

# Fast Optimal Twig Joins

Nils Grimsmo  
Norwegian University of  
Science and Technology  
nilsgri@idi.ntnu.no

Truls A. Bjørklund  
Norwegian University of  
Science and Technology  
trulsamu@idi.ntnu.no

Magnus Lie Hetland  
Norwegian University of  
Science and Technology  
mlh@idi.ntnu.no

## ABSTRACT

In XML search systems twig queries specify predicates on node values and on the structural relationships between nodes, and a key operation is to join individual query node matches into full twig matches. Linear time twig join algorithms exist, but many non-optimal algorithms with better average-case performance have been introduced recently. These use somewhat simpler data structures that are faster in practice, but have exponential worst-case time complexity. In this paper we explore and extend the solution space spanned by previous approaches. We introduce new data structures and improved strategies for filtering out useless data nodes, yielding combinations that are both worst-case optimal and faster in practice. An experimental study shows that our best algorithm outperforms previous approaches by an average factor of three on common benchmarks. On queries with at least one unselective leaf node, our algorithm can be an order of magnitude faster, and it is never more than 20% slower on any tested benchmark query.

## 1. INTRODUCTION

XML has become the de facto standard for storing and transferring semistructured data due to its simplicity and flexibility [6], with XPath and XQuery as the standard query languages. XML documents have tree structure, where elements (tags) are internal tree nodes, and attributes and text values are leaf nodes. Information may be encoded both in structure and content, and query languages need the expressional power to specify both.

*Twig pattern matching* (TPM) is an abstract matching problem on trees, which covers a subset of XPath, which again is a subset of XQuery. TPM is important because it represents the majority of the workload in XML search systems [6]. Both data and queries (twigs) in TPM are node-labeled trees, with no distinction between node types. Figure 1 shows a twig query and data with a match. A match is a mapping of query nodes to data nodes that respects labels and the ancestor-descendant (A–D) and parent-child (P–C)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were presented at The 36th International Conference on Very Large Data Bases, September 13-17, 2010, Singapore.

*Proceedings of the VLDB Endowment*, Vol. 3, No. 1  
Copyright 2010 VLDB Endowment 2150-8097/10/09... \$ 10.00.

relationships specified by the query edges, respectively represented by double and single lines in figures here.

*Twig joins* are algorithms for evaluating TPM queries on indexed data, where the index typically has one list of data nodes for each label. A query is evaluated by reading the label-matching data nodes for each query node, and combining these into full query matches. There exist algorithms that perform twig joins in worst-case optimal time [3], but current non-optimal algorithms that use simpler data structures are faster in practice [10, 11].

In this paper we present twig join algorithms that achieve worst-case optimality without sacrificing practical performance. Our main contributions are (i) a classification of filtering methods as *weak* or *strict*, and a discussion of how filtering influences practical and worst-case performance; (ii) level split vectors, a data structure yielding linear-time result enumeration with almost no practical overhead; (iii) *getPart*, a method for merging input streams that gives additional inexpensive filtering and practical speedup; (iv) *TJStrictPost* and *TJStrictPre*, worst-case optimal algorithms that unify and extend previous filtering strategies; and (v) a thorough experimental comparison of the effects of combining different techniques. Compared to the fastest previous solution, our best algorithm is on average three times as fast, and never more than 20% slower.

The scope of this paper is twig joins reading simple streams from label-partitioned data. See Section 6 for orthogonal related work that introduces other assumptions on how to partition and access the underlying data.

## 2. BACKGROUND

A schema-agnostic system for indexing labeled trees usually maintains one list of data nodes per label. Each node is stored with position information that enables checking A–D and P–C relationships in constant time. A common

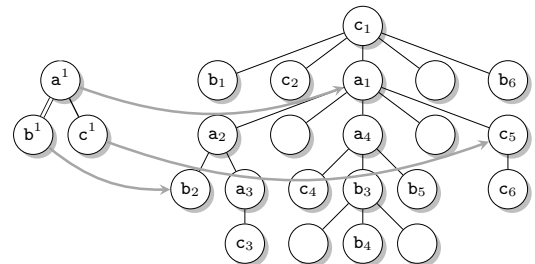
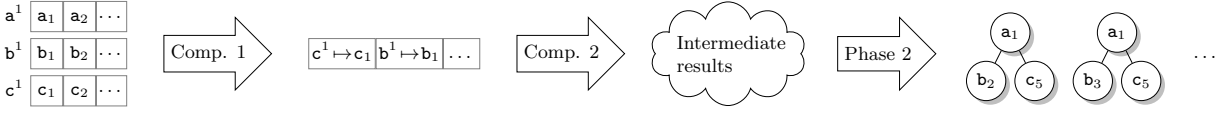


Figure 1: Twig query and data with matches.



**Figure 2: Work-flow of twig join algorithms, with input stream merge component, intermediate result construction component, and result enumeration phase.**

approach is to assign intervals to nodes, such that containment reflects ancestry. Tree depth can then be used determine parenthood [15].

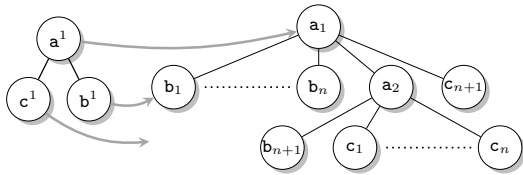
An early approach to twig joins was to evaluate query tree edges separately using binary joins, but when A–D edges are involved, this can give huge intermediate results even when the final set of matches is small [2]. This deficiency led to the introduction of multi-way twig join algorithms. TwigStack [2] can evaluate twig queries without P–C edges in linear time. It only uses memory linear in the maximum depth of the data tree. However, when P–C and A–D edges are mixed, more memory is needed to evaluate queries in linear time [13]. The example in Figure 3 hints at why.

More recent algorithms, which are used as a starting point for our methods, relax the memory requirement to be linear in the size of the input to the join. They follow a general scheme illustrated in Figure 2. The scheme has two phases, where the first phase has two components. The first component merges the stream of data node matches for each query node into a single stream of query and data node pairs. The second component organizes these matches into an intermediate data structure where matched A–D and P–C relationships are registered. This structure is used to enumerate results in the second phase.

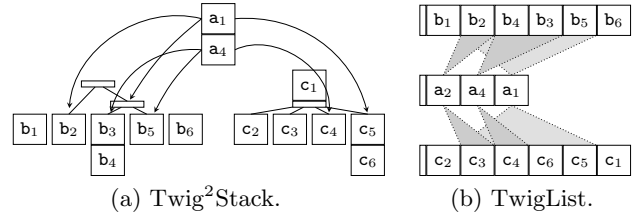
The algorithms broadly fall into two categories. So-called top-down and bottom-up algorithms process and store the data nodes in preorder and postorder, respectively, and filter data nodes on matched *prefix paths* and *subtrees* before they are added to the intermediate results. Many algorithms use both types of filtering, which means the processing is a hybrid of top-down and bottom-up.

Twig<sup>2</sup>Stack [3] was the first linear twig join algorithm. It reorders the input into a single postorder stream to build intermediate results bottom-up. The data nodes matching a query node are stored in a composite data structure (postorder-sorted lists of trees of stacks), as shown in Figure 4(a). Matches are added to the intermediate results only if relations to child query nodes are satisfied, and each match has a list of pointers to usable child query node matches.

HolisticTwigStack [8] uses similar data structures, but builds intermediate results top-down in preorder, and fil-



**Figure 3: Hard case with restricted memory. It cannot be known whether  $b_1, \dots, b_n$  are useful before  $c_{n+1}$  is seen, or whether  $c_1, \dots, c_n$  are useful before  $b_{n+1}$  is seen.**



**Figure 4: Intermediate result data structures when evaluating the query in Figure 1.**

ters matches on whether there is a usable match for the parent query node. It uses the *getNext* function from the TwigStack algorithm [2] as input stream merge component, which implements an inexpensive weaker form of bottom-up filtering. The combined filtering does not give worst-case optimality, but results in faster average-case evaluation of queries than for Twig<sup>2</sup>Stack [8].

One approach for improving practical performance is using simpler data structures for intermediate results. TwigList [11] evaluates data nodes in postorder like Twig<sup>2</sup>Stack, but stores intermediate nodes in simple vectors, and does not differentiate between A–D and P–C relationships in the construction phase. Given a parent and child query node, the descendants of a match for the parent are found in an interval in the child’s vector, as shown in Figure 4(b). Interval start indexes are set as nodes are pushed onto a global stack in preorder, and end indexes are set as nodes are popped off the global stack in postorder. A node is added to the intermediate results if all descendant intervals are non-empty. Compared to Twig<sup>2</sup>Stack, this gives weaker filtering and non-linear worst-case performance, but is more efficient in practice [11], according to the authors because of less computational overhead and better spatial locality.

TwigFast [10] is an algorithm that uses data structures similar to those of TwigList, but stores data nodes in preorder. It uses the same preorder filtering as HolisticTwigStack, and inherits postorder filtering from the getNext input stream merge component. There are several algorithms that utilize both types of filtering, but among these TwigFast has the best practical performance [1, 3, 10]. Figure 5 gives an overview of twig join algorithms, and various properties that are introduced in Section 3.

### 3. PREMISES FOR PERFORMANCE

To make algorithms that are both fast in practice and worst-case optimal, we need an understanding of how filtering strategies and data structures impact performance.

For any graph  $G$ , let  $V(G)$  be its node set and  $E(G)$  be its edge set. Let a *matching problem*  $\mathfrak{M}$  be a triple  $\langle Q, D, I \rangle$ , where  $Q$  is a query tree,  $D$  is a data tree, and  $I \subseteq V(Q) \times V(D)$  is a relation such that for  $q \mapsto q' \in I$  the node label

Algorithm	Ref.	Filtering order	Checking of path		Interm. results	Optimal
				subtree		
getNext	[2]	GN	none	weak	N/A	N/A
TwigStack	[2]	GN+pre	strict	weak	complex	no
Twig <sup>2</sup> Stack	[3]	post	none	strict	complex	yes
$\frac{1}{2}$ PathStack + T <sup>2</sup> S	[3]	pre+post	weak	strict	complex	yes
$\frac{1}{2}$ TwigStack + T <sup>2</sup> S	[1]	GN+pre+post	weak	strict	complex	yes
HolisticTwigStack	[8]	GN+pre	weak	weak	complex	no
TwigList	[11]	post	none	weak	vectors	no
TwigMix	[10]	GN+pre+post	weak	weak	vectors	<i>incorrect</i>
TwigFast	[10]	GN+pre	weak	weak	vectors	no
TJStrictPost	Sect. 4	pre+post	strict	strict	vectors	yes
TJStrictPre	Sect. 4	GN+pre(+post)	strict	strict	vectors	yes

**Figure 5: Previous combinations of prefix path and subtree filtering. Intermediate result storage order given by last item in “filtering order”. GN is the node order returned by the getNext input stream merger.**

of  $q$  equals the node label of  $q'$ . Each edge  $\langle p, q \rangle \in Q$  has a label  $L(\langle p, q \rangle) \in \{\text{“A–D”}, \text{“P–C”}\}$ , specifying an ancestor–descendant or parent–child relationship. Let a *node map* for  $\mathfrak{M}$  be any function  $M \subseteq I$ . Assume a given  $\mathfrak{M} = \langle Q, D, I \rangle$  when not otherwise specified.

**DEFINITION 1 (WEAK/STRICT EDGE SATISFACTION).**

The node map  $M$  weakly satisfies a downward edge  $e = \langle p, q \rangle \in E(Q)$  iff  $M(p)$  is an ancestor of  $M(q)$ , and strictly satisfies  $e$  iff  $M(p)$  and  $M(q)$  are related as specified by  $L(e)$ .

**DEFINITION 2 (MATCH).**

Given subgraphs  $Q'' \subseteq Q' \subseteq Q$ , the node map  $M : V(Q') \rightarrow V(D)$  is a weak (strict) match for  $Q''$  iff all edges in  $Q''$  are weakly (strictly) satisfied by  $M$ . If  $Q'' = Q$  we call  $M$  a weak (strict) full match.

Where no confusion arises, the term weak (strict) match may also be used for  $M(Q)$ .

We denote the set of unique strict full matches by  $O$ . As is common, we view the size of the query as a constant, and call a twig join algorithm linear and optimal if the combined data and result complexity is  $\mathcal{O}(I + O)$  [3].<sup>1</sup>

The results presented in the following all apply to both weak and strict matching, unless otherwise specified. The following lemma implies that we can use filtering strategies that only consider parts of the query.

**LEMMA 1 (FILTERING).** *If there exists a  $Q' \subseteq Q$  containing  $q$  where no match  $M'$  for  $Q'$  contains  $q \mapsto q'$ , then there exists no match  $M$  for  $Q$  containing  $q \mapsto q'$ .*

**PROOF.** By contraposition. Given a match  $M \ni q \mapsto q'$  for  $Q$ , for any  $Q' \subseteq Q$  containing  $q$ , the match  $M \setminus \{p \mapsto p' \mid p \notin Q'\}$  matches  $Q'$  and contains  $q \mapsto q'$ .  $\square$

### 3.1 Preorder Filtering on Matched Paths

Many current algorithms use the getNext input stream merge component [2], which returns data nodes in a relaxed preorder, which only dictates the order of matches for query nodes related by ancestry. This is not detailed in the original

<sup>1</sup>For Twig<sup>2</sup>Stack the combined data, query and result complexity is  $\mathcal{O}(I \log Q + Ib_Q + OQ)$ , where  $b_Q$  is the maximum branching factor in the query [3]. The TJStrict algorithms we present in Section 3 have the same complexity when using a heap-based input stream merger, and  $\mathcal{O}(IQ + OQ)$  complexity when using a getNext-based input stream merger. Note that the total number of data nodes references in the input and output are  $|I|$  and  $|O| \cdot |Q|$ , respectively.

description [2] and is easy to miss.<sup>2</sup> The TwigMix algorithm incorrectly assumes strict preorder [10] (See Appendix E).

**DEFINITION 3 (MATCH PREORDER).** *The sequence of pairs  $q_1 \mapsto q'_1, \dots, q_k \mapsto q'_k \in V(Q) \times V(D)$  is in global match preorder iff for any  $i < j$  either (1)  $q'_i$  precedes  $q'_j$  in tree preorder, or (2)  $q'_i = q'_j$  and  $q_i$  precedes  $q_j$  in tree postorder. The sequence is in local match preorder if (1) and (2) hold for any  $i < j$  where  $q_i = q_j$  or  $\langle q_i, q_j \rangle \in E(Q)$  or  $\langle q_j, q_i \rangle \in E(Q)$ .*

The following definition formalizes a filtering criterion commonly used when processing data nodes in preorder, local or global.

**DEFINITION 4 (PREFIX PATH MATCH).**  *$M$  is a prefix path match for  $q_k \in Q$  iff it is a match for the (simple) path  $q_1 \dots q_k$ , where  $q_1$  is the root of  $Q$ .*

To implement prefix path match filtering, preorder algorithms maintain the set of open nodes, i.e., the ancestors, at the current position in the tree. Most algorithms have one stack of open data nodes for each query node, and given a current pair  $q \mapsto q'$  pop non-ancestors of  $q'$  from the stacks of  $q$  and its parent [2, 8, 10]. Weak filtering can then be implemented by checking if (i)  $q$  is the root, or (ii) the stack for the parent of  $q$  is non-empty. If  $q'$  is not filtered out, it is pushed onto the stack for  $q$ , and added to the intermediate results. This can be extended to strict checking of P–C edges by inspecting the top node on the parent’s stack. Strict prefix path matching is rarely used in practice, as can be seen from the fourth column in Figure 5.

The implementation of prefix path checks is the reason for the secondary ordering on query node postorder for match pairs in Definition 3. Without the secondary ordering problems arise when multiple query nodes have the same label: A data node could be misinterpreted as a usable ancestor of itself when checking for non-empty stacks, or hide a proper parent of itself when checking top stack elements.

Algorithms storing intermediate results in postorder use a global stack for the query [3, 11], and inspection of the top stack node cannot be used to implement prefix path matching when the query contains A–D edges, as ancestors may be hidden deep in the stack. Extending these algorithms to implement prefix path filtering requires maintaining additional data structures.

<sup>2</sup>To be precise, getNext also returns matches for sibling query nodes in order.

The choice of preorder filtering does not influence optimality, as illustrated by the following example.

**EXAMPLE 1.** Assume non-branching data generated by  $/(\alpha_1/)^n \dots (\alpha_m/)^n \beta/\gamma$ , and the query  $\|\alpha_1\| \dots \|\alpha_m\|/\gamma$ , where  $\alpha_1, \dots, \alpha_m, \beta, \gamma$  are all distinct labels. Unless it is signaled bottom-up that the pattern  $\alpha_m/\gamma$  is not matched below, the result enumeration phase will take  $\Omega(n^m)$  time, because all combinations of matches for  $\alpha_1, \dots, \alpha_m$  will be tested.

### 3.2 Postorder Filtering on Matched Subtrees

The ordering of match pairs required by most bottom-up algorithms is symmetric with the global preorder case:

**DEFINITION 5 (MATCH POSTORDER).** A sequence of pairs  $q_1 \mapsto q'_1, \dots, q_k \mapsto q'_k$  is in match postorder iff for any  $i < j$  either (1)  $q'_i$  precedes  $q'_j$  in tree postorder, or (2)  $q'_i = q'_j$  and  $q_i$  precedes  $q_j$  in tree preorder.

The second property is required for the correctness of both Twig<sup>2</sup>Stack [3], where a data node could hide proper children of itself, and TwigList [11], where a node could be added as a descendant of itself.

**DEFINITION 6 (SUBTREE MATCH).**  $M$  is a subtree match for  $q \in Q$  iff it is a match for the subtree rooted at  $q$ .

Example 1 also illustrates why strict subtree match checking is required for optimality, because no node labeled  $\gamma$  is a direct child of a node labeled  $\alpha_m$  in the data. As described in Section 2, Twig<sup>2</sup>Stack and TwigList respectively implement strict and weak subtree match filtering, and for these algorithms strict filtering is required for optimality. TwigList could be extended to strict filtering by traversing descendant intervals to look for direct children, but this would have quadratic cost if implemented naively, as descendant intervals may overlap.

In algorithms storing data in preorder, nodes are added to the intermediate results after passing preorder checks. If nodes are stored in arrays, later removing a node failing a postorder check from the middle of an array would incur significant cost. Note that many of the algorithms listed in Figure 5 inherit weak subtree match filtering from the input stream merger used, as described later in Section 4.4.

### 3.3 Result Enumeration

Even if the strict subtree match checking sketched for TwigList above could be implemented efficiently, results would still not be enumerated in linear time, as usable child nodes may be scattered throughout descendant intervals, as shown by the following example:

**EXAMPLE 2.** Assume a tree constructed from the nodes  $\{\mathbf{a}_1, \dots, \mathbf{a}_n, \mathbf{b}_1, \dots, \mathbf{b}_{2n}\}$ , labeled **a** and **b**. Let each node  $\mathbf{a}_i$  have a left child  $\mathbf{b}_i$ , a right child  $\mathbf{b}_{n+i}$ , and a middle child  $\mathbf{a}_{i+1}$  if  $i < n$ . Given the query  $\|\mathbf{a}/\mathbf{b}$ , each node  $\mathbf{a}_i$  is part of two matches, one with  $\mathbf{b}_i$  and one with  $\mathbf{b}_{n+i}$ , but to find these two matches,  $2n - 2i$  useless **b**-nodes must be traversed in the descendant interval. This gives a total enumeration cost of  $\Omega(n^2)$ .

The following theorem formalizes properties of the intermediate result storage in Twig<sup>2</sup>Stack that are key to its optimality.

**THEOREM 1 (LINEAR TIME RESULT ENUMERATION [3]).** The result set  $O$  can be enumerated in  $\Theta(O)$  time if (i) the data nodes  $d$  such that  $\text{root}(Q) \mapsto d$  is part of a strict full match can be found in time linear in their number, and (ii) given a pair  $q \mapsto q'$ , for each child  $c$  of  $q$ , the pairs  $c \mapsto c'$  that are part of a strict subtree match for  $q$  together with  $q \mapsto q'$  can be enumerated in time linear in their number.

### 3.4 Full Matches

When different types of filtering strategies are combined, it may be interesting to know when additional filtering passes will not remove more nodes.

**THEOREM 2 (FULL MATCH).** (i) A pair  $q \mapsto q'$  is part of a full match iff (ii) it is part of a prefix path match that only uses pairs that are part of subtree matches.

**PROOF SKETCH.** ( $i \Rightarrow ii$ ) Follows from Lemma 1. ( $i \Leftarrow ii$ ) Let  $\mathfrak{M} = \langle Q, D, I \rangle$  be the initial matching problem, and  $\mathfrak{M}' = \langle Q, D, I' \rangle$  be the matching problem where  $I'$  is the set of pairs that are part of subtree matches in  $\mathfrak{M}$ . The theorem is true for pairs with the query root, as for the query root a subtree match is a full match. Assume that there is a prefix path match  $M_{\downarrow q} \ni q \mapsto q'$  for  $q$  in  $\mathfrak{M}'$ , and that  $p$  is the parent of  $q$ . By construction,  $M_{\downarrow q} \ni p \mapsto p'$  is also a prefix path match for  $p$ . We use induction on the query node depth, and prove that if  $p \mapsto p'$  is part of a full match  $M_p$  for  $p$ , then  $q \mapsto q'$  must be part of a full match for  $q$ . Let  $Q_q$  be the subtree rooted at  $q$ , and  $Q_p = Q \setminus Q_q$ . Let  $M'_p = M_p \setminus \{r \mapsto r' \mid r \in Q_q\}$ . By the assumption  $q \mapsto q' \in I'$ , there exists a subtree match  $M_{\uparrow q} \ni q \mapsto q'$  for  $q$ . Then the node map  $M_q = M'_p \cup M_{\uparrow q}$  is a full match for  $q$ , because (1)  $p \mapsto p' \in M'_p$  and  $q \mapsto q' \in M_{\uparrow q}$  must satisfy the edge  $\langle p, q \rangle$  as they are used together in  $M_{\downarrow q}$ , and (2)  $Q_p$  and  $Q_q$  can be matched independently when the mapping of  $p$  and  $q$  is fixed.  $\square$

In other words, if nodes are filtered first in postorder on strictly matched subtrees, and then in preorder on strictly matched prefix paths, the intermediate result set contains *only* data nodes that are part of the *final* result. The opposite is not true: In the example in Figure 1, the pair  $c \mapsto c_3$  would not be removed if strict prefix path filtering was followed by strict subtree match filtering.

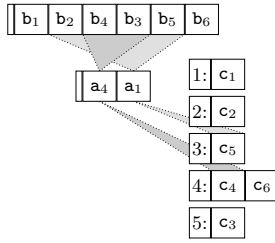
Note that strict full match filtering is not necessary for optimal enumeration by Theorem 1, and that the optimal algorithms we present in the following do not use it. They use prefix path match filtering followed by subtree match filtering, where the former is only used to speed up practical performance. On the other hand, the input stream merge component we introduce in Section 4.4 gives inexpensive weak full match filtering.

## 4. FAST OPTIMAL TWIG JOINS

In this section we create an algorithmic framework that permits any combination of preorder and postorder filtering. First we introduce a new data structure that enables strict subtree match checking and linear result enumeration.

### 4.1 Level Split Vectors

A key to the practical performance of TwigList and TwigFast is the storage of intermediate nodes in simple vectors [11], but this scheme makes it hard to improve worst-case behavior.



**Figure 6: Intermediate data structures using level split vectors and strict subtree match filtering. As opposed to in Figure 4(b),  $a_2$  does not satisfy.**

To enable direct access to usable child matches below both A–D and P–C edges, we *split* the intermediate result vectors for query nodes below P–C edges, such that there is one vector for each data tree level observed, as shown in Figure 6. Given a node in the intermediate results, matches for a child query node below an A–D edge can be found in a regular descendant interval, while the matches for a child query node below a P–C edge can be found in a *child interval* in a child vector. This vector can be identified by the depth of the parent data node plus one.

In the following we assume that level split vectors can be accessed by level in amortized constant time. This is true if level split vectors are stored in dynamic arrays, and the depth of the deepest data node is asymptotically bounded by the size of the input, that is,  $d \in \mathcal{O}(I)$ . If this bound does not hold, which can be the case when  $|I| \ll |D|$ , expected linear performance can be achieved by storing level split vectors in hash maps.

When nodes are added to the intermediate results in postorder, checking for non-empty descendant and child intervals inductively implies strict subtree match filtering. This is illustrated for our example in Figure 6. As each check takes constant time, the intermediate results can be constructed in  $\Theta(I)$  time, as for TwigList [11]. Result enumeration with this data structure is a trivial recursive iteration through nodes and their child and descendant intervals, which is almost identical to the enumeration in TwigList. The difference is that no extra check is necessary for P–C relationships, and that the result enumeration is  $\Theta(O)$  by Theorem 1 when strict subtree match filtering is applied.

## 4.2 The TJStrictPost Algorithm

Algorithm 1 shows the general framework we use for postorder construction of intermediate results, extending algorithms like TwigList [11]. It allows using any combinations of the preorder and postorder checks described in Section 3, from none to weak and strict, and allows using either simple vectors or level split vectors. A global stack is used to maintain the set of open data nodes, and if prefix path matching is implemented, a local stack for each query node is maintained in parallel. The input stream merge component used is a priority queue implemented with a binary heap. The postorder storage approach used here requires global ordering, and cannot read local preorder input (see Appendix E).

The correctness of Algorithm 1 follows from the correctness of the filtering strategies described in Sections 3.1 and 3.2, and the correctness of TwigList [11], with the enumeration algorithm trivially extended to use child intervals when level split vectors are used.

---

### Algorithm 1 Postorder construction.

---

```

While  $\neg$ EOF:
  Read next  $q \mapsto d$ .
  While non-ancestors of  $d$  on global stack:
    Pop  $q' \mapsto d'$  from global and local stack.
    If  $q' \mapsto d'$  satisfies postorder checks:
      Set interval end index for  $d'$ 
      in the vector of each child of  $q'$ .
      Add  $d'$  to intermediate results for  $q'$ .
  If  $d$  satisfies preorder checks:
    For each child of  $q$ , set interval start index for  $d$ .
    Push  $q \mapsto d$  on global stack.
    Push  $d$  on local stack for  $q$ .
Clean remaining nodes from the stacks.
Enumerate results.

```

---

We now define the TJStrictPost algorithm, which builds on this framework. Detailed pseudocode can be found in Appendix A. The algorithm uses level split vectors, and, as opposed to the previous twig join algorithms listed in Figure 5, it includes *strict* checking of both matched prefix paths and subtrees. The former is implemented by checking the top data node on the local stack of the parent query node, while the latter is implemented by checking for non-empty child and descendant intervals. A  $\Theta(I + O)$  running time follows from the discussion in Section 4.1.

#### 4.2.1 A note on TwigList:

As noted in the original description, chaining nodes with the same tree level into linked lists inside descendant intervals can improve practical performance in TwigList [11]. However, as the first child match with the correct level must still be searched for, further changes are needed to achieve linear worst-case evaluation. This can be implemented by maintaining a vector for each query node with the *previous* match on each tree level at any time. A node must then be given pointers to such previous matches as it is pushed on stack in TwigList. When the node is popped off stack, it can then be checked if any children have been found, and intermediate results can be enumerated in linear time, assuming that  $d \in \mathcal{O}(I)$ , as for our solution.

## 4.3 The TJStrictPre Algorithm

Algorithm 2 shows the general framework we use to construct intermediate results in preorder, extending algorithms like TwigFast [10]. It supports any combination of preorder and postorder filtering, simple or level split vectors, and input in global or local preorder. As opposed to with postorder storage, nodes are inserted directly into intermediate result vectors after they have passed a prefix path check. Local stacks store references to open nodes in the intermediate results. If strict subtree match filtering is required, or weak subtree match filtering is not implied by the input stream merger, intermediate results are filtered bottom-up in a post-processing pass.

TwigFast is reported to have faster average case query evaluation than previous twig joins [10], and we hope to match this performance in a worst-case linear algorithm. The TJStrictPre algorithm is similar to TJStrictPost, and uses strict checking of prefix paths and subtrees, and stores intermediate results in level split vectors. See detailed pseudocode in Appendix B. TJStrictPre uses the getPart in-

---

**Algorithm 2** Preorder construction.

---

```
While  $\neg$ EOF:
  Read next  $q \mapsto d$ .
  For the stack of both  $q$ 's parent and  $q$  itself:
    Pop non-ancestors of  $d$ ,
    and set their end indexes.
  If  $d$  satisfies preorder checks:
    For each child of  $q$ , set interval start index for  $d$ .
    Add  $d$  to intermediate results for  $q$ .
    Push reference to  $d$  on stack for  $q$ .
Clean stacks.
Clean intermediate results with postorder checks.
Enumerate results.
```

---

put stream merger, which is an improvement of getNext described in Section 4.4. If the post-processing pass can be performed in linear time, then the algorithm can evaluate twig queries in  $\Theta(I + O)$  time, by the same argument as for TJStrictPost.

The filtering pass is implemented by cleaning intermediate result vectors bottom-up in the query, in-place overwriting data nodes not satisfying subtree matches, as described in detail in Appendix D. The indexes of kept nodes are stored in a separate vector for each query node, and are used to translate old start and end indexes into new positions. To achieve linear traversal, the intermediate result vector of a node is traversed in parallel with the index vectors of the children after they have been cleaned. For level split vectors, there is one separate index vector per used level, and a separate vector iterator is used per level when the parent is cleaned. Also, there is an array giving the level of each stored data node in preorder, such that split and non-split child vectors can then be traversed in parallel. Start values are updated as nodes are pushed onto a stack in preorder, while end values are updated as nodes are popped off in postorder.

#### 4.4 The getPart Input Stream Merger

The getNext input stream merge component implements weak subtree match filtering in  $\Theta(I)$  time, and is used to improve practical performance in many current algorithms using preorder storage [8, 10]. Assume in the following discussion that there is one preorder stream of label-matching data nodes associated with each query node. The input stream merger repeatedly returns pairs containing a query node and the data node at the head of its stream, implementing Comp. 1 in Figure 2.

The getNext function processes the query bottom-up to find a query node that satisfies the following three properties: (1) when its stream is forwarded at least until its head follows the heads of the streams of the children in postorder, it still precedes them in preorder, (2) all children satisfy properties 1 and 2, and (3) if there is a parent, it does not satisfy 1 and 2. Property 2 implies that weak subtree filtering is achieved, and Property 3 implies that local preorder by Definition 3 is achieved.

The procedure is efficient if leaf query nodes have relatively few matches, which can be the case in practice in XML search when all query leaf nodes are selective text value predicates. However, if the internal query nodes are more selective than the leaf nodes, or if not all leaves are selective, the overhead of using the getNext function may

outweigh the benefits.

To improve practical performance we introduce the *getPart* function, which requires the following property in addition to the above three: (4) if there is a parent, then the current head of stream is a descendant of a data node that was the head of stream for the parent in some previous subtree match for the parent. This inductively implies that nodes returned are also weak prefix path matches, and from the ordering of the filtering steps, the result is weak *full* match filtering by Theorem 2. To allow forwarding streams to find such nodes, the algorithm can no longer be stateless, as shown by the following example:

*EXAMPLE 3. Assume that the heads of the streams for query nodes  $\mathbf{a}^1$  and  $\mathbf{b}^1$  in Figure 1 are  $\mathbf{a}_3$  and  $\mathbf{b}_2$ , respectively. Then it cannot be known by only inspecting heads of streams whether or not any usable ancestors of  $\mathbf{b}_2$  were seen before  $\mathbf{a}_3$ , and  $\mathbf{b}_2$  must be returned regardlessly.*

Property 4 is implemented in getPart by maintaining for each query node, the data node *latest* in the tree *postorder* that has been part of a weak full match. This value is updated when a query node is found to satisfy all four properties. To ensure progress, streams are forwarded top-down in the query to match the stored value or the current head for the parent node. Note that multiple top-down and bottom-up passes may be needed to find a satisfying node, but each such pass forwards at least one stream past useless matches. See detailed pseudocode in Appendix C.

## 5. EXPERIMENTS

The following experiments explore the effects of weak and strict matching of prefix paths and subtrees, different input stream merge functions, and level split vectors.

We have used the DBLP, XMark and Treemark benchmark data, and run the commonly used DBLP queries from the PRIX paper [12], the XMark queries from the XPathMark suite part A [5] (except queries 7 and 8, which are not twigs), and the Treemark queries from the TwigList paper [11]. In addition, we have created some artificial data and queries. Details can be found in Appendix F. The experiments were run on a computer with an AMD Athlon 64 3500+ processor and 4 GB of RAM. All queries were warmed up by 3 runs and then run 100 times, or until at least 10 seconds had passed, measuring average running-time.

All algorithms are implemented in C++, and features are turned on or off at compile time to make sure the overhead of complex methods does not affect simpler methods. Feature combinations are coded with 5 letter tags. We use **Heap**, getNext and getPart for merging the input streams, and store intermediate results in prEorder or pOstorder. We use no (-), **Weak** or **Strict** prefix path match filtering, and no (-), **Weak** or **Strict** subtree match filtering. Intermediate results are stored in simple vectors (-) or Level split vectors. The previous algorithms TwigList and TwigFast are denoted by HO-W- and NEWW-, respectively, while TJStrictPost and TJStrictPre are denoted by HOSSL and PESSL. Note that filtering checks are not performed in intermediate result construction if the given filtering level is already achieved by the input stream merger. Strict subtree match filtering is implemented by descendant interval traversal when not using level split vectors. With preorder storage an extra filtering pass is used to implement subtree match filtering. Worst-case optimal algorithms match the pattern **\*\*SL**.

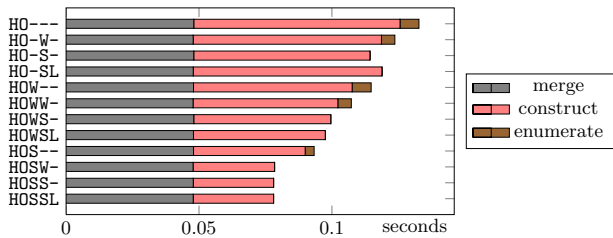


Figure 7: Query `//annotation/keyword` on XMark data. Cost divided into merging input, building intermediate results, and result enumeration.

We present no performance comparisons with the Twig<sup>2</sup>Stack algorithm because it does not fit into our general framework. Getting an accurate and fair comparison would be an exercise in independent program optimization. TwigList is previously reported to be 2–8 times faster than Twig<sup>2</sup>Stack for queries similar to ours [11].

### 5.1 Checked Paths and Subtrees

Figure 7 shows results for running the XMark query `//annotation/keyword`, with cost divided into different components and phases. Filtering lowers the cost of building intermediate data structures because their size is reduced, and the cost of enumerating results because redundant traversal is avoided. Note that this query was chosen because it shows the potential effects of prefix path and subtree match filtering, and it may not be representative.

Figure 8 shows the effects of prefix path vs. subtree match filtering averaged over all queries on DBLP, XMark and Treebank. Heap input stream merging was used because it allows all filtering levels, and postorder storage was used to avoid extra filtering passes. Each timing has been normalized to 1 for the fastest method for each query. Raw timings are listed in Appendix G. As opposed to in Figure 7, there is on average little difference between the methods using at least *some* filtering both in preorder and postorder. The benefits of asymptotic constant time checks when using level split vectors seems to be outweighed by the cost of maintaining and accessing them, but only by a small margin.

### 5.2 Reading Input

Figure 9 shows the effect of using different input stream mergers. The labels in the artificial data tree used are Zipf

		Prefix path							
		-		W		S			
		avg	max	avg	max	avg	max		
Subtree	--	1.84	3.5	1.29	1.61	1.23	1.62	1.45	3.5
	W-	1.24	1.45	1.03	1.13	1.01	1.06	1.09	1.45
	S-	1.24	1.46	1.03	1.10	1.02	1.08	1.10	1.46
	SL	1.32	1.55	1.09	1.19	1.06	1.20	1.16	1.55
		1.41	3.5	1.11	1.61	1.08	1.62	1.20	3.5

Figure 8: The effect of parent match filtering. vs. child match filtering. Running DBLP, XPathMark and Treebank queries. Normalizing query times to 1 for the fastest method for each query. Showing arithmetic mean and maximum for normalized time.

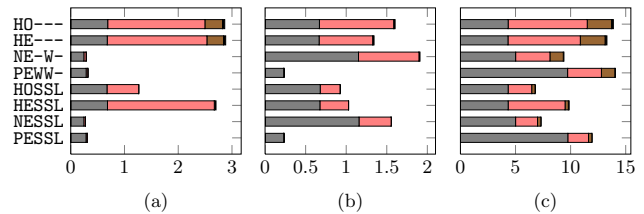


Figure 9: Running queries on Zipf data. (a) Selective leaves. (b) Selective internal nodes. (c) No selective nodes.

distributed ( $s = 1$ ), with a, b, y and z being the labels on 30%, 13%, 1.0% and 1.0% of the nodes, respectively. The data and queries were chosen to shed light on both the benefits and the possible overhead of using the advanced input methods.

For the first query, `//a/b[y][z]`, the leaves are very selective, and both `getNext` and `getPart` very efficiently filter away most of the nodes. The input stream merging is slightly more efficient for the simpler `getNext`. In the second query, `//y/z[a][b]`, the internal nodes are selective, while the leaves are not. Here `getPart` efficiently filters away many nodes, while `getNext` does not, making it even slower than the simple heap, due to the additional complexity. The third query shows a case where `getPart` performs worse than both the other methods. In this query, `//a[a][a][a][a][a]`, all query nodes have very low selectivity, and are equally probable. The filtering has almost no effect, and only causes overhead. Note the cost difference between `HOSSL` and `HESSL`, which is due to the additional filtering pass over the large intermediate results.

### 5.3 Combined Benefits

Figure 10 shows the effects of combining different input stream mergers and additional filtering strategies. The same queries as in Figure 8 are evaluated, and the first column shows the same tendencies: There is not much difference between the strategies as long as you do at least weak match filtering on both prefix path and subtree.

		Input stream merger									
		HO		HE		NE		PE			
		avg	max	avg	max	avg	max	avg	max		
Match filtering	---	7.0	33	6.6	30					6.8	33
	-W-	4.5	23	7.5	34	3.7	19			5.2	34
	-S-	4.5	23	7.5	34	3.7	18			5.2	34
	-SL	4.8	25	8.3	38	3.8	19			5.6	38
	W--	4.9	24	4.9	23					4.9	24
	W-	3.7	19	5.4	25	3.2	15	1.02	1.11	3.3	25
	WS-	3.7	19	5.4	26	3.2	15	1.04	1.15	3.3	26
	WSL	3.9	20	5.7	27	3.2	15	1.08	1.22	3.5	27
	S--	4.8	24	4.8	24					4.8	24
	SW-	3.7	19	5.2	26	3.2	15	1.03	1.12	3.3	26
	SS-	3.7	19	5.1	26	3.2	15	1.05	1.17	3.3	26
	SSL	3.9	20	5.5	27	3.2	15	1.05	1.20	3.4	27
		4.4	33	6.0	38	3.4	19	1.04	1.22	4.1	38

Figure 10: Input mergers vs. filtering strategies.

In the second column all methods using any subtree match filtering are more expensive, because with preorder storage, subtree match filtering is performed in a second pass over the intermediate results. A second pass is also used for subtree match filtering in the third and fourth columns, but in practice the getNext and getPart components have already filtered away more nodes, the intermediate results are smaller, and the second pass is less expensive.

Note the difference between using getNext and getPart. The new method is more than three times as fast on average, and is more than one order of magnitude faster for queries where only some of the leaf nodes are selective. The getPart function also fast forwards through useless matches for the leaves that are not selective, while getNext passes all leaf matches on to the intermediate result construction component. Also note that the maximum overhead of using PESSL, the fastest worst-case optimal method, is at most 20% in any benchmark query tested.

## 6. RELATED WORK

This work is based on the assumption that label-partitioning and simple streams is used. Orthogonal previous work investigates how the underlying data can be accessed. If the streams support *skipping*, both unnecessary I/O and computation can be avoided [7]. Our getPart algorithm, which is detailed in Appendix C, can be modified to use any underlying skipping technology by changing the implementation of FwdToAncOf() and FwdToDescOf(). *Refined partitioning* schemes with structure indexing can be used to reduce the number of data nodes read for each query node [4,9]. Our twig join algorithms are independent of the partitioning scheme used, assuming multiple partition blocks matching a single query node are merged when read. Another technique is to use a node encoding that allows reconstruction of data node ancestors, and use *virtual streams* for the internal query nodes [14]. Our getPart algorithm could be changed to generate virtual internal query node matches from leaf query node matches, as complete query subtrees are always traversed. For a broader view on XML indexing see the survey by Gou and Chirkova [6]. XPath queries can be rewritten to use only the axis *self*, *child*, *descendant*, and *following* [16]. To add support for support for the *following*-axis, we would have to add additional logic for how to forward streams, and modify the data structures to store start indexes for the new relationship.

## 7. CONCLUSIONS AND FUTURE WORK

In this paper we have shown how worst-case optimality and fast evaluation in practice can be combined in twig joins. We have performed experiments that span out and extend the space of the fastest previous solutions. For common benchmark queries our new and worst-case optimal algorithms are on average three times as fast as earlier approaches. Sometimes they are more than an order of magnitude faster, and they are never more than 20% slower.

In future work we would like to combine the new techniques with previous orthogonal techniques such as skipping, refined partitioning and virtual streams. Also, it would be interesting to see an elegant worst-case linear algorithm reading local preorder input and producing preorder sorted results, that does not perform a post-processing pass over the data, and does not need the assumption  $d \in \mathcal{O}(I)$ .

**Acknowledgments.** This material is based upon work supported by the iAd Project funded by the Research Council of Norway, and the Norwegian University of Science and Technology. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors, and do not necessarily reflect the views of the funding agencies.

## 8. REFERENCES

- [1] Radim Bača, Michal Krátký, and Václav Snášel. On the efficient search of an XML twig query in large DataGuide trees. In *Proc. IDEAS*, 2008.
- [2] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: Optimal XML pattern matching. In *Proc. SIGMOD*, 2002.
- [3] Songting Chen, Hua-Gang Li, Junichi Tatemura, Wang-Pin Hsiung, Divyakant Agrawal, and K. Selçuk Candan. Twig<sup>2</sup>Stack: bottom-up processing of generalized-tree-pattern queries over XML documents. In *Proc. VLDB*, 2006.
- [4] Ting Chen, Jiaheng Lu, and Tok Wang Ling. On boosting holism in XML twig pattern matching using structural indexing techniques. In *Proc. SIGMOD*, 2005.
- [5] Massimo Franceschet. XPathMark web page. <http://sole.dimi.uniud.it/~massimo.franceschet/xpathmark/>.
- [6] Gang Gou and Rada Chirkova. Efficiently querying large XML data repositories: A survey. *Knowl. and Data Eng.*, 2007.
- [7] Haifeng Jiang, Wei Wang, Hongjun Lu, and Jeffrey Xu Yu. Holistic twig joins on indexed XML documents. In *Proc. VLDB*, 2003.
- [8] Zhewei Jiang, Cheng Luo, Wen-Chi Hou, and Qiang Zhu Dunren Che. Efficient processing of XML twig pattern: A novel one-phase holistic solution. In *Proc. DEXA*, 2007.
- [9] Raghav Kaushik, Philip Bohannon, Jeffrey F. Naughton, and Henry F. Korth. Covering indexes for branching path queries. In *Proc. SIGMOD*, 2002.
- [10] Jiang Li and Junhu Wang. Fast matching of twig patterns. In *Proc. DEXA*, 2008.
- [11] Lu Qin, Jeffrey Xu Yu, and Bolin Ding. TwigList: Make twig pattern matching fast. In *Proc. DASFAA*, 2007.
- [12] Praveen Rao and Bongki Moon. PRIX: Indexing and querying XML using Prüfer sequences. In *Proc. ICDE*, 2004.
- [13] Mirit Shalem and Ziv Bar-Yossef. The space complexity of processing XML twig queries over indexed documents. In *Proc. ICDE*, 2008.
- [14] Beverly Yang, Marcus Fontoura, Eugene Shekita, Sridhar Rajagopalan, and Kevin Beyer. Virtual cursors for XML joins. In *Proc. CIKM*, 2004.
- [15] Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. On supporting containment queries in relational database management systems. *SIGMOD Rec.*, 2001.
- [16] Ning Zhang, Varun Kacholia, and M. Tamer Özsu. A succinct physical storage scheme for efficient evaluation of path queries in XML. In *Proc. ICDE*, 2004.

## APPENDIX

This version of the article has fixed bugs from the original.

### A. TJSTRICTPOST PSEUDOCODE

Algorithm 3 shows more detailed pseudocode for the TJStrictPost algorithm described in Section 4.2. Positions are assumed to be encoded using BEL (*begin, end, level*) [15]. The function `GetVector( $q, level$ )` returns the regular intermediate result vector if  $q$  is below an A–D edge, or the split vector given by *level* if  $q$  is below a P–C edge.

---

#### Algorithm 3 TJStrictPost

---

```

1: function EvaluateGlobal():
2:   ( $c, q, d$ )  $\leftarrow$  MergedStreamHead()
3:   while  $c \neq \text{EOF}$ :
4:     ProcessGlobalDisjoint( $d$ )
5:     if Open( $q, d$ ):
6:       Push( $S_{global}, (d.end, q)$ )
7:       ( $c, q, d$ )  $\leftarrow$  MergedStreamHead()
8:       ProcessGlobalDisjoint( $\infty$ )

9: function ProcessGlobalDisjoint( $d$ ):
10:  while  $S_{global} \neq \emptyset \wedge \text{Top}(S_{global}).end < d.end$ :
11:    ( $\cdot, q$ )  $\leftarrow$  Pop( $S_{global}$ )
12:    Close( $q$ )

13: function Open( $q, d$ ):
14:  if CheckParentMatch( $q, d$ ):
15:     $u \leftarrow$  new Intermediate( $d$ )
16:    MarkStart( $q, u$ )
17:    Push( $S_{local}[q], u$ )
18:    return true
19:  else:
20:    return false

21: function Close( $q$ ):
22:   $u \leftarrow$  Pop( $S_{local}[q]$ )
23:  MarkEnd( $q, u$ )
24:  if CheckChildMatch( $q, u$ ):
25:    Append(GetVector( $q, u.d.level$ ),  $u$ )

26: function MarkStart( $q, u$ ):
27:  for  $r \in q.children$ :
28:     $u.start[r] \leftarrow$  GetVector( $r, u.d.level + 1$ ).size + 1

29: function MarkEnd( $q, u$ ):
30:  for  $r \in q.children$ :
31:     $u.end[r] \leftarrow$  GetVector( $r, u.d.level + 1$ ).size

32: function CheckParentMatch( $q, d$ ):
33:  if Axis( $q$ ) = “//”:
34:    return IsRoot( $q$ ) or  $S_{local}[\text{Parent}(q)] \neq \emptyset$ 
35:  else:
36:    if IsRoot( $q$ ): return  $d.level = 1$ 
37:    else: return  $S_{local}[\text{Parent}(q)] \neq \emptyset \wedge d.level =$ 
    Top( $S_{local}[\text{Parent}(q)].level + 1$ )

38: function CheckChildMatch( $q, u$ ):
39:  for  $r \in q.children$ :
40:    if  $u.end[r] < u.start[r]$ : return false
41:  return true

```

---

### B. TJSTRICTPRE PSEUDOCODE

Algorithm 4 shows more detailed pseudocode for the TJStrictPre algorithm described in Section 4.3.

### C. GETPART FUNCTION

Pseudocode for the `getPart` function is shown in Algorithm 5, where what is conceptually different from the previous `getNext` function is colored **dark blue**.

`getPart` forwards nodes both to catch up with the parent and child streams, whereas `getNext` only does the latter.

---

#### Algorithm 4 TJStrictPre

---

```

1: function EvaluateLocalTopDown():
2:   ( $c, q, d$ )  $\leftarrow$  MergedStreamHead()
3:   while  $c \neq \text{EOF}$ :
4:     if  $\neg \text{IsRoot}(q)$ :
5:       ProcessLocalDisjoint(Parent( $q$ ),  $d$ )
6:       ProcessLocalDisjoint( $q, d$ )
7:       Open( $q, d$ )
8:       ( $c, q, d$ )  $\leftarrow$  MergedStreamHead()
9:     for  $q \in Q$ :
10:      ProcessLocalDisjoint( $q, \infty$ )
11:      FilterPass( $q.root$ )

12: function ProcessLocalDisjoint( $q, d$ ):
13:  while Top( $S_{local}[q]$ ). $d.end < d.end$ :
14:    Close( $q$ )

15: function Open( $q, d$ ):
16:  if CheckParentMatch( $q, d$ ):
17:     $u \leftarrow$  new Intermediate( $d$ )
18:     $V \leftarrow$  GetVector( $q, d.level$ )
19:    if  $\neg \text{IsLeaf}(q)$ :
20:      MarkStart( $q, u$ )
21:      Push( $S_{local}[q], (V, V.size)$ )
22:      Append( $V, u$ )
23:      return  $\neg \text{IsLeaf}(q)$ 
24:    else:
25:      return false

26: function Close( $q$ ):
27:  if  $\neg \text{IsLeaf}(q)$ :
28:    ( $V, i$ )  $\leftarrow$  Pop( $S_{local}[q]$ )
29:    MarkEnd( $q, V[i]$ )

```

---

The `getNext` algorithm is completely stateless, and only inspects stream heads. When a match for a query subtree is found, the stream for the subtree root node is read and forwarded. Then it is not possible to know in the next call on this point in the query, whether the child subtrees were once part of a match or not. In the `getPart` function we save one extra value per query node, stored in the  $M$  array, namely the latest match in the tree postorder which was part of a weak match for the *entire query*. When considering a query subtree, the currently interesting data nodes are those that are either part of a match using a previous head in the parent stream, or part of a new match using the current head in the parent stream (see Lines 9–13).

The forwarding of streams based on child stream heads is very similar to in `getNext` (Lines 17–28). Unless the search is short-circuit (Line 22), the stream is forwarded at least until the head is an ancestor of all the child heads (Line 28). The query node itself is returned if an ancestor of the child heads was found, and unless the previous  $M$  value is an ancestor of the current head, it is updated. When a child query node is returned, it is known whether or not it is part of a match.<sup>3</sup>

### D. EXTRA FILTERING PASS

Algorithm 6 gives pseudocode for the extra filtering pass used to obtain strict subtree match filtering when using pre-order storage in TJStrictPre.

During the clean-up (Line 1-15), nodes failing checks are overwritten, and it is stored in the  $C$  vectors which values were not dropped. The query nodes are visited bottom-up by the `FilterPass` function, updating the vectors of one query node at a time, based on the cleanup in non-leaf child

<sup>3</sup>Thanks to Radim Bača for pointing out an error in the original published pseudocode, where Lines 30–31 in Algorithm 5 were different.

---

**Algorithm 5** GetPart

---

```
1: function MergedStreamHead():
2:   while true:
3:      $(c, d, q) \leftarrow \text{GetPart}(Q.\text{root})$ 
4:     if  $c \neq \text{MISMATCH}$ :
5:       if  $c \neq \text{EOF}$ :
6:         Fwd( $q$ )
7:       return  $(c, d, q)$ 

8: function GetPart( $q$ ):
9:   if  $\neg \text{IsRoot}(q)$ :
10:     $p \leftarrow \text{Parent}(q)$ 
11:    FwdToDescOf( $q, M[p]$ )
12:    if  $\neg \text{Eof}(q) \wedge \neg \text{Eof}(p) \wedge M[p].\text{end} < H(q).\text{end}$ :
13:      FwdToDescOf( $q, H(p)$ )
14:    if  $\text{IsLeaf}(q)$ :
15:      if  $\text{Eof}(q)$ : return  $(\text{EOF}, q, \perp)$ 
16:      else: return  $(\text{MATCH}, q, H(q))$ 
17:     $(d_{\min}, q_{\min}) \leftarrow (\infty, \perp)$ ;  $(d_{\max}, q_{\max}) \leftarrow (0, \perp)$ 
18:    for  $r \in q.\text{children}$ :
19:       $(c_r, d_r, q_r) \leftarrow \text{GetPart}(r)$ 
20:      if  $c_r \neq \text{EOF}$ :
21:        if  $c_r = \text{MISMATCH}$ : flag  $\text{MISMATCH}$ 
22:        elif  $q_r \neq r$ : return  $(\text{MATCH}, d_r, q_r)$ 
23:        if  $d_r.\text{begin} < d_{\min}.\text{begin}$ :  $(d_{\min}, q_{\min}) \leftarrow (d_r, q_r)$ 
24:        if  $d_r.\text{begin} > d_{\max}.\text{begin}$ :  $(d_{\max}, q_{\max}) \leftarrow (d_r, q_r)$ 
25:      else:
26:        FwdToEof( $q$ )
27:    if  $q_{\min} = \perp$ : return  $(\text{EOF}, \perp, q)$ 
28:    FwdToAncOf( $q, d_{\max}$ )
29:    if flagged  $\text{MISMATCH}$ :
30:      return  $(\text{MISMATCH}, \perp, q_r)$ 
31:
32:    if  $\neg \text{Eof}(q) \wedge H(q).\text{begin} < d_{\min}.\text{begin}$ :
33:      if  $\text{IsRoot}(q) \vee H(q).\text{end} < M[p].\text{end}$ :
34:        if  $M[q].\text{end} < H(q).\text{end}$ :  $M[q] \leftarrow H(q)$ 
35:        return  $(\text{MATCH}, H(q), q)$ 
36:      else:
37:        if  $d_{\min}.\text{begin} < M[q].\text{end}$ :
38:          return  $(\text{MATCH}, d_{\min}, q_{\min})$ 
39:        else:
40:          if  $\text{Eof}(q)$ : return  $(\text{EOF}, \perp, q)$ 
41:          else: return  $(\text{MISMATCH}, \perp, q)$ 

42: function FwdToEof( $q$ ):
43:   while  $\neg \text{Eof}(q)$ : Fwd( $q$ )

44: function FwdToDescOf( $q, d$ ):
45:   while  $\neg \text{Eof}(q) \wedge H(q).\text{begin} \leq d.\text{begin}$ : Fwd( $q$ )

46: function FwdToAncOf( $q, d$ ):
47:   while  $\neg \text{Eof}(q) \wedge H(q).\text{end} \leq d.\text{begin}$ : Fwd( $q$ )
```

---

query nodes. The FilterPass and FilterPassPost functions go through all data nodes in preorder and postorder respectively, updating interval start and end indexes.

The AllNodes call returns a special iterator to all intermediate data nodes for a query node, sorted in total preorder. For query nodes with an incoming P-C edge and level split vectors, the order in which nodes were inserted on different levels was recorded during construction in TJStraightPre. Details are omitted in Algorithm 4, where an extra statement must be added after line 22, storing a reference to the used vector.

The FwdIter function contains the logic for updating the start and end indexes. Each query node has an iterator for each vector, which is utilized when traversing the matches for the parent query node. In essence, the segments of child and descendant intervals which contain references to nodes which were not saved during a cleanup pass are discarded.

## E. GETNEXT AND POSTORDER

Many algorithms use the getNext function [2] for merging

---

**Algorithm 6** FilterPass

---

```
1: function CleanUp( $q$ ):
2:   if  $\text{Axis}(q) = \text{"/"}$ :
3:     CleanUpVector( $\text{GetVector}(q, \cdot), C[q]$ )
4:   else:
5:     for  $h \in \text{used levels}$ :
6:       CleanUpVector( $\text{GetVector}(q, h), C_h[q]$ )

7: function CleanUpVector( $V, C$ ):
8:    $i \leftarrow j \leftarrow 0$ 
9:   while  $i < V.\text{size}$ :
10:    if  $\text{CheckChildMatch}(q, V[i])$ :
11:       $V[j] \leftarrow V[i]$ 
12:      Append( $C, i$ )
13:       $j \leftarrow j + 1$ 
14:     $i \leftarrow i + 1$ 
15:   Resize( $V, j$ )

16: function FilterPass( $q$ ):
17:   if  $\text{IsLeaf}(q)$ : return
18:   for  $r \in q.\text{children}$ :
19:     FilterPass( $r$ )
20:   if  $\text{NonLeafChildren}(q) \neq \emptyset$ :
21:     for  $u \in \text{AllNodes}(q)$ :
22:       FilterPassPost( $q, u, d$ )
23:     for  $r \in \text{NonLeafChildren}(q)$ :
24:        $u.\text{start}[r] \leftarrow \text{FwdIter}(r, u.\text{start}[r], u, d)$ 
25:     Push( $S_{\text{local}}[q], u$ )
26:     FilterPassPost( $q, \infty$ )
27:   CleanUp( $q$ )

28: function FilterPassPost( $q, d$ ):
29:   while  $S_{\text{local}}[q] \neq \emptyset \wedge \text{Top}(S_{\text{local}}[q]).\text{end} < d.\text{end}$ :
30:      $u \leftarrow \text{Pop}(S_{\text{local}}[q])$ 
31:     for  $r \in \text{NonLeafChildren}(q)$ :
32:        $u.\text{end}[r] \leftarrow \text{FwdIter}(r, u.\text{end}[r], u, d)$ 

33: function FwdIter( $q, pos, d$ ):
34:   if  $\text{Axis}(q) = \text{"/"}$ :
35:     while  $I[q] < C[q].\text{size} \wedge C[q][I[q]] < pos$ :
36:        $I[q] \leftarrow I[q] + 1$ 
37:     return  $I[q]$ 
38:   else:
39:      $h \leftarrow d.\text{level} + 1$ 
40:     while  $I_h[q] < C_h[q].\text{size} \wedge C_h[q][I_h[q]] < pos$ :
41:        $I_h[q] \leftarrow I_h[q] + 1$ 
42:     return  $I_h[q]$ 
```

---

the input streams instead of a heap or linear scan [8, 10], because it cheaply filters away many useless nodes by implementing weak subtree match filtering. In this Appendix we show why using getNext with postorder intermediate result construction gives problems regardless of whether local or global stacks are used.

The getNext function does not return the data nodes in strict preorder, as assumed in the correctness proof for the TwigMix algorithm [10], but in local preorder (see Definition 3). As explained in Section 3.1, the top-down algorithms using getNext maintain one stack or equivalent structure for each internal query node. When a new match for a given query node is seen, the typical strategy [2] is popping non-ancestor nodes off the parent query node's stack, and the query node's own stack.

If a global stack is combined with using getNext input, errors may occur, as for the example query and data in Figure 11. With local preorder the ordering between the nodes not related through ancestry is not fixed. Assume that the ordering is

$$\langle \mathbf{a}^1 \mapsto \mathbf{a}_1, \mathbf{a}^1 \mapsto \mathbf{a}_2, \mathbf{b}^1 \mapsto \mathbf{b}_1, \mathbf{c}^1 \mapsto \mathbf{c}_1, \mathbf{d}^1 \mapsto \mathbf{d}_1, \mathbf{c}^1 \mapsto \mathbf{c}_2, \mathbf{e}^1 \mapsto \mathbf{e}_1 \rangle.$$

Then  $\mathbf{c}^1 \mapsto \mathbf{c}_2$  will pop  $\mathbf{a}^1 \mapsto \mathbf{a}_2$  off stack before  $\mathbf{e}^1 \mapsto \mathbf{e}_1$  is

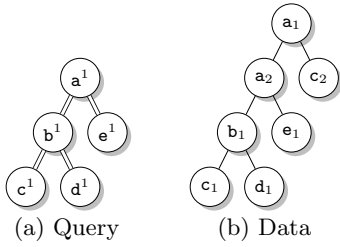


Figure 11: Problematic case with local preorder input and postorder storage.

Name	Size	Nodes
DBLP	676 MB	35 900 666
XMark	1 118 MB	32 298 988
Treebank	83 MB	3 829 512

Figure 12: Benchmark datasets used in experiments.

observed, and  $e^1 \mapsto e_1$  will never be added as a descendant of  $a^1 \mapsto a_2$ .

But using local stacks and a bottom-up approach also gives errors, because data nodes are added to the intermediate structures as they are popped off stack when using postorder storage, which is too late when using getNext and local stacks. If the typical approach is used,  $c^1 \mapsto c_2$  will only pop  $b_1$  off the stack of  $b^1$  and  $c_1$  off the stack of  $c^1$ . Then  $d^1 \mapsto d_1$  will never be added to the child structures of  $b^1 \mapsto b_1$ , because it is popped to late.

It may be possible to modify the local stack approach to work with postorder storage and getNext input, but this would require carefully popping nodes on ancestor and descendant stacks in the right order.

## F. BENCHMARK DATA AND QUERIES

Figure 12 gives some details on the benchmark data used in our experiments, and Figure 13 lists the queries we have used.

## G. EXTENDED RESULTS

Figure 14 shows the timings that the aggregates in Section 5 are based on.

## H. BAD BEHAVIOR

In this appendix we list some experiments showing the super-linear behavior of previous twig join algorithms.

Figure 15(a) shows the *exponential* behavior of TwigList and TwigList (HO-W-) and TwigFast (NEWW-) with the data and query from Example 1. The data is  $(\alpha_1)^n \dots (\alpha_m)^n \beta / \gamma$  with  $m = 10$  and  $n = 100$ , and the query is  $\| \alpha_1 \| \dots \| \alpha_k \| / \gamma$ , with  $k$  varying from 1 to 7.

Figure 15(b) shows the results of an experiment based on Example 2 with varying  $n = 10\,000$ . The simple query  $\| a/b$  has quadratic cost even when strict prefix path and subtree match filtering is used, if P-C child matches are not directly

#	data	Hits	Source
D1	DBLP	6	[12]
	<code>//inproceedings[author/text()='Jim Gray']</code>		
	<code>[year/text()='1990']/@key</code>		
D2	DBLP	21	[12]
	<code>//www[editor]/url</code>		
D3	DBLP	13	[12]
	<code>//book/author[text()='C. J. Date']</code>		
D4	DBLP	2	[12]
	<code>//inproceedings[title/text()='</code>		
	<code>"Semantic Analysis Patterns.']/author</code>		
X1	XMark	40 726	[5]
	<code>/site/closed_auctions/closed_auction</code>		
	<code>/annotation/description/text/keyword</code>		
X2	XMark	124 843	[5]
	<code>//closed_auction//keyword</code>		
X3	XMark	124 843	[5]
	<code>/site/closed_auctions/closed_auction//keyword</code>		
X4	XMark	40 726	[5]
	<code>/site/closed_auctions/closed_auction</code>		
	<code>[annotation/description/text/keyword]/date</code>		
X5	XMark	124 843	[5]
	<code>/site/closed_auctions/closed_auction</code>		
	<code>[../keyword]/date</code>		
X6	XMark	32 242	[5]
	<code>/site/people/person[profile/gender]</code>		
	<code>[profile/age]/name</code>		
T1	Treebank	1 183	[11]
	<code>//S/VP//PP[../NP/VBN]/IN</code>		
T2	Treebank	152	[11]
	<code>//S/VP/PP[IN]/NP/VBN</code>		
T3	Treebank	381	[11]
	<code>//S/VP//PP[../NN][../NP[../CD]/VBN]/IN</code>		
T4	Treebank	1 185	[11]
	<code>//S[../VP][../NP]/VP/PP[IN]/NP/VBN</code>		
T5	Treebank	94 535	[11]
	<code>//EMPTY[../VP/PP//NNP][../S[../PP//JJ]/VBN]</code>		
	<code>//PP/NP//_NONE_</code>		

Figure 13: Benchmark queries used in experiments.

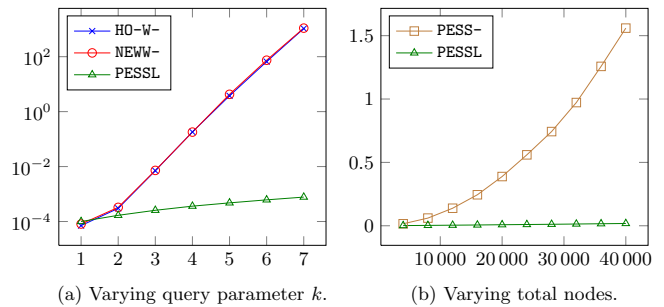


Figure 15: (a) Exponential behavior without strict matching. (b) Quadratic behavior without optimal enumeration. Query time in seconds.

accessible. Many of the  $a$  nodes in the data are nested, and have a small number of  $b$  children, but a large number of  $b$  descendants. For approaches using simple vectors, overlapping descendant intervals are scanned for direct children, and this results in quadratic running time.

	D1	D2	D3	D4	X1	X2	X3	X4	X5	X6	T1	T2	T3	T4	T5
HO---	2.53	0.43	1.07	1.59	0.57	0.11	0.11	0.75	0.25	0.25	0.31	0.30	0.40	0.47	0.50
HO-W-	1.42	0.25	0.44	1.12	0.43	0.10	0.11	0.60	0.24	0.21	0.18	0.17	0.24	0.32	0.32
HO-S-	1.42	0.26	0.44	1.11	0.43	0.10	0.11	0.61	0.24	0.21	0.17	0.18	0.24	0.32	0.33
HO-SL	1.70	0.28	0.48	1.22	0.46	0.10	0.11	0.65	0.26	0.23	0.18	0.19	0.24	0.34	0.33
HOW--	1.96	0.31	0.32	1.18	0.34	0.08	0.09	0.49	0.19	0.25	0.22	0.22	0.32	0.42	0.42
HOWW-	1.26	0.19	0.33	0.92	0.32	0.08	0.08	0.46	0.19	0.21	0.16	0.17	0.22	0.31	0.30
HOWS-	1.34	0.19	0.32	0.91	0.31	0.08	0.08	0.45	0.19	0.21	0.16	0.16	0.21	0.30	0.32
HOWSL	1.31	0.20	0.31	0.96	0.31	0.09	0.09	0.46	0.20	0.23	0.17	0.17	0.23	0.33	0.32
HOS--	2.03	0.31	0.32	1.17	0.32	0.08	0.08	0.47	0.19	0.25	0.21	0.19	0.27	0.35	0.40
HOSW-	1.27	0.20	0.31	0.91	0.30	0.08	0.08	0.44	0.19	0.21	0.16	0.15	0.21	0.29	0.29
HOSS-	1.27	0.21	0.32	0.92	0.30	0.08	0.08	0.44	0.19	0.21	0.17	0.15	0.21	0.30	0.30
HOSSL	1.32	0.20	0.31	0.97	0.30	0.08	0.08	0.46	0.19	0.23	0.17	0.18	0.23	0.34	0.31
HE---	2.46	0.40	1.07	1.48	0.55	0.09	0.09	0.70	0.20	0.23	0.30	0.29	0.36	0.45	0.48
HE-W-	2.73	0.40	1.25	1.67	0.72	0.09	0.10	0.89	0.21	0.27	0.35	0.34	0.43	0.50	0.54
HE-S-	2.74	0.40	1.25	1.66	0.72	0.09	0.10	0.89	0.21	0.27	0.34	0.34	0.43	0.49	0.54
HE-SL	3.14	0.42	1.47	1.83	0.80	0.09	0.10	0.97	0.23	0.32	0.37	0.40	0.45	0.54	0.58
HEW--	1.97	0.31	0.34	1.13	0.34	0.08	0.08	0.48	0.19	0.24	0.23	0.22	0.28	0.42	0.42
HEWW-	2.13	0.32	0.34	1.24	0.38	0.08	0.09	0.53	0.19	0.27	0.29	0.28	0.35	0.45	0.49
HEWS-	2.13	0.32	0.34	1.24	0.39	0.08	0.09	0.52	0.20	0.28	0.29	0.28	0.35	0.44	0.49
HEWSL	2.33	0.33	0.35	1.31	0.42	0.08	0.09	0.57	0.20	0.32	0.28	0.30	0.37	0.48	0.51
HES--	1.99	0.31	0.34	1.16	0.33	0.08	0.08	0.47	0.19	0.24	0.22	0.19	0.28	0.33	0.40
HESW-	2.15	0.32	0.34	1.26	0.36	0.08	0.09	0.51	0.20	0.28	0.25	0.21	0.31	0.35	0.45
HESS-	2.14	0.33	0.34	1.24	0.37	0.08	0.09	0.51	0.20	0.29	0.24	0.21	0.31	0.35	0.45
HESSL	2.35	0.33	0.36	1.33	0.40	0.08	0.09	0.55	0.21	0.34	0.26	0.24	0.36	0.38	0.48
NOWW-	0.96	0.22	0.09	0.71	0.49	0.11	0.15	0.85	0.34	0.30	0.08	0.08	0.16	0.34	0.22
NE-W-	1.03	0.25	0.08	0.93	0.59	0.12	0.15	0.99	0.40	0.33	0.08	0.09	0.17	0.34	0.27
NE-S-	1.03	0.25	0.08	0.89	0.68	0.12	0.16	1.10	0.39	0.35	0.08	0.09	0.17	0.34	0.29
NE-SL	1.06	0.26	0.08	0.92	0.77	0.12	0.16	1.20	0.42	0.36	0.09	0.09	0.17	0.34	0.29
NEWW-	0.94	0.23	0.08	0.72	0.51	0.11	0.14	0.87	0.35	0.31	0.07	0.07	0.15	0.31	0.23
NEWS-	0.95	0.23	0.08	0.72	0.54	0.11	0.15	0.90	0.36	0.32	0.08	0.08	0.15	0.32	0.24
NEWSL	0.95	0.23	0.08	0.73	0.55	0.11	0.15	0.93	0.37	0.33	0.08	0.08	0.15	0.32	0.24
NESW-	0.95	0.23	0.08	0.74	0.49	0.11	0.14	0.87	0.36	0.31	0.07	0.07	0.15	0.30	0.23
NESS-	0.93	0.23	0.08	0.72	0.51	0.11	0.15	0.89	0.36	0.32	0.08	0.07	0.15	0.31	0.23
NESSL	0.94	0.23	0.08	0.73	0.54	0.11	0.15	0.92	0.37	0.33	0.08	0.08	0.15	0.31	0.24
PEWW-	0.22	0.04	0.08	0.05	0.33	0.06	0.09	0.43	0.16	0.20	0.06	0.06	0.04	0.13	0.10
PEWS-	0.22	0.04	0.08	0.05	0.34	0.06	0.09	0.44	0.16	0.21	0.06	0.06	0.04	0.13	0.10
PEWSL	0.22	0.04	0.08	0.05	0.36	0.06	0.09	0.49	0.18	0.22	0.06	0.06	0.05	0.13	0.10
PESW-	0.22	0.04	0.08	0.05	0.31	0.06	0.09	0.45	0.18	0.20	0.06	0.06	0.04	0.12	0.10
PESS-	0.22	0.04	0.08	0.05	0.33	0.07	0.09	0.43	0.16	0.21	0.06	0.06	0.04	0.13	0.10
PESSL	0.22	0.04	0.08	0.05	0.35	0.06	0.10	0.44	0.17	0.22	0.06	0.06	0.05	0.13	0.10

Figure 14: Time for all tested methods on all benchmark queries. All queries were warmed up by 3 runs and then run 100 times, or until at least 10 seconds had passed, measuring average running-time.