IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received September 10, 2020, accepted October 12, 2020, date of publication October 22, 2020, date of current version November 2, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3033008

Solving the Intractable Problem: Optimal Performance for Worst Case Scenarios in XML Twig Pattern Matching

SHTWAI ALSUBAI[1] **AND SIOBHÁN NORTH**[2]
[1]Department of Computer Science, College of Computer Engineering and Sciences, Prince Sattam Bin Abdulaziz University, Al-Kharj 11942, Saudi Arabia
[2]Department of Computer Science, The University of Sheffield, Sheffield S1 4DP, U.K.

Corresponding author: Shtwai Alsubai (sa.alsubai@psau.edu.sa)

ABSTRACT In the history of databases, eXtensible Markup Language (XML) has been thought of as the standard format to store and exchange semi-structured data. With the advent of IoT, XML technologies can play an important role in addressing the issue of processing a massive amount of data generated from heterogeneous devices. As the number and complexity of such datasets increases there is a need for algorithms which are able to index and retrieve XML data efficiently even for complex queries. In this context *twig pattern matching*, finding all occurrences of a twig pattern query (TPQ), is a core operation in XML query processing. Until now holistic joins have been considered the state-of-the-art TPQ processing algorithms, but they fail to guarantee an optimal evaluation except at the expense of excessive storage costs which limit their scope in large datasets. In this article, we introduce a new approach which significantly outperforms earlier methods in terms of both the size of the intermediate storage and query running time. The approach presented here uses Child Prime Labels (Alsubai & North, 2018) to improve the filtering phase of bottom-up twig matching algorithms and a novel algorithm which avoids the use of stacks, thus improving TPQs processing efficiency. Several experiments were conducted on common benchmarks such as DBLP, XMark and TreeBank datasets to study the performance of the new approach. Multiple analyses on a range of twig pattern queries are presented to demonstrate the statistical significance of the improvements.

INDEX TERMS XML, holistic joins, XML databases, structural XML query processing.

I. INTRODUCTION

XML technology has emerged as the de facto standard for storage of semi-structure data and for data exchange in e-business [19]. *Twig Pattern Matching* (TPM) is a core operation in XPath [41] and XQuery [42] which are popular XML query languages. A Twig Pattern Query (TPQ) is an XML *path expression* which represents the basic building block of XML query languages. The existing literature on XML query processing is extensive and focuses particularly on the twig pattern matching problem because it is the hardest [6], [40]. A Twig Pattern Match is defined as mapping function M between a given tree pattern query Q and an XML document D, $M : Q \rightarrow D$ that maps nodes of the query Q into nodes of the document D preserving structural relationships and satisfying the predicates of Q. Formally, TPM has to find all matches of a given tree pattern query Q on an XML document D [1], [10], [16], [18], [21], [33], [40]. For a document D and a query Q with n nodes $(q_1, \ldots, q_n)$, a complete match is an n-dimensional tuple $(e_1, \ldots, e_n)$ which consists of the database elements that identify a distinct match of Q in D. An output match is a projection of a complete match such that the database elements corresponding to non-output query nodes are excluded [36]. The answer to Q on D is an ordered set of all the output matches of Q on D where the tuples are sorted in order of the common prefixes of the individual root-to-leaf paths.

The pioneering twig join algorithm, *TwigStack* was proposed in [10]. In the literature several twig join algorithms have been proposed to improve on this initial approach. In discussing them in this article, these algorithms have been grouped into the two main categories. On one hand, top-down twig join algorithms process TPQs by reading the nodes in pre-order traversal of the input document and checking child descendant extensions for internal query nodes. On the other hand, bottom-up algorithms store elements of

The associate editor coordinating the review of this manuscript and approving it for publication was Jenny Mahoney.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

the input document in post-order manner and inspect matching elements through virtual sub-trees. A major drawback of this second approach is its high memory consumption because all elements mapping to leaf query nodes reside in the main memory until the entire document has been completely processed but, in terms of processing time, it is faster than top down algorithms [6].

In the literature, top-down processing, which is based on *getNext()* [10], has been combined with bottom-up algorithms as a filter in order to reduce memory usage and thus improve the overall performance. The main weakness of using a top-down filter is that it does not provide an optimal evaluation for TPQs which include Parent-Child (P-C) relationships. This is a major bottleneck because such relationships are common in queries. The problem is a result of the restricted access mechanism (i.e., a single sequential scan of partitions of the input document) adopted in order to guarantee linear I/O Cost. Figure 1 depicts the simple partition technique used by XML query processing called a *tag streaming* scheme where each query node q is associated with a stream T_q consisting of all nodes with the same tag as q sorted in an order compatible with the depth-first traversal of the XML tree. Holistic algorithms can not guarantee that the head elements would form matches to TPQs comprising of P-C edges [6], [13], [20]. This dilemma (i.e., two head elements block each other with respect to P-C relationship) means that holistic algorithms must either output useless intermediate results or risk missing some potential answers to TPQs. The advantages of a top-down filter in bottom-up algorithms are that they speed up the sequential reading of the input streams but avoid storing elements which do not have ancestors likely to participate in the final solution. However, no top-down filter algorithm can remove leaf query nodes effectively when a mixture of P-C and A-D queries are processed.

The work of [3] proposed a new filtering strategy in twig joins which takes advantage of the the properties of prime numbers to avoid an additional pre-processing step. The Child Prime Label (CPL) algorithm is an extension of the *getNext()* core function in the classical holistic twig joins algorithm, TwigStack [10]. This new filtering function can filter out irrelevant elements efficiently without either violating the *document order* or consuming additional space. A new top-down holistic algorithm TwigStackPrime was presented, which reduced memory consumption and the computation overhead of twig pattern matching when P-C edges are involved. In particular, holistic algorithms using the CPL indexing technique were shown to be I/O and CPU optimal when a TPQ has only A-D edges or where there are P-C edges to connect leaf query nodes. This analysis was confirmed by experimental results on a wide range of real-world, benchmark and artificial datasets.

Over the past decade, two holistic joins, proposed in [7], [20], have been considered as the best top-down and bottom-up combinations. The authors of [20] proposed a new preorder filtering function called *getPart()* which introduced two improvements to the original *getNext()* function. Compared

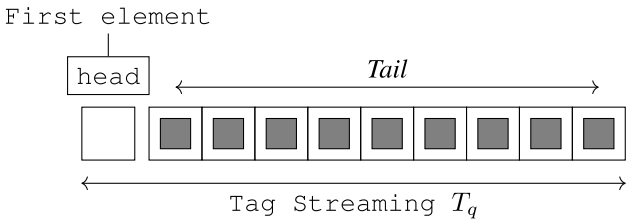


FIGURE 1. Tag streaming model of a query node q [50].

with the existing preorder filtering function, *getPart()* returns child query nodes if and only if they have a relevant ancestor processed by previous calls of *getPart()* and stored in intermediate storage. The second improvement is to skip irrelevant elements depending on the current query node's descendants and ancestors, in contrast to the *getNext()* which is only able to skip useless elements based on the current query node's descendants. Other authors [7] further improved the *getPart()* function by avoiding unnecessary function calls. The new advanced preorder filtering function is called *getMatch()*. The *getPart()* function serves as the advanced preorder filtering strategy for a family of twig matching algorithms devised in [20]. The alternative is the GTPStack algorithm [7] which uses the *getMatch()* function.

However, two issues must be taken into account when a combination of preorder and postorder filtering is considered [7], [20], [29]:

1) The filtering algorithm must return each element of a query node q in document order. This is important because when visiting document elements in postorder (i.e, reversed order) it is possible to determine whether or not e satisfies the twig pattern query directly without further investigation because all its descendants have been visited.
2) Elements must be pushed into the intermediate storage in preorder since the answer to TPQ Q with n nodes can be represented as an n-ary relation where each *tuple* $(e_1, \ldots, e_n)$ consists of the database elements that identify a distinct match of Q in D. Some fields may be duplicated and some may not be in the document order, but tuples have to be sorted in order of the common prefixes of the individual root-to-leaf paths. As a result, if we push elements into the intermediate storage in postorder, the enumeration outputs the resulting tuples unordered.

In this article, we explain how the improvement of the recent top-down algorithm, TwigStackPrime [3] can be transformed into new state-of-the-art bottom-up algorithm by exploiting ideas proposed in [3], [7], [20], [29], [50] and extend the original holistic join approach to solve these issues. We also show that it is not necessary to maintain a set of stacks to adopt the level split vectors intermediate storage when preorder storing is required. As a result, we can prove that for a certain class of TPQs our bottom-up approaches have linear time complexity with respect to the size of the

input and output and the linear space complexity with respect to the longest path of the XML tree. To the best of our knowledge, this is the first work which guarantees optimal worst case evaluation for bottom-up holistic joins without performing pre-processing (i.e., stream pruning). The main contributions of this article are summarized as follows:

- Introducing the CPL approach to the advanced preorder filtering functions used by the TJStrictPre and GTPStack algorithms, namely *getPart()* and *getMatch()*, respectively.
- A set of novel bottom-up holistic twig matching algorithms which are based on a new advanced preorder filtering function which has the ability to preserve *the document order*, unlike previous filtering strategies, such as [30], [32], and filter out irrelevant elements when P-C relationships are invloved in TPQs. Full proofs of correctness for the algorithms necessary to evaluate subsets of TPQs containing P-C and A-D axes are provided as well.
- Lastly, the paper provides an empirical proof of improvements of the holistic algorithms proposed, based on the CPL approach, over other related methods from the literature.

The paper is organized as follows. The following section, Section II, reviews the related work. In Section III, we give an overview of models and techniques used in this article. Section IV introduces the TwigPrime algorithm and its optimisations while Section V experimentally verifies and validates the advantages of the new approaches. Finally, the paper is concluded in Section VI.

II. RELATED WORK

A. NODE LABELLING SCHEME

Most existing XML query processing algorithms [5], [7], [10], [20], [25], [29], [32], [43] rely on XML indexing techniques to access only XML data relevant to the XML query. In XML, there are two basic types of index. The first indexes each node in an XML document by recording its position. This group are known as node labels or labelling schemes. Node indexing approaches index each node in an XML document by assigning an unique label (based on a labelling scheme) to every node. This label encodes its positional information within the XML tree. The values of labels are a reflection of the chosen labelling scheme. This group of indices uses nodes as the basic unit of a query which provides the opportunity to perform structural queries very efficiently by exploiting information encoded in the labels. According to [37], a labelling scheme has to guarantee uniqueness and order preservation of node labels, and ensure that the hierarchical relationships between a pair of data nodes can be determined directly from the labels. To better explain the mechanisms of node indexing methods and their properties, [22] classified node indexing into four distinct types; Sub-tree labelling, Prefix-based labelling, Multiplicative labelling and Hybrid labelling. A full discussion of the different categories lies beyond the scope of this article but, all XML query processing algorithms which perform structural join operations to match a given query against an XML document rely on either sub-tree labelling schemes or prefix-based labelling schemes. A well-known example of sub-tree labelling is the regional labelling scheme proposed in [49]. In this approach, each node is assigned with a 3-tuple as $<start, end, level>$. *Start* and *end* contain values of positions corresponding to the opening tag $<tag>$ and the closing tag $</tag>$ of the subtree and *level* represents the depth of the node within the XML tree. The two basic relationships Ancestor-Descendant (A-D) and Parent-Child (P-C) can be determined easily from this. Given two nodes u and v, u is an ancestor of v if and only if $u.start < v.start < v.end < u.end$. Furthermore, a P-C relationship is defined as node u is the parent of node v if and only if $u.start < v.star < v.end < u.end$, $v.level = u.level + 1$. An example of the regional labelling scheme can be found in Figure 5. A classic example of multiplicative labelling is a prime number labelling scheme. This was proposed to support labelling dynamic XML documents. In a prime number labelling scheme [46], every node is given a unique prime number called the self-label. Then the label for each node is the product of its self-label and its parent-label. This labelling scheme completely avoids re-labelling when a new data node is inserted, only the simultaneous congruence value to determine the document order needs to be recalculated.

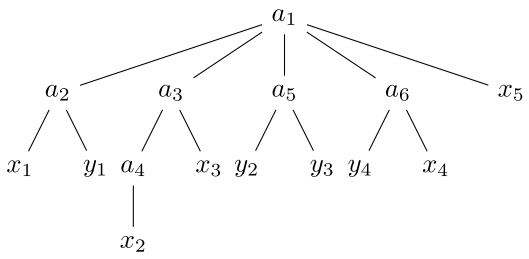
B. TWIG QUERIES

The holistic join was introduced by [10] as a new approach to evaluate query twig patterns efficiently. The work was an extension to the sophisticated PathStack algorithm which decomposed queries into a set of binary structural relationships. They proposed the decomposition of twigs into a set of root-to-leaf paths and evaluated each root-to-leaf path using the PathStack algorithm. The final results are produced by a merge join operation on the intermediate results. The algorithm is called TwigStack and has shown a significant performance improvement in reducing intermediate results in comparison to the binary structural join algorithms. The TwigStack algorithm only guarantees an optimal evaluation of twig queries with A-D relationships connecting all query nodes. The optimal evaluation in an holistic approach means every query node pushed into the encoding data structure (in the case of TwigStack is a chain of stacks) must be part of the final result. This is established by scanning them sequentially [15]. However, TwigStack's performance suffers from its generation of useless intermediate results when twig queries encounter P-C relationships. TwigStack performs twig evaluation in two phases: the first phase is to decompose a twig pattern query into single root-to-leaf paths and the second is to match them against XML data. The second phase is the merge phase in which all matching results produced by the first phase are merged to compute the final query results. Obviously, the second phase is an expensive process since an n-way merge has to be performed where n is the number

of single paths in the twig query. TwigStack is considered the keystone for algorithms in this family and many research papers have suggested improvements to it [2], [4], [11]–[13], [20], [23], [24], [26], [31]–[35], [38], [43], [44], [47], [48]. Nevertheless, an optimal evaluation of a tree pattern query with any arbitrary combination of A-D and P-C relationships has been proven to be impossible by [15] for the TwigStack algorithm and its variants. In order to speed up the query processing and avoid reading useless data nodes, the authors of TwigStack [10] proposed an XB-tree index, which is a variant of B-tree index, to reduce the disk-read costs of TwigStack by skipping over input streams corresponding to inner query nodes which do not satisfy A-D relationships with child query nodes.

In [13], the authors reviewed the sub-optimality of the existing clustering technique used in TwigStack where an XML document is clustered into tag streams which group together elements with the same tag name. They proposed two different novel streaming schemes, namely: *prefix path* and *tag + level* streaming schemes. A *tag + level* streaming scheme clusters all the elements which have the same tag and are located at the same level. A prefix-path streaming scheme, or PPS for short is an ordered set of elements which have the same prefix path. Figure 2 presents different streaming schemes over a given XML tree, elements are grouped based on similarity in tag names and level in case of *tag + level* streaming scheme or unique path in case of *prefix path* streaming scheme. For the sake of simplicity, the number associated with each tag indicates the level using *tag + level* streaming scheme as in Figure 2c and the path associated with each tag indicates the unique path using *prefix path* streaming scheme as in Figure 2d. For example, a^1 streaming list contains all elements with a-tagged node and appear at level 1. Based on the new streaming schemes, they proposed an extension to TwigStack called iTwigJoin. Their algorithm is optimal for queries with A-D edges only when tag streaming schemes (i.e., label lists) are applied. The use of *tag + level* streaming scheme in iTwigJoin guarantees the optimality in two classes of queries: A-D or P-C edges only. In addition, the iTwigJoin depends on *prefix path* (i.e, iTwigJoin + PPS) and is optimal in three classes of queries: A-D, P-C edges or one branching query node only. It has been proven in [28] that the efficiency of iTwigJoin reduces when the number of streams for every query node is increased.

TwigStack and its variants described above all work by decomposing twig queries into individual root-to-leaf paths and processing the queries top-down to filter out irrelevant nodes which may match query nodes' tags but do not satisfy its structural constraints. Top-down filtering can be seen as prefix path matching where a sequence of steps in an XPath expression connects descendants to their ancestors. For example, consider a given query which consists of k query nodes as $q_1/q_2/\ldots/q_k$. If a document element e corresponding to q_2 in the mapping function $q_2 \rightarrow e$ such that e is satisfied and if and only if it has an ancestor element corresponding to q_1 which also satisfies the mapping function and so on



(a) an XML tree

a:$\{a_1, a_2, a_3, a_4, a_5, a_6\}$
x:$\{x_1, x_2, x_3, x_4, x_5\}$
y:$\{y_1, y_2, y_3, y_4\}$

(b) tag streams

a^0:$\{a_1\}$
a^1:$\{a_2, a_3, a_5, a_6\}$
a^2:$\{a_4\}$
x^1:$\{x_5\}$
x^2:$\{x_1, x_3, x_4\}$
x^3:$\{x_2\}$
y^2:$\{y_1, y_2, y_3, y_4\}$

(c) tag+level streams

/a:$\{a_1\}$
/a/a:$\{a_2, a_3, a_5, a_6\}$
/a/a/a:$\{a_4\}$
/a/x:$\{x_5\}$
/a/a/x:$\{x_1, x_3, x_4\}$
/a/a/a/x:$\{x_2\}$
/a/a/y:$\{y_1, y_2, y_3, y_4\}$

(d) prefix path streams

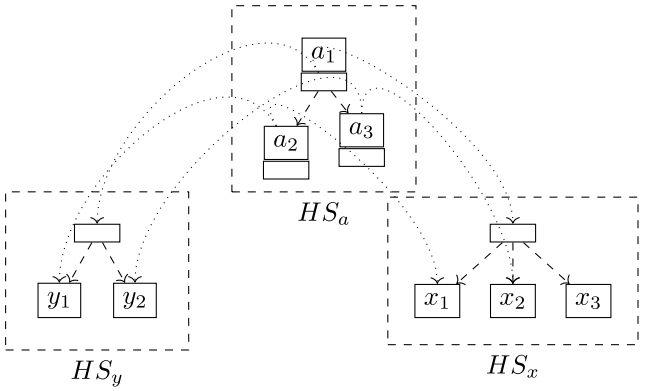
FIGURE 2. Illustration of different data partitioning schemes.

to the leaf query node q_k, then each element in the entire path will be pushed into their corresponding stacks in the intermediate storage. This means that the top-down process checks document elements in pre-order and stores them in post-order.

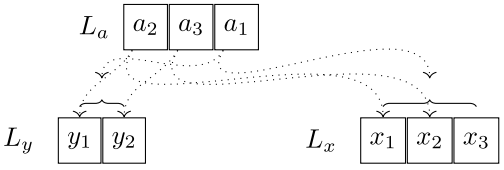
The alternative approach of examining XML queries against document elements in post-order was first introduced by [12]. In this article the authors prove that decomposition of twigs into a set of single paths and enumeration of these paths is not necessary to process twig pattern queries. The key idea of their approach is based on the proposition that when visiting document elements in post-order (i.e, reversed order) any element e the determination whether or not e satisfies the twig query sub-rooted at e is implicit because all its descendants have been visited. They proposed a new algorithm to process twig queries without merge joining single paths using a new encoding to store twig results in main memory. Their representation is a tree of stacks in which every query node n is associated with a hierarchical stack HS_n which consists of an ordered sequence of stack trees. Pointers are heavily used to capture the basic relationships between elements in different hierarchical stacks as shown in Figure 3. The researchers developed a new algorithm called Twig^{2}Stack to evaluate a wider range of XML queries than TwigStack and its

variations including generalized twig pattern (GTP) queries which are a fundamental building block for XQuery processing. GTP queries contain both mandatory (corresponding to the FOR or WHERE clauses) and optional (corresponding to LET clause) relationships. Twig^{2}Stack produces its solution using an enumeration function over the pointers in the hierarchical stacks. In the same context, a new algorithm was proposed in [26], called HolisticTwigStack. This algorithm combined the filtering strategy of TwigStack to Twig^{2}Stack in order to reduce memory consumption. The major limitations with both algorithms is the complex stack structure. Although they both reduce the cost of query execution by eliminating the merge phase (second phase in TwigStack), the pointers in both algorithms, especially Twig^{2}Stack, are complex and expensive to maintain. In the worst case the entire document needs to be loaded into the main memory. To overcome these drawbacks, a new algorithm called TwigList was proposed in [38]. TwigList replaced hierarchical stacks with a list for every node and used simple intervals to capture structural relationships. TwigList used a global stack to read the whole document in pre-order and adds data nodes to the corresponding lists in post-order manner if they satisfy the mapping function conditions. To demonstrate the difference between the two algorithms consider Figure 3b; element a_1 in L_a has two intervals specified by four pointers in two 2-tuples, namely $<start_y, end_y>$ and $<start_x, end_x>$, $start_y$ records first element matches a_1 as one of its descendants with y-tagged node while end_y records last element is one of a_1's descendants with a y-tagged node. a_1 has $<1, 2>$ as its recorded interval for contained elements corresponding to query node y. Reference [29] extended TwigList by combining the features of two-phased holistic algorithms and one-phased algorithms, namely TwigStack and TwigList. They improved TwigList by applying the filtering strategy applied in TwigStack to select useful elements before pushing them into the TwigList stack. To do this they proposed two novel algorithms, called TwigMix and TwigFast to improve the efficiency of TwigList. When twig pattern queries contain only A-D edges both algorithms guarantee all elements in intermediate results contribute to the final results. In their experiments TwigMix and TwigFast significantly outperformed TwigList.

The authors in [20] proposed a new storage scheme, called a level split approach which splits the intermediate list connected to its parent list into levels equal to the depth of the XML tree as shown in Figure 4 for the same XML tree and a given query as in Figure 3. In their paper, a combination of preorder and postorder filtering methods is adopted to develop two algorithms, namely: TJStrictPre and TJStrictPost. Their experimental results demonstrated the ability of the new method to eliminate useless elements in inner lists, and so the number of intermediate results is far smaller than in TwigList and TwigFast. These approaches can guarantee linear CPU and I/O complexity of the output enumeration relative to the output size. However, they suffer from large intermediate results in comparison to the query output. In [7],



(a) Twig^{2}Stack's trees of stacks



(b) TwigList's simple vectors

FIGURE 3. The intermediate storage of Twig^{2}Stack and TwigList when processing a TPQ Q_1 = a[//y]/x over the XML tree T_1 in Figure 5.

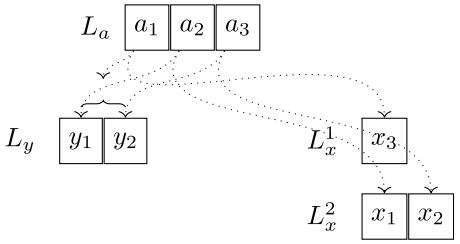


FIGURE 4. Illustration of level split list approach introduced in [20]. L_x^2 stores elements which have level values equal to 2 in the XML tree.

the authors improved the filtering strategy proposed in [20] by eliminating unnecessary self-nested matching checks (i.e., recursive calls) similar to the approach introduced in [28]. Table 1 provides the breakdown of twig matching algorithms according to filtering strategies, intermediate storage and optimal query types.

III. PRELIMINARIES

A. DOCUMENT AND QUERY MODEL

In XML, both data and queries are represented using a tree-structured model. An XML Tree is a rooted, node-labelled tree as $T = (V, E, r, \sum_V, \mu)$ where $V = \{v_1, \ldots, v_n\}$ is a finite set of nodes. $E = \{(u, v) \in V \times V\}$ is a set of edges. $r \in V$ is a distinguished node called the root. $\sum_V$ is the set of element names appearing in T. $\mu : V - \{r\} \rightarrow \sum_V$ is a labelling function which associates an element name with each node other than the root. The level of any node in T is the number of distinct element(s) in the unique path between it and the root thus level(r) = 0.

Definition 1 (Child Relationship): Given two nodes *u and v* in a rooted, labelled tree where $u, v \in V$, v is a child of u if and only if $\exists e \in E : e = (u, v)$. This relationship is denoted as PC or P-C and if v is a child of u then u is a parent of v.

TABLE 1. Previous algorithms and their filtering properties. A simple vector is the default for algorithms adopting element references to store intermediate results.

Algorithm	Ref.	Filtering		Intermediate Results		Optimal
		preorder	postorder	paths	element references	
TwigStack	[10]	✓		✓		A-D edges
iTwigJoin+TL	[13]	✓		✓		P-C edges
iTwigJoin+PPL	[13]	✓		✓		one branching query node or all P-C edges before A-D edges
$Twig^2Stack$	[12]		✓		✓	optimal enumeration
TwigList	[38]		✓		✓	no
TwigFast	[29]	✓			✓	A-D edges
TJStrictPost	[20]	✓	✓		✓	A-D edges
TJStrictPost+ Split Vector	[20]	✓	✓		✓	optimal enumeration
TJStrictPre	[20]	✓			✓	A-D edges
TJStrictPre+ Split Vector	[20]	✓	✓		✓	A-D edges + optimal enumeration
GTPStack+ Split Vector	[7]	✓	✓		✓	A-D edges + optimal enumeration
TwigStackPrime	[3]	✓		✓		P-C edges connected to the leaf query nodes
TwigPrime	Section IV	✓	✓		✓	P-C edges connected to the leaf query nodes + optimal enumeration

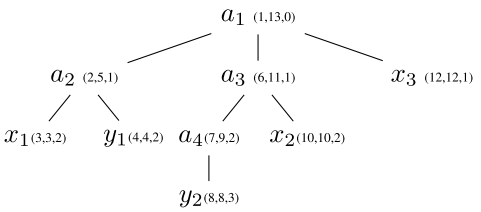
Definition 2 (Descendant Relationship): Given two nodes u and v in a rooted, labelled tree where $u, v \in V$, v is a descendant of u if and only if $\exists n_1, \ldots, n_k \in V$ such that $(u, n_1) \in E$, $(n_1, n_2) \in E, \ldots, (n_k, v) \in E$ where $1 \leq k <$ *the depth of the tree*. If v is a descendant of u then u is an ancestor of v. This relationship is denoted as AD or A-D.

Formally, Twig Pattern Query (TPQ) is also a rooted, node-labelled tree $TPQ = (V, E, r, \sum_V, \mu)$ where $V = \{v_1, \ldots, v_n\}$ is a finite set of query nodes. $E = \{(u, v) \in V \times V\}$ is a set of edges which represents parent-child or ancestor-descendant relationships between connected query nodes. The set of child edges is denoted by $E_/$, while the set of descendant edges is denoted by $E_{//}$. $r \in V$ is a distinguished query node called the root. $\sum_V$ is the set of element names appearing in TPQ. $\mu : V \rightarrow \sum_V$ is a labelling function which associates an element name with each node. The difference between an XML tree and a twig pattern is in the type of their edges, an XML tree can only have parent-child edges connecting its nodes, while the twig pattern is an extension that can handle the Ancestor-Descendant structural relationships as edges connecting its nodes. In practice, a twig pattern is much smaller than the original XML tree. It can be seen as a representation of a user query although translating an XML query plan into a twig pattern is not a simple task [21]. Complex XML queries are divided into several twig patterns because a single twig pattern can represent only a single XPath path expression. The complexity of XML queries determine the difficulty of translating them into twig pattern(s). In XML query optimization, the process of translating user queries to twig patterns and then optimising them has been the subject of considerable research [51].

B. CHILD PRIME LABELS

In holistic twig joins, head elements pointed to by cursors of streams are classified to three types with respect to a twig pattern query Q with n nodes:

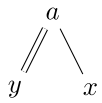
- Matching element where e_n has a minimal extension to q_n.



(a) an XML tree T_1

$for\ \$a\ in\ doc("T_1")//a$
$for\ \$y\ in\ \$a//y$
$for\ \$x\ in\ \$a/x$
$return\ <r>\$a, \$y, \$x, </r>$

(b) an XQuery query

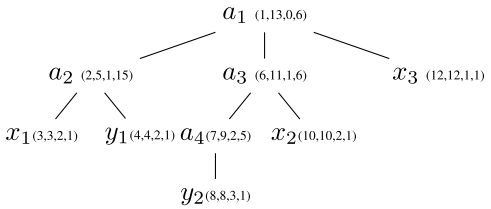


(c) a TPQ Q_1

FIGURE 5. A sample XML tree, XQuery query and the corresponding TPQ.

- Useless element where e_n can not participate in a match to Q with the current or future elements.
- Blocked element where e_n is a neither a matching or useless element.

Recently, a new indexing technique which can be used in conjunction with existing labelling schemes and minimises the number of blocked nodes during the processing of TPQs with P-C edges was proposed in [3]. The key idea of this work is that it can be used in addition to the triplet of range-based labelling scheme to prevent elements becoming blocked. The name of the new approach, *Child Prime Labels (for short CPL)*, arises from the exploitation of child relationships in XML trees and the property of prime numbers. All the distinct tags in the XML tree are identified and assigned unique prime numbers. Then, the intuition of the CPL is to use the modulo function to test whether or not an element has an element with a particular tag name among its children. The leaf elements will be annotated with 1 as their CPLs, while the inner elements (i.e., parent elements) are assigned CPLs by multiplying the prime numbers of the distinct tags its of child elements. The immediate child elements of inner elements



(a) an XML tree T_1

Tag	Key
a	2
x	3
y	5

(b) tag index(T_1).

FIGURE 6. An XML tree labelled with range-based augmented with CPL and the corresponding tag indexing.

can be derived from their labels so that the process of handling P-C relationship among head elements in the streams can be resolved by computation.

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 $P = \{p_1, p_2, \ldots, p_m\}$. The bijective mapping function $f : C \rightarrow P$ 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:

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

The unique prime number associated with q is obtained from *tag indexing* where a lookup table to find unique prime numbers associated with distinct tags within a given XML document during query processing. As in the original work of [3], the original range-based labelling scheme is extended to incorporate the CPL information so that each range-based label with CPL is a quadruple =*(start, end, level, CPL)*. The first three attributes are those of the original labelling scheme, see Figure 5 and the last is the CPL. For illustration, consider the XML tree in Figure 6. The tag names are assigned prime numbers as they are identified when scanning the tree in depth-first traversal. Prime numbers are assigned as follows: $a \rightarrow 2$, $x \rightarrow 3$, $y \rightarrow 5$. The element a_2 has a CPL value equals to 15 as $CPL(a_2) = f(x) \times f(y) = 3 \times 5 = 15$ while the element a_3 has a CPL with value of 6 as $CPL(a_3) = f(a) \times f(x) = 2 \times 3 = 6$.

The most important advantage of the CPL approach over other related filtering strategies is that it has the ability to filter out elements without consuming extra storage or distorting the node processing order which is a problem of the TwigStackList algorithm [32].

C. NOTATION

Throughout this article, the term *element* is a reference to a data node in an XML tree and *node* refers to a query node in a twig pattern. It is also useful to name some auxiliary operations on TPQ and its nodes used in the twig matching process. These operations are:

- *children(q)* which returns all child nodes of q.
- *subtree(q)* returns all nodes of 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 true if q is the root and false otherwise.
- *isLeaf(q)* returns true if q is a leaf node and false otherwise.
- *getRoot(TPQ)* returns the root of the query.
- *parent(q)* returns the parent query node of q.
- *getVector(q,Integer level)* returns the regular intermediate result list if q is below an A-D edge or a split list by level if q is below a P-C axis.

Streams are implemented by a retrieval mechanism similar to inverted lists in the field of Information retrieval [9]. Every stream T_q in TPQ is equipped with a cursor, C_q, which initially points to the first element in T_q. As shown in Figure 1, the stream of query node q has two parts: *head* which is pointed by C_q and the remaining elements referred to as the *tail*. As with earlier approaches the streams end with a virtual end element labelled with infinity values as (∞, ∞, ∞).

The following operations are defined over every cursor of a stream in TPQ.

- *getStart(C_q)* returns the start attribute of the head element for query node q.
- *getEnd(C_q)* returns the final attribute of the head element corresponding to query node q.
- *getLevel(C_q)* returns the level attribute of the head element for query node q.
- *advance(C_q)* moves the cursor of q forward by one position to point to the next element.
- *eof* (T_q) returns true if C_q points to the end of stream for T_q and false otherwise.

IV. OPTIMAL TWIG JOINS

This section introduces a new bottom-up holistic twig matching algorithm which combines the advantages of the previous approaches [3], [7], [20], [29]. The new algorithm is called TwigPrime and is based on a total rewrite of TwigFast [29]. In other words, the algorithm combines the efficient selection of useful elements for TPQs with both P-C and A-D edges introduced in [3] and uses a level split data structure as the primary intermediate storage. It contains a further improvement on TwigFast by strictly checking prefix path matching for P-C relationships before storing the intermediate results. It includes an extension to the state-of-the-art filtering strategies *getPart()* and *getMatch()* which can apply the CPL approach in order to explore the potential benefit of the CPL approach in a contemporary one phased holistic algorithms.

A. TwigPrime

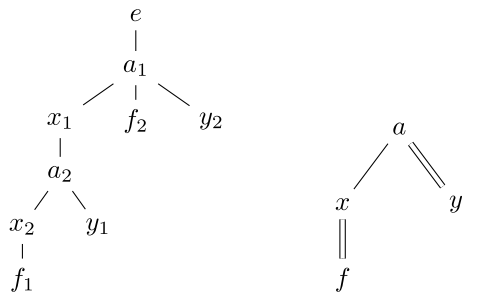
The TwigPrime approach can be seen as a new alternative to the TwigFast algorithm. It differs from the original TwigFast in that it adopts the advanced preorder function *getNext()* which is based on the CPL approach and the use of a level split data structure to store the intermediate results. The use of pointers in TwigPrime and its refined versions is similar to that in [29].

The structure of the main algorithm, TwigPrime presented in Algorithm 1 is more complex than the original TwigFast algorithm. In [29], there is one list containing matches for each query node, the list is sorted in preorder. Each element in the list records intervals for each child query node. Interval start values are recorded as elements appended to intermediate lists while interval end positions are recorded only when the elements can not be part of any other match. In order to construct intervals whilst avoiding the use of stacks, each element appended to the list has a pointer to the closest ancestor in that list. Each list also has a tail pointer which indicates the candidate parent query node. An advantage of this approach is that there is no overhead from maintaining a set of stacks. However, it does not perform prefix path filtering checks so that elements can be added without having relevant parents.

TwigFast can not be adapted to a level split data structure directly because there are specific families of nodes that have to be treated exceptionally. To illustrate the difficulty Figure 8 shows the intermediate results after running TwigFast to process Q_1 against T_2 in Figure 7. When f_2 is returned, the algorithm will record the interval end positions for x_1 and x_2 since the tail for x-node points to x_2 and x_2 has an ancestor pointer to x_1 in the same list. When a level split approach is used, if x_2 is only pointed by the tail pointer, then a match including x_1 could be missed. Thus, to ensure the descendant intervals are set correctly, there must be a tail for each level and each tail must be checked separately introducing a new form of filtering.

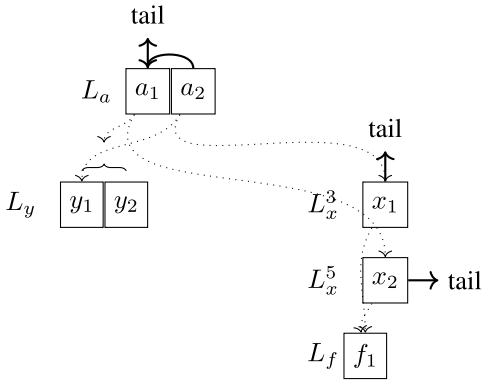
The level split tail filtering described by Algorithm 2 only happens in one situation, but a relatively common one. It is necessary if the incoming element for a query node q_n has an A-D relationship to the parent query node q_p that has, in turn, a P-C relationship with the parent query node, the tail for every level split list corresponding to the query node q_p must be checked to record the end positions correctly. This definition can be formalized as in Definition 3. It should be also noted that using a pointer to the closest ancestor in the same list is unnecessary when an element has a P-C relationship to the parent as they are stored in different lists. Henceforth, a tail pointer is sufficient to track potential parents or ancestors for query nodes under P-C edges.

Definition 3 (AD Follows PC): Given a query node p, which is connected with P-C edge to its parent, and its A-D child c, suppose n separate level split lists of p has been visited. In intermediate lists of p, all elements which are pointed by n tails will be checked. The tail elements that are not ancestors of the current element e_c will be assigned their end interval values.

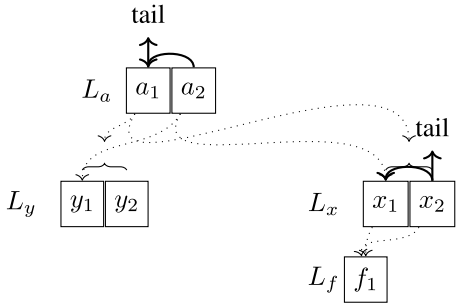


(a) an XML tree T_2 (b) Q_1

FIGURE 7. An example to illustrate tail pointers for level split data structure.



(a) Nested tail checks used for level split approach



(b) Straightforward tail checks used in TwigFast [29]

FIGURE 8. Intervals for intermediate storage handling approaches after processing f_1.

Definition 4 (Strictly Matching): A query node in a TPQ Q as $q_n \in Q$ with an edge $e = (q_n, q_v) \in E$ is a strict match of an XML element $M(q_n) \in D$ if and only if $M(q_n)$ and $M(q_v)$ are related as specified by e.

Definition 5 (Weakly Matching): A query node in a TPQ Q as $q_n \in Q$ with an edge $e = (q_n, q_v) \in E$ is a weak match of an XML element $M(q_n) \in D$ if $M(q_n)$ is an ancestor of $M(q_v)$.

Definition 6 (Prefix-Path Matching): A query node in a TPQ Q as $q_n \in Q$ is a strict/weak prefix-path match of an XML element $M(q_n) \in D$ if and only if the simple path $q_1, \ldots, q_n$ is a strict/weak match of q_n, where q_1 is the root query node.

Algorithm 1 TwigPrime

Input: A TPQ Q with n nodes $(q_1, \ldots, q_n)$
Output: All n-dimensional tuples $(e_1, \ldots, e_n)$ as answers for Q

// initialise $L_{n_i} = \emptyset$ for each $n_i \in TPQ\ Q$ if n_i is root or $n_i \in childrenAD(parent(n_i))$ and $n_i.tail = -1$
// initialise an array of $L[]_{n_i} = \emptyset$ for each $n_i \in TPQ\ Q$ if $n_i \in childrenPC(parent(n_i))$
while $\neg end(getRoot(Q))$ **do**
  q_{act} = getNext(getRoot(Q)) or getPart(getRoot(Q)) or getMatch(getRoot(Q)) // using CLP filtering Introduced in [3]
  v_{act} = getElement(q_{act})
  if $\neg$ *isRoot*(q_{act}) **then**
    setEndPointerParent(q_{act},parent(q_{act})) // see Algorithm 2
  end
  if *isRoot*(q_{act}) $\vee$ *getParentTail*(q_{act}) $\neq -1$ **then**
    if $q_{act} \in$ *childrenPC*(*parent*(q_{act})) **then**
      h = level(v_{act}) - 1 // parent should be stored one level higher
      v_p = getVectorElement(parent(q_{act}),h)
      if $\neg$ *PCrealtionship*(v_p,v_{act}) **then**
        // here to perform strict prefix path filtering which TwigFast [29] misses
        advance(q_{act})
        continue // skip the following lines and moves to the next cycle
      end
    end
    if $\neg$ *isLeaf*(v_{act}) **then**
      // set the end values for all elements in $L_{q_{act}}$ which are not ancestor of v_{act}
      // $\forall\ n_i \in$ children(q_{act}) $v_{act}.start_{n_i}$ = length(getVector(n_i))
      // $v_{act}.ancestor$ = getTail(q_{act}) // pointer to the closest ancestor or -1 if it does not have one
      // setTail(q_{act}) to length(getVector(q_{act})) // this to set the tail pointing to v_{act} as the open element for this list
    end
    // append v_{act} to the corresponding list
  end
  advance(q_{act})
end
// Process remaining open elements using an imaginary "end" element whose start and end are all ∞
// Clean intermediate results with postorder checks
// Enumerate results

Definition 7 (Subtree Matching): A node in a TPQ Q as $q_n \in Q$ is a strict/weak subtree match of an XML element $M(q_n) \in D$ if and only if all query nodes which are

Algorithm 2 Level Split Tail Filtering

Function `isADfollowsPC` (*Query node q*) **:**
  p = parent(q)
  if $q \in ChildrenAD(p)$ **then**
    if $p \in ChildrenPC(parent(p))$ **then**
      return: true
    end
  end
  return: false
Function `getTail` (*Query node q, Integer h*) **:**
  if *isRoot*(*q*) $\vee q \in$ *childrenAD*(*parent*(*q*)) **then**
    return: q.tail
  else
    return: q[h].tail
  end
Function `getParentTail` (*Query node q,Query node p*) **:**
  h = level(getElement(q))
  if $\neg$ *isADfollowsPC*(*q*) **then**
    return: getTail(p, h-1)
  else
    $\forall$ level $\in$ used level **if** *getTail*(*p,level*)$\neq -1$) **then**
      return: getTail(p,level)
    end
    return: -1
  end
Procedure `setEndPointerParent` (*Query node q, Query node p*) **:**
  if *isADfollowsPC*(*q*) **then**
    $\forall$ level $\in$ used level of L_p
    if *getTail*(*p,level*) $\neq -1$) **then**
      v_{act} = getVectorElement(p,level)
      if *getEnd*(v_{act}) $<$ *getStart*(*getElement*(*q*)) **then**
        markEnd(v_{act}) // set the end valuse for ecah $n_i \in children(p)$
        // $v_{act}.end_{n_i}$ = length(getVector(n_i)) -1
      end
    end
  else
    while *getParentTail*(*q,p*) $\neq -1$ **do**
      v_{act} = getVectorElement(p,0)
      if *getEnd*(v_{act}) $<$ *getStart*(*getElement*(*q*)) **then**
        markEnd(v_{act}) // set the end values for each $n_i \in children(p)$
        setTail(v_{act}) // set tail for the particular query node.
      else
        break
      end
    end
  end
Function `getVectorElement` (*Query node q, Integer level*) **:**
  // return the current element pointed by the tail of the regular intermediate result list if q is below an A-D edge or split list given by level if q is below a P-C axis.
Procedure `setTail` (*Query node q,Integer level*) **:**
  // set the tail to point to the closest ancestor of the current tail if any exists, otherwise -1. if q is below an A-D edge q.tail or q[level].tail given by level if q is below a P-C axis.

child or descendant nodes of q_n are in a strict/weak prefix-path match of the simple paths starting from q_n as the root to each one of its children and descendants in Q.

The above definitions are fundamental to the new approach. The main algorithm of TwigPrime, Algorithm 1 describes the general framework for constructing intermediate results in preorder sequence, thus extending TwigFast [29]. It supports any combination of preorder and postorder filtering and either simple or level split vectors. It also can be extended to use advanced preorder filtering functions such as *getPart()* and *getMatch()* since elements are stored in preorder. It invokes *getNext()* iteratively based on the CPL approach [3] to identify the next query node for processing. Unlike the original TwigFast algorithm, elements are passed straight to the intermediate result storage if they pass a strict prefix path filtering (see Definition 6). If the head element of q_{act} fails to satisfy the strict prefix path matching, its cursor is shifted to point to the next element in the stream and the algorithm proceeds to the next iteration. But first the algorithm performs a weak prefix match by determining the end positions for any element which is not an ancestor of the head element of q_{act} the intermediate lists corresponding to $parent(q_{act})$ according to Definition 3 when level split vectors are used to avoid false negative errors. This is performed by calling the *setEndPointerParent* in Algorithm 2. After that, if the head element of q_{act} has the right ancestor extension and q_{act} is not a leaf query node, the algorithm updates the end values for elements in the same list which are not ancestors. Then, the start positions for intervals of element v_{act} will be determined. These are equal to the current lengths of v_{act}'s child lists and the tail and ancestor pointers are updated. The purpose of these pointers is to identify elements which still have potential descendants. For example, in Figure 8, when y_2 is the head element of $q_{act} = y$, it indicates that a_2 will not have any further descendant so that the end positions intervals for a_2 are recorded. Then, the current element is appended into the corresponding list. The cursor of the current query node q_{act} is advanced and the algorithm proceeds to the next cycle. When all streams have ended, the algorithm concludes the top-down processing by using the largest range-based label (∞, ∞, ∞) to update the intervals of all open elements. To perform strict subtree matching checks, the intermediate results are filtered bottom-up in post-processing order which ensures optimal enumeration. That is, an internal element e_q is removed from the list L_q if and only if for any $n_i \in children(q)$, $e_q.start_{n_i} > e_q.end_{n_i}$. Finally, once the intermediate storage contains elements with their intervals, TwigPrime will enumerate the output by applying the enumeration algorithm introduced in [38] and extended in [7], [20] to use child intervals when level split approach is applied.

The improvement of TwigStackPrime [3] can trivially be ported to algorithms in bottom-up approach such as TJStrict-Pre, TJStrictPost and GTPStack. In this article, we propose new algorithms, namely TJStrictPrePrime, TJStrictPostPrime and GTPStackPrime, which are less involved modifications of the original ones. This is due to the fact that they are based on advanced preorder filtering strategies (i.e., *getPart* and *getMatch*) which are extensions of *getNext*. To achieve such an improvement, *getPart* and *getMatch* are modified to use the CPL approach to improve structural relationship checks. As a result, *getPart(q)* and *getMatch(q)* return an element e_q of a query node $q \in TPQ$ with four properties:

1) 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 head element of a query node $q_i = children(q)$.
2) each of its child elements satisfies recursively the first property.
3) if q has P-C edge(s) connected to 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 using the CPL approach).
4) if $\neg isRoot(q)$, then e_q has a relevant ancestor e_p stored in the main algorithm which has been the head element of a query node p = parent(q) in previous calls of *getPart(p)* or *getMatch(p)*, respectively).

In the same way, TwigPrimePart and TwigPrimeMatch are proposed as refined versions of TwigPrime to utilise the *getPart()* [20] and *getMatch()* [7] functions.

1) ANALYSIS OF TwigPrime

This section shows the correctness of the new algorithms and analyses their complexities.

Lemma 1: Let e_q be an element corresponding to the query node q in the intermediate storage. Then its child and descendant intervals are correctly recorded.

Proof: Query node q is either leaf or internal. If q is a leaf query node, the lemma holds. Otherwise, it is shown by the proof of TwigStackPrime in [3], e_q is returned by the advanced preorder filtering because it satisfies the following properties (1) the current element in stream q has a descendant element in each stream q_i, for $q_i \in$ childrenAD(q), (2) the current element in stream q has a child element in each stream q_i, for $q_i \in$ childrenPC(q), and (3) each current element in stream q_i recursively satisfies the first and second property. Therefore, e_q is appended into the intermediate list before child and descendant elements of e_q are stored in their corresponding lists, and the start positions of the intervals thus can be set correctly at Line 21 of TwigPrime. Using an advanced preorder filtering strategy property, it will be known that all elements in the XML tree which are part of some solutions at subtree rooted at e_q will be returned in preorder. Henceforth, all child and descendant elements of e_q are stored in the intermediate storage while e_q is pointed by the tail of q and the procedure *setEndPointerParent* correctly records the end values for e_q's intervals. For both cases the lemma holds.

The next theorem will be used to prove the correctness of TwigPrime, TwigPrimePart and TwigPrimeMatch.

Theorem 1: Given a twig pattern query Q and an XML document D, Algorithms *TwigPrime*, *TwigPrimePart* and

TwigPrimeMatch correctly construct the intermediate results of Q on D.

Proof: In Algorithm *TwigPrime*, *getNext(getRoot(Q))* is repeatedly invoked to determine the next query node to be processed. It is shown by the proof of TwigStackPrime in [3] that all elements returned by $q_{act} = getNext(getRoot(Q))$ satisfy the following properties (1) the current element in stream q has a descendant element in each stream q_i, for $q_i \in$ childrenAD(q), (2) the current element in stream q has a child element in each stream q_i, for $q_i \in$ childrenPC(q), and (3) each current element in stream q_i recursively satisfies the first and second property. If $q_{act} \neq getRoot(Q)$, Line 7, the algorithm sets the end values for all elements in the intermediate lists $L_{parent(q_{act})}$ that are not ancestors of the head element of q_{act} by Using an advanced preorder filtering strategy property. After that, it is already known q_{act} satisfies the three properties so that Line 9 checks whether the tail of $parent(q_{act})$ is pointed to proper ancestor 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 the ancestor extension which guarantee its participation in a weak match of prefixed path from itself to the root. After that, if q_{act} is connected to the parent query node with P-C edge, Lines 13-16 ensure that the current element has a strict match of a prefixed path. If the head element fails to pass a strict prefix path filtering, then it can be skipped safely to proceed to the next cycle. Otherwise, the corresponding list of v_{act} is cleaned by setting end values of intervals for elements which do not contain the head of v_{act}, and the start positions of intervals for v_{act} are recorded, using Lemma 1. Then, if v_{act} has an ancestor in the same list, the ancestor pointer of v_{act} is pointed to $q_{act}.tail$. Otherwise, $v_{act}.ancestor$ is set to -1 indicating that it does not have a proper ancestor in the list. Finally, $q_{act}.tail$ is updated to point to v_{act}, and v_{act} is appended into its corresponding intermediate list. Once the intermediate storage containing elements with their intervals correctly set, it is straightforward to perform the output enumeration.

The correctness of the enumeration algorithm follows from the correctness of the TwigList enumeration method [38] which is trivially extended to use child intervals when elements are stored in level split lists as in [20]. Moreover, TJStrictPrePrime, TJStrictPostPrime and GTPStackPrime are correct due to the correctness of the preorder filtering used in TwigStackPrime [3] and the correctness of the original algorithms introduced in [7], [20].

With respect to the space and time complexity of these algorithms, the new algorithms read elements from data streams only once in a single forward scan through advanced preorder filtering functions. When elements are appended to the intermediate storage, each child check and interval set take constant time. Therefore, the worst-case time and space complexity when building the intermediate storage is $O(f \times |Input|)$ where f is the maximum fanout at any query node in a TPQ with n query nodes and *Input* is the sum of the lengths of the n input lists. Therefore it is possible to

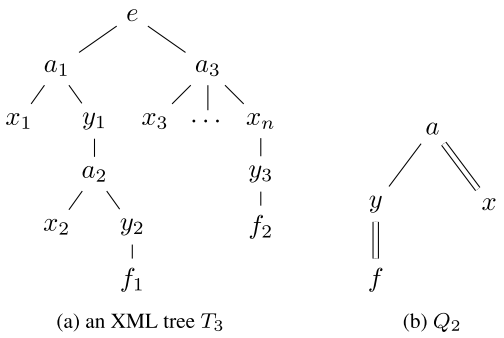
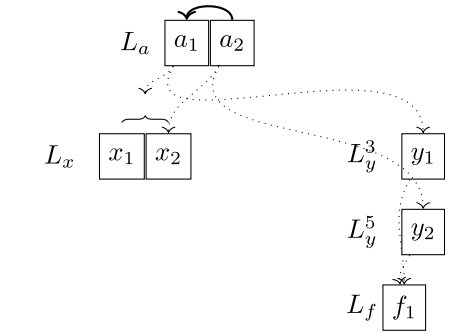


FIGURE 9. An example to illustrate the basic notations of TwigPrime.

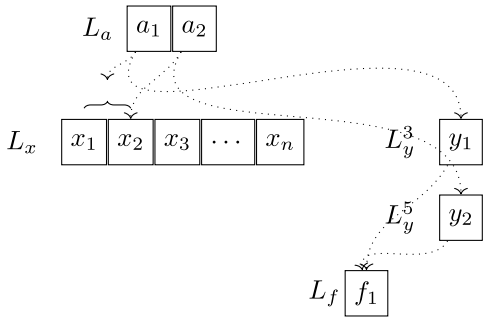
suggest that the new approaches guarantee optimal evaluation for the case where the TPQ has A-D edges or there are only P-C edges connected to the leaf query nodes similar to that provided by TwigStackPrime. Interested readers may refer to [3] to find more details on the optimality conditions. Thus, elements are only stored in the intermediate result if they contribute to the final result. This means that, the intermediate result can be enumerated in linear time $O(|Output|)$ where *Output* is the number of matched elements. However, in the case where P-C axes connect internal query nodes, linear performance for output enumeration can be achieved by performing a strict subtree filtering (i.e., cleaning intermediate result lists bottom-up in the query, by overwriting elements not satisfying subtree matches), but the algorithms can not guarantee optimal evaluation. To put it another way, they can provide optimal enumeration (i.e., all elements in internal lists must be part of the final result). Consequently, the worst-case I/O and CPU time complexity is linear with respect to the sum of the input list sizes and the size of the output result. For example, to demonstrate the difference between optimal evaluation and enumeration, TwigPrime in Figure 10a guarantees optimal evaluation while TJStrictPre and GTPStack of Figure 10b provides optimal enumeration by performing extra passes over the intermediate lists. The space complexity of the new approaches is $O(|Intput|)$ which is linear with respect to the total number of elements whose tags appear in TPQs. This is because they construct the intermediate results directly. However, when the new algorithms are optimal, the $\Omega(u)$ lower bound is matched, where u is the total number of elements to which query nodes can be matched (i.e., optimal evaluation) [19], [39]. However, an early enumeration approach introduced in [12] can significantly reduce the intermediate storage size necessary. The early enumeration starts when the incoming element corresponding to the first branching query node does not have a relevant ancestor in the corresponding intermediate list or stack of the first branching query node.

V. EXPERIMENTAL EVALUATION

This section describes experiments that explore the effects of the CPL approach, different advanced preorder filtering strategies and different intermediate storage approaches in



(a) TwigPrime



(b) TJStrictPre with extra pass filtering and GTPStack

FIGURE 10. One-phased algorithms and their corresponding intermediate storages for processing Q_2 against T_3 in Figure 9.

TABLE 2. Characteristics of the experimental datasets.

Dataset	Size MB	# of nodes		Depth		Distinctive tags
		# of elements	# of attributes	Max	Avg	
DBLP	127	3332130	404276	6	2.9	40
TreeBank	86	2437666	1	36	7.8	251
XMark	116.5	2120857	1	12	5.5	83
Random	31	3948087	0	13	7	6
Zipf	25.5	3641776	0	26	18.5	7

TABLE 3. Zipf TPQ templates for XPath expressions.

Template	Query template	# of queries
t_1	$//\alpha/\beta[//\chi]/\delta$	10
t_2	$//\epsilon[/\eta]/\gamma$	10
t_3	$//\alpha[/\epsilon][/\eta]//\gamma$	10
t_4	$//\alpha/\beta//\chi/\delta$	10
t_5	$//\alpha/\alpha[//\beta]/\chi//\chi[/\delta]//\epsilon$	10

bottom-up holistic twig matching algorithms. It compares the performance of the new bottom-up twig matching algorithms, namely TwigPrime, TwigPrimePart, TwigPrimeMatch, TJStrictPrePrime, TJSTrictPostPrime and GTPStackPrime against state-of-the-art holistic algorithms: TwigList [38], TwigFast [29], TJStrictPre [20], TJStrictPost [20] and GTPStack [7], across a variety of significantly different XML datasets. To improve the efficiency of the output enumeration, TwigList and TwigFast include the strategy of next sibling links introduced in [38]. With the exception of TwigList and TwigFast, the algorithms in the experiments are implemented, by default, to use the level split approach except those labelled with the "_" as suffix to indicate intermediate results are stored in simple lists (e.g., TwigPrime_). When TwigPrime, TwigPrimePart, TwigPrimeMatch TJStrictPrePrime, TJSTrictPostPrime and GTPStackPrime use the simple list approach, they apply next sibling links. TwigPrime_N, TwigPrimePart_N and TwigPrimeMatch_N stand for TwigPrime, TwigPrimePart and TwigPrimeMatch which use the simple list approach and the strategy of next sibling links. Throughout this section, the term "CPL" refers to bottom-up holistic algorithms based on the CPL approach combined with the level split, while the term "CPL_" refers to bottom-up holistic algorithms based on the CPL and simple list approaches. As a result, the "CPL" includes TwigPrime, TwigPrimePart, TwigPrimeMatch, TJStrictPrePrime, TJSTrictPostPrime and GTPStackPrime algorithms. The "CPL_" refers to the set of algorithms including TwigPrime_, TwigPrimePart_ TwigPrimeMatch_, TJStrictPrePrime_, TJSTrictPostPrime_. Note that the CPL approaches are used to denote bottom-up twig matching algorithms using the CPL relationship introduced in [3] regardless the approach used to store intermediate results. Versions of algorithms are implemented as new algorithms to make sure the overhead of the complex methods does not affect the simpler ones.[1]

A. XML DATASETS AND QUERIES

The algorithms were tested using five datasets with a variety of characteristics and sizes, these collections are commonly used in many approaches [7], [20], [29]; DBLP, XMark, TreeBank, Random and Zipf. Table 2 shows the datasets used in the experiments and their properties. They were deliberately chosen to test the algorithms as thoroughly as possible. DBLP is highly structured and is very wide and shallow, while TreeBank is a deep-recursive dataset with a very many distinct tags and an irregular structure. The XMark dataset is well-known benchmark XML dataset which can be generated with the factor f to control the size: we used $f = 1$. We also generated two synthetic datasets called Random and Zipf. The Random dataset contains six tags and has maximum depth sets to 13 and *fan-out* ranging from 0 to 6. The Zipf dataset was generated using the Zipfian distribution to spread the node labels within the dataset. In this article, the Zipf dataset contains seven different labels from a to g, where a is the most common ($\approx$38.55%) and g the least (($\approx$0.055%). Queries for DBLP, XMark, TreeBank and Random datasets are similar to those used in the experiments of [3]. The XML structured queries for evaluation over the Zipf dataset were generated using five query templates shown in Table 3. Templates specify relationships between query nodes. For each template, ten TPQs were randomly generated such that $\alpha, \beta, \chi, \delta \in \{a, b, d, g\}$ and $\epsilon, \eta, \gamma \in \{a, b, c, d, e, f, g\}$. In order to illustrate the difference between the algorithms clearly and make the experiment more comprehensive, ten

[1] 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.

recursive TPQs were generated abased on the template t_5. The complete list of the TPQs can be found in Table 4.

B. METRICS

Our experiments compared two variables for each TPQ. The comparison was based on two metrics; storage and processing time. The storage measure is simply the number of elements stored in the intermediate storage. The running time is more complicated. It is the running time (in milliseconds) of the whole TPQ including filtering and listing the results. All TPQs were executed 103 times, with the timing for the first three runs excluded to avoid cold cache issues. The I/O cost for tag indexing files for the set of algorithms in ''CPL'' and ''CPL_'' was not included because it is negligible, and the cost of reading the tag indexing is constant over the TPQs for each dataset [3] because it only needs to be read once for a set of TPQs over a particular dataset.

C. EXPERIMENTAL RESULTS

The experimental results are very similar over the five datasets so, in this article, we focus on the results from the TreeBank dataset because it is the most complex XML document from most aspects of query processing [6], [35] even though it is not the largest in our experiments. This means that the suboptimal evaluation of the existing approaches can be demonstrated here most clearly. This is shown in Figure 11. This dataset also demonstrates the effectiveness of the CPL filtering strategy in bottom-up approaches.

1) INTERMEDIATE STORAGE

The most significant attribute, with regard to efficiency, of any of the holistic twig matching algorithms is the intermediate storage size. The intermediate result size for each algorithm was evaluated by computing a ratio of the number of elements stored by each algorithm and the number of relevant elements for each collection. A ratio of 1, indicates the algorithm was optimal for all queries tested because no unnecessary results were stored. Beyond that, the smaller the ratio the better, since this shows how successful the algorithm was in filtering out irrelevant elements. As shown in Table 5, the ''CPL'' and ''CPL_'' approaches failed to provide optimal evaluation for TQ_2, TQ_5, TQ_8 and TQ_{11} because useless elements were stored but they stored several orders of magnitude fewer elements than the comparable algorithms. This result may be explained by the fact that the ''CPL'' and ''CPL_'' approaches use the CPL information to eliminate irrelevant elements from the parent streams, but due to the restricted access mechanism, they may store useless elements as they consider only the CPL relationship between two streams while processing TPQs in bottom-up (i.e., elements are forwarded to contain their descendants). As a result, when there are P-C edges between internal query nodes, many useless elements can be removed from the streams safely (e.g., TQ_{11}). For TQ_6, the ''CPL'' and ''CPL_'' approaches performed efficiently by storing only useful elements of 16 062, whereas the number of elements stored in the state-of-the-art algorithms (i.e., TJStrictPre and GPTStack) was between 374 370 for the level split approaches and 563 741 for the simple list approaches. On the other hand, TwigList and TwigFast built up the intermediate storage with 770 052 and 669 312 elements in order to evaluate TQ_6, respectively.

Figure 11 provides a summary of an analysis of the experimental results. To avoid extreme value differences in storage ratio, experimental results of TwigList regarding the number of elements stored in the intermediate storage are not included in the illustrative graphs since the remaining algorithms use an improved version of TwigList. However, TwigList has storage ratios 577.95 for DBLP, 6.34 for XMark, 34.42 for TreeBank and 65 for Random. It can be seen from the data in this figure that the CPL algorithms significantly outperformed other methods. An almost optimal evaluation was achieved in complex datasets with many recursions in the structure (e.g., Treebank, Random and Zipf datasets) and an optimal evaluation was achieved for relatively structured XML collections with a lot of repetitive subtrees (e.g., DBLP and XMark datasets).

2) PROCESSING TIME

Processing time has also been improved but to make the evaluation easier a ratio showing the improvement of all algorithm pairs for all TPQs over each dataset in the experiments has been used to demonstrate the scale of this improvement. The improved ratio (IR) of algorithm A over algorithm B for a set of queries issued over an XML dataset can be computed using Formula 2 [7], [28], where T_A and T_B are the median running times for algorithms A and B, respectively.

$$IR_{A,B} = \frac{T_B - T_A}{T_B} \tag{2}$$

The ''CPL'' and ''CPL_'' approaches significantly outperformed the other algorithms tested with respect to processing time. Eleven different versions of the new approaches are all noticeably faster than the other algorithms, and the combination of TwigPrime and the *getMatch()* function using the simple list approach showed a better performance than the other combinations of TwigPrime (i.e., TwigPrimeMatch_). The reason for this is the use of the CPL approach to filter useless elements and the *getMatch()* to avoid redundant computations. When the new algorithm TwigPrime uses the level split approach, the *getPart()* function has the best performance of all. This is can be attributed to the use of an additional vector which stores one extra value for each query node to check the latest ancestors that form a weak full match for the entire TPQ in *getPart()*. The alternative, *getMatch()* has to check several tails and ancestors to determine whether an element is useful or not. Figure 12 shows results for running TQ_6 with cost divided into two phases, GTPStack algorithm is excluded because it is almost 382 slower than the fastest algorithm. The CPL filtering minimises the cost of constructing intermediate results because the size is reduced, and the cost of enumerating results because unnecessary traversal is avoided. Note that TQ_6 is the most expensive query in the experiments,

TABLE 4. Experimental TPQs.

TPQ	Dataset	XPath expression	Matches
DQ_1	DBLP	/dblp/inproceedings[//title]//author	88
DQ_2	DBLP	//www[editor]/url	21
DQ_3	DBLP	//article[//sup]//title//sub	278
DQ_4	DBLP	//article[/sup]//title/sub	0
XQ_1	XMark	/site/closed_auctions/closed_auction[annotation/description/text/keyword]/date	4042
XQ_2	XMark	/site/closed_auctions/closed_auction//keyword	12527
XQ_3	XMark	/site/closed_auctions/closed_auction[//keyword]/date	12527
XQ_4	XMark	/site/people/person[profile[gender][age]]/name	3243
XQ_5	XMark	//item[location][//mailbox//mail//emph]/description//keyword	16956
XQ_6	XMark	//people/person[//address/zipcode]/profile/education	3241
TQ_1	TreeBank	//S[//MD]//ADJ	19
TQ_2	TreeBank	//S/VP/PP[/NP/VBN]/IN	152
TQ_3	TreeBank	//VP[/DT]//PRP_DOLLAR_	3
TQ_4	TreeBank	//S[/JJ]/NP	5
TQ_5	TreeBank	//S[VP[DT]//NN]/NP	32
TQ_6	TreeBank	//S[//VP/IN]//NP	20311
TQ_7	TreeBank	//S/VP/PP[//NP/VBN]/IN	320
TQ_8	TreeBank	//EMPTY/S//NP[/SBAR/WHNP/PP//NN]/_COMMA_	17
TQ_9	TreeBank	//SINV//NP[/PP//JJR][//S]/NN	4
TQ_{10}	TreeBank	//NP[/NN]/PP	43942
TQ_{11}	TreeBank	//S[//VP][//NP]/VP/PP[IN]/NP/VBN	1185
RQ_1	Random	//b//e//a[//f][d]	1331
RQ_2	Random	//a//b[//e][c]	18033
RQ_3	Random	//e//a[/b][c]	11216
RQ_4	Random	//a[//b/d]//c	59568
RQ_5	Random	//b[d/f]/c[e]/a	377
RQ_6	Random	//c[//b][a]/f	47159
RQ_7	Random	//a[c//e]/f[d]	1906
RQ_8	Random	//d[a//e/f]/c[b]	204
RQ_9	Random	//a[d][c][b][e]//f	3757

TABLE 5. Ratio of the number of elements stored in intermediate storage and the number of relevant elements for each query.

Algorithm \ Query	TQ1	TQ2	TQ3	TQ4	TQ5	TQ6	TQ7	TQ8	TQ9	TQ10	TQ11
TwigFast	1.00	48.01	3917.14	33888.07	7284.63	41.67	23.21	5.22	5.43	2.17	71.32
TwigList	402.53	232.06	19616.71	40842.00	9754.96	47.94	112.21	4248.84	24792.07	3.40	542.80
CPL_	1.00	1.32	1.00	1.00	1.20	1.00	1.10	1.01	1.00	1.00	4.07
CPL	1.00	1.16	1.00	1.00	1.13	1.00	1.00	1.01	1.00	1.00	3.97
TJStrictPost	1.00	3.17	931.86	4732.87	1919.36	23.31	1.99	1.42	2.07	1.12	43.99
TJStrictPost_	1.00	3.54	932.14	4733.80	1919.59	23.38	2.38	1.48	2.21	1.25	44.13
TJStrictPre	1.00	3.17	931.86	4732.87	1919.36	23.31	1.99	1.42	2.07	1.12	43.99
TJStrictPre_	1.00	23.13	2323.71	10813.93	3540.26	35.10	12.17	3.38	3.00	1.45	56.16
GTPStack	1.00	3.17	931.86	4732.87	1919.36	23.31	1.99	1.42	2.07	1.12	43.99

it touches a very high proportion of the document and has a great many query results. It was chosen because it shows the effects of the CPL approach. For TQ_6, only the "CPL" and "CPL_" algorithms can provide optimal evaluation and hence the reduction in the CPU cost of the algorithms.

Figure 13 shows the summary of the results for this experiment on DBLP. Overall, these results show that the CPL_ approaches (i.e., TJStrictPostPrime_, TwigPrimeMatch_N and TwigPrimePart_) provided an efficient solution and improved the overall performance. The results obtained from the experiments on XMark, TreeBank and Random datasets can be compared in Figure 14. For example, on the XMark dataset, the improvement of TwigPrimeMatch_N over TJStrictPre and GPTStack is more than 11% and 72%, respectively. Interestingly, the improvement of TwigPrimeMatch_N was on the TreeBank dataset, over TJStrictPre and GPTStack was observed to be more than 73% and 98%, respectively.

The Zipf collection and queries were included to gain an insight into the advantages and disadvantages of using

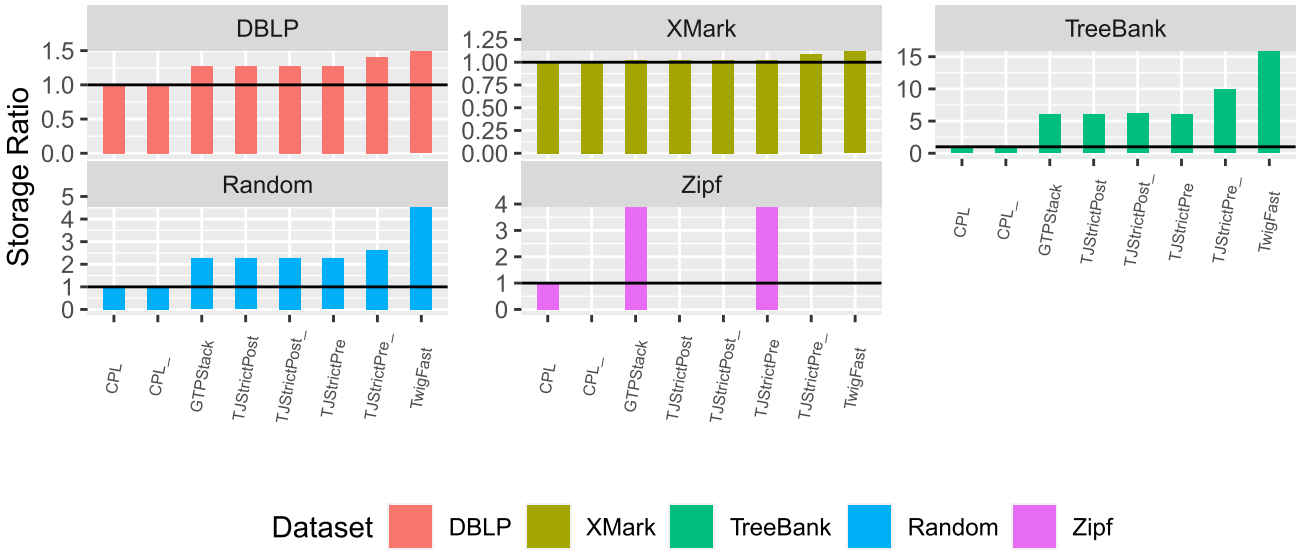


FIGURE 11. Ratio of the number of elements stored in intermediate storage and the number of relevant elements for each dataset. The y-intercepts equal 1 indicating the optimal approach.

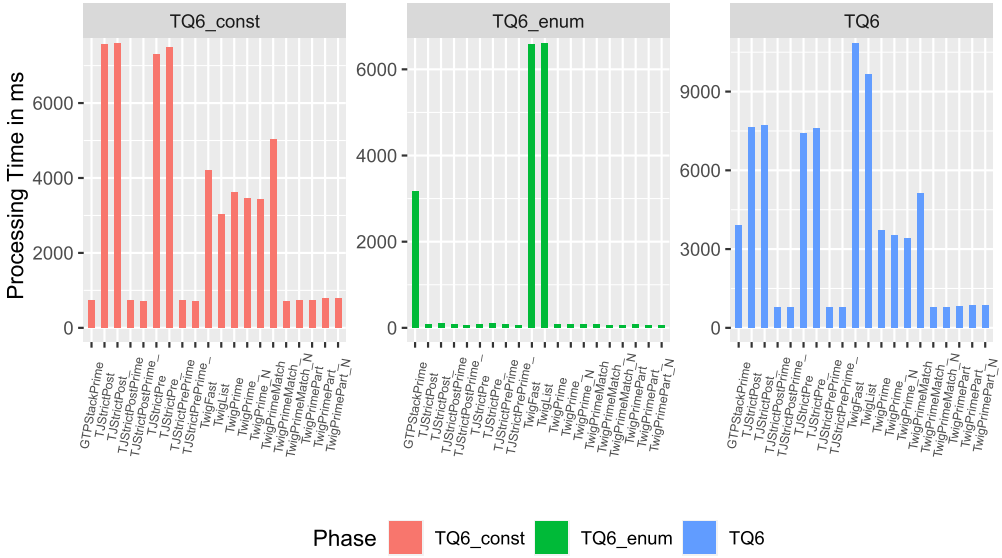


FIGURE 12. Running TQ_6 on TreeBank dataset. Cost divided into constructing intermediate results (TQ_6_const) and result enumeration (TQ_6 _enum).

combinations of the CPL approach, different preorder filtering functions (i.e., *getNext()*, *getPart()* and *getMatch()*) and level split intermediate results. In order to do a sensible comparison only algorithms using the level split approach to build the intermediate storage and storing elements in preorder were compared. Approaches which store elements in postorder in the intermediate storage, such as TJStrictPost and TJStrictPostPrime, were not included in the performance comparison because they output the result tuples unordered, hence they are not directly comparable. One of the properties of the Zipf dataset is that every element has exactly two children and the longest path in the document is 26. Figure 13 provides the experimental data on Zipf dataset. From this data, we can see that the use of the CPL approach improved the existing TJStrictPre by 52%. Moreover, the improvement of the winning algorithm, i.e., TJStrictPrePrime,

Closer inspection of Figures 13 and 14 shows that GPTStack and GPTStackPrime have their performance degraded because of the use of linked lists as the main data structure to store intermediate results. This affects the enumeration performance when A-D relationships exist in TPQs issued over datasets with high repetitive structures such as TreeBank dataset as descendants may overlap, see Figure 12.

3) SUMMARY

The experimental results described above have shown that the CPL approach can filter out irrelevant elements effectively mostly without any overhead. The number of elements stored

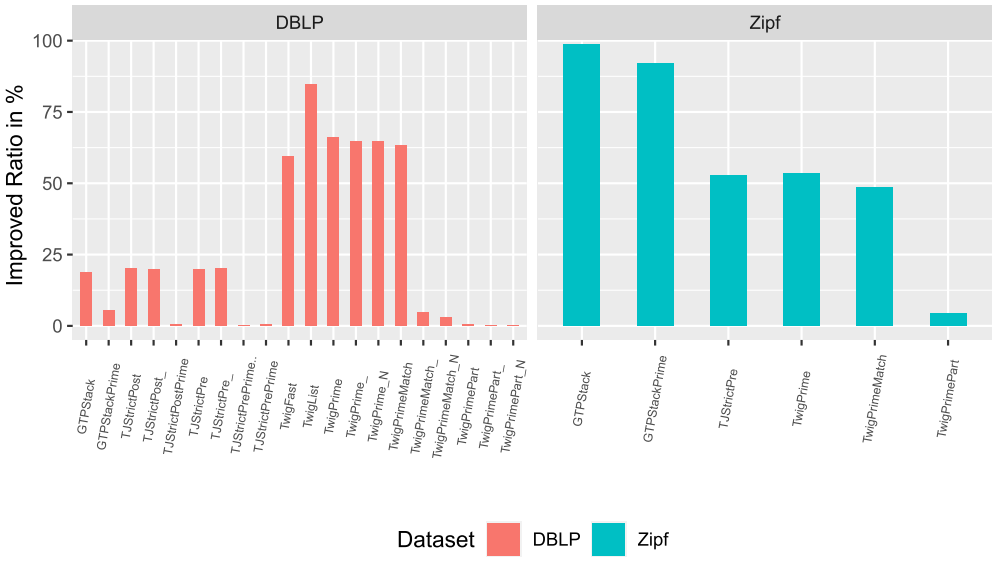


FIGURE 13. The IR of TJStrictPostPrime_ compared to all approaches tested for all queries on DBLP and The IR of TJStrictPrePrime compared to all approaches tested for all queries on Zipf.

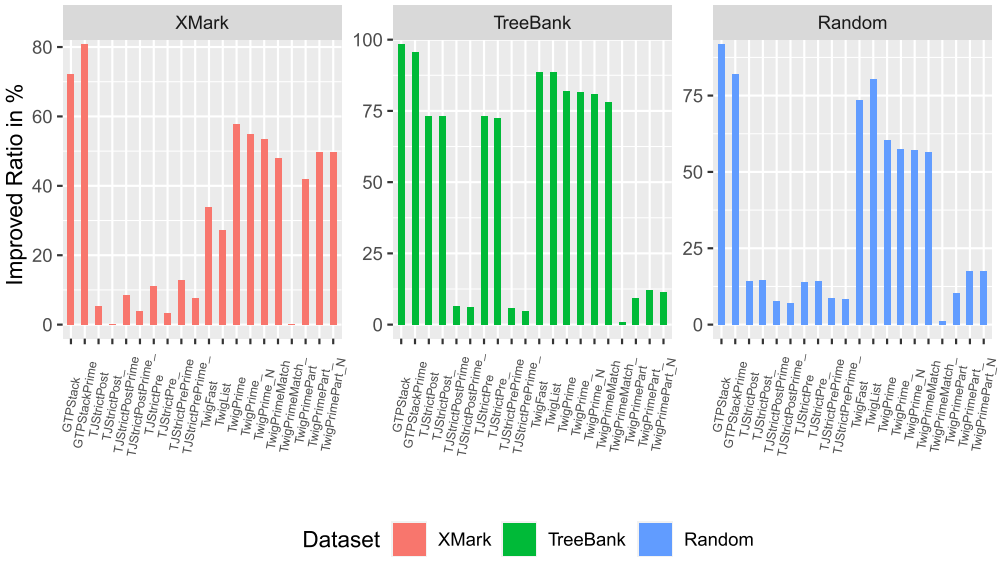


FIGURE 14. The IR of TwigPrimeMatch_N compared to all algorithms tested for all queries on XMark, TreeBank and Random.

by the new algorithms (i.e., "CPL" and "CPL_") is always fewer than that stored by the other up-to-date approaches. In most cases the algorithms proposed have a far better performance than state-of-the-art algorithms. This is because the CPL approach can filter out many useless elements before storing them in the intermediate storage, thus the overall running time is decreased. Surprisingly, the use of simple list to store intermediate results was found to outperform the utilization of the level split technique for all the proposed approaches. This must be because the CPL filtering minimises the cost of building intermediate results because their results are smaller and the cost of enumerating results because redundant traversal is avoided. The outcomes of the experiments appear to support the assumption that using of the same advanced preorder filtering function and the same design of algorithm (e.g., pointers, local stacks with references and level split technique) for all TPQs is not always the best approach. In all cases, however, the algorithms based on the CPL_ and CPL approaches significantly outperformed the other-related algorithms in the experiments. The improvement of the new approaches over state-of-the-art algorithms on common benchmarks such as DBLP, XMark and TreeBank datasets reaches 20%, 72% and 98%, respectively.

VI. CONCLUSION

In this article, we have presented new approaches that use the CPL indexing to improve filtering phase of bottom-up twig matching algorithms. We also introduced a novel design of algorithm which uses the level split approach along with the CPL technique thus avoiding stacks. We have performed experiments that compare our technique with the fastest previous solutions: GTPStack, TJStrictPre, TJStrictPost, TwigFast and TwigList. For common benchmark queries our new CPL algorithms are more than an order of magnitude faster than the other related methods. In terms of space consumption, the new algorithms can filter out many irrelevant elements effectively and it can be observed that the number of elements stored by the algorithms is significantly fewer than that stored by the existing approaches. However, there is still room for improvement.

In future work we would like to combine the algorithms proposed with previous orthogonal approaches such as useless elements skipping [17], [24], [24], refined partitioning [8], [11], virtual streams [27] and content search [44], [45]. Since the study was limited to holistic algorithms which do not use structural summaries, it was not possible to evaluate the performance of the CPL approaches with methods which combine structural summaries and node labelling schemes.

Lastly, it would be interesting to address general subclass of TPQs rather than A-D and P-C edges. Further work also needs to be done to investigate the behaviour of the CPL approach when processing logical expressions and in supporting the GTP semantics.

REFERENCES

[1] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu, "Structural joins: A primitive for efficient XML query pattern matching," in *Proc. 18th Int. Conf. Data Eng.*, 2002, pp. 141–152.

[2] S. A. Aghili, L. Hua-Gang, D. Agrawal, and A. El Abbadi, "TWIX: Twig structure and content matching of selective queries using," in *Proc. 1st Int. Conf. InfoScale*, 2006, p. 42.

[3] S. Alsubai and S. North, "TwigstackPrime: A novel twig join algorithm based on prime numbers," in *Web Information Systems and Technologies*, T. A. Majchrzak, P. Traverso, K.-H. Krempels, and V. Monfort, Eds. Cham, Switzerland: Springer, 2018, pp. 1–20.

[4] R. Bača and M. Krátký, "On the efficiency of a prefix path holistic," in *Proc. 6th Int. XML Database Symp. Database XML Technol. (XSym)*, no. 201, Z. Bellahsène, E. Hunt, M. Rys, and R. Unland, Eds. Lyon, France, Berlin, Germany: Springer, Aug. 2009, pp. 25–32.

[5] R. Bača and M. Krátký, "XML query processing," in *Proc. 16th Int. Database Eng. Appl. Symp. (IDEAS)*, 2012, pp. 8–13.

[6] R. Bača, M. Krátký, I. Holubová, M. Nečaský, T. Skopal, M. Svoboda, and S. Sakr, "Structural XML query processing," *ACM Comput. Surv.*, vol. 50, no. 5, pp. 1–41, 2017.

[7] 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. 369–393, Oct. 2012.

[8] R. Bača, M. Krátký, and V. Snášel, "On the efficient search of an XML twig query in large dataguide trees," in *Proc. Int. Symp. Database Eng. Appl. (IDEAS)*, New York, NY, USA, 2008, pp. 149–158.

[9] R. Bača, J. Walder, M. Pawlas, and M. Krátký, "Benchmarking the compression of XML node streams," in *Database Systems for Advanced Applications. DASFAA* (Lecture Notes in Computer Science), vol. 6193, M. Yoshikawa, X. Meng, T. Yumoto, Q. Ma, L. Sun, and C. Watanabe, Eds. Berlin, Germany: Springer, 2010.

[10] N. Bruno, N. Koudas, and D. Srivastava, "Holistic twig joins: Optimal XML pattern matching," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Madison, WI, USA, 2002, pp. 310–321.

[11] B. Chen, T. W. Ling, M. T. Ozsu, and Z. Zhu, "On label stream partition for efficient holistic," in *Proc. 12th Int. Conf. Database Syst. Adv. Appl. (DASFAA)*, Bangkok, Thailand, 2007, pp. 807–818.

[12] S. Chen, H.-G. Li, J. Tatemura, W.-P. Hsiung, D. Agrawal, K. Sel, U. Candan, and K. S. Candan, "Twig^{2}Stack: Bottom-up processing of generalized-tree-pattern queries over XML documents," in *Proc. 32nd Int. Conf. Very Large Data Bases*, Seoul, Sounth Korea, 2006, pp. 283–294.

[13] T. Chen, J. Lu, and T. W. Ling, "On boosting holism in XML twig pattern matching using structural indexing techniques," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 2005, pp. 455–466.

[14] Z. Chen, H. V. Jagadish, L. V. S. Lakshmanan, and S. Paparizos, "From tree patterns to generalized tree patterns: On efficient evaluation of XQuery," in *Proc. VLDB*, 2003, pp. 237–248.

[15] B. Choi, M. Mahoui, and D. Wood, "On the optimality of holistic algorithms for twig queries," in *Proc. Database Expert Syst. Appl. 14th Int. Conf. (DEXA)*, Prague, Czech Republic, Sep. 2003, pp. 28–37.

[16] D. Che, T. W. Ling, and W.-C. Hou, "Holistic Boolean-twig pattern matching for efficient XML query processing," *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 11, pp. 2008–2024, Nov. 2012.

[17] M. Fontoura, V. Josifovski, E. Shekita, and B. Yang, "Optimizing cursor movement in holistic twig joins," in *Proc. 14th ACM Int. Conf. Inf. Knowl. Manage. (CIKM)*, 2005, pp. 784–791.

[18] G. Gou and R. Chirkova, "Efficiently querying large XML data repositories: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 19, no. 10, pp. 1381–1403, Oct. 2007.

[19] N. Grimsmo and T. A. Bjørklund, "Towards unifying advances in twig join algorithms," in *Proc. 21st Australas. Conf. Database Technol.*, vol. 104, 2010, pp. 57–66.

[20] N. Grimsmo, T. A. Bjørklund, and M. L. Hetland, "Fast optimal twig joins," *Proc. VLDB Endowment*, vol. 3, nos. 1–2, pp. 894–905, Sep. 2010.

[21] M. Hachicha and J. Darmont, "A survey of XML tree patterns," *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 1, pp. 29–46, Jan. 2013.

[22] S.-C. Haw and C.-S. Lee, "Data storage practices and query processing in XML databases: A survey," *Knowl.-Based Syst.*, vol. 24, no. 8, pp. 1317–1340, Dec. 2011.

[23] H. Jiang, W. Wang, H. Lu, and J. Yu, "Holistic twig joins on indexed XML documents," in *Proc. 29th Int. Conf. Very Large Data Bases*, vol. 29, 2003, pp. 273–284.

[24] H. Jiang, H. Lu, W. Wang, and B. C. Ooi, "XR-tree: Indexing XML data for efficient structural joins," in *Proc. 19th Int. Conf. Data Eng.*, 2003, pp. 253–264.

[25] Z. Jiang, C. Luo, and W.-C. Hou, "An efficient one-phase holistic twig join algorithm for XML data," in *Proc. 15th ACM Int. Conf. Inf. Knowl. Manage.*, Arlington, VA, USA, 2006, pp. 786–787.

[26] Z. Jiang, C. Luo, W.-C. Hou, Q. Zhu, and D. Che, "Efficient processing of XML twig pattern: A novel one-phase holistic solution," in *Proc. Database Expert Syst. Appl. 18th Int. Conf. (DEXA)*, Regensburg, Germany, Berlin, Germany: Springer, Sep. 2007, pp. 87–97.

[27] S. Lee, B.-G. Ryu, and K.-L. Wu, "Examining the impact of data-access cost on XML twig pattern matching," *Inf. Sci.*, vol. 203, pp. 24–43, Oct. 2012.

[28] G. Li, J. Feng, Y. Zhang, and L. Zhou, "Efficient holistic twig joins in leaf-to-root combining with root-to-leaf way," in *Proc. 12th Int. Conf. Database Syst. Adv. Appl. (DASFAA)*, Bangkok, Thailand, vol. 1, R. Kotagiri, P. R. Krishna, M. Mohania, E. Nantajeewarawat, Eds. 2007, pp. 834–849.

[29] J. Li and J. Wang, "Fast matching of twig patterns," in *Database and Expert Systems Applications. DEXA* (Lecture Notes in Computer Science), vol. 5181, S. S. Bhowmick, J. Küng, and R. Wagner, Eds. Berlin, Germany: Springer, 2008.

[30] J. Li and J. Wang, "TwigBuffer: Avoiding useless intermediate," in *Proc. 13th Int. Conf. Database Syst. Adv. Appl. (DASFAA)*, New Delhi, India, J. R. Haritsa, R. Kotagiri, V. Pudi, Eds. Berlin, Germany: Springer, Mar. 2008, pp. 1–8.

[31] G. Liu, M. Yao, D. Wang, and E. Chen, "A novel three-phase XML twig pattern matching algorithm based on version tree," in *Proc. 8th Int. Conf. Fuzzy Syst. Knowl. Discovery (FSKD)*, vol. 3, 2011, pp. 1678–1688.

[32] J. Lu, T. Chen, and T. W. T. Ling, "Efficient processing of XML twig patterns with parent child edges: A look-ahead approach," in *Proc. 13th ACM Int. Conf. Inf. Knowl. Manage.*, Washington, DC, USA, 2004, pp. 533–542.

[33] J. Lu, T. W. Ling, Z. Bao, and C. Wang, "Extended XML tree pattern matching: Theories and algorithms," *IEEE Trans. Knowl. Data Eng.*, vol. 23, no. 3, pp. 402–416, Mar. 2011.

[34] J. Lu, T. W. Ling, T. Yu, C. Li, and W. Ni, "Efficient processing of ordered XML twig pattern," in *Proc. 16th Int. Conf. Database Expert Syst. Appl. (DEXA)*, Copenhagen, Denmark, K. V. Andersen, J. Debenham, R. Wagner, Eds. Berlin, Germany: Springer, Aug. 2005, pp. 300–309.
[35] J. Lu, X. Meng, and T. W. Ling, "Indexing and querying XML using extended dewey labeling scheme," *Data Knowl. Eng.*, vol. 70, no. 1, pp. 35–59, Jan. 2011.
[36] P. Lukas, R. Bača, and M. Krátký, "Cooking Lightweight XML query processor with binary joins and comparing it with holistic joins: Technical report," *Comput. Res. Repository (CoRR)*, vol. abs/1703.0, pp. 1–24, Mar. 2017.
[37] C. Mathis, T. Härder, K. Schmidt, and S. Bächle, "XML indexing and storage: Fulfilling the wish list," *Comput. Sci. Res. Develop.*, vol. 30, no. 1, pp. 51–68, 2012.
[38] L. Qin, J. X. Yu, and B. Ding, "TwigList: Make twig pattern matching fast," in *Proc. 12th Int. Conf. Database Syst. Adv. Appl. (DASFAA)*, R. Kotagiri, P. R. Krishna, M. Mohania, and E. Nantajeewarawat, Eds. Bangkok, Thailand, Berlin, Germany: Springer, 2007, pp. 850–862.
[39] M. Shalem and Z. Bar-Yossef, "The space complexity of processing XML twig queries over indexed documents," in *Proc. IEEE 24th Int. Conf. Data Eng.*, Apr. 2008, pp. 824–832.
[40] M. A. Tahraoui, K. Pinel-Sauvagnat, C. Laitang, M. Boughanem, H. Kheddouci, and L. Ning, "A survey on tree matching and XML retrieval," *Comput. Sci. Rev.*, vol. 8, pp. 1–23, May 2013.
[41] *XML Path Language (XPath)*, 2.0 2nd ed., W3C, Cambridge, MA, USA, 2010.
[42] *XQuery 3.0: An XML Query Language*, W3C, Cambridge, MA, USA, 2014.
[43] H. Wu, C. Lin, T. W. Ling, and J. Lu, "Processing XML twig pattern query with wildcards," in *Database and Expert Systems Applications. DEXA* (Lecture Notes in Computer Science), vol. 7446, S. W. Liddle, K. D. Schewe, A. M. Tjoa, and X. Zhou, Eds. Berlin, Germany: Springer, 2012.
[44] H. Wu, T. W. Ling, B. Chen, and L. Xu, "TwigTable: Using semantics in XML twig pattern query processing," in *Journal on Data Semantics XV* (Lecture Notes in Computer Science), vol. 6720, S. Spaccapietra, Ed. Berlin, Germany: Springer, 2011.
[45] H. Wu, T. W. Ling, and G. Dobbie, "TP+output: Modeling complex output information in XML twig pattern query," in *Proc. 7th Int. XML Database Symp. (XSym)*, Singapore, 2010, pp. 128–143.
[46] X. Wu, M. L. Lee, and W. Hsu, "A prime number labeling scheme for dynamic ordered XML trees," in *Proc. 20th Int. Conf. Data Eng.*, 2004, pp. 66–78, doi: 10.1109/ICDE.2004.1319985.
[47] X. Xu, Y. Feng, and F. Wang, "Efficient processing of XML twig queries with all predicates," in *Proc. 8th IEEE/ACIS Int. Conf. Comput. Inf. Sci. (ICIS)*, Jun. 2009, pp. 457–462.
[48] T. Yu, T. W. Ling, and J. Lu, "TwigStackList¬: A holistic twig join algorithm for twig query with not-predicates on XML data," in *Database Systems for Advanced Applications*, vol. 3882, M. L. Lee, K.-L. Tan, and V. Wuwongse, Eds. Berlin, Germany: Springer, 2006, pp. 249–263.
[49] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman, "On supporting containment queries in relational database management systems," *ACM SIGMOD Rec.*, vol. 30, no. 2, pp. 425–436, Jun. 2001.
[50] S. Alsubai, "Child prime label approaches to evaluate XML structured queries," Ph.D. dissertation, Univ. Sheffield, Sheffield, U.K., 2018.
[51] P. Michiels, G. A. Mihaila, and J. Simeon, "Put a tree pattern in your algebra," in *Proc. IEEE 23rd Int. Conf. Data Eng.*, Apr. 2007, pp. 246–255.

SHTWAI ALSUBAI received the bachelor's degree in information system from King Saud University, Saudi Arabia, in 2008, the master's degree in computer science from CLU, USA, in 2011, and the Ph.D. degree from The University of Sheffield, U.K., in 2018. He is currently an Assistant Professor in computer science. His research interests include XML, XML query processing, XML query optimization, machine learning, and natural language processing.

SIOBHÁN NORTH received the degree in mathematics and the Ph.D. degree in computer science from The University of Sheffield. She is currently a Senior Lecturer in computer science. She currently works in two areas such as XML databases and formal languages. The XML database work currently concerns indexing and compression techniques, and the formal language work relates to translation between Z and SAL.