TwigStackPrime: A Novel Twig Join Algorithm Based on Prime Numbers

Shtwai Alsubai and Siobhán North

Department of Computer Science,
the University of Sheffield, Sheffield, UK
{safalsubai1,s.north}@sheffield.ac.uk

Abstract. The growing number of XML documents leads to the need for appropriate XML querying algorithms which are able to utilize the specific characteristics of XML documents. A labelling scheme is fundamental to processing XML queries efficiently. They are used to determine structural relationships between elements corresponding to query nodes in twig pattern queries (TPQs). This article presents a design and implementation of a new indexing technique which exploits the property of prime numbers to identify Parent-Child (P-C) relationships in TPQs during query evaluation. The Child Prime Label (CPL, for short) approach can be efficiently incorporated within the existing labelling schemes. Here, we propose a novel twig matching algorithm based on the well known TwigStack algorithm [3], which applies the CPL approach and focuses on reducing the overhead of storing useless elements and performing unnecessary join operations. Our performance evaluation demonstrates that the new algorithm significantly outperforms the previous approaches.

Key words: XML Databases, Holistic Twig Join Algorithm, Node Labelling, Twig Pattern Query

1 Introduction

As enterprises and businesses produce and exchange XML-formatted information more frequently, consequently, there is an growing requirement for effective handling of queries on data which conforms to an XML format [17, 15, 16]. Recently, several approaches have been proposed in the literature to process XML queries [14, 3, 10, 20, 9, 17, 7, 12, 8, 11]. Due to the definition of relationships in XML as nested tags, data in XML documents are self-describing and flexibly organized [16, 8]. Therefore, the basic XML data model is a labelled and ordered tree.

In most XML query languages, such as XPath and XQuery, a twig (small tree) pattern can be represented as a node-labelled tree whose edges specify the relationship constraints among its nodes and they are either Parent-Child or Ancestor-Descendant. As a result, an XML query is defined as a complex selection on elements of an XML document specified by structural information of the selected elements. Improving the efficiency of tree patterns matching is a

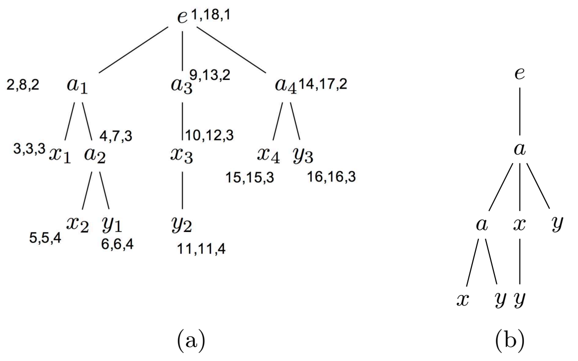


Fig. 1. (a) Range-based labelling scheme and (b) its *DataGuide*

core operation in processing of an XML query [2, 10, 9, 17, 4] since tree patterns are the basis for querying structured tree-based data model such as XML.

Generally, the purpose of XML indexing is to improve the efficiency and scalability of query processing by reducing the search space. Without an index, XML retrieval algorithms have to scan all the data. Most existing XML query processing algorithms [20, 22, 12, 9, 8, 21] rely on XML indexing techniques to scan only the XML data relevant to XML queries, therefore, XML query performance is improved.

In XML, there are two basic type of indices. The first one is to index each node in an XML document by recording its positional information [13, 15]. This group is well-known as node label or labelling schemes. In this group of indices, every node in an XML document is assigned an unique label to record its position within the original XML document. The labelling scheme should enable determination of the structural information, i.e., Parent-Child (P-C) and Ancestor-Descendant (A-D) relationships. As a result, for any given two elements in an XML document, the relationship between them (if it exists) can be computed in constant time. A well-known example of node labelling is the containment labelling scheme proposed in [13]. In this approach, each node is assigned with a tuple of three values as $< start, end, level >$. *Start* and *end* contain values of positions corresponding to the opening tag $< tag - name >$ and the closing tag $< /tag - name >$. Level represents the depth of the node within its XML tree. The two basic relationships Ancestor-Descendant and Parent-Child can be determined efficiently. Given two nodes u and v, u is an ancestor of v if and only if $u.start < v.star < v.end < u.end$. A Parent-Child relationship is defined as node u is the parent of node v if and only if $u.start < v.star < v.end < u.end$, $v.level = u.level + 1$.. by way of explanation, the u node is in the range of node v.

The alternative to node labelling uses root-to-node paths in the XML document and is well-known as graph indexing (also referred to as structural summary or path indexing). Because an XML document can be modelled as rooted, ordered, labelled tree, a labelled path is defined as a sequence of tag names in the form of $tag_1/tag_2/\ldots/tag_n$ from the root represented by tag_1 to node n tagged by tag_n. For illustration, consider the XML tree in Figure 1, elements tagged by a can be stored in different storage structures according to their unique labelled paths. Consequently, elements corresponding to the path e/a are $\{a_1, a_3, a_4\}$, while a_2 is stored alone in its distinct labelled path $e/a/a$. A classic example of a path index is *DataGuide* [18]. The main drawback of this approach is that it only supports a simple path queries. There exist some graph indices that cover twig path queries as [19] but one of the limitations with these indices is that they are very large [20].

Both node and graph indexing are essential to XML query processing algorithms, they play important role in providing efficient evaluation of queries with respect to CPU complexity and memory consumption overhead [22, 2]. According to [23], a labelling scheme has to guarantee uniqueness and order preservation of node labels, thus the hierarchical relationships between a pair of nodes can be determined efficiently. The labelling scheme should enable checking all XPath relationships by computations only. To better understand the mechanisms of node indexing methods and their properties, [15] classified node indexing into four distinct types; range-based, prefix-based, multiplicative and hybrid labelling. A range-based labelling scheme will be adopted in this article to explore the effect of the new indexing mechanism. For sake of simplicity the following example 1 aims to explain the use of labels in the determination of hierarchical relationships in XML trees.

Example 1. Consider Figure 1, the structural relationships between the elements can be determined according to the properties for ancestor-descendant and parent-child relationships, respectively. Consider the relationship between node a_1 and y_1, as the elements are labelled based on containment labelling scheme proposed in [13]. a_1 is an ancestor of y_1 because $2 < 6 < 8$. Also, a_1 is a parent node of a_2 because the parent-child conditions are satisfied as $2 < 4 < 8 \; and \; 2 + 1 = 3$.

Organization. The rest of this article is organised as follows. Section 2 shows the related work. The new indexing technique will be introduced in Section 3. In Section 4, we present a holistic twig join matching algorithm *TwigStackPrime*. Section 5 presents thorough experimental studies about the performance between the new algorithm and the previous approaches. We conclude the paper in Section 6

2 Related Work

Every XML query processing algorithm which performs structural join operations to match a given query against an XML document relies on either range-based labelling schemes or prefix-based labelling schemes [3, 10, 17, 14]. This is due to the fact that labelling schemes where nodes are considered as the basic unit of a query provides great flexibility in performing any structural query matching efficiently. The information gained from labels varies according to the chosen labelling scheme. To determine the effects of the range-based labelling scheme, [13] proposed multi-predict merge-join algorithm based on the positional information of the XML tree. An alternative representation, a prefix scheme, of labels of an XML tree can be seen in [10]. In this sort of labelling scheme, each node is associated with a sequence of integers that represents the node-ID path from the root to the node. This approach can be exemplified by the Dewey system used by librarians, the sequence of components in a Dewey label is separated by "." where the last component is called the self label (i.e., the local order of the node) and the rest of the components are called the parent label. For instance, {1.2.3} is the parent of {1.2.3.1}. Another approach, [1] addressed the limitations of information encoded within labels produced by existing labelling schemes. It focus on performing join operations earlier, at leaf levels, where the selectivity of query nodes is at its peak for data-centric XML documents. The significance of the proposed approach stems from a comprehensive labelling scheme that encodes additional structural information, called *Nearest Common Ancestor, NCA for short* rather than the basic relationships among elements of XML documents. None of the previous approaches have taken the breadth of every node into account. In this paper, we propose a novel approach to overcome the previous limitations.

One of the most important problems in XML query processing is tree pattern matching. Generally, tree pattern matching is defined as mapping function M between a given tree pattern query Q and an XML document D, $M : Q \to D$ that maps nodes of Q into nodes of D where structural relationships are preserved and the predicates of Q are satisfied. Formally, tree pattern matching must find all matches of a given tree pattern query Q on an XML document D.

Early work on processing twig pattern matching decomposed twigs into a set of binary structures, then performed structural joins to obtain individual binary matchings. The final solution of the twig query is computed by stitching together the binary matches. In [3], the authors introduced the first holistic twig join algorithm for matching an XML twig pattern, called *TwigStack*. It works in two phases. Firstly, twig patterns are decomposed into a set of root-to-leaf paths queries and the solutions to these individual paths are computed from the data tree. Then, the intermediate paths are merge joined to form the final result. The authors of [3] proposed a novel prefix filtering technique to reduce the number of irrelevant elements in the intermediate paths.

The classical holistic twig join algorithm *TwigStack* only considers the ancestor-descendant relationship between query nodes to process a twig query efficiently without storing irrelevant paths in intermediate storage. It has been reported [3] that it has the worst-case I/O and CPU complexity when all edges in twigs are "//" (AD relationship) linear in the sum of the size of the input and output lists. However, *TwigStack*'s performance suffers from generating useless intermediate results when twig queries encounter Parent-Child relationships.

The authors of [9] proposed the first refined version of *TwigStack*. They introduced a new buffering technique to process twig queries with *P-C* relationships more efficiently by looking ahead some elements with *P-C* relationships in lists to eliminate redundant path solutions. *TwigStackList* guarantees every single path generated is a part of the final result if twig queries do not have *P-C* under branching query nodes. Subsequently, *TwigStackList* ensures optimal CPU and I/O cost when twig queries contain only Ancestor-Descendant edges below branching nodes and allows the occurrence of Parent-Child elsewhere [9]. The authors of [6] have proven that the *TwigStack* algorithm and its variants which depend on a single sequential scan of the input lists can not be optimal for evaluation of tree pattern query with any arbitrary combination of ancestor-descendant and parent-child relationships.

The approach to examine XML queries against document elements in post-order was first introduced by [4], $Twig^2Stack$. The decomposition of twigs into a set of single paths and enumeration of these paths is not necessary to process twig pattern queries. The key idea of their approach is based on the proposition that when visiting document elements in post-order, it can then be determined whether or not they contribute to the final result before storing them in intermediate storage which is trees of stacks to ensure linear processing. *TwigList* [11] replaced the complex intermediate storage proposed in $Twig^2Stack$ with lists (one for every query node) and pointers with simple intervals to capture structural relationships. The authors in [7] proposed a new storage scheme, level vector split which splits the list connected to its parent list with *P-C* edge to a number of levels bounded by the maximum depth of the XML tree. A combination of pre-order and post-order filtering methods is adopted to develop two algorithms, namely: *TJStrictPre* and *TJStrictPost*. Although, they can prune irrelevant elements when P-C edges exist, they still perform unnecessary computations and store useless elements corresponding to leaf query nodes.

3 Child Prime Label

We present a new indexing technique which can be applied to the existing labelling schemes to skip scanning useless elements in the streams during the processing of twig pattern queries with Parent-Child edges. The key idea of our work is to find an appropriate, refined labelling scheme such that, for any given query node in the TPQ, the set of its child query nodes in the XML document, this forms the major bottleneck in determining structural relationship because Parent-Child can be resolved efficiently. This novel approach results in considerably fewer single paths stored than existing algorithm. It also increases the overall performance and reduces the memory overhead, and the result is shown clearly in our experiments.

The idea is to identify all the distinct tags in the XML tree and assign them with unique prime numbers. Then, the intuition of the CPL is to use the modulo function to create a mapping from an integer to a set of element names. The leaf elements will not be annotated with CPLs, whereas the inner elements (i.e., parent elements) are assigned CPLs. During depth-first scanning, an element is assigned the next available prime number if its tag has not been examined. After that, we check the CPL parameter of its parent element to see whether it is divisible by the assigned prime number or not. If it is, we process the next element, otherwise the product of parent element's CPL is multiplied by the new prime number. We index tags for each XML tree in *tag indexing* to create a mapping from an element tag to a prime number as in Equation 1. The tag indexing is implemented by a mean of *hash* table. For illustration, consider an element e, with all distinct names of children, $C = \{c_1, c_2, \ldots, c_m\}$ and a list of prime numbers $P = \{p_1, p_2, \ldots, p_n\}$. The bijective mapping function $f : C \to P$ for all element $p \in P$, there is a unique element $c \in C$ such that $f(c) = p$. Then, the CPL for element e can be computed as follows:

$$CPL(e) = \begin{cases} \prod_{i=1}^{m} f(c_i), & \text{if } m \geq 1 \\ \emptyset, & \text{otherwise} \end{cases} \tag{1}$$

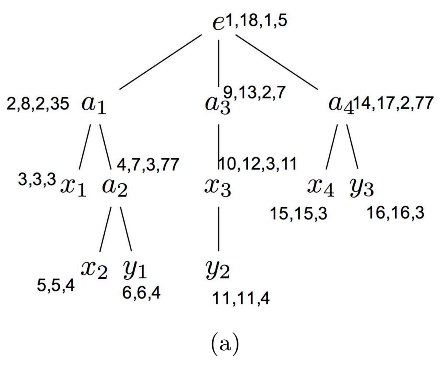
Proposition 1 (Uniqueness). *There is only one unique set of prime factors for any number.*

To explore the effect of CPL approach, we extend the original range-based labelling scheme to incorporate the CPL information. Each range-based label with CPL is presented as quadruple =*(start,end,level,CPL)*. The first three attributes remain the same as in the original labelling scheme see Section 1. According to Proposition 1, all distinct names of immediate child elements for a particular element in the XML tree can be obtained from having the corresponding prime numbers associated with tag names of its children.

Definition 1. ***(Child Prime Label)*** *A child prime label is assigned to each element in an XML document as an extra parameter into the range-based label. A child prime label indicates the multiplication of distinct prime numbers for every internal elements within the document. For example, node u is encoded quadruple =*$(start_u, end_u, level_u, CPL_u)$*.*

Property 1. In any XML labelling scheme that is augmented with Child Prime Label, for any nodes x,y and z in an XML document, x has at least one or more child nodes of tag(y) and tag(z) if $CPL_x \ mod \ key_tag(y) \times key_tag(z) = 0 \ where \ key_tag(y) \ and \ key_tag(z)$ are unique prime numbers.

To demonstrate the effect of child prime label, consider the XML tree in Fig. 2 and the tag indexing table on the top right, queries in XML are expressed as twigs since data is represented as tree. The answer to an XML query is all occurrences of it in an XML document under investigation. So, if we issue the simple twig query $Q \ = a[x]/y$, only two elements will be considered for further processing, namely $a_2 \ and \ a_4$. This is because of $CPL_{a_2} \ mod \ key_tag(x) \times key_tag(y) = 77 \ mod \ 7 \times 11 \ equals \ 0$.



Tagname	Key
e	3
a	5
x	7
y	11

(b)

Fig. 2. (a) a sample of an XML tree labelled with the original range-based augmented with CPL parameters and (b) its corresponding tag indexing.

4 Twig Join Algorithm

4.1 Notation

There is abstract data type called a stream, which is a set of elements with the same tag name, where the elements are sorted in ascending document order. Each query node q in a twig pattern is associated with an element stream, named T_q which has a cursor C_q which initially points to the first element in T_q at the beginning of a query processing. To ensure the linear processing in the filtering phase of holistic algorithms, only the first element is accessible and the rest of the elements are unseen by the algorithms. There are also some auxiliary operations on streams and TPQ and its nodes to facilitate the twig matching process. Supported operations are as follows: *getStart(C_q)* returns the start attribute of the first element of q. *getEnd(C_q)* returns the end attribute of the first element of q. *getLevel(C_q)* returns the level attribute of the first element of q. *getCPL(C_q)* returns the CPL attribute of the head element corresponding to query node q. *tagPrime(q)* returns the unique prime number associated with q from *tag indexing*. *advance(C_q)* forward the cursor of q to the next element. $eof(T_q)$ to judge whether C_q points to the end of stream of T_q. *children(q)* returns all child nodes of q. *subtree(q)* returns all child nodes which are in the subtree rooted at q. *childrenAD(q)* returns all child nodes which have A-D relationship with q. *childrenPC(q)* returns all child nodes which have P-C relationship with q. *isRoot(q)* returns boolean values to see whether q is the root or not. *getRoot(TPQ)* returns the query root of the input TPQ. *parent(q)* returns the parent query node of q. *isLeaf(q)* returns boolean values to see whether q is a leaf node or not.

4.2 TwigStackPrime

In this section, we present a new holistic twig join algorithm, called TwigStack-Prime. It can be seen as an alternative to TwigStack algorithm. The structure of the main algorithm, *TwigStackPrime* presented in Algorithm 2 is not much different from the original holistic twig join algorithm *TwigStack* [3] which uses two phases to compute answers to a TPQ. In the first phase, solutions to root-to-leaf paths in a TPQ are found and stored in output arrays (Lines 1-11). It repeatedly calls the *getNext* algorithm (see Algorithm 1) with the query root as the parameter to return the next query node for processing. In the second phase (Line 12), solutions in the output arrays are merge-joined based on their common branching query nodes and query matches are returned as the query result. The number of output arrays is equal to the number of leaf query nodes (i.e., the number of individual root-to-leaf paths in a TPQ).

getNext is a fundamental function which is called by the main algorithm to decide the next query node to be processed. It is used to guarantee that the current element associated with the query node returned is part of the final output since all the basic structural relationships are thoroughly checked by *getNext* or its supporting subroutine *getElement*. *getNext(q)* returns an element e_q of a query node $q \in TPQ$ with three properties:

i e_q has a descendant element e_{q_i} in each of the streams corresponding to its child elements where e_{q_i} is the first element of a query node q_i = *children(q)* (this property is checked in Lines 9-11).

ii each of its child elements satisfies recursively the first property (this property is checked in Lines 4-5).

iii if q has Parent-Child edge(s) with its child query nodes, then e_q has a child e_{q_i} in T_{q_i} for each query node q_{q_i} = childrenPC(q) (this property is checked in Lines 21-23 of *getElement* function).

In the function *getElement(q)*, if q does not have P-C edges, the first element of q is returned. Otherwise, Line 22 checks CPL relationship for all child query nodes with P-C relationships. If the first element does not satisfy the CPL relationship (the third property), the function skips all elements which do not satisfy the CPL relationship. Otherwise, the first element is found to satisfy the CPL relationship, then it is returned in Line 26 if the stream is unfinished. In case the stream reaches the end, Line 24 returns virtual end element labelled with infinity values as (∞, ∞, ∞) to complete the query processing.

Compared to the original *TwigStack* which does not apply CPL relationships, the effect of *TwigStackPrime* can be illustrated in the following example.

Example 2. Consider the XML tree of Fig. 3 and Q_1 = a[//x]/y. Assume the tree is labelled with range-based labelling and CPL approach as in Fig. 2. Initially, the cursors point at the first elements in streams. *getNext(a)* is called since a is the root query node. The first call of *getNext(a)* in *TwigStack* returns a_1 because it satisfies the descendant extension condition, but *TwigStackPrime* skips a_1 since it does not satisfy the CPL relationship that is CPL of a_1 is not

Algorithm 1: getNext(q) [2]

```
   Input: q is a query node
   Result: a query node in TPQ which may or may not be q
1  if isLeaf(q) then
       return: q
2  foreach node $n_i$ in children(q) do
3      $g_i$ = getNext($n_i$) if $g_i \neq n_i$ then
           return: $g_i$
4  $n_{max}$ = a query node with the maximum start value $\in$ children(q)
5  $n_{min}$ = a query node with the minimum start value $\in$ children(q)
6  while getEnd(getElement(q)) < getStart(getElement($n_{max}$)) do
7      advance(q)
8  if getStart(getElement(q)) < getStart(getElement($n_{min}$)) then
       return: q
9  else
       return: $n_{min}$
10 Function getQCPL(Query node q):
11     // the prime number assigned to the query node which is the product of its
       child query node prime numbers
12     qCPL = 1
13     foreach node $n_i$ in childrenPC(q) do
14         qCPL = qCPL $\times$ tagPrime($n_i$)
       return: qCPL
15 Function getElement(Query node q):
16     if childrenPC(q) > 0 then
17         while $\neg$ eof($C_q$) $\wedge$ getCPL($C_q$) % getQCPL(q) $\neq$ 0 do
18             advance(q)
19     if eof($C_q$) then
           return: $\infty, \infty, \infty, 1$ // out of range label
20
21     else
           return: $C_q$ // the current head element in the stream of q
22
```

divisible by the prime number associated with the tag name y. The algorithm has to skip n elements with tag x since they are useless to the first element a_2.

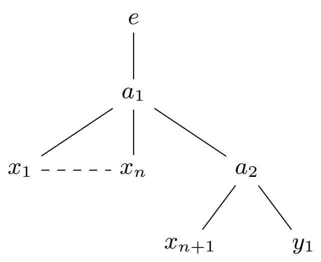


Fig. 3. Illustration to the suboptimal processing of TwigStack.

Algorithm 2: TwigStackPrime [2]

```
Input: TPQ Q
1  while ¬end(getRoot(Q)) do
2      $q_{act}$ = getNext(getRoot(Q)) // see Algorithm
3      if ¬ isRoot(q) then
4          cleanStack(getElement($q_{act}$), parent($q_{act}$))
5      if isRoot(q) ∨ ¬ empty($S_{parent(q_{act})}$) then
6          cleanStack(getElement($q_{act}$),$q_{act}$)
7          moveToStack($q_{act}$)
8          if isLeaf($q_{act}$) then
9              outPathSolution($q_{act}$) // Blocked solutions
10     else
11         advance($q_{act}$)
12 MergeAllPathSolutions() // Phase 2
13 Function cleanStack(Query node $q_{act}$,Query node q):
14     // pop any element in $S_q$ which is not the ancestor of getElement($q_{act}$)
15     while $\neg empty(S_q) \wedge getEnd(top(S_q)) < getStart(getElement(q_{act}))$ do
16         $pop(S_q)$
17 Function moveToStack(Query node q):
18     // p is a pointer to the top parent stack if q is the root p is null
19     // p = top($S_{parent(q)}$)
20     $push(C_q, p)$ to $S_q$
21 Function end(Query node q):
       return: $\forall n_i \in subtree(q) : isLeaf(n_i) \wedge eof(C_{n_i})$
```

After this, *TwigStackPrime* can ensure that a_2 satisfies the three properties and thus is pushed into the stack for query node a. For instance, $CPL(a_1) \rightarrow 35$ mod $tagPrime(y) \rightarrow 11$ is not equal to zero. The algorithm terminates after performing one recursive calls of *getNext(a)*. On the other hand, *TwigStack* has to iterate *n+1* times to answer match to Q_1. *TwigStack* also generates n useless paths for Q_1 over the given XML tree.

4.3 Analysis of TwigStackPrime

In this section, we show the correctness of our algorithms. The correctness of *TwigStackPrime* algorithm can be shown analogously to *TwigStack* due to the fact that they both use the same stack mechanism. In other words, the correctness of Algorithm 2 follows from the correctness of *TwigStack* [3].

Definition 2 (Head element). *For each query node q in a TPQ Q, the element indicated by the cursor C_q is the head element of q.*

Definition 3 (Child and Descendant Extension). *A query node q has the child and descendant extension if the following properties hold:*

– *$\forall$ $n_i \in childrenAD(q)$, there is an element e_i which is the head of T_{n_i} and a descendant of e_q which is the head of T_q.*

– *$\forall$ $n_i \in childrenPC(q)$, there is an element e_q which is the head of T_q and its CPL parameter is divisible by tagPrime(n_i).*
– *$\forall$ $n_i \in children(q)$, n_i must have the child and descendant extension.*

The above definition is a key for establishing the correctness of the following lemmas:

Lemma 1. *For any arbitrary query node q' which is returned by getNext(q), the following properties hold:*

1. *q' has the child and descendant extension.*
2. *Either $q == q'$ or q' violates the child and descendant extension of the head element e_q of its parent(q').*

Proof. (Induction on the number of child and descendants of q). If q is a leaf query node, it is returned in Line 2 because it verifies all the properties 1 and 2a in Lemma 1. Otherwise, the algorithm recursively gets g_i = *getNext(n_i)* for each child of q in Line 4. If for some i, there is $g_i \neq n_i$, and it is known by inductive hypothesis that g_i verifies the properties 1 and 2b with respect to q, so the algorithm returns g_i in Line 6. Otherwise, by inductive hypothesis that all q's child nodes satisfy properties 1 and 2a with their corresponding sub-queries. At *getElement(q)* (Lines 21-25), *getNext* advances from T_q all segments that do not satisfy the divisibility by the product of prime numbers in $childrenPC(q)$ returned from *getQCPL*. After that, the algorithm advances from T_q (Lines 9-10) all segments that are beyond the maximum start value of $n_i \in children(q)$. Then, if q satisfies properties 1 and 2a, it is returned at Line 12. Otherwise, Line 13 guarantees that $n_i \in children(q)$ with the smallest start value satisfies properties 1 and 2b with respect to start value of q's head element e_q is returned.

Lemma 2. *Suppose $getNext(q)$ returns a query node q' and $q \neq q'$ at either Line 4 or 13 of getNext. Then there is no new solution involving top element of the parent stack of q' denoted as p which has end value less than the start value of the head element of q' or some elements which are in children(p).*

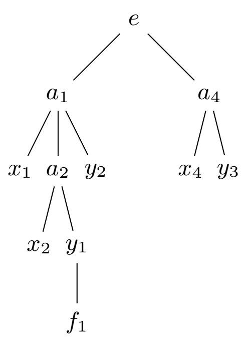
Proof. Suppose that on contrary, there is a new solution using some elements of p = parent(q') in S_p denoted as e_{S_p} for which getEnd(e_{S_p}) ¡ getStart(q'). Using range-based property, it will be known that all elements from children(p) in some solutions must have end values less than the end value of e_{S_p}, therefore less than the start value of the head element of q'. Since $getNext(q) = q'$ and from Line 3 of *getNext* for each child node n_i of p (including q'), it is *getNext(n_i)* $= n_i$ and getStart(q') $\leq$ getStart(n_i). Using Lemma 1, it will be known that each n_i has a child and descendant extension, and thus all elements of children(n_i) have start values greater than getStart(n_i), therefore greater than getStart(q'), which is a contradiction.

Theorem 1. *Given a twig pattern query Q and an XML document D, Algorithm TwigStackPrime correctly returns answer to Q on D.*

Proof. In Algorithm *TwigStackPrime*, *getNext(root)* is repeatedly invoked to determine the next query node to be processed. Using lemma 1, it is known that all elements returned by q_{act} = *getNext(root)* have the child and descendant extension. If $q_{act} \neq root$, Line 4, the algorithm pops from $S_{parent(q_{act})}$ all elements that are not ancestors of the head element of q_{act} by Lemma 2. After that, it is already known q_{act} has a child and descendant extension so that Line 5 checks whether $S_{parent(q_{act})}$ is empty or not. If so, it indicates that it does not have the ancestor extension, and it can be discarded safely to continue with the next iteration. Otherwise, the current head element of q_{act} has both the ancestor and child and descendant extensions which guarantee its participation in at least one root-to-leaf path. Then, $S_{q_{act}}$ is cleaned by popping elements which do not contain the head of q_{act}. Then, the item in the stack is used to maintain pointers from itself to the query root. Finally, if q_{act} is a leaf node, we compute all possible combinations of single paths with respect to q_{act}, line 8-9.

The correctness holds for TPQs with both A-D and P-C relationships, it can be shown that *TwigStackPrime* algorithm is optimal when P-C axes exist only in the deepest level of a twig query. The intuition is simple since the CPL relationship can detect hidden immediate child elements only in two streams related by P-C relationships. Henceforth, we can conclude the following result.

Theorem 2. *Consider a twig pattern query Q with n query nodes, and only Ancestor-Descendant edges or there are Parent-Child edges to connect leaf query nodes, and an XML document D. TwigStackPrime has worst-case I/O and CPU time complexities linear in the sum of the size of the n input lists and the output list.*



(a) an XML tree.

Tagname	Key
e	3
a	5
x	7
y	11
f	13

(b) tag indexing.

Fig. 4. Sub-optimal evaluation of *TwigStackPrime* where redundant paths might be generated.

Example 3. Consider the XML tree of Fig. 4 and Q_2 = a[/x]/y/f, the head elements in their streams are $a \rightarrow a_1,\ x \rightarrow x_1,\ y \rightarrow y_1$ *and* $f \rightarrow f_1$. The first call of *getNext(root)* inside the main algorithm will return $a \rightarrow a_1$ because it has A-D relationship with all head elements and satisfies *CPL* with x and y, and its descendant $y \rightarrow y_1$ also satisfies the child and descendant extension with respect to *f*. However, *TwigStackPrime* produces the useless path (a_1, x_1) because y_2 does not have child of f-node [2].

5 Experimental Evaluation

In this section we present the performance comparison of twig join algorithms, namely: *TwigStackPrime* the new algorithm based on *Child Prime Label* approach, along with *TwigStack* [3]. The original twig join algorithm that was reported to have optimal worst-case processing with A-D relationship in all edges, and *TwigStackList* is the first refined version of *TwigStack* to process P-C efficiently [9]. *TwigStackList* was chosen in this experiment because it utilizes a simple buffering technique to prune irrelevant elements from streams. We evaluated the performance of these algorithms against both real-world and artificial datasets. The performance comparison of these algorithms was based on the following metrics:

1. Number of intermediate solutions: the individual root-to-leaf paths generated by each algorithm.
2. Processing time: the main-memory running time without counting I/O costs. All twig pattern queries were executed 103 times and the first three runs were excluded for cold cache issues. We did not count the I/O cost for tag indexing files for *TwigStackPrime* algorithm because it is negligible, and the cost to read the tag indexing is constant over a series of twig pattern queries.

5.1 Experimental Settings

All the algorithms were implemented in Java JDK 1.8. The experiments were performed on 2.9 GHz Intel Core i5 with 8GB RAM running in Mac OS X El Capitan. The benchmarked data sets used in the experiments and their characteristics are shown in Table 1. The selected datasets and benchmark are significantly more frequent in the literature of XML query processing [3, 9, 7, 12, 8, 11]. DBLP is a highly structured document and is very wide and shallow, while TreeBank is a deep-recursive dataset with a large number of distinct tags and has irregular structure. Both are real-world and obtained from the University of Washington XML repository [25]. The XMark dataset is well-known benchmarked XML dataset [24]. To ensure fair comparison, DBLP and XMark datasets were selected because they are both considered as data-oriented and have very strong structures. We also generated Random dataset similar to that in [9] but we have the two parameters: *depth* and *fan-out*. The depth of randomly generated tree has maximum value sets to 13 and *fan-out* has range from 0 to 6, respectively. This dataset was created to test the performance where the XML document combines features of DBLP and TreeBank, being structured and deeply-recursive at the same time.

Table 1. Characteristics of XML datasets used in the experiments.

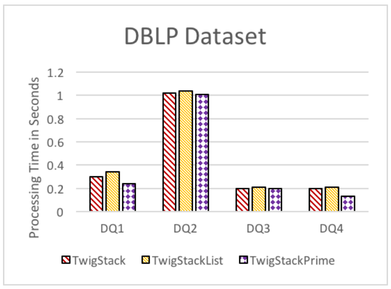
	DBLP	TreeBank	XMark	Random
Rangae-based MB	65.3	43	35.3	69.4
CPL MB	70.3	47.9	40.1	74.1
$\triangle$ size MB	5	4.9	4.8	4.7
Tag Indexing Size KB	0.48	3	1	0.049
Nodes (Millions)	3.73	2.43	2.04	3.94
Max/Avg depth	6/2.9	36/7.8	12/5.5	13/7
Distinct Tags	40	251	83	6
Largest Prime Numbers	151	1597	379	19

The XML structured queries for evaluation over these dataset were chosen specifically because it is not common for queries, which contain both '//' and '/', to have a significant difference in performance for tightly-structured document such as DBLP and XMark. TreeBank twig queries were obtained from [9] and [7]. Twig patten queries over the random data set were also randomly generated. Table 2 shows the XPath expressions for the chosen twig patterns. The code indicates the data set and its twig query, for instance, TQ2 refers to the second query issued over TreeBank dataset.

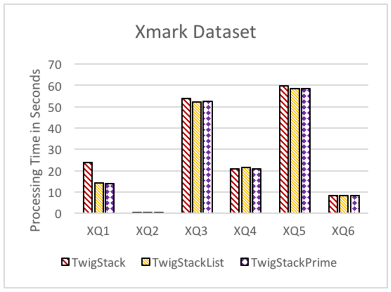
5.2 Experimental Result

We compared *TwigStackPrime* algorithm with *TwigStack* and *TwigStackList* over the above mentioned twig pattern queries against the data sets selected. The Kruskal-Wallis test is a non-parametric statistical procedure was carried out on processing time, the *p-value* turns out to be nearly zero (p-value less than 2.2^{-16}), it strongly suggests that there is a difference in processing time between two algorithms at least as shown in Fig. 5.

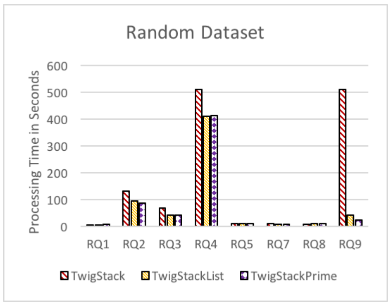
TwigStackPrime vs. TwigStack We compare the performance between *TwigStackPrime* and *TwigStack*. Table 3 shows that *TwigStackPrime* always generates fewer root-to-leaf paths than *TwigStack*. This is because *TwigStack-Prime* uses CPL relationships to prune irrelevant elements. For instance, in TQ_6, *TwigStackPrime* produced only 22 565 useful paths, whereas the number of intermediate paths in *TwigStack* was 702 391. Although DBLP and XMark have relatively regular structures, *TwigStack* still produced irrelevant paths. For this type of datasets, *TwigStackPrime* shows optimal performance by generating only paths contributing in the final results. Since there is a difference in performance suggested by the Kruskal-Wallis test, we ran pairwise comparison based



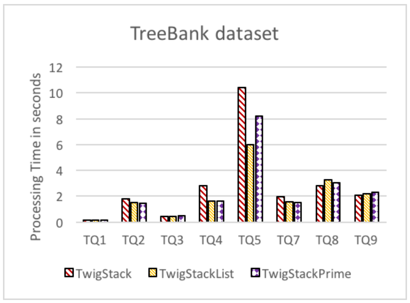
(a) DBLP



(b) XMark



(c) Random



(d) TreeBank

Fig. 5. Processing time for twig pattern queries against DBLP in 5a and XMark in 5b. 5c and 5d shows processing time for TPQs on Random and TreeBank datasets, respectively [2].

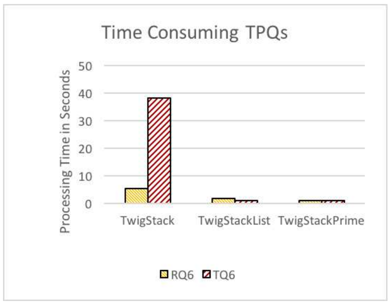


Fig. 6. The processing time taken by each algorithm to run the two most expensive queries in the experiments, normalizing query times to 1 for the fastest algorithm for each query [2].

Table 2. Experimental TPQs.

Code	XPath expression	Result
DQ_1	/dblp/inproceedings[//title]//author	88
DQ_2	//www[editor]/url	21
DQ_3	//article[//sup]//title//sub	278
DQ_4	//article[/sup]//title/sub	0
XQ_1	/site/closed_auctions/closed_auction[annotation/description/text/keyword]/date	4042
XQ_2	/site/closed_auctions/closed_auction//keyword	12527
XQ_3	/site/closed_auctions/closed_auction[//keyword]/date	12527
XQ_4	/site/people/person[profile[gender][age]]/name	3243
XQ_5	//item[location][//mailbox//mail//emph]/description//keyword	16956
XQ_6	//people/person[//address/zipcode]/profile/education	3241
TQ_1	//S[//MD]//ADJ	19
TQ_2	//S/VP/PP[/NP/VBN]/IN	152
TQ_3	//VP[/DT]//PRP_DOLLAR_	3
TQ_4	//S[/JJ]/NP	5
TQ_5	//S[VP[DT]//NN]/NP	32
TQ_6	//S[//VP/IN]//NP	20311
TQ_7	//S/VP/PP[//NP/VBN]/IN	320
TQ_8	//EMPTY/S//NP[/SBAR/WHNP/PP//NN]/_COMMA_	17
TQ_9	//SINV//NP[/PP//JJR][//S]/NN	4
RQ_1	//b//e//a[//f][d]	1331
RQ_2	//a//b[//e][c]	18033
RQ_3	//e//a[/b][c]	11216
RQ_4	//a[//b/d]//c	59568
RQ_5	//b[d/f]/c[e]/a	377
RQ_6	//c[//b][a]/f	47159
RQ_7	//a[c//e]/f[d]	1906
RQ_8	//d[a//e/f]/c[b]	204
RQ_9	//a[d][c][b][e]//f	3757

on Manny-Whitney test which showed that in most test twig queries *TwigStack-Prime* outperformed *TwigStack* as depicted in Fig. 5. In our experiments, we used TQ_6 and RQ_6 because they touch very large portions of their datasets and produce quite huge results. For TQ_6 and RQ_6, *TwigStackPrime* were more than 40 and 5 time faster than *TwigStack*, respectively.

TwigStackPrime vs. TwigStackList We now compare the performance between *TwigStackPrime* and *TwigStackList*. For highly structured datasets, both *TwigStackPrime* and *TwigStackList* are optimal as presented in Table 3. However, none of the algorithms are optimal in the other datasets because they have redundant paths and many tags are deeply recursive. In most queries, *TwigStackPrime* generated relatively fewer paths than *TwigStackList*. This is because *TwigStackPrime* uses CPL relationships to prune useless elements while *TwigStackList* utilises a simple buffering technique bounded by the number of elements in the longest path of the queried XML dataset. For example, RQ_9 where some of branching edges are P-C, *TwigStackPrime* can guarantee optimal evaluation because RQ_9 is its optimal class of query as mentioned in Theorem 2. *TwigStackPrime* produced 8786 useful paths whereas *TwigStackList* generated

17 328 useless paths. Even though RQ_4 is optimal for *TwigStackList* because it does not have P-C in branching axes, *TwigStackPrime* evaluated RQ_4 efficiently see Fig. 5 and Table 3.

Since there was a difference in performance, we ran pairwise comparison based on Manny-Whitney test which showed that in most twig queries tested *TwigStackPrime* outperformed *TwigStackList*, however, they showed same performance in XQ_2 , XQ_3 *and* XQ_6 see Fig. 5. For expensive queries, pairwise comparison based on Manny-Whitney test between *TwigStackPrime* and *TwigStackList* resulted in $p - value < 0.001$ which suggests a significant difference and *TwigStackPrime* has the best performance. When evaluating RQ_6, *TwigStackPrime* has the best performance, it is roughly twice as fast than *TwigStackList*.

Summary It can be seen in Fig. 5 the only twig queries where *TwigStackPrime* has slower performance comparing to the others is TQ_3 *and* TQ_9 because they touch very little of the dataset. According to the experimental results, we can draw the following two conclusions:

1. The CPL approach is a new source of improvement for holistic twig matching algorithms since it can reduce the number of elements processed and the size of intermediate result when TPQs contain Parent-Child edges.
2. *TwigStackPrime* significantly outperformed *TwigStack* and *TwigStackList* for different types of XML documents in terms of their structures including shallow and deep datasets. *TwigStackPrime* showed a superior performance in avoiding the storage of unnecessary paths while processing time is improved.

Table 3. Single paths produced by each algorithm [2].

Code	TwigStack	TwigStackList	TwigStackPrime
DQ_1	147	139	139
DQ_4	98	0	0
XQ_1	9414	6701	6701
TQ_2	2236	388	441
TQ_3	10663	11	5
TQ_4	70988	30	10
TQ_6	702391	22565	22565
TQ_8	58	27	26
TQ_9	29	17	8
RQ_1	2076	1843	1795
RQ_2	29914	24235	23057
RQ_3	20558	16102	15505
RQ_4	67005	57753	57753
RQ_5	3765	901	1093
RQ_6	201835	98600	72084
RQ_7	6880	2791	3219
RQ_8	746	322	406
RQ_9	179546	26114	8786

6 Conclusion and Future Work

In this paper, we proposed the CPL approach to improve the pre-filtering strategy in twig join algorithms when *P-C* edges are involved in TPQs. The key to the *TwigStackPrime* is the use of the CPL approach as the labelling scheme and of the advanced preorder filtering function *getNext*, which both enable fast determination of P-C relationships between elements of XML documents while scanning them in preorder traversal. This property is exploited to reduce storage space by skipping irrelevant elements from the streams and to improve the overall performance.

Compared to the previous labelling schemes, the CPL approach can be used to derive a set of the tag names of child elements associated with their inner elements. P-C edges, hence, can be solved in very efficient way. *TwigStackPrime* algorithm shows the general framework we use for introducing the CPL approach into existing twig matching algorithms, extending algorithm like *TwigStack*.

Existing research revolves around improving the efficiency of twig matching algorithms and extending querying algorithms to make them more able to handle positional predicates and order axes in XPath expressions. A promising approach for speeding up the query processing would be to combine our approach with the previous orthogonal algorithms to propose a new one-phase twig matching algorithm that we hope will be superior to the previous approaches. The current preliminary idea is to examine processing ordered twig patterns and positional predicate in a way that would consume less time and memory than the existing approaches. We will consider one-phase and ordered twig matching algorithms as our future work.

References

1. S Alireza Aghili, Li Hua-Gang, Divyakant Agrawal and Amr El Abbadi. TWIX: twig structure and content matching of selective queries using. *InfoScale '06: Proceedings of the 1st international conference on*, page 42, 2006.
2. Shtwai Alsubai and Siobhán North. A Prime Number Approach to Matching an XML Twig Pattern including Parent-Child Edges. In *The 13th International Conference on Web Information Systems and Technologies (WEBIST 2017)*, pages 204–211, Porto, 2017. SCITEPRESS Science and Technology Publications, Lda.
3. Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal XML pattern matching. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 310–321, Madison, Wisconsin, 2002. ACM.
4. Songting Chen, Hua-Gang Li, Junichi Tatemura, Wang-Pin Hsiung, Divyakant Agrawal, K Sel, #231, uk Candan, and K Selçuk Candan. Twig2Stack: bottom-up processing of generalized-tree-pattern queries over XML documents, 2006.
5. Ting Chen, Jiaheng Lu, and Tok Wang Ling. On Boosting Holism in XML Twig Pattern Matching Using Structural Indexing Techniques. *Science*, pages 455–466, 2005.
6. B Choi, M Mahoui, and D Wood. On the optimality of holistic algorithms for twig queries. *Database and Expert Systems Applications*, pages 28–37, 2003.

7. Nils Grimsmo, Truls Amundsen Bjørklund, and Magnus Lie Hetland. Fast optimal twig joins. *VLDB*, 3(1-2):894–905, sep 2010.
8. Jiang Li and Junhu Wang. Fast Matching of Twig Patterns. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5181 LNCS:523–536, 2008.
9. Jiaheng Lu, Ting Chen, and Tok Wang T.W. Ling. Efficient Processing of XML Twig Patterns with Parent Child Edges : A Look-ahead Approach. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management*, number i, pages 533–542, Washington, D.C., USA, 2004. ACM.
10. Jiaheng Lu, Xiaofeng Meng, and Tok Wang Ling. Indexing and querying XML using extended Dewey labeling scheme. *Data & Knowledge Engineering*, 70(1):35–59, 2011.
11. Lu Qin, Jeffrey Xu Yu, and Bolin Ding. TwigList: Make Twig Pattern Matching Fast. In Ramamohanarao Kotagiri, P Radha Krishna, Mukesh Mohania, and Ekawit Nantajeewarawat, editors, *Advances in Databases: Concepts, Systems and Applications: 12th International Conference on Database Systems for Advanced Applications, DASFAA 2007, Bangkok, Thailand, April 9-12, 2007. Proceedings*, pages 850–862. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.
12. Huayu Wu, Chunbin Lin, Tok Wang Ling, and Jiaheng Lu. Processing XML twig pattern query with wildcards. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7446 LNCS:326–341, 2012.
13. Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. On supporting containment queries in relational database management systems. *ACM SIGMOD Record*, 30:425–436, 2001.
14. S Al-Khalifa, H V Jagadish, N Koudas, J M Patel, D Srivastava, and Wu Yuqing. Structural joins: a primitive for efficient XML query pattern matching. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 141–152, 2002.
15. Haw, S.-C. and Lee, C.-S. (2011). Data storage practices and query processing in XML databases: A survey. *Knowledge-Based Systems*, 24(8):1317–1340.
16. Gang, G. and Chirkova, R. (2007). Efficiently Querying Large XML Data Repositories: A Survey. *Knowledge and Data Engineering, IEEE Transactions on*, 19(10):1381–1403.
17. Lu, J., Ling, T. W., Bao, Z., and Wang, C. (2011a). Extended XML tree pattern matching: Theories and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 23(3):402–416.
18. R. Goldman and J. Widom, Dataguides: Enabling query formulation and optimization in semistructured databases, Proc. Int. Conf. Very Large Data Bases, pp. 436445, 1997.
19. R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth, Covering indexes for branching path queries, Proc. 2002 ACM SIGMOD Int. Conf. Manag. data - SIGMOD 02, p. 133, 2002.
20. T. Chen, J. Lu, and T. W. Ling, On Boosting Holism in XML Twig Pattern Matching Using Structural Indexing Techniques, Science (80-.)., pp. 455466, 2005.
21. R. Bača, M. Krátký, T. W. Ling, and J. Lu, Optimal and efficient generalized twig pattern processing: a combination of preorder and postorder filterings, VLDB J., vol. 22, no. 3, pp. 369393, Oct. 2012.
22. R. Bača and M. Krátký, XML query processing, Proc. 16th Int. Database Eng. Appl. Sysmposium - IDEAS 12, pp. 813, 2012.

23. C. Mathis, T. Härder, K. Schmidt, and S. Bächle, XML indexing and storage: fulfilling the wish list, Comput. Sci. - Res. Dev., pp. 118, 2012.
24. A. Schmidt, F. Waas, M. Kersten, R. Busse, M. J. Carey, and G. B. Amsterdam, XMark : A Benchmark for XML Data Management, in VLDB 02 Proceedings of the 28th international conference on Very Large Data Bases, 2002, pp. 974985.
25. G. Miklau, UW XMLData Repository. [Online]. Available: http://www.cs.washington.edu/research/xmldatasets/. [Accessed: 04-Feb-2016].