On Label Stream Partition for Efficient Holistic Twig Join

Bo Chen[1], Tok Wang Ling[1], M. Tamer Özsu[2], and Zhenzhou Zhu[1]

[1] School of Computing, National University of Singapore
{chenbo, lingtw, zhuzhenz}@comp.nus.edu.sg
[2] David R. Cheriton School of Computer Science, University of Waterloo
tozsu@uwaterloo.ca

Abstract. Label stream partition is a useful technique to reduce the input I/O cost of holistic twig join by pruning useless streams beforehand. The *Prefix Path Stream (PPS)* partition scheme is effective for non-recursive XML documents, but inefficient for deep recursive XML documents due to the high CPU cost of pruning and merging too many streams for some twig pattern queries involving recursive tags. In this paper, we propose a general stream partition scheme called *Recursive Path Stream (RPS)*, to control the total number of streams while providing pruning power. In particular, each recursive path in *RPS* represents a set of prefix paths which can be recursively expanded from the recursive path. We present the algorithms to build *RPS* scheme and prune RPS streams for queries. We also discuss the adaptability of *RPS* and provide a framework for performance tuning with general *RPS* based on different application requirements.

1 Introduction

An XML document contains hierarchically nested elements, which can be naturally modeled as a labeled ordered tree. Standard query languages for XML usually specify a twig pattern query and retrieve a subset of XML elements in the document. A twig pattern can be represented as a node-labeled tree whose edges are either Parent-Child (P-C) or Ancestor-Descendant (A-D) relationships.

Extensive research efforts have been put into efficient twig pattern query processing with label-based structural joins. Following the early binary structural join algorithms [1, 12], Bruno et al. [2] proposed holistic *TwigStack* join algorithm to solve the problem of useless intermediate result in binary structural joins. It produces no useless intermediate result for twig patterns with only A-D relationships, which is defined as optimality. However, TwigStack is not optimal for twig query with P-C relationship. Several following works [3, 5, 6, 8, 10, 9] suggest different ways of optimizing TwigStack, such as indexing [6], partitioning [3] label streams, exploring Extended Dewey Label scheme [9], etc.

Most *TwigStack* optimization techniques focus on reducing intermediate results and input I/O cost. [3] further defines the optimality of twig pattern matching as minimal possible I/O cost in reading label streams and maintaining intermediate results. Though I/O is an important metric in traditional database management, it alone does not well represent the performance in twig pattern query processing, especially with stream partition approach. For example, in [3], the prefix path stream (PPS) partition scheme performs very well in terms of I/O cost. However, its response time is the worst for deep recursive data as a result of high CPU cost of pruning and merging too many streams.

In this paper, in view of the success and limitation of label stream partition in [3], we study the I/O and CPU tradeoffs for stream partition of holistic twig joins and focus on optimizing response time rather than optimizing pure I/O cost addressed previously. In particular,

1. We propose a novel stream partition technique called *recursive path stream (RPS)* partition, which can effectively achieve the I/O benefit of PPS partition [3] while solving PPS's problem of high CPU cost.
2. We also introduce a framework of adaptability of different streaming schemes and further partition of recursive path streams to flexibly fit different application requirements.
3. Our experiment results show that *RPS* is superior to other partition schemes for deep recursive data, while for non-recursive data, *RPS* is better than original *TwigStack* and as good as *PPS*.

Though our discussion in this paper focuses on label stream partition, our technique can be easily combined with other previous works, such as label indexing [6] and Extended Dewey Labeling scheme [9], to utilize their benefits.

The rest of the paper is organized as follows: we present related work in Section 2. In Section 3, we discuss the motivation and our Recursive Path Stream scheme (RPS) in detail. Experiment results are shown in Section 4. Finally, we conclude the paper and discuss possible future research in Section 5.

2 Related Work

Twig join processing is central to XML query evaluation. Extensive research efforts have been put into efficient twig pattern query processing with label-based structural joins. Zhang et al. [12] first proposed *multi-predicate merge join (MPMGJN)* based on containment ($DocId, Start, End, Level$) labeling of XML document. The later work by Al-Khalifa et al. [1] proposed an improved stack-based structural join algorithm, called *Stack-Tree-Desc/Anc*. Both of these are binary structural joins and may produce large amount of useless intermediate results. Bruno et al. [2] then proposed a holistic twig join algorithm, called *TwigStack*, to address and solve the problem of useless intermediate results. However, *TwigStack* is only optimal in terms of intermediate results for twig query with only A-D relationship. It has been proven [4] that optimal evaluation of twig patterns with arbitrarily mixed A-D and P-C relationships is not feasible.

There are many subsequent works that optimize *TwigStack* in terms of I/O, or extend *TwigStack* for different problems. In particular, a *List* structure is introduced in *TwigStackList* [8] for wider range of optimality. *TSGeneric* [6] is based on indexing each stream and skipping labels within one stream. Chen et al. [3] divides one stream (originally associated with each tag) into several sub-streams associated to each prefix path or each tag+level pair and prunes some sub-streams before evaluating the twig pattern. We call this approach as stream partition. Lu et al. [9] uses *Extended Dewey* labeling scheme and scans only the labels of leaf nodes in a twig query. Further techniques of processing twig queries with OR-predicate [5], NOT-predicate [11] and ordered twig queries [10] have also been proposed.

Our proposal is also based on label stream partition like [3]. However, we extend the solution into general optimization of both I/O and CPU cost to reduce response time. It is worth noting that our technique can be easily combined with other works discussed above to achieve their benefits.

3 Recursive Path Stream

3.1 Motivation and Terminology

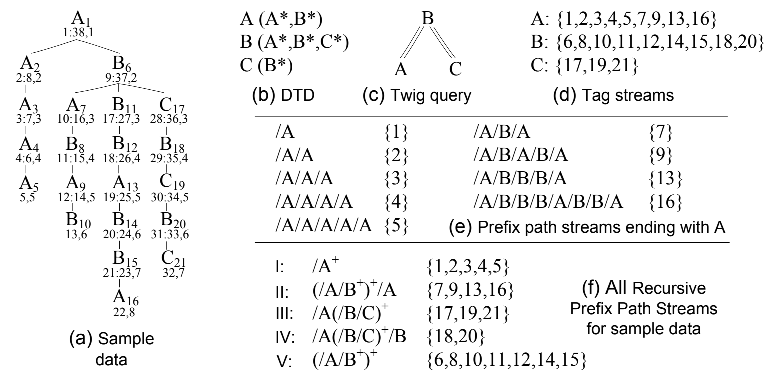
We model XML documents as labeled ordered trees. Each element, attribute and text value in the tree is associated with a label according to some labeling scheme, e.g. *containment* or *prefix* labeling schemes. One XML label uniquely identifies one element in the document. XML queries use twig patterns to match relevant portions of data in an XML document. Twig pattern edges can be parent-child (P-C) or ancestor-descendant (A-D) relationships. XML documents usually have DTD or schema information to specify their structure and to guide users writing queries.

Fig. 1(b) shows a sample DTD. Fig. 1(c) is a twig pattern query with respect to the DTD in (b). Double lines indicates A-D relationship among query nodes while single line indicating P-C relationship is not shown in the example. A sample XML tree conforming to the DTD is given in Fig. 1(a). Elements are associated with containment labels. For illustration purpose, we also show the document order of each element as subscripts n, and we use n to refer to the n^{th} element as well as its label.

To process the query of Fig. 1(c) over XML tree in Fig. 1(a), original *TwigStack* algorithm [2] scans all the labels of tags A, B and C. The set of labels of a tag is usually referred to as a *tag stream*, and the process of scanning the tag stream is called *tag streaming*. (We restrict our discussions from stream indexes, though our approach can be easily extended with stream indexes [6].) The tag streams that *TwigStack* algorithm needs to scan for this query are shown in Fig. 1(d).

Observe that elements A_1 to A_5 do not contribute to the final results of query Q in 1(c). Therefore, Chen et al. [3] propose to partition each tag stream into *prefix path streams (PPS)* and prune prefix path streams that definitely do not contribute to final results before twig join, thus saving input I/Os. There are 21 prefix paths for sample data in Fig. 1(a). Fig. 1(e) shows all the streams of paths ending with tag A. The five prefix path streams of tag A on the left column can be pruned before processing Q ([3]) as there are no B in the prefix path.

Prefix path stream scheme saves input I/Os. However, it needs to check all the paths to prune the useless ones. Moreover, holistic twig join algorithms require



/A	{1}	/A/B/A	{7}
/A/A	{2}	/A/B/A/B/A	{9}
/A/A/A	{3}	/A/B/B/B/A	{13}
/A/A/A/A	{4}	/A/B/B/B/A/B/B/A	{16}
/A/A/A/A/A	{5}		

I:	$/A^+$	{1,2,3,4,5}
II:	$(/A/B^+)^+/A$	{7,9,13,16}
III:	$/A(/B/C)^+$	{17,19,21}
IV:	$/A(/B/C)^+/B$	{18,20}
V:	$(/A/B^+)^+$	{6,8,10,11,12,14,15}

Fig. 1. Example XML document and Query

scanning labels in document order. Therefore, PPS scheme has to merge-sort all the prefix path streams for each tag during run time. The pruning and merge-sorting can be CPU expensive for deep recursive data with many prefix paths for each tag. In Fig. 1(e), we first need to prune 5 streams, then merge-sort 4 streams on the right column during holistic twig join.

We observe that prefix paths for A in Fig. 1(e) can be grouped and represented as the first two special paths in Fig. 1(f), where the '+' sign in $/A^+$ indicates there may be one or more consecutive $/A$'s in a prefix path. We term the special path as *Recursive Path.* The following introduces the terminology used in the paper.

Recursive Path (**RP**) is a special representation of a set of prefix paths that are recursively built on some tags. One or a sequence of tags in RP enclosed within '+' can be recursively expanded to represent prefix paths of different lengths. We call tags enclosed within a '+' as a **Recursive Component** (**RC**). RC's can be recursive, e.g. $(/A/B^+)^+/A$. Only P-C relationship is allowed between consecutive tags in RP. Each RP has a set of RC's. We can also view one prefix path as an RP with empty RC set, representing a singular path set of itself. If two RP's has the same tag sequence, but different RC sets, they can be combined into a **general form** such that the RC set of the general form is the union of RC sets of the two RP's. Each RP is associated with a label stream, called **Recursive Path Stream** (**RPS**). This stream contains the labels of elements of all the prefix paths represented by the RP in document order.

In Fig. 1(f), we have only five recursive paths for 21 prefix paths. For query node A in Fig. 1(c), we can prune RP I and only scan the stream of II since there is no B in I. In this way, we save both I/O and CPU cost. We call RP II and its stream as the **Potential Solution Path** (**PSP**) and **Potential Solution Stream** for the query node A.

3.2 Building RPS scheme from XML Data

We present the algorithm to extract RPS from XML data in Fig. 2. The algorithm, *BuildRPS*, uses SAX event parser and extracts recursive paths and their

```
Algorithm 1 BuildRPS
      Input: Events $e$ from SAX parser;
      Output: $RPS$; /* $RPS$ maps RP to stream */
1.    initialize Stack $ST$; /* $ST$ is the stack for start tags */
2.    initialize empty Hashtable $RPS$;
3.    while there are more events $e$
4.        if $e$ is start tag then
5.            push tag $t$ of $e$ onto $ST$;
7.            scan from the bottom to top of $ST$ to get path $p$ for the element;
8.            let $len$ = the number of tags in $p$;
9.            for ($n = 1$, $n \leq \lfloor len/2 \rfloor$, $n$++);
10.               while (there are consecutive occurrences of a same sequence of
                          tags of length $n$ in $p$)
                  /* checking from root to leaf to ensure same PP gives same RP */
11.                   change $p$ by replacing all occurrences of the same sequence
                      by one recursive component in $p$;
12.                   let $len$ = new number of tags in $p$; /* $len$ should be decreased */
13.               end while
14.           end for
15.           if (there is a path $p'$ in $RPS$ with the same tag sequence of $p$) then
16.               generate the general form $p''$ of $p'$ and $p$;
                  /* the recursive component set of $p''$ is the union of $p'$ and $p$'s RC set */
17.               associate $p''$ with the stream of $p'$ and remove $p'$ in $RPS$;
18.           else      put $p$ into $RPS$;
19.           generate and append the start and level values of current element's
                      label to corresponding recursive path;
20.       else if $e$ is end tag then
21.           pop ST;
22.           complete the label of $e$ in RPS by generating and adding the end value;
23.   end while
```

Fig. 2. Algorithm for building Recursive Path Stream (RPS) scheme

label streams with one pass of the data. This version of *BuildRPS* only handles XML elements, but can be easily extended for attributes.

BuildRPS works in three steps for each element in the XML document. Step 1 (lines 4–14) computes the element's path p and compacts it into recursive path (RP). It searches for consecutive occurrences of the same tag sequences of length n (where n ranges from 1 to half of the length of p since the length of the tag sequence can be at most the half of p in order to have two consecutive occurrences of the same tag sequences) from root to leaf of p. If there are such consecutive occurrences, lines 11 & 12 compact p by replacing the multiple same sequences by one sequence as the recursive component (RC) and set length *len* to the new length of p. Step 2 (line 15–20) combines RPs of the same tag sequence into their *general form* and appends the partial label of *start* and *level* values to the corresponding stream. This is to ensure that two different RPs produced by the algorithm represents two disjoint set of prefix paths. Step 3 (lines 21–23) completes the label of the ending element by adding the *end* value.

Example 1. Consider how *BuildRPS* algorithm extracts the RP $(/A/B^+)^+/A$ and its label stream in Fig. 1(f) for data of Fig. 1(a). When the scan reaches start tag of A_7, steps 1 first computes its path p_1, $/A/B/A$. Since p_1 is uncompactable, and this is the first path with tag sequence (A, B, A), step 2 associates p_1 with the partial label of A_7 without *end* value. Then after start tag of A_9 is reached, the algorithm gets the path p_2, $/A/B/A/B/A$, compacts it into recursive path rp_1, $(/A/B)^+/A$, with $\{(/A/B)^+\}$ as the RC set. Now, since p_1 and rp_1 have the same tag sequence and their general form is identical to rp_1 with a singular RC set, step 2 replaces p_1 by rp_1 and appends the partial label of A_9 to the label stream. Then after scanning the end tags of A_9 and A_7, step 3 completes their labels with *end* values. When the scan reaches the start tag of A_{13}, step 1 computes rp_2, $(/A/B^+)/A$, then step 2 combines it with rp_1 to get the general form $(/A/B^+)^+/A$ to replace rp_1 and appends the stream. Similar actions are taken when the start and end tags of A_{16} and end tag of A_{13} are reached.

Note that step 2 does not produce $/A(/B/A)^+$ for p_2, $/A/B/A/B/A$, since it searches from the root to leaf. The algorithm first finds the consecutive occurrences of $/A/B$ and immediately changes p_2 into rp_1 which does not contain consecutive occurrences of $/B/A$ any more.

The time complexity of *BuildRPS* is $O(D * L^3)$, where D and L are the size and maximum depth of the document. The followings are two properties of RPS scheme computed by *BuildRPS*. The proofs are omitted due to lack of space.

Property 1: Same prefix paths are always compacted to the same recursive path with shortest possible tag sequence.

Property 2: Two different recursive paths represent two disjoint prefix path sets as well as disjoint label streams.

3.3 Identifying Potential Solution Paths

We now discuss the process of identifying potential solution (and pruning useless) paths for a twig pattern query. The algorithm is based on the following two properties of a recursive path.

Property 3: For any two tags T_1 and T_2 in a recursive path P, T_1 is an **ancestor tag** of T_2 *if* T_1 appears before T_2 in P or there exists some recursive component in P containing both T_1 and T_2

Property 4: For any two tags T_1 and T_2 in a recursive path P, T_1 is a **parent tag** of T_2 *if* T_1 appears before T_2 consecutively in P or there exists some recursive component RC in P such that T_2 is the first tag and T_1 is the last tag of RC.

Example 2. Consider the recursive path $/A(/B/C/D)^+$. A, C and D are all ancestor tags of B since 1) A appears before B and 2) there is one recursive component containing all B, C and D. C and D will appear before B if we expand $(/B/C/D)^+$ once to get $/A/B/C/D/B/C/D$. However, only A and D are the parent tags of B as they appear before B consecutively after the expansion.

Algorithm 2 IdentifyPSP

```
    Input: Twig query $Q$ and RPS partition scheme $P$
    Output: Potential Solution Path sets $Pset$s for all query node $N$ in $Q$
1.  initialize $Pset$ of each query node as empty set.
2.  depth first search query twig $Q$, upon returning from current query node $N$;
3.  let $Cset$ of $N$ be an empty set /* $Cset$ is "Candidate PSP set" */
4.  get query path $qp$ from query root to $N$
5.      if $N$ is leaf query node then
6.          let $Cset$ be all recursive paths ending with tag $N$ in $P$;
7.      else if $N$ is non-branching internal query node then
8.          let $Cset$ = getCset($N$, $PSet$ of child of $N$);
9.      else if $N$ is branching query node then
10.         for $Pset$ of each child $Ci$ of N's children
11.             let $Cset_i$ = getCset($N$, $Pset$);
12.         end for
13.         let $Cset$ be the intersection of all $Cset_i$'s;
14.     for each $rp$ in $Cset$
15.         if checkPSP ($rp$, $qp$) == true then put $rp$ in $Pset$ of $N$;
16.     end for
17. end depth first search
18. for each query node $N$
19.     for each $rp$ in $Pset$ of $N$
20.     if $\neg\exists$ $rp'$ in $Pset$ of root s.t. tag sequence of $rp'$ is a prefix of $rp$ then
21.         remove $rp$ from $Pset$ of $N$;
22.     end for
23. end for

Function getCset($N$, $childPset$) /* get $Cset$ of $N$ based on Pset of $N$'s child */
1.  let $Cset$ be empty set;
2.  for each $rp$ in $childPset$
3.      put each RP whose tag sequence is a prefix of $rp$ and ends with $N$ into $Cset$;
4.      for each recursive component $rc$ containing but not ending with $N$ in $rp$
5.          get tag sequence $ts$ by repeating tags up to $N$ in $rc$ once;
6.          put into $Cset$ the RP of tag sequence from the root to repeated $N$ in $ts$;
7.      end for
8.  end for
9.  return $Cset$;

Function checkPSP ($rp$, $qp$) /* check if $rp$ is potential solution path of $qp$ */
1.  let tag set $s_1$ be $N$ where $N$ is the leaf node of $rp$;
    /*elements in tag set are of the same name, differentiated by the positions in $rp$*/
2.  for each $qp$ tag $T$ from leaf to root /* $T$'s parent is dummy if $T$ is root */
3.      let $PT$ be parent tag of $T$ in $qp$ and $E$ be the edge between $PT$ and $T$;
4.      if $T$ is the root then return BOOLEAN($E$ is A-D OR $s_1$ contains root of $rp$);
5.      let tag set $s_2$ be $\{e_2 \mid e_2$ and $pt$ have identical tag $\wedge$
                $\exists e_1 \in s_1$ s.t. $e_2$ is the parent (or ancestor based on $E$) tag of $e_1$ in $rp\}$;
6.      if $s_2$ is empty then return false;
7.      else        let $s_1$ be $s_2$;
8.  end for
```

Fig. 3. Algorithm for Identifying Potential Solution Paths in RPS scheme

We show algorithm *IdentifyPSP* in Fig. 3. It identifies the Potential Solution Path (PSP) set ($Pset$) for each query node in a given twig query, with two phases: bottom-up pruning from query leaves and top-down pruning from query root. The bottom-up phase first propagates branching information for pruning from branches to the branching nodes; whereas top-down phase then propagates the combined branching information to each individual branch.

In the bottom-up pruning phase (lines 2-17 of Main), it visits each query node N in depth first order. Upon returning from N, it first computes N's Candidate Potential Solution Path set ($Cset$) (lines 3-13), then checks each recursive path rp in $Cset$ if it is a PSP to be put into $Pset$ (lines 14-16). Note that for branching node, the $Cset$ is the intersection of the $Cset$s computed based on the $Pset$s of each child query node. We will shortly discuss how to compute $Cset$ of a query node based on its child's $Pset$. In the top-down pruning phase (lines 18-23), for $Pset$ of each non-root query node, it removes all the recursive paths (RP) for which there exists no RP as its prefix in the $Pset$ of the query root.

There are two auxiliary functions for the algorithm: *getCset* and *checkPSP*. Function *getCset* finds the $Cset$ of query node N based on the $Pset$ of N's child. It puts into $Cset$ all recursive path rp such that rp ends with N and is a prefix of either 1) any rp' in $Pset$ of N's child or 2) tag sequence expanded from rp' by repeating any single recursive component once. Function *checkPSP* checks if the given recursive path rp is the PSP for query path qp. It recursively scans query tag T from the leaf to the root of qp. For each T and T's parent query node PT, it computes the T's ancestor (or parent depending on the query edge between T and PT) tags that are same to tag name PT. When none can be found or T is the root query node, *checkPSP* returns.

The time complexity of *IdentifyPSP* is $O(|Q| * |rp| * (F_Q * |rc| + D_q * D_{rp}))$, where $|Q|$, $|rp|$, F_Q, $|rc|$, D_q and D_{rp} are number of query nodes, number of RPs (for each tag if we have a mapping from tags to their RPs), maximum query fan-out, maximum number of RCs in one RP, query depth and maximum depth of RP respectively. Since most of the above values are usually small, saving in IO is usually worth the efforts in pruning. The following theorem shows the correctness of *IdentifyPSP*. However, due to lack of space, we cannot provide the proofs for the time complexity and the theorem.

Theorem 1. *Given query Q and RPS scheme, labels of streams pruned by IdentifyPSP algorithm do not contribute to final answer of Q.*

Example 3. Let us trace algorithm *IdentifyPSP* on the query and RPS scheme in Fig. 1(c) and (f). In bottom-up phase, depth first search first returns from query node A. Within the two candidate RPs {I, II} of A, only RP II in Fig. 1(f) is identified as PSP since query root B appears as an ancestor tag of A in II, but not in I. So, the current PSP set of A is {II}. Similarly, {III} is the PSP set for C. Now, according to function *getCset*, the candidate PSP sets for B are {V} based on A's PSP set and {IV, V} based on C's PSP set (IV is in the candidate PSP set of B since it is a prefix of the expansion of RC in III). So the intersection is {V}, which is then identified as the PSP set of B. The top-down pruning does not take effects in this case. Thus the final PSP set of A, B and C are {II}, {V} and {III} respectively. However, suppose we modified the sample data to have one more RP as $/A/D^{+}/B/A$, it would be PSP for A after bottom-up phase but pruned after top-down phase since V is not its prefix.

3.4 Adaptability of Different Stream Partition Schemes

As mentioned in Sec. 3.1, uncompactable prefix path is a special case of recursive path. The RPS scheme is a generalization of PPS. Applying RPS to non-recursive data generates the same stream partition as PPS. Therefore, it is safe to replace PPS with RPS in non-recursive data. Besides, we can further partition streams in RPS according to different application requirements. For example, when there are many A's at depths more than three, but most queries are only interested in A's at depth less than or equal to two, we can partition the stream associated with $/A^{+}$ into streams $/A^{1:2}$ and $/A^{3+}$, meaning RC $/A$ can be repeated at most twice and at least three times respectively.

However, for irregular data, even RPS may generate too many streams and result in long query response time. For example, if we change the DTD in Fig. 1(a) as "every element of (A, B, C) can have any element of the three as their children", we may have the following deep uncompactable data path

$$/A/B/C/B/A/B/C/A/C/B/A/B/C/\ldots.$$

In such case, it is better to use non-partitioned Tag Streaming to avoid merge-sorting overwhelming number of streams during holistic twig joins.

4 Experimental Evaluation

We experimentally compare the performance of RPS with non-partitioned tag stream scheme and existing partition schemes: PPS and Tag+level. Results show that, RPS and PPS are comparable and better than Tag or Tag+level in non-recursive or light recursive data (e.g. XMark). In deep recursive data (e.g. TreeBank), RPS significantly out-performs others for total query response time.

4.1 Experimental Settings

Implementation and Hardware We implemented all algorithms in Java. Different stream partition schemes were compared based on TwigStack holistic join [2]. The experiments were performed on a normal PC with 2.6GHz Pentium 4 processor and 1GB RAM running Windows XP.

XML Data Sets We use two well-known data sets (XMark and TreeBank) for our experiments. The characteristics and the number of streams for each partition technique of these two data sets are shown in Table 1. We choose these two data sets because XMark is light recursive with non-recursive tags, while TreeBank is deep recursive. In this way, we can study the performance of various stream partition methods with different levels of recursion in XML data.

Table 1. XML Data Sets

	XMark	Treebank
Size	113MB	82MB
Nodes	2.0 million	2.4 million
Max Depth	12	36
Ave Depth	5	8
Tags	75	251
Tag+Level#	119	2237
PPSs #	514	338724
RPSs #	415	119748

Table 2. Tested Queries

XM1	//site/people/person/name
XM2	//site//people/person[/name]//age
XM3	//text[//bold]//emph//keyword
XM4	//text[/emph/keyword]/bold
XM5	//listitem[//bold]/text[//emph]//keyword
TB1	//S[//ADJ]//MD
TB2	//VP[/DT]//PRP_DOLLAR_
TB3	//PP[/NP/VBN]/IN
TB4	/S/VP//PP[//NP/VBN]/IN
TB5	//S//NP[//PP/TO][//VP/_NONE_]/JJ

Queries We select a wide range of representative queries (shown in Table 2) for each data set (XM for XMark and TB for TreeBank). In particular, XM1 and XM2 contain non-recursive tags, while the rest all contain recursive tags. XM1 is a path query. XM2–4, TB1–4 are simple twig queries with only one branching node of fan-out two. Except incoming root query edge, XM3 and TB1 have only A-D edges; XM4 and TB3 have only P-C edges; while XM2, TB2 and TB4 have a mixture of A-D and P-C edges. For complex twig queries, XM5 has two branching nodes whereas TB5 has one branching node of fan-out three. The number of various label streams for all the tags of each query before and after pruning is shown in Table 3. We can see the number of RPSs is much smaller (up to 67% less) than PPSs.

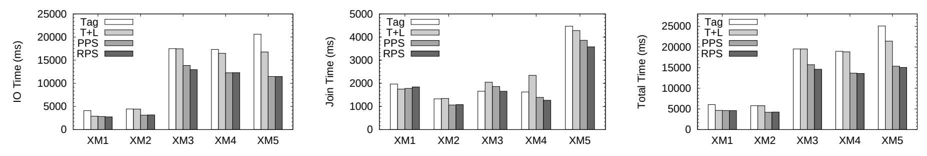
Table 3. Number of Streams before and After Pruning for various Partition Schemes

	Tag + Level		PPS		RPS	
	before	after	before	after	before	after
XM1	7	4	11	4	11	4
XM2	8	6	12	5	12	5
XM3	27	25	330	198	240	144
XM4	27	25	330	132	240	96
XM5	31	23	348	198	249	99
TB1	62	46	12561	1623	5126	743
TB2	87	86	38527	2455	12067	814
TB3	118	100	97285	1164	29563	624
TB4	177	138	123669	1874	38693	798
TB5	209	182	132503	2805	42915	1341

Performance Measures We compare RPS with non-partitioned Tag streams and existing PPS and tag+level partition schemes. The presented performance measures include pruning time, IO time of reading labels, CPU time of structural join (including merge-sorting streams) and total response time of each query. The IO time and CPU time of joins are estimated by reading all labels into memory (IO time) before in-memory structural join (CPU time). Although the number of labels (or bytes) scanned for each query is also an important measure for the effectiveness of partition schemes, it is not shown due to space limitations as their experiment results are similar to IO time.

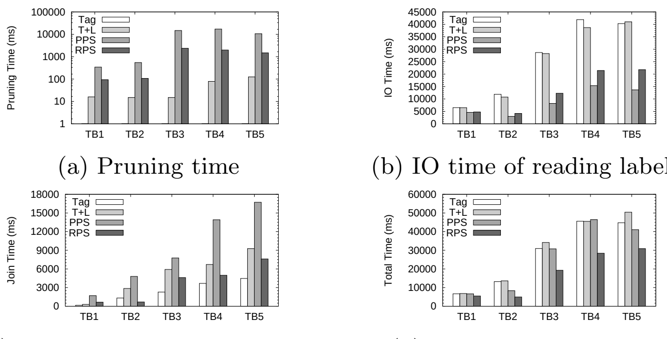
4.2 Experiment Results and Analysis

We show the experiments results for XMark data in Fig. 4. We did not show the pruning time for XMark as it is only a few milliseconds, for all queries, thus is a negligible component of total response time. It is clear that holistic twig join with RPS partition is faster than Tag+Level (T+L) partition and non-partitioned Tag in both input reading and structural join as a result of less labels scanned and processed. We can also observe that RPS is comparable to PPS for XM1 and XM2 containing non-recursive query tags and slightly better than PPS for XM3–5 containing recursive query tags in terms of structural join and total response time. Theoretically, the number of labels scanned in PPS is less than or equal to RPS. So, it is interesting to see RPS is better than PPS in input reading for XM3 as shown in Fig. 4(a). This is the result of the larger overhead of PPS to read the same number of labels in more streams compared to RPS.



(a) IO time of reading labels (b) CPU time of structural join (c) Total processing time

Fig. 4. Experimental Results for XMark dataset (metrics of different scales)



(a) Pruning time (b) IO time of reading labels

(c) CPU time of structural join (d) Total processing time

Fig. 5. Experimental Results for TreeBank dataset (metrics of different scales)

The results for TreeBank data set are shown in Fig. 5. We can see from Fig. 5(a), RPS is much faster than PPS, but slower than Tag+Level in pruning phase as expected. In reading inputs (Fig. 5(b)), PPS is the best since it reads the least amount of labels by pruning more label streams; RPS is a bit slower than PPS, but much faster than Tag and Tag+level. For CPU time of structural join (Fig.

5(c)), non-partitioned Tag scheme is the best. Although PPS processes the least amount of labels, it is still the worst in structural join time due to high cost of merge-sorting too many streams. RPS is better than Tag+level in structural join time in general because RPS processes much less labels, which outweighs the overhead of merge-sorting more streams. For RPS alone, although it is not the best in any of the pruning, input reading or structural join, the beneficial trade-off between IO and CPU helps RPS to be the best in overall query response time (up to 2 times faster than the most competitive ones) as shown in Fig. 5(d).

5 Conclusion and Future Work

In this paper, we propose a novel stream partition scheme for efficient holistic twig joins, namely recursive path stream. RPS scheme is a generalization of prefix path stream proposed in [3]. Experiment results show that RPS is more efficient than other stream partition techniques in recursive XML data while it is as good as PPS and better than others in non-recursive data. As a part of future work, we would like to study the cost model for holistic twig joins with stream partition and indexing.

References

1. 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 *Proc. of ICDE Conference*, pages 141–152, 2002.
2. N. Bruno, D. Srivastava, and N. Koudas. Holistic twig joins: optimal XML pattern matching. In *Proc. of SIGMOD Conference*, pages 310–321, 2002.
3. T. Chen, J. Lu, and T. W. Ling. On boosting holism in XML twig pattern matching using structural indexing techniques. In *Proc. of SIGMOD Conference*, 2005.
4. B. Choi, M. Mahoui, and D. Wood. On the optimality of the holistic twig join algorithms. In *Proc. of DEXA*, pages 28–37, 2003.
5. H. Jiang, H. Lu, and W. Wang. Efficient processing of XML twig queries with or-predicates. In *Proc. of SIGMOD Conference*, 2004.
6. H. Jiang, W. Wang, H. Lu, and J. Yu. Holistic twig joins on indexed XML documents. In *Proc. of VLDB Conference*, pages 273–284, 2003.
7. C. Li, T. W. Ling, and M. Hu. Efficient processing of updates in dynamic XML data. In *Proc. of ICDE*, 2006.
8. J. Lu, T. Chen, and T. W. Ling. Efficient processing of XML twig patterns with parent child edges: a look-ahead approach. In *Proc. of CIKM*, pages 533–542, 2004.
9. J. Lu, T. W. Ling, C. Chan, and T. Chen. From region encoding to extended dewey: On efficient processing of XML twig pattern matching. In *Proc. of VLDB Conference*, pages 193–204, 2005.
10. J. Lu, T. W. Ling, T. Yu, C. Li, and W. Ni. Efficient processing of ordered XML twig pattern. In *Proc. of DEXA*, 2005.
11. T. Yu, T. W. Ling, and J. Lu. Twigstacklistnot: A holistic twig join algorithm for twig query with not-predicates on XML data. In *Proc. of DASFAA*, 2006.
12. C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In *Proc. of SIGMOD Conference*, pages 425–436, 2001.