Holistic Twig Joins: Optimal XML Pattern Matching

Nicolas Bruno
Columbia University
nicolas@cs.columbia.edu

Nick Koudas
AT&T Labs–Research
koudas@research.att.com

Divesh Srivastava
AT&T Labs–Research
divesh@research.att.com

Abstract

XML employs a tree-structured data model, and, naturally, XML queries specify patterns of selection predicates on multiple elements related by a tree structure. Finding all occurrences of such a twig pattern in an XML database is a core operation for XML query processing. Prior work has typically decomposed the twig pattern into binary structural (parent-child and ancestor-descendant) relationships, and twig matching is achieved by: (i) using structural join algorithms to match the binary relationships against the XML database, and (ii) stitching together these basic matches. A limitation of this approach for matching twig patterns is that intermediate result sizes can get large, even when the input and output sizes are more manageable.

In this paper, we propose a novel holistic twig join algorithm, `TwigStack`, for matching an XML query twig pattern. Our technique uses a chain of linked stacks to compactly represent partial results to root-to-leaf query paths, which are then composed to obtain matches for the twig pattern. When the twig pattern uses only ancestor-descendant relationships between elements, `TwigStack` is I/O and CPU optimal among all sequential algorithms that read the entire input: it is linear in the sum of sizes of the input lists and the final result list, but independent of the sizes of intermediate results. We then show how to use (a modification of) B-trees, along with `TwigStack`, to match query twig patterns in sub-linear time. Finally, we complement our analysis with experimental results on a range of real and synthetic data, and query twig patterns.

1 Introduction

XML employs a tree-structured model for representing data. Queries in XML query languages (see, e.g., [7, 4, 2]) typically specify patterns of selection predicates on multiple elements that have some specified tree structured relationships. For example, the XQuery expression:

```
book[title = 'XML']//author[fn = 'jane' AND ln = 'doe']
```

matches `author` elements that (i) have a child subelement `fn` with content `jane`, (ii) have a child subelement `ln` with content `doe`, and (iii) are descendants of `book` elements that have a child `title` subelement with content `XML`. This expression can be represented as a node-labeled twig (or small tree) pattern with elements and string values as node labels.

Finding all occurrences of a twig pattern in a database is a core operation in XML query processing, both in relational implementations of XML databases, and in native XML databases. Prior work (see, for example, [11, 23, 16, 18, 17, 26, 1]) has typically decomposed the twig pattern into a set of binary (parent-child and ancestor-descendant) relationships between pairs of nodes, e.g., the parent-child relationships (`book`, `title`) and (`author`, `fn`), and the ancestor-descendant relationship (`book`, `author`). The query twig pattern can then be matched by (i) matching each of the binary structural relationships against the XML database, and (ii) "stitching" together these basic matches.

For solving the first sub-problem of matching binary structural relationships, Zhang et al. [26] proposed a variation of the traditional merge join algorithm, the multi-predicate merge join (MPMGJN) algorithm, based on the `(DocId, LeftPos:RightPos, LevelNum)` representation of positions of XML elements and string values (see Section 2.3 for details about this representation). Their results showed that the MPMGJN algorithm could outperform standard RDBMS join algorithms by more than an order of magnitude. More recently, Al-Khalifa et al. [1] took advantage of the same representation of positions of XML elements to devise I/O and CPU optimal join algorithms to match binary structural relationships on an XML database.

The second sub-problem of stitching together the basic matches obtained using binary "structural" joins requires identifying a good join ordering in a cost-based manner, taking selectivities and intermediate result size estimates into account. In this paper, we show that a basic limitation of this (traditional) approach for matching query twig patterns is that intermediate result sizes can get very large, even when the input and final result sizes are much more manageable. As a result, we seek a better solution to the problem of matching query twig patterns efficiently.

In this paper, we propose a novel *holistic twig join* approach for matching XML query twig patterns, wherein no large intermediate results are created. Our technique uses the `(DocId, LeftPos : RightPos, LevelNum)` representation of positions of XML elements and string values (that succinctly captures structural relationships between nodes in the XML database). It also uses a chain of linked stacks to compactly represent partial results to individual query root-to-leaf paths, which are then composed to obtain matches to the query twig pattern.

Since a great deal of XML data is expected to be stored in relational database systems (all the major DBMS vendors including Oracle, IBM and Microsoft are providing system support for XML data), our study provides evidence that RDBMS systems need to augment their suite of query processing strategies to include holistic twig joins for efficient XML query processing. Our study is equally relevant for native XML query engines, since holistic twig joins are an efficient set-at-a-time strategy for matching XML query patterns, in contrast to the node-at-a-time approach of using tree traversals.

1.1 Outline and Contributions

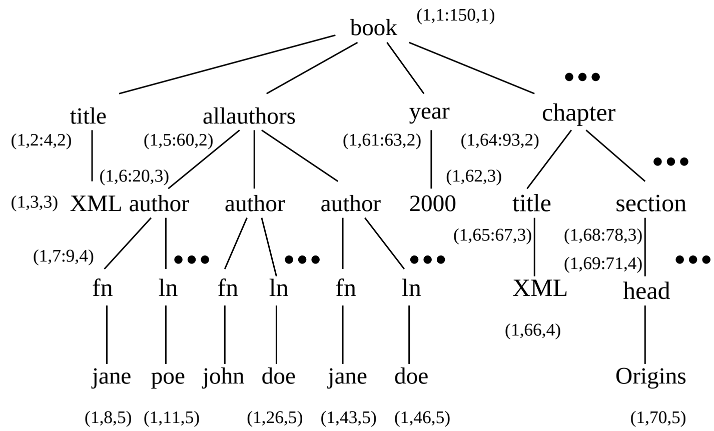
We begin by presenting background material (data model, query twig patterns, and positional representations of XML elements) in Section 2. Our main contributions are:

- We develop two families of *holistic path join* algorithms in Section 3 to match XML query root-to-leaf paths efficiently. The first, `PathStack`, generalizes the Stack-Tree-Desc binary structural join algorithm of [1], while the second, `PathMPMJ`, generalizes the MPMGJN binary join algorithm of [26]. We analyze `PathStack` and show that it is I/O and CPU optimal among all sequential algorithms that read the entire input, and has worst-case complexities linear in the sum of input and output sizes *but independent of the sizes of intermediate results.*

- We then develop `TwigStack` in Section 4, a *holistic twig join* algorithm that (i) refines `PathStack` to ensure that results computed for one root-to-leaf path of a twig pattern are likely to have matching results in other paths of the twig pattern, and (ii) merges results for the different root-to-leaf paths in the query twig pattern, to compute the desired output. When the query twig uses only ancestor-descendant relationships between elements, we analytically demonstrate that `TwigStack` is I/O and CPU optimal among all sequential algorithms that read the entire input.

```
<book>
   <title> XML </title>
   <allauthors>
      <author>
         <fn>jane</ > <ln>poe</ >
      </author>
      <author>
         <fn>john</ > <ln>doe</ >
      </author>
      <author>
         <fn>jane</ > <ln>doe</ >
      </author>
   </allauthors>
   <year> 2000 </year>
   <chapter>
      <title> XML </title>
      <section>
         <head> Origins </head> ...
      </section> ...
   </chapter> ...
</book>
```

(a)



(b)

Figure 1: (a) A sample XML database fragment, (b) Tree representation

- Finally, in Section 5 we present experimental results on a range of real and synthetic data, and query twig patterns, to complement our analytical results:
  - We show the substantial performance benefits of using holistic twig joins over binary structural joins (for arbitrary join orders).
  - We show that `PathStack` is significantly faster than `PathMPMJ`among holistic path join algorithms. This validates the analytical results demonstrating the I/O and CPU optimality of `PathStack`.
  - For the case of twig patterns, we show that the use of `TwigStack` is better (both in time and space) than the independent use of `PathStack` on each root-to-leaf path, even when the twig pattern contains parent-child structural relationships.
  - We show how to use a modification of B-trees, denoted XB-trees, along with `TwigStack`, to perform matching of query twig patterns in *sub-linear time*.

We describe related work in Section 6, and conclude by discussing ongoing and future work in Section 7.

2 Background

2.1 Data Model and Query Twig Patterns

An XML database is a forest of rooted, ordered, labeled trees, each node corresponding to an element or a value, and the edges representing (direct) element-subelement or element-value relationships. Node labels consist of a set of (attribute, value) pairs, which suffices to model tags, `ID`s, `IDREF`s, etc. The ordering of sibling nodes implicitly defines a total order on the nodes in a tree, obtained by a preorder traversal of the tree nodes. For the sample XML document of Figure 1(a), its tree representation is shown in Figure 1(b). (The utility of the numbers associated with the tree nodes will be explained in Section 2.3.)

Queries in XML query languages like XQuery [2], Quilt [4] and XML-QL [7] make use of (node labeled) twig patterns for matching relevant portions of data in the XML database. The twig pattern node labels

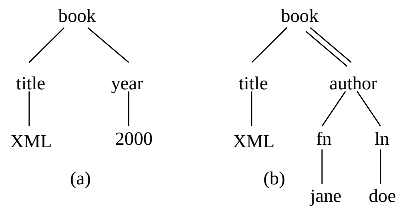


Figure 2: Query twig patterns

include element tags, attribute-value comparisons, and string values, and the query twig pattern edges are either parent-child edges (depicted using a single line) or ancestor-descendant edges (depicted using a double line). For example, the XQuery expression:

```
book[title = 'XML' AND year = '2000']
```

which matches `book` elements that (i) have a child `title` subelement with content `XML`, and (ii) have a child `year` subelement with content `2000`, can be represented as the twig pattern in Figure 2(a). Only parent-child edges are used in this case. Similarly, the XQuery expression in the introduction can be represented as the twig pattern in Figure 2(b). Note that an ancestor-descendant edge is used between the `book` element and the `author` element.

In general, at each node in the query twig pattern, there is a *node predicate* on the attributes (e.g., tag, content) of the node in question. For the purposes of this paper, exactly what is permitted in this predicate is not material. Similarly, the physical representation of the nodes in the XML database is not relevant to the results in this paper. It suffices for our purposes that there be efficient access mechanisms (such as index structures) to identify the nodes in the XML database that satisfy any given node predicate q, and return a stream of matches T_q.

2.2 Twig Pattern Matching

Given a query twig pattern Q and an XML database D, a *match* of Q in D is identified by a mapping from nodes in Q to nodes in D, such that: (i) query node predicates are satisfied by the corresponding database nodes (the images under the mapping), and (ii) the structural (parent-child and ancestor-descendant) relationships between query nodes are satisfied by the corresponding database nodes. The *answer* to query Q with n nodes can be represented as an n-ary relation where each tuple $(d_1, \ldots, d_n)$ consists of the database nodes that identify a distinct match of Q in D.

Finding all matches of a query twig pattern in an XML database is a core operation in XML query processing, both in relational implementations of XML databases, and in native XML databases. In this paper, we consider the twig pattern matching problem:

> Given a query twig pattern Q, and an XML database D that has index structures to identify database nodes that satisfy each of Q's node predicates, compute the answer to Q on D.

Consider, for example, the query twig pattern in Figure 2(a), and the database tree in Figure 1. This query twig pattern has one match in the data tree that maps the nodes in the query to the root of the data and its first and third subtrees.

2.3 Representing Positions of Elements and String Values in an XML Database

The key to an efficient, uniform mechanism for set-at-a-time (join-based) matching of query twig patterns is a positional representation of occurrences of XML elements and string values in the XML database (see, e.g., [5, 6, 26]), which extends the classic inverted index data structure in information retrieval [21].

We represent the position of a string occurrence in the XML database as a 3-tuple `(DocId, LeftPos, LevelNum)`, and the position of an element occurrence as a 3-tuple `(DocId, LeftPos:RightPos, LevelNum)`, where (i) `DocId` is the identifier of the document; (ii) `LeftPos` and `RightPos` can be generated by counting word numbers from the beginning of the document `DocId` until the start and the end of the element, respectively; and (iii) `LevelNum` is the nesting depth of the element (or string value) in the document. Figure 1(b) shows 3-tuples associated with some tree nodes, based on this representation (the `DocId` for all nodes is chosen to be one).

Structural relationships between tree nodes whose positions are recorded in this fashion can be determined easily: (i) *ancestor-descendant*: a tree node n_2 whose position in the XML database is encoded as $(D_2, L_2 : R_2, N_2)$ is a descendant of a tree node n_1 whose position is encoded as $(D_1, L_1 : R_1, N_1)$ iff $D_1 = D_2, L_1 < L_2$, and $R_2 < R_1$[1]; (ii) *parent-child*: a tree node n_2 whose position in the XML database is encoded as $(D_2, L_2 : R_2, N_2)$ is a child of a tree node n_1 whose position is encoded as $(D_1, L_1 : R_1, N_1)$ iff $D_1 = D_2, L_1 < L_2, R_2 < R_1$, and $N_1 + 1 = N_2$. For example, in Figure 1(b), the author node with position $(1, 6 : 20, 3)$ is a descendant of the book node with position $(1, 1 : 150, 1)$, and the string "`jane`" with position $(1, 8, 5)$ is a child of the `author` node with position $(1, 7 : 9, 4)$.

A key point worth noting about this representation of node positions in the XML data tree is that checking an ancestor-descendant relationship is as simple as checking a parent-child relationship (we can check for an ancestor-descendant structural relationship without knowledge of the intermediate nodes on the path). Also, this representation of positions of nodes allow for checking order (e.g., node n_2 follows node n_1) and structural proximity (e.g., node n_2 is a descendant within 3 levels of n_1) relationships.

3 Holistic Path Join Algorithms

3.1 Notation

Let q (with or without subscripts) denote twig patterns, as well as (interchangeably) the root node of the twig pattern. In our algorithms, we make use of the following twig node operations: `isLeaf`: $Node \rightarrow Bool$, `isRoot`: $Node \rightarrow Bool$, `parent`: $Node \rightarrow Node$, `children`: $Node \rightarrow \{Node\}$, and `subtreeNodes`: $Node \rightarrow \{Node\}$. Path queries have only one child per node, otherwise `children`(q) returns the set of children nodes of q. The result of `subtreeNodes`(q) is the node q and all its descendants.

Associated with each node q in a query twig pattern there is a *stream* T_q. The stream contains the positional representations of the database nodes that match the node predicate at the twig pattern node q (possibly obtained using an efficient access mechanism, such as an index structure). The nodes in the stream are sorted by their `(DocId, LeftPos)` values. The operations over streams are: `eof`, `advance`, `next`, `nextL`, and `nextR`. The last two operations return the `LeftPos` and `RightPos` coordinates in the positional representation of the next element in the stream, respectively.

In our stack-based algorithms, `PathStack` and `TwigStack`, we also associate with each query node q a *stack* S_q. Each data node in the stack consists of a pair: (positional representation of a node from T_q, pointer to a node in $S_{\texttt{parent}(q)}$). The operations over stacks are: `empty`, `pop`, `push`, `topL`, and `topR`. The last two

[1]For leaf strings, the `RightPos` value is the same as the `LeftPos` value.

operations return the `LeftPos` and `RightPos` coordinates in the positional representation of the top element in the stack, respectively. At *every* point during the computation, (i) the nodes in stack S_q (from bottom to top) are guaranteed to lie on a root-to-leaf path in the XML database, and (ii) the set of stacks contain a *compact encoding* of partial and total answers to the query twig pattern, which can represent in linear space a potentially exponential (in the number of query nodes) number of answers to the query twig pattern, as illustrated below.

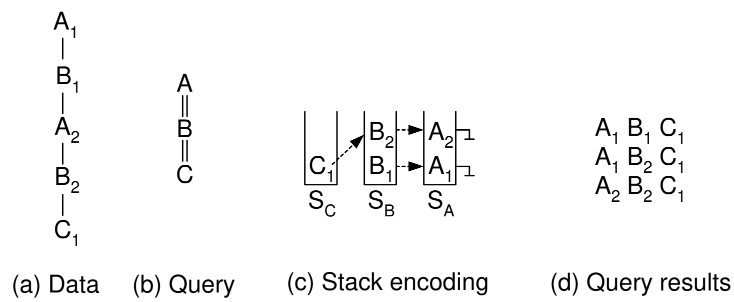


Figure 3: Compact encoding of answers using stacks.

Example 3.1 *Figure 3 illustrates the stack encoding of answers to a path query for a sample data set. The answer* $[A_2, B_2, C_1]$ *is encoded since* C_1 *points to* B_2*, and* B_2 *points to* A_2*. Since* A_1 *is below* A_2 *on the stack* S_A*,* $[A_1, B_2, C_1]$ *is also an answer. Finally, since* B_1 *is below* B_2 *on the stack* S_B*, and* B_1 *points to* A_1*,* $[A_1, B_1, C_1]$ *is also an answer. Note that* $[A_2, B_1, C_1]$ *is not an answer, since* A_2 *is* above *the node (*A_1*) on stack* S_A *to which* B_1 *points.* ▮

We make crucial use of this compact stack encoding in our algorithms, `PathStack` and `TwigStack`.

3.2 PathStack

Algorithm `PathStack`, which computes answers to a query path pattern, is presented in Figure 4, for the case when the streams contain nodes from a single XML document. When the streams contain nodes from multiple XML documents, the algorithm is easily extended to test equality of `DocId` before manipulating the nodes in the streams and stacks.

The key idea of Algorithm `PathStack` is to repeatedly construct (compact) stack encodings of partial and total answers to the query path pattern, by iterating through the stream nodes in sorted order of their `LeftPos` values; thus, the query path pattern nodes will be matched from the query root down to the query leaf. Line 2, in Algorithm `PathStack`, identifies the stream containing the next node to be processed. Lines 3-5 remove partial answers from the stacks that cannot be extended to total answers, given knowledge of the next stream node to be processed. Line 6 augments the partial answers encoded in the stacks with the new stream node. Whenever a node is pushed on the stack $S_{q_{min}}$, where q_{min} is the leaf node of the query path, the stacks contain an encoding of total answers to the query path, and Algorithm `showSolutions` is invoked by Algorithm `PathStack` (lines 7-9) to "output" these answers.

A natural way for Algorithm `showSolutions` to output query path answers encoded in the stacks is as n-tuples that are in sorted leaf-to-root order of the query path. This will ensure that, over the sequence of invocations of Algorithm `showSolutions` by Algorithm `PathStack`, the answers to the query path are also computed in leaf-to-root order. Figure 5 shows such a procedure for the case when only ancestor-descendant edges are present in the query path.

```
Algorithm PathStack(q)
while ¬end(q)
  q_min = getMinSource(q)
  for q_i in subtreeNodes(q) // clean stacks
    while (¬empty(S_{q_i}) ∧ topR(S_{q_i}) < nextL(T_{q_min}))
      pop(S_{q_i})
  moveStreamToStack(T_{q_min}, S_{q_min}, pointer to top(S_{parent(q_min)}))
  if (isLeaf(q_min))
    showSolutions(S_{q_min}, 1)
    pop(S_{q_min})

Function end(q)
  return ∀q_i ∈ subtreeNodes(q) : isLeaf(q_i) ⇒ eof(T_{q_i})

Function getMinSource(q)
  return q_i ∈ subtreeNodes(q) such that nextL(T_{q_i})
    is minimal

Procedure moveStreamToStack(T_q, S_q, p)
push(S_q, (next(T_q), p))
advance(T_q)
```

Figure 4: Algorithm `PathStack`

When parent-child edges are present in the query path, we also need to take the `LevelNum` information into account. `PathStack` does not need to change, but we need to ensure that each time `showSolutions` is invoked, it does not output incorrect tuples, in addition to avoiding unnecessary work. This can be achieved by modifying the recursive call (lines 6-7) to check for parent-child edges, in which case only a *single* recursive call (`showSolutions`($SN-1, S[SN]$.`index`$[SN]$.pointer_to_the_parent_stack)) needs to be invoked, after verifying that the `LevelNum` of the two nodes differ by one. Looping through all nodes in the stack $S[SN-1]$ would still be correct, but it would do more work than is strictly necessary.

If we desire the final answers to the query path be presented in sorted root-to-leaf order (as opposed to sorted leaf-to-root order), it is easy to see that it does not suffice that each invocation of `showSolutions` output answers encoded in the stack in the root-to-leaf order. To produce answers in the sorted root-to-leaf order, we would need to "block" answers, and delay their output until we are sure that no answer prior to them in the sort order can be computed. The details of how to achieve this naturally extend the intuitions of Algorithm Stack-Tree-Anc from [1], and are presented in the next section.

Example 3.2 *Consider the leftmost path,* `book–title–XML`*, in each of the query twigs of Figure 2. If we used the binary structural join algorithms of [26, 1], we would first need to compute matches to one of the parent-child structural relationships:* `book–title`*, or* `title–XML`*. Since every* `book` *has a* `title`*, this binary join would produce a lot of matches against an XML books database, even when there are only a few books whose title is XML. If, instead, we first computed matches to* `title–XML`*, we would also match pairs under* `chapter` *elements, as in the XML data tree of Figure 1(b), which do not extend to total answers to the query path pattern. Using Algorithm* `PathStack`*, partial answers are* compactly *represented in the stacks, and not output. Using the XML data tree of Figure 1(b), only one* total answer*, identified by the mapping* [`book` $\rightarrow (1, 1:150, 1)$, `title` $\rightarrow (1, 2:4, 2)$, `XML` $\rightarrow (1, 3, 3)$]*, is encoded in the stacks.* ▮

```
Procedure showSolutions(SN, SP)
// Assume, for simplicity, that the stacks of the query nodes from the root to the current
//     leaf node we are interested in can be accessed as S[1], ..., S[n].
// Also assume that we have a global array index[1..n] of pointers to the stack elements.
// index[i] represents the position in the i'th stack that we are interested in for the
//     current solution, where the bottom of each stack has position 1.

// Mark we are interested in position SP of stack SN.
index[SN] = SP
if (SN == 1) // we are in the root
  // output solutions from the stacks
  output (S[n].index[n], ..., S[1].index[1])
else // recursive call
  for i = 1 to S[SN].index[SN].pointer_to_parent
    showSolutions(SN − 1, i)
```

Figure 5: Procedure `showSolutions`

3.2.1 Blocking results

Consider the simple path query $A//D$. At any point during the query processing, if node a from A's stack is found to be an ancestor of node d from D's stream, then every node a' from A's stack that is an ancestor of a is also an ancestor of d. Since $a'.L < a.L$, we must delay output of the solution (a, d) until after (a', d) has been output. But there remains the possibility of a new element d' after d in D's stream joining with a' as long a' is on A's stack. Therefore, we cannot output the pair (a, d) until the ancestor node a' is popped from A's stack. Meanwhile, we can build up large join results that cannot yet be output. This situation is further aggravated for longer path queries. We now present a procedure that returns solutions in root-to-leaf order and generalizes the ideas in [1] for binary structural joins and is based on the concept of *blocking*.

For this purpose, we maintain two linked lists associated with each element n in the stacks: the first, *(S)elf-list*, represents all blocked descendant extensions with root element n, and the second second, *(I)nherit-list* represents all blocked descendant extensions with root elements that were descendants of n in the stack (only the top elements in the stacks have inherit-lists). At any point during the algorithm, the descendant extensions in the self- and inherit-lists of node n need to be expanded with the elements in the stacks of the ancestor nodes in the query, in the same way as in the procedure `showSolutions`. The main ideas of the algorithm remain the same, but we do not output solutions (using `showSolutions`) as soon as we detect them. Instead, we accumulate in the self- and inherit-lists the partial solutions found in root-to-leaf order, so when we finally output results they are guaranteed to be in the right order.

In particular, when a new element is pushed to a stack, we initialize its self- and inherit- lists to empty. Suppose we are popping element D_c from the stack of query node D. Depending on the current configuration, we proceed as follows [2]:

(a) Node D is not the query root, but has parent node A. Element D_c is not at the bottom of the stack, but has ancestor D_p (Figure 6(a)). In this case, we first determine nodes A_c and A_p (the nodes in A's stack pointed to by D_c and D_p respectively). Then we append the self- and inherit-lists (in that order) from D_c to the self-lists of all elements in A's stack starting with A_c (inclusive) and ending in A_p (exclusive).

(b) Node D is not the query root, but has parent node A. Element D_c is at the bottom of the stack. (Figure 6(b)). In this case, we first determine node A_c (the node in A's stack pointed to by D_c). Then

[2] In the case of a binary path query, only scenarios (c) and (d) are applicable and the technique reduces to that of [1].

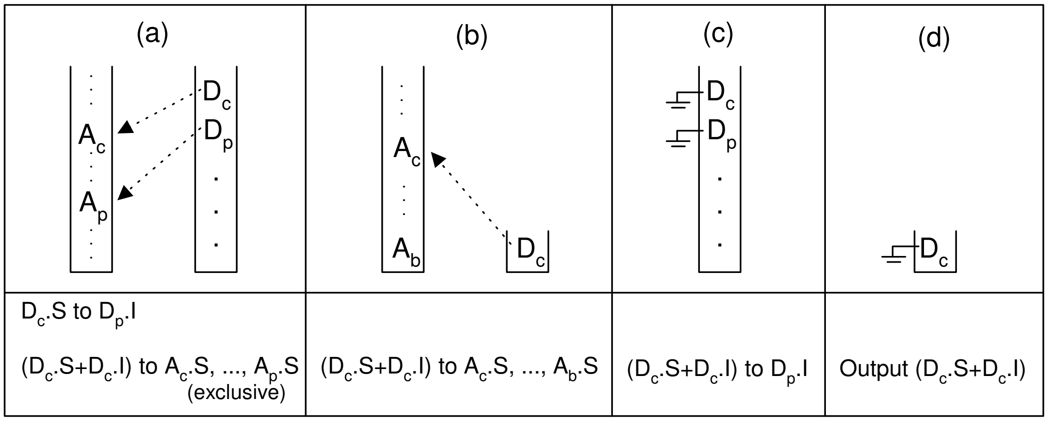


Figure 6: Possible stack configurations when blocking results.

we append the self- and inherit-lists (in that order) from D_c to the self-lists of all elements in A's stack starting with A_c and ending in A's bottom element.

(c) Node D is the query root. Element D_c is not at the bottom of the stack, but has ancestor D_p (Figure 6(c)). In this case we append the self- and inherit-lists (in that order) from D_c to the inherit-list of element D_p.

(d) Node D is the query root. Element D_c is at the bottom of the stack (Figure 6(d)). We first output the contents of the self-list and then the contents of the inherit-list of element D_c.

It is fairly simple to show that in cases (a),(b), and (c) we preserve all solutions when rearranging the linked lists. Also, each time we append one list to another, we are guaranteed that no future element will be appended out of order. Therefore, in case (d) we output solutions in the desired order. As an informal proof, consider for instance, case (a). All descendant extension from D_c's self- and inherit- lists need to be expanded with all elements from A_c to the bottom of the stack. For that purpose, we append the self- and inherit-lists D_c to all nodes in the parent query node from A_c to A_p exclusive. Since we also append this descendant extensions to the inherit-list of D_p, we are guaranteed that in a future iteration this solutions will reach to all the remaining nodes in stack A. So we did not lose any solution when manipulating the lists. Also, when we pop element D_c, we know that no new element from D's stream can start before D_c (and all its descendants) does, so any new solution will start after every descendant extension in D_c's self and inherit lists. In contrast, we cannot append these lists to the self list of A_p and its ancestors in the stack, because some solutions that start before them might be blocked in D_p and its ancestors. We solve that case by appending D_c's self- and inherit-lists to D_p's inherit list. The other cases can be explained similarly.

The only operation we perform over lists is “append” (except for the final read out). Since our implementation of the linked lists maintains the head and tail pointers, these append operations can be carried out in constant time and require no copying. Therefore, we only need to have access to the tail of each list in memory as computation proceeds. The rest of the list can be paged out. When list x is appended to list y, it is not necessary that the head of list x be in memory, since the append operation only establishes a link to this head in the tail of y. So all we need is to know the pointer for the head of each list, even if it is paged out. Each list page is thus paged out at most once, at paged back in again only when the list is ready for output. Also, the total number of entries in the lists is exactly equal to the number of entries in the output. We have then that the I/O required to maintain lists of results is proportional to the size of the output, provided that there is enough memory to hold in buffers the tail of each list).

3.3 Analysis of PathStack

The following proposition is a key to establishing the correctness of Algorithm `PathStack`.

Proposition 3.1 *If we fix node Y, the sequence of cases between node Y and nodes X on increasing order of* `LeftPos` *(L) is:* $(1|2)^*3^*4^*$. *Cases 1 and Cases 2 are interleaved, then all nodes in Case 3 before any node in Case 4, and finally all nodes in Case 4 (see Figure 7).* ▮

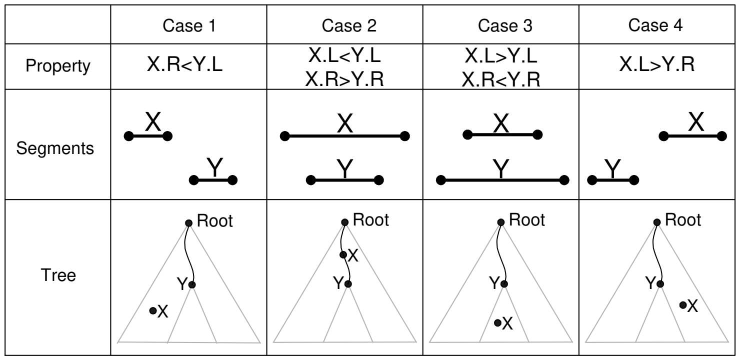


Figure 7: Cases for `PathStack` and `TwigStack`

Lemma 3.1 *Suppose that for an arbitrary node q in the path pattern query, we have that* `getMinSource`$(q) = q_N$. *Also, suppose that t_{q_N} is the next element in q_N's stream. Then, after t_{q_N} is pushed on to stack S_{q_N}, the chain of stacks from S_{q_N} to S_q verifies that their labels are included in the chain of nodes in the XML data tree from t_{q_N} to the root.* ▮

For each node $t_{q_{min}}$ pushed onto stack $S_{q_{min}}$, it is easy to see that the above lemma, along with the iterative nature of Algorithm `showSolutions`, ensures that all answers in which $t_{q_{min}}$ is a match for query node q_{min} will be output. This leads to the following correctness result:

Theorem 3.1 *Given a query path pattern q and an XML database D, Algorithm* `PathStack` *correctly returns all answers for q on D.* ▮

We next show optimality. Given an XML query path of length n, `PathStack` takes n input lists of tree nodes sorted by `(DocId, LeftPos)`, and computes an output sorted list of n-tuples that match the query path. It is straightforward to see that, excluding the invocations to `showSolutions`, the I/O and CPU costs of `PathStack` are linear in the sum of sizes of the n input lists. Since the cost of `showSolutions` is proportional to the size of the output list, we have the following optimality result:

Theorem 3.2 *Given a query path pattern q with n nodes, and an XML database D, Algorithm* `PathStack` *has worst-case I/O and CPU time complexities linear in the sum of sizes of the n input lists and the output list. Further, the worst-case space complexity of Algorithm* `PathStack` *is the minimum of (i) the sum of sizes of the n input lists, and (ii) the maximum length of a root-to-leaf path in D.* ▮

It is particularly important to note that the worst-case time complexity of Algorithm `PathStack` is *independent of the sizes of any intermediate results.*

3.4 PathMPMJ

A straightforward generalization of the MPMGJN algorithm [26] for path queries proceeds one stream at a time to get all solutions. Consider the path query $q_1//q_2//q_3$. The basic idea is as follows: Get the first (next) element from the stream T_{q_1} and *generate all solutions* that use that particular element from T_{q_1}. Then, advance T_{q_1} and *backtrack* T_{q_2} and T_{q_3} accordingly (i.e., to the earliest position that might lead to a solution). This procedure is repeated until T_{q_1} is empty. The *generate all solutions* step recursively starts with the first marked element in T_{q_2}, gets all solutions that use that element (and the calling element in T_{q_1}), then advances the stream T_{q_2} until there are no more solutions with the current element in T_{q_2}, and so on. We refer to this algorithm as `PathMPMJNaive`, in our experiments.

It turns out that maintaining only one mark per stream (for backtracking purposes) is too inefficient, since all marks need to point to the earliest segment that can match the current element in T_{q_1} (the stream of the root node). A better strategy is to use a *stack* of marks, as described in Algorithm `PathMPMJ` in Figure 8. In this optimized generalization of MPMGJN, each query node will not have a single mark in the stream, but "k" marks where k is the number of its ancestors in the query. Each mark points to an earlier position in the stream, and for query node q, the i'th mark is the first point in T_q such that the element in T_q starts after the current element in the stream of q's i'th ancestor.

```
Algorithm PathMPMJ(q)
while (¬eof(T_q) ∧ (isRoot(q) ∨ nextL(q) < nextR(parent(q))))
  for (q_i ∈ subtreeNodes(q)) // advance descendants
    while (nextL(q_i) < nextL(parent(q_i)))
      advance(T_{q_i})
    PushMark(T_{q_i})
  if (isLeaf(q)) outputSolution() // solution in the streams' heads
  else PathMPMJ(child(q))
  advance(T_q)
  for (q_i ∈ subtreeNodes(q)) // backtrack descendants
    PopMark(T_{q_i})
```

Figure 8: Algorithm `PathMPMJ`

Theorem 3.3 *Given a query path pattern q and an XML database D, Algorithm* `PathMPMJ` *correctly returns all answers for q on D.* ▮

While the two extensions of MPMGJN appear similar, the difference between their performance is noticeable, as we shall see in the experimental section. Further, as was the case with MPMGJN, Algorithm `PathMPMJ` is not asymptotically optimal either.

4 Twig Join Algorithms

4.1 Limitations of Using PathStack

A straightforward way of computing answers to a query twig pattern is to decompose the twig into multiple root-to-leaf path patterns, use `PathStack` to identify solutions to each individual path, and then merge-join these solutions to compute the answers to the query. This approach, which we experimentally evaluate in Section 5, faces the same fundamental problem as the techniques based on binary structural joins, towards a holistic solution: *many intermediate results may not be part of any final answer*, as illustrated below.

Example 4.1 *Consider the query sub-twig rooted at the* `author` *node of the twig pattern in Figure 2(b). Against the XML database in Figure 1(b), the two paths of this query:* `author-fn-jane`*, and* `author-ln-doe`*, have two solutions each, but the query twig pattern has only one solution.* ∎

In general, if the query (root-to-leaf) paths have many solutions that do not contribute to the final answers, using `PathStack` (as a sub-routine) is suboptimal, in that the overall computation cost for a twig pattern is proportional not just to the sizes of the input and the final output, but also to the sizes of intermediate results. In this section, we seek to overcome this suboptimality using Algorithm `TwigStack`.

4.2 TwigStack

Algorithm `TwigStack`, which computes answers to a query twig pattern, is presented in Figure 9, for the case when the streams contain nodes from a single XML document. As with Algorithm `PathStack`, when the streams contain nodes from multiple XML documents, the algorithm is easily extended to test equality of `DocId` before manipulating the nodes in the streams and on the stacks.

```
Algorithm TwigStack(q)
   // Phase 1
01 while ¬end(q)
02   q_act = getNext(q)
03   if (¬isRoot(q_act))
04     cleanStack(parent(q_act), nextL(q_act))
05   if (isRoot(q_act) ∨ ¬empty(S_parent(q_act)))
06     cleanStack(q_act, next(q_act))
07     moveStreamToStack(T_q_act, S_q_act, pointertotop(S_parent(q_act)))
08     if (isLeaf(q_act))
09       showSolutionsWithBlocking(S_q_act, 1)
10       pop(S_q_act)
11   else advance(T_q_act)
   // Phase 2
12 mergeAllPathSolutions()

Function getNext(q)
01 if (isLeaf(q)) return q
02 for q_i in children(q)
03   n_i = getNext(q_i)
04   if (n_i ≠ q_i) return n_i
05 n_min = minarg_n_i  nextL(T_n_i)
06 n_max = maxarg_n_i  nextL(T_n_i)
07 while (nextR(T_q) < nextL(T_n_max))
08   advance(T_q)
09 if (nextL(T_q) < nextL(T_n_min)) return q
10 else return n_min

Procedure cleanStack(S, actL)
01 while (¬empty(S) ∧ (topR(S) < actL))
02   pop(S)
```

Figure 9: Algorithm `TwigStack`

Algorithm `TwigStack` operates in two phases. In the first phase (lines 1-11), some (*but not all*) solutions to individual query root-to-leaf paths are computed. In the second phase (line 12), these solutions are merge-joined to compute the answers to the query twig pattern.

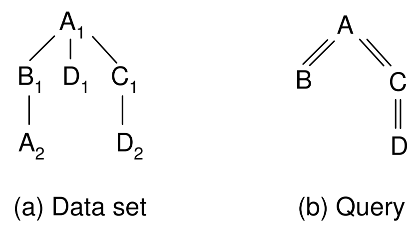


Figure 10: Example data set and query to illustrate the `getNext` procedure.

The *key difference* between `PathStack` and the first phase of `TwigStack` is that before a node h_q from the stream T_q is pushed on its stack S_q, `TwigStack` (via its call to `getNext`) ensures that: (i) node h_q has a *descendant* h_{q_i} in each of the streams T_{q_i}, for $q_i \in \texttt{children}(q)$, and (ii) each of the nodes h_{q_i} recursively satisfies the first property. Algorithm `PathStack` does not satisfy this property (and it does not need to do so to ensure (asymptotic) optimality for query *path patterns*). Thus, when the query twig pattern has only ancestor-descendant edges, each solution to each individual query root-to-leaf path is *guaranteed* to be merge-joinable with at least one solution to each of the other root-to-leaf paths. This ensures that no intermediate solution is larger than the final answer to the query twig pattern.

The second merge-join phase of Algorithm `TwigStack` is linear in the sum of its input (the solutions to individual root-to-leaf paths) and output (the answer to the query twig pattern) sizes, only when the inputs are in sorted order of the common prefixes of the different query root-to-leaf paths. This requires that the solutions to individual query paths be output in root-to-leaf order as well, which necessitates blocking; `showSolutions` (from Figure 5), cannot be used and needs to be extended with the ideas of Section 3.2.1.

Example 4.2 *Consider again the query of Example 4.1, which is the sub-twig rooted at the* `author` *node of the twig pattern in Figure 2(b), and the XML database tree in Figure 1(b). Before Algorithm* `TwigStack` *pushes an* `author` *node on the stack* $S_{\texttt{author}}$*, it ensures that this* `author` *node has: (i) a descendant* `fn` *node in the stream* $T_{\texttt{fn}}$ *(which in turn has a descendant* `jane` *node in* $T_{\texttt{jane}}$*), and (ii) a descendant* `ln` *node in the stream* $T_{\texttt{ln}}$ *(which in turn has a descendant* `doe` *node in* $T_{\texttt{doe}}$*). Thus, only one of the three* `author` *nodes (corresponding to the third author) from the XML data tree in Figure 1(b) is pushed on the stacks. Subsequent steps ensure that only one solution to each of the two paths of this query:* `author–fn–jane`*, and* `author–ln–doe`*, is computed. Finally, the merge-join phase computes the desired answer.* ∎

4.3 Analysis of TwigStack

In this section we discuss the correctness of algorithm `TwigStack` for processing twig queries, and then we analyze its complexity.

Definition 4.1 *Consider a twig query Q. For each node $q \in \texttt{subtreeNodes}(Q)$ we define the* head *of q, denoted h_q, as the first element in T_q that participates in a solution for the sub-query rooted at q. We say that a node q has a* minimal descendant extension *if there is a solution for the sub-query rooted at q composed entirely of the head elements of* `subtreeNodes`*(q).*

Example 4.1 *Consider the data set and query of Figure 10. The first element in T_A that participates in a solution for the query, h_A=A_1 (such solution involves the nodes A_1, B_1, C_1, and D_2). Similarly, $h_B = B_1$, $h_C = C_1$ and $h_D = D_1$, since those are the first elements that participate in a solution for the sub-queries rooted at B, C, and D, respectively. In this example, nodes B and D have minimal descendant extensions.*

Node C does not have a minimal descendant extension since its first solution (using elements C_1 and D_2) does not use $h_D = D_1$. For the same reason, node A does not have a minimal descendant extension.

Proposition 3.1, based on Figure 7, is also important for establishing the following lemma, used for proving the correctness of Algorithm `TwigStack`.

Lemma 4.1 [**Descendant extension**] *Suppose that for an arbitrary node q in the twig query we have that* $\texttt{getNext}(q) = q_N$. *Then, the following properties hold:*

1. *q_N has a minimal descendant extension.*
2. *For each node $q' \in$ $\texttt{subtreeNodes}(q_N)$, the first element in $T_{q'}$ is $h_{q'}$.*
3. *Either (a) $q = q_N$ or (b) $\texttt{parent}(q_N)$ does not have a minimal right extension because of q_N (and possibly other nodes). In other words, the solution rooted at $p = \texttt{parent}(q_N)$ that uses h_p does not use h_q for node q but some other element whose L component is larger than that of h_q.*

Proof: (Induction on the number of descendants of q). If q is a leaf node, it trivially verifies all properties, so we return it in line 1 of the algorithm. Otherwise, we get $n_i = \texttt{getNext}(q_i)$ recursively for each child q_i of q in lines 2-4. If for some i, we have that $n_i \neq q_i$ but some descendant of q_i, we know by inductive hypothesis that n_i verifies properties (1), (2) and (3-b) above with respect to q_i. All properties are still valid with respect to node q, so we return node n_i in line 4. Otherwise, we know by inductive hypothesis that all of q's children satisfy properties (1), (2) and (3-a) above with respect to their corresponding sub-queries. We first advance from T_q all segments that are in Case 1 with respect to *any* head segment in the n_i's streams (lines 7-8). After that, we know that q is in case 2, 3 or 4 with respect to every n_i by using Proposition 3.1. If q is in Case 2 with respect to all n_i, it satisfies properties (1), (2) and (3-a) above, so we return it (line 9). Otherwise, we can guarantee that the minimal n_i verifies properties (1), (2) and (3-b) above, since its parent (q) does not have a minimal descendant extension because of n_i (and possibly other children), so we return such minimal node n_i in line 10 (the element in q does not satisfy property (3) at the moment, but could do so later on). To verify the last condition (lines 9-10) we simply check if the left bound of the element in q is lower than the left bound of the minimal n_i. If that is the case, we know that q's element starts before any n_i and ends after each n_i (since we checked that it ends after each n_i starts). ∎

Lemma 4.2 *Consider the following fragment of code:*

```
while ¬end(q)
      qN=getNext(q); advance(qN)
```

The following properties hold:

1. `getNext` *returns in node q_N* all and only *elements in the data set with a descendant extension.*
2. *If element x is element y's ancestor in some solution, then x is returned by* `getNext` *before y.*

Proof: (Induction on the number of calls to `getNext`). In particular, we will show that property (1) above holds by proving that all elements that are skipped in line 10 of `getNext` are guaranteed not to have any descendant extension. Consider the first call to `getNext` for query q, and assume that $\texttt{getNext}(q) = q_N$. By Lemma 4.1, property (1), we know that all the first elements in the streams of $\texttt{subtreeNodes}(q_N)$ are part of the *first* descendant extension involving those nodes. Therefore, all elements that were advanced in step

10 of `getNext` are guaranteed not to be part of any descendant extension, and property (1) holds. Also, either q_N=q (with no ancestors) or the solution rooted at q_N is guaranteed not to be part of any solution involving q_N's ancestors, because each descendant extension from p=`parent`(q_N), using some element x from p's stream, verifies $x.L \geq h_p.L > h_{q_N}.L$ (using Lemma 4.1, property (3)). Therefore, property 2 holds.

For subsequent calls to `getNext`, we proceed as follows. Assume that `getNext`(q) = q_N and consider each element x (from node $q_x \in$ `subtreeNodes`(q_N)) that is advanced in line 10. Element x cannot be part of any descendant extension with elements in the streams of `subtreeNodes`(q_x), since x ends before the first solution in some child's stream (see condition in line 9 of `getNext`). Besides, x cannot be part of any descendant extension with elements that were already processed by `getNext` by property (2) of the inductive hypothesis. Therefore, x is guaranteed not to be part of any descendant extension, and property (1) holds. Now suppose that $q_N \neq q$ and p = `parent`(q_N). Using Lemma 4.1, property (3), we know that $h_p.L > h_{q_N}.L$. Therefore, any solution involving element $h_q.L$ must use some element from p that was already returned by `getNext`. Using property (2) of the inductive hypothesis, we know that all elements from p's ancestors in such solutions were returned by `getNext` before the corresponding element from p, which in turn was returned before h_{q_N}. Property (2) holds as well. ∎

We then know that when some node q_N is returned by `getNext`, h_{q_N} is guaranteed to have a descendant extension in `subtreeNodes`(q_N). We also know that any element in the ancestors of q_N that uses h_{q_N} in a descendant extension was returned by `getNext` before h_{q_N}. Therefore, we can maintain for each node q in the query, the elements that are part of a solution involving other elements in the streams of `subtreeNodes`(q). Then, each time that q_N = `getNext`(q) is a leaf node, we output all solutions that use h_{q_N}. We now prove that we can achieve that goal by maintaining a single stack per node in the query.

Lemma 4.3 *Suppose that in the current iteration we have* `getNext`(q)=q_N. *Then, there is no new solution involving: (a) some element from* q_N *that ends before* h_{q_N} *starts, and (b) some element from the streams in* `subtreeNodes`(q_N).

Proof: Suppose that on the contrary, there is a new solution using some (already processed) element from q_N (denoted s_{q_N}) for which $s_{q_N}.R < h_{q_N}.L$. Using the containment property, we know that all elements from `subtreeNodes`(q_N) in such solution must end before $s_{q_N}.R$, and therefore, before $h_{q_N}.L$. Since `getNext`(q) = q_N we know (Lemma 4.1) that q_N has a minimal descendant extension, and therefore, all elements in the streams of `subtreeNodes`(q_N) start after h_{q_N} does, which is a contradiction. ∎

The lemma above guarantees that for each node q, the set of elements that can be part of a new solution are exactly those including the last element from q returned by `getNext`. Therefore, a single stack per node suffices to keep track of all elements with potential new solutions. The following lemma shows that we can easily chain the elements of the stacks in the same way as in `PathStack` and use `showSolutions` to output all solutions.

Lemma 4.4 *Suppose that, in the current iteration, we have* `getNext`(q)=q_N, *and* $q \neq q_N$. *Let* p = `parent`(q_N). *Then, there is no "new" solution involving: (a) some element from* p *that ends before* h_{q_N} *starts, and (b) some element from the streams in* `subtreeNodes`(p).

Proof: The proof is similar to that of Lemma 4.3. Suppose that on the contrary, there is a new solution using some element from p (denoted s_p) for which $s_p.R < h_{q_N}.L$. We know that all elements from `subtreeNodes`(p) in such solution must end before $s_p.R$, and therefore, before $h_{q_N}.L$. Since `getNext`(q) = q_N we know (see algorithm `getNext` for `TwigStack`) that for all children n_i of p (including q_N): (1) `getNext`(n_i) = n_i, and (2) $h_{q_N}.L \leq h_{n_i}.L$. Using Lemma 4.1 we know that each n_i has a minimal descendant extension, and therefore all elements in the streams of `subtreeNodes`(n_i) start after $h_{n_i}.L$, and therefore, after $h_{q_N}.L$. We have a contradiction and the lemma holds. ∎

The lemma above guarantees that each time `getNext` returns node q_N we can clean from q_N's parent stack all elements that do not include h_{q_N} (since they cannot participate in any new solution) and set the h_{q_N}'s pointer in q_N's stack to the top of q_N's parent stack. Using the two lemmas above, we prove our main result for `TwigStack`.

Theorem 4.1 *Given a query twig pattern q, and an XML database D, Algorithm* `TwigStack` *correctly returns all answers for q on D.*

Proof: In Algorithm `TwigStack`, we repeatedly find `getNext`(q) for query q. Assume that `getNext`$(q) = q_N$. Let A_{q_N} be the set of nodes in the query that are ancestors of q_N. Using Lemma 4.2 we know that `getNext` already returned all elements from the streams of nodes in Aq_N that are part of a solution that uses h_{q_N}. If $q \neq q_N$, in line 3 we pop from `parent`(q_N)'s stack all elements that are guaranteed not to participate in any new solution (Lemma 4.4). After that, in line 4 we test whether h_{q_N} participates in a solution. We know that q_N has a descendant extension by Lemma 4.1, property 1. If $q \neq q_N$ and `parent`(q_N)'s stack is empty, node q_N does not have an ancestor extension. Therefore it is guaranteed not to participate in any solution, so we advance q_N in line 10 and continue with the next iteration. Otherwise, node q_N has both ancestor and descendant extensions and therefore it participates in at least one solution. We then clean q_N's stack (Lemma 4.3) and after that we push h_{q_N} to it (lines 5-6). By Lemma 4.4 we know that the pointer to the top of `parent`(q_N)'s stack correctly identifies all solutions using h_{q_N}. Finally, if q_N is a leaf node, we decompress the stored solutions from the stacks (lines 7-9). ∎

While correctness holds for query twig patterns with both ancestor-descendant and parent-child edges, we can prove optimality only for the case where the query twig pattern has only ancestor-descendant edges. The intuition is simple. Since we push into the stacks only elements that have both a descendant and an ancestor extension, we are guaranteed that no element that does not participate in any solution is pushed into any stack. Therefore, the merge postprocessing step is optimal, and we have the following result.

Theorem 4.2 *Consider a query twig pattern q with n nodes, and only ancestor-descendant edges, and an XML database D. Algorithm* `TwigStack` *has worst-case I/O and CPU time complexities linear in the sum of sizes of the n input lists and the output list. Further, the worst-case space complexity of Algorithm* `TwigStack` *is the minimum of (i) the sum of sizes of the n input lists, and (ii) n times the maximum length of a root-to-leaf path in D.* ∎

It is particularly important to note that, for the case of query twigs with ancestor-descendant edges, the worst-case time complexity of Algorithm `TwigStack` is *independent of the sizes of solutions to any root-to-leaf path of the twig.*

4.4 Suboptimality for Parent-Child Edges

Theorem 4.2 holds only for query twigs with ancestor-descendant edges. Unfortunately, in the case where the twig pattern contains a parent-child edge between two elements (e.g., see the query in Example 4.2), Algorithm `TwigStack` is no longer guaranteed to be I/O and CPU optimal. In particular, the algorithm might produce a solution for one root-to-leaf path that does not match with any solution in another root-to-leaf path.

Consider the query twig pattern with three nodes: A, B and C, and parent-child edges between (A, B) and between (A, C). Let the XML data tree consist of node A_1, with children (in order) A_2, B_2, C_2, such that A_2 has children B_1, C_1. The three streams T_A, T_B and T_C have as their first elements A_1, B_1 and C_1 respectively. In this case, we *cannot* say if any of them participates in a solution without advancing other streams, and we cannot advance any stream before knowing if it participates in a solution. As a result, optimality can no longer be guaranteed.

4.5 Using XB-Trees

Algorithms `PathStack` and `TwigStack` need to process each node in the input lists to check whether or not it is part of an answer to the query (path or twig) pattern. When the input lists are very long, this may take a lot of time. In this section, we propose the use of a variant of B-trees, denoted *XB-tree*, on the input lists to speed up this processing.

4.5.1 XB-Tree Description

As its name suggests, the XB-tree is a variant of the B-tree, designed for indexing the positional representation `(DocId, LeftPos : RightPos, LevelNum)` of elements in the XML tree. We describe the index structure when all nodes belong to the same XML document; the extension to multiple documents is straightforward.

The nodes in the leaf pages of the XB-tree are sorted by their `LeftPos` (L) values; this is similar to the leaf pages of a B-tree on the L values. The difference between a B-tree and an XB-tree is in the data maintained at internal pages. Each node N in an internal page of the XB-tree consists of a bounding segment $[N.L, N.R]$ (where L denotes `LeftPos` and R denotes `RightPos`) and a pointer to its child page $N.page$ (which contains nodes with bounding segments completely included in $[N.L, N.R]$). The bounding segments of nodes in internal pages might partially overlap, but their L positions are in increasing order. Besides, each page P has a pointer to the parent page $P.parent$ and the integer $P.parentIndex$ which is the index of the node in $P.parent$ that points back to P. The construction and maintenance of an XB-tree is very similar to those in a B-tree, using the L value as the key; the difference is that the R values need to be propagated up the index structure.

4.5.2 Using XB-Trees

We maintain a pointer $act = (actPage, actIndex)$ to the $actIndex$'th node in page $actPage$ of the XB-tree. There are two operations over the XB-tree that affect this pointer:

1. `advance`. If $act = (actPage, actIndex)$ does not point to the last node in the current page, we simply advance $actIndex$. Otherwise we replace act with the value $(actPage.parent, actPage.parentIndex)$ and recursively advance it.
2. `drillDown`. If $act = (actPage, actIndex)$, $actPage$ is not a leaf page, and N is the $actIndex$'th node in $actPage$, we replace act with $(N.page, 0)$ so that it points to the first node in $N.p$.

Initially $act = (rootPage, 0)$, pointing to the first node in the root page of the XB-tree. When act points to the last node in $rootPage$ and we advance it, we finish the traversal.

We can modify the previous algorithms easily to use XB-trees. Algorithm `TwigStackXB`, in Figure 11, extends Algorithm `TwigStack` so that it uses XB-trees. The only changes are in the lines indicated by parentheses. The function `isPlainValue` returns true if the actual pointer in the XB-tree is pointing to a leaf node (actual value in the original stream). If we define `isPlainValue`(T)=true when T is not an XB-tree but a regular file, this algorithm reduces to the previous one.

```
Algorithm TwigStackXB(q)
while ¬end(q)
  q_act = getNext(q)
  if (isPlainValue(T_{q_act}))
    if (¬isRoot(q_act))
      cleanStack(parent(q_act), next(q_act))
    if (isRoot(q_act) ∨ ¬empty(S_{parent(q_act)}))
      cleanStack(q_act, next(q_act))
      moveStreamToStack(T_{q_act}, S_{q_act}, pointer to top(S_{parent(q_act)}))
      if (isLeaf(q_act))
        showSolutionsWithBlocking(S_{q_act}, 1)
        pop(S_{q_act})
    else advance(T_{q_act})
  else if (¬isRoot(q_act) ∧ empty(S_{parent(q_act)}) ∧ nextL(T_{parent(q_act)}) > nextR(T_{q_act}))
    advance(T_{q_act}) // Not part of a solution
  else // Might have a child in some solution
    drillDown(T_{q_act})
  // Phase 2
mergeAllPathSolutions()

Function getNext(q)
if (isLeaf(q)) return q
for q_i in children(q)
  n_i = getNext(q_i)
  if (q_i ≠ n_i ∨ ¬isPlainValue(T_{n_i})) return n_i
n_min = minarg_{n_i} nextL(T_{n_i})
n_max = maxarg_{n_i} nextL(T_{n_i})
while (nextR(T_q) < nextL(T_{n_max}))
  advance(T_q)
if (nextL(T_q) < nextL(T_{n_min})) return q
else return n_min

Procedure cleanStack(S, actL)
while (¬empty(S) ∧ (topR(S) < actL))
  pop(S)
```

Figure 11: Algorithm `TwigStackXB`

Theorem 4.3 *Given a query twig pattern q and an XML database D, Algorithm* `TwigStackXB` *correctly returns all answers for q on D.* ∎

While we do not have any analytical results about the efficiency of Algorithm `TwigStackXB`, we show experimentally that it performs matching of query twig patterns in *sub-linear time*.

5 Experimental Evaluation

In this section we present experimental results on the performance of the join algorithms introduced in Sections 3 and 4 using both real and synthetic data.

5.1 Experimental Setting

We implemented all XML join algorithms in C++ using the file system as a simple storage engine. All experiments were run on a 550Mhz Pentium III processor with 768MB of main memory and a 2GB quota

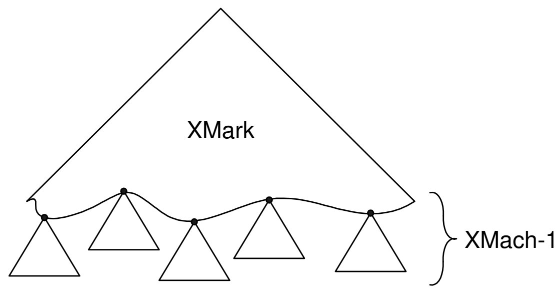


Figure 12: The *Benchmark* data set combines both XMark and XMach-1 documents.

of disk space, running Red Hat Linux 7.1. We used the following synthetic and real-world data sets for our experiments:

***Random*:** We generated random trees using three parameters: *depth*, *fan-out* and *number of different labels*. For most of the experiments presented involving *Random* data sets, we generated full binary and ternary trees. Unless specified explicitly, the node labels in the trees were uniformly distributed. We tried other configurations (larger fanout and random depth in the tree) obtaining consistent results.

***Benchmark*:** We used a combination of the XMark [25] and XMach-1 [24] benchmarks. The XMark benchmark is based on a rich DTD that models a database in an Internet auction site. However, XMark usually produces shallow XML files with not too many nested levels. On the other hand, the XMach-1 benchmark is based on a simpler DTD that models documents. As a result, the produced XML files can model deeply nested sections and subsections. To take advantage of both models, we generated the *Benchmark* data set by merging partial results from XMark and XMach-1. In particular, we first generated a temporary XML file using the XMark benchmark with scaling factor equal to one. We then replaced each of the 105,396 `description` tags (that originally corresponded to free text) with a small XML fragment produced by the XMark benchmark (see Figure 12). The resulting data set has depth 37 and around 11.7 million nodes.

***DBLP*:** The real data set is an "unfolded" fragment of the DBLP database. In the original DBLP data set, each author is represented by a name, a homepage, and a list of papers. In turn, each paper contains a title, the conference where it was published, and a list of coauthors. We generated our unfolded fragment of DBLP in the following way. We started with an arbitrary author and converted the corresponding information to XML format. Further, for each paper, we replaced each coauthor name with the actual information for that author. We recursively continued unfolding authors until we reached a previously traversed author, or a depth of 200 authors. The resulting XML data set has depth 805 and around 3 million nodes. It represents 93,536 different papers from 36,900 unique authors.

5.2 Holism: Binary Structural Joins vs `PathStack`

In this first experiment we compare our holistic `PathStack` algorithm against strategies that use a combination of binary structural joins [1]. For this purpose, we used a *Random* data set consisting of 1M nodes and six different labels, namely: $A_1, A_2, \ldots, A_6$. [3] We issued the path query $A_1//A_2//\ldots//A_6$ and evaluated

[3] Note that the actual XML data can contain many more labels, but that does not affect our techniques since we only access the indexed streams present in the query.

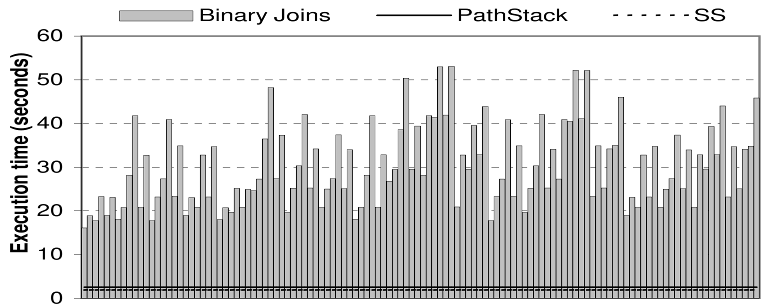


Figure 13: Holistic versus structural binary joins for path queries

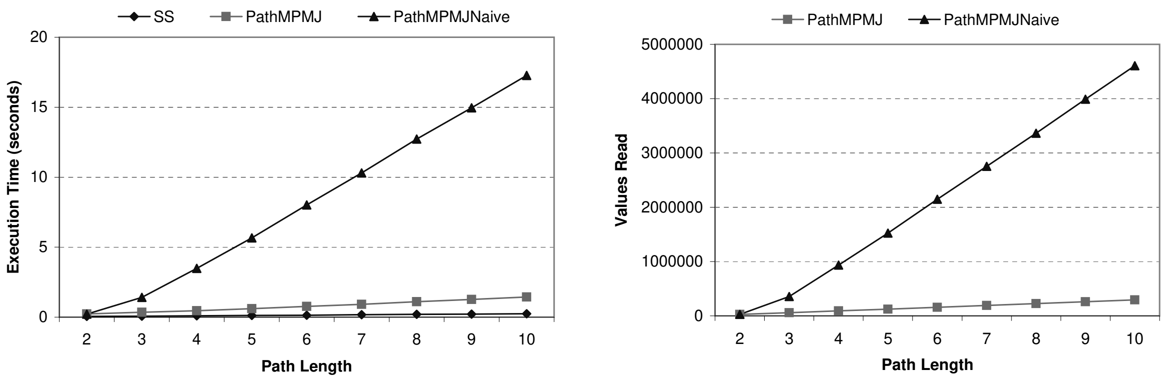
it using `PathStack`. Then, we evaluated all binary structural join strategies resulting from applying all possible join orders. Figure 13 shows the execution time of all binary join strategies, where each strategy is represented with a single bar. We also show with a solid line the execution time of `PathStack`, and with a dotted line the time it takes to simply read the input data using a sequential scan (labeled SS).

For this query, `PathStack` took $2.53s$, slightly more than the $1.87s$ taken by the sequential scan over the input data. In contrast, the strategies based on binary structural joins ranged from $16.1s$ for the join ordering $((((A_1//A_2)//A_3)//A_4)//A_5)//A_6$, to $53.07s$ for the join ordering $A_1//((A_2//(A_3//(A_4//A_5)))//A_6)$. The first conclusion we can draw from these execution times is that optimization plays an important role for binary structural joins, since a bad join ordering can result in a plan that is more than three times worse than the best plan. The second conclusion we can draw is that the holistic strategy is superior to the approach of using binary structural joins for arbitrary join orders; in this case, it results in more than a six-fold improvement in execution time over the best strategy that uses binary structural joins.

5.3 Paths: `PathStack` vs `PathMPMJ`

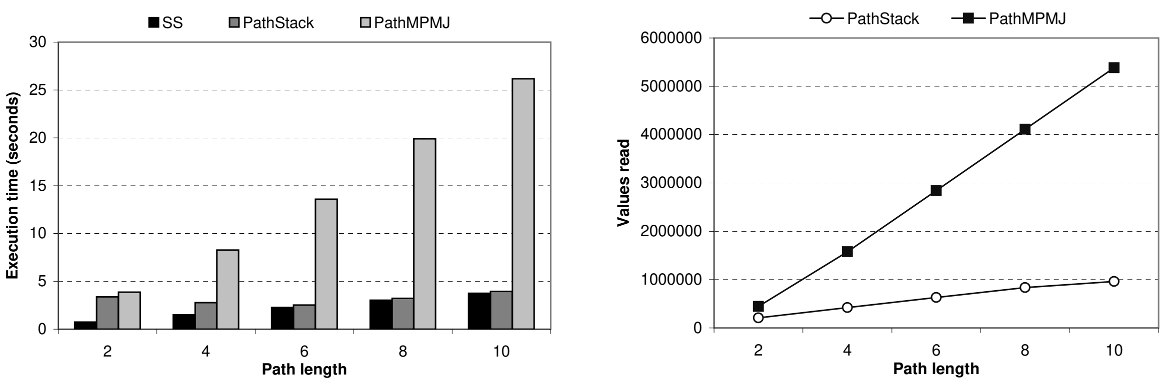
In this section we study the efficiency of the different holistic path join algorithms of Section 3. We first compare the two versions of `PathMPMJ`. We used a 64K *Random* data set with 10 different nodes $A_1, \ldots A_{10}$, and issue path queries of different lengths. Figure 14(a) shows the execution times of both techniques, as well as the time taken for a sequential scan over the input data. `PathMPMJNaive` is much slower compared to the optimized `PathMPMJ` (generally over an order of magnitude). The reason is that `PathMPMJNaive` is too conservative when backtracking and reads several times unnecessary portions of the data. As shown in Figure 14(b), `PathMPMJNaive` read as much as 15 times the number of elements `PathMPMJ` did. Since the performance of `PathMPMJNaive` degrades considerably with the size of the data set and the length of the input query, we do not consider this strategy for the remainder of this paper.

We now compare `PathStack` against the optimized `PathMPMJ`. In Figure 15 we show the execution time and the number of nodes read from disk for path queries of different length and a *Random* data set of 1M nodes and 10 different values. Clearly, `PathStack` results in considerably better performance than `PathMPMJ`, and this difference increases with longer path queries. This is explained by the fact that `PathStack` makes a single pass over the input data, while `PathMPMJ` needs to backtrack and read again large portions of data. For instance, for a path query of length 10, `PathMPMJ` reads the equivalent of five times the size of the original data, as seen in Figure 15(b). In Figure 15(a), for path queries of length two, the execution time of `PathStack` is considerably slower than that of the sequential scan, and closer to `PathMPMJ`. This behavior is due to the fact that for the path query of length two, the number of solutions is rather large (more than 100K), so most of the execution time is used in processing these solutions and writing them back to disk. For longer path queries, the number of solutions is considerably smaller, and the execution time of `PathStack`



(a) Execution time (b) Number of elements read

Figure 14: `PathMPMJ` versus `PathMPMJNaive` for path queries using *Random* data



(a) Execution time (b) Number of elements read

Figure 15: `PathStack` versus `PathMPMJ` using *Random* data sets

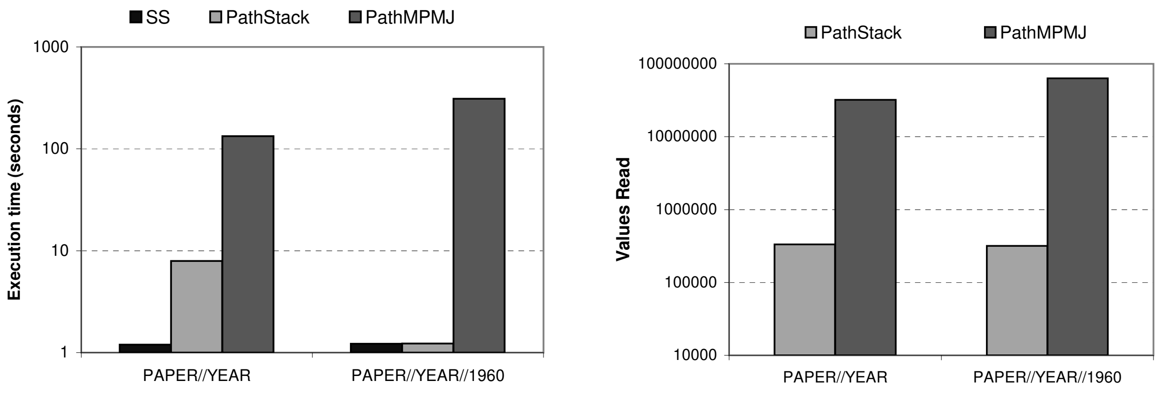
is closer to that of the sequential scan and much smaller than that of `PathMPMJ`.

Figure 16 shows the execution time and number of values read for two simple path queries over the unfolded *DBLP* data set (note the logarithmic scale on the Y axis). Due to the specific nesting properties between nodes in this data set, the `PathMPMJ` algorithm spends much time backtracking and reads several times the same values. For instance, for the path query of length three in Figure 16, `PathMPMJ` reads two orders of magnitude more elements than `PathStack`.

Finally, Figure 17 shows the execution time and number of values read for a family of long path queries over the *Benchmark* data set. In particular, we generated 10 different queries by starting with the query template `REGIONS/NAMERICA//ITEM//DESCRIPTION/SECTION/SECTION/SECTION/HEAD/WORD-i` and replacing the label `WORD-i` with each one of the 10 most frequent words in the data set. As in the previous experiments, `PathStack` results in significantly more efficient executions than `PathMPMJ`.

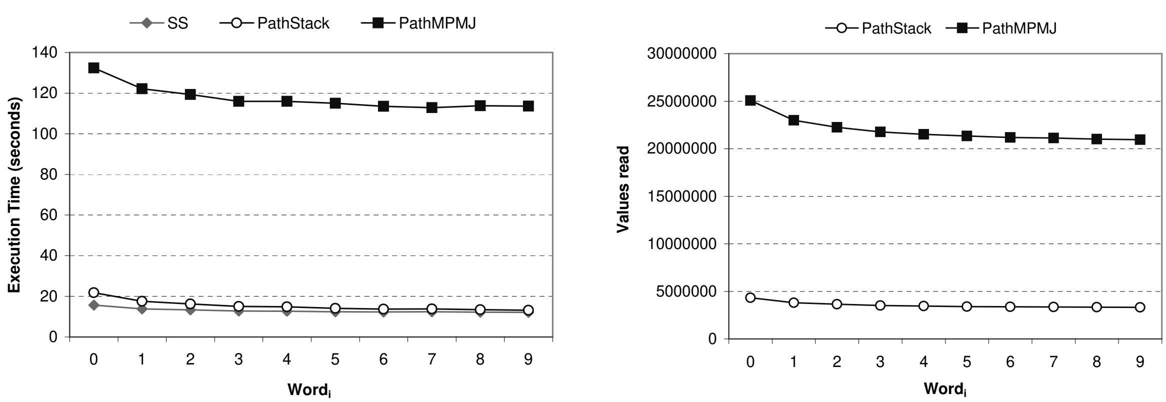
5.4 Twigs: `PathStack` vs `TwigStack`

We now focus on twig queries, and compare our `TwigStack` algorithm against the naive application of `PathStack` to each branch in the tree followed by a merge step. As shown in Section 4, `TwigStack` is optimal for ancestor/descendant relationships, but it is provably suboptimal for parent/child relationships. In this section, we analyze these two cases separately.



(a) Execution time (b) Number of elements read

Figure 16: `PathStack` versus `PathMPMJ` for the unfolded *DBLP* data set



(a) Execution time (b) Number of elements read

Figure 17: `PathStack` versus `PathMPMJ` for the *Benchmark* data set

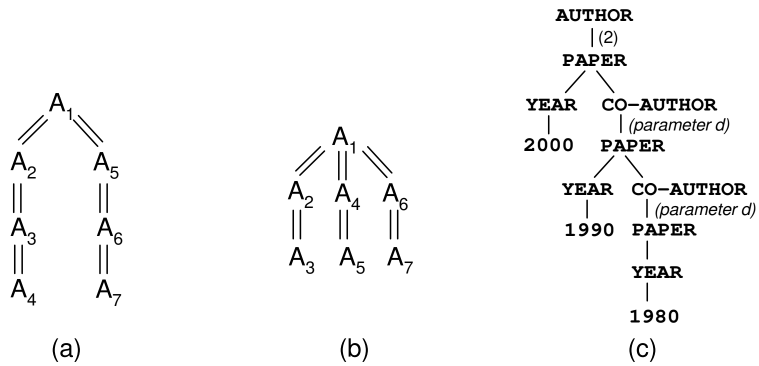
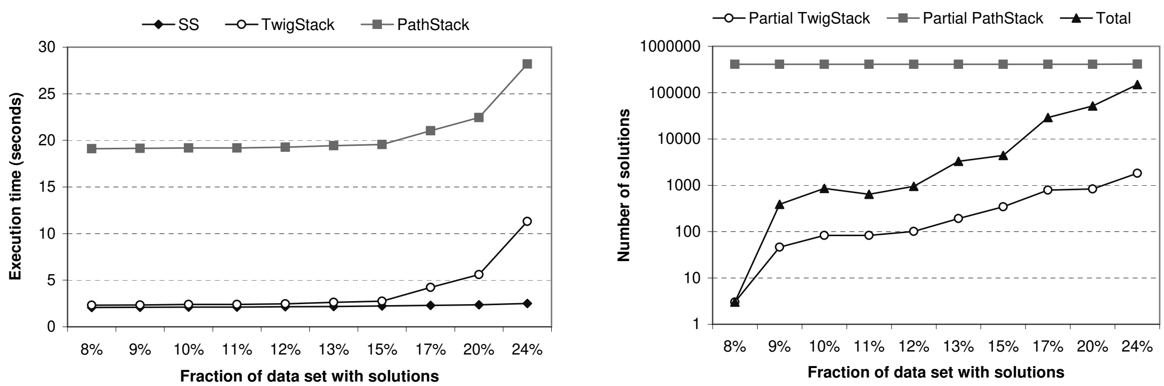


Figure 18: Twig queries used in the experiments of Section 5.4.1.



(a) Execution time

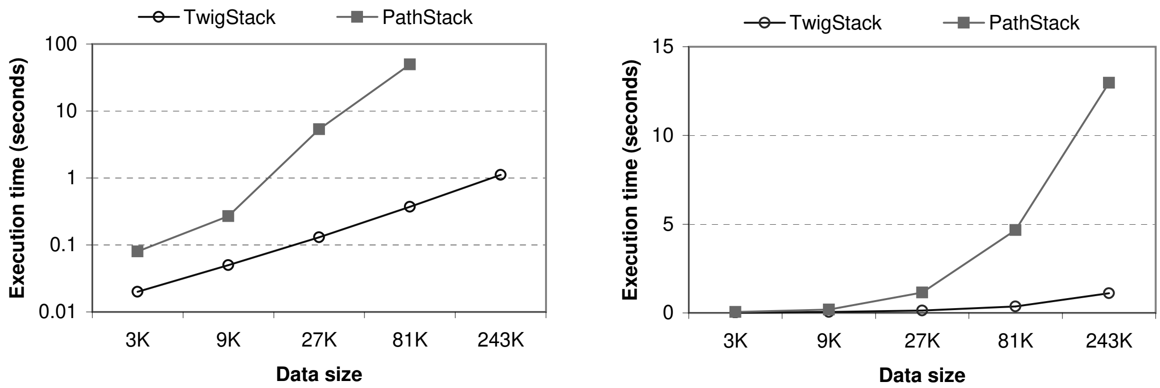
(b) Number of solutions

Figure 19: `PathStack` versus `TwigStack` for a binary twig query.

5.4.1 Ancestor-Descendant Relationships

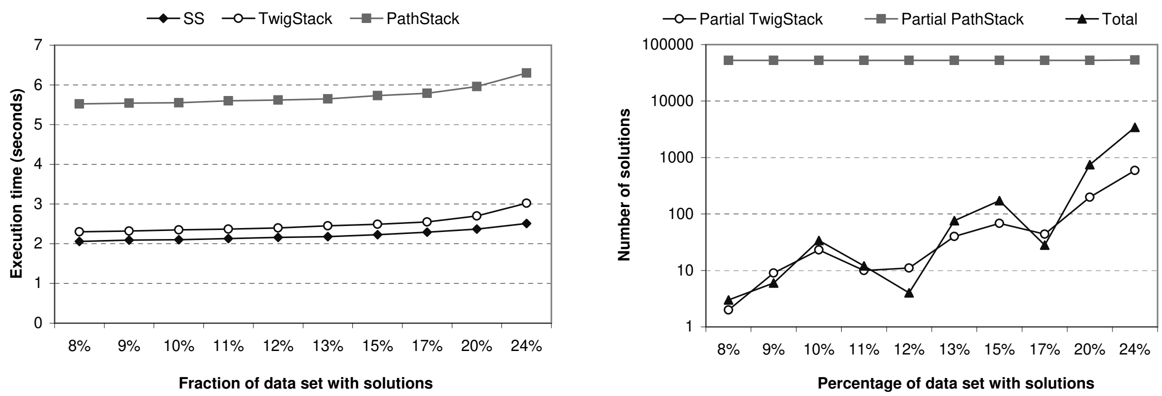
For the first experiment in this section, we used the query shown in Figure 18(a) over different *Random* data sets. Each data set was generated as a full ternary tree. The first subtree of the root node contained only nodes labeled A_1, A_2, A_3 and A_4. The second subtree contained nodes labeled A_1, A_5, A_6 and A_7. Finally, the third subtree contained all possible nodes. Clearly, there are many partial solutions in the first two subtrees but those do not produce any complete solution. Only the third subtree contains actual solutions. We varied the size of the third subtree relative to the sizes of the first two from 8% to 24% (beyond that point the number of solutions became too large). Figure 19 shows the execution time of `PathStack` and `TwigStack`, and the number of partial solutions each algorithm produces before the merging step. The consistent gap between `TwigStack` and `PathStack` results from the latter generating all partial solutions from the first two subtrees which are later discarded in the merge step $(A_1//A_2//A_3//A_4) \bowtie (A_1//A_5//A_6//A_7)$. As seen in Figure 19(b), the number of partial solutions produced by `PathStack` is several orders of magnitude larger than that of the `TwigStack` algorithm. The number of total solutions to the query twig computed by both algorithms is, of course, the same.

For the second experiment, we use the twig query of Figure 18(b) and generated different *Random* data sets in the following way. As before, each data set is a full ternary tree. The first subtree does not contain any nodes labeled A_2 or A_3. The second subtree does not contain any A_4 or A_5 nodes. Finally, the third subtree does not contain any A_6 or A_7 nodes. Therefore, there is not even a single solution for the query twig,



(a) No parent-child edges (b) With parent-child edges

Figure 20: `PathStack` versus `TwigStack` for a ternary twig query.



(a) Execution time (b) Number of solutions

Figure 21: `PathStack` versus `TwigStack` for a parent-child twig query

although each subtree contains a large number of partial solutions. The main difference with the previous experiment is that we need to materialize an intermediate join result before getting the final answer (in the previous query, the result from the first and only merge join was the final answer). Therefore, there is no execution strategy using `PathStack` that avoids materializing a big intermediate result. Figure 20(a) shows the execution time for `PathStack` and `TwigStack` for different data sizes (note the logarithmic scale). For the last data set (with 243K nodes), `PathStack` could not finish, since the intermediate result filled all the available space on disk (2GB).

5.4.2 Parent-Child Relationships and Range Constraints

As explained in Section 4, `TwigStack` is not optimal for parent/child relationships or ancestor/descendant relationships with range constraints. In this section we show that even in this case, `TwigStack` performs much better than using `PathStack`. For that purpose, we modified the queries in Figure 18 adding the following constraint: all ancestor-descendant relationships must be connected by a path with length between one and three (see Figures 22(a) and 22(b)). Figures 20(b) and 21 show the results for these experiments. We can see that even in the presence of parent-child constraints, `TwigStack` is considerably more efficient than `PathStack`. In particular, Figure 21(b) shows that the number of partial solutions produced by `TwigStack`

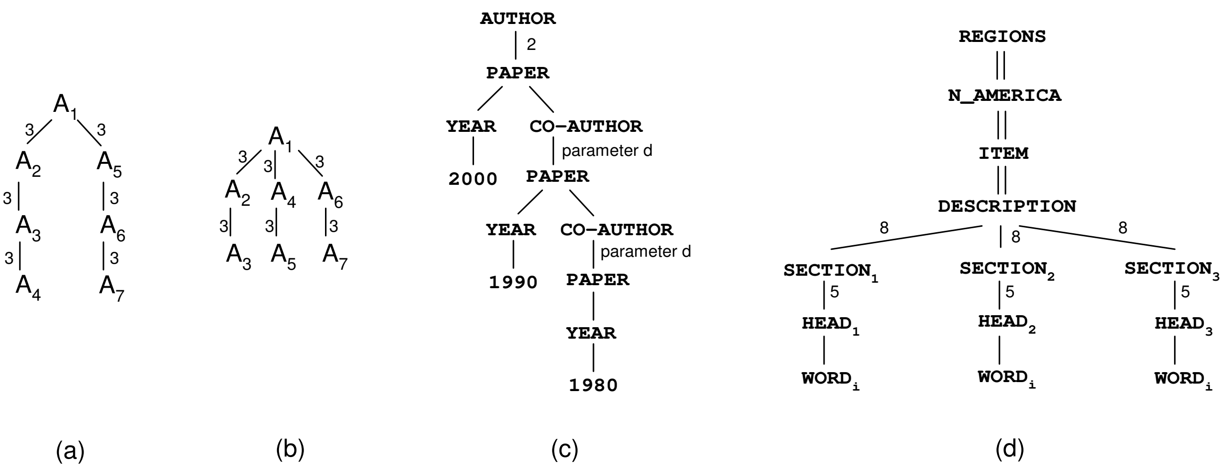
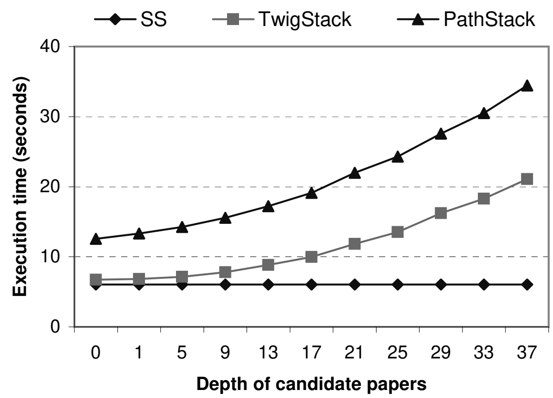
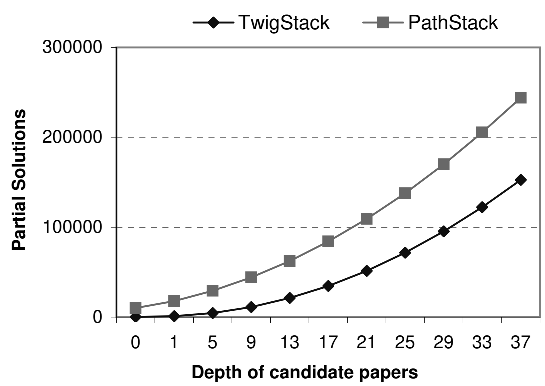


Figure 22: Twig queries used in the experiments of Section 5.4.2.



(a) Execution time



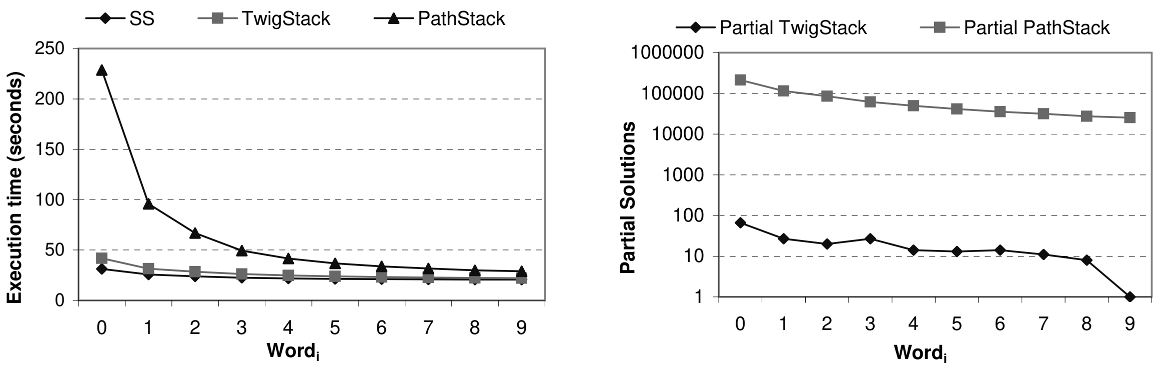
(b) Number of partial solutions

Figure 23: `PathStack` versus `TwigStack` on a real data set

(though not minimal) is small. The non-minimality is evident from the observation that the number of partial solutions produced by `TwigStack` is sometimes much larger than the number of total solutions to the query twig.

We also evaluated the query of Figure 22(c) over the unfolded *DBLP* data set. This query asks for authors with papers published in the year 2000, who have some coauthor with a paper published in 1990, who in turn has some coauthor with a paper in 1980. We vary the allowed depth parameter d in the relationship `COAUTHOR // PAPER`, i.e., the number of coauthors and papers we can traverse from a given author, from 0 (no solutions) to 37. The results are shown in Figure 23. We can see that for these queries, `TwigStack` is again more efficient than `PathStack`.

Finally, we evaluated the query of Figure 22(d) over the *Benchmark* data set. This query asks for items in the region `N_AMERICA` which have descriptions consisting of at least three sections that contain a specific word `WORD-i`. By replacing `WORD-i` with the *i-th* most common word in the data set for $i = 1 \dots 10$, we obtain a family of twig queries. In our *Benchmark* data set there are few situations in which an item description contains each one of the three sections specified in the query. Therefore, although the number of partial solutions is large, there are not many solutions for the query. The results we obtained for this family of



(a) Execution time (b) Number of partial solutions

Figure 24: `PathStack` versus `TwigStack` on the *Benchmark* data set

queries are shown in Figure 24. Again, `TwigStack` is more efficient than `PathStack`, specially for the most frequent words (`WORD-0` to `WORD-4`).

5.5 Sub-Linearity: Using XB-Trees

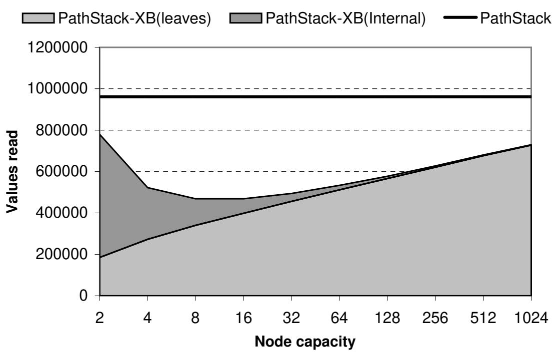
In this section, we study the advantages of using XB-trees to process path and twig queries. In particular, we show that the number of nodes that need to be read from the XB-tree (counting both leaf and internal nodes) is significantly smaller than the size of the input, which causes sub-linear behavior by our algorithm. As we will see, XB-trees with small node capacities can effectively skip many leaf nodes, but the number of internal nodes traversed is large. On the other hand, for large node capacities there are fewer internal node accesses, but XB-trees cannot skip many leaf nodes since otherwise they could miss some solutions. We experimentally obtained the best results when using node capacities ranging from 4 to 64.

For the experiments in this section, we evaluated different queries using `PathStack` and `TwigStack`, with and without XB-trees. We varied the node capacity of the XB-trees between 2 and 1,024 values per index node. Figure 25(a) shows the number of values read in the XB-tree (separated into internal and leaf accesses) for the data set and path queries used in Section 5.3. Similarly, Figure 25(b) shows the results when using the twig query of Figure 18(a) and the data sets of Section 5.4. In turn, Figure 25(c) shows the results for the twig query in Figure 22(c) over the unfolded *DBLP* data set. Finally, Figure 25(d) shows the results for the twig query in Figure 22(d) over the *Benchmark* data set.

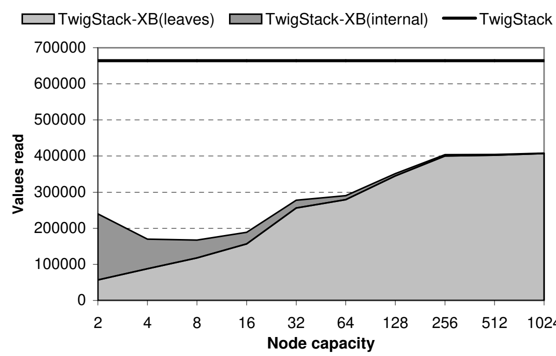
In general, the total number of nodes visited in the XB-tree is consistently smaller than the input data size for a wide range of node capacities. For the *Random* data set, we obtained better results for complex queries. In those situations, XB-trees can prune significant portions of the input data. In contrast, for simpler queries, we need to go deep in the XB-tree nodes, in many cases down to the leaves, since there are many solutions and they are dispersed throughout the whole data set. For data sets with solutions concentrated around certain portions of the data, the impact of XB-trees is more significant, since many internal nodes can be skipped.

6 Related Work

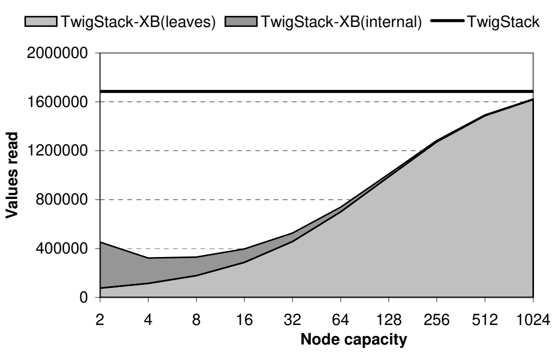
Join processing is central to database implementation [12]. For inequality join conditions, band join [8] algorithms are applicable when there exists a fixed arithmetic difference between the values of join attributes.



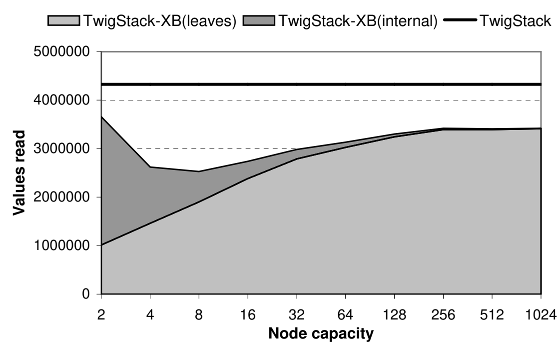
(a) *Random* data, path query



(b) *Random* data, twig query



(c) *DBLP* data, twig query



(d) *Benchmark* data, twig query

Figure 25: Using XB-trees with different queries and data sets

Such algorithms are not applicable in our domain as there is no notion of fixed arithmetic difference. In the context of spatial and multimedia databases, the problem of computing joins between pairs of spatial entities has been considered, where commonly the predicate of interest is overlap between spatial entities [13, 19, 14] in multiple dimensions. The techniques developed in this paper are related to such join operations. However, the predicates considered as well as the techniques we develop are special to the nature of our structural join problem.

In the context of semistructured and XML databases, query evaluation and optimization has attracted a lot of research attention. In particular, work done in the Lore DBMS [20, 15, 16], and the Niagara system [18], has considered various aspects of query processing on such data. XML data and various issues in their storage as well as query processing using relational database systems have recently been considered in [11, 23, 22, 3, 9, 10]. In [11, 23, 10], the mapping of XML data to a number of relations was considered along with translation of a subset of XML queries to relational queries. In subsequent work [22, 3, 9], the authors considered the problem of publishing XML documents from relational databases. Our holistic join strategy for query twig patterns can leverage these previous techniques.

The representation of positions of XML elements `(DocId, StartPos : EndPos, LevelNum)` is essentially that of Consens and Milo, who considered a fragment of the PAT text searching operators for indexing text databases [5, 6], and discussed optimization techniques for the algebra. This representation was used to compute containment relationships between "text regions" in the text databases. The focus of that work was on theoretical issues, without elaborating on efficient algorithms for computing these relationships.

Finally, the recent works of Zhang et al. [26] and Al-Khalifa et al. [1] are closely related to ours. They proposed binary structural join algorithms as primitives for matching query twig patterns. Our Algorithm `PathMPMJ` is a generalization of the MPMGJN algorithm of [26] to match query paths, and Algorithms `PathStack` and `TwigStack` are generalizations of the stack-based algorithms of [1] to match query paths and query twig patterns, respectively.

7 Conclusion

In this paper we developed holistic join algorithms for matching XML query twig patterns, a core operation central to much of XML query processing, both for native XML query processor implementations and for relational XML query processors. In particular, Algorithm `TwigStack` was shown to be I/O and CPU optimal for a large class of query twig patterns, and practically efficient.

There is more to efficient XML query processing than is within the scope of this paper. We have initiated efforts to address some of these issues. One such issue involves the use of axes like `following-sibling` in XPath expressions, in addition to the more commonly used `child` and `descendant` axes (used in this paper to specify twig patterns). How can we compute answers to XPath expressions with such axes? Another issue involves the piecing together of holistic twig joins with value-based joins (including links across documents) to build effective query plans.

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 IEEE International Conference on Data Engineering*, 2002.

[2] S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu XQuery 1.0: An XML Query Language. W3C Working Draft. Available from http://www.w3.org/TR/xquery, Dec. 2001.

[3] M. Carey, J. Kiernan, J. Shanmugasundaram, E. Shekita, and S. Subramanian. XPERANTO: Middleware for publishing object relational data as XML documents. *Proceedings of VLDB*, 2000.

[4] D. D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML query language for heterogeneous data sources. In *WebDB (Informal Proceedings)*, 2000.

[5] M. P. Consens and T. Milo. Optimizing queries on files. In *Proceedings of ACM SIGMOD*, 1994.

[6] M. P. Consens and T. Milo. Algebras for querying text regions. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1995.

[7] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A query language for XML. Available from http://www.w3.org/TR/NOTE-xml-ql., 1998.

[8] D. DeWitt, J. Naughton, and D. Schneider. An evaluation of non equijoin algorithms. *Proceedings of ACM SIGMOD*, 1991.

[9] M. Fernandez and D. Suciu. SilkRoute: Trading between relations and XML. *WWW9*, 2000.

[10] T. Fiebig and G. Moerkotte. Evaluating queries on structure with access support relations. *Proceedings of WebDB*, 2000.

[11] D. Florescu and D. Kossman. Storing and querying XML data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[12] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys, Vol. 25 No. 2*, June 1993.

[13] N. Koudas and K. C. Sevcik. Size separation spatial join. *Proceedings of ACM SIGMOD*, 1997.

[14] M.-L. Lo and C. V. Ravishankar. Spatial hash-joins. *Proceedings of ACM SIGMOD*, 1996.

[15] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record 26(3)*, 1997.

[16] J. McHugh and J. Widom. Query optimization for XML. In *Proceedings of VLDB*, 1999.

[17] U. of Washington. The Tukwila system. Available from http://data.cs.washington.edu/integration/tukwila/.

[18] U. of Wisconsin. The Niagara system. Available from http://www.cs.wisc.edu/niagara/.

[19] J. M. Patel and D. J. DeWitt. Partition based spatial merge join. *Proceedings of ACM SIGMOD*, 1996.

[20] D. Quass, J. Widom, R. Goldman, H. K, Q. Luo, J. McHugh, A. Rajaraman, H. Rivero, S. Abiteboul, J. Ullman, and J. Wiener. LORE: A lightweight object repository for semistructured data. *Proceedings of ACM SIGMOD*, page 549, 1996.

[21] G. Salton and M. J. McGill. *Introduction to modern information retrieval.* McGraw-Hill, New York, 1983.

[22] J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as XML documents. In *Proceedings of VLDB*, 2000.

[23] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proceedings of VLDB*, 1999.

[24] XMach-1. Available from http://dbs.uni-leipzig.de/en/projekte/XML/XmlBenchmarking.html.

[25] The XML benchmark project. Available from http://www.xml-benchmark.org.

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