TwigBuffer: Avoiding Useless Intermediate Solutions completely in Twig Joins

Jiang Li and Junhu Wang

School of Information and Communication Technology
Griffith University, Gold Coast, Australia
Jiang.li@student.griffith.edu.au, J.Wang@griffith.edu.au

Abstract. Twig pattern matching plays a crucial role in XML data processing. `TwigStack` [2] is a holistic twig join algorithm that solves the problem in two steps: (1) finding potentially useful intermediate path solutions, (2) merging the intermediate solutions. The algorithm is optimal when the twig pattern has only //-edges, in the sense that no useless partial solutions are generated in the first step (thus expediting the second step and boosting the overall performance). However, when /-edges are present, a large set of useless partial solutions may be produced, which directly downgrades the overall performance. Recently, some improved versions of the algorithm (e.g., `TwigStackList` and `iTwigJoin`) have been proposed in an attempt to reduce the number of useless partial solutions when /-edges are involved. However, none of the algorithms can avoid useless partial solutions completely. In this paper, we propose a new algorithm, `TwigBuffer`, that is guaranteed to completely avoid the useless partial solutions. Our algorithm is based on an ingenious strategy to buffer and manipulate elements in stacks and lists. Experiments show that `TwigBuffer` significantly outperforms previous algorithms when arbitrary /-edges are present.

1 Introduction

The importance of fast processing of XML data is well known. *Twig pattern matching*, which is to find all matchings of a query tree pattern in an XML data tree, lies in the center of all XML processing languages. Therefore, finding efficient algorithms for twig pattern matching is an important research problem.

Over the last few years, many algorithms have been proposed to perform twig pattern matching. Most of these algorithms find twig pattern matching in two phases. In the first phase, a query tree is decomposed into smaller pieces, and solutions against these pieces are found. In the second phase, all of these partial solutions are joined together to generate the final results. Binary *structural join*(e.g.,[1]) and *holistic twig join*(e.g., [2–5]) are two important types of two-phase twig pattern matching algorithms. Holistic twig join algorithms have significantly reduced the number of useless intermediate path solutions compared with structural join algorithms. When only //-edges are present in a query tree,

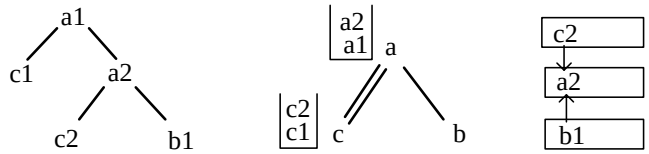


Fig. 1. An example to basic ideas of `TwigBuffer`

all of the intermediate paths will definitely appear in the final solutions. However, if /-edges are involved, none of them can completely eliminate the useless intermediate path solutions.

In this paper, we present a novel holistic twig join algorithm,`TwigBuffer`, that completely avoids the useless partial solutions for arbitrary twig patterns. With ingenious manipulation of data elements in *buffer stacks* and *result lists*, the algorithm also ensures the linear worst-case complexity in the first phase. Our experiments show that this algorithm significantly outperforms previous algorithms when arbitrary /-edges are present in the twig pattern.

The rest of the paper is organized as follows. `TwigBuffer` is presented in detail in Section 2. In Section 3, we show `TwigBuffer` correctly finds all twig matchings with low worse-case complexity. The experiment results are reported in Section 4. Finally, Section 5 concludes this paper.

2 `TwigBuffer`: our holistic twig join algorithm

2.1 Overview of `TwigBuffer`

We explain the basic ideas used in `TwigBuffer` using the example in Fig. 1.

`TwigBuffer` avoids useless partial solutions by doing a *thorough* check of the P-C relationships, so that the current element of node n is regarded useful only if it has a descendant in T_j for each child j of n, and it has a child in T_i for each child i of n that has the P-C relationship with n, and every child of n recursively satisfies the above condition. Like `TwigStackList`, it buffers elements read from the streams in order to check the P-C relationships; but unlike `TwigStackList`, it uses two different buffering policies. The first buffering policy, *PCBuffering*, is used to buffer ancestors elements (and to some extent this is similar to the buffering in `TwigStackList`). To conduct thorough checking of the P-C relationships, when the buffer stack is not empty, it will check the top element only. To ensure no useful element is abandoned, a second type of buffering, *Sbuffering*, is used to buffer elements that are potentially descendants of elements in the stack, so that these elements can be checked later for A-D or P-C relationships with the elements in the parent stack.

In the example above, a_1 and a_2 will be buffered using the first buffering policy because they are ancestors of b_1. Now the current element of node a is a_2 (note: in `TwigStackList`, the current element is a_1).After this step, the current elements of query nodes become a_2, c_1 and b_1, but we cannot abandon c_1 yet,

because we cannot conclude that it is not in a final solution (e.g., if a_1 has a b-child too). Therefore, we use a second buffering policy to buffer c_1 and c_2. Now the current elements become a_2, c_2 and b_1, and they are a solution of the twig pattern. After popping up a_2, we go back to process a_1. Since a_1 does not have a b-child, we know it can be abandoned.

2.2 Notation and Data structures

Similar with other holistic twig pattern matching algorithms (e.g. TwigStack, TwigStackList), the Containment labels of the elements with the same value are stored or organized in one stream for access. For each stream T_n, there exist a pointer PT_n pointing to the current element. The function $Advance(T_n)$ can make the pointer PT_n to point to the next element in the stream T_n. In addition, we can use $isEnd(T_n)$ to judge whether PT_n points to the position after the last element in the stream T_n.

There are two major types of data structures used in `TwigBuffer`. One is stacks for buffering elements read from the streams. The other is lists used for compactly storing and representing partial root-to-leaf solutions. Elements in a buffer stack are arranged in ascending order of the *start* value from bottom to top, but unlike TwigStackList, they may not be strictly nested. The elements in a result list are also in ascending order and strictly nested. An element is an ancestor of the elements after it.

For the buffer stacks, the basic operations are: *isEmpty*, *length*, *pop*, *push*, *top* and *bottom*. The functions built on the lists are: *head*, *tail*, *removeTail*, *isEmpty* and *insert*. The function *insert* should be noted. With it, an element can be inserted into a result list and the ascending order of elements is maintained. Similar to the linked stack in `TwigStack`, each element in the list contains a pointer pointing to an element in the parent list (See Fig. 1).

During query processing, the current element of each query node should be known. The basic rule is that if the buffer stack is not empty, the current element should stand at the top of the stack. Otherwise the current element will be the one in the stream pointed to by PT_n. Based on this rule, the function $getElement(n)$ will return the current element of node n for processing, and the function $proceed(n)$ will make the current element to be the next one:

$getElement(n)$:	returns $top(S_n)$ if S_n is not empty;
	returns $getElementFromStream(n)$ otherwise.
$proceed(n)$:	$pop(S_n)$ if S_n is not empty, advance(T_n) otherwise.

For the nodes in twig pattern Q, the functions $isRoot(n)$ and $isLeaf(n)$ checks whether node n is the root and is the leaf respectively, $parent(n)$ and $children(n)$ returns the parent of n and the set of children of n, respectively, and $isPCChild(n)$ returns TRUE if n is not the root and n is connected to its parent by a /-edge.

For any two elements e_1 and e_2 in data tree t, the function $isAD(e_1, e_2)$ returns TRUE iff e_1 is an ancestor of e_2. The function $subtreeNodes(q)$ returns all of the roots of q's subtrees.

2.3 `TwigBuffer`

The *getNext(n)* function The function $getNext(n)$, shown in Algorithm 1, is a core function of `TwigBuffer`. The function takes a query node n as input , and returns a query node that may be n itself or a descendant of n.

Algorithm 1 getNext(n)

```
if isLeaf(n) then
    return n
for all node n_i in children(n) do
    if isPCChild(n_i) AND isAD(getElement(n), getElement(n_i)) then
        PCBuffering(n, n_i)
    r_i = getNext(n_i)
    if r_i ≠ n_i then
        return r_i
n_min = minarg_{n_i ∈ children(n)} getElement(n_i).start
n_max = maxarg_{n_i ∈ children(n)} getElement(n_i).start
if ¬isEmpty(BS_n) AND ¬isEmpty(BS_{n_min}) then
    while getElement(n_min).end < getElement(n).start do
        SBuffering(n, n_min)
        if getElement(n_min).end < getElement(n).start then
            Proceed(n)
if ¬isEmpty(BS_{n_max}) AND ¬isEmpty(BS_n) then
    if getElement(n).end < getElement(n_max).start then
        return n_max
while getElement(n).end < getElement(n_max).start do
    Proceed(n)
PCBuffering(n, n_max)
n_min = minarg_{n_i ∈ children(n)} getElement(n_i).start
if getElement(n_min).start < getElement(n).start then
    return n_min
if ret := getNotSatisfyPC(n) then
    return ret
else
    return n
```

Before explaining the details of $getNext(n)$, we need to introduce the *buffering schemes*, which play an essential role in the matching process. Generally, there are two situations that buffering will happen:

First, if the query node n has the P-C relationship with its child n_i, and the current element in T_n is an ancestor of the current element in T_{n_i}, then buffering will occur and the general rule is:

PCBuffering: *Given a query node p and its P-C child c, suppose the current element of c is e_c. In the stream T_p, all of the elements that are ancestors of e_c will be buffered. The elements that are not ancestors of e_c will be skipped*

because they will not contribute to the final solutions. Apply this rule recursively on parent(p) if p is also a P-C child.

Second, a different buffering occurs when the pointer of a stream needs to advance, but its parent's buffer stack is not empty. In other words, the current element in the stream can not be abandoned at current stage because it may be in the final solutions. The buffering rule is:

SBuffering: *Given a query node p and its child c, suppose some elements are buffered (through PCBuffering or SBuffering) in the buffer stack BS_p of the node p. In the stream T_c, all of the elements whose start value lies in the range (*bottom*(BS_p).start,* top*(BS_p).start) will be buffered in BS_c firstly, and then, the first set of strictly nested elements whose start value lies in the range (*top*(BS_p).start,* top*(BS_p).end) will also be buffered. After buffering, all of the elements in T_p on the left of the nested elements just mentioned above are skipped because they will not contribute to fianl solutions. The buffering process above may change the current element of node c, which means the PCBuffering on p may become invalid if c is a P-C child, so PCBuffering on p needs redo.*

In the explanation below, we use $current(n)$ to denote the current element of node n.

Algorithm 2 Subroutines

```
procedure PCBuffering(p, c)
    e_p = getElementfromStream(p)
    e_c = getElement(c)
    while e_p.start < e_c.start do
        if e_p.end > e_c.end then
            ClearBufferStack(p, e_p)
            MovetoBufferStack(p, e_p)
        Advance(T_p)
    if isPCChild(p) AND isAD(getElement(parent(p)), getElement(p)) then
        PCBuffering(parent(p), p)
procedure SBuffering(p, c)
    Buffer all the elements of node c in the range of
    (Bottom(BS_p).start, Top(BS_p).start)
    Buffer the first set of nested elements of node c in the range of
    (Top(BS_p).start, Top(BS_p).end)
    if there are elements buffered AND isPCChild(c) then
        PCBuffering(p, c)
    while getElementfromStream(p).end < getElement(c).start do
        Advance(T_p)
procedure getNotSatisfyPC(n)
    for all node n_i in children(n) do
        if isPCChild(n_i) then
            if getElement(n_i).level − getElement(n).level ≠ 1 then
                return n_i
```

In Algorithm 1, lines 11 to 15 is an important step, which deals with the situation that both buffer stacks of a query node n and its child n_{min} are not empty. n_{min} is the child node whose current element has the minimum start value. The current element of node n should proceed if its start position is greater than the end position of the current element of node n_{min}, because it can not contribute to any useful path solutions in the future. Lines 16 to 18 deal with the situation that both buffer stacks of query node n and n_{max} are not empty.

Line 18 returns n_{max} because $current(n_{max})$ cannot contribute to the final solution. Lines 19 to 20 check whether $current(n)$ lies to the left of the current element of at least one of n's children, and if so, it cannot contribute to the final solution, and will be skipped. Line 21 should be noted. If the node n proceeds in the last step, the PCBuffering on the node n needs redone. All the ancestors of current element of node n_{max} will be buffered. Line 22 is used for re-acquire the n_{min}, because n_{min} may change due to the buffering actions.

The main algorithm Algorithm 3 presents the details of the main algorithm. It iteratively invokes *getNext*(n) to get the appropriate query node for further processing. If ancestors or parents can be found in the parent result list, the current element of the returned node will be moved into the result list. Otherwise, the returned node can not contribute to final solutions in the future, so its current element should be abandoned.

Line 4 should be noted. It is used for cleaning self result list to guarantee the elements are strictly nested. Additionally, when an element is inserted into the result list, the ascendant order in start value should be always kept. It should be noted that the action of clean parent's result list does not exist in TwigBuffer, but it exists in TwigStack and TwigStackList. This change is mainly because `TwigBuffer` adopts more complex buffering schemes, clean parent's result list may cause some elements removed too early.

3 Correctness and complexity of `TwigBuffer`

Due to the limit of space, the correctness proof is not included. This part can be found in the full paper. For **complexity**, since `TwigBuffer` uses more complicated buffering schemes to avoid useless partial solutions, one would naturally wonder whether the first phase of the algorithm has become computationally too expensive. We point out that although the element manipulation in `TwigBuffer` is more complex, the worse-case time complexity remains linear in the sum of the number of nodes in Q and the lengths of the output list.

4 Experiments

4.1 Experimental set-up

We implement TwigStack, TwigStackList and TwigBuffer in C programming language. The XML parser we used is Libxml2. All the experiments were per-

Algorithm 3 TwigPatternMatching(Q)

```
while ¬end(Q) do
    q_act := getNext(root(Q))
    if isRoot(q_act) OR ¬isEmpty(RL_parent(q_act)) then
        cleanSelfResultList(q_act)
        MovetoResultList(q_act)
        if isLeaf(q_act) then
            showSolutions(q_act)
            removefromRL(q_act)
    else
        if length(BS_parent(q_act)) > 1 AND getElement(q_act).start > bottom(BS_parent(q_act)).start then
            SBuffering(parent(q_act), q_act)
        else
            Proceed(q_act)
procedure END(q)
    return ∀q_i ∈ subtreeNodes(q) : isLeaf(q_i) ⋀ isEmpty(BS_n) ⋀ end(T_n)
procedure CLEANSELFRESULTLIST(n)
    if isEmpty(BS_n) then
        while getElement(n).end < tail(RL_n).start OR getElement(n).start > tail(RL_n).end do
            RemoveTail(n)
procedure MOVETORESULTLIST(n)
    p:= getPointer(n)
    e := getElement(n)
    Insert result node (e, p) to RL_n, the ascendant order in start value should be kept
procedure GETPOINTER(n)
    p points to the end of RLparent(n)
    e := getElement(n)
    while e.start < p.start OR e.end > p.end do
        p := previous(p)
    return p
```

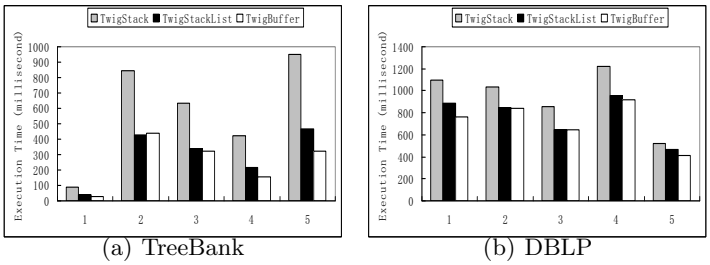
formed on 1.6GHz Intel Centrino Duo processor with 1G RAM. The operating system is Windows XP. We used the following two data sets for evaluation: TreeBank and DBLP obtained from the University of Washington XML repository. The metrics of evaluation we selected is running time.

The queries on both data sets are presented in Table 1. It can be seen that all the queries have different twig structures. This consideration will make the comparisons more comprehensive.

The experimental results are illustrated in Fig.2. As shown, the performance of `TwigStackList` is nearly the same with `TwigBuffer` on the queries that do not have /-edges or /-edges happen under non-branching nodes. However, `TwigBuffer` performs better than `TwigStackList` when the queries have /-edges under branching nodes.

Data set	Query	XPath expression
TreeBank	Q1	//S[//MD]//ADJ
TreeBank	Q2	//S[//VP/IN]//NP
TreeBank	Q3	//S[/JJ]/NP
TreeBank	Q4	//S/VP/PP[/IN]/NP/VBN
TreeBank	Q5	//EMPTY[//VP/PP//NNP][/S[//PP//JJ]/VBN]//PP/NP
DBLP	Q1	//dblp/inproceedings[//title]/author
DBLP	Q2	//dblp/article[//author][//title]//year
DBLP	Q3	//dblp/inproceedings[//cite][//title]/author
DBLP	Q4	//dblp/article[//author][//title][//url][//ee]//year
DBLP	Q5	//article[//volume][//cite]//journal

Table 1. Queries over TreeBank and DBLP



(a) TreeBank (b) DBLP

Fig. 2. Experiment results

5 Conclusion

We presented a novel holistic twig join algorithm that efficiently finds root-to-path matchings. Our algorithm completely avoids useless intermediate path matchings for arbitrary twig patterns, and thereby improves the overall performance of previous two-phase twig join algorithms. The better overall performance of our algorithm has been substantiated in our experiments.

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 *ICDE*, pages 141–, 2002.
2. N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *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 *SIGMOD Conference*, pages 455–466, 2005.
4. J. Lu, T. Chen, and T. W. Ling. Efficient processing of XML twig patterns with parent child edges: a look-ahead approach. In *CIKM*, pages 533–542, 2004.
5. T. Yu, T. W. Ling, and J. Lu. TwigStackList-: A holistic twig join algorithm for twig query with not-predicates on XML data. In *DASFAA*, pages 249–263, 2006.