Prefix Path Streaming: a New Clustering Method for XML Twig Pattern Matching

Ting Chen, Tok Wang Ling, Chee-Yong Chan

School of Computing, National University of Singapore
Lower Kent Ridge Road, Singapore 119260
{chent,lingtw,chancy}@comp.nus.edu.sg

Abstract. Searching for all occurrences of a twig pattern in a XML document is an important operation in XML query processing. Recently a class of holistic twig pattern matching algorithms has been proposed. Compared with the prior approaches, the holistic method avoids generating large intermediate results which do not contribute to the final answer. The method is CPU and I/O optimal when twig patterns only have ancestor-descendant relationships.The holistic twig-pattern matching method proposed earlier [1] operates on element streams which cluster all XML elements with the same tag name together. In this paper we introduce a clustering method called *Prefix Path Streaming* (PPS) and new holistic twig pattern matching algorithms based on *PPS*. *PPS* clusters elements of XML documents according to the paths from root to the elements. This clustering approach avoids unnecessary scanning of irrelevant portion of XML documents.More importantly, we develop optimal algorithms based on *PPS* streaming which can process a large class of twig patterns consisting of both ancestor-descendant and parent-child relationships.

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 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[7] 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 parent-child or ancestor-descendant relationships. After that, each binary relationship is processed using *structural join* techniques[8][5] and the final match results are obtained by "stitching" individual binary join results together. This approach may generate large and possibly unnecessary intermediate results. Bruno et al.[1] propose a novel *holistic* method of XML path and twig pattern matches which avoids storing intermediate results unless they contribute to the final results. The method is CPU and I/O optimal for all path(with no branching) patterns and twig patterns whose edges are entirely ancestor-descendant edges. However the approach is found to be suboptimal if there are parent-child relationships in twig patterns.

The original holistic method groups all elements in XML document with the same tag name together. We call this clustering method *Tag Streaming*. Choi et al.[3] show that it is not possible to develop an optimal holistic twig join algorithm for twig patterns having both ancestor-descendant and parent-child relationships using Tag Streaming. In this paper, we propose a new XML document clustering scheme: *Prefix-Path Streaming* (PPS), which groups XML document elements with the same root-to-element tag sequence into the same stream (we call such a stream *Prefix-Path Stream*). Prefix-Path Streaming is simple and it can avoid unnecessary scanning of portions of XML documents which do not contribute to final matching results. More importantly,we demonstrate that for twig patterns with only P-C edges or only A-D edges and twig patterns with only one node having fan-out factor larger than 1(or *1-branch* twig pattern), PPS streaming can provide I/O and CPU optimal solution. To the best of our knowledge, this is the first method which can process all the above classes of twig pattern optimally.

The rest of this paper is organized as follows: Section 2 is the preliminary which introduces XML data model, twig pattern query and notations used in this paper. In section 3, we review the current holistic method for twig pattern matching and discuss its problems. In section 4 we introduce our new clustering method: Prefix-Path Streaming (PPS). Section 5 explains how to prune unnecessary streams given a twig pattern. Section 6 shows our holistic pattern matching algorithms based on PPS: *PPS-TwigStack*. Section 7 shows the performance of our new methods. Section 8 concludes the paper.All the proofs of results are omitted in this paper and can be found in a full version on line[2].

2 Preliminary

A XML document is commonly modelled as a rooted, ordered and labelled tree. In this paper wherever the word “element” appears, it refers to either element or attribute in a XML document. Many XML query processing algorithms rely on certain numbering schemes. [4] uses *(startPos: endPos, LevelNum)* to label elements in a XML file.*startPos* and *endPos* are calculated by performing a preorder(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.

A *twig pattern match* in a XML database can be represented as an n-ary tuple $< d_1, d_2, \ldots, d_n >$ consisting of the database nodes that identify a distinct match of Q with n nodes in D. 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 repeated tag in twig pattern Q.

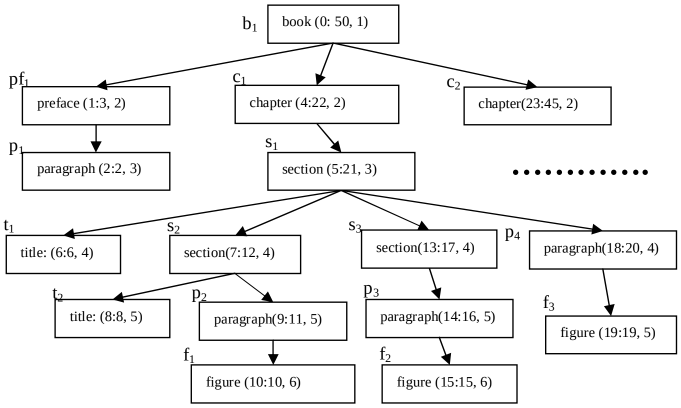

Fig. 1. A sample XML document with node labels in parentheses. Each element is also given an identifier (e.g. s_1) for easy reference.

3 Related Work

The holistic method *TwigStack*,proposed by Bruno et al[1], 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 an element stream T_q containing all elements of tag q. Elements in each stream are sorted by their *startPos*. Each stream has a cursor which can either move to the next element or read the element under it. The algorithm operates in two 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. Thus the algorithm ensures 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.* However, with the presence of P-C edges in twig patterns, the TwigStack method is no longer optimal.

Example 1. Let us evaluate the twig pattern $section[title]/paragraph//figure$ on the XML document in Fig.1. Because of the P-C edge *section/paragraph*, in order to know that element s_1 is in a match, we have to advance $T_{paragraph}$ to p_4 and T_{figure} to f_3. By doing so, we skip nodes p_2, p_3, f_1 and f_2 and miss the matches involving p_2 and f_1. However, since s_2 and s_3 have not been seen, we have no way to tell if any of the above four elements might appear in the final match results. The only choice left is to store all the four elements in which only two are useful (unnecessary storage).

4 Prefix-Path Streaming(PPS)

Tag Streaming does not consider the context where an element appears. In this section, we introduce a new streaming approach.

The *prefix-path* of an element in a XML document tree is the path from the document root to the element. A *prefix-path stream(PP stream)* contains all elements in the document tree with the same prefix-path, ordered by their *startPos* numbers. Each PP stream T has a *label* or *label(T)* which is the common prefix-path of its elements. A PP stream is said to of *class N* if its label ends with tag N. Two PP streams are of the same class if their labels end with the same tag. Referring again to the example in Fig.1 there will be 11 streams using PPS streaming compared with 7 streams generated by Tag Streaming. For example, instead of a single stream $T_{section}$ in Tag Streaming, now we have two PP streams: $T_{book-chapter-section}$ and $T_{book-chapter-section-section}$.

4.1 Differences between Tag and Prefix-Path Streaming

PPS streaming,unlike Tag Streaming, allows parallel access to PP streams containing elements with the same tag. The benefits can be immediately seen on the twig pattern query in Example 1: under PPS streaming,p_4 is placed in a different PP stream as p_2 and p_3; now we can find s_1 is the parent of p_4 without skipping any elements. In this subsection we explore further the properties of PPS streaming and its advantages over Tag Streaming in the problem of twig pattern matching.

Definition 1. *(Match Order) Given a twig pattern Q and a certain streaming method, a match M_A to Q_A is said to be* ahead *of a match M_B to Q_B or $M_A \leq M_B$ if for each node $q \in (Q_A \cap Q_B)$, $e_q^A.startPos \leq e_q^B.startPos$ or e_q^A and e_q^B are placed on different streams, where e_q^A and e_q^B are of tag q and belong to match M_A and M_B respectively. A and B are nodes in Q.*

As an example,for twig pattern $Q1 : //section[title]/paragraph//figure$, the two matches $M_1 =< s_1, t_1, p_4, f_3 >$ and $M_2 =< s_2, t_2, p_2, f_1 >$ in Fig.1 are incomparable under Tag Streaming because $t_1.startPos < t2.startPos$ but $p_4.startPos > p_2.startPos$. However under PPS streaming, we have both $M_1 \leq M_2$ and $M_2 \leq M_1$ simply because the eight elements in M_1 and M_2 are placed on eight different streams. Note that under Match Order, $M_1 \leq M_2$ and $M_2 \leq M_1$ doesn't mean $M_1 = M_2$.

Definition 2. *Given a twig pattern Q and two elements a with tag A and b with tag B, $a \leq_Q b$ if a is in a match M_A to Q_A such that $M_A \leq M_B$ for any match M_B to Q_B containing element b.*

Intuitively, $a \leq_Q b$ implies that it is possible to determine if a is in a match to Q_A without skipping any matches of b to Q_B. For example, given the twig pattern $Q : //section[//title]//paragraph//figure$, $s_1 \leq_Q s_2$ in Fig.1 under Tag Streaming because s_1 has a match (s_1, t_1, p_2, f_1) which is ahead of any match s_2 for Q. However,for $Q1 : //section[title]/paragraph//figure$, the only match of s_1 is $< s_1, t_1, p_4, f_3 >$ and the only match of s_2 is $< s_2, t_2, p_2, f_1 >$ since the two matches are not "comparable", neither $s_1 \leq_{Q1} s_2$ nor $s_2 \leq_{Q1} s_1$ under Tag Streaming. On the other hand, we have $s_1 \leq_{Q1} s_2$ as well as $s_2 \leq_{Q1} s_1$ under PPS streaming.

Lemma 1. *For two elements a of tag A and b of tag B in a A-D only twig pattern Q, suppose a has matches to Q_A and b has matches to Q_B, either $a \leq_Q b$ or $b \leq_Q a$ under Tag Streaming.*

Analogous to Lemma 1, PPS-streaming has similar properties for a larger classes of twig patterns.

Lemma 2. *For two elements a of tag A and b of tag B in a A-D only or a P-C only or a 1-branch twig pattern Q, if a has matches to Q_A and b has matches to Q_B, either $a \leq_Q b$ or $b \leq_Q a$ under PPS streaming.*

The significance of Lemma 1 and 2 can be understood as follows. Suppose the contrary, if there is a pair of elements a and b such that neither $a \leq_Q b$ nor $b \leq_Q a$ for twig pattern Q, then to determine if a is in a match to Q *may* skip elements in b's match and vice versa. In the above situation, the only way to ensure result correctness is to store potentially useless elements which sacrifices optimality. Therefore the two lemmas pave the way for designing optimal twig pattern matching algorithms for their respective class of twig patterns.

5 Pruning PP streams

Using labels of PP streams, we can statically prune away PP streams which contain no match to a twig pattern. For example, the stream $T_{book-preface-para}$ of the XML document in Fig.1 is certainly not useful for the twig pattern

$//section[title]/paragraph//figure$. Although all the elements in the PP stream have element name *paragraph*, they do not have *section* parents!

A PP stream T_1 is an *ancestor PP stream* of PP stream T_2 if label(T_1) is a prefix of label(T_2). T_1 is the *parent PP 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 following recursive formula helps us determine the useful streams for evaluating a twig pattern Q. For a PP stream of class N with label l, we define U_l to be the set of all descendant PP streams of T_l(including T_l) which are useful for the sub-twig of Q_N except that we only use stream T_l (not its descendant streams) to match node N.

$$U_l = \begin{cases} \{T_l\} & \text{if N is a leaf node;} \\ \{T_l\} \bigcup \{\bigcup_{N_i \in child(N)} C_i\} & \text{if none of } C_i \text{ is null;} \\ null & \text{if one of } C_i \text{ is null.} \end{cases}$$

where $C_i = \bigcup U_{l_i}$ for each child stream label l_i of l if the edge $< N, N_i >$ is a P-C edge ;or $C_i = \bigcup U_{l_i}$ for each descendant stream label l_i of l if the edge $< N, N_i >$ is a A-D edge. Function $child(N)$ returns the children nodes of N in the twig pattern Q.

The base case is simple because if N is a leaf node, any PP stream of class N must contain matches to the trivial single-node pattern Q_N. As for the recursive relationship, note that for a PP stream with label l of class N to be useful to the sub-twig pattern rooted at N, for each and every child node N_i of N, there should exist some non-empty set U_{l_i} which are useful to the sub-twig Q_{N_i} AND the structural relationship of l and l_i satisfies the edge between N and N_i. In the end the set $\bigcup U_{l_r}$ contains all the useful PP streams to a query pattern Q, where l_r is the labels of PP streams of class $root(Q)$.

6 Holistic Twig Join based on PPS

The algorithm *PPS-TwigStack* reports all the matches in a XML document to a twig pattern. *PPS-TwigStack* is also a stack-based algorithm. It associates a *stack* for each query node in the twig pattern. A stack S_{q_1} is the parent stack of S_{q_2} if q_1 is the parent node q_2 in Q. Each item in the stack S is a 2-tuple consisting of an element e and a pointer to the *nearest* ancestor of e in the parent stack of S. The parent pointer is used to reconstruct result paths. This idea is first used in [1]. A query node N is also associated to *all* the useful PP streams of class N. A PP stream has the same set of operations as a stream in Tag Streaming. Fig.2(a) shows the main data structures used.

The algorithm PPS-TwigStack, similar to TwigStack, in each step finds a "potential" matching element a. Depending on the current stack states, a is then being pushed into its stack if it's really in some match and/or being used only for updating stack matching states.

With Lemma 2,in each step, we repeatedly find element a of tag A with the following properties in the remaining portions of streams:

1. a is in a match to Q_A and $a \leq_Q e$ for each remaining element e.

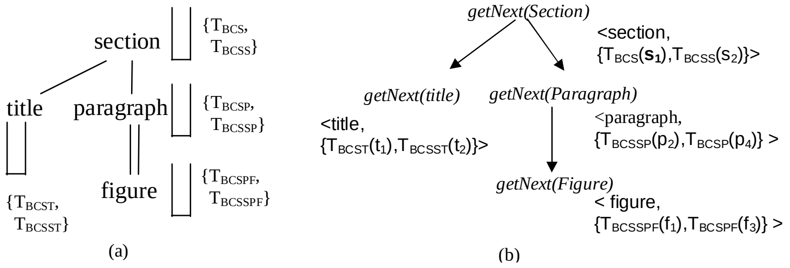
section
{T_{BCS}, T_{BCSS}}
title
paragraph
{T_{BCSP}, T_{BCSSP}}
figure
{T_{BCSPF}, T_{BCSSPF}}
{T_{BCST}, T_{BCSST}}
(a)
getNext(Section)
<section, {$T_{BCS}(s_1)$,$T_{BCSS}(s_2)$}>
getNext(title)
getNext(Paragraph)
<title, {$T_{BCST}(t_1)$,$T_{BCSST}(t_2)$}>
<paragraph, {$T_{BCSSP}(p_2)$,$T_{BCSP}(p_4)$} >
getNext(Figure)
< figure, {$T_{BCSSPF}(f_1)$,$T_{BCSPF}(f_3)$} >
(b)

Fig. 2. Illustration of Algorithm $PPS - TwigStack$ (a) Main data structures used.We use a shorter form to denote PP streams.E.g. $T_{book-chapter-section}$ is represented as T_{BCS} (b) The call stack of the first getNext() call with return values. The elements in parenthesis indicates the cursor position of each stream right after the call.

2. $a.startPos < d.startPos$ for each d of tag D which is in a match to $Q_D(D \in Q_A)$.
3. For each sibling node SI_i of A,there is an element s_i of tag SI_i which satisfies property (1) and (2) w.r.t. SI_i.
4. $a.startPos < s_i.startPos$ for each sibling node SI_i.
5. There is no element a' in streams of class $parent(A)$ such that a' is parent(ancestor) of a if the edge $< A, parent(A) >$ is a P-C(A-D) edge.

Property (1) implies the element a is in a match to Q_A and thus *may* be in a match to Q. Meanwhile in searching for a, no potential matching element is skipped because $a \leq_Q e$ for every e. Property (5) guarantees that ancestor element of a match to Q or sub-twig of Q is always found before any descendant element in the same match. Properties (2)-(4) ensure that whenever an element is popped out of its stacks, all matches involving it have been found. Combining (2)-(5),we can see that if a is in a match to Q, its parent or ancestor element in the match MUST be still in the stack $S_{parent(A)}$; so if no such element exists there we can just discard a.

At the heart of our algorithm, the method getNext() implements the basic searching strategy. Lines 2-3 in the *main* routine find the element satisfying the five conditions. Lines 4-12 maintain the states of matching in stacks(which are identical to those of $TwigStack$). The getNext() method returns two things:

1. a node q_x in the sub-twig Q_q;and
2. a set of PP streams $\{T^i_{q_x}\}$ of class q_x whose head elements are in matches to Q_{q_x}. Moreover, among head elements of streams in $\{T^i_{q_x}\}$, the one with the smallest $startPos$ satisfies all the five properties w.r.t. sub-twig Q_q.

getNext() works recursively. For a node q, it makes recursive calls for all child nodes of q(lines 3-4 of textsf(getNext()).

– If getNext(q_i) $= q_i$ for all child nodes q_i, we are able to find an element e which satisfies the first four properties(line 17). Then in lines 7-16 we

Algorithm 1 *PPS-TwigStack*

```
while ¬end($q$) do
    $< q_{min}, \{T_{q_{min}}\} > =$ getNext($q$);
    $T_{min}$ = the PP-stream in set $\{T_{q_{min}}\}$ whose head element has the smallest startPos;
    if ¬isRoot($q_{min}$) then
        cleanStack(parent($q_{min}$),nextL($q_{min}$))
    if isRoot($q_{min}$) ∨ ¬empty($S_{parent(q_{min})}$) then
        cleanStack($q_{min}$,next($q_{min}$))
        moveStreamToStack($T_{min}$,$S_{q_{min}}$,pointer to top of $S_{parent(q_{min})}$)
        if isLeaf($q_{min}$) then
            showSolutionWithBlocking($S_{q_{min}}$,$l$)
            pop($S_{q_{min}}$)
    advance($T_{min}$)
mergeAllPathSolutions()
```

try to find if there is an element of tag q which is in match of Q_q and parent(ancestor) of e. If such an element exists,e does not satisfy property (5) and we return $< q, T_q >$ in lines 18-19 and keeps on searching. Otherwise, e is the element satisfying all five properties and can be returned(line 20).

– Otherwise getNext(q_i) $\neq q_i$ for some child nodes q_i, which suggests the element satisfying the five properties have been found. Therefore we just return what getNext(q_i) returns.

Example 2. Consider the twig pattern query $section[title]/paragraph//figure$ on the XML document in Fig.1. The call stack of the first getNext(section) is shown in Fig.2(b). We find s_1 in a match because we recursively find the first two *paragraph* elements and the first two *title* elements from their respective PP streams which are in matches to $paragraph//figure$ and $/title$. After considering the four combinations of the *title* and *paragraph* elements (2x2), we find two match elements s_1 and s_2 push s_1 into its stack $S_{section}$. The subsequent calls find t_1,s_2,t_2,p_2,f_1,p_3,f_2,p_4 and f_3 in order.

PPS-TwigStack performs a single forward scan of all useful PP streams. It never stores any paths which are only partial matches to one branch of the twig pattern but do not appear in the final match results. Therefore, the worst case I/O complexity is linear in the sum of sizes of the useful PP streams and the output list. The worst case CPU complexity is O($(|max\_PPSStreams\_per\_tag|)^{|Q|} \times (|Input\_list| + |Output\_list|)$) where $|Q|$ is the twig pattern size, which is also independent of the size of partial matches to any root-to-leaf path of the twig. The factor of $|max\_PPSStream\_per\_tag|^{|Q|}$ is because getNext() needs to enumerate PP stream combinations. The space complexity is the maximum length of a root-to-leaf path. If a twig pattern is not in any of the three classes aforementioned, we can simply run *TwigStack* by simulating a single stream T_q in Tag Streaming using multiple PP streams of class q.

Algorithm 2 *getNext(q)*

```
if isLeaf($q$) then
    return $< q, \{T_q\} >$ /* $T_q$ contains PP streams of class q not yet end*/
for $q_i$ in children($q$) do
    $< n_i, \{T_{c_i}\} >$ = getNext($q_i$)
    if $n_i \neq q_i$ then
        return $< n_i, \{T_{c_i}\} >$
for $< T_{c_1}^{\alpha_1}, T_{c_2}^{\alpha_2}, \ldots, T_{c_n}^{\alpha_n} > \in \{T_{c_1}\} \times \{T_{c_2}\} \times \ldots \times \{T_{c_n}\}$ do
    for each $T_q^j$ of class $q$ satisfying the P-C or A-D relationship with each $T_{c_i}^{\alpha_i}$ do
        $T_{min}$ = the stream in $< T_{c_1}^{\alpha_1}, T_{c_2}^{\alpha_2}, \ldots, T_{c_n}^{\alpha_n} >$ whose head has the smallest $startPos$
        $T_{max}$ = the stream in $< T_{c_1}^{\alpha_1}, T_{c_2}^{\alpha_2}, \ldots, T_{c_n}^{\alpha_n} >$ whose head has the largest $startPos$
        if $\neg$mark[$T_q^j$] then
            while head($T_q^j$).$endPos$ < head($T_{max}$).$startPos$ do
                advance($T_q^j$);
        if head($T_q^j$).$startPos$ < head($T_{min}$).$startPos$ then
            mark[$T_q^j$] = $true$;
            add $T_q^j$ to set $\{T_q\}$
$T_1$ = min($T_q$); $T_2$= min($\cup\{T_{c_i}\}$); /* min is a function which returns the PP-stream in the input set whose head has the smallest startPos */
if head($T_1$).$startPos$ < head($T_2$).$startPos$ then
    return $< q, \{T_q\} >$
return $< c_i, \{T_{c_i}\} >$ /*$T_2$ is of class $c_i$ */
```

7 Experiments

All our experiments were run on a 2.4GHz Pentium 4 processor with 512MB memory. We first measure the effect of PPS pruning on the following five twig queries in XMark[6] benchmark. For each twig pattern, Fig.3 shows the execution time and the total number of pages read using $TwigStack$ and $PPS-TwigStack$ with pruning on a XMark document of size 130MB.

1. */site/people/person/name*
2. */site/open_auctions/open_auction/bidder/increase/*
3. */site/closed_auctions/closed_auction/price*
4. */site/regions//item*
5. */site/people/person[//ageand//income]/name*

PPS-TwigStack is optimal for A-D or P-C only twig patterns and 1-branch twig patterns. Even without pruning, PPS-TwigStack still outperforms TwigStack in these classes of queries. We build XML documents of 1 million nodes with tags A, B, C, D, E such that no PP stream will be pruned away. The XML data generated has two sub-trees: the first contains only partial matches while the second contains full matches. We vary the size of the second sub-tree from 5% to 30% of the total document size and run the twig query $A/B[C//D]//E$.

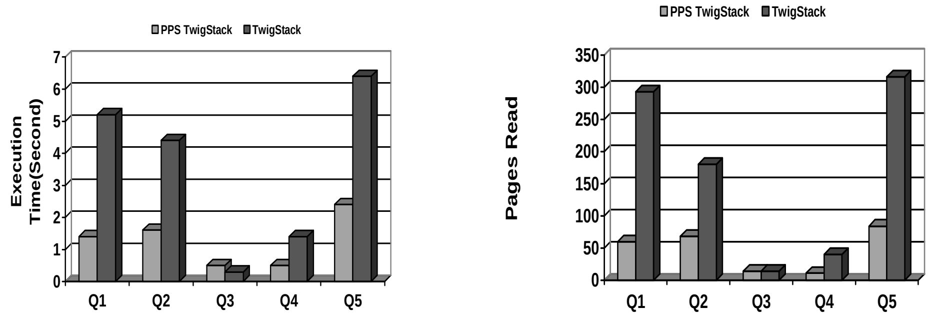

Fig. 3. Comparison of execution time and I/O cost of TwigStack and PPS-TwigStack on a 130MB XMark data

$PPS - TwigStack$ generates much smaller intermediate result sizes. As an example, when the ratio is 10%,$PPS - TwigStack$ produces 9,674 paths whereas $TwigStack$ produce 121,532.

8 Conclusion

In this paper, we address the problem of holistic twig join on XML documents. We propose a novel clustering method Prefix-Path Streaming (PPS) to group XML elements. The benefits of PPS streaming include reduced I/O cost and optimal processing of a large class of twig patterns with both A-D and P-C edges. In particular we show that A-D and P-C only twig and 1-branch twig patterns can be evaluated optimally using PPS.

References

1. N. Bruno, D. Srivastava, and N. Koudas. Holistic twig joins: optimal xml pattern matching. In *ICDE Conference*, 2002.
2. T. Chen, T.W. Ling, and C.Y. Chan. Prefix path streaming:a new clustering method for optimal holistic xml twig pattern matching. Technical report, National University of Singapore, http://www.comp.nus.edu.sg/ chent/dexapps.pdf.
3. B. Choi, M. Mahoui, and D. Wood. On the optimality of the holistic twig join algorithms. In *In Proceeding of DEXA*, 2003.
4. M.P. Consens and T.Milo. Optimizing queries on files. In *In Proceedings of ACM SIGMOD*, 1994.
5. S.Al-Khalifa, H. V. Jagadish, Nick Koudas, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural joins: A primitive for efficient xml query pattern matching. In *In Proceedings of ICDE*, pages 141–152, 2002.
6. XMARK. Xml-benchmark. http://monetdb.cwi.nl/xml.
7. XPath. http://www.w3.org/TR/xpath.
8. C. Zhang, J.F. Naughton, D.J. DeWitt, Q. Luo, and G.M. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD Conference*, 2001.