Towards Unifying Advances in Twig Join Algorithms

Nils Grimsmo **Truls A. Bjørklund**

Department of Computer and Information Science
Norwegian University of Science and Technology
{nilsgri,trulsamu}@idi.ntnu.no

Abstract

Twig joins are key building blocks in current XML indexing systems, and numerous algorithms and useful data structures have been introduced. We give a structured, qualitative analysis of recent advances, which leads to the identification of a number of opportunities for further improvements. Cases where combining competing or orthogonal techniques would be advantageous are highlighted, such as algorithms avoiding redundant computations and schemes for cheaper intermediate result management. We propose some direct improvements over existing solutions, such as reduced memory usage and stronger filters for bottom-up algorithms. In addition we identify cases where previous work has been overlooked or not used to its full potential, such as for virtual streams, or the benefits of previous techniques have been underestimated, such as for skipping joins. Using the identified opportunities as a guide for future work, we are hopefully one step closer to unification of many advances in twig join algorithms.

Keywords: Twig join, indexing, semi-structured data.

1 Introduction

Twig matching is the most heavily used building block for systems offering search in XML with languages like XPath and XQuery [12]. XML has become the de-facto standard for storage of semi-structured data, and the standard for data exchange between disjoint information systems. XPath is a declarative language, and XQuery is an iterative language which uses XPath as a building block. XPath queries can be evaluated in polynomial time [11].

Most academic work related to indexing and querying XML focuses on the *twig matching problem*, which is equivalent to a sub-set of XPath: Given a labeled data tree and a labeled query tree, find all matchings of the query nodes to the data nodes, where the data nodes satisfy the ancestor-descendant (a-d) and parent-child (p-c) relationships specified by the query tree edges.

The example in Figure 1 shows the relation between twig matching and XML search. The tree in part (a) is an abstraction of the XML document in (c). Real XML separates element (tag), attribute and text nodes, but in the abstract model there is only one type of nodes. The XPath and XQuery examples in (d) both specify the same structure as the abstract twig query in (b), where double edges symbolize a-d relationships.

This work focuses on twig matching in indexed data trees. In a typical setting, all data nodes with the same label are stored together, using some encoding which specifies tree positions. To evaluate a query, one stream of data nodes with matching label is read for each query node, and are joined to form twig matches.

This paper gives a structured analysis of recent advances in twig join algorithms, which leads to the identification of a number of opportunities for further improvements. Some direct improvements are identified, such as reduced memory usage in bottom-up algorithms and stronger top-down filters. We highlight cases where new combinations of competing and orthogonal techniques would have clear advantages, but also cases where important previous work has been has been compared to unfairly in our view. We note some open challenges, such as updatability in strong structural summaries, and more efficient detection of cases where simpler and faster algorithms can be used (Section 3.7).

The analysis explores techniques for avoiding redundant computations (Section 3.1), schemes for intermediate result management (Section 3.2), top-down filters for bottom-up algorithms (Section 3.3), skipping joins (Section 3.4), refined access methods (Section 3.5) and virtual streams (Section 3.6).

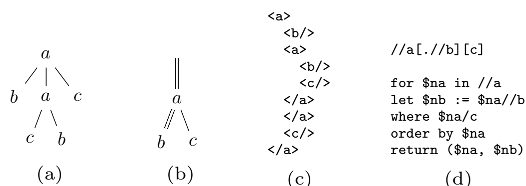


Figure 1: XML and twig matching relation. (a) Abstract data tree. (b) Twig query. (c) XML Data. (d) XPath (above) and XQuery (below).

2 Background: Concepts and Techniques

This section goes through some fundamental concepts and techniques which are useful for the understanding of later algorithms. First we formally define the problem.

Definition 1 (Twig matching)**.** Given a rooted unordered labeled query tree Q and a rooted ordered labeled data tree D, find all complete matchings of the nodes in Q, such that the matched nodes in D follow the structural requirements given by the a-d and p-c edges in Q.

See errata in Appendix A.

First author supported by the Research Council of Norway under the grant NFR 162349.

Copyright ©2010, Australian Computer Society, Inc. This paper appeared at the Twenty-First Australasian Database Conference (ADC2010), Brisbane, Australia, January 2010. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 104, Heng Tao Shen and Athman Bouguettaya, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

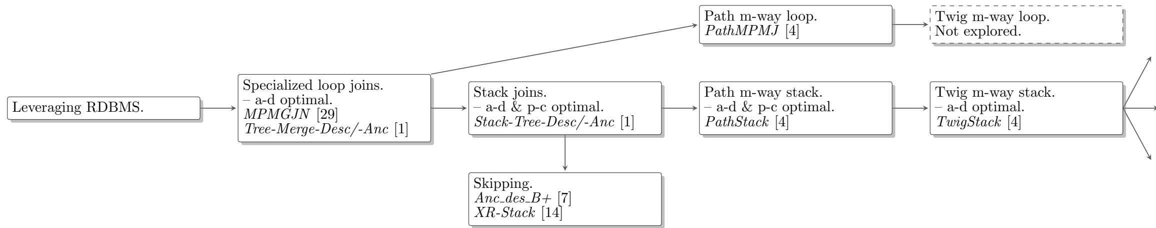
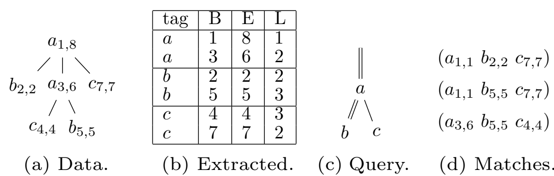


Figure 2: The early history of twig joins. Continued in Figure 8.

Note that there is a slight difference between the semantics of twig matching and XPath. A twig query returns all legal combinations of node matches, while in XPath there is a single query return node. The generality of returning all legal combinations of matches in twig matching may have been the academic focus point because it is useful for the flexibility in XQuery. XPath can also specify more than a-d and p-c relationships, but a majority of XPath queries in practice use only the a-d and p-c axis [12].

Many early approaches to search in semi-structured data used combinations of indexing and tree navigation, but the main focus the last decade has been on indexing with inverted lists and structural joins of streams of query node matches [1]. This paper only considers twig join algorithms.

Indexing and node encoding is critical for the efficiency of twig joins. Usually data nodes are indexed (partitioned) on node labels, using for example inverted lists. Two aspects of how data is stored inside partitions are important: How the position of a node is encoded, and how the nodes in the partition are ordered. For most algorithms nodes are stored in depth first traversal pre-order, such that ascendants are seen before descendants. The positional information which follows nodes must allow decision of a-d and p-c relationships. The most common is the regional *begin,end,level* (BEL) encoding [29], which is used in the data extraction example in Figure 3. It reflects the positions of opening and closing tags in XML (see Figure 1c). The *begin* and *end* numbers are not the same as pre- and post-order traversal numbers, but give the same sorting orders.



tag	B	E	L
a	1	8	1
a	3	6	2
b	2	2	2
b	5	5	3
c	4	4	3
c	7	7	2

(a) Data. (b) Extracted. (c) Query. (d) Matches.

Figure 3: Tree indexing and querying example.

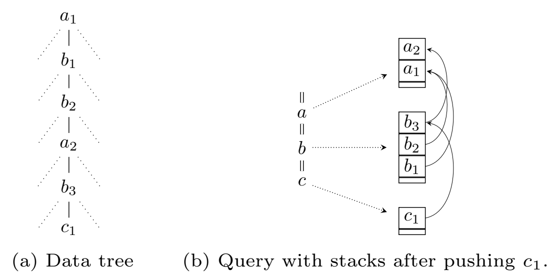
In the following, let T_q denote the stream of matches for query node q, and C_q denote the current data node in this stream. For simplicity, polynomial factors in the size of the query are ignored in asymptotic notation.

The early history of twig joins is shown in Figure 2. An early approach for schema agnostic XML indexing was to store nodes with BEL encoding in an RDBMS, and specify query node relations as a number of inequalities. But these theta-joins are expensive. Specialized loop structural joins which leveraged the knowledge that the data encoded is a tree where introduced [29, 1]. These have $\mathcal{O}(I+O)$ cost for evaluating an a-d relationship, where I and O are the sizes of the input and output streams, but quadratic worst-case for p-c relationships. Stack joins were introduced to get optimal evaluation for all binary structural joins.

A problem with combining the evaluation of a number of binary relationships to answer a query, is that the intermediate results may be of size exponential in the query, even if the output is small. This lead to the introduction of multi-way join algorithms.

Stacks are key data structures in most modern twig join algorithms. Their use here is motivated by their use in depth first tree traversals. To join streams of ascendants and descendants, a stack of currently nested ancestor nodes is maintained. Nodes are popped off the ancestor stack when a non-contained (disjoint) node is seen in either stream. In a path or twig multi-way algorithm, there must be one stack S_{q_i} for each internal query node q_i. The matches for different query nodes must be processed in total pre-order, to ensure that ancestor nodes are added before descendants need them.

In each step in the *PathStack* algorithm [4], the current data node is used to clean all stacks by popping non-containing nodes, before it is pushed on stack. Figure 4b shows the stacks for a query when evaluated on the data in Figure 4a, right after the node c_1 has been pushed. When the current query node is a leaf, all related matches are output. To enable linear time enumeration of the matches encoded in the stacks, each data node pushed onto a stack has a pointer to the closest containing data node in the parent stack, which would be the top of the parent stack as the data node was pushed. Nodes above on the parent stack cannot be ascendants, as the data nodes are read in pre-order. In the example, b_2 and a_2 are not usable together. Because a stack only contain nested nodes, the space needed is $\mathcal{O}(d)$, where d is the maximal depth of the data tree.



(a) Data tree (b) Query with stacks after pushing c_1.

Figure 4: Data structures for *PathStack*.

A technique for getting path matches sorted on higher query node matches first is critical for the efficiency of *TwigStack* and other twig multi-way algorithms. Delaying out-of-order output is achieved by maintaining so-called self- and inherit-lists for each stacked node [1]. The lists for the data and query in Figure 4 is shown in Figure 5.

As a node is popped off stack, the contents of its lists are appended to the inherit-lists of the node below on the same stack, if there is one. This is to maintain correct output order. See for example the lists for b_2 and b_1 in the example. But if the popped node can use some ancestor node in the parent stack, which the node below in its own stack cannot, the contents

of the lists must be appended to the self-lists there. This is decided from the inter-stack pointers. In the example, popping node b_3 results in adding $(a_2 b_3 c_1)$ to the self-list of a_2. *PathStack* has $\mathcal{O}(I+O)$ complexity both with and without delaying output, where I is now the total input size.

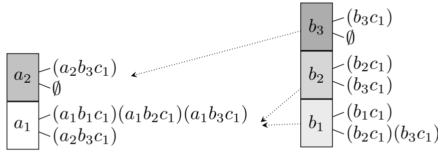


Figure 5: Stack nodes with final self- and inherit lists for the data and query in Figure 4. Darker nodes popped first.

TwigStack [4] was the first holistic twig join algorithm. Using *PathStack* on each root-to-leaf path in a twig query and merging the matches, may lead to many useless intermediate results, because path matches need not be part of complete matches. *TwigStack* improved on this, and achieved $\mathcal{O}(I+O)$ complexity for queries with a-d edges only.

It is a two-phase algorithm, where the first phase outputs matches for each root-to-leaf path, and the second phase merge joins the path matches. The first phase does two things which are critical for the performance of the algorithm: It only outputs path matches which possibly are part of some complete query match, and outputs paths sorted on higher query nodes first, using the technique from [1]. This allow a linear merge in phase two.

TwigStack does additional checking before pushing nodes on stack compared to *PathStack*. The data node at the head of the stream for a query node q is not pushed on stack before it has a so called "solution extension", which means that the heads of the streams of all child query nodes are contained by C_q, and that child nodes recursively satisfy this property. Also, a node is not pushed on stack unless there is a useable ancestor data node on the stack for the parent query node.

Pseudo-code for *TwigStack* is shown in Algorithm 1 (adapted from [4]). It is included here to ease the depiction of the improvements discussed in the following sections. Each query node q has an associated stream T_q with current element C_q, and a stack S_q. The algorithm revolves around a recursive function *getNext*(q), which returns a (locally) uppermost query node in the subtree of q which has a solution extension. If the parent of the returned q has a usable ancestor data node on stack, this means C_q is part of a full solution extension identified earlier, and C_q is pushed on S_q. A path match is found when a leaf node is pushed on stack, but output is delayed to make sure paths are ordered on the query nodes top down (called "blocking" in [4]). Note that actually pushing a leaf node on stack is unnecessary, as it will be popped right off.

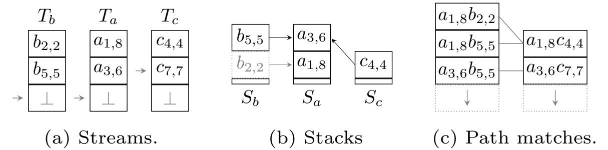
The *getNext*() traversal is bottom up, and is short cut if some node does not have a solution extension (see line 20). Leaves trivially have solution extensions. The traversal has the side effect of advancing the treated query node at least until it contains all it's children (line 23). If it does not contain all children at this point, the child currently with the first pre-order data node (lowest *begin* value) is returned to be forwarded in line 12.

Figure 6 shows the state of the algorithm when evaluating the query in Figure 3c, right after node $b_{5,5}$ has been processed. After the first call to *getNext*(a), when all the streams where at their start position,

Algorithm 1 TwigStack

```
1:  function TwigStack(Q)
2:      while not atEnd(Q)
3:          q := getNext(Q.root)
4:          if not isRoot(q)
5:              cleanStack(S_parent(q), C_q)
6:          if isRoot(q) or not empty(S_parent(q))
7:              cleanStack(S_q, C_q)
8:              push(S_q, C_q, top(S_parent(q)))
9:              if isLeaf(q)
10:                 outputPathsDelayed(C_q)
11:                 pop(S_q)
12:         advance(T_q)
13:     mergePathSolutions()

14: function getNext(q)
15:     if isLeaf(q)
16:         return q
17:     for q_i ∈ children(q)
18:         q_j := getNext(q_i)
19:         if q_j ≠ q_i
20:             return q_j
21:     q_min = min arg_{q_i ∈ children(q)} {C_{q_i}.begin}
22:     q_max = max arg_{q_i ∈ children(q)} {C_{q_i}.begin}
23:     while C_q.end < C_{q_max}.begin
24:         advance(C_q)
25:     if C_q.begin < C_{q_min}.begin
26:         return q
27:     else
28:         return q_min
```

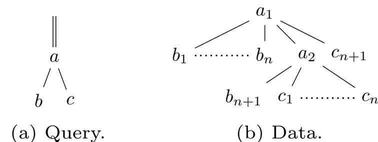


(a) Streams. (b) Stacks (c) Path matches.

Figure 6: TwigStack state when evaluating query in Figure 3c, after processing node $b_{5,5}$.

a itself was returned as it had a solution extension, and $C_a = a_{1,8}$ was pushed on stack. For the second call to *getNext*(a), this was not the case, and b was returned, with head $C_b = b_{2,2}$. Since $b_{2,2}$ had a usable ancestor $a_{1,8}$ on the parent stack S_a, $a_{1,8}$ must have had a solution extension, in which the subtree rooted at $b_{2,2}$ was usable. So $C_b = b_{2,2}$ was pushed on its own stack S_b, and since it was a leaf, the path matching $(a_{1,8} b_{2,2})$ was output. After all paths have been found they are merge joined.

TwigStack suboptimality for mixed a-d and p-c queries comes from having to output path matches without knowing whether the data nodes used can satisfy all their p-c relationships. The algorithm cannot always decide this from the nodes on the stacks and the heads of the streams. For the example in Figure 7 it cannot be decided if the path matches $(a_1, b_1), \ldots, (a_1, b_n)$ are part of a full match before the node c_{n+1} is seen.



(a) Query. (b) Data.

Figure 7: Bad case for *TwigStack*.

For a-d only queries, queries where a p-c edge never follows an a-d edge [24], or on data with non-recursive schema [9], twig joins can be solved with linear cost in the size of the input and the output, using $\mathcal{O}(d)$ memory. Sadly, recursive schema, where nodes with a given label may nest nodes with the

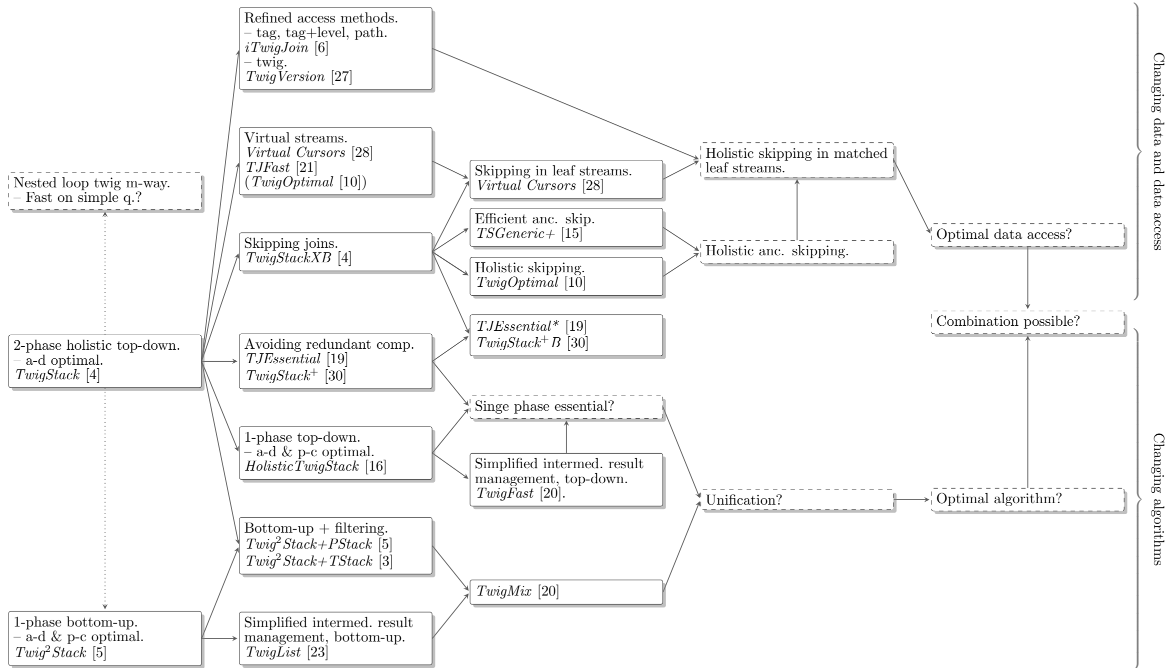


Figure 8: Advances and opportunities in twig joins.

same label, are common in XML in practice [8], and so are mixed queries of the type mentioned above. No algorithm can solve the general problem given tag streaming, linear index size, and a query evaluation memory requirement of $\mathcal{O}(d)$ [24]. One alternative is storing multiple sort orders on disk, instead of only tree pre-order. This would require $\Omega(m^{\min m,d} \cdot D)$ disk space in the worst case, where m is the number of structurally recursive labels and D is the size of the document [9]. Another alternative is to do multiple scans of the streams, but this would require $\Omega(d^t)$ passes in the worst case, where t is a linear metric on the complexity of the query [9]. So, the only viable alternatives left seem to be relaxing the $\mathcal{O}(d)$ space requirement, or using something different than tag partitioning. The following section investigates this, but also many practical speedups to *TwigStack*.

3 Advances

A multitude of different improvements have been presented after the introduction of *TwigStack*. Figure 8 gives an overview of these, with a separation between improved join algorithms and changes to how data is indexed and accessed. The rest of this paper is devoted to a structured review of these advances. Our goal is to identify further improvements, and to shed light on whether it is likely that combining these advances is possible and beneficial.

3.1 Avoiding Redundant Computation

TwigStack may perform many redundant checks in the calls to *getNext*(). Each time a node is returned, the full subtree below has been inspected. The *TJEssential* [19] algorithm improved three specific deficiencies, exemplified in Figure 9.

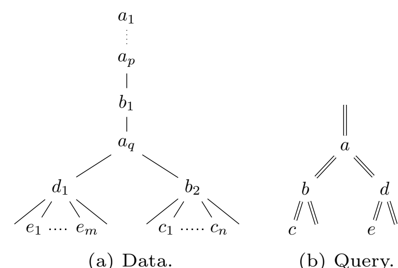
The first deficiency is from self-nested matching nodes. For query node a and data nodes a_1 to a_p in the example, it is unnecessary to recursively check the full subtrees below b and d in each round while pushing the nodes onto S_a. The usefulness of $a_2, \ldots, a_p$ can be seen from the fact that a_1 had a new solution extension, and that $a_2, \ldots, a_p$ contains b_1 and d_1, the heads of the streams of a's children.

The second observation is on the order in which child nodes are inspected. If the child b is inspected before d in line 18 of Algorithm 1, *getNext*(a) will call *getNext*(b) before *getNext*(d) shortcuts the search. There will be $m-1$ redundant calls *getNext*(b) while forwarding the leaf node e.

The third observation is that many useless calls could be made after a stream has reached its end. Assuming that b_2 was the last b-node in Figure 9, no a node later in the tree order would ever be pushed onto stack, and T_a could be forwarded to its end. Also, if S_b was empty, any descendant of b in the query could have their stream forwarded to the end, as the remaining nodes could not be part of a solution.

TJEssential is a total rewrite of *TwigStack*, and is more complex than the original algorithm. *TwigStack*$^+$ [30] is a less involved modification, which only changes the *getNext*() procedure, such that it does not return before a solution is found. *TwigStack*$^+$ does not catch any of the tree above cases, but reduces computation for scattered node matches in practice.

Opportunity 1 (Removing redundant computation in top-down one-phase joins)**.** The improvement of *TwigStack*$^+$ can trivially be ported to recent algorithms such as *HolisticTwigStack* and *TwigFast*, which improve other aspects of *TwigStack* (see Section 3.2). A challenge is to do the same for all the three improvements of *TJEssential*. Also, case three above could be extended to more efficient aligning for multi-document XML collections.



(a) Data. (b) Query.

Figure 9: Giving redundant checks in *TwigStack*.

3.2 Top-down vs. Bottom-up

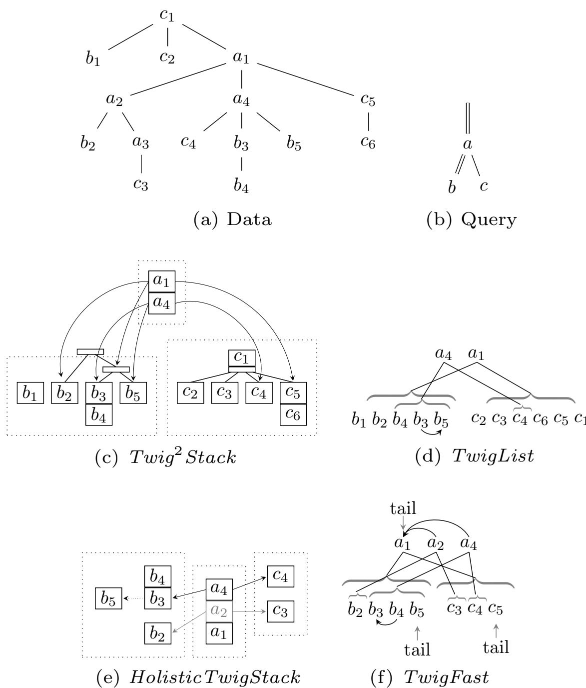
There are two main lines of algorithmic improvements over *TwigStack* which give optimal evaluation of mixed a-d and p-c queries by relaxing the $\mathcal{O}(d)$ memory requirement: Bottom-up algorithms which read nodes in post-order, and later algorithms which go back to top-down and pre-order. Differences between these are illustrated in Figure 10.

Twig²Stack [5] generates a *single combined* stream with post-order sorting for all query node matches with the help of a single stack. With post-order processing it can be decided if an entire subtree has a match at the time the top node is seen.

Figure 10c shows the *hierarchies of stacks* built while processing a query. For each query node, a list of trees of stacks is maintained. A data node strictly nests all nodes *below* in the stack, and all nodes in child stacks in the tree. The lists of trees are stored sorted in post-order, and are linked together by a common root if an ancestor node is processed. From the post-order, the nodes to be linked will always be found at the end of the list, and the new root will always be put at the end. The order naturally maintains itself, and good locality is achieved.

Instead of each node on stack having a pointer to an ancestor node on a parent stack as in *TwigStack*, each stacked data node has for each related child query node, a list of pointers to top stack nodes matching the query axis relationship. Nodes are only added if a-d and p-c relationships can be satisfied, and p-c pointers are only added when levels are correct, as seen for the a and c nodes in the example.

TwigList [23] is a simplification of *Twig²Stack* using simple lists and intervals given by pointers, which improves performance in practice. For each query node, there is a post-order list of the data nodes used so far. Each node in a list has, for each child query node, a single recorded interval of contained nodes,



(a) Data (b) Query (c) *Twig²Stack* (d) *TwigList* (e) *HolisticTwigStack* (f) *TwigFast*

Figure 10: (a) Data. (b) Query. (c) Hierarchies of stacks for *Twig²Stack*. (d) Intervals for *TwigList*. Curved arrows are sibling pointers. (e) Lists of stacks for *HolisticTwigStack* right before c_5 is processed. Previously popped nodes shown in gray. (f) Intervals for *TwigFast* after c_5 has been processed. Curved arrows are ancestor pointers.

as shown in Figure 10d. Interval start and end positions are recorded as nodes are pushed and popped on and off the global stack. All descendant data nodes are processed in between. Compared with the list of pointers in *Twig²Stack*, enumeration of matches is not as efficient for p-c edges, but sibling pointers can remedy this.

HolisticTwigStack [16] is a modification of *TwigStack* which uses pre-order processing, but maintains complex stack structures like *Twig²Stack*. The argument against *Twig²Stack* was a high memory usage, caused by the fact that all query leaf matches are kept in memory until the tree is completely processed, as they could be part of a match. *HolisticTwigStack* differentiates between the top-most branching node and its ancestors, for which a regular stack is used, and lower query nodes, which have multiple linked lists of stacks, as shown in Figure 10e. Each query node match has one pointer to the first descendant in pre-order for each child query node. For "lower" query nodes, new data nodes are pushed onto the current stack if contained, otherwise a new stack is created and appended to the list. As a match for an "upper" query node is popped, the node below on stack must inherit the pointers. Node a_1 would inherit the pointers from both a_2 and a_4 in the example in Figure 10e, and the related lists of child matches would be linked.

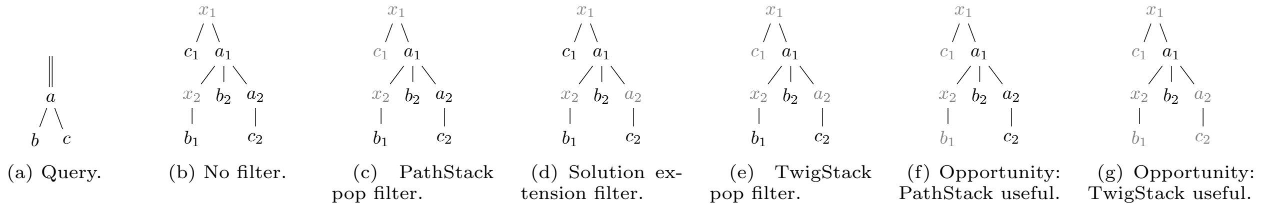
TwigFast [20] is a simplification of *HolisticTwigStack* similar to *TwigList*. There is one list containing matches for each query node, naturally sorted in pre-order, and data nodes in the lists have pointers giving the interval of contained matches for child query nodes, as shown in Figure 10f. Each data node put into the list has a pointer to its closest ancestor in the same list, and there is a "tail pointer", which gives the last position where a node can be the ancestor of following nodes in the streams. These pointers are used for the construction of the intervals.

Different advantages of top-down and bottom-up algorithms can be seen in Figure 10. A top-down algorithm can avoid storing b_1 and c_2, while a bottom-up algorithm is unable to decide that these nodes cannot be part of a solution. On the other hand, a bottom-up algorithm can decide that a_2 is not usable, because it cannot satisfy the p-c relationship between a and c. Both approaches can decide that a_3 is not useful because it does not have a b descendant.

The worst case space complexity of twig pattern matching is an open problem, and the known bounds are $\Omega(\max d, u)$ and $\mathcal{O}(I)$, where u is the number of nodes which are part of a solution [24]. However, practical space savings are possible.

Opportunity 2 (Top-down memory usage)**.** *TwigStack* treats queries as a-d only in the stack construction part of phase one. A node returned from *getNext*() is pushed on stack if it has a usable *ancestor* on the parent stack, even if the query specifies a p-c relationship. For example does not c_3 have to be pushed on stack in Figure 10e, because it does not have a usable parent. Strictly checking p-c relationships before adding intermediate results would reduce memory usage in practice. This optimization was identified for *TJFast* [21] (see Section 3.6), but the later *HolisticTwigStack* and *TwigFast* do not take advantage of this opportunity.

Opportunity 3 (Bottom-up memory usage)**.** Assume a query node q with a p-c relationship to the parent query node. If a candidate match for q is pushed onto a stack in *Twig²Stack*, and the data node below on the stack does not have an incoming pointer, this means the node below will never get a matching parent, and can be popped off stack. For example



(a) Query. (b) No filter. (c) PathStack pop filter. (d) Solution extension filter. (e) TwigStack pop filter. (f) Opportunity: PathStack useful. (g) Opportunity: TwigStack useful.

Figure 11: Filtering approaches for bottom-up up processing. Filtered nodes shown in gray.

could the node c_6 be dropped in Figure 10c. Also, when stack trees for q are merged, some ancestor data node a_i is seen. Then all the stack trees which do not get or have an incoming pointer can be dropped, as all later candidates for the parent query node will be after in the post-order. In the example, the stack trees containing the single nodes c_2 and c_3 could be dropped when a_1 is seen.

Note that improvements on this is hard to transfer directly to *TwigList* unless the lists are implemented as linked lists. But this is by far inferior to using arrays and array doubling on modern hardware, as done in *TwigList* [22]. Another solution is to keep one list for each level for query nodes which have a p-c relationship to the parent query node. Siblings would then be stored contiguously, and interval pointers would implicitly be to a list on a given level. When the first ancestor of a segment of nodes in need of a parent is seen, the useless nodes can be *over-written*. This modification would also make sibling pointers unnecessary and improve efficiency of result enumeration. Figure 12 shows the proposed approach for the data and query in Figure 10, where gray list items can be overwritten.

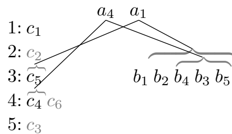


Figure 12: Proposal for multi-level lists for *TwigList*.

3.3 Filtering

Low memory top-down approaches have been used as filters to bottom-up algorithms to reduce space usage by avoiding useless nodes. Note that this does not result in a perfect solution. Assume that node a_1 in Figure 10a had a different label. A $\mathcal{O}(d)$ space top-down pre-order approach could not decide that b_2 in the example was not part of a match, and a bottom-up algorithm would have to keep it in memory until the entire tree was read. Figure 11c-e shows the effects of different previously proposed filters. **PathStack Pop Filter.** In the original *Twig²Stack* paper [5], *PathStack* was proposed as a pre-filter to allow early result enumeration. *PathStack* is run as usual, but without its result enumeration. As disjoint nodes are popped off their stacks, they are passed to *Twig²Stack*. When the bottom node is popped from the stack of the root query node, all results can be output, and the hierarchical stacks destroyed. A side effect of this procedure is that only nodes that are part of some prefix path match are used (these are not necessarily part of a full root-to-leaf path match). In Figure 11c, node c_1 is avoided. Note that one data node may result in the popping of multiple nodes on multiple stacks, and that *Twig²Stack* must receive descendants before ascendants.

Solution Extension Filter. *TwigMix* [20] is an algorithm which combines the simplified data structures in *TwigList* with the *getNext*() function from *TwigStack* as a filter. This combination gives efficient evaluation for queries involving p-c edges, and reduced memory usage in practice. An advantage of this approach over *Twig²Stack+PathStack* is that there is no overhead of maintaining an extra set of stacks, and that internal nodes are filtered holistically. The downside is that nodes are added without even having a possible parent or ancestor. Figure 11d shows that node a_2 is filtered, because it never has a solution extension (misses b node below), while nodes c_1 and b_1 are not filtered.

TwigStack Pop Filter. *TwigStack* can also be used as a filter for *Twig²Stack* [3]. A node is never added to the hierarchical stacks if it is not popped from a top-down stack in *TwigStack*. As a node is never pushed on stack if it does not have a usable ancestor, which again has a solution extension, this gives additional filtering, at the cost of maintaining the top-down stacks. Figure 11e shows the improvements both over *PathStack* and solution extension as filters. An issue is that *Twig²Stack* expects the stream of nodes to be in post-order, and that *TwigStack* may pop nodes off stacks out of this order. When a node is returned from *getNext*(), only the related stack and the parent stack are inspected. Also, *TwigStack* does not keep leaf matches on stack, but nested leaf matches may arrive later. In [3] this is solved by keeping an extra queue of data nodes into which popped nodes are placed if the algorithm decides later popped nodes may precede them.

A different solution could be to allow nested nodes on query leaf stacks, and to inspect all stacks when popping disjoint nodes to ensure post-order, as with the *PathStack* filter. Also, *Twig²Stack* actually does not need to see nodes in strict post-order, but only to see descendants before ascendants. Hence, not all stacks in the query would have to inspected, only ascendant and descendant stacks of the current node.

Opportunity 4 (Stronger filters)**.** There are further possibilities for filters with $\mathcal{O}(d)$ space usage. Instead of using all nodes popped off stacks in *PathStack*, one could use the nodes which would be *used* in full path match. As leaf nodes are pushed on stack, a simplified enumeration algorithm could be run, tagging nodes which take part in solutions. As can be seen in Figure 11f, this is an improvement over the previous *PathStack* filter, but only partially over the solution extension filter, which to a greater extent filters matches for higher query nodes. Leaves trivially have solution extensions. The "*PathStack useful*" filter works well on lower query nodes. Note that as the bottom-up algorithms to a greater extent handle upper nodes themselves, a filter is of most use if it removes lower query node candidates effectively. An even stronger filter would be to only use nodes which would have been output as parts of path matches in *TwigStack*, as shown in Figure 11g. None of [5, 20, 3] compare with using any other type of filter. A thorough comparison should compare both the practical

space reductions filters give, their absolute costs, and how their use affect the total computational cost.

Opportunity 5 (Unification or assimilation)**.** When comparing absolute performance gains presented in the respective papers, *TwigFast* is the winner on performance for pure tag streaming. As this is a very important result, it should be verified independently. Before *TwigFast* is picked as the method of choice, at least the following should be answered: (i) Can the improvements discussed in Section 3.1 be applied? (ii) Is it superior to improved top-down and bottom-up combinations? (iii) Does the picture change when random access gets more expensive compared to computation? [20] does not comment on the spatial locality of memory access patterns in the intermediate result data structures in *TwigFast*, while they are very good for *TwigList* [23].

3.4 Skipping Joins

Skipping is a useful technique when the streams to be joined have very different sizes. Skipping is used to jump forward in a stream to avoid reading and processing parts of the streams which cannot contain useful nodes. Figure 13 shows cases where different skipping techniques and data structures can be used.

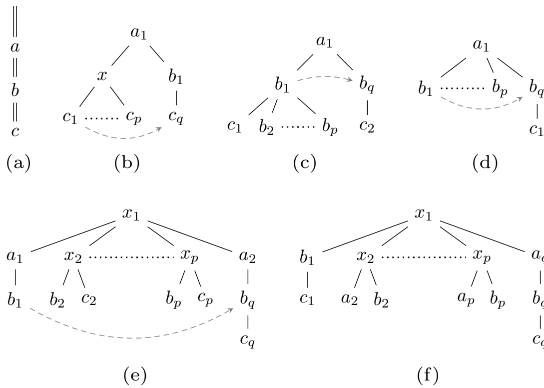


Figure 13: Benefits of skipping techniques. (a) Query. (b) Descendants easily skipped with B-tree. (c) Skip past discarded ascendant. (d) XR-tree needed to skip ascendants. (e) Holistic skipping preferred. (f) Holistic skipping with XR-tree needed.

Simple B-tree skipping can be used to skip in descendant streams, and to some extent in ancestor streams. It is trivial to skip in the descendant stream to find the first possible contained node, which is the first node with a larger *begin* value. In Figure 13b, T_c is forwarded from c_1 to c_q to find the first possible ascendant of b_1.

But skipping to find the next ascendant of a node using the same approach is not effective, as any node with a lower *begin* value may be a match. A trick for ancestor skipping was introduced in [7]. If a node b_i is popped off stack S_b due to disjointness with the current data node in some query node, T_b is forwarded to the first node not contained by the popped node, a b_j such that $b_i.end < b_j.begin$. An example of this can be seen in Figure 13c. If c_2 pops b_1 off stack, T_b can be forwarded beyond b_1 to b_q, because no descendant of b_1 could be useful.

XR-trees enable ancestor skipping in the general case [14]. Figure 13d shows an example where the above trick cannot be used. The XR-tree is B-tree variant which can retrieve all R ancestors or descendants of a node from N candidates in $\mathcal{O}(\log N + R)$ time. Typically one tree is built for each tag. To find all a ascendants of a node d_k, find the node a_i with the nearest preceding *begin* value, and then all a ascendants of a_i in the XR-tree for a. Conceptually the XR-tree contains for each node, the list of ascendants, which gives quadratic space usage when implemented naively. Linear space usage is achieved by not storing information redundantly internally in XR-tree nodes, and by storing common information in internal XR-tree nodes.

TSGeneric+ (also called *XRTwig*) [15] extends the use of the XR-tree to *TwigStack*, and does two major modifications to the algorithm. The first is to skip forward to containment of the first child in the *getNext*() procedure (see line 23 in Algorithm 1). The second change is more involved. Before calling *getNext*() on all children in line 18, a "broken" edge in the query sub-tree is repeatedly picked, and the two related nodes are "zig-zag" forwarded until they match. This is only done if the query node does *not* have data nodes on the stack. Choosing which edge to fix is either top-down, bottom-up or by statistics.

Holistic skipping was introduced in the *TwigOptimal* [10] algorithm, which uses B-trees. Figure 13e shows a case where the approach from *TSGeneric+* would be very expensive, reading all nodes b_2-b_p and c_2-c_p to fix the edge between b and c. *TwigOptimal* processes the query bottom-up then top-down. In the bottom-up phase, nodes are forwarded to contain their descendants, and in the top-down phase, nodes are forwarded until they are contained by their parent. To avoid as many data structure reads as possible, nodes are forwarded to "virtual positions", which have only *begin* values. When a full traversal did not forward any node, the node with the minimal current *begin* value is forwarded to a real data node.

The name of the *TwigOptimal* algorithm may be slightly misleading, as the optimality is given skip structures on *begin* values only. Only *TSGeneric+* using simple B-trees is compared with. The effects of the two contributions, holistic skipping and the virtual positions, are not separately tested. *TwigOptimal* would not be efficient neither on the example in Figure 13c nor 13d. The approach is best when there are more matches for lower query nodes. An common exception from this is queries with leaf value predicates in XML. [10] mentions skipping to the closest ancestor and then backtracking to the first ancestor as a possible practical speed-up.

Opportunity 6 (Holistic effective ancestor skipping)**.** Figure 13f shows a case where both *TSGeneric+* with XR-trees and *TwigOptimal* would fail to be efficient. The former would zig-zag join a_2-a_p and b_2-b_p, and the latter would be unable to forward T_a to a_q without checking at least all of a_2-a_p for ancestry of c_q. Combining holistic skipping and data structures for efficient ancestor skipping is required in a robust solution.

Opportunity 7 (Simpler and faster skipping data structures)**.** The XR-tree is a dynamic data structure which supports insertions and deletions [14]. In regular keyword search engines, simpler data structures are usually preferred to the heavier B-trees when the data is static or semi-static. Similar simpler data structures should also be created for efficient ancestor skipping. If their use is still expensive, techniques similar to the trick used to skip past discarded ascendants should be applied when possible.

3.5 Refined Access Methods

There are alternatives to indexing and accessing data by node labels, such as using label and level, or the root-to-node path strings of labels (called *tag+level* and *prefix path streaming* [6]). With refined partitioning some method must be used to identify the

useful partitions for each query node. For prefix path streaming this would be the partitions with data paths matching the root-to-node downward paths in the query.

Structural summaries are directory structures used to classify nodes based on their surrounding structure. They where first used in combination with tree traversals, but have later been integrated with pure partitioning schemes [18]. The most common is a *path summary*, which is a minimal tree containing all unique root-to-node label paths seen in the data. The data nodes associated with a summary node is called the extent of the node. Figure 14a shows the path summary for the data in Figure 14b, and the extents are shown in Figure 14d.

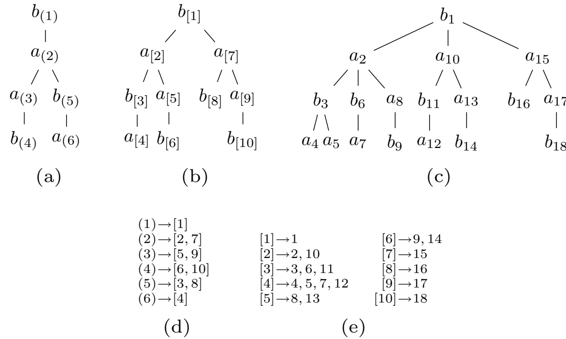


Figure 14: Structural partitioning example. (a) is path summary for (b), with extents shown in (d). (b) is F&B summary for (c), with extents shown in (e).

Many alternative summary structures have been devised for general graphs. A structure which is also directly useful for trees is the stronger F&B-index [17], where two nodes are in the same equivalence class if they have the same prefix path, *and* have children of the same equivalence classes. In the example, (b) is the F&B summary of the tree in (c). For graphs, the F&B index can be found in $\mathcal{O}(m \log n)$, where m and n are the number of edges and nodes in the graph. It is not known whether the F&B index can be found more efficiently for trees.

Opportunity 8 (Updates in stronger summaries)**.** Simple path summaries are usually small, and are easily updateable. When traversing a data tree for indexing, the path summary is used as a deterministic automaton, where new nodes are added on the fly when needed. Data nodes can be put in the correct extents immediately. If a data tree is updated, only the data nodes whose extent changes are affected. An interesting question is the updatability of stronger structural summaries. In the worst case for the F&B index, the structure of an entire *containing subtree* below the global root could change if a data node is added or removed, by causing or removing equivalence with another subtree. What are the implications of strategies lessening the restrictions on the F&B index? Would this give critical "fragmentation effects" in practice? And are updates cheaper in coarser variants, such as the F+B-index [17]?

Opportunity 9 (Hashing F&B summaries)**.** In some search scenarios, there are many small documents, which are parts of a virtual tree. Document updates can be implemented as document deletes and inserts. With simple path summaries, documents can be added with cost linear in the document size, by traversing the summary deterministically. However, more refined summaries are not deterministic. Are stronger summaries like the F&B index suitable in this model? A challenge is that matching a new document in the F&B index has cost linear in the size of the *summary* in the worst case, not the document. Assume now that a_2, a_{10} and a_{15} in Figure 14c are document roots. The structure of each document is classified by a node on depth two in the F&B summary in Figure 14b. If a new document is added below b_1, it will either have the structure defined by $a_{[2]}$ or $a_{[7]}$, or a new subtree will be added below $b_{[1]}$. One possibility is to index the F&B summary by *hashing* each level 2 subtree, as these represent full document structures. When a new document is indexed, a summary of the document structure can be built and hashed, to identify a match in the global F&B index.

TwigVersion [27] is a twig matching approach which introduces a novel two-layer indexing scheme, with an F&B summary of the data, and a path summary of the F&B summary. This reduces the expense of matching in the F&B index. But as they only compare to twig join algorithms which do not use structural summaries, and also introduce many other ideas, it is hard to assess the usefulness of the two-layer approach itself. They compare their two layer approach with a pure F&B index, but do not state how they search in it.

A common way to *use* path summaries is to match each individual root-to-leaf path, and *prune* away matches which cannot be part of a full match [6, 2]. Another solution, which is more robust for large path summaries, is to label partition the summary and run a full twig join algorithm on it. In [3] a novel combination of *Twig*^{2}*Stack* and *TwigStack* is used for matching in large path summaries (see Section 3.3).

Opportunity 10 (Exploring summary structures and how to search them)**.** Many twig join algorithms have leveraged the benefits of path summaries. Stronger summaries like the F&B index are not as commonly used, maybe because of worst case size and implementational complexity. Using different algorithms to search various types and combinations of summaries has not been thoroughly explored. An evaluation should address the total benefits of different single- and multi-level strategies, but also detail the local cost of specific matching methods in specific summary types of different sizes.

Multi-stream access may be required for a single query node when partitioning on path, as there may be many path matches. One solution is to merge the streams. Another is to have a tag partitioned store, and filter the data nodes on matching path ID [28]. A speedup to this approach is to *chain* together nodes with the same path [18]. This is also useful when indexing text nodes on value and integrating structure information.

S^3 [13] is a twig matching system which takes all possible combinations of individual streams matching query nodes based on prefix path, and solves each combination separately, merging the results of each evaluation. This approach does not give polynomial worst-case bounds.

Blocking is the reason for the sub-optimality of *TwigStack*. When partial matches must be output to evaluate the data and query in Figure 15a, it is because b_1 and c_1 blocks each others access to c_2 and b_2 respectively.

Using more fine grained partitioning and streaming solves some blocking issues, because there are multiple heads of streams. A partitioning which refines another always inherits the reduced blocking [6]. In *tag+level streaming* there is no blocking when p-c edges only are used below the query root [6]. But in mixed queries, such as in Figure 15b, blocking can

still occur. There the stream for tag d level 3 is $[d_1, d_2]$ and the stream for c level 4 is $[c_1, c_2]$. There are only two matches for the query, and data nodes c_1 and d_1 block each other.

Prefix path streaming results in no blocking when there is a single branching node in the query. It solves the case in Figure 15b optimally, but not the one in 15c. There the stream for the path *abac* is $[c_1, c_2]$, and the stream for the path *ababe* is $[e_1, e_2]$. Even though c_2 is also usable in the match with root a_1, it cannot be known whether or not c_1 is usable, because e_1 blocks for e_2.

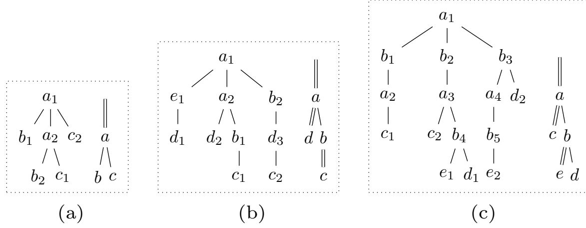


Figure 15: Cases of data and query blocking with (a) tag streaming, (b) tag+level streaming, (c) prefix path streaming. Adapted from [6].

iTwigJoin [6] uses a specialized approach for accessing multiple useful streams, which supports any partitioning scheme. In its variant of $getNext(q)$, it considers for each matching stream, the streams which are *usable together* with this stream, for each child of q. This reduces the amount of blocking when more fine grained partitioning is used. The space usage for *iTwigJoin* is $\mathcal{O}(d)$, and the running time is $\mathcal{O}(t(I + O))$ when no blocking occurs, where t is the number of useful streams.

Opportunity 11 (Access methods for multiple matching partitions)**.** Strategies for accessing multiple useful streams include merging, filtering and chaining of input, informed merging access like in *iTwigJoin*, and merging the output of multiple joins. Many authors do not argue for the rationale of their choice of how to access multiple useful partitions for a node. A new access paradigm is presented in [6], but only tag streaming is compared with. The benefit of the method for accessing multiple matching streams is not separated from the benefit of reduced total input size. The technique reduces the number of intermediate paths output by phase one in *TwigStack*, and would undoubtedly reduce the amount of memory needed for intermediate results in time optimal top-down algorithms like *HolisticTwigStack* and *TwigFast*, but it is not certain if it is a win-win both on memory and speed in practice.

3.6 Virtual Streams

Another approach that can lead to reading less input is using "virtual streams" for internal nodes, by inferring the existence of nodes from their descendants. This requires a position encoding which allows ancestor position reconstruction [26], such as Dewey [25]. Ancestor label paths must also be infereable, and a path summary is an excellent tool for this. Consider the example in Figure 16, where streams of nodes matching leaf label paths are shown. For the node 1.2.1.2 with path (4) *a.a.b.a*, it can be inferred that one candidate for the query root is 1.2 with path (2) *a.a*.

Virtual Cursors is an implementation of virtual streams using Deweys and path summaries [28]. Generating a "next" match for an internal query node

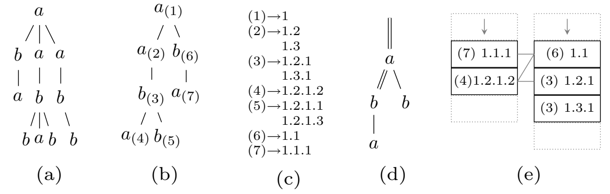


Figure 16: Virtual stream example. (a) Data. (b) Path summary. (c) Extents of summary nodes. (d) Query. (e) Leaf streams.

is done by going through the prefixes of a leaf node's Dewey and path, and using those where the ending tag is correct. The search stops when the new Dewey is lexicographically greater than the previous, meaning later in the pre-order. [28] does not give details on how ancestor candidates are generated, but this can be done in time linear in the depth of the leaf match used.

Forwarding the entire query is done by repeatedly picking a leaf with a "broken" path, forwarding it to containment by the maximal ancestor, and then forwarding all ancestors virtually to contain the leaf. In the system described by [28], tag streaming with path ID filtering was used, and B-trees were used for skipping during leaf forwarding.

Other virtual stream approaches have later been introduced. *TJFast* [21] is an independently developed algorithm which does not use a structural summary, but stores with each data node the root-to-node label path and the Dewey encoding together in a compressed format. Label paths are matched for each node processed. An improvement over *Virtual Cursors* as described, is that path matching is also done when generating internal nodes, giving fewer useless candidates. Also, non-branching internal nodes can be ignored during query evaluation, because they are implicit from the path matchings of below and above nodes. *TJFast* does not produce streams for internal nodes, but maintains *sets* of currently possible candidates.

TwigVersion [27] and S^3 [13] (see Section 3.5) are non-holistic approaches which combine structural summaries and inference of internal node matches. *TwigVersion* computes sets of matches bottom-up. Each child node query generates a set of candidates for its parent query node based on its own matches, and these sets are then intersected. S^3 uses the potentially exponential number of ways a query matches the summary, and evaluates each such match, merging the results. For one summary matching, it looks at the query leaf nodes pairwise, and merge joins sets based on lowest common ancestor query nodes. This could give large useless intermediate results. The holistic skipping algorithm *TwigOptimal* [10] does partially implement virtual streams through its "virtual positions" (see Section 3.4).

Opportunity 12 (Improved virtual streams)**.** To reduce the number of matches and make it possible to ignore non-branching internal nodes, only structurally matching internal nodes should be generated. A structural summary can be used to avoid repeated matching of paths. However, how to store path matching information is not obvious. Given a matching path for a leaf query node, there may be an exponential number of combinations for matching the above query nodes. Should the matches be calculated on the fly as in *TJFast*, kept in independent sets for each node above a leaf match, or encoded in stacks? Or is it enough to store candidates for the lowest branching node above each leaf match, if the

query nodes on a path are processed bottom-up?

It is common to store Deweys in a succinct format to reduce space usage, but in addition, some scheme should also be devised to reduce the redundancy of using related Deweys in ascendant and descendant nodes. It is also preferable if node encodings do not have to be fully de-compressed to be compared during query evaluation, but that the compressed storage format allows for cheap direct comparisons.

Opportunity 13 (Holistic skipping among leaf streams)**.** In some sense, virtual streams *are* skipping by not generating unrelated matches for internal query nodes. The *Virtual Cursors* algorithm does perform skipping which is "path-holistic", in the way broken root-to-leaf paths are fixed. The order in which leaves are picked is not specified [28], but query node pre-order could have been used. If the lexicographically largest broken leaf was picked, the skipping would become truly holistic.

The work in [28], in combination with some intermediate result handling method from Section 3.2, may be a suitable starting point in the hunt for the "ultimate" twig matching approach, but the work is not much compared with, or even referenced.

3.7 Query Difficulty Classes

As mentioned in Section 2, the "difficulty" of twig joins comes from mixture of a-d edges followed by p-c edges in queries, in combination with structurally recursive labels in the data. [24] shows that when an a-d edge is never followed by a p-c edge downwards in a query, it can be evaluated in linear time and $\mathcal{O}(d)$ space. When there in addition is a single return node (as in XPath), it can also be evaluated in $\mathcal{O}(1)$ space. If after combining all the advances listed in this paper, faster evaluation methods still exist for some classes of queries, practical implementations should take advantage of this.

Opportunity 14 (Identifying and using difficulty classes)**.** Can the correctness of using a simpler matching algorithm be decided not only from the query, but also from the query and the data? Structural summaries give possibilities for this. What happens if there are only single path candidates for some query nodes? What happens when the tree level for matches of a query node is fixed? What happens if data node matches with given paths for some query nodes fix path matches for other query nodes?

In [2] additional information is collected in a path summary, noting whether a node always has a given child, and whether there is at most one child. This information is used there to simplify query evaluation when there are non-return nodes in the query, such as in XPath. Could such statistics also allow detection of more cases where query evaluation can be simplified for general twig matching?

4 Conclusion

We have given a structured analysis of recent advances, and identified a number of opportunities for further research, focusing both on join algorithms and index organization strategies. Hopefully this has given an overview which has led us one step further towards unification of the numerous advances in this field.

One conclusion is that given its sheer volume, it seems nearly impossible to consider *all* related work when presenting new twig join techniques. The field would benefit greatly from an open source repository of algorithms and data structures.

References

[1] S. Al-Khalifa, H. Jagadish, N. Koudas, J. Patel, and D. Srivastava. Structural joins: A primitive for efficient XML query pattern matching. In *Proc. ICDE*, 2002.

[2] A. Arion, A. Bonifati, I. Manolescu, and A. Pugliese. Path summaries and path partitioning in modern XML databases. In *Proc. WWW*, 2006.

[3] R. Bača, M. Krátký, and V. Snášel. On the efficient search of an XML twig query in large DataGuide trees. In *Proc. IDEAS*, 2008.

[4] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *Proc. SIGMOD*, 2002.

[5] S. Chen, H.-G. Li, J. Tatemura, W.-P. Hsiung, D. Agrawal, and K. S. Candan. Twig^{2}Stack: bottom-up processing of generalized-tree-pattern queries over XML documents. In *Proc. VLDB*, 2006.

[6] T. Chen, J. Lu, and T. W. Ling. On boosting holism in XML twig pattern matching using structural indexing techniques. In *Proc. SIGMOD*, 2005.

[7] S. Chien, Z. Vagena, D. Zhang, V. Tsotras, and C. Zaniolo. Efficient structural joins on indexed XML documents. In *Proc. VLDB*, 2002.

[8] B. Choi. What are real DTDs like. Technical Report MS-CIS-02-05, University of Pennsylvania, 2002.

[9] B. Choi, M. Mahoui, and D. Wood. On the optimality of holistic algorithms for twig queries. In *Proc. DEXA*, 2003.

[10] M. Fontoura, V. Josifovski, E. Shekita, and B. Yang. Optimizing cursor movement in holistic twig joins. In *Proc. CIKM*, 2005.

[11] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *Proc. VLDB*, 2002.

[12] G. Gou and R. Chirkova. Efficiently querying large XML data repositories: A survey. *Knowl. and Data Eng.*, 2007.

[13] S. K. Izadi, T. Härder, and M. S. Haghjoo. S^3: Evaluation of tree-pattern XML queries supported by structural summaries. *Data Knowl. Eng.*, 2009.

[14] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-tree: Indexing XML data for efficient structural joins. In *Proc. ICDE*, 2003.

[15] H. Jiang, W. Wang, H. Lu, and J. Yu. Holistic twig joins on indexed XML documents. In *Proc. VLDB*, 2003.

[16] Z. Jiang, C. Luo, W.-C. Hou, and Q. Z. D. Che. Efficient processing of XML twig pattern: A novel one-phase holistic solution. In *Proc. DEXA*, 2007.

[17] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In *Proc. SIGMOD*, 2002.

[18] R. Kaushik, R. Krishnamurthy, J. F. Naughton, and R. Ramakrishnan. On the integration of structure indexes and inverted lists. In *Proc. SIGMOD*, 2004.

[19] 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. Advances in Databases: Concepts, Systems and Applications*, 2007.

[20] J. Li and J. Wang. Fast matching of twig patterns. In *Proc. DEXA*, 2008.

[21] J. Lu, T. Ling, C. Chan, and T. Chen. From region encoding to extended Dewey: On efficient processing of XML twig pattern matching. In *Proc. VLDB*, 2005.

[22] L. Qin. Personal correspondence, 2009.

[23] L. Qin, J. X. Yu, and B. Ding. TwigList: Make twig pattern matching fast. In *Proc. DASFAA*, 2007.

[24] M. Shalem and Z. Bar-Yossef. The space complexity of processing XML twig queries over indexed documents. In *Proc. ICDE*, 2008.

[25] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *Proc. SIGMOD*, 2002.

[26] F. Weigel. *Structural summaries as a core technology for efficient XML retrieval.* PhD thesis, Ludwig-Maximilians-Universität München, 2006.

[27] X. Wu and G. Liu. XML twig pattern matching using version tree. *Data & Knowl. Eng.*, 2008.

[28] B. Yang, M. Fontoura, E. Shekita, S. Rajagopalan, and K. Beyer. Virtual cursors for XML joins. In *Proc. CIKM*, 2004.

[29] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. *SIGMOD Rec.*, 2001.

[30] J. Zhou, M. Xie, and X. Meng. TwigStack$^+$: Holistic twig join pruning using extended solution extension. *Wuhan University Journal of Natural Sciences*, 2007.

A Errata

1. Algorithms *TwigList* [23], *HolisticTwigStack* [16] and *TwigFast* [20] are incorrectly referred to as optimal in Figure 8, in Section 3.2, and in Section 3.5. Only algorithm $Twig^2Stack$ [5] is optimal as described.