On Boosting Holism in XML Twig Pattern Matching Using Structural Indexing Techniques

Ting Chen, Jiaheng Lu and Tok Wang Ling
School of Computing National University of Singapore
3 Science Drive 2, Singapore 117543
{chent,lujiahen,lingtw}@comp.nus.edu.sg

ABSTRACT

Searching for all occurrences of a twig pattern in an XML document is an important operation in XML query processing. Recently a holistic method *TwigStack* [2] has been proposed. The method avoids generating large intermediate results which do not contribute to the final answer and is CPU and I/O optimal when twig patterns only have ancestor-descendant relationships. Another important direction of XML query processing is to build structural indexes [3][8][13][15] over XML documents to avoid unnecessary scanning of source documents. We regard XML structural indexing as a technique to partition XML documents and call it *streaming scheme* in our paper. In this paper we develop a method to perform holistic twig pattern matching on XML documents partitioned using various streaming schemes. Our method avoids unnecessary scanning of irrelevant portion of XML documents. More importantly, depending on different streaming schemes used, it can process a large class of twig patterns consisting of both ancestor-descendant and parent-child relationships and avoid generating redundant intermediate results. Our experiments demonstrate the applicability and the performance advantages of our approach.

1. INTRODUCTION

XML data is often modelled as labelled and ordered tree or graph. Naturally twig (a small tree) pattern becomes an essential part of many XML queries. A twig pattern can be represented as a node-labelled tree whose edges are either *Parent-Child* (P-C) or *Ancestor-Descendant* (A-D) relationship. For example, the following twig pattern written in XPath[19] format:

$section[/title]/paragraph//figure$ ..... (Q1)

selects $figure$ elements which are descendants of some *paragraph* elements which in turn are children of *section* elements having at least one child element *title*.

Prior work on XML twig pattern processing usually decomposes a twig pattern into a set of binary relationships which can be either parent-child or ancestor-descendant relationships. After that, each binary relationship is processed using *structural join* techniques [1][20] and the final match results are obtained by "stitching" individual binary join results together. The main problem with the above solution is that it may generate large and possibly unnecessary intermediate results because the join results of individual binary relationships may not appear in the final results.

Bruno et al. [2] proposes a novel holistic XML path and twig pattern matching method *TwigStack* which avoids storing intermediate results unless they contribute to the final results. The method, unlike the decomposition based method, avoids computing large redundant intermediate results. The method is CPU and I/O optimal for all path (with no branch) patterns and twig patterns whose edges are entirely ancestor-descendant edges. Meanwhile the space complexity of their algorithm is bounded by the longest path in the source XML document. However the approach is found to be suboptimal if there are parent-child relationships in twig patterns. That is, the method may still generate redundant intermediate results in the presence of P-C relationships in twig patterns.

An important assumption of the original holistic method is that an XML document is clustered into element streams which group all elements with the same tag name together and assign each element a containment label[2]. We call this clustering method *Tag Streaming*. In recent years there have been considerable amount of research on XML indexing techniques [3][8][13][15] to speed up queries upon XML documents. In general, these XML indexes can be regarded as summary of XML source documents and thus much smaller in sizes. From another point of view, XML structural indexing can also be viewed as methods to partition XML documents for query processing. Interestingly, *Tag Streaming* used in *TwigStack* can be regarded as a trivial XML indexing technique which groups all elements with the same tag together. Up till now, very little research has been done on performing holistic twig matching on XML documents partitioned by other structural indexing techniques than Tag Streaming. Furthermore, little is known about if more sophisticated XML structural indexing methods may allow optimal processing of other classes of twig pattern (besides A-D only twig patterns). Note that in view of the terminology used in the original holistic pattern matching paper, we call the combination of XML indexing methods with containment labels as *XML streaming schemes*.

In this paper, we demonstrate that in general a more "sophisticated" (we will give a formal definition in later sec-

SIGMOD 2005 June 14-16, 2005, Baltimore, Maryland, USA

tions) XML streaming scheme has the following two advantages than a simpler one in twig pattern processing: (1) Reduce the amount of input I/O cost; (2) Reduce the sizes of redundant intermediate results. The main reason behind is the increased "parallelism" to access elements with the same tag and the additional "context" information we know about each element.

The main contributions of our work are:

- By studying in detail two XML streaming schemes: (1) a new *Tag+Level* scheme, which partitions elements according to their tags and levels; (2)*Prefix Path Streaming* (PPS), first proposed in [4], which partitions elements according the label path from the root to the element, we show rigourously the impact of choosing XML streaming schemes on optimality of processing different classes of XML twig patterns.

- We develop a holistic Twig Join algorithm *iTwigJoin* which works *correctly* on any XML streaming scheme (as long as elements in a stream are ordered by their pre-orders). Applied on the *Tag+Level* scheme the algorithm can process A-D or P-C only twig patterns optimally; applied on the *PPS* scheme the algorithm can process A-D only or P-C only or 1-Branchnode only twig patterns optimally.

- Using experiments we study the tradeoff between the increase in overhead to manage more element streams and the reduction in both input I/O cost and intermediate result sizes caused by various XML streaming schemes. The goal is to find streaming schemes for different types of XML documents which can process a large class of twig pattern optimally and meanwhile incur little overhead.

The remaining parts of our paper are organized as follows: Section 2 reviews the state of the art twig pattern matching algorithm: *TwigStack* and its problems. Section 3 introduces several alternative streaming methods. Section 4 describes how to prune away irrelevant streams to a twig pattern. Section 5 explains in detail the properties of these streaming schemes used in twig pattern matching. Section 6 gives algorithms which perform holistic join over streams. Section 7 presents our experiment results. Section 8 surveys related work. Section 9 concludes the paper.

2. BACKGROUND

In this section, we discuss our XML data model and briefly review the problem of XML twig pattern matching and the state-of-the-art holistic twig pattern matching method: *TwigStack*. Notations related to XML twig pattern query are also introduced.

2.1 XML Data Model and Twig Pattern Query

In this paper, an XML document is modelled as a rooted, ordered and labelled tree. Without loss of generality, wherever the word "element" appears, it refers to either element or attribute in an XML document.

Many join algorithms on XML documents rely on certain numbering schemes. For example, the binary XML structural join discussed in [1][20] and the twig join discussed in [2] use *(startPos: endPos, LevelNum)* (an example of *region encoding*) to label elements in an XML file. *startPos* and *endPos* are calculated by performing a pre-order (document order) traversal of the document tree; *startPos* is the number in sequence assigned to an element when it is first encountered and *endPos* is equal to one plus the *endPos* of the last element visited. Leaf elements have *startPos* equal to *endPos*. *LevelNum* is the level of a certain element in its data tree. Element A is a descendant of Element B if and only if $startPos(A) > startPos(B)$ and $endPos(A) < endPos(B)$. A sample XML document tree labeled with the above scheme is shown in Fig.1.

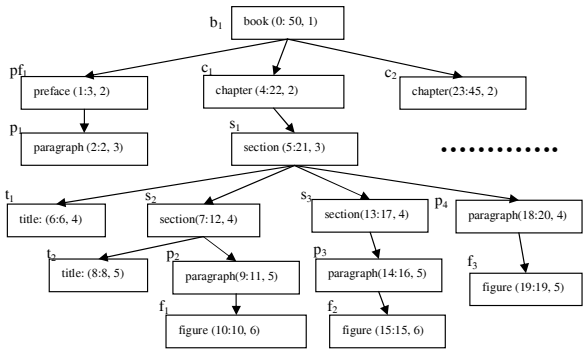


Figure 1: A sample XML document with node labels in parentheses. Each element is also given an identifier (e.g. s_1) for easy reference.

In their paper [2], a *twig pattern match* in an XML database D is defined as an n-ary tuple $< d_1, d_2, \ldots, d_n >$ consisting of the database nodes that identify a distinct match of the twig pattern Q with n nodes in D. In the same paper, the problem of twig pattern matching is defined as:

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

As an example, the matches to twig pattern $Q1$ of Section 1 in Fig.1 are tuples $< s_1, t_1, p_4, f_3 >$ and $< s_2, t_2, p_2, f_1 >$.

In the remaining sections of this paper, we use Q to denote a twig pattern and Q_A to denote the subtree of Q rooted at node A. We use *node* to refer to a query node in twig pattern and *element* to refer to a data node in XML data tree. We use M or $M =< e_1, e_2, \ldots, e_n >$ to denote a match to a twig pattern or sub-twig pattern where e_i is an element in the match tuple. We assume there is no node with identical tag in twig pattern Q. Since an element tag corresponds to a unique node in the twig, we use tag q and query node q interchangeably from now on.

2.2 XML Stream Model

In our paper, each XML "stream" is a posting list (or inverted list) accessed by a simple iterator. An XML streaming scheme is a combination of XML structural indexing techniques and element labelling scheme. More specifically, we partition an XML document into streams (in the terminology of XML structural indexing, the corresponding term for *stream* is *extent*). The only addition is to assign an region coding label to each element in the streams. In this paper, all elements in a stream are of the same tag and ordered by their *startPos*. We usually use T to denote a stream. A stream T has two parts: $head(T)$ which is the stream's first element and $tail(T)$ which is the rest of the stream. One

can only read the head of a stream but not the tail portion of a stream (Fig. 2).

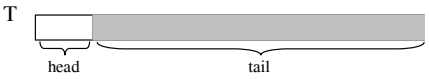


Figure 2: XML Stream Model

2.3 Holistic Twig Join: TwigStack

The holistic method *TwigStack*,proposed by Bruno et al[2], is CPU and I/O optimal for all path patterns and A-D only twig patterns. It associates each node q in the twig pattern with a stack S_q and a stream T_q containing all elements of tag q. Each stream has an imaginary *cursor* which can either move to the next element or read the element under it. The algorithm operates in two main phases: (1) TwigJoin. In this phase a list of element paths are output as intermediate results. Each element path matches one root-to-leaf path of the twig pattern. (2) Merge. In this phase, the list of element paths are merged to produce the final output.

When all the edges in the twig are A-D edges, *TwigStack* ensures that each path output in phase 1 not only matches one path of the twig pattern but is also part of a match to the entire twig pattern. Thus the algorithm guarantees that *its time and I/O complexity is independent of the size of partial matches to any root-to-leaf path of the descendant-only twig pattern.* In this paper, we define a twig pattern matching algorithm is *optimal* if all the following three conditions are satisfied: (1) (Scan Once) Each stream whose elements' tag appears in the twig pattern is scanned only once. (2) (No redundant output) None of the intermediate paths output in Phase 1 is redundant. (3) (Bounded space complexity) The space required by the algorithm is bounded by a factor which is independent of source document size.

However, with the presence of P-C edges in twig patterns, the TwigStack method is no longer optimal.

EXAMPLE 2.1. *Suppose we evaluate the pattern $A[/B]/C$ on the XML document in Fig. 3(a). Element a_1 is in match $< a_1, b_1, c_{n+1} >$. However, under the tag streaming scheme, with stream cursor positions shown in Fig. 3(b), we can not tell from the current head elements (e.g. a_1,b_1,c_1) that a_1 is indeed in a match. Indeed, we observe that under tag streaming, the XML document in Fig. 3(c) can not be distinguished from the XML document in Fig. 3(a) because they have exactly the same set of streams and corresponding head elements. However, in the second document a1 is not in any match. Consequently, for the document in Fig. 3(a) we have to scan and stored all the elements in the stream T_c until c_{n+1} before we are certain that a_1 is in a match. By doing so we violate the bounded space requirement.*

Noticeably, if we change the twig pattern from $A[/B]/C$ to $A[//B]//C$ and evaluate on the same document, under tag streaming the set of head elements now form a match $< a1, b1, c1 >$ and the previous problem does not exist anymore.

3. XML STREAMING SCHEMES

In this section, we formally introduce various streaming techniques used in this paper and notations about XML streams.

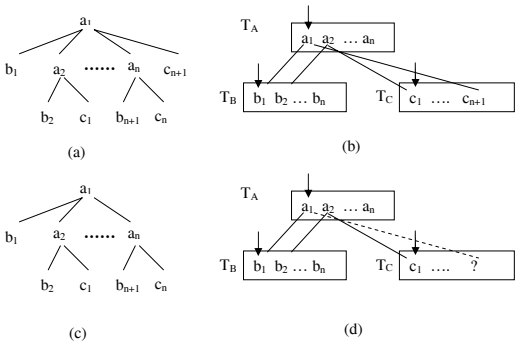


Figure 3: The problem of twig Join using Tag Streaming

Tag Streaming A *Tag Stream* in the tag streaming scheme contains all elements in the document tree with the same tag. For example,a stream T_A contains all elements of tag A.

Tag+Level Streaming The *level* of an element in an XML document tree is equal to the number of nodes from the root to the element. A *Tag+Level* stream contains all elements in the document tree with the same tag and level. A Tag+Level stream can be uniquely identified by the common tag and level of all its elements. For example,a stream T_A^2 contains all elements of tag A and located in level 2.

Prefix-Path Streaming (PPS) The *prefix-path* of an element in an XML document tree is the path from the document root to the element. A *prefix-path stream (PPS stream)* contains all elements in the document with the same prefix-path, ordered by their *startPos* numbers. A PPS stream T can be uniquely identified by its *label*, which is the common prefix-path of its elements. For example, a stream T_{ABA} contains all elements of tag A and of prefix-path ABA.

Independent of the XML streaming scheme used, we call a stream of class q if it contains elements of tag q. Identical to the notion of *refinement* [15] in XML structural indexes, we call a streaming scheme α is a *refinement* of streaming scheme β if for any two elements in a stream under α, the pair of elements are also in a stream under β. It can be proven that $Tag+Level$ streaming is a refinement over Tag Streaming and PPS Streaming is a refinement of both Tag and $Tag + Level$ streaming.

3.1 Notions of XML Streams related to Twig Pattern Matching

We first define the following notions of *ancestor/descendant streams* and *parent/child streams*.

Under Tag+Level streaming, a stream T_1 is an *ancestor stream* of stream T_2 if the level of T_1 is larger than that of that of T_2. T_2 is called the descendant stream of T_1. T_1 is the *parent stream* of the level of T_1 is equal to that of T_2 plus 1. T_2 is T_1's child stream.

Likewise under PPS streaming, a PPS stream T_1 is an *ancestor stream* of PPS stream T_2 if label(T_1) is a prefix of label(T_2). T_1 is the *parent stream* of T_2 if label(T_1) is a prefix of label(T_2) and label(T_2) has one more tag than

label(T_2). The definitions of descendant and child stream for PPS streaming follow.

Given an P-C or A-D edge $< q_1, q_2 >$ for which q_1 is the parent node in a twig pattern Q, two streams T_1 of class q_1 and T_2 of class q_2 is said to *satisfy the structural relationship* of edge $< q_1, q_2 >$: (1) Under Tag Streaming, the two streams automatically qualify. (2) Under Tag+Level Streaming or PPS Streaming, T_1 is the parent stream of T_2 if $< q_1, q_2 >$ is a P-C edge; T_1 is the ancestor stream of T_2 if $< q_1, q_2 >$ is an A-D edge.

Intuitively, two streams are said to satisfy an edge if there exist two elements, one from each stream, that satisfy the P-C and A-D edge relationship. Finally we have the following definition which is frequently used later on

DEFINITION 3.1. *(Solution streams) The* Solution Streams *of a stream T of class q for q's child edge $< q, q_i >$ in a twig pattern Q (or* $\text{soln}(T, q_i)$*) are the streams of class q_i which satisfy the structural relationship of edge $< q, q_i >$ with T.*

4. PRUNING XML STREAMS IN VARIOUS STREAMING SCHEMES

Using labels of Tag+Level or Prefix-Path streams labels, we can prune away streams which apparently contain no match to a twig pattern. The technique used in stream pruning of Tag+Level and PPS streams are very similar.

The following recursive formula helps us determine the useful streams for evaluating a twig pattern Q using the two streaming schemes. For a stream T of class q, we define U_T to be the set of all descendant streams of T (including T) which are useful for the sub-twig of Q_q except that we only use stream T (not any other stream of class q) to match node q.

$$U_T = \begin{cases} \{T\} & \text{if q is a leaf node;} \\ \{T\} \bigcup \{\bigcup_{q_i \in child(q)} C_i\} & \text{if none of } C_i \text{ is } \{\}; \\ \{\} & \text{if one of } C_i \text{ is } \{\}. \end{cases}$$

where $C_i = \bigcup_{T_c \in soln(T, q_i)} U_{T_c}$

Function $child(q)$ returns the child nodes of q in the twig pattern Q. Apparently, under different streaming approaches, we just need to switch the definition of *child/descendant stream* and *solution streams*.

The base case is simple because if q is a leaf node, any stream of class q must contain matches to the trivial single-node pattern Q_q. As for the recursive relationship, note that for a stream T of class q to be useful to the sub-twig Q_q, for each and every child node q_i of q, there should exist some non-empty set U_{T_c} which are useful to the sub-twig Q_{q_i} AND the structural relationship of T and T_c satisfies the edge between q and q_i. In the end the set $\bigcup U_{T_r}$ contains all the useful streams to a query pattern Q, where T_r is a stream of class $root(Q)$. Notice that the above recursive relationship can be easily turned into an efficient algorithm using standard *dynamic programming* without worrying about excessive re-computation. We omit the algorithm to save space.

EXAMPLE 4.1. *For the XML document in Fig.5(a) under Tag+Level streaming there are seven streams: T_A^1 :$\{a_1\}$, T_A^2 :$\{a_2\}$, T_E^2 :$\{e_1\}$, T_B^2 :$\{b_2\}$, T_B^3 :$\{b_1\}$, T_D^3 : $\{d_1, d_2, d_3\}$ and T_C^4 :$\{c_1, c_2\}$. For the twig pattern in Fig.5(b) we have $U_{T_D^3}$ is $\{T_D^3\}$, $U_{T_C^4}$ is $\{T_C^4\}$, $U_{T_B^2}$ is $\{\}$, $U_{T_B^3}$ is $\{T_B^3, T_C^4\}$, $U_{T_A^1}$ is $\{\}$, $U_{T_A^2}$ is $\{T_A^2, T_B^3, T_C^4, T_D^3\}$.*

So the final useful streams are $U_{T_A^2} \bigcup U_{T_A^1}$ and the two streams T_A^1 and T_B^2 are pruned.

Given a twig pattern query Q and a set of streams under some streaming scheme, we say those streams surviving pruning *useful* streams. It is obvious that we only need to search *useful* streams for matches and from now on all the streams mentioned in the remainder of this paper are assumed to be *useful*.

5. XML STREAMING SCHEMES AND TWIG PATTERN MATCHING

As we have known, *TwigStack* based on Tag Streaming is optimal for A-D only twig patterns. With the help of more sophisticated streaming schemes, in this section we show that a larger class of twig patterns can be processed optimally.

5.1 XML Streaming Model and Twig Pattern Matching

Based on the simple head-element-access-only XML streaming model, we have to decide if a head element is in a match to a given twig pattern before we can move to the next element in the stream. However, the difficulty to devise efficient XML twig pattern matching method lies in the fact that we can not determine only from the *head elements* of various streams if any head element is in a match to a given twig pattern. Instead, the head elements of some streams *may* form a match to a given twig pattern with *tail* portions of other streams. However, since we can not access the tail portions of streams, any premature declaration saying such head elements are *indeed* in some matches can result in misjudgement and in consequence redundant intermediate output.

EXAMPLE 5.1. *Example 2.1 shows that none of the head elements of the three streams in Fig. 3(b) are in matches to $A[/B]/C$ which consists entirely of current head elements. Neither we can tell from the current head elements that any-one of them is* not *in a match: a_1, b_1 may form a match with element behind c_1 (which is indeed c_{n+1}); c_1 may form a match with element after a_1 and a_2 (which are indeed a_2 and b_2). Under such case, to store any head element may result in redundant intermediate paths whereas to discard any head element may cause loss of matches.*

If we use Tag+Level streaming (Fig. 4(a)) for the document in Fig. 3(a), the above problem does not arise anymore because now we have a match $< a_1, b_1, c_{n+1} >$ which consists of only head elements of their respective streams. Therefore a_1 is determined to be a match. Note that the documents in Fig. 3(a) and (c) now can be distinguished by Tag+Level streaming and we can determine for sure that a_1 is not in any match using Fig. 4(b).

Example 5.1 shows that: due to the introducing of P-C edge in twig patterns, matches to a twig pattern may not entirely consist of current head elements of streams under Tag Streaming. Because of our XML stream model, matches consisting of *tail* portions of streams can only be regarded as *possible* but *not guaranteed* matches. The existence of such *possible matches* means possibility of introducing of redundant intermediate results in tag streaming scheme if

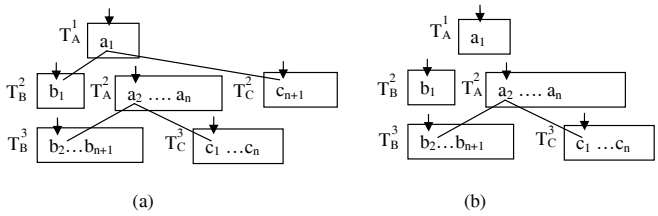


Figure 4: Tag+Level Streaming for files in Fig. 3

we do not want to lose any matches. In the next subsection, we will define formally the concept of *possible but not guaranteed match* under our XML stream model.

5.2 Possible Twig Pattern Match

Based on our XML stream model, we define *possible match* to a twig pattern Q. We emphasize that a possible twig pattern match may not be a real match and its existence is due to the restricted access mechanism of XML stream.

DEFINITION 5.1. *(Possible match of a twig pattern Q) A tuple t with n fields is called a* possible match *of a twig pattern Q if:*

1. *Each field t.q of t corresponds to one node q of Q. The value of t.q can either be $head(T_q)$ or $tail(T_q)$ where T_q is a stream of class q; and*
2. *For each field t_q, for each child (if any) q_i of q, the streams T_q and T_{q_i} (which t_{q_i} corresponds to) satisfy the edge $< q, q_i >$ (see the definition in Section 3.1) and*
   (a) *If t_q and t_{q_i} are of value $head(T_q)$ and $head(T_{q_i})$, then $head(T_{q_i})$ is a parent or ancestor of $head(T_q)$ depending on the edge $< q, q_i >$.*
   (b) *If t_q is of value $head(T_q)$ and t_{q_i} is $tail(T_{q_i})$, then $head(T_q).end > head(T_{q_i}).start$.*
   (c) *If t_q is of value $tail(T_q)$ and $t.q_i$ is $head(T_{q_i})$, then $head(T_q).start < head(T_{q_i}).start$.*
   (d) *If t_q is of value $tail(T_q)$ and t_{q_i} is $tail(T_{q_i})$, there is no additional restriction.*

An important difference between a *possible match* and a *match* to a twig pattern is that the former can always be told from the current head elements as seen from conditions in the definition whereas the later may not. Informally, the reason why we call the tuple satisfying the above conditions a possible match is that the tuple can be *instantiated* to matches to Q. If a possible match of Q consists of only head elements, it is called a *minimal match*; otherwise it is called a *blocked match*. Analogously, we can have similar definition to *possible match* to a sub-twig Q_q of Q rooted at node q.

For the sake of simplicity, we always use the current head element of stream T instead of symbol $head(T)$ in a possible match. We also append an imaginary "end" element (whose *startPos* and *endPos* are all infinity) to the end of each stream so that the definitions of $head(T)$ and $tail(T)$ can be applied on single-element streams.

EXAMPLE 5.2. *For the XML document in Fig. 5(a), under Tag Streaming ,suppose all streams have not advanced, the tuple $t :< a_1, d_1, b_2, tail(T_C^4) >$ is not a possible match for the query Q in Fig. 5(b) because T_B^2 and T_C^4 do not satisfy edge B/C. However, the tuple is a possible but not minimal match for the query Q' in Fig. 5(c) because $b_2.endPos > c_1.startPos$.*

As another example, the tuple $t' : < tail(T_A^2), d_1, b_1, c_1 >$ is not a possible match for Q because $a_2.startPos > d_1.startPos$. However, it is trivial that d_1 is a possible and also minimal match *for Q_D where Q_D is a sub-query of Q rooted at D.*

EXAMPLE 5.3. *For the XML document in Fig. 6(a), under PPS streaming, there are 8 PPS streams: $T_A : \{a_1\}$, $T_{AB} : \{b_1, b_2, b_3\}$, $T_{ABA} : \{a_2, a_3, a_4\}$, $T_{ABD} : \{d_2\}$, $T_{ABAC} : \{c_1, c_2\}$, $T_{ABAB} : \{b_4, b_5\}$, $T_{ABABD} : \{d_1\}$, $T_{ABABE} : \{e_1, e_2\}$.*

Suppose no stream has yet moved, then the tuple $< b_1, tail(T_{ABABE}), tail(T_{ABD}) >$ is not a possible match for Q''_B. Suppose now we move the stream T_{AB} to b_3 and T_{ABA} to a_3. The tuple $< a_1, c_1, b_3, d_2, tail(T_{ABABE}) >$ is a possible match for Q''. The tuple $< a_3, Tail(T_{ABAC}), b_4, d_1, e_1 >$ is also a possible match for Q''.

5.3 Classifying Head Elements

Using the concept of *possible match*, we classify the current *head elements* of useful streams to the following three types with respect to a twig pattern Q:

1. (Matching element) Element e of type E is called a matching element if e is in a minimal match to Q_E but not in any possible match to Q_P where P is the parent of E or E is the root of Q.
2. (Useless element) Element e is called a useless element if e is not in any possible match to Q_E.
3. (Blocked element) Otherwise e is a blocked element or we say e is blocked.

Informally, we can think a *useless element* as an element that can be thrown away safely. A *matching element* e is an element which is at least in a match to Q_E and we can tell if it is in a match to Q in an efficient way which we will discuss in the section 6 *Algorithm*.

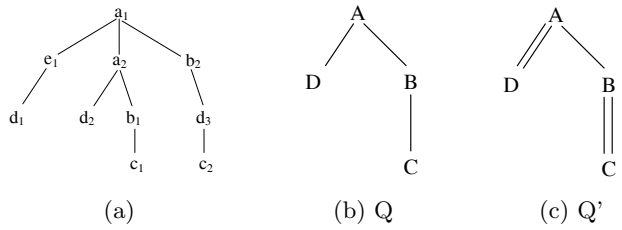
Notice that a head element e is blocked if e is in a possible but not in any minimal match to Q_E or e is in a minimal match to Q_E but is some possible match to Q_P where P is the parent of E. In the first case, e is not guaranteed to be in a match. In the later case, if e is in a possible match to Q_P, taking away e first can cause errors in determining if elements which are ancestor of e are indeed in possible matches to Q_P. Therefore it is unsafe to advance the stream where e is the head.

For an optimal twig pattern matching algorithm to proceed, it should never happen that all current head elements are *blocked* because in such a situation we can not advance any stream without storing elements which are not guaranteed in a match.

Notice that Example 2.1 gives an example where all head elements are blocked under Tag Streaming.

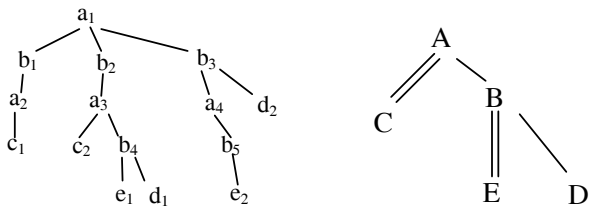
Although better than Tag Streaming, the following example shows that there are also queries under which all head elements are blocked under Tag+Level streaming.

EXAMPLE 5.4. *Under Tag+Level Streaming, for the XML file in Fig.5(a), Q in Fig.5(b) and Q' in Fig.5(c), suppose no stream cursor has moved, we have*



(a) (b) Q (c) Q'

Figure 5: A Sample XML Document and Two Queries



(a) (b)Q"

Figure 6: A Sample XML Document for which PPS also can't help

Head	*Q*	*Q'*
a_1	*Useless*	*Blocked*
a_2	*Blocked*	*Blocked*
b_1	*Blocked*	*Blocked*
b_2	*Useless*	*Blocked*
c_1	*Blocked*	*Blocked*
d_1	*Matching*	*Blocked*

For instance, for Q', d_1 is blocked because it is in a possible match $< d_1, a_1, b_2, tail(T_C^4) >$ to Q'_A.

The following example shows that PPS streaming can avoid some *all blocked* situation occurring in $Tag + Level$ streaming but also has its limitation.

EXAMPLE 5.5. *Under PPS Streaming, for the XML file in Fig.5(a) and Q' in Fig.5(c), different from Tag+Level streaming, we have both a_1 and a_2 as matching elements. The main reason is that d_1 and d_2 are now in two different PP streams T_{AED} and T_{AAD} and so are c_1 and c_2.*

However, for the document in Fig.6(a) and the query in Fig. 6(b), suppose the stream T_{AB} advances to b_3 and T_{ABA} advances to a_3. We have all head elements, namely, a_1, a_3, b_3, b_4, d_1, e_1, c_1 and c_2 blocked.

5.4 Properties of Different Streaming Techniques

The last subsection gives us some negative results and thus limits of various streaming schemes. Now we try to prove that for a certain class of twig pattern query, a particular streaming scheme can prevent the situation whereby all head elements are blocked. This can be seen as a *necessary* condition for optimal twig pattern matching. Now first we introduce an auxiliary lemma

LEMMA 5.1. *Given two streaming schemes α and β, suppose α is a refinement of β. Suppose we have a set of streams in β being further partitioned under α, we have*

1. *A matching head element in β is also a matching head element in α.*
2. *A useless head element in β is also a useless head element in α.*

It is easy to find example to prove that the opposite direction in each of the two conclusions in the lemma is not true.

In proving the following lemmas, we use the following important observations: a blocked or useless element for sub-twig pattern Q_q of Q is also a blocked or useless element for Q; if e is a matching element for sub-twig pattern Q_q of Q but not in possible match to Q_q, it is also a matching element for Q.

LEMMA 5.2. *Under Tag Streaming, for an A-D only query Q, there always exists a head element which is either a matching or useless element for Q.*

PROOF. We use induction on the sub-queries of Q.

(Base case) Suppose a node q in Q which is the parent of leaf nodes $q_1, q_2, \ldots, q_n$ and T_q has not ended. Note that if any stream T_{q_i} has ended, $head(T_q)$ is a useless element for Q_q and we are done. Otherwise all streams $T_{q_1}, \ldots, T_{q_n}$ are not end and it is obvious that each sub-twig Q_{q_i} for i from 1 to n has a minimal match. There are the following cases:

1. If $head(T_q).endPos < head(T_{q_i}).startPos$ for some q_i, then $head(T_q)$ is a useless element for Q_q.
2. Else if $head(T_q).startPos > head(T_{q_i}).startPos$ for some q_i, then $head(T_{q_i})$ is a matching element for Q_q but is not in possible match to Q_q.
3. Otherwise $head(T_q)$ is ancestor of $head(T_{q_i})$ for each q_i and $head(T_q)$ is a matching element for Q_q.

Note that in the base case we can find either a useless or matching element w.r.t sub-twig Q_q.

(Induction) Suppose a node q in Q has child nodes $q_1, \ldots, q_n$, by the induction hypothesis, if there is a node q_i for which $head(T_{q_i})$ is not a matching element for Q_{q_i}, we must find either a useless element for Q_{q_i} (which is also a useless element for Q_q) under the above case 1 or a matching element for a sub-twig of Q_{q_i} but not in possible match to Q_{q_i} (which is also a matching element for Q_q) under case 2 and thus done. Otherwise each $head(T_{q_i})$ is a matching element for Q_{q_i}, we proceed with the same argument (with the three cases) in the base case. Note that the induction step ends when q is the root of Q and in such case $head(T_q)$ is a matching element we are done. □

LEMMA 5.3. *Under Tag+Level streaming, for an A-D only or P-C only query Q, there always exists a head element which is either a useless or matching element for Q.*

PROOF. (Sketch) According to Lemma 5.1, for an A-D only twig pattern, the above statement is true.

Given a P-C only query Q with n nodes, we can partition the streams into a few groups each of which has n streams and contains possible matches to Q. For example, streams T_A^2, T_D^3, T_B^3 and T_C^4 form a group for the query $A[/D]/B/C$ in Fig. 5. Notice that it is impossible that two elements from streams of different groups can be in the same match. For each group of n streams, we can perform the same analysis as in the proof of Lemma 5.2 to find out either a useless or matching element for Q. □

LEMMA 5.4. *Under Prefix-Path streaming, for an A-D only or P-C only and one branchnode only queries, there always exists a head element which is either a useless or matching element.*

PROOF. (Sketch)According to Lemma 5.1, for a A-D or P-C only twig pattern, the above statement is true. For a one branchnode only query,we first prove the special case where the root node is also the branch node. Suppose the PPS stream T_{min} is the one whose head element with the smallest *startPos* among all streams.

1. T_{min} is of class q where q is not the root of Q and L is the label of T_{min}. Suppose q_l is a leaf node of Q and descendant of q. Suppose $T_{q_l}^{min}$ is a stream of class q_l having label L' with the following properties: (1) $L' = L + L''$ and the string L'' matches the path query Q_q (2) $T_{q_l}^{min}$ has the head element with the minimum *startPos* among all streams of class q_l satisfying property (1). Now if $head(T_{min})$ is not an ancestor of $head(T_{q_l}^{min})$, then $head(T_{min})$ is a useless element and we are done. Otherwise take any stream T_m of class q_m where q_m is a node between q and q_l and having a label which is prefix of L' and for which L is a prefix. If for any such T_m, $head(T_m).endPos < head(T_{q_l}^{min}).startPos$ then $head(T_m)$ is a useless element; otherwise $head(T_{min})$ is a matching element.

2. T_{min} is of class q where q is the root of Q. We can consider the leaf nodes for each branch of Q and repeat the same argument as (1) for each branch.

In the case where the branch node is not the root node, we can reduce it to the first case. But to save space we omit it here. □

Interested readers may wonder if there is a streaming scheme whereby the *all blocked* situation never occurs, we point out that at least the costly $FB-BiSimulation$ scheme [11] is able to do that because the scheme is so refined that from the label (or index node) of each stream we can tell if all elements in the stream are in a match or not.

6. TWIG JOIN INDEXED XML DOCUMENT

In this section, we describe a twig pattern matching method $iTwigJoin$ applicable to any streaming schemes discussed so far based on the notion of possible match. Our method can work *correctly* for all twig patterns and meanwhile they are also *optimal* for certain classes of twig patterns depending on the streaming scheme used.

6.1 Main Data Structures

There are two important components in our twig pattern matching algorithm, namely: (1) A stream management system to control the advancing of various streams. (2) A temporary storage system to store partial matching status and output intermediate paths.

The role of the temporary storage system can be summarized as follows: it *only* keeps elements from streams which are in possible matches with elements which are still in the streams. The elements in the temporary storage system has dual roles: (1) they will be part of intermediate outputs (2) when a new element e with tag E is found to be in a possible match to sub-twig Q_E of twig pattern Q, we can know if e is in a possible match to Q by checking if e has a parent or ancestor element p in the temporary storage which is a possible match to Q_P where P is the parent node of E in Q. Similar to $TwigStack$, we associate each node q in a twig pattern with a stack S_q. At any time during computation, all elements in a stack are located on the same path in source XML documents. The property is ensured through the following *push* operation: when we *push* a new element e with tag E into its stack S_E, all the elements in the stack which are not ancestor of e will be pop out.

As for the stream management system, depending on different streaming schemes, each node q is associated with all useful streams of class q. Each stream has the following operations: $head(T)$ returns the head element of stream T; $T.advance()$ moves the stream cursor to the next element.

6.2 Algorithm: iTwigJoin

6.2.1 Overview

The flow of our algorithm $iTwigJoin$ is similar to that of $TwigStack$. In each iteration, an element e is selected by the stream management system from the remaining portions of streams. To avoid redundant intermediate output, we always try to select a *matching* element unless all head elements are *blocked*. The element is then be used to update the contents of stacks associated with each query node in the twig pattern. The detail of updating process will be discussed shortly. During the update, partial matching paths will be outputted as intermediate results. The above process ends when all streams corresponding to leaf nodes the twig pattern end. After that, the lists of intermediate result paths will be merged to produce final results.

6.2.2 Algorithm Details

We divide our algorithm into two parts. One for the stream management system (Algorithm 2) and another for the temporary storage system(Algorithm 1).

The stream management system of $iTwigJoin$, in each iteration, discards useless elements and selects a head element e of tag E from the remaining portions of streams with the following two properties:

1. e is in a possible match to Q_E but not in a possible match to Q_P where P is the parent of E in Q. (Notice that e can be either a matching element or a blocked element but not a useless one.)

2. e is the element with the smallest *startPos* among all non-useless head elements of tag E' where E' is a node in the sub-twig Q_E.

The first property guarantees that the element e is at least in a possible match to Q_E. Equally importantly, a parent/ancestor element in a match is always selected before its child/descendant by the property. The second property is important to ensure that the space used our temporary storage system is bounded as we will explain in the next section.

Before studying in detail how the stream management system works, let us first look at the temporary storage system in Algorithm 1. After the element e with tag E is selected (line 3) , we first pop out elements in S_E and $S_{parent(E)}$ whose *endPos* is smaller than $e.startPos$ (line 5) as they are guaranteed to have no more matches as we will prove shortly. In line 6, we check in our temporary storage system

if there is an element in stack S_P which is parent or ancestor of the selected element e (depending on edge $< P, E >$) where P is the parent node of E. If there is such an element, e is then pushed into S_E (line 7) and if E is a leaf node a number of intermediate paths containing e as the leaf element are output (line 9); otherwise e is discarded.

Now we study how the stream management system works. The function call $getNext(root)$ (Algorithm 2) plays the role of selecting an element from the remaining portions of the streams with the two aforementioned properties. It works recursively: (1) For the base case where q is a leaf node in Q, it just returns q (line 1-2). (2) Suppose q has children q_1,q_2,...,q_n, we first call $getNext(q_1)$,...,$getNext(q_n)$ (line 5). If any of the recursive call does not return q_i, we have found the element because it satisfies the above two properties mentioned w.r.t Q_{q_i} and is not in a possible match to Q_{q_i}; thus it also satisfies the two properties w.r.t Q_q and consequently Q. So it returns what the call returns (line 6-7). Otherwise for each stream T_q^j of class q, for each child node q_i of q from 1 to n, in line 11-12 we find the stream $T_{q_i}^{min}$ which is of class q_i and also has the smallest $startPos$ among all *solution streams* of T_q^j for edge $< q, q_i >$. Let T_{max} be the stream whose head has the largest $startPos$ among $T_{q_i}^{min}$ for i from 1 to n (line 14) . We next advance T_q^j until $head(T_q^j).endPos > head(T_{max}).startPos$ (line 15-17). Notice that all elements skipped are *useless* elements because they are not in *possible match* to Q_q. After the advancing, $head(T_q^j)$ is in a possible match to Q_q and can be either *blocked* or *matching*. Finally, let T_q^{min} be the current stream of class q with the smallest $startPos$ and T_{child}^{min} be the stream with the smallest $startPos$ among ALL streams of class q_c where q_c is any child node of q. If $head(T_{child}^{min}).startPos < head(T_q^{min}).startPos$, the element $head(T_{child}^{min})$ satisfies the two properties aforementioned: it will not be in any possible matches to Q_q because $head(T_{child}^{min})$ $.startPos < head(T_q^{min}).startPos$ and the satisfaction of property 2 is obvious. Thus the child node is returned. Otherwise q is returned and the recursion proceeds because $head(T_q^{min})$ may be in a match to $Q_{parent(q)}$.

EXAMPLE 6.1. *For the sample XML document in Fig.5(a) and the twig pattern query Q' in Fig.5c, the PPS streaming scheme can provide an optimal solution. The following table traces the entire matching process with the elements selected in each iteration by $getNext(root)$ and the corresponding stack operation. The word "push" means an element of tag E is pushed into its stack S_E.*

Note that all elements selected are matching *elements. As an example, when a_1 is selected, it is in the match $< a_1, d_1, b_2, c_2 >$ which are all head elements of their Prefix Path streams.*

Step	*Selected*	*Stack Operation*
1	a_1	*push*
2	d_1	*push , output* a_1/d_1
3	a_2	*push*
4	d_2	*push , pop* d_1,*output* a_1/d_2,a_2/d_2
5	b_1	*push*
6	c_1	*push , output* $a_2/b_1/c_1$
7	b_2	*push*
8	d_3	*push , pop* d_2, *output* a_1/d_3
9	c_2	*push , pop* c_1, *output* $a_1/b_2/c_2$

On the other hand, for the query Q'' in Fig. 6b and the document in Fig. 6a, $iTwigJoin$ based on PPS streaming scheme is no longer optimal because Q'' has two branchnodes. For example, when a_1 is selected, it is only in the non-minimal possible match $< a_1, c_1, b_3, d_2, tail(T_{ABABD}) >$. Notice that the current head element of stream T_{ABABD} is e_1, which is also in a possible match. However, if the $tail(T_{ABABD})$ does not contain the element e_2 , we will output two redundant intermediate paths: a_1/c_1 and $a_1/b_3/d_2$. Of course if we just discard a_1 we may lose the two paths if e_2 is there instead.

Even though our algorithm $iTwigJoin$ is not optimal in this case, compared with $TwigStack$, it still output less redundant intermediate paths: note that $TwigStack$ will also output the redundant path $a_1/b_2/e_1$ because $< a_1, c_1, b_2, e_1, d_1 >$ is in a match to $A[//C]//B[//D]//E$.

Step	*Selected*	*Stack Operation*
1	a_1	*push*
2	c_1	*push , output* a_1/c_1
3	a_3	*push*
4	c_2	*push , pop* c_1,*output* a_3/c_2
5	b_4	*push*
6	e_1	*push , output* $a_3/b_4/e_1$, $a_1/b_4/e_1$
7	d_1	*push , output* $a_3/b_4/d_1$
8	b_3	*push , pop* b_4
9	e_2	*push , pop* e_1, *output* $a_1/b_3/e_2$
10	d_2	*push , pop* d_1, *output* $a_1/b_3/d_2$

$iTwigJoin$, applied on Tag Streaming, is essentially identical to $TwigStack$. Except the definition of *Solution Streams* (Algorithm 2 line 12), *$iTwigJoin$* is independent of the underlying streaming scheme used. Thus given a new streaming scheme (assuming elements in the stream are of the same tag and ordered in document-order) other than the three discussed, *$iTwigJoin$* is still applicable after we work out the new definition of solution streams which is often quite easy.

Algorithm 1 *iTwigJoin*

```
1: Prune-Streams(Q) //See Section 4
2: while ¬end(root) do
3:    q = getNext(root);
4:    T_min = the stream with the smallest startPos among all
      streams of class q
5:    pop out elements in S_q and S_parent(q) which are not an-
      cestor of head(T_min)
6:    if isRoot(q) ∨ existParAnc(head(T_min),q) then
7:       push head(T_min) into stack S_q
8:       if isLeaf(q) then
9:          showSolutionWithBlocking(S_q); //See TwigStack for
            details
10:      end if
11:   end if
12:   advance(T_min)
13: end while
14: mergeAllPathSolutions()
```

Function: end(QueryNode q)

1: return **true** if all streams associated with leaf nodes of Q_q end;
2: Otherwise return **false**;

Function: existParAnc (Element e, Node q)

1: return **true** if e has a parent or ancestor element in stack $S_{parent(q)}$ depending on edge $< parent(q), q >$)

6.3 Algorithm Analysis

Algorithm 2 *getNext(q)*

```
1: if isLeaf(q) then
2:    return q
3: end if
4: for for i = 1 to n do
5:    q_x =getNext(q_i) //q_1,...,q_n are children of q
6:    if q_x <> q_i then
7:       return q_x
8:    end if
9: end for
10: for each stream T_q^j of class q do
11:    for i = 1 to n do
12:       T_{q_i}^{min} = min(soln(T_q^j,q_i)); //See definition 3.1 for soln
13:    end for
14:    T_max = max({T_{q_1}^{min},...,T_{q_n}^{min}});
15:    while head(T_q^j).endPos < head(T_max).startPos do
16:       T_q^j.advance();
17:    end while
18: end for
19: T_q^{min} = min(streams(q))
20: T_{child}^{min} = min(streams(q_1)⋃... ⋃streams(q_n));
21: if T_q^{min}.startPos < T_{child}^{min}.startPos then
22:    return q;
23: end if
24: return q_c where T_{child}^{min} is of class q_c
```

Function: min(a set of streams)

1: return the stream with the smallest *startPos* in the set

Function: max(a set of streams)

1: return the stream with the largest *startPos* in the set

In this section, we first prove the correctness of our algorithm. Next we are going to show that depending on streaming schemes used, our algorithm is optimal for several important classes of twig pattern queries.

LEMMA 6.1. *The $getNext(root)$ of Algorithm iTwigJoin returns all elements which are in matches to a given twig pattern Q.*

Essentially, we can show that the $getNext(root)$ call of our algorithm returns all elements e of tag E which are in possible match to sub-query Q_E of Q, which is a superset of elements in matches to Q. The most important observation is that Property 1 of the element returned by $getNext()$ guarantees that a parent/ancestor element in a possible match is always returned before its child/descendant element.

LEMMA 6.2. *In Algorithm iTwigJoin, when an element is popped out of its stack, all its matches have been reported.*

PROOF. An element e of tag E is popped out of its stack S_E because we push into S_E or child stack S_C (line 5 of Algorithm 1) an element e' and $e'.startPos > e.endPos$. Suppose e has some matches yet output, there must exist a child/descendant element c (with tag C) of e not yet be returned by $getNext(root)$. It is easy to see that $e'.startPost > c.endPos$ too. Since c will also be in a possible match to Q_C, by the second property of the $getNext(root)$ function, c will be returned before e' because $c.startPos < e'.startPos$. Contradiction. □

The above two lemmas show all elements in matches will be reported by our Stream Manager and no element will be removed from our temporary storage system before all its matches have been reported. Thus we can come to the follow theorem

THEOREM 6.3. *The algorithm iTwigJoin correctly reports all matches to a given twig pattern .*

The following lemma shows that the space complexity of our algorithm is bounded.

LEMMA 6.4. *Algorithm iTwigJoin uses space bounded by $|Q| * L$ where L is the longest path in the XML source document and $|Q|$ is the number of nodes in Q.*

This is easy to see because all the elements in any stack are located on the same path of the source document.

The following lemma shows the optimality of our algorithm for certain classes of twig pattern queries depending on streaming schemes. The essential idea is that under the combination of twig pattern types and streaming schemes, every $getNext(root)$ call only returns an element which is a *matching element*. Because a matching element e of tag E is in a *real* match, no redundant intermediate results will be outputted. The proofs can be extended from the proofs of Lemma in Section 5.4.

LEMMA 6.5. *Depending on streaming schemes, our algorithm iTwigJoin is optimal in the following classes of queries:*

1. *Tag Streaming: A-D only twig pattern.*
2. *Tag + Level Streaming: A-D or P-C only twig pattern.*
3. *Prefix-Path Streaming: A-D or P-C or one branch-node only twig pattern.*

LEMMA 6.6. *The CPU time of iTwigJoin is $O(no\_streams \times |Q| \times |INPUT + OUTPUT|)$ where no_streams is the total number of useful streams for the twig pattern query Q.*

In the actual implementation, for stream T_q^j of class q, we keep a number $min(T_q^j, q_i)$ for each child edge of q: $< q, q_i >$ to keep track of the minimum *startPos* of head elements of streams in $soln(T_q^j, q_i)$. Notice the number is used in Line 12 of algorithm 2. Notice that when a stream advances, at most O($no\_streams$) *min* numbers will be updated.

Since each element is scanned only once, we calculate the CPU time by bounding the time interval from the previous element scan event to the current one. There are two places to scan an element in the program: Line 12 of Algorithm 1 and Line 16 of Algorithm 2. Notice that if the current scan occurs in Line 16 of Algorithm 2, the time lapse after the previous scan event is at most O($no\_streams$) for update those $min(T_q^j, q_i)$ of various streams and at most O($|Q|$) on lines 10 to 15 of Algorithm 2. If we scan an element in Line 12 of Algorithm 1, the maximum time interval is O($|Q| * no\_streams$) when the previous scan also occurs at line 12 of algorithm 1. Therefore the total CPU time spent is $O(no\_streams \times |Q| \times |INPUT + OUTPUT|)$ when added in the output size.

7. EXPERIMENTS

In this section we present experimental results. We first apply the two streaming schemes (i.e. Tag+Level and PPS) to XML documents with different characteristics to demonstrate their applicability of different kinds of XML files.

	XMark	Treebank
Size	113MB	77MB
Nodes	2.0 million	2.4 million
Tags	77	251
Max Depth	12	36
Average Depth	5	8
No. of Streams using Tag+Level	119	2237
No. of streams using PPS	548	338740

Table 1: XML Data Sets used in our experiments

Next we conduct a comprehensive study of twig pattern processing performances based on various streaming schemes and the algorithms we discussed. Our experiment results show significant advantages of new XML streaming schemes and twig join algorithms over the original $TwigStack$ approach and its recent variant.

7.1 Experiment Settings and XML Data Sets

We implemented all algorithms in Java 1.5. All our experiments were performed on a system with 2.4GHz Pentium 4 processor and $512MB$ RAM running on windows XP. We used the following real-world (i.e. TreeBank [16]) and synthetic data sets (i.e. XMark [18]) for our experiments: (1) *XMark* This well-known XML data set is synthetic and generated by an XML data generator. It contains information about an auction site. Its DTD is recursive. (2) *TreeBank* We obtained the $TreeBank$ data set from the University of Washington XML repository [16]. The DTD of Treebank is also recursive. TreeBank consists of encrypted English sentences taken from the Wall Street Journal, tagged with parts of speech. The deep recursive structure of this data makes it ideal for experiments of twig pattern matching algorithms. Their main characteristics can be found in Table 1.

The reason why we select the above two XML data sets is because they represent two important types of data: $XMark$ is more "information oriented" and has many repetitive structures and fewer recursions whereas $TreeBank$ has inherent tree structure because it encodes natural language parse trees.

7.1.1 Number of Streams Generated by various Streaming Techniques

Table 1 also shows the statistics of applying *Tag+Levl* and *Prefix-Path* streaming schemes. It is easy to see that on an information-oriented data source like XMark, the numbers of streams resulted from *Tag+Level* as well as *Prefix-Path* streaming are small compared with the total number of nodes (2 million) in the document. This shows that in the document, most of the elements with the same tag appear in relatively few different "contexts". On the other hand, in a much more deep recursive data like $Treebank$, *Tag+Level* still results in relatively few number of streams compared with element numbers (2.4 million). However the number of streams under *Prefix-Path* streaming is so large that it is nearly *16%* of the number of elements.

The above data shows that $Tag + Level$ can be applied to a wider range of XML documents whereas $Prefix - Path$ streaming is better to use in more information-oriented XML data.

7.2 Twig Pattern Matching on various streaming schemes

XMark1	//site/people/person/name
XMark2	//site//people//person[//name][//age]//income
XMark3	//text[/bold]/emph/keyword
XMark4	//listitem[//bold]/text//emph
XMark5	//listitem[//bold]/text[//emph]/keyword
Tree1	S//ADJ[//MD]
Tree2	S[/JJ]/NP
Tree3	S/VP/PP[/NP/VBN]/IN
Tree4	S/VP//PP[//NP/VBN]/IN
Tree5	S//NP[//PP/TO][/VP/_NONE_]/JJ

Table 2: Queries used in our experiments

	T+L	T+L pruning	PPS	PPS pruning
XMark1	7	4	17	4
XMark2	9	7	19	6
XMark3	27	24	330	132
XMark4	24	19	249	144
XMark5	31	23	348	162
Tree1	62	46	12561	1714
Tree2	91	81	78109	18
Tree3	177	138	123669	474
Tree4	177	138	123669	1876
Tree5	209	175	132503	1878

Table 3: Number of Streams Before and After Pruning for XMark and TreeBank Datasets

7.2.1 Queries

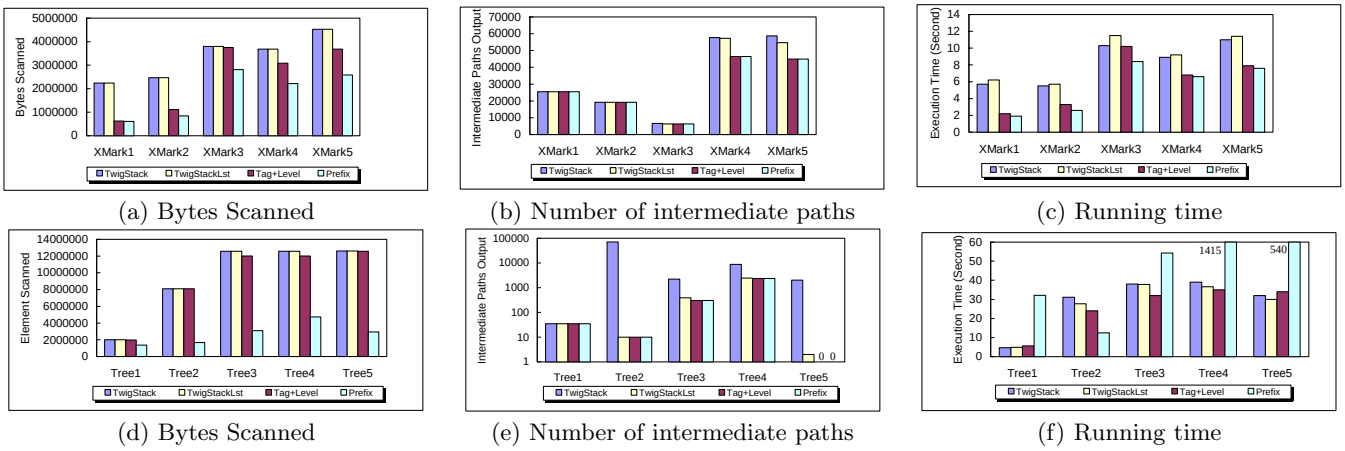
We select representative queries (shown in Table 2) which cover the classes of twig pattern query that fall within and outside the optimal sets of different streaming schemes.

The selected queries over the XMark dataset include: (1) a Path query (XMark1) (2)an A-D only query (XMark2) (3)a P-C only query (XMark3) (4) One branchnode (but neither A-D nor P-C only) query (XMark4) (5) A Query (XMark5) which does not fall in the above four types and are not theoretically optimal under any of our streaming schemes.

The selected queries over the TreeBank dataset include: (1) an A-D only query (Tree1) (2) two P-C only queries (Tree2 and Tree3) (3) two queries (Tree4 and Tree5) which do no fall in the above two categories and are not theoretically optimal under Tag+Level streaming.

7.2.2 Performance Measures

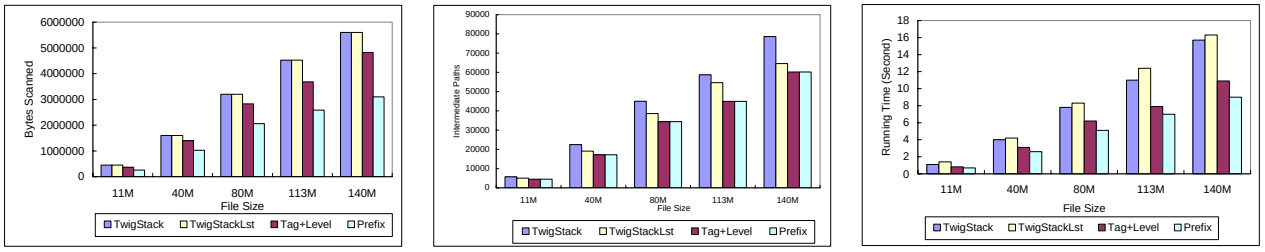
We compare four algorithms: namely $TwigStack$ on Tag Streaming scheme, a recent proposed variant of $TwigStack$: $TwigStackLst$ [14] and $iTwigJoin$ on Tag+Level and Prefix-Path Streaming respectively. $TwigStackLst$ is also based on Tag Streaming. It is different from $TwigStack$ by allowing look ahead a limited amount of elements in a stream to avoid redundant intermediate paths. The method is shown not be optimal for P-C only or 1-branchnode only twig pattern. We consider the following performance metrics to compare the performance of twig pattern matching algorithms based on three streaming schemes: (1) Number of elements scanned (2) Number of intermediate paths produced (3) Running time. We also record the number of streams whose tags appear in the twig pattern and the number of useful streams after streaming pruning for each query under different streaming schemes in Table 3.



(a) Bytes Scanned (b) Number of intermediate paths (c) Running time

(d) Bytes Scanned (e) Number of intermediate paths (f) Running time

Figure 7: Performance Comparison over XMark (a-c) and TreeBank (d-f) datasets; the number of intermediate paths output by Tag+Level and PPS is also the number of merge-joinable paths



(a) Bytes Scanned (b) Number of intermediate paths (c) Running time

Figure 8: Performance Comparison of four algorithms over XMark of different sizes using XMark5

7.2.3 *Scalability*

We also test the four algorithms on data sets of different sizes. We present in Fig. 8 the performance results tested on XMark benchmark size of 11MB, 40MB, 80MB and 113MB and 140MB on the twig pattern $XMark5$.

7.3 Performance Analysis

In terms of number of bytes scanned (Fig. 7(a) and (d)), based on the XMark benchmark, we can see that both PPS and Tag+Level can prune large portions of irrelevant data: PPS from 40% to 300% and Tag+Level from 4% to 250%. Meanwhile, PPS can prune more data than Tag+Level. As for $Treebank$, Tag+Level saves fewer I/O (from 0% to 5%) compared with PPS.

With respect to the numbers of intermediate paths output by various algorithms (Fig. 7(b) and (e)), $iTwigJoin$ based on PPS and Tag+Level avoids redundant intermediate paths produced by $TwigStack$ and $TwigStackLst$ based on Tag Streaming. For XMark, the reduction ratio goes up to 25% (XMark5) and for Treebank as high as 79800:10 (Tree2). A somewhat surprising result is that although there are queries which fall outside of the theoretical optimal classes of Tag+Level and PPS (e.g. XMark3,Tree3 and Tree4 for Tag+Level and XMark5 for PPS), the numbers of intermediate paths output by Tag+Level and PPS for these queries are also the numbers of merge-joinable paths! This shows that in real XML data sets the theoretical worse cases (which we construct in Section 5) seldom occur.

Combining the savings in both input I/O cost and intermediate result sizes, $iTwigJoin$ in general achieves faster running time (Fig. 7(c) and (f)). For XMark, $iTwigJoin$ based on PPS is *always* faster than that based on Tag+Level streaming which in turn is faster than that of $TwigStack$. For $Treebank$, $iTwigJoin$ based on Tag+Level streaming loses slightly (-5%) in A-D only query (Tree1) and the query $Tree5$ where there are over 175 streams involved but wins in other cases; however, except Tree2 where there are 18 streams left after pruning, $iTwigJoin$ based on PPS requires unacceptable running times (1412s for Tree4 and 540s for Tree5) because it needs to join too many streams. $iTwigJoin$, based on Tag+Level or PPS, needs to join more streams than that of $TwigStack$. This makes $iTwigJoin$ more CPU-intensive than that of $TwigStack$. Our experiment results show that even the number of streams goes up to 150, $iTwigJoin$ still has edges over $Twigstack$ because of the saving in input I/O and intermediate outputs.

8. RELATED WORK

Structural join is essential to XML query processing because XML queries usually impose certain structural relationships (e.g. P-C or A-D relationships). For binary structural join, Zhang et al [20] proposed a multi-predicate merge join (MPMGJN) algorithm based on region labelling of XML elements. The later work by Al-Khalifa et al [1] gave a stack-based binary structural join algorithm, called Stack-Tree-Desc/Anc which is optimal for an A-D and P-C binary relationship. Wu et al [17] studied the problem of binary join order selection for complex queries. More

recently, N. Bruno et al [2] proposed a holistic twig join algorithm, namely TwigStack, to avoid producing a large intermediate result. TwigStack is I/O optimal for queries with only ancestor-descendant edge. Choi et al [6] demonstrated that no matter how elements in a Tag Streaming stream are ordered, it is impossible to achieve optimal holistic matching. Lu et al [14] proposed a look-ahead method to reduce the number of redundant intermediate paths. Jiang et al [10] used an algorithm based on indexes built on element containment labels. The method can "jump" elements and achieve sub-linear performance for twig pattern queries. However it still does not solve the problem of redundant intermediate results in the presence of P-C relationship. Jiang et al [9] considered the problem of processing twig pattern query with *and* and *or* predicates. BLAS by Chen et al [5] proposed a dual labelling scheme: D-Label exactly the same as the region coding and P-Label for accelerating P-C relationship processing. The method decomposes a twig pattern into several P-C only path queries and then join the results.

Our XML streaming schemes follows another line of XML research on XML structural indexing. The PPS scheme is indeed a special case of Dataguide [7] used in tree-structured XML data. Milo et al [15] proposed *1-Index* to compute simulation and bisimilation sets of graph to partition data nodes. Kaushik et al [11] proposed the use of *Forward and Backward bisimilation* as a covering index for XML branch queries. Kaushik et al [13] gave a k-bisimilation partition method based on local similarity to reduce the index graph size. Chen et al [3] and He et al [8] further reduced the index graph sizes.

Recently Kaushik et al [12] proposed to process XML Path queries by integrating structural indexes and region label. However the approach still stores all elements of the same tag in a list and does not use holistic matching technique. Chen et al [4] first proposed to perform holistic twig pattern matching on XML document partitioned using PPS scheme and show its optimality in A-D only, P-C only and 1-Branchnode only twig patterns. However the method can *only* be applied to the above three classes of twig patterns.

9. CONCLUSIONS

In this paper, we apply XML structural indexing techniques to increase the amount of "holism" in XML twig pattern matching. We have developed theory to explain the optimal classes as well as limits of streaming schemes like Tag Streaming, Tag+Level Streaming and Prefix-Path Streaming: Tag+Level Streaming can be optimal for both A-D and P-C only twig patterns whereas PPS streaming can be optimal for A-D only, P-C only and one branchnode only twig patterns assuming there is no repetitive tag in the twig patterns. In general, we argue that a more refined streaming scheme can provide optimal solution for a larger class of twig patterns. We have developed a unified framework to perform twig pattern matching on all three streaming schemes discussed and we comment our algorithm is also applicable to all streaming schemes as long as the elements in a stream are ordered by document order. Our method reduces both input I/O cost and redundant intermediate result sizes and can achieve good performance.

10. REFERENCES

[1] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Proceedings of ICDE 2002*, pages 141–152, 2002.

[2] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *Proceedings of SIGMOD 2002*, pages 310–321, 2002.

[3] Q. Chen, A. Lim, and K. W. Ong. D(k)-index: An adaptive structural summary for graph-structured data. In *Proceedings of SIGMOD 2003*, pages 134–144, 2003.

[4] T. Chen, T. W. Ling, and C. Y. Chan. Prefix path streaming: a new clustering method for optimal XML twig pattern matching. In *Proceedings of DEXA 2004*, pages 801–811, 2004.

[5] Y. Chen, S. B. Davidson, and Y. Zheng. Blas: an efficient Xpath processing system. In *In Proceedings of SIGMOD Conference 2004*, pages 47–58, 2004.

[6] B. Choi, M. Mahoui, and D. Wood. On the optimality of the holistic twig join algorithms. In *In Proceeding of DEXA 2003*, pages 28–37, 2003.

[7] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of VLDB 97*, pages 436–445, 1997.

[8] H. He and J. Yang. Multiresolution indexing of XML for frequent queries. In *Proceedings of ICDE 2004*, pages 683–694, 2004.

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

[10] H. Jiang, W. Wang, H. Lu, and J. X. Yu. Holistic twig joins on indexed XML documents. In *In Proceeding of VLDB 2003*, pages 273–284, 2003.

[11] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In *Proceedings of SIGMOD 2002*, 2002.

[12] R. Kaushik, R. Krishnamurthy, J. F. Naughton, and R. Ramakrishnan. On the integration of structure indexes and inverted lists. In *Proceedings of SIGMOD 2004*, 2004.

[13] R. Kaushik, P. Sheony, P. Bohannon, and E. Gudes. Exploiting local similarity for efficient indexing of paths in graph structured data. In *Proceedings of ICDE 2002*, pages 129–140, 2002.

[14] J. H. Lu, T. Chen, and T. W. Ling. Efficient processing of XML twig patterns with parent child edges: a look-ahead approach. In *In Proceedings of CIKM Conference 2004*, pages 533–542, 2004.

[15] T. Milo and D. Suciu. Index structures for path expressions. In *Proceedings of ICDT 99*, pages 277–295, 1999.

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

[17] Y. Wu, J. M. Patel, and H. V. Jagadish. Structural join order selection for XML query optimization. In *Proceedings of ICDE 2003*, pages 443–454, 2003.

[18] XMARK. http://monetdb.cwi.nl/xml.

[19] XPath. http://www.w3.org/TR/xpath.

[20] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In *Proceedings of SIGMOD 2001*, pages 425–436, 2001.