Twig[2]Stack: Bottom-up Processing of Generalized-Tree-Pattern Queries over XML Documents

Songting Chen[1], Hua-Gang Li[2]*, Junichi Tatemura[1]
Wang-Pin Hsiung[1], Divyakant Agrawal[1], K. Selçuk Candan[1]

[1]NEC Laboratories America, 10080 North Wolfe Road, Suite SW3-350, Cupertino, CA 95014
[2]Department of Computer Science, University of California, Santa Barbara, CA 93106
(songting,tatemura,whsiung,agrawal,candan)@sv.nec-labs.com, huagang@cs.ucsb.edu

ABSTRACT

Tree pattern matching is one of the most fundamental tasks for XML query processing. Holistic twig query processing techniques [4, 16] have been developed to minimize the intermediate results, namely, those root-to-leaf path matches that are not in the final twig results. However, useless path matches cannot be completely avoided, especially when there is a parent-child relationship in the twig query. Furthermore, existing approaches do not consider the fact that in practice, in order to process XPath or XQuery statements, a more powerful form of twig queries, namely, Generalized-Tree-Pattern (GTP) [8] queries, is required. Most existing works on processing GTP queries generally calls for costly post-processing for eliminating redundant data and/or grouping of the matching results.

In this paper, we first propose a novel *hierarchical stack encoding* scheme to compactly represent the twig results. We introduce `Twig2Stack`, a bottom-up algorithm for processing twig queries based on this encoding scheme. Then we show how to efficiently enumerate the query results from the encodings for a given GTP query. To our knowledge, this is the first GTP matching solution that avoids *any* post path-join, sort, duplicate elimination and grouping operations. Extensive performance studies on various data sets and queries show that the proposed `Twig2Stack` algorithm not only has better twig query processing performance than state-of-the-art algorithms, but is also capable of efficiently processing the more complex GTP queries.

1. INTRODUCTION

The rich content and the flexible semi-structure of XML documents demand efficient support for complex declarative queries. XML documents can be viewed as ordered tree structures where each tree node corresponds to document ELEMENTS(ATTRIBUTES) and edges represent parent-child (element→sub-element) relationships. Figure 1 depicts one sample XML document tree. Common XML query languages, such as XPath [21] and XQuery [22], issue *structural queries* over the XML data. Due to their significance to many practical applications, efficient processing of such structural queries has received significant attentions from both academic and industrial communities [3, 4, 16, 23]. One of the most common structural queries is the *tree (twig) pattern* query. A sample tree pattern query is shown on the right side of Figure 1. Here a document element a can be a match to query node A when it has path matches for *both* $//A/B//D$ and $//A/B/C$ [1].

Efficient matching of tree pattern queries over XML data is one of the most fundamental challenges for processing XQuery [13]. Most existing works on processing twig queries decompose the twig queries into paths and then join the path matches [3, 23]. This approach may introduce very large intermediate results. Consider the sample XML document tree and a tree pattern query in Figure 1. For instance, the path match $(a1, b4, d4)$ for path $//A/B//D$ does not lead to any final result since there is no child C node under $b4$. To solve this problem, *holistic twig pattern matching* [4, 16] has been developed in order to minimize the intermediate results, i.e., only to enumerate those root-to-leaf path matches that will be in the final twig results. However, when the twig query contains parent-child relationship, these solutions may still generate large numbers of useless matches [4, 16]. In this paper, we will show that the explicit join of individual root-to-leaf path matches or even the enumeration of these path matches are *not* necessary for processing twig queries.

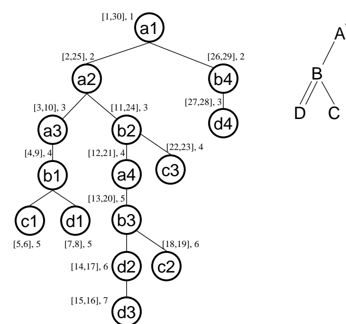

Figure 1: A Sample XML Document Tree and A Twig Query

Yet another challenge for processing XQuery is that there may be multiple path expressions in the FOR, LET, WHERE and RETURN clauses, all with *different* semantics. Existing work shows that it is better to consider the matching of these expressions as a whole in terms of a *generalized tree pattern* (GTP) [8] in order to avoid

*The work has been done during the author's internship at NEC Laboratories America.

[1]In this paper, we assume the document elements with labels in lower-case letter match the query nodes with labels in the corresponding upper-case letter. For instance, $a2$ and $a3$ match node A.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '06, September 12-15, 2006, Seoul, Korea.
Copyright 2006 VLDB Endowment, ACM 1-59593-385-9/06/09.

repetitive work. Figure 2 depicts two sample XQuery statements and their respective GTPs. In $XQuery_1$, D is not a return node, i.e., only its existence is of interest. In XQuery_2, node C is optional (in general, any expression in the LET or RETURN clauses is optional) in the sense that a B element can be a match even without any descendant C elements. Any matching C elements, however, must be *grouped* together under their common B ancestor element.

Most existing works on holistic twig query processing focus only on returning the entire twig results [4, 14, 16]. In practice, however, returning the entire twig results is seldom necessary for either XPath or XQuery and may consequently cause duplicate elimination or ordering problems (Section 2). Moreover, many XQuery statements in practice require grouping the results [8]. Applying post-duplicate elimination, sorting and grouping operations to address these problems has already been shown to be expensive in many existing works [8, 10].

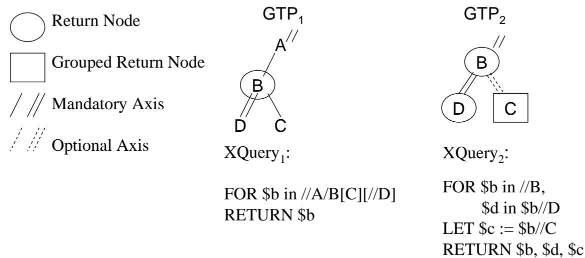

Figure 2: Generalized Tree Pattern and XQuery

In this paper, we provide a comprehensive solution to tackle the above challenges. In summary, the main contributions are:

- We propose a novel *hierarchical stack encoding* scheme to compactly represent the twig results. This scheme also reduces the complexity for twig query processing.
- Based on this encoding scheme, we propose `Twig`^{2}`Stack`, a bottom-up query processing algorithm for a given *generalized tree pattern (GTP)* [8], which is a fundamental building block for XQuery processing.
- Then we show how to efficiently enumerate the GTP query results from the encodings. To our knowledge, this is the first GTP matching solution that is free of any post path-join, sort, duplicate elimination, and grouping operations.
- We propose an early result enumeration mechanism by using a hybrid of top-down and bottom-up computation method to reduce the runtime memory usage.
- Extensive performance studies on various data sets and queries show that our `Twig`^{2}`Stack` algorithm not only has better performance for twig query processing than existing works, such as `TwigStack` [4] and `TJFast` [16], but is also capable of efficiently processing the more complex GTP queries.

2. DATA MODEL AND QUERY LANGUAGE

An XML document is modeled as a nested structure of *elements*. The scope of an element is defined by its start-tag and end-tag. An example XML document tree is demonstrated in Figure 1.

The common query languages over XML are XPath [21] and XQuery [22]. One fundamental task for processing XPath and XQuery is to match twig patterns queries. The concept of *generalized twig pattern (GTP)* is introduced in [8] to consider the evaluation of an XQuery as a whole to avoid repetitive work.

We now give a brief review of GTP. As shown in Figure 2, GTP query may have solid and dotted edges, representing mandatory and optional structural relationships, respectively. In this paper, we consider parent-child (PC) and ancestor-descendant (AD) relationships. The mandatory semantics corresponds to those path expressions in the FOR or WHERE clauses. The optional semantics corresponds to those path expressions in the LET or RETURN clauses. For a given GTP, not all nodes are return nodes. For the path expressions in the FOR clause, only the last node is the return node. One example is the B node of GTP_1 in Figure 2. For the path expression in LET or RETURN clause, we may need to group the matching elements under their common ancestor element. One example is the C node of GTP_2 in Figure 2. Please refer to [8] for detailed definition of GTP and how to translate XQuery into GTPs.

These rich semantics introduce new challenges for handling the *duplicates* and *ordering issues*. We now briefly review how the query results are generated when there are non-return nodes in the GTP query through the following three examples. Consider the document tree in Figure 1. (i) For path query $//B//D$, let us first assume B and D are both return nodes. The final matches are $(b1, d1)$, $(b2, d2)$, $(b2, d3)$, $(b3, d2)$, $(b3, d3)$ and $(b4, d4)$. (ii) Now let us assume D is the only return node. In this case, the results should be $(d1)$, $(d2)$, $(d3)$ and (d4). Clearly, if we were to generate the distinct path matches first as in the first case, duplicate elimination becomes unavoidable. (iii) Lastly, let us consider path query $//A/B$ where B is the only return node. The results are $(b1)$, $(b2)$, $(b3)$ and $(b4)$. This order is different from the order for the entire path matches, namely, $(a1, b4)$,$(a2, b2)$,$(a3, b1)$ and $(a4, b3)$. If we were to generate these entire path matches first, sorting these B elements becomes unavoidable.

In this paper, we use the region encoding for the XML document, which is widely used in XML query processing [4, 19, 23]. Region encoding associates each XML document element with a 3-tuple *[LeftPos, RightPos], Level*. Here *Level* is the depth of the element in the document tree. *LeftPos* and *RightPos* are both integers. Given any two document elements, e_1 and e_2, e_1 is e_2's ancestor *iff* $e_1.LeftPos < e_2.LeftPos$ and $e_2.RightPos < e_1.RightPos$. Furthermore, if $e_1.Level = e_2.Level - 1$, then e_1 is e_2's parent. This encoding allows efficient structural checking between two document elements. Figure 1 also includes the region encodings.

3. EFFICIENT GTP PROCESSING

In this section, we propose the `Twig`^{2}`Stack` algorithm for processing a GTP query. We start with an in-depth study of the existing `PathStack` [4] algorithm. Our `Twig`^{2}`Stack` algorithm is inspired by the similar principles.

3.1 Motivation

Bruno et.al [4] proposed a novel path matching algorithm, called `PathStack`, for processing linear path expressions. Consider the path query $//A/B//D$ and the data path $a1, a2, b2, a4, b3, d2, d3$ in Figure 1. The entire path query is processed in a top-down fashion by visiting the document elements in pre-order. First, each query node E is associated with a stack, $S[E]$. The algorithm pushes the document element into the stack *iff* the relationship between this element and the *top* element in its parent stack satisfies the axis requirement in the query. Once a document element is pushed into the leaf stack, `PathStack` algorithm knows that there must be some answers to this path query. Figure 3 shows how this query is processed given the above input. The result enumeration is done in reverse, i.e., starting from the elements in the leaf query nodes. For example, $d3$ points to $b3$ and implicitly to $b2$ as well since the axis is AD. $b3$ and $b2$ point to $a4$ and $a2$, respectively. So

the path matches for $d3$ are $(a2, b2, d3)$ and $(a4, b3, d3)$.

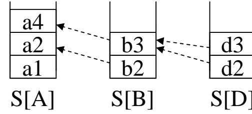

Figure 3: PathStack: Path-based Stack Encoding

The following main observation can be made from this algorithm. That is, the partial/complete path results are encoded by recording the AD relationships either 1) between the elements in different query stacks by using explicit edges, or 2) between the elements in the same stack, which are implicitly recorded based on their positions in the stack. Note that this partial/complete path result encoding scheme not only *avoids to generate large intermediate results*, but also *reduces the query processing cost*. The reason is that only the *top* element in the parent stack needs to be checked. In Figure 3, $d2$ only needs to consider its relationship to $b3$ not $b2$.

In [4], the authors further proposed the `TwigStack` algorithm for holistic processing of twig queries. The basic idea is to output a matching path only when it contributes to the final twig results. This is done by checking the node indexes to make sure all the branches are satisfied. However, this approach is shown to be not optimal for twig queries with PC edges, i.e., the resulting root-to-leaf path matches are not guaranteed to be in the final twig results.

Inspired by the `PathStack` algorithm, our goal is to process twig queries with similar principles, i.e., we need an effective mechanism to encode partial/complete twig results in order to minimize the intermediate results and to reduce the query processing cost. In particular, we need to record the AD relationships between the document elements in different query nodes as well as those in the same query node in order to minimize the intermediate results. While we can use explicit edges to record the AD relationships for the document elements in different query nodes (similar to `PathStack`), for the elements in the same query node, we find that they naturally form a *tree structure* as well. For example, in Figure 1, both $a2$, $a3$ and $a4$ satisfy the twig query node A. Here $a2$ is an ancestor for both $a3$ and $a4$, while $a3$ and $a4$ have no AD relationship. Based on this observation, we propose to organize the elements that match the same query node in a *hierarchical* structure to explicitly capture their AD relationships.

In this paper, we show that such a hierarchical structure can be produced based on the *post-order* document traversal. Intuitively, for any document element e, when we visit e during pre-order traversal, we can only determine if e satisfies a *prefix path* query from root to query node E. When we visit e during post-order traversal, we can determine if e satisfies a *sub-twig* query rooted at query node E. The latter is based on the fact that only when the entire subtree of e is visited, can we determine if e has required descendant elements. Hence in order to encode the twig results, a post-order document traversal is required.

3.2 Hierarchical Stack Encoding

3.2.1 Notations and Data Structures

In this section, we introduce a *hierarchical stack encoding* scheme to capture the AD relationships for the elements in the same query node as motivated in the previous section.

For each query node N of a twig query Q, we associate it with a *hierarchical stack* $HS[N]$. Each hierarchical stack $HS[N]$ consists of an ordered sequence (the sequence order will be described shortly) of *stack trees* ST. A stack tree ST is an ordered tree (the tree order will be described shortly), where each tree node is a *stack* S. For example, in Figure 4, $HS[A]$ contains one stack tree, while $HS[D]$ contains three stack trees.

Each stack S contains zero or more document elements. The AD relationship between the document elements in a stack tree ST is implicitly captured as follows: *one document element is an ancestor for all elements below in the same stack and is also an ancestor for all elements in its descendant stacks.* Note that any two elements have no AD relationship if their corresponding stacks have no AD relationship. Consider $HS[A]$ in Figure 4, $a2$ is ancestor for both $a3$ and $a4$, while $a3$ and $a4$ have no AD relationship.

In order to create the hierarchical structure among stacks when visiting the document elements in the post-order, we associate each stack S with a region encoding, similar to that for a document element. The *LeftPos* for a stack S is defined as the smallest *LeftPos* among all the elements in stack S and all of S's descendant stacks. The *RightPos* for a stack S is defined as the largest *RightPos* among all the elements in stack S and all of S's descendant stacks. For instance, in Figure 4, the top stack of $HS[B]$ has region encoding $[4, 24]$ (Level is not used for stacks), where 4 is the smallest $LeftPos$ and 24 is the largest $RightPos$ among its descendant elements. The region encodings for other stacks are shown in the figure. Next, the region encoding for a stack tree ST is the same as the encoding of ST's root stack. Finally, for a given hierarchical stack $HS[N]$, its stack trees are ordered based on their $RightPos$. For a given stack S, its child stacks are also ordered based on their $RightPos$.

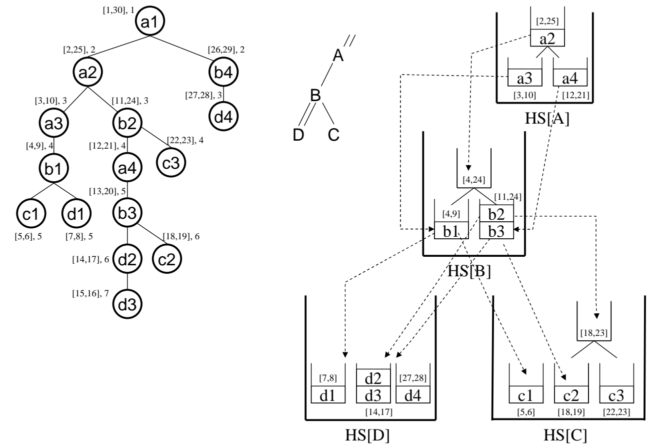

Figure 4: Hierarchical Stack Encoding

3.2.2 Creating Hierarchical Stacks through Merging

Since the leaves are visited before the root in post-order document traversal, the hierarchical structure is built in a bottom-up manner. We now introduce a *merge* operation for creating the hierarchy among stacks. The merge operation is leveraged to combine multiple stack trees into a single one. Intuitively, this means that the entire *sub-tree* of e has been visited. Figure 5 shows how the stack trees in $HS[A]$ in Figure 4 are created.

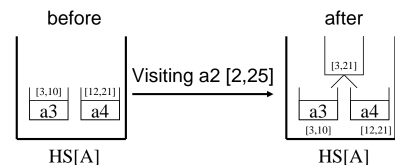

Figure 5: Example of Merge Operation

EXAMPLE 1. *During the post-order traversal of the XML document in Figure 1, we visit a3, a4 and then a2. The hierarchical stack $HS[A]$ before visiting a2 in Figure 5 is on the left side. When visiting a2, we find that a2 is an ancestor of both stacks (we extend the notion of AD relationship between document element and stack trees based on their encodings). Hence, a new merged stack is created.* □

Figure 6 depicts the pseudo code of the *merge* algorithm. Here *createMergedStackTree* (line 12) creates a new stack and lets all stack trees in STS (if more than one) be its children. Line 5-10 is to process one query step, which will be described in the next section. The newly created or merged stack tree must always have the largest *RightPos*. Hence, their order is naturally maintained.

```
Boolean merge (HierarchicalStack HS[M], docElement e, Axis axis)
Boolean Satisfied = FALSE;
StackTreeSet STS = empty;
1.  BEGIN
2.    FOR each stack tree ST of HS[M]
                    //Visit in descending order of ST.RightPos
3.      IF ST.RightPos < e.LeftPos
4.        break; //No need to keep visiting more stack trees;
5.      IF axis = PC AND ST.top.Level = e.Level+1
6.        Satisfied = TRUE;
7.        addPCEdge(e, M, ST.top);
8.      ELSE IF axis = AD
9.        Satisfied = TRUE;
10.       addADEdge(e, M, ST.top);
11.     STS = STS ∪ ST;
12.   createMergedStackTree(STS);
13.   return Satisfied;
14. END
```

Figure 6: Pseudo Code of Merge Algorithm

LEMMA 1. *Assume that for a given document element e, the stack trees ST_1, ST_2, ..., and ST_n are merged and a new root stack ST_{n+1} is created. For any document element e' visited during the rest of the post-order document traversal, it will be either an ancestor of* all *ST_1, ST_2, ..., and ST_n or* none *of them.*

Lemma 1 immediately follows that while e' may be pushed into ST_{n+1}, denoting its AD relationships to all ST_i, it will *never* be pushed into any $ST_i (i = 1..n)$. In Example 1, any future element must be either the ancestor of *both* $a3$ and $a4$ or none of them.

3.3 Bottom-up Twig Query Processing

3.3.1 Overview

In this section, we present Twig²Stack algorithm for processing GTP queries based on the hierarchical stack encoding scheme [2]. Recall that Twig²Stack algorithm visits the document elements in post-order. This can be easily achieved even when the document elements are indexed in document-order as in [4]. That is, we maintain a global stack for the document elements that are on the same path. Given a document element e visited in pre-order (with the minimum $LeftPos$), we pop up all the elements in the stack that are not $e's$ ancestors. Then we push e into the stack. The popped document elements are in post-order (line 2,3,6 in Figure 7).

[2]As can be seen in Figure 4, for a given twig query, all the hierarchical stacks form a tree structure. At the same time, the internal of a hierarchical stack also forms a tree structure. We thus name our algorithm Twig²Stack.

The essential idea of Twig²Stack is as follows. Given a document element e, we push it into a hierarchical stack $HS[E]$ (with the matching label, i.e., either the same label or wildcard '*') *iff* it satisfies the sub-twig query rooted at this query node E. Only E's child query nodes M need to be checked due to the fact that all the elements in $HS[M]$ must have already satisfied the sub-twig query rooted at M. Finally, the hierarchical stack structure (as in Section 3.2) is maintained using the *merge* algorithm in Figure 6 either *when checking one query step* or *when pushing one document element into the hierarchical stack*.

Maintaining the hierarchical structure among stacks has a critical impact on efficient processing of twig queries. It serves *multiple* purposes. First, it encodes the partial/complete twig results in order to minimize the intermediate results as motivated in Section 3.1. Second, it reduces the query processing cost as we will describe in the next section. Third, it enables efficient result enumeration as we will describe in Section 4.

3.3.2 Checking One Query Step

Given a document element e, we push it into $HS[E]$ *iff* all the query steps to E's child nodes M have been satisfied. Due to the nature of post-order traversal, all e's descendant elements must have already been visited and have been pushed into $HS[M]$ if satisfied. Hence, the checking of one query step $E \rightarrow M$ for e can be done through the *merging* of $HS[M]$.

Assume that there are n stack trees in $HS[M]$, namely, ST_1, ..., ST_n. First, let us assume none of the ST_i $(i = 1..n)$ is e's descendant, since the entire sub-tree of e is visited due to the post-order traversal, we can safely conclude that e cannot satisfy E.

Next, let us assume ST_p,...,ST_n $(1 \leq p \leq n)$ are e's descendants. We denote $ST_i.top$ as the top element of the root stack of stack tree ST_i. Note that $ST_i.top$ may be empty if the top stack does not contain any element. In this case, (1) when the query step $E \rightarrow M$ is an AD relationship, then all the elements in stack trees ST_p,...,ST_n, satisfy this query step. We encode the results of this query step by creating edges from e to $ST_p.top$,...,$ST_n.top$. Such an edge implicitly means that $ST_i.top$ and the elements in the descendant stacks are e's descendants. (2) When the query step $E \rightarrow M$ is a PC relationship, obviously only $ST_p.top$,...,$ST_n.top$ might be child of e. We then create edges to those top elements if their $Level$ equals $e.Level + 1$. By Lemma 1, we guarantee that these edges encode all possible answers from e for this query step, since no elements will ever be pushed below $ST_p.top$,...,$ST_n.top$ during the rest of the document traversal.

Finally, after this query step checking phase, we merge the stack trees ST_p,...,ST_n by creating a new root stack. The primary purpose of this stack tree merging step is to *reduce* the future query processing costs, since only this new root stack needs to be considered for the remaining document elements. The pseudo code for checking one query step is in line 5-10 in Figure 6.

EXAMPLE 2. *In Figure 4, when visiting $a2$, we merge the stack trees in $HS[B]$ while checking the PC axis (create one edge to $b2$)* [3]. *When visiting $a1$, we only need to check the top element in $HS[B]$ and conclude that its $//A/B$ step is not satisfied.* □

3.3.3 Twig²Stack Algorithm and Analysis

Putting together, Figure 7 depicts the pseudo code of Twig²Stack algorithm. Given a document element e visited in post-order, we first check if e can be pushed into its corresponding hierarchical stack $HS[E]$, or in other words, if all E's child axes have been satisfied (line 3-4 in *MatchOneNode*) as described in Section 3.3.2.

[3]The edges shown in Figure 4 are conceptual for easy understand-

```
Procedure Twig2Stack(docElement e)
Stack docPath;
docElement currentElem;
1. BEGIN
2.   WHILE docPath not empty AND docPath.top is not e's ancestor
3.     currentElem = docPath.pop();
4.     FOR each query node E with matching label of currentElem
5.       MatchOneNode(currentElem, HS[E]);
6.   docPath.push(e);
7. END

Procedure MatchOneNode (docElement e, HierarchicalStack HS[E])
Boolean Satisfied;
1. BEGIN
2.   Satisfied = TRUE;
3.   FOR each child query node M of E & Satisfied
4.     Satisfied = merge(HS[M], e, axis(E→M));
5.   IF Satisfied
6.     merge(HS[E], e, "");
7.     push (HS[E], e);
8. END
```

Figure 7: Pseudo Code of Twig^{2}Stack Algorithm

Once e satisfies all the axes requirements for query node E, we push e into the hierarchical stack $HS[E]$. Meanwhile, we also need to maintain the hierarchical structure of the elements in $HS[E]$. To achieve this, we also merge the stack trees in $HS[E]$ based on e (line 6 in *MatchOneNode*) and push e to the top of the merged stack (line 7). When there is no existing stack tree that is the descendant of e, a new stack will be created to hold e.

Finally, we show how easily the optional axis (Section 2.1) can be supported. We push an element into the stack *iff* all its mandatory axes are satisfied, while we create edges for both mandatory and optional children. This avoids a potentially expensive *left-outer-join* operation as required in prior work [8]. AND/OR twig query [14] can also be easily supported. Figure 4 depicts a running example of the `Twig`^{2}`Stack` algorithm.

THEOREM 1. *For any document element e, it is pushed into $HS[E]$ iff it satisfies the sub-twig query rooted at E.*

PROOF. *1) "→": The proof is straightforward based on the dynamic programming nature of the* `Twig`^{2}`Stack` *algorithm.*

2) "←": If E is a leaf query node, then any document element e with matching labels satisfies this query node and will be pushed into $HS[E]$. The theorem is trivially true. For a non-leaf query node E, we prove the theorem by contradiction.

Assume one element e satisfies E but is not in $HS[E]$. Then at least one query step $E \rightarrow M$ failed when merging $HS[M]$. Since e satisfies E, there must exist one element m which satisfies M and the structural relationship between e and m satisfies the query step $E \rightarrow M$. There can be two reasons why the merging of $HS[M]$ failed. They are either (i) m in $HS[M]$ however the structural relationship between e and m is not captured through merging to satisfy the query step. Clearly, m must reside in one stack tree in $HS[M]$ and e must be able to find this stack tree as its descendant - AD relationship is thus satisfied. If m is the child of e, then m must be at the top of one stack tree - PC relationship is thus satisfied. Hence, case (i) is not possible. (ii) m is not in $HS[M]$, or in other words, m does not satisfy M. By applying the same reasoning, we can conclude that there must exist one p element that satisfies query node P (P is M's child query node) but not in $HS[P]$. Eventually when reaching the leaf query node, as stated before, all the elements are satisfied and must be in the corresponding hierarchical stack. Hence, case (ii) is also not possible. □

ing. In implementation, we create multiple edges as in Figure 6.

3.4 Space Complexity, Memory Requirement

Theorem 2 provides the worst case space and time complexity of `Twig`^{2}`Stack`. Most existing work on twig processing requires enumerating the root-to-leaf path matches [4, 14, 16]. The number of path matches in the worst case is exponential in terms of the size of query $O(|D|^{|Q|})$. In `Twig`^{2}`Stack`, all path matches are always encoded (until the result enumeration phase − in fact, some path matches need not ever be enumerated for GTP queries).

THEOREM 2. *For a given twig query Q and an XML document D, both the space and time complexity of* `Twig`^{2}`Stack` *algorithm in Figure 7 are $O(|D||B|)$, where $|B|$ is the maximum of 1) B_1 =max(number of query nodes with the same label in Q) and 2) B_2 = max(total number of children of query nodes with the same labels in Q). Obviously, $|B| \leq |Q|$.*

PROOF. *Sketch. We show the case when the query have distinct labels. There are two costs in* `Twig`^{2}`Stack`*, namely, the cost for merging stacks and the cost for checking all branches of one query node. For the merge cost, assume that there are n elements in one hierarchical stack. It is easy to show that the merge cost is $O(n)$, since once the stack trees are merged, they need not be considered for merging again. When all query nodes have distinct labels, obviously the total merge cost is $O(|D|)$. The branch checking cost is bounded by the maximum fan-out of the query nodes.* □

Note that the memory requirement of `Twig`^{2}`Stack` is higher than `TwigStack`. In `TwigStack`, the memory requirement is just proportional to the document depth. In comparison, `Twig`^{2}`Stack` may keep the entire document in the memory in the worst case.

However, we claim that the worst case will not practically happen unlike the traditional main memory XML database that always stores the entire DOM tree in the memory before processing [1]. `Twig`^{2}`Stack` only stores document elements that satisfy some part of the twig query at *runtime*. More specifically, there are the following constraints on the data to be stored. First, only the elements that have labels matching with the query need to be stored. The XMark [11] dataset, for example, has 77 distinct labels. It is unlikely that a single query is interested in all these labels [4]. Second, when the selectivity of a twig is high, only small portion of elements will be pushed in the hierarchical stacks. Third, when value predicates are in the query, evaluation of them during the traversal will also contribute decreasing the number of elements pushed. Hence, it is unlikely to keep the entire document in the memory in practice. In Section 4.4, we will further describe a mechanism, called *early result enumeration*, that can significantly reduce the memory requirement for most practical queries.

3.5 Optimization for Non-Return Node

When there are non-return nodes in a GTP query, which is very common in XPath or XQuery, we can further optimize the space and computation costs. Here we call a query node N is an *existence-checking* node *iff* 1) N is not a return node and 2) there is no return node below N. When a query node N is an *existence-checking* node, only *the root stack* and *its top element* of each stack tree need to be retained. The reason is that, at any time, the parent query node only needs to check the top element (or root stack) and the existence of such a top element (or root stack) suffices. Hence,

[4]A wildcard in the query can potentially match any labels. Nonetheless, the other two constraints and the early result enumeration mechanism still apply.

once the stack trees are merged, they are no longer useful. Also we can avoid creating any edges to an *existence-checking* node.

Figure 8 depicts this optimization based on the XML document and query in Figure 4. B is the only return node. In this case, both C and D are *existence-checking* nodes. A is not an *existence-checking* node although it is a non-return node. The reason is that we cannot simply throw the elements in $HS[A]$ away since they serve as *bridges* to the return node B for result enumeration (Section 4). For *existence-checking* nodes, such as C and D, any descendant stacks or non-top elements can be safely removed as the shaded rectangles shown in the figure. For PC relationship, such as C, the child elements can be discarded after visiting their parents. We need not create any edges to C, D as well.

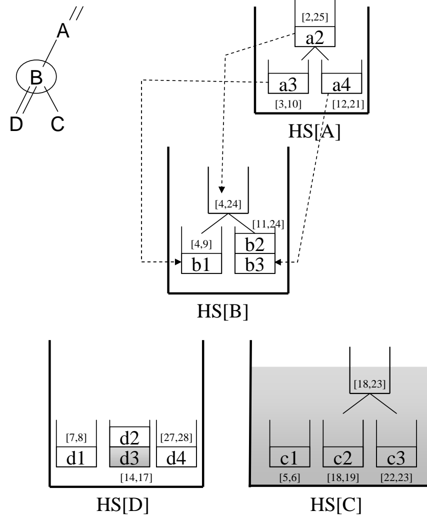

Figure 8: Optimization for Existence-Checking Query Nodes

4. ENUMERATION OF GTP RESULTS

In the previous section, we describe `Twig`^{2}`Stack` algorithm for bottom-up twig query processing and partial/complete twig results encoding. In this section, we propose an efficient solution to enumerate the query results for a GTP query that are *duplicate free* and preserve *document order* from the encodings. We first assume that we do not enumerate the query results until the entire document has been processed by the `Twig`^{2}`Stack` algorithm. In Section 4.4, we describe how to enumerate the results earlier and the memory consumed by the hierarchical stacks can also be freed up.

Before we proceed, we define two functions, namely, *pointAD*(e, $HS[M]$) and *pointPC*($e, HS[M]$), where e is a match of E and M is one child node of E. *pointPC*($e, HS[M]$) returns all the elements in $HS[M]$ that satisfy the PC relationship with e, while *pointAD*(e, $HS[M]$) returns all the elements in $HS[M]$ that satisfy the AD relationship with e. Clearly, *pointPC* is the same as the edges created by the *merge* algorithm in Figure 6. *pointAD* is computed by *expanding* these edges. Such expansion simply returns all the descendant elements with respect to one edge. In Figure 4, we have *pointPC(a2, HS[B])* = {*b2*}, *pointPC(a3, HS[B])* = {*b1*}, *pointAD(b1, HS[D])* = {*d1*} and *pointAD(b2, HS[D])* = {*d2,d3*}.

4.1 Source of Duplicates and Out-of-Orderness

When dealing with a GTP which may contain *non-return* nodes, duplicate and out-of-order results may occur. Such phenomena can be easily explained under our hierarchical stack encoding scheme.

EXAMPLE 3. *In Figure 4, first we assume that only D is a return node. We have pointAD(b2, HS[D])* = {*d2,d3*} *and pointAD(b3, HS[D])* = {*d2,d3*}. *The latter generates only duplicates. In this case, b3 is descendants of b2.*

Second, we assume that only B is a return node. We have pointPC(a2, HS[B]) = {*b2*}, *pointPC(a3, HS[B])* = {*b1*} *and pointPC(a4, HS[B])* = {*b3*}, *while the correct return order should be* {*b1, b2, b3*}. *This output order is no longer consistent with the order of their parents. In this case, b2 is an ancestor of a4 and is thus an ancestor of any elements in pointPC(a4, HS[B]).* □

The above example shows that the duplicate problem occurs when the two elements with AD relationship point to the same descendant element, while the out-of-order problem occurs when the two elements with AD relationship point to their respective child elements that no longer preserve order. This observation suggests that if we maintain the elements returned by *pointPC* and *pointAD* in an *ordered sequence of trees (in pre-order)* (SOT) structure, i.e., maintain their structural relationships, we may solve the duplicate and out-of-order problems. Such SOT structure is already preserved in the hierarchical stack and can be easily produced.

4.2 Computing Total Effects for Non-return Nodes

As described above, the duplicate and out-of-order problems may occur when handling non-return nodes. We formulate this as the problem of computing the *total effects* for a non-return node below.

Problem statement: Assume a non-return query node E and its child query node M. For a given set of elements $eSOT$ in $HS[E]$ maintained in sequence of trees *(SOT) format, we want to compute its* total effects *on the query node M, namely, a set of elements $resultSOT$ in $HS[M]$ maintained also in SOT format, with each element in $resultSOT$ having at least one element in $eSOT$ that satisfies the query step $E \rightarrow M$.*

When the query step $E \rightarrow M$ is an AD relationship, clearly only the *root* element of each tree in $eSOT$ needs to be considered. The final $resultSOT$ is simply a union of all *pointAD*($root, HS[M]$). All other elements in $eSOT$ are *guaranteed* to only generate duplicates as shown in the first case in Example 3.

When the query step $E \rightarrow M$ is a PC relationship, a naïve way to handle the order problem is to sort all the elements in *pointPC*(e, $HS[M]$) for all e in $eSOT$. In fact, sorting is not necessary since all the elements e in $eSOT$ and their child elements in $pointPC(e, HS[M])$ already preserved their respective document order by the `Twig`^{2}`Stack` algorithm.

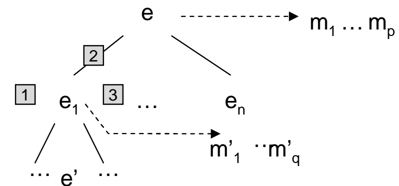

Figure 9: Intuition of Computing Total Effects under PC Axis

Figure 9 provides basic intuition regarding how this order problem can be resolved. Assume that one element e with children e_1,...,e_n in an SOT tree and *pointPC*($e, HS[M]$) equals m_1,...,m_p. Both e_1,...,e_n and m_1,...,m_q are in document order. Starting from e_1 and m_1, there are three possible positions for m_1. (1) m_1 is on the *left* side of e_1. In this case, we should add m_1 into *resultSOT* since there will be no other result element that appears before m_1 in the document order or is a descendant of m_1. (2) m_1 is

an *ancestor* of e_1[5]. In this case, m_1 must also be an ancestor for all *pointPC*$(e_1, HS[M])$ and all *pointPC*$(e', HS[M])$, where e' is any descendant of e_1 in $eSOT$. Assume that we recursively compute the *total effects* of e_1 and all its descendant elements e' as SOT_1. A new SOT tree will be created with m_1 being the root and SOT_1 being its children. (3) m_1 is on the *right* side of e_1. In this case, we should add the *total effects* of e_1 and all its descendant elements e' into *resultSOT*. Finally, since both lists are ordered, we need to scan them only once. Figure 10 depicts the details of this algorithm. *tree(m, subSOT)* in line 14 is to create a new tree with m being the root and all the trees in *subSOT* being its children. The time complexity of *ComputeTotalEffect* algorithm in Figure 10 is $O(n)$, where $n = \sum pointAD(root, HS[M])$ when the query step is AD and $n = \sum pointPC(e, HS[M])$ when it is PC.

```
SOT computeTotalEffects(SOT eSOT, Axis axis, HierarchicalStack HS[M])
SOT resultSOT = mSOT= subSOT=empty;
docElement childElements[], e, m;
1.  BEGIN
2.    FOR each tree t[i] in eSOT //Assume eSOT contains a sequence of trees t[1..p]
3.     IF Axis = AD
4.        resultSOT = resultSOT UNION pointAD(t[i].root, HS[M]);//root element only
5.     ELSE //Axis = PC
6.        mSOT = pointPC(t[i].root, HS[M]);//child elements m that t[i].root points to
7.        childElements = t[i].root.children();//t[i].root's children in tree t[i]
8.        WHILE m = mSOT.next();
9.           WHILE e = childElements.next() and e.RightPos < m.LeftPos
10.             resultSOT = resultSOT UNION computeTotalEffects(e, axis, HS[M]);
11.          subSOT = empty;
12.          WHILE e = childElements.next() and
                      m.LeftPos < e.LeftPos and m.RightPos > e.RightPos
13.             subSOT = subSOT UNION computeTotalEffects(e,axis, HS[M]);
14.          resultSOT = resultSOT UNION tree(m, subSOT);
15.       WHILE e = childElements.next()
16.          resultSOT = resultSOT UNION computeTotalEffects(e,axis, HS[M]);
17.   RETURN resultSOT;
18. END
```

Figure 10: Pseudo Code for Computing Total Effects

EXAMPLE 4. *Assume that A is a non-return node in Figure 4. The SOT for HS[A] contains one tree with a2 being the root and a3, a4 being its children. The total effects of these three elements on B contains two trees, namely, (b1) and (b2, b3) with b2 being b3's parent (since b2 is ancestor of a4 and pointPC(a4, HS[B])=b3). The total effects of (b2,b3) on D contains one tree (d2, d3). In this case, only $pointAD(b2, HS[D])$ needs to be considered.* □

4.3 Complete Enumeration Algorithm

We now present the complete algorithm `EnumTwig²Stack` (Figure 11) to enumerate the results for a given GTP query, which may contain return nodes, group return nodes and non-return nodes. In this paper, we return the GTP results in a tuple format (similar to [15]). That is, each column corresponds to one return node and stores the matching document element ID. When a query node is a group return node, then a list of matching elements' ID is stored. When a query node is optional, the column value may be *null*. It is also easy to return the GTP results in tree format or to include value, attribute information.

Similar to `PathStack` [4], the results are enumerated in the reverse order in contrast to the computation. For `Twig²Stack`, we enumerate the results *top-down* so to only visit these elements that are in the final results. Initially, the stack trees in the query root node represent an SOT structure and serve as a starting point of the enumeration algorithm. Note that when the query starts with "/", only the top element of the SOT tree with Level=1 can be in the final answer. For example, the SOT for the root query node A in Figure 4 is the tree of $a2, a3, a4$.

[5]A special case, $m_1 = e_1$, can be handled similarly.

When traversing down the query nodes, if current query node E is a return node, then we need to *repetitively* traverse down the query nodes for each element in the SOT (line 6). The reason is that each of these elements will lead to some distinct answers. If this query node is also a branch node, we simply do a *Cartesian* product of all the sub-twig results from different branches (line 9). Here *setColumn(e,BranchResult)* in line 10 is to set the E column as e for all the tuples in *BranchResult*. When current query node E is a non-return node, we compute the *total effects* of these elements on E's *non-existence-checking* child node M as in Section 4.2 [6].

Finally, when we reach the leaf node, we convert the resulting SOT into tuples (line 3). In particular, if it is a return node, then we create one tuple for each element in SOT by visiting each tree in pre-order. If it is a group return node, then we just create one tuple with the column value being a list of all the elements in SOT. This way our enumeration solution achieves an *automatic* grouping in the sense that all the elements that belong to the same group already stay together in SOT. An explicit grouping operator over path matches [8] can thus be avoided.

```
GTPResult EnumTwig²Stack (queryNode E, SOT eSOT)
GTPResult totalResult = branchResult = empty;
SOT mSOT;
1.  BEGIN
2.    IF no return node below E
3.      return convert2Tuple(eSOT); //No need to further traverse down
4.    ELSE //there is return node below E, need to traverse down
5.      IF E is return node
6.        FOR each element e in eSOT //Visit each tree in eSOT in pre-order
7.          branchResult = empty;
8.          FOR each E's child non-existence-checking query node M
9.             mSOT = computeTotalEffects(e, axis(E→M), HS[M]);
10.            branchResult = branchResult × EnumTwig²Stack(M, mSOT);
11.         totalResult = totalResult UNION setColumn(e, branchResult);
12.       return totalResult;
13.     ELSE // E is non-return node
14.       mSOT = computeTotalEffects(eSOT, axis(E→M), HS[M]);
15.       return EnumTwig²Stack(M, mSOT);
16. END
```

Figure 11: Pseudo Code of EnumTwig²Stack Algorithm

EXAMPLE 5. *Assume that A, B and D are the return nodes in Figure 4. For each of the elements $a2$, $a3$ and $a4$ (in that order), we need to traverse down the query nodes.*

Now assume that D is the only return node. First, since A is not a return node, we compute the total effects of A's SOT tree (a2,a3,a4) on B as b1 and (b2,b3). Next, since B is also not a return node and the axis between B and D is an AD relationship, only the top elements of B's SOT trees need to be considered, i.e., b1 and b2. Finally, the result tuples are (d1), (d2) and (d3). Clearly, this is much better than first enumerating 9 path matches and then merge-joining (or semi merge-joining) these path matches and finally applying duplicate elimination / sort operation. □

THEOREM 3. *EnumTwig²Stack algorithm in Figure 11 correctly return the results for a given GTP query. The total time complexity for processing a GTP query, including both Twig²Stack and EnumTwig²Stack is $O(|D||B|)$ + $O(|subTwigResults|)$.* □

Here the *subTwigResults* are the results of a sub-twig query, which is a minimal sub-twig in the original twig query that contains all the return nodes. For instance, if A and D are return nodes in Figure 4, then $|subTwigResult|$ is the number of matches when A, B and D are all return nodes in the twig query.

[6]A non-return query node E can have at most one non-existence-checking child query node M in either XPath or XQuery.

4.4 Early Result Enumeration

`Twig²Stack` algorithm described in Section 3 employs a pure bottom-up approach. Note that a *hybrid* approach is possible that integrates both top-down and bottom-up methods. In particular, we use `PathStack` [4] (briefly introduced in Section 3.1) for top-down computation and use `Twig²Stack` for bottom-up computation. More specifically, for any element e, it is pushed into the hierarchical stack $HS[E]$ *iff* it satisfies the *sub-twig* query rooted at E as well as the *prefix* path query from root to E.

To implement the above idea, each query node E is associated with two stacks, one for `PathStack`, $S[E]$, one for `Twig²Stack`, $HS[E]$, respectively. A document element e visited in pre-order is first pushed into the top-down stack $S[E]$ based on `PathStack` algorithm. Once e is popped out from the top-down stack $S[E]$ (post-order), we push it into the hierarchical stack $HS[E]$. Note that `PathStack` is a quite efficient algorithm, i.e., $O(1)$ for pushing or popping an element. Hence, the extra cost is small while the benefit can be significant since the condition for pushing elements into hierarchical stacks become more stringent. Another key advantage for this hybrid approach is that we can enumerate the query results earlier and all the data in the hierarchical stacks can be cleaned up. This will greatly reduce the memory requirement as discussed in Section 3.4.

Assume that the *top branch node* in a GTP query is E. Whenever the elements in $S[E]$, i.e., the top-down stack, are all popped out, we can start to enumerate the query results and then clean up all the hierarchical stacks. The following example in Figure 12 illustrates the main idea of this early result enumeration mechanism.

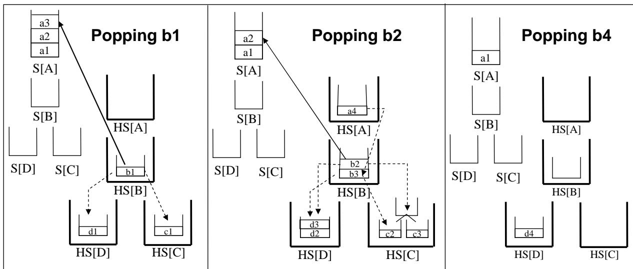

Figure 12: Running Example for Early Result Enumeration

We re-use the query and the data in Figure 4. The top branch node for this query is B. In hybrid query processing mode, when $b1$ is popped out of the top-down stack $S[B]$ and pushed into the hierarchical stack $HS[B]$ (the leftmost portion of Figure 12), we can start to enumerate the query results. Here the solid edge denotes the edge used for `PathStack`, while the dotted edge denotes the edge used for `Twig²Stack`. The result enumeration algorithm is also a hybrid of `PathStack` and `Twig²Stack` enumeration algorithms, which is quite straightforward. After the result enumeration, the data in the hierarchical stacks can be removed. Intuitively, this is due to the fact that the sub-tree of $b1$ will not appear in any future results. There might exist a potential blocking issue whether the enumerated results can be output immediately. Here when A is not return node, then we can output the enumerated results right away. When A is return node, however, the $a3$ results need to be kept in the temporary space (disk) until all $a1$ (and then $a2$) results are enumerated out. Similar issue exists for `PathStack` and can be resolved without sorting [4].

The middle portion of Figure 12 depicts the status when $b2$ is popped out from the top-down stack $S[B]$. In this case, although $a4$ has been popped earlier and we know that there are some matches to the entire twig query, we cannot clean up hierarchical stacks, because these data may lead to new matches for $b2$. We can only clean up the hierarchical stacks when $b2$ is popped out. The rightmost portion shows the status when $b4$ is popped out from the top-down stack. Clearly, this early result enumeration mechanism can greatly reduce the memory requirement.

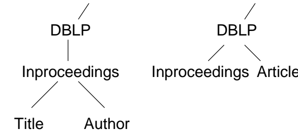

Figure 13: Two Real Examples for Early Result Enumeration

Figure 13 depicts two real queries in order to gain some more insights on the usefulness of this early result enumeration technique in practice. The twig query on the left side is a common one to find the title and authors for each proceeding. The top branch node for this query is $inproceedings$. By early result enumeration, we can output the twig results per proceeding and the memory requirement is also just one proceeding. This is independent of how many proceedings in the XML file are, i.e., how large is the XML file. The top branch node for the query on the right side of Figure 13 is the document root. In this case, early result enumeration does not work because the root element is popped at the end of the document. Now assume there are 100000 inproceedings and 100000 articles. This twig query will give a 100000^2 number of distinct twig results. Such a full Cartesian product provides little extra information and is not very useful in practice. Hence such worst case is unlikely to occur in practice. Also we note that in the case of when the top branch node is the document root, there is a simple solution. That is, we can process each branch sub-twig query of the root separately, i.e., $/dblp/inproceedings$... and $/dblp/article$..., and perform a Cartesian product at the end. This way the early result numeration technique remains effective when processing individual branches. A more in-depth study for resolving such worst case memory requirement problem remains our future work.

5. EXPERIMENTAL EVALUATIONS

5.1 Experimental Setup

We implemented our `Twig²Stack` algorithm using Java 1.4.2 and performed experiments on a PC with a Pentium M-2GHz processor and 2GB of main memory. We set the Java virtual machine memory size to 500M. We compared `Twig²Stack` with two other twig join algorithms: `TwigStack` [4] and `TJFast` [16] [7]. `TJFast` has the best performance in terms of I/O cost and CPU time among the existing twig join algorithms, while `TwigStack` is the classical holistic twig join algorithm.

Datasets. A set of synthetic and real datasets are used for the experimental evaluation, which represent a wide range of XML datasets (Figure 14). In particular, the synthetic dataset includes XMark [11]. The scaling factors of 1 to 5 were selected to generate a set of XMark synthetic datasets for scalability analysis. The two real datasets included are DBLP and TreeBank from [20]. The DBLP dataset is a wide and shallow document, while the TreeBank dataset is a deep recursive document.

Metrics. We compared the three twig join algorithms in terms of the *query processing time* and the *total execution time*.

[7] The implementation of these two algorithms in Java were kindly provided by Jiaheng Lu from the National University of Singapore.

- *query processing time*: the time spent on performing the structural matching. For Twig²Stack, it is the time to perform the merging of hierarchical stacks and the result enumeration. For TwigStack, it is the time to perform computing and enumerating path matches, and finally merge-joining the path matches [4]. For TJFast, it is the time to perform analysis of the extended dewey ID of the leaf element to infer its ancestors' label, enumerating path matches, and finally merge-joining the path matches [16].
- *total execution time*: *query processing time* plus *the scanning cost of the document elements*. The scanning cost is basically IO cost. For both TwigStack and Twig²Stack, their scanning costs are the same, namely, for accessing the document elements corresponding to all query nodes. For TJFast, the scanning cost is for accessing the document elements corresponding to *only* those query leaf nodes. Hence, TJFast accesses fewer number of document elements, while the size per element may be larger since extended dewey ID for leaf elements typically is larger than region encoding.

	DBLP	TreeBank	XMark				
			1*	2	3	4	5
Dataset Size (MB)	127	82	113	226	340	454	568
Nodes (Million)	3.3	2.4	1.7	3.3	5	6.7	8.3
Max/Avg Depth	6/2.9	36/7.9	12/5.5				

Figure 14: Statistics of XML Data Sets

Query Name	Dataset	Twig Query
DBLP-Q1	DBLP	//dblp/inproceedings[title]/author
DBLP-Q2	DBLP	//dblp/article[author][.//title]//year
DBLP-Q3	DBLP	//inproceedings[author][.//title]//booktitle
XMark-Q1	XMark	/site/open_auctions[.//bidder/personref]//reserve
XMark-Q2	XMark	//people//person[.//address/zipcode]/profile/education
XMark-Q3	XMark	//item[location]/description//keyword
TreeBank-Q1	TreeBank	//S/VP//PP[.//NP/VBN]/IN
TreeBank-Q2	TreeBank	//S/VP/PP[IN]/NP/VBN
TreeBank-Q3	TreeBank	//VP[DT]//PRP_DOLLAR_

Figure 15: Twig Queries used for the Experimental Evaluation

Twig Queries. Figure 15 shows all the twig queries used for the experiments. For each data set, three twig queries are selected, which have different combinations of parent-child and ancestor-descendant relations and different selectivities over the datasets.

5.2 Full Twig Query Processing

We first compare Twig²Stack with TwigStack and TJFast for processing the full twig query (all query nodes are return nodes).

5.2.1 Performance Results

Figure 16 depicts the performance results based on the queries in Figure 15 on DBLP, XMark (scale 1) and Treebank datasets in Figure 14. For each twig query, we record the *query processing time*, *total execution time* and *IO time* for all three algorithms.

DBLP Dataset: Figure 16.(a) reports the *query processing time*, (b) reports the *total execution time* and (c) reports the *IO time*. Twig²Stack achieves one order of magnitude performance gain over TwigStack, and is two to three times faster than TJFast in terms of the *query processing time*. Our detailed cost breakdown shows that this is primarily due to the fact that Twig²Stack avoids generating any path matches. Actually, the enumeration of path matches is non-trivial, even when all the generated path matches are in the final results. The reason is that enumerating path matches requires either traversing the PathStack for TwigStack [4] or analyzing the extended dewey ID using the DTD transducer for TJFast [16]. The same element may also exist in many path matches, resulting in duplicated efforts.

In comparison, although Twig²Stack may also push a document element e into $HS[E]$ with e potentially not being in the final results, the cost of merging $HS[E]$ and all its child hierarchical stacks is *not* wasted. The reason is that they reduce the query processing cost, i.e., merging cost, for the remaining elements.

The *total execution time* of Twig²Stack and TJFast is closer as depicted in Figure 16.(b). The reason can be explained in Figure 16.(c), i.e., TJFast saves more IO cost since it only needs to access the elements corresponding to the leaf query nodes [8]. Note that, our Twig²Stack algorithm developed in this paper is dedicated for optimizing the twig query processing cost. It is possible to extend Twig²Stack to further reduce the IO cost. A viable approach is to create a variant of B^+ tree index on the document elements so that Twig²Stack can skip scanning the elements that cannot satisfy the query steps. A similar approach, called XB-tree, is developed in [4] and has been shown to be quite effective. This is certainly a promising future direction to explore.

XMark Dataset: Figure 16.(d), (e) and (f) depicts the results on the XMark dataset with scale factor 1. For the query processing time of this data set, Twig²Stack again shows consistent order of magnitude performance gain over TwigStack and is several times faster than TJFast. Our detailed cost breakdown shows the same reason, i.e., path enumeration. For the total execution cost of this dataset, TJFast actually introduces larger IO cost for Q3. Q3 contains three leaf query nodes with only one non-leaf node. Hence, the saving on scanning the elements corresponding to one non-leaf query node is smaller than the cost paid for having a larger extended dewey ID per element.

TreeBank Dataset: Figure 16.(g), (h) and (i) depicts the results. For the query processing time, Twig²Stack again significantly outperforms TwigStack, and is two to three times faster than TJFast for Q1. For Q2 and Q3, since the selectivity of path matches is very high, only total 300 and 5 matches, respectively, the query processing time for Twig²Stack and TJFast becomes closer. The saving on IO cost for TJFast is bigger for this dataset. The reason is that the twig queries, especially Q2 has many distinct non-leaf query nodes. Meanwhile, since TreeBank is a narrow dataset, this means that the number of occurrences for even higher level elements is high, resulting in a large index size.

5.2.2 Experiments on Scalability

We now report the scalability expermental results of Twig²Stack algorithm in terms of the size of the XML document in Figure 17. We vary the XMark scale factor from 1 to 5. As can be seen, all three algorithms grow linearly in terms of the document size. Twig²Stack again has much better *query processing time*.

5.3 GTP Query Processing

In this section, we will study the performance of Twig²Stack algorithm for processing GTP queries. Since neither TwigStack nor TJFast is fine tuned for processing GTP queries, we will not include them in the experiments as it will be unfair if we simply apply a post-processing on top of these two algorithms. On the other hand, it is non-trivial to derive an efficient mechanism for

[8]Accessing only the leaf elements however is not always sufficient, e.g., when the query includes the values or attributes of the non-leaf query nodes.

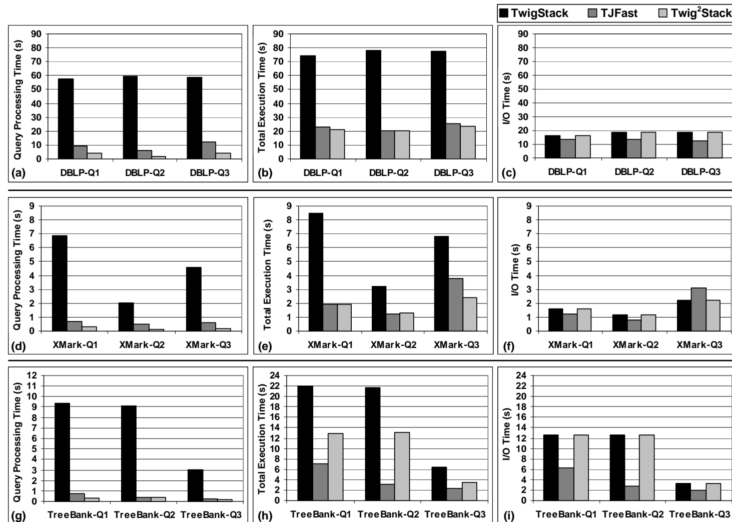

Figure 16: Full Twig Query Processing on DBLP, XMark and TreeBank Datasets

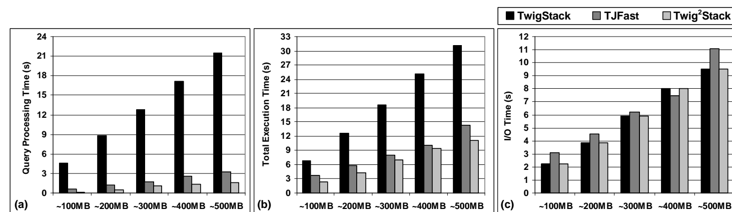

Figure 17: Scalability for Full Twig Query Processing

processing GTP queries based on path-joins. The reason is that efficient path-joins require all the path matches be sorted in root-to-leaf order in order to use the cheap *merge join*. This root-to-leaf order however may not always correspond to the order of return nodes. Hence a post-sort becomes unavoidable. Alternatively, if we can produce the path matches in the order of return nodes, then the path join cannot use the efficient merge join method.

5.3.1 GTP Queries over DBLP Dataset

We use DBLP-Q1 in Figure 15 as the baseline twig query and then arbitrarily set some query node as non-return node or group return node. Figure 18 depicts different GTPs and their query processing cost. Note that the IO cost is the same for all GTPs.

Figure 18.(a) is the baseline twig query with every node being a return node. (b) is a GTP query with *title* being a non-return node. In this case, the query processing cost for this GTP reduced compared to that for (a). The reason is that node *title* is an *existence-checking* node. In this case, we can avoid maintaining the hierarchical structure in $HS[title]$ and avoid creating any edges from *inproceedings* element to *title* element (Section 3.4). The result enumeration also need not access $HS[title]$. (c) is a GTP query with *author* being a non-return node. In this case, the saving is even bigger since there are several authors per inproceedings while there is only one title per inproceedings in the DBLP dataset. Finally, (d) is a GTP query with *author* being a group return node. Compared to (b), where the only difference is the way how *author* is returned, (d) results in a much cheaper cost. The reason is that for (d), we group multiple authors to a list and create a single tuple, while for (b), we have to create one tuple per author. Most existing works [8, 10] first generate path matches as (b) and then apply *grouping*. This approach obviously is less efficient than ours.

5.3.2 GTP Queries over XMark Dataset

We now use XMark-Q1 in Figure 15 as the baseline twig query and then arbitrarily set non-return nodes and optional axes. Figure 19 depicts different GTPs and their query processing cost.

Figure 19.(a) is the baseline twig query. (b) is a GTP query with *address* and *zipcode* being non-return nodes. In this case, the query processing cost is reduced compared to that for (a), since *address* and *zipcode* become existence-checking nodes. (c) is a GTP query with only *education* being the return node. Compared to (b), although we still have to maintain $HS[people]$ and $HS[person]$ since they have return node below, the final result size is reduced and so does the result enumeration cost. (d) is a GTP query with the axis between *person* and *address* being optional, while (e) is a GTP query in addition has the axis between *profile* and *education* being optional. In both cases, the number of twig matches is several times larger than that of (a), while the increase of query processing time is small. In contrast, optional semantics is often supported by using *Outer-Join* of path matches [8], while outer-join is known to be in general more expensive than inner-join.

5.4 Runtime Memory Usage

In this section, we report the memory usage for processing the above twig queries and how our early result enumeration technique helps to reduce the memory usage. Table 1 depicts the memory

DBLP-Q1	(a)	(b)	(c)	(d)
GTP Query	dblp, inproceedings, title, author	dblp, inproceedings, title, author	dblp, inproceedings, title, author	dblp, inproceedings, title, author
Query Processing Time (s)	4.597	3.433	1.362	2.445

Figure 18: GTP Query Processing on DBLP Dataset

XMark-Q2	(a)	(b)	(c)	(d)	(e)
GTP Query	people, person, address, profile, zipcode, education	people, person, address, profile, zipcode, education	people, person, address, profile, zipcode, education	people, person, address, profile, zipcode, education	people, person, address, profile, zipcode, education
Query Processing Time (ms)	130	80	60	150	180

Figure 19: GTP Query Processing on XMark Dataset

usage for processing the twig queries in Figure 16, with or without early result enumeration (ERM) enabled (Section 4.4).

First, let us consider DBLP dataset. The total memory consumptions for all three queries are quite high. This is due to the low selectivity of these queries. Basically, all the *inproceedings* (*articles*) are selected by those queries (selectivity can be much higher if value predicates are present). Note that the memory usage is even bigger than the file size (127M) due to those pointers, a situation that has already been observed in main memory XML database [17]. Note that when the early result enumeration technique is employed, the runtime memory usage is significantly reduced to less than 1Kbytes! The reason is that as soon as one *inproceedings* (*article*) has been visited, we can output the results and free up the memory. The memory requirement is thus bounded to the matches per *inproceedings* (*article*). Note that such matches are typically just related to the type of the document (e.g., from DTD) and is independent of the size of the document.

Next, let us consider TreeBank (TB) dataset. TreeBank is a dataset with many distinct labels and with quite irregular structures. The selectivity is thus very high, which consequently results in low total memory usage even without early result enumeration enabled (compared to 82M document size). Nonetheless, early result enumeration reduces the runtime memory usage to just several Kbytes.

Finally, let us consider XMark (XM) dataset. Two scale factors, s=1 (100MBytes) and s=10 (1GBytes) are used to generate the document. The total memory usage (without early result enumeration) grows as the scale factor increases for all three queries. Note that the early result enumeration becomes ineffective for $Q1$. From the query itself, we find that its top branch node is *open_auctions*. In XMark dataset, there is only one *open_auctions* element which contains a huge number of *open_auction* elements. This is almost equivalent to the second case in Figure 13 (Section 4.4), where the root *dblp* has a huge fan-out. This hints us that a promising way to address this worst case memory problem is to find those query nodes which have a huge fan-out (subtree) in the document, and then effectively decompose the processing of individual branches (i.e., hybrid query plan). Next, our early result enumeration technique is very effective for handling $Q2$ and $Q3$, i.e., the runtime memory usage is independent of the document file size. Here the top branch nodes for $Q2$ and $Q3$ are *person* and *item*, respectively. Since the information contained in each *person* and in each *item* can typically be considered as constantly small, the runtime memory usage remains stably low. As a final remark, we study the sample queries in XMark [11]. Among the total 20 queries, the top branch nodes are *open_auction*, *close_auction*, *person* and *item* [9], all of which are small trees. Hence, our early result enumeration technique likely would be useful for most practical queries.

6. RELATED WORK

Tree pattern queries over XML documents have been extensively studied recently. Most existing techniques rely on the region encodings [19] to capture the structural relationship between document elements. Early work decomposes the tree pattern queries into a set of binary components. Then the matches of each individual component are stitched together to get the final results [3]. The main drawback of the decomposition method is the large intermediate results. Hence, in [4], the first holistic twig join algorithm, called `TwigStack`, was proposed. However, `TwigStack` achieves optimality for twig queries with AD relations only. For twig queries with PC relations, useless path matches cannot be completely avoided. Chen et al. proposed `iTwigJoin` [6] that exploits different data partition strategies to further boost the holism. More recently, Lu et al. [16] proposed `TJFast` to access only leaf elements by exploiting extended Dewey IDs. Unfortunately, useless path matches cannot be avoided in all cases as shown theoretically in [9]. Aghili et al. [2] incorporates a binary labeling algorithm (based on the notion of nearest common ancestors) as a pre-processing filtration step to reduce the search space and computes the twig matches in a bottom-up fashion. However, their technique is mostly effective for the scenarios where the search involves selective keywords at the leaf nodes of twig queries. In this paper, we show that enumerating path matches is *not* necessary for processing twig queries and hence do not have such optimality problem. Furthermore, most these existing works do not consider how to efficiently process the more powerful GTP queries.

[9]An exception is $Q7$, where the top branch node is the document root. This can be easily handled by decomposing the twig queries at the root node and performing a final Cartesian product.

	DBLP - ERM	DBLP + ERM	TB - ERM	TB + ERM	XM (100M / 1G) - ERM	XM (100M / 1G) + ERM
$Q1$	150M	0.7K	9M	15K	9M / 87M	9M / 87M
$Q2$	105M	0.9K	1.5M	10K	7M / 82M	2.2K / 2.6K
$Q3$	170M	0.8K	6K	3K	10M / 118M	0.9K / 1K

Table 1: Memory Usage for Twig Queries in Figure 16

Processing GTP queries generally calls for costly post-processing [8]. Chen et al. proposed a stack-based matching algorithm for graph input data [5]. A hybrid of top-down and bottom-up computation paradigm is employed (similar to ours). However, they do not maintain the hierarchical structure of a single stack since Lemma 1 does not hold for graph data.

Bottom-up tree pattern matching has been extensively studied in the area of classic tree pattern matching [12]. Note that these early work however do not consider AD relationship, which is common for XML queries. Recently, a bottom-up tree homomorphism algorithm for XPath containment checking is proposed in [18]. Only the existence of such a match is returned, which is enough for containment checking. In this paper, we show how to return the full matches for GTP queries under this computation paradigm.

7. CONCLUSIONS

In this paper, we proposed a novel hierarchical stack encoding scheme to compactly represent the twig results and introduced a bottom-up twig processing algorithm. Then we showed how to efficiently enumerate the GTP results from the encodings.

There are many promising future directions. For example, tree pattern matching has also been extensively studied in XML stream environment [7, 15]. Most existing techniques on holistic twig processing, such as those in [4, 16], cannot be applied to XML streams. The reason is that they need to look up other node indexes to see if this path will participate in the final twig matches. Such *look-ahead* feature is not available in XML stream environment, since the document is sequentially scanned. In comparison, our `Twig²Stack` algorithm can be *directly* applied. That is, in XML stream environment, the *start-tags* follows pre-order, while the *end-tags* follows post-order! As a quick comparison, state-of-the-art nested XPath (one return node) matching solution `TwigM` [7] has complexity $O(|D||Q|(|Q| + RC))$ (where R is the document depth, C is candidate solution with $|C| \leq |D|$), while `Twig²Stack` has complexity as low as $O(|D||B|)$ (from Theorem 3) and is capable of processing the more complex GTP queries. It is interesting future work to conduct the actual performance comparison to various existing XML stream solutions [7, 15]. Other interesting future works include how to handle worst case memory issues by using hybrid query plans, how to exploit the index, how to support other axes, and how to handle multiple GTP queries over XML streams.

8. REFERENCES

[1] Galax: An implementation of xquery. http://db.bell-labs.com/galax/optimization/.

[2] A. Aghili, H. Li, D. Agrawal, and A. E. Abbadi. TWIX: Twig Structure and Content Matching of Selective Queries using Binary Labeling. In *INFOSCALE*, 2006.

[3] S. Al-Khalifa, H.V.Jagadish, J.M.Patel, Y. Wu, N.Koudas, and D.Srivastava. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proceedings of ICDE*, pages 141–152, 2002.

[4] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proceedings of SIGMOD*, pages 310–321, 2002.

[5] L. Chen, A. Gupta, and M. E. Kurul. Stack-based Algorithms for Pattern Matching on DAGs. In *Proceedings of VLDB*, pages 493–504, 2005.

[6] T. Chen, J. Lu, and T. W. Ling. On Boosting Holism in XML Twig Pattern Matching. In *Proceedings of SIGMOD*, pages 455–466, 2005.

[7] Y. Chen, S. Davidson, and Y. Zheng. An Efficient XPath Query Processor for XML Streams. In *Proceedings of ICDE*, 2006. To appear.

[8] Z. Chen, H. V. Jagadish, L. V. S. Lakshmanan, and S. Paparizos. From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery. In *Proceedings of VLDB*, pages 237–248, 2003.

[9] B. Choi, M. Mahoui, and D. Wood. On the Optimality of Holistic Algorithms for Twig Queries. In *Proceedings of DEXA*, pages 28–37, 2003.

[10] Y. Diao and M. Franklin. Query Processing for High-Volume XML Message Brokering. In *Proceedings of VLDB*, pages 235–244, 2003.

[11] A. R. S. et al. The XML Benchmark Project. Technical Report, INS-R0103, CWI, 2003.

[12] C. M. Hoffmann and M. J. O'Donnell. Pattern Matching in Trees. *Journal of the ACM*, 29(1):68–95, 1982.

[13] H. V. Jagadish, L. V. S. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A Tree Algebra for XML. In *Proceedings of DBPL*, pages 149–164, 2001.

[14] H. Jiang, H. Lu, and W. Wang. Efficient processing of twig queries with or-predicates. In *Proceedings of SIGMOD*, pages 59–70, 2004.

[15] V. Josifovski, M. Fontoura, and A. Barta. Query XML Streams. *VLDB Journal*, 14(2):197–210, 2005.

[16] J. Lu, T. W. Ling, C. Y. Chan, and T. Chen. From Region Encoding To Extended Dewey: On Efficient Processing of XML Twig Pattern Matching. In *Proceedings of VLDB*, pages 193–204, 2005.

[17] A. Marian and J. Simeon. Projecting XML Documents. In *Proceedings of VLDB*, pages 213–224, 2003.

[18] G. Miklau and D. Suciu. Containment and Equivalence of a Fragment of XPath. *Journal of the ACM*, 51(1):2–45, 2004.

[19] M.P.Consens and T.Milo. Optimizing Queries on Files. In *Proceedings of SIGMOD*, pages 301–312, 1994.

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

[21] W3C. XML Path Language (XPath) Version 1.0. W3C Recommendation. http://www.w3.org/TR/xpath.html, 1999.

[22] W3C. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, 2005.

[23] C. Zhang, J. Naughton, D. Dewitt, Q. Luo, and G. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proceedings of SIGMOD*, pages 425–436, 2001.