query tree (lines 9-13): If q_{max} is lower in the tree, i.e. closer to the leaf, the algorithm calls *skipAncestors* on q_{max}'s parent node. Otherwise, it calles *skipDescendants* on q_{min} only if its parent stack is empty (line 12).

The reason for this additional condition is that if there are some elements in q_{min}'s parent stack, then these elements are qualified parents of q_{min} (otherwise, they would have been popped out of the stack when q_{min} was pushed). In this case, q_{min} can not be skipped because it could be part of a solution. Both *skipAncestors* and *skipDescendants* return true if a skipping occurred.

If no skipping happened, q_{min} is pushed in its stack because probably it will participate in the final result. Lines 16-18 check if the q_{min} is a leaf node, then *showSolutions* is called to output the solutions from the stacks.

QuickEnd function: The first feature that distinguishes *QuickStack* from other algorithms is the *QuickEnd* function. The idea is that *QuickStack* terminates whenever the stream of any node in the tree ends if the stack of this node is empty. To prove the correctness of *QuickEnd* function, notice that the nodes are pushed in the stacks in an increasing order of their *start* position. So, if a node's stack is empty and its stream has finished, it is guaranteed that this node will not participate in any future solutions. Thus, the algorithm can terminate safely without missing any answer.

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 5. QuickEnd function

Example 2: Consider the query //a//b//c and the element sets shown in Figure 6.

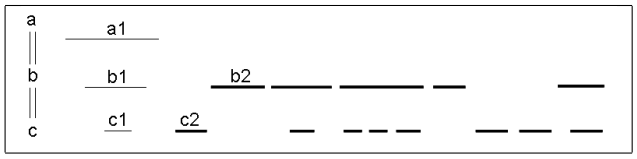

Figure 6. the early termination of QuickStack

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 in S_c and it empties all the stacks. At this point, *QuickEnd* condition is satisfied because a's stream ends and its stack is empty. Therefore, the algorithm terminates without having to examine all the remaining elements (the thick line segments).

Referring to our bookstore dataset, the query /bookstore [num=1] /book/title returns the titles of all the books contained in the first store that appear in the file. Using the *end* function of *TwigStack*, the algorithm have to examine all the *book* and *title* elements before giving the result, while *QuickEnd* stops the algorithm immediately after the books of the first store are processed.

Skipping Ancestors: The key idea for skipping ancestors is that the element that does not contain an element of the child node, could not participate in the solution. So there is no need to process this element and all its parents.

Function skipAncestors(q,$childStart$): boolean

1: skipping=false
2: while($\neg$ eos(T_q) $\wedge\ T_q.end < childStart$)
3: skipping=true
4: advance(T_q)
5: if(isRoot(q))
6: return skipping
7: else
8: return skipAncestors(parent(q),$max(T_q.start, childStart)$)
$\vee$ skipping

Figure 7. skipAncestors function

skipAncestors is applied on a query node q with the *start* value of the current element from *child(q)*'s stream. The function is called recursively on all the parents of q (up to the root) and it returns true if a skipping happened at any level of the query tree. In line 2, all the elements that are before the child element, i.e. end before *childStart*, are skipped. In line 8, the function is called on q's parent and the second parameter gets the maximum of the *start* of the current element in q's stream and *childStart*. The reason for taking the maximum is that the element with the bigger start is the first element that could participate in a solution. Therefore, skipping to that element allows more skipping to occur, while maintaining the correctness of the results.

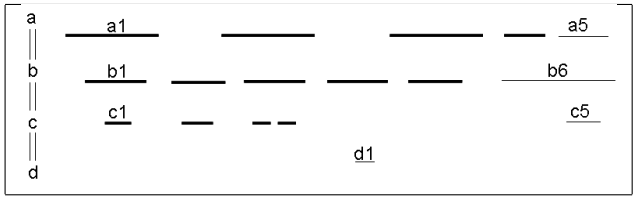

Figure 8. Skipping ancestor elements

In Figure 8, d_1 is q_{max} and *skipAncestors* is called on c_1 to skip all the elements that end before d_1.*start*. Next, *skipAncestors* is called on nodes b and a to skip the elements before c_5 (because c_5.*start* $> d_1$.*start*). As a result, the cursors advance directly to c_5, b_6, a_5 without the need to process any of the elements before.

As another example, assume we have the query: /bookstore/book/chapter [title="XML DTD"] on the bookstores dataset. The query asks for all the books that has a chapter about XML DTD. Actually, a very small portion of the books in the file will satisfy this query, thus *skipAncestors* will skip a lot of *book* and *bookstore* elements without processing.

Skipping Descendants: *skipDescendants* applies similar technique as *skipAncestors*: if an element does not have a parent, i.e. is not contained in an element form the parent

node's stream, this element and all its descendants will not participate in a solution.

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 9. skipDescendants function

The function is called on node q, and the *start* value of $T_{parent(q)}$. Like *skipAncestors*, the function returns true if a 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 called on the child node of q [1].

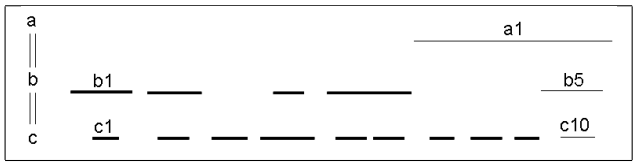

Figure 10. Skipping descendant elements

In the example depicted in Figure 10, *skipDescendants* is called on b_1 (q_{min}), and it forwards the cursor directly to the element b_5, then *skipDescendants* is called again on c_1, and c's cursor is forwarded to c_{10}, the first node that starts after b_5.*start*.

Consider again the bookstores dataset and the query: /bookstore [num=10 or num=100] /book/title. *skipDescendants* will skip all the *book* and *title* elements that are before *bookstore10*, and then will skip the elements that appear between *bookstore10* and *bookstore100* (because they will not have a parent, and *skipDescendants* will be activated). Finally, after processing the elements of *bookstore100*, *QuickEnd* will terminate the algorithm.

4. Multiquery QuickStack (MQS)

In this section, we consider the general scenario of matching multiple path queries against an XML document. This scenario usually occurs in the XML Filtering Systems, where a continuously arriving streams of XML documents are passed through a filtering engine to match stored queries representing users' interests.

[1]Here, there is not need to take the maximum as in *skipAncestors* because T_q.*start* is always bigger than *parentStart*

The problem definition is as follows: *Given an XML document D and a set of path 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.*

When evaluating multiple queries against an XML document, we have several options to consider:

1. Option 1 (Naive way): evaluate each query separately, i.e. for each query: parse the document to build its index, and then execute the algorithm on that index.
2. Option 2: Parse the document once, but while parsing, concurrently build the indices for all the queries.
3. Option 3: construct a *generalized query* (defined later) that represents all the queries. Then, when the document arrives, parse the document and build the index for the *generalized query*. Afterward, run *QuickStack* for each query on this common index and check the predicates during the execution of the algorithm.
4. Option 4: Build the index for the *generalized query* as in option 3, but then for each query: run *QuickStack* on the query's own index after extracting it from the common index.
5. Option 5: develop an algorithm that processes several queries simultaneously to eliminate redundant processing while answering the queries.

A *generalized query* for a set of queries is the query whose index contains the elements of all the queries' indices. It can be represented by a tree that contains all the nodes form the path 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,
and
Q3=/bookstore[num>10 and num<20]/book[title="XML tutorial" and price <20]/chapter/title.
The *generalized query* from these three queries is shown in Figure 11. Note that the predicate on the title of the book in Q3 does not appear in the *generalized query* because Q1 and Q2 do not have this predicate.

Clearly, the naive way is very wasteful because, as we will see in the experiments section, parsing the document is a very expensive operation. Option 3 is also not a good idea, because running *QuickStack* on the index of the *generalized query* restricts its ability to skip the elements, and thus hurts performance. For instance, when we execute *QuickStack* on Q3's own index, 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 index in Figure 11, the stream of *book* node

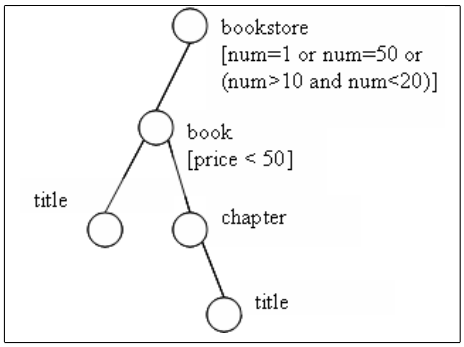

Figure 11. the generalized query from Q1, Q2, and Q3

will be much longer than it should be, and the algorithm will spend a lot of time processing extra elements that can not be part of Q3's result. The experiments showed that option 4 is more efficient than option 2, thus **it will be considered in our experiments**.

Option 5 considers designing an algorithm similar to *Index-Filter* [5], but which relies on *QuickStack* instead of *PathStack*. We plan to explore this option further as part of our future work.

In some situations, when the input documents are static and we receive batches of input queries to process, we can parse the document offline to build and materialize the *generalized index* over the document. The query tree of the *generalized index* represents the whole structure of the file and each node is associated with all the elements from the document that match it. In this case, We can not impose any predicate on the tree nodes because we do not know in advance the queries that will be asked. Although this approach can save us the time of parsing the document at run time, extracting query indices from this *generalized index* can be more expensive that extracting the indices from the index of the *generalized query*.

In our implementation, when we have Parent-Child edge in the query tree, we check the parent of the element related to the child node before adding it to the index to see if it matches the parent node. For example, instead of adding every *title* element that appear in the bookstore document to both leaf nodes of the tree in Figure 11, we check its direct parent in the document: if it is a *book* element, we add it to the child of node *book*, otherwise, we add it to the child of node *chapter*. This extra check during building the index avoids having irrelevant elements in the streams, hence speed up the execution of the algorithm.

5. Experiments

In section 5.1, we present experimental results on the processing of a single query using the three twig pattern matching algorithms, namely *QuickStack*, *TwigStack* and *PathStack*. We implemented all algorithms in JAVA using JDK 1.5, sharing as much code and data structures as possible for a fair comparison. In section 5.2, we study multiple queries processing and compare the performance of our approach against *YFilter* (version 1.0) [19]. *YFilter* system is developed at the University of California at Berkeley and is implemented in Java using J2SE1.4.x.

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.

5.1. QuickStack vs TwigStack / PathStack for single query

We conducted our experiments on both synthetic and real-world data. We used the synthetic bookstore dataset that we described earlier, and the real-world DBLP [16] and the NASA [18] datasets.

5.1.1 Bookstores

File size	121MB
Number of elements in the document	
Number of *bookstore* elements	1000
Number of *book* elements	147,680
Number of chapter elements	1,846,217
Total number of elements	6,132,372

Table 1. bookstores file characteristics

Each bookstore has between 50 and 250 books and each book contains 5 to 20 chapters. We used the single root-to-leaf queries in Table 2 over the bookstores data. We choose different queries so that we can give a comprehensive comparison between the three algorithms.

Query	Build Index	PathStack	TwigStack	QuickStack
Q1	2558	924	77	10
Q2	3464	5874	422	78
Q3	3456	13310	810	122
Q4	3560	10645	792	50
Q5	3539	10154	630	13
Q6	3500	5374	430	222
Q7	3125	13541	680	1
Q8	3077	12936	656	1

Table 3. the execution time (in ms) for the three algorithms and the time to build the index: bookstores

Table 3 gives the execution times of the three algorithms in milliseconds. The first column is the query asked and the second column is the time taken to parse the XML file and

Query	XPath expression
Q1	/*/bookstore [num=1] /book/price
Q2	//bookstore [num > 100 and num < 115] /book/ chapter [title="chapter11"] /num_of_pages
Q3	//bookstore [num = 10 or num =120] /book/chapter/title
Q4	//bookstore [num = 200] /book [price ≥ 20 and price ≤30] /chapter/title
Q5	//bookstore/book [title="book6985"] /chapter/title
Q6	//bookstore [@state="PA"]/book [price < 30]/chapter [title="chapter4"] /num_of_pages
Q7	//library//book/chapter/title
Q8	//bookstore [@state="CA"] /book/chapter/num_of_pages

Table 2. Queries over bookstores dataset

Query	XPath expression
Q1	//inproceedings [author="Michael Stonebraker" and year=2003] /title
Q2	//inproceedings [title="Ratio Rules: A New Paradigm for Fast, Quantifiable Data Mining."] /author
Q3	//inproceedings [author="Christos Faloutsos" or author="Rajeev Agrawal" or author="Soumen Chakrabarti" and year=2000] /author
Q4	//inproceedings [title="Spatial Join Selectivity Using Power Laws."] /cite
Q5	//article [author="Michael Stonebraker"] /cite

Table 4. Queries over DBLP dataset

build the index. The last three columns shows the execution time of *PathStack*, *TwigStack* and *QuickStack* once the index has been built. Remember that the time to build the index is the same among the three algorithms.

The table shows that *QuickStack* constantly outperforms *PathStack* and *TwigStack*. For instance, for Q8, *QuickStack* reduces the time for query matching by more than 99% and the overall query processing time (including the time to build the index) by about 18%.

Figure 12 shows the execution time after excluding the time to build the index. We omit *PathStack* from all the figures because it is always slower than both *TwigStack* and *QuickStack*.

It can be seen that the higher the selectivity of the query (the fewer the matches of the query in the document), the bigger the difference in the execution time between *QuickStack* and *TwigStack*. For instance, *QuickStack* performs about 50 times faster for query Q5 because Q5 is very selective (remember that the title of each book is unique in the file). On the other hand, the difference is not that significant for Q6 because the query returns a lot of results, thus skipping the elements becomes less effective.

Both Q7 and Q8 do not have any match in the document because *library* element (Q7) does not exist and *CA* state (Q8) is not among the states included in the file. To detect this, *TwigStack* has to go through all the elements in the lists, while *QuickStack* detects it right away (query evaluation time is 1 ms), resulting in this huge difference in execution time between the two algorithms.

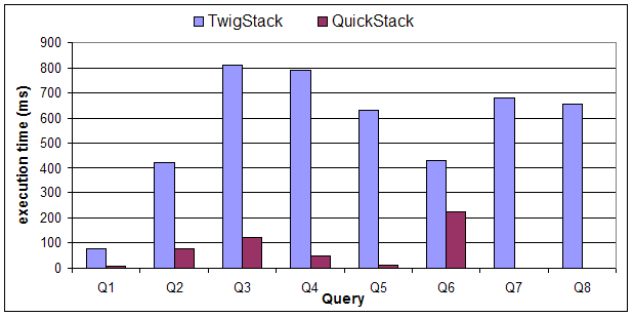

Figure 12. Execution time on Bookstores after building the index

5.1.2 DBLP

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 210MB and 4,884,836 elements in total. The file contains information about 1,075,088 authors, 298,413 inproceedings and 173,630 articles. It has a maximum depth of 6 and an average depth of about 2.9.

The execution times for the queries in Table 4 appear in Table 5 and Figure 13.

Query	Build Index	PathStack	TwigStack	QuickStack
Q1	4692	1266	77	13
Q2	4596	2168	128	16
Q3	5015	1804	138	19
Q4	4117	713	53	13
Q5	4036	576	48	17

Table 5. the execution time (in ms) for the three algorithms and the time to build the index: DBLP

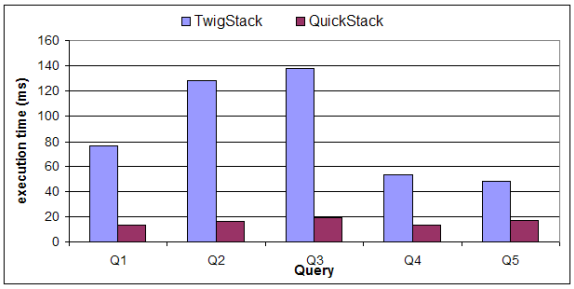

Figure 13. Execution time on DBLP after building the index

5.1.3 NASA

This dataset is converted from 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. Table 7 and Figure 14 show the execution time of the algorithms for the queries in Table 6.

Query	Build Index	PathStack	TwigStack	QuickStack
Q1	410	569	32	1
Q2	373	646	43	11
Q3	360	516	31	11
Q4	394	790	64	16
Q5	377	635	37	1

Table 7. the execution time (in ms) for the three algorithms and the time to build the index: NASA

5.2 A comparison with YFilter for multiple queries

In this section, we compare the performance of *QuickStack*, *TwigStack*, and *YFilter*, for varying number of input queries. We use a collection of queries similar to the ones in Table 2. We will refer to *QuickStack* and *TwigStack* for multiple queries as *MQS* and *MTS*, respectively.

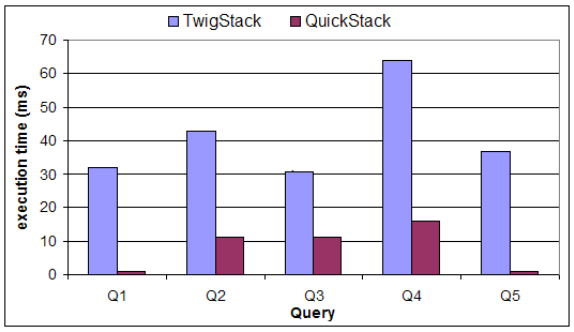

Figure 14. Execution time on NASA after building the index

From Figure 16, we can see that for a small number of queries, the performance of *MTS* and *MQS* is almost similar since the time to build the index dominates the cost. However, as the number of input queries increases, *MQS* outperforms *MTS*.

The figure also shows that *MQS* outperforms *YFilter* when the number of queries is small. For instance, for 20 queries, *MQS* reduces execution time by 64% compared with *YFilter*. The reason is that *MQS* can exploit indices built over the document to avoid processing large portions of the input that will not participate in a match. However, these gains tend to diminish as we increase the number of queries because *YFilter* has the advantage that its performance is independent on the number of input queries. These results suggest using a hybrid approach that switches automatically between *MQS* and *YFilter* based on the number of input queries and the size of the document.

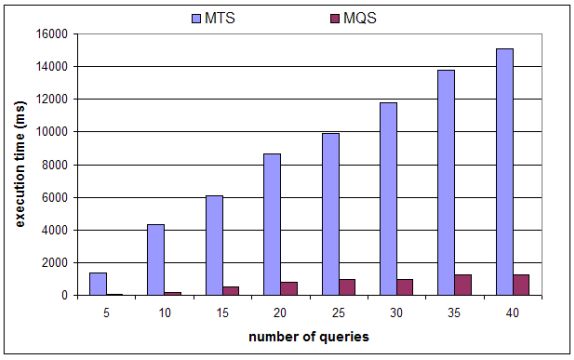

Figure 15. Time to execute the queries after building the index

Query	XPath expression
Q1	//dataset [title="Astrographic Catalogue"] /reference//author/lastName
Q2	//dataset//fields/field [name="DE"] /definition
Q3	//dataset//reference/source//author [lastName="Mermilliod"] /initial
Q4	//dataset//fields/field [name="L"] /definition/footnote/para
Q5	//dataset//fields/field [name="weird name"] /definition

Table 6. Queries over NASA dataset

Number of Queries	YFilter	Index Time	MTS		MQS	
			evaluate	total	evaluate	total
1	46887	2643	80	2723	12	2655
5	46887	5479	1350	6829	35	5514
10	46887	8605	4316	12921	150	8755
15	46887	12140	6106	18246	497	12637
20	46887	16096	8632	24728	776	16872
25	46887	16403	9904	26307	961	17364
30	46887	20285	11760	32045	952	21237
35	46887	23349	13769	37118	1261	24610
40	46887	30193	15090	45283	1270	31463

Table 8. the execution time (in ms) of *MQS*, *MTS* and *YFilter* with different number of queries

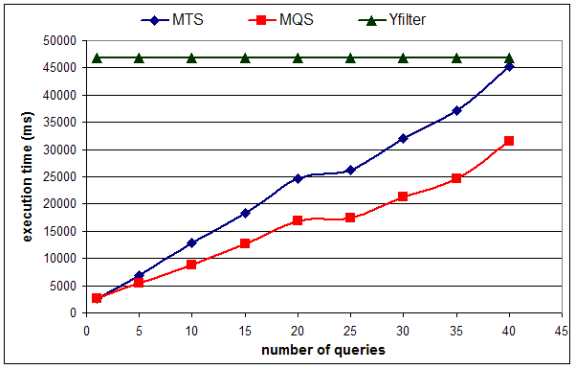

Figure 16. Total time to evaluate multiple queries

6 Discussion

- *QuickStack* can be easily generalized to compute the answer for any twig pattern. First the twig is decomposed into multiple root-to-leaf path patterns and then *QuickStack* is applied on each individual path. The partial solutions are then merged to compute the final answer of the query. Since experiments showed that *QuickStack* is significantly faster than *TwigStack* for single path queries, it could result in a faster execution for more complex queries as well. However, in this case, *QuickStack* is not guaranteed to be optimal, because it may generate intermediate results that are not part of the final result.
- The document index can be augmented with *XR-tree* [13] to accelerate the performance of *QuickStack* when the element streams are very long. So instead of scanning the elements sequentially when calling *skipAncestors* or *skipDescendants* functions, *XR-tree* helps pointing directly to the desired element.

7 Conclusions and Future work

In this paper we presented *QuickStack*, an enhanced holistic join algorithm for matching XML query patterns. *QuickStack* can most effectively avoid unnecessary elements by skipping both ancestors and descendants that do not have a match in the document. Experimental results showed that our method is much more efficient than *TwigStack* for queries with single root-to-leaf paths.

Regarding our future work, we will try to modify the algorithm to be more suitable for complex XML queries. In particular, we plan to extend the *TwigStackList* [17] algorithm and introduce our skipping techniques, presented in this work.

Another avenue for future work, encouraged by the experimental results, is to design a new powerful algorithm specifically designed to answer multiple queries simultaneously. Exploiting the commonalities among queries would allow us to create a more efficient and scalable system.

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 the 18th IEEE International Conference on Data Engineering*, 2002.

[2] M. Altinel and M. J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proceedings of the 26th VLDB Conference*, 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 the 19th IEEE International Conference on Data Engineering*, 2003.

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

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

[8] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient Structural Joins on Indexed XML Documents. In *Proceedings of the 28th VLDB Conference*, 2002.

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

[10] Y. Diao and M. J. Franklin. High-Performance XML Filtering: An Overview of YFilter. In *Proceedings of the 19th IEEE International Conference on Data Engineering*, 2003.

[11] Paul F. Dietz. Maintaining Order in a Linked List. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, 1982.

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

[13] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-Tree: Indexing XML Data for Efficient Structural Joins. In *Proceedings of the 19th IEEE International Conference on Data Engineering*, 2003.

[14] H. Jiang, H. Lu, W. Wang, and J. X. Yu. Holistic Twig Joins on Indexed XML Documents. In *Proceedings of the 29th VLDB Conference*, 2003.

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

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

[17] 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 Conference*, 2004.

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

[19] YFilter 1.0 release. Available from: http://yfilter.cs.umass.edu.