TRACK: A Novel XML Join Algorithm for Efficient Processing Twig Queries

Dongyang Li **Chunping Li**

School of Software, Tsinghua University
MOE Key Laboratory for information System Security, China
Email: lidy05@mails.tsinghua.edu.cn

Abstract

In order to find all occurrences of a tree/twig pattern in an XML database, a number of holistic twig join algorithms have been proposed. However, most of these algorithms focus on identifying a larger query class or using a novel label scheme to reduce I/O operations, and ignore the deficiency of the *root-to-leaf* strategy. In this paper, we propose a novel twig join algorithm called Track, which adopts the opposite *leaf-to-root* strategy to process queries. It brings us two benefits: (i)avoiding too much time checking the element index to make sure all branches are satisfied before a new element comes. (ii)using the tree structure to encode final tree matches so as to avoid the merging process. Further experiments on diverse data sets show that our algorithm is indeed superior to current algorithms in terms of query processing performance.

Keywords: XML, Twig Join Algorithm.

1 Introduction

With the increasing amount of XML for exchanging data on the web, finding useful information efficiently among these large volumes of data has been a meaningful task. Researchers often represent XML documents as tree models. Meanwhile, several XML query languages have been proposed, such as XPath and XQuery, etc. These query expressions are usually modeled as tree patterns that specify a set of constraints, so the query process is converted to finding all occurrences of the tree pattern of a query in the forest of XML documents.

XML query has been extensively studied in recent years. Previous researches build various structural indexes for XML documents, and expect to use them to accelerate the query processing time. But in most cases, we should enlarge the size of the index in order to increase the query processing time. Furthermore, because of fixed structure of the index, almost all of them are proved to be efficient only for a portion of queries (Cooper 2001, Chung 2002, Goldman 1997, Kaushik 2002, Kaushik-Shenoy 2002). Hence, a novel method named *region scheme* is proposed. It associates each XML document element with a 4-tuple which exclusively identifies the position of the element.

Based on *region scheme*, most existing researches focus on structural join algorithms to capture the relationships between elements. Zhang (Zhang 2002) proposed the multi-predicate merge join, which is a variation of the traditional merging join algorithm. Later Al-Khalifa (AL-Khalifa 2002) pointed out the bottleneck of algorithms at that time, and then used stacks to store the intermediate results so as to avoid repeatedly scanning the input lists. N.Bruno (Bruno 2002) concluded the stack-based algorithms and brought forward a holistic join algorithm, named TwigStack, which uses linked stacks to compactly represent partial results to path patterns, and then composes them to obtain matches for the tree pattern.

In this paper, we propose a novel twig join algorithm Track, which first partitions queries into axes, and then processes each axis individually using the structural join algorithm. During the processing of each axis, we selectively recover the tree structure for one query node and build links between forests to compactly represent final twig matches for the twig query. Our algorithm first builds the tree structure for leaf nodes, and then traces *leaf-to-root* pathes back to the root. Because of its *leaf-to-root* strategy, it avoids too much time checking the element index to make sure all branches are satisfied before a new element comes. Finally, we output all twig matches in a reverse root-to-leaf manner.

The rest of the paper is organized as follows. In Section 2, we introduce basic notations and definitions used in the paper. In Section 3, we analysis the limitation of current algorithms. Section 4 shows the whole procedure of Track. In Section 5, we show the experimental results by comparing our algorithm with other methods. In Section 6, we have the concluding remarks and the future work.

2 Background

An XML document can be modeled as a rooted, ordered tree where each node corresponds to an element or a value, and each edge represents the element-subelement or element-value relationship. Figure 1 shows an example of this model.

Query expressions in XML query languages can be represented as path patterns or tree patterns for matching relevant portions of data. For example, the query expression:

book//author[fn= 'jane' AND ln= 'poe']

asks for the author of a book whose first name is jane and the last name is poe. In the above instance, node labels in tree patterns include elements or string values, and there are two kinds of edges, the single slash '/' which denotes the parent-child relationship, and the double slash '//' which denotes the ancestor-descendant relationship.

Region scheme is derived from the inverted index data structure in information retrieval (Salton 1983). Similar to the technique to deal with text words in

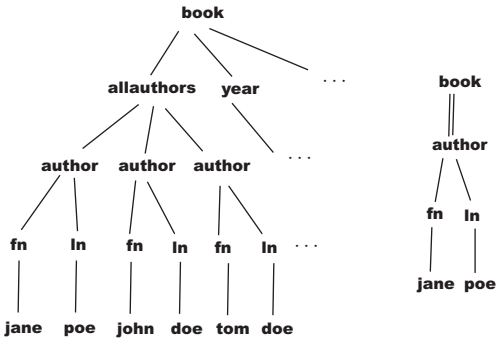

Figure 1: *A Sample XML Document and A Query Pattern*

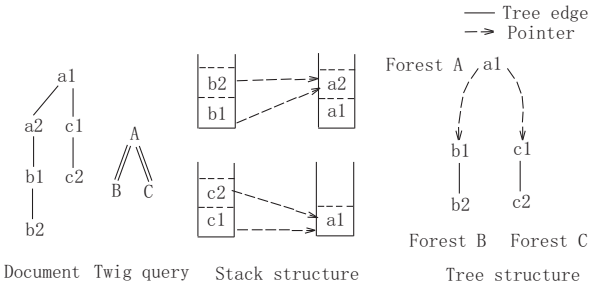

Figure 2: *TwigStack for twig query processing*

traditional IR systems, *Region scheme* maps elements with the same tag name to an inverted list. In other words, each inverted list records all occurrences of elements with the same tag name in XML documents. In general, one record is represented as a 4-tuple(*doc*, *start*, *end*, *depth*) to identify the position of an element exclusively. While *doc* reduces the searching space to one document, *start* which is generated by a *pre-order* traversal of the document trees exactly finds the occurrence position. *end* is the maximal *start* of elements in the subtree of current element, and *depth* gives additional information to determine the parent-child relationship of tree elements. Given two elements u, v, the positional relationship between them is defined as follows:

1. iff $u.end < v.start$, u is a pre-node of v;
2. iff $u.start < v.start$, and $u.end > v.end$, u is an ancestor of v.
3. iff $u.start < v.start$, $u.end > v.end$, and $u.depth + 1 = v.depth$, u is a parent of v.
4. iff $u.start > v.start$, and $u.end < v.end$, u is a descendant of v.
5. iff $u.start > v.start$, $u.end < v.end$, and $u.depth - 1 = v.depth$, u is a child of v.
6. iff $u.start > v.end$, u is a sub-node of v.

3 Related works

Bruno (Bruno 2002) designed a stack-based algorithm TwigStack, which used linked stacks to compactly represent partial results to path patterns, and then composed them to obtain matches for the tree pattern. For each query node q, we associate a stream T_q and a stack S_q with it. The stream T_q stores all elements with the same tag name q in XML documents in {*doc*, *start*} sequence. The stack S_q temporarily stores the partial results, and every node in S_q consists of a pair: {an element from T_q, a pointer to a node of the parent stack}.

Consider a twig query and a document in Figure 2, the elements are processed in the sequence of *a1*, *a2*, *b1*, *b2*, *c1*, *c2*. When the first element *a1* comes, TwigStack checks the children nodes *B*, *C* of *A*, and finds that there exist two elements *b1* and *c1* which are descendants of *a1*. Hence, *a1* is pushed into the stack S_A. As *a2* comes, it does not satisfy the condition. When *b1*, *b2*, *c1*, or *c2* comes, we separately output the partial results {*a1*, *b1*} , {*a1*, *b2*} , {*a1*, *c1*} , and {*a1*, *c2*}. Finally, all partial results are merged to generate the final twig results {*a1*, *b1*, *c1*}, {*a1*, *b2*, *c1*}, {*a1*, *b1*, *c2*}, and {*a1*, *b2*, *c2*}.

From above example, we observe that TwigStack uses the stack structure and pointers between them to encode relationships of elements, it ensures that one node is guaranteed to be an ancestor for nodes above it in the same stack, and is also an ancestor for nodes in descendant stacks which point to it. This encoding scheme not only avoids generating large intermediate results, but also reduces the query processing cost.

However, TwigStack has to check the element index to make sure all branches are satisfied before a new element comes. such as *a1* and *a2* in the above example. Moreover, for using stacks and pointers between them to encode relationships, TwigStack encounters obstacles when following-sibling relationships exist in the query. So it has to output the partial results at first, and then merge them to get the final twig results. Motivated by the observation, we want to recover the tree structure of elements. Based on the tree structure and the pointers between them, the positional relationships between elements can be greatly embodied, and the final twig matches can be easily encoded. Similar to stacks in TwigStack, we associate a forest with each query node in Track, and each forest consists of several trees stored in ordered sequence. Consider the example in Figure 2, we build F_a, F_b, and F_c for all query nodes A, B, and C. In addition, pointers between forests are built to connect elements in different forests.

DEFENITION 3.1 Given two elements n_1 and n_2 in the same forest F_n, iff n_2 is an descendant of n_1 in the forest, n_2 must be an descendant of n_1 in the original document.

DEFENITION 3.2 Given two elements p_1 and n_1 in forest F_p and F_n, iff p_1 points to n_1, or implicitly points to n_1, n_1 must be an descendant of p_1 in the original document. (Note that if p_1 points to an ancestor of n_1, it implicitly points to n_1)

Based on TwigStack, some improvements have been made to identify a larger query class or to modify it to adapt to various labeling schemes (Jiang 2004, Lu 2004). A generic algorithm, called TSGeneric+ (Jiang 2003) was proposed, which could skip redundant elements based on extra indexes built on element labels. Lu proposed TwigStackList which took the level information of elements into account, and resulted in less intermediate matches for queries with both ancestor-descendant and parent-child axes(Lu 2004). Other researches have focused on identifying a larger optimal query class (Jiang 2004, Lu-Ling 2005).

On the other hand, Chen studied the limitation of basic region scheme (Chen 2005), and then used both tag and level information to partition the elements, this strategy avoids unnecessary scan for irrelevant portion of XML documents, and reduces useless matches. Because *Region scheme* only encodes the information of a single label, Lu proposed a novel algorithm TJFast by using a new labeling scheme (Lu 2005), which called *extended Dewey*. *extended Dewey* enables us to derive all the element names from the root to current element alone from the label of current element. For example in Figure 2, we get the ancestor sets {*a1, a2*} and {*a1*} for *b1* and *c1*. Hereafter, the intersection {*a1*} are generated to get the final

match {*a1, b1, c1*}. It is clear that TJFast merely accesses leaf elements which leads to the reduction of I/O operations.

4 Track

In this section, we start with an introduce to notations used in our approach, and then describe the main algorithm in TRACK. Finally, we describe the procedure of building the tree structure and processing one axis in details.

4.1 Notation

Let Q denote a tree pattern. We use following query node operations: *isleaf*(n), *parent*(n), and *directBranchOrLeaf*(n). Function *directBranchOrLeaf*(n) returns the set of all branching nodes and leaf nodes such that in the path from n to those nodes there is no branching nodes, e.g., the twig query in Figure 3, *directBranchOrLeaf*(A)={B}, and *directBranchOrLeaf*(B)={C,D}.

We define a data stream S_q and a forest F_q associated with node q in the tree pattern Q. The stream S_q contains the region scheme labels of document elements that match the twig query node q, and these labels are sorted by their (doc, start) values. Two operations over S_q are: *end*(S_q), and *current*(S_q). For the forest F_q, we define seven functions: *isempty*(F_q), *eof*(F_q), *current*(F_q), *isroot*(n), *ancestorpath*(n, F_q), *RecoverForest(q)*, and *DeleteNode(q, n)*, where n is a node in F_q. *isroot*(n) tests whether n is a root node of a tree in F_q, while *ancestorpath*(n, F_q) returns the set of ancestor nodes for n in F_q. For example in Figure 3, *ancestorpath*(*d1*, F_b)={*b1*} and *ancestorpath*(*d3*, F_d)={*d1*, *d3*}.

In this paragraph, we describe the forest operation *RecoverForest*() in details. Since elements in streams are stored in the *pre-order* sequence, the tree structure is recovered in a *root-to-leaf* manner. When a new element comes, we only need to check the ancestor path of last node which has been added in the forest, and find the right position to insert the element. If no ancestor is found, a new tree is built. Consider the example in Figure 3 again, after *d2* has been inserted into F_d, we sequentially check whether *d3* is the descendant of the set *ancestorpath*(*d2*, F_d)={*d2*, *d1*}. Finally, attach *d3* to node *d1*. When *d4* comes, *d3* and *d1* are checked, but none of them are ancestors of *d4*. So we initialize a new tree whose root is *d4*.

ForestOperation

Function *RecoverForest*(QueryNode q)
 while(¬end(q))
  let n denote last node added, check
  ancestorpath(n, F_q) to find ancestors
  of current(S_q);
  if we find at least one ancestor, attach
  current(S_q) to the tree;
  else create a new tree;

Function *DeleteNode*(Forest q, FNode n)
 if(*isroot*(n))
  break up Tree n into several trees whose
  roots are children nodes of n;
 else
  attach children nodes of n to the parent
  of n;

In GTrack, we also need to delete nodes from the forest F_q by using *DeleteNode(q, n)* for the branching node q. Supposing node n is deleted from the forest, if n is a root node of a tree in F_q, we have to break up tree n into several trees whose roots are children nodes of n; or else attach children nodes of n to the parent of n.

4.2 Main Algorithm

In this section, we put together all functions defined before, and depict the whole procedure of Track. Given a query Q, we first recursively partition Q into individual query paths, and then process these query paths separately in a reverse *leaf-to-root* sequence by using function *ProEdge*(). Lastly when encountering branching nodes, we use function *ProBEdge*() to modify the tree structure of branching nodes to satisfy all children constraints. Detailed description for *ProEdge*() and *ProBEdge*() are left in the next section.

Algorithm *Track*(*root*)
 foreach node $n \in$ *directBranchOrLeaf*(*root*)
  Parse(n, *root*)
 ShowSolution()

Function *Parse*(n, b)
 if(¬ isleaf(n))
  foreach node $q \in$ *directBranchOrLeaf*(n)
   Parse(q, n)
 else *RecoverForest*(n)
 while(*parent*(n) $\neq b$)
  ProEdge(n, *parent*(n))
  n = *parent*(n)
 if(*isempty*(F_b)) *ProEdge*(n, b)
 else *ProBEdge*(n, b)

An inherent property of Track is that nodes in the upper tree structure must satisfy all subtree constraints. In other words, nodes in the upper tree structure are more selective than the lower ones. Hence, we make use of *ShowSolution*() to output twig matches encoded in the tree structure in a *root-to-leaf* manner. For each node in F_{root}, *ShowSolution*() traverses the forests through pointers between them in the *root-to-leaf* manner. Consider Figure 3 again, after the forests are built, *ShowSolution*() outputs all matches, {*a1*, *b1*, (*d1*,*d2*,*d3*,*d4*), *c1*}.

An immediate method for processing parent-child edges in query path patterns is to change *ShowSolution*(), which traverses the forests through pointers between them, and meanwhile checks the depth information of root nodes for outputs. In Figure 3, the query is changed to $A/B[./D]/C$. When function *ShowSolution*()is invoked, we get possible matches {*a1*, *b1*, (*d1*,*d4*), *c1*}, and then check them for the final result {*a1*, *b1*, (*d1*,*d4*), *c1*}.

I/O time of Track is no more than TwigStack, which needs to read elements in the entire input lists. In our approach, nodes in the upper tree structure are more selective than the lower ones, which means we only need to read elements which satisfy the subtree constraints. On the other hand, compared with TwigStack, whose memory requirement is just proportional to the document depth, Track may keep all elements that have the label matching with the query in the worst case. However, because nodes in the upper tree structure is very selective in practice, merely a small portion of elements are stored in the memory.

4.3 Processing the Query Axis

In Track, we first builds the tree structure for leaf nodes, and then traces *leaf-to-root* pathes step by step

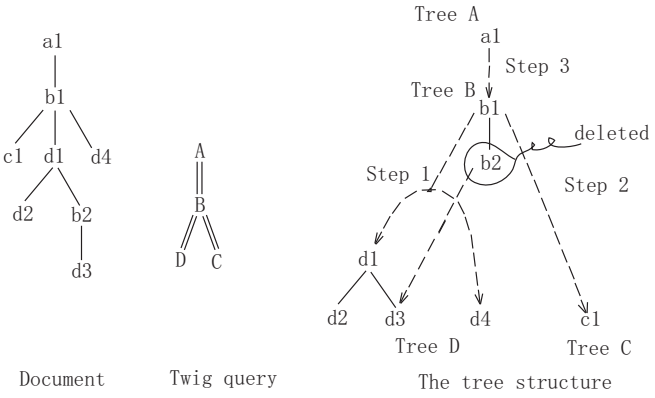

Figure 3: *An example for Track*

back to the root. In each step, function *ProEdge*() is called to process an individual axis $n//p$. In *ProEdge*(), the main task is to recover the tree structure F_p for the parent node p, and establish links to represent ancestor-descendant relationships between nodes in both forests. We first fetch nodes in forest F_n in the *pre-order* sequence, and then find all corresponding ancestors for current(F_n) in each recursion. When current(F_n) is a root node, we first check the ancestor path of last tree in F_n. If at least one ancestor exists, we merge the last tree and the current one in F_n, and set pointers of ancestors to the new tree. After it, we check S_p for left ancestors still stored in the stream, add them into the right position in F_p and set pointers of them point to current(F_n). When current(F_n) is not a root node, we only check S_p.

We take the query step $D//B$ in Figure 3 as an example. When *d1* being visited, we find the only ancestor *b1*, add it into F_b, and set the pointer of *b1* to *d1*. When *d2* or *d3* comes, we only check S_p, and the former finds no ancestor and the latter finds *b2*. As *d4* comes, we check the set *ancestorpath*(*d1*, F_b)={*b1*}, and then merge last tree *d1* and the current one *d4*. Finally, set the pointer of *b1* to the new tree.

Function *ProEdge*(QNode n, QNode p)
while($\neg eof(F_n)$)
  if($isroot(current(F_n))$)
    let root denote the root of last tree in F_n, check *ancestorpath*(*root*, F_p) to find ancestors of $current(F_n)$;
    if ancestors exist, merge the last tree and the current one in F_n, and set pointers of ancestors point to the new tree;
  if($current(F_n).start < current(S_p).start$)
    forward $current(F_n)$ to the next node in the pre-order sequence;
  elseif($current(F_n).start > current(S_p).end$)
    forward $current(S_p)$ to the next element;
  else
    add $current(S_p)$ to be a child of the nearest ancestor in F_p, if no ancestor exists, create a new tree;
    set the pointer of $current(F_p)$ point to the node $current(F_n)$;
    forward $current(S_p)$ to the next element;

Suppose that the forest of the child node has been recovered in previous steps, we only build the tree structure for the parent node in current step. However, when the parent node is a branching node, we have to consider the case in a special way. Such as node B in Figure 3, before processing the axis $C//B$,

Function *ProBEdge*(QNode n, QNode p)
while($\neg eof(F_n)$)
  if($isroot(current(F_n))$)
    let root denote the root of last tree in F_n, check ancestorpath(root, F_p) to find ancestors of current(F_n);
    if ancestors exist, merge last tree and the current one in F_n, and set pointers of ancestors point to the new tree;
  if($current(F_n).start < current(F_p).start$)
    forward current(F_n) to the next node in the pre-order sequence;
  elseif($current(F_n).start > current(F_p).end$)
    forward $current(F_p)$ to the next element;
    $DeleteNode(F_P, current(F_P))$;
  else
    set the pointer of $current(F_p)$ point to the node $current(F_n)$;
    forward $current(F_p)$ to the next element;

the forest F_b has been built when processing $D//B$, so we only need to delete nodes in F_b which do not satisfy the added query constraints. Meanwhile, build links between nodes in F_b and F_c.

We denote the modified function to handle the case as *ProBEdge*(). Compared with *ProEdge*(), *ProBEdge*() first replaces all $current(S_p)$ with $current(F_p)$. On the other hand, after forwarding $current(F_p)$ to the next element as $current(F_n).start$ is larger than $current(F_p).end$, which means $current(F_p)$ does not satisfy the added constraints, we use function *DeleteNode*() to delete $current(F_p)$ from F_p. Finally, we need not insert $current(F_p)$ which has been already in F_p when added constraints are satisfied.

Figure 3 presents a complete running example of Track. We first recover tree structures for leaf query nodes C and D by using function *RecoverForest*(), and then call function *ProEdge*() to process the axis $D//B$. Note that the detailed description has been showed in above paragraphs, we omit it here. Thereafter, we modify F_b to satisfy the axis $C//B$ in *ProBEdge*(). When *c1* comes, we find only one ancestor *b1* in existing forest F_b, and use function *DeleteNode*() to delete *b2* from F_b. Finally, we process the axis $B//A$, and recover the forest F_a which has one element *a1*.

5 Experiment

We compare two twig join algorithms, TwigStack and TJFast with Track in C++ on a 1.73G Pentium IV processor running Windows XP with 512MB of main memory. TwigStack is a holistic algorithm which is proved to be quite efficient, especially when queries contain only ancestor-descendant relationships. TJFast adopts a new extended Dewey labeling scheme which can derive all the element names from the root to current element alone from the label of current element. Hence, the method merely needs to access leaf elements which leads to the reduction of I/O operations.

5.1 Data Sets and Query Type

Four representative data sets are used for the experimental evaluation, including two real data sets *DBLP* and *TreeBank*, and two synthetic data sets *XMark* and *Random*. *DBLP* is a wide and shallow data set, with the size 127MB and 3.3 millions elements, while

Table 1: *Queries*

	Data set	Path Queries
Q1	DBLP	/dblp/inproceedings[./title]/author
Q2	DBLP	/dblp/article[./author][./title]//year
Q3	DBLP	/dblp/inproceedings[./author][.//title]//booktitle
Q4	TreeBank	//S/VP//PP[.//NP/VBN]/IN
Q5	TreeBank	/S[.//VP/IN]//NP
Q6	TreeBank	//VP[./DT]//PRP_DOLLAR
Q7	XMark	/site/open_auctions[.//bidder/personref]//reserve
Q8	XMark	//people//person[.//address/zipcode]/profile
Q9	XMark	//item[./location]/description//keyword
R1	Random	//A1//A2//A3//A4
R2	Random	//A1//A2//A3[.//A4]
R3	Random	//A1//[.//A4//A5]//A2//A3

TreeBank has a deep recursive structure which makes it an interesting case for experiments, the file is 82MB and has 2.4 millions elements. *XMark* is a well-known mark data with the size 111MB and 3.0 millions elements, while *Random* is a data set with twenty labels *A1*, ... , *A20*, which are uniformly distributed. We generate Random by using two parameters: depth, and fan-out. For all experiments in this paper, we only generated full binary and ternary trees. We also increase the size from 30MB to 150MB by 30MB for the scalability analysis.

Table 1 shows the twig queries used for the experiment. For each data set, three queries are selected. These queries have diverse structures and combine both ancestor-descendant relationships and parent-child relationships to give a more practical evaluation.

5.2 Experiment and Evaluation

Figure 4 shows the performance of three algorithms for processing different twig queries. For each twig query, we record the query processing time and I/O operations for all three algorithms.

TwigStack vs Track We first compare TwigStack with Track. From Figure 4, we observe that our algorithm outperforms TwigStack for all types of selected queries, it is always 3-5 times faster than TwigStack, but I/O operations of both algorithms are nearly the same. Hence, we conclude that the I/O operation does not dominate the query processing time. For Q1-Q3 in *DBLP* in Table 1, because *DBLP* has a steady structure, TwigStack generates almost no useless partial results, so the performance degradation of TwigStack is primarily due to the checking mechanism and enumerating of partial matches for the final twig results. Because *TreeBank* has a deep recursive structure, Q4-Q6 generate a great amount of useless intermediate results, which aggravate the degradation of TwigStack. Like *DBLP*, *XMark* also has a steady structure , so Track shows a similar order of magnitude performance gain over TwigStack for Q7-Q9.

TJFast vs Track Since TJFast only needs to access leaf elements, I/O operations of TJFast are relatively fewer than the other two algorithms. However, the new labeling scheme not only increases the size of the index, but also adds the difficulty to easily process different query classes. For Q1-Q3 in *DBLP*, TJFast saves 14%-17% of I/O operations, but Track is 5-6 times faster than TJFast. The reason might be explained that TJFast needs to analyze the /emphextended Dewey ID using the DTD transducer, and enumerate intermediate results for final twig results. For Q4-Q6 in *TreeBank*, TJFast saves many I/O operations compared with Track, but wastes too much time analyzing the extended Dewey ID because of the recursive structure of *TreeBank*. TJFast shows the best performance for Q7-Q9 in *XMark*. For each query, about 50% of I/O operations are saved, but Track is still 2 times faster.

5.3 Scalability and Stability

We first report the scalability of Track in term of the size of the synthetic data set *Random*. Figure 5 shows three representative queries which are selected for testing the scalability of three algorithms as the size increases from 30MB to 150MB. Clearly, all algorithms grows linearly in terms of the document size. TJFast shows a little advantage to process the path query R1, but for R2 and R3, Track has much better query processing time.

Intuitively, changing the processing sequence of twig queries may influence the query processing time in Track. Taking Q2 in Table 1 as an example, we can process the query path title-author-article, or year-article first. For each query of Q4-Q9 in Table 1, we show the influence in Figure 5. An immediate observation is that changing the sequence nearly does not influence the efficiency, so Track is stable in most cases.

6 Conclusion and Future Work

In this paper, we propose a novel twig join algorithm Track, which selectively recovers the tree structure of elements for the same query node and uses links between forests to compactly represent final results for the twig query. Experiment evaluation shows that our algorithm is quite efficient in most cases.

There are some promising directions in the future. Our algorithm processes all axes using the structural join algorithm. However, the structural join algorithm is inefficient for processing parent-children axes. Enlighten by the extended Dewey labeling scheme in TJFast, we propose a new labeling scheme, which limits the ancestor path information for each element to its parent element alone. Combined with Track, it not only maintains the index in an acceptable size, but also significantly accelerates the processing of parent-children axes.

Acknowledgement. This work was supported by Chinese 973 Research Project under grant No.2002CB312006.

References

Cooper, B., Sample, N., Franklin, M., Hjaltason, and G., Shadmon, M. (2001), A Fast Index For

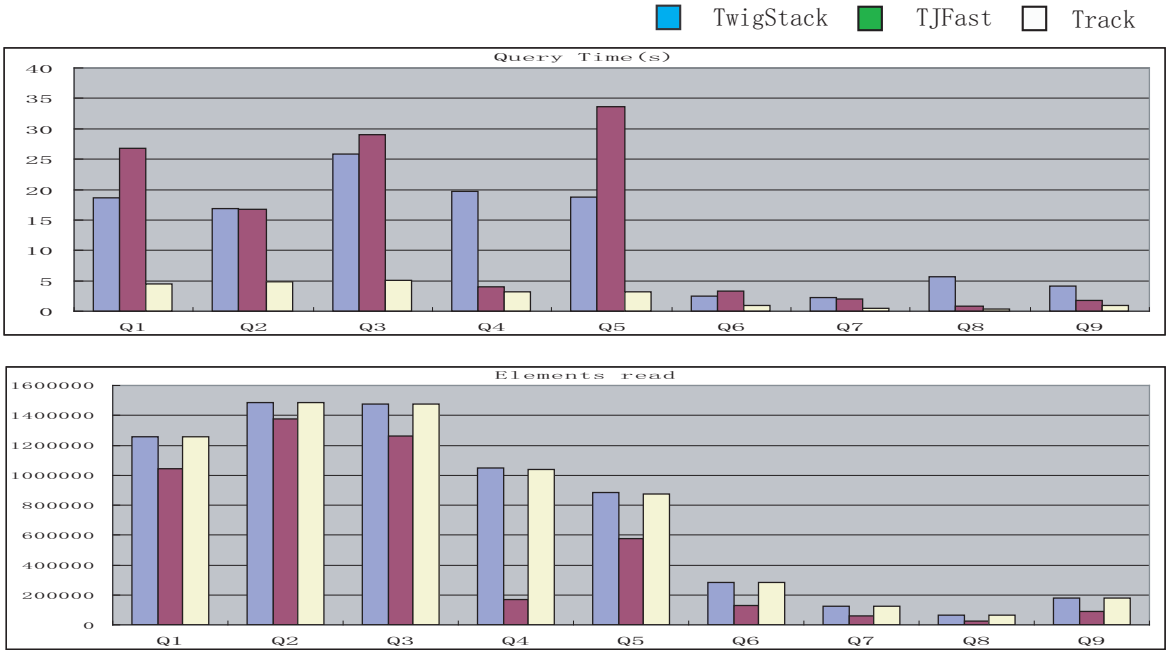

Figure 4: *Experimental evaluation*

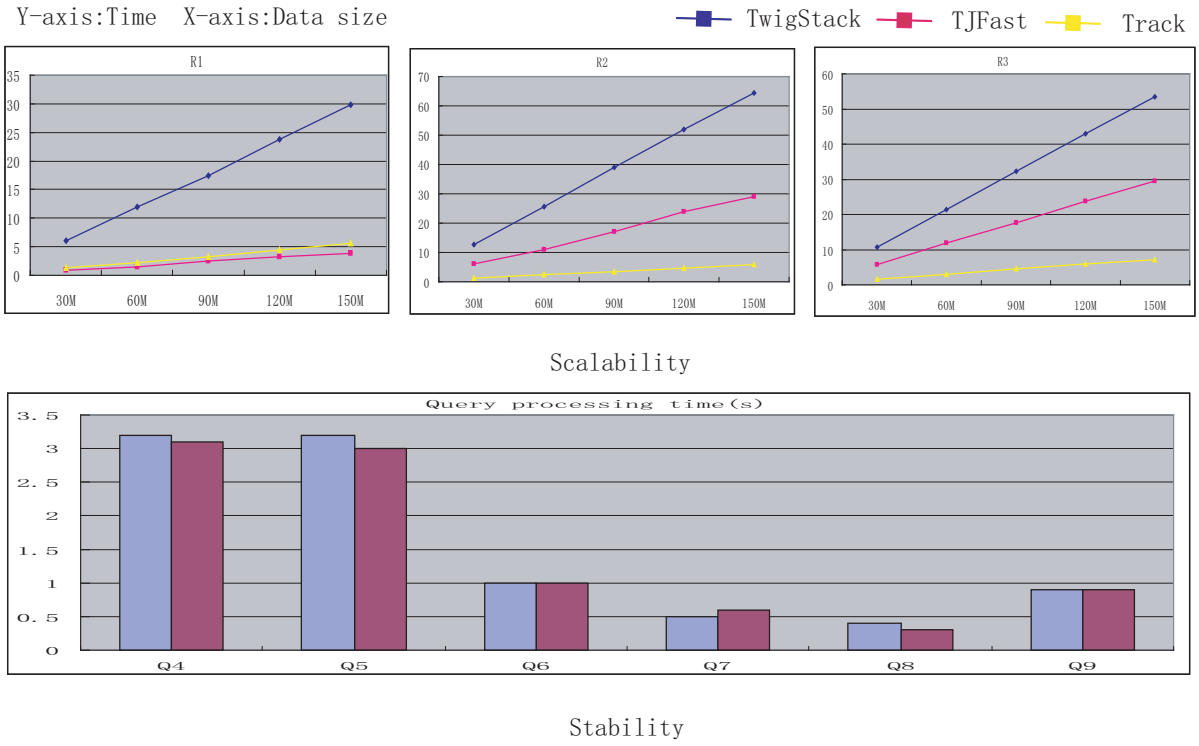

Figure 5: *Scalability and Stability*

Semistructured Data. In *Proceedings of VLDB*, pages 341-350.

Chung, C., Min, J., and Shim, K., (2001), APEX: An Adaptive Path Index for XML Data. In *Proceedings of SIGMOD*, pages 121-132.

Zhang, C., Naughton, J., Dewitt, Q.Luo, D., and Lohman, G.(2001), On Supporting Containment Queries in Relational Database Management Systems. In *Proceedings of SIGMOD*, pages 425-436.

Salton, G. and McGill, M.J.(1983), Introduction to modern information retrieval. McGraw-Hill, New York.

Jiang, H., Lu, H., and Wang, W.(2004), Efficient Processing of XML Twig Queries With OR-predicates. In *Proceedings of SIGMOD*, pages 274-285.

Jiang, H., Wang, W., and Lu, W.(2003), Holistic Twig Joins on Indexed XML Documents. In *Proceedings of VLDB*, pages 273-248.

Lu, J., Chen, T., and Ling, T.(2004), Efficient Processing of XML Twig Patterns with Parent Child edges: A Look-Ahead Approach. In *Proceedings of CIKM*, pages 533-542.

Lu, J., Ling, T., Yu, T., Li, C. and Ni, W.(2005), Efficient Processing of Ordered XML Twig Pattern Matching. In *Proceedings of DEXA*, pages 300-309.

Lu, J., Ling, T.W., Chan, C.Y., and Chen, T.(2005), From Region Encoding To Extend Dewey: On Efficient Processing of XML Pattern Matching. In *Proceedings of VLDB*, pages 193-204.

Bruno, N., Koudas, N., and Srivastava, D.(2002), Holistic Twig Joins: Optimal XML Pattern Matching. In *Proceedings of SIGMOD*, pages 310-321.

Goldman, R., and Widom, J. (1997), DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of VLDB*, pages 436-445.

Kaushik, R., Bohannon, P., Naughton, J., and Korth, H.(2002), Covering indexes for branching path queries. In *Proceedings of SIGMOD*, pages 133-144.

Kaushik, R., Shenoy, P., Bohannon, P., and Gudes, E.(2002), Exploiting Local Similarity for Efficient Indexing of Paths in Graph Structured Data. In *Proceedings of ICDE*, pages 129-140.

Al-Khalifa, S., Jagadish, N.Koudas, H., Patel, J., Srivastana, D., and WU, Y.(2002), Structural joins: A Primitive for Efficient XML Query Pattern Matching. In *Proceedings of ICDE*, pages 141-152.

Chen, T., Lu, J. and Ling, T., (2005), On boosting holism in XML Twig Pattern Matching Using Structural Indexing Techniques. In *Proceedings of SIGMOD*, pages 455-466.