Signature-based Filtering Techniques for Structural Joins of XML Data

Huan Huo Guoren Wang Chuan Yang Rui Zhou
Northeastern University, Shenyang 110004, China

Abstract

Queries on XML documents typically combine selections on element contents, and, via path expressions, the structural relationships between tagged elements. Efficient support for structural joins is thus the key to efficient implementation of XML queries. With a stack to keep ancestor-descendant structural relationships, stack-tree join algorithm enhances the performance of structural joins by reducing deducible unnecessary comparisons. However, stack-tree join cannot prevent "unwanted" comparisons between elements that do no participate in the join. To solve this problem, we propose a signature filter, which takes advantage of encoding schemes proposed for XML and occupies a little space. Then we present an pointer-based signature filter to skip the "unwanted " elements. In order to further improve the filtering efficiency, we finally propose an optimized pointer-based filter with the conjunction of two signatures. Performance study shows that our signature-based filters have excellent filtering performance and significantly improve the performance of structural joins.

1 Introduction

XML [1]is emerging as a *de facto* standard for information representation and data exchange on the web. It can be represented as a tree-structural model with data contents and their structural relationships. Evaluating the primitive structural relationships, parent-child and ancestor-descendant, is thus the key for XML query processing.

Path expression is used to describe XML query and plays an important role in XML query languages like XPath [2] and XQuery [3]. The key technique for expediting path expression processing is evaluating query operations like "/"(parent-child relationship) and "//"(ancestor-descendant relationship). One simple mechanism for path expression evaluation is traversing the XML data tree. However, the performance varies a lot with the size of XML document [10]. It is quite possible to traverse the whole XML data tree with only a few results. Consequently, Structural join algorithm [4] is proposed.

Recently, many research works [4–8] focus on structural join algorithms to evaluate "/" and "//" operations. Given a path expression "*a*//*d*", firstly two sets of candidate nodes of *a* and *d* are created separately as *A-List* and *D-List*, and then all "*a*-*d*" pairs matching "//"are output from the two lists, in which every element is encoded as (*StartPos*,*EndPos*) pair. Stack-tree join, as a widely used structural join algorithm, is given in [4].

Stack-tree join algorithm uses a stack to keep deducible structural relationships and reduces the comparisons that can be deduced with the help of stack. For example, given two path expressions "a//b" and "b//c", stack-tree join algorithm gets the results of "a//c" without extra comparison, since the stack at all times has a sequence of ancestor nodes, and each node in stack is a descendant of the node below it. Therefore, deducible comparisons are implied by the sequence of elements in stack and need not attending stack-tree join algorithm. We name such comparisons as deducible comparisons.

However, stack-tree join cannot avoid the comparisons that produce no join results, which we name as "unwanted" comparisons. So far several approaches have been proposed with index on how to avoid these unwanted comparisons, such as B+-tree [7] and XR-tree [8]. These enhanced structural join algorithms gain better performance on CPU time. But the I/O cost of complex indices becomes another problem. Hence signature filter [13, 14], which uses a sequence of bits identifying the elements or element sets to shed the unwanted elements as early as possible, shows promising performance with the merits of little space cost and reducing most of the unwanted comparisons.

In this paper, we firstly develop a simple signature filter, which is generated for every candidate element set, according to the range of element code (*StartPos, EndPos*). As a shortcut of element list, signature filter enhances the efficiency of structural joins by comparing element signature and signature filter to test "//"("/") relationship beforehand so as to avoid unwanted comparisons in the stack. However, considering signature filter still scans all the elements in the input lists, we then build a pointer-based signature filter, which makes comparison between two signature filters and adds a pointer to every "1" bit in the filters to skip unnecessary element accesses to the lists. In order to further improve the filtering efficiency, we finally propose an optimized pointer-based filter with the conjunction of two signatures. Based on these three types of filters, we design corresponding structural join algorithms.

The rest of this paper is organized as follows. Section 2 presents background and some important definitions such as structural joins and signature filter. Section 3 describes the basic idea of signature filter and its structural join algorithm. Section 4 gives a detailed statement of our pointer-based signature filter, as well as corresponding structural join algorithm, and its optimization. Section 5 shows experimental results. Our conclusions are contained in Section 6.

2 Background and Related Work

Structural join algorithm takes advantage of XML encoding representation to efficiently match "parent-child" and "ancestor-descendant" relationships. In this section, we begin with presenting XML encoding approaches, then overview the related work of structural join algorithms and finally introduce signature filter.

2.1 XML Encoding Representation

XML database is typically modelled as a tree. By mapping an element (or string value) on the tree to an n-tuple code, XML encoding approaches [5, 6] can directly represent the positions of elements and string values so as to clearly reflect the relationship between data.

In a widely accepted encoding approach [6], the position of an element is represented as a 4-tuple (*DocId, StartPos, EndPos, Level*), where (i) *DocId* is the identifier of the document, which can be omitted if one single document involved (in this paper, only single document is considered and *DocId* is thus omitted.Operations are similar in multi-document); (ii) *StartPos* is the number given in a pre-order traversal of the tree and *EndPos* is the number given in a post-order traversal of the tree, and *StartPos* and *EndPos* are the same when the node is a leaf; (iii) and *Level* is the nesting depth of the element (or string value) helping to identify "parent-child" relationship. Figure 1 gives an example of XML data tree with (*StartPos, EndPos*) representation.

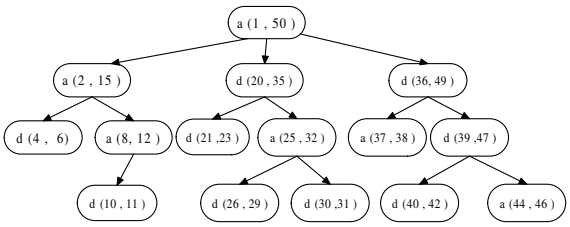


Figure 1. Sample XML data tree

Taking *StartPos* and *EndPos* as a range of element node, we can see that the ranges of two arbitrary element codes are either inclusive or exclusive. And, we can get the ascendant-descendant relationship between nodes by testing if the ranges of element nodes are inclusive.

Node *a* is the ascendant of *d* if and only if $a.StartPos < d.StartPos$ and $d.EndPos < a.EndPos$. If $a.Level + 1 = d.Level$, then *a* is the parent of *d*.

2.2 Structural Joins

Conceptually, structural join algorithm is used to find all structural relationships. Take "$a//d$" for example, supposing the candidate element nodes of a and d are in the set of *A-List* and *D-List* respectively, we have to find all the "$a//d$" pairs matching ancestor-descendant relationship, which means to evaluate the structural joins between ancestor list *A-List* and descendant list *D-List*. As described in Section 2.1, all pairs of "a-d" satisfying $a.StartPos < d.StartPos$ and $d.EndPos < a.EndPos$ are the results.

Stack-tree join [4] is one of the widely accepted structural join algorithms. The input lists *A-List* and *D-List* in stack-tree are sorted by *StartPos* attribute and *A-List* is managed by a stack assuring only one traversal on both *A-List* and *D-List*. As described in Section 1, the deducible comparisons can be skipped by stack. However, all the elements in *A-List* and *D-List* are compared and possibly some of them produce no results. Therefore, unwanted element accesses reduce the efficiency of stack-tree join.

2.3 Signature Filter

In database systems, filtering [11, 12] is a key technique to improve the performance of join algorithms by avoiding unwanted elements evaluation. The basic idea is to generate a filter when scanning the elements in one set, and then filter out the useless elements in another set.

Signature filter [9] is applied for hash join algorithms in object-oriented database systems. A signature is generated by hash function as follows: hash the join attribute when an object is accessed and return a bit vector composed of "0" or "1", which is the signature of the object; and then add up all the signatures of the objects in the set by "$\vee$" to get the signature of the set. We call the signature of the set is the signature filter of the join operation.

As shown in Figure 2, all the objects (a_1, a_2, a_3) get the signatures by hash function and *A-Filter* (i.e. 110 110 111 110) is the filter of the join generated by superimposing all the signatures of objects in A. When scanning set D, we evaluate the signature of object d_i by the same hash function on the join attribute, and name the signature S_{di}. All

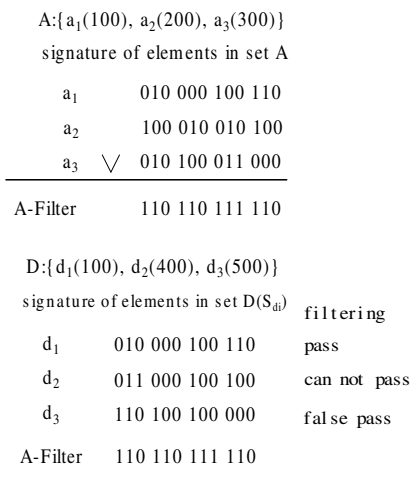


Figure 2. Signature filter in ODMS

signatures of the elements in set D are listed in Figure 2.

If *A-Filter* $\vee$ $S_{di} \neq$ *A-Filter*, object d_i is filtered out; if *A-Filter* $\vee$ $S_{di} =$ *A-Filter*, object d_i passes the filter. For example, as for d_1, it can take the join operation and produce the join results since *A-Filter* $\vee$ $S_{d1} =$ *A-Filter*; as for object d_2, it is filtered out because *A-Filter* $\vee$ $S_{d2} \neq$ *A-Filter*; as for object d_3, it passes the filter like d_1 but the joins with d_3 produce no results, which is named false pass.

The filtering performance can be evaluated by filtered-out rate and false pass rate. Filtered-out rate is the probability that an object which produces no results is filtered out while false pass rate is the probability that a passed object failed to produce any results.

3 Signature-Based Filter for Structural Join

In this section, we propose a range-based signature filter according to an XML encoding scheme and present a novel structural join algorithm based on it.

3.1 Range-based Signature Filter

There have been a lot of ways to form a signature filter such as hashing. Considering that structural join algorithm falls back on XML encoding representation, we take

advantage of XML encoding technique and put forward a range-based signature filter.

Definition 1 *Divide the code range of an XML document into m consecutive equal-length intervals, each of which is represented by a bit of a vector. If an element code range intersects with the interval or intervals, we set the corresponding bit or bits of the vector to "1" and name it as the* range signature *of the element and the length of the signature is m.* Range-based signature filter *is defined as the sum ("$\vee$") of all the element signatures.*

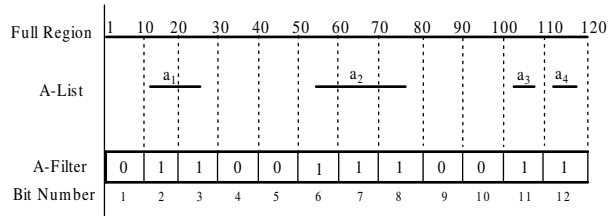


Figure 3. Range-based Signature Filter

Figure 3 shows the construction of range-based signature filter. Set *A-List* comprises four elements: $\{a_1(12, 22), a_2(55, 75), a_3(105, 108), a_4(115, 119)\}$, each of which is represented by a line to describe the range of the element (*StartPos,EndPos*). And the range of the whole document $(1, 120)$ is divided into 12 segments, each of which is represented by one bit. Element $a_1(12, 22)$ intersects with intervals $[11, 20]$ and $[21, 30]$, so the $2nd$ and the $3rd$ bits of the signature are set to "1"; element $a_2(55, 75)$ intersects with intervals $[51, 60]$, $[61, 70]$ and $[71, 80]$, so the $6th$, the $7th$ and the $8th$ bits are set to "1"; likewise, the $11th$ and the $12th$ bits are set to "1" according to a_3 and a_4. So we get the signature filter *A-Filter* of *A-List* (i.e. 0110 0111 0011).

Filtering The Descendant Elements. Given *A-Filter* (0110 0111 0011), when scanning set *D-List*, we get element d_i and its signature S_{di}. If *A-Filter* $\vee S_{di} \neq$ *A-Filter*, d_i is filtered out; if *A-Filter* $\vee S_{di} =$ *A-Filter*, d_i passes the filter. In other words, the ancestors of d_i may exist in *A-List* only if the corresponding bits of d_i in *A-Filter* are "1", so d_i passes the filter.

As shown in Figure 4, the corresponding bit for d_1 is "1", so *A-Filter* $\vee S_{d1} =$ *A-Filter* and element d_1 passes the filter; the corresponding bits for d_2 and d_3 are "0", so *A-Filter* $\vee S_{di} \neq$ *A-Filter* and elements d_2 and d_3 are filtered out; and the corresponding bits for d_4 is "01", so *A-Filter* $\vee S_{d4} \neq$ *A-Filter* and element d_4 is filtered out.

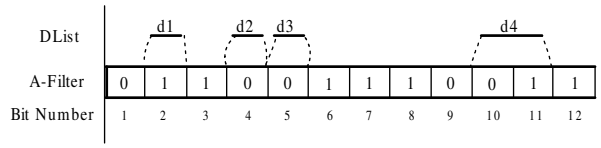


Figure 4. Filtering the descendant elements

Filtering The Ancestor Elements. Filtering the ancestor elements is similar to filtering the descendant elements. We use *D-Filter* (0101 1000 0110) to filter the elements in *A-List*, and if *D-Filter* $\wedge\ a_i = 0$, a_i is filtered out while if *D-Filter* $\wedge a_i \neq 0$, a_i passes the filter. In other words, the descendants of a_i may exist in *D-List* only if one of the corresponding bits of a_i in *D-Filter* is "1".

As shown in Figure 5, the corresponding bits for a_1 is "10", so *D-Filter* $\wedge S_{a1} \neq 0$ and element a_1 passes the filter; the corresponding bits for a_2 are all "0", so *D-Filter* $\wedge S_{a2} = 0$ and element a_2 is filtered out. Obviously, a_3 passes the filter and a_4 is filtered out.

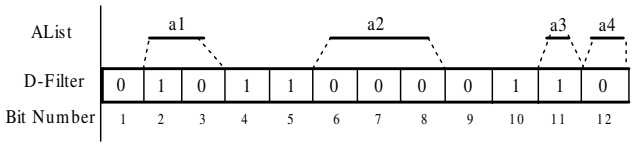


Figure 5. Filtering the ancestor elements

3.2 Structural Join Algorithm with Range-based Signature Filter

In stack-tree join algorithm, *A-List* and *D-List* are scanned once respectively [4]. In our approach, the filter scans *A-List* and *D-List* and returns to stack-tree join operation the elements that cannot be filtered out. Thus the structural join algorithm with signature filter only scans the two input lists once as well.

Here we firstly present two basic filtering algorithms: Algorithm 1, FilterDescendantElement() and Algorithm 2,

FilterAncestorElement(). The basic idea of the former is to filter a_i and the following elements by *D-filter* when scanning *A-List*, until bumping into an element a_i which cannot be filtered and then return. Similarly, the latter filters d_i and the following elements by *A-filter* when scanning *D-List*.

Algorithm 1 FilterDescendantElement()

Input: Signature filter *A-Filter* of *A-List*; $CurrentD$ in *D-List*;

Output: the next element in *D-List* which may produce join results;

Description:

1: $d\text{-}signature \leftarrow$ signature for $CurrentD$;
2: **while** !$D\text{-}List.end()$ **do**
3: **if** $A\text{-}Filter \vee d\text{-}signature = A\text{-}Filter$ **then**
4: **return** $CurrentD$;
5: **end if**
6: $CurrentD \leftarrow D\text{-}List.next()$;
7: **end while**

Algorithm 2 FilterAncestorElement()

Input: Signature filter *D-Filter* of *D-List*; $CurrentA$ in *A-List*;

Output: the next element in *A-List* which may produce join results;

Description:

1: $a\text{-}signature \leftarrow$ signature for $CurrentA$;
2: **while** !$A\text{-}List.end()$ **do**
3: **if** $D\text{-}Filter \wedge a\text{-}signature \neq 0$ **then**
4: **return** $CurrentA$;
5: **end if**
6: $CurrentA \leftarrow A\text{-}List.next()$;
7: **end while**

Algorithm 3 presents the entire structural join algorithm with signature filter. Given two ordered input lists *A-List* and *D-List*, the algorithm sets $CurrentA$ and $CurrentD$ as the first elements in two lists. Then it scans the two input lists respectively to perform the join operation till the end. To ensure only one traversal for each input list, the algorithm manages *A-List* by a stack. If the elements from a_i to a_j are the ancestors of d, the algorithm pushes the elements into the stack before performing join operation with d. In this way, the next element of d can be compared with elements of a in the stack without scanning *A-List* from a_i. This is similar to stack-tree algorithm but the difference lies in that the algorithm can filter out some ancestors and descendants which have no contributions for the join, as shown in step 8 and step 11.

Algorithm 3 Structural Join Algorithm based on Signature Filter

Input: ordered ancestor list *A-List*; ordered descendant list *D-List*;

Description:

1: $CurrentA \leftarrow$ the first element of *A-List*;
2: $CurrentD \leftarrow$ the first element of *D-List*;
3: **while** (!$A\text{-}List.end()$&&!$D\text{-}List.end()$) **do**
4: **if** (($CurrrentA.StartPos > stack[top].EndPos$) && ($CurrentD.StartPos > stack[top].EndPos$)) **then**
5: stack.pop();
6: **else if** ($CurrentA.StartPos < CurrentD.StartPos$) **then**
7: {stack.push($CurrentA$);
8: $CurrentA \leftarrow FilterAncestorElement(CurrentA)$;}
9: **else**
10: {output all pairs ($a \in stack$,$CurrentD$);
11: $CurrentD \leftarrow FilterDescendantElement(CurrentD)$;}
12: **end if**
13: **end while**

4 Pointer-Based Signature Filter for Structural Join

Although range-based signature filter reduces some elements attending stack-tree join, it still has to scan every element in two input lists. The range-based filtering principle replaces the comparisons between element codes with that between signature filter and the element signatures. In fact, it does not reduce the access to input elements.

In this section, we propose a pointer-based signature filter, in which we add a pointer to every bit with "1", and then build an enhanced filtering algorithm, which reduces the access to the input elements by replacing the comparison between signature filter and the element signatures with that between two signature filters.

4.1 Pointer-based Signature Filter

Range-based signature filter is a simple identification of the candidate elements, but it cannot locate the corresponding element after comparing two signature filters. To solve the problem, we propose pointer-based signature filter.

Definition 2 *Pointer-based signature filter is made of a bit vector and a pointer array. On the basis of range-based signature filter, a pointer is added to every bit of* "1" *in the filter pointing to the elements who set the bit* "1". *If more than one element sets the bit* "1" *repeatedly, the corresponding pointer of the bit points to the element with the minimal* StartPos *code.*

Figure 6 shows the construction of pointer-based signature filter : a_1 sets the $2nd$ and the $3rd$ bits of *A-Filter* to "1" and the pointers of the two bits point to a_1; a_2 sets the $6th$, the $7th$ and the $8th$ bits of *A-Filter* to "1" and the pointers of the three bits point to a_2; a_3 sets the $8th$ bit of *A-Filter* to "1" but the pointer of the bit points to a_2 not a_3, because the *StartPos* of a_2 is less than that of a_3; a_4 sets the $11th$ bit of *A-Filter* to "1" and the pointer of it points to a_4.

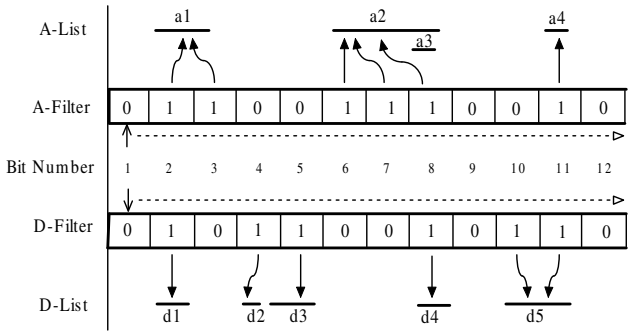


Figure 6. Pointer-based signature filter

Similarly, the $2nd$, $4th$, $5th$, $8th$, $10th$ and $11th$ bits of *D-Filter* are "1" and the corresponding pointers point to d_1, d_2, d_3, d_4, d_5, d_5. Among them, d_2 and d_3 set the $4th$ bit of *D-Filter* with "1" but the pointer of the $4th$ bit points to d_2 because the *StartPos* of d_2 less than that of d_3.

Different from comparing the signature filter with element signatures in range-based filtering principle, pointer-based signature filtering principle compares two signature filters. If the ith bit of *A-Filter* and that of *D-Filter* are all "1" (*A-Filter*$[i] \wedge$ *D-Filter*$[i] = 1$), there are probably join results produced by the elements pointed by the corresponding pointers of them. And the structural join algorithm begins from the elements pointed by the two pointers till all the elements who set the ith bits of *A-Filter* and *D-Filter* have been scanned. If the ith bits of *A-Filter* and *D-Filter* are not all "1"(*A-Filter*$[i] \wedge$ *D-Filter*$[i] \neq 1$), the join algorithm moves to the next bits of *A-Filter* and *D-Filter*.

Now we describe the filtering process of pointer-based signature filter (see Figure 6). The $1st$ bits of *A-Filter* and *D-Filter* are all "0", it moves to the next bits; the $2nd$ bits are all "1", it performs structural join of $\{a_1\}$ and $\{d_1\}$; scanning through the $3rd \sim 7th$ bits, it performs structural joins of $\{a_2, a_3\}$ and $\{d_4\}$ since the $8th$ bits are all "1"; similarly, it performs structural join of $\{a_4\}$ and $\{d_5\}$ scanning through the $9th \sim 10th$ bits.

4.2 Structural Join with Pointer-based Signature Filter

Structural join algorithm with range-based signature filter in Section 3.2 operates when getting an *Ancestor* or a *Descendant* element every time. While structural join algorithm with pointer-based signature filter operates by checking if the *Descendant* elements "offside" after it outputs the results every time.

Definition 3 *Supposing* StartPos *of* d_i *belongs to the interval that is represented by the* jth *bit in the signature, if* StartPos *of the next element* d_{i+1} *outruns the range, we call* d_{i+1} *is* offside.

Since the algorithm outputs the results in the order of input descendant elements, we can promise integrity of the join results by checking if there is any offside of descendant elements (not ancestor elements). In structural join algorithm, we use *Flag* to identify whether d_{i+1} is offside. When *Flag* is "true", the elements, which pass the signature filter, have finished the structural join operation. Before going on with the structural joins, we turn to compare the following bit pairs of two signature filters and then give the passing elements to stack-tree join operation.

Algorithm 4 gives the structural join algorithm with pointer-based signature filter, which only provides the elements that might produce join results to the algorithm. At the beginning, the algorithm scans two signature filters and sets the $Flag$ "true". In steps 21 and 22, if the next descendant element (d_{i+1}) is offside, then $Flag$ is set "true". Filtering is performed when $Flag$ is "true", as shown in steps $5 \sim 12$: it scans the signature filter until *A-Filter*[i]$\wedge$ *D-Filter*[i]$= 1$, then returns the element pointed by the pointer corresponding to the ith bit (*A-Pointer*[i] and *D-Pointer*[i]).

Algorithm 4 Structural Join Algorithm based on Pointer-based Signature Filter

Input: ordered ancestor list *A-List*; ordered descendant list *D-List*;
Description:

1: $CurrentA \leftarrow$ the first element of *A-List*;
2: $CurrentD \leftarrow$ the first element of *D-List*;
3: $Flag = true$;
4: **while** (!*A-List*.$end()$&&!*D-List*.$end()$) **do**
5: **if** ($Flag$) **then**
6: {
7: **while** ($A - Filter[i] \wedge D - Filter[i]! = 1$) **do**
8: $i = i + 1$;
9: **end while**
10: $CurrentA \leftarrow$ the element pointed by A-Pointer[i];
11: $CurrentD \leftarrow$ the element pointed by D-Pointer[i];
12: $Flag = false$;}
13: **else if** (($CurrrentA.StartPos > stack[top].EndPos$) && ($CurrentD.StartPos > stack[top].EndPos$)) **then**
14: stack.pop();
15: **else if** ($CurrentA.StartPos < CurrentD.StartPos$) **then**
16: {stack.push($CurrentA$);
17: $CurrentA \leftarrow$ A-List.next();}
18: **else**
19: {output all pairs ($a \in stack$,$CurrentD$);
20: $CurrentD \leftarrow$ D-List.next();
21: **if** ($CurrentD$ offside) **then**
22: $Flag = true$;}
23: **end if**
24: **end if**
25: **end while**

Structural join algorithm with pointer-based signature filter efficiently reduces the unwanted accesses to the elements. However, the pointers occupy lots of memory space if the number of bits with "1" is larger. So we first conjuncts (logical AND) the two signature filters to reduce the number of bits with "1" as possible. Then we compact the pointer vector by discarding all bits of "0".

Figure 7 gives the compacted one of the signature filter described in Figure 6. In Figure 6, the $2nd$, $8th$,$10th$ bits are all "1",which means that there are probably join results between the elements pointed by the corresponding pointers, so we only keep the pointers of them and delete others. Then we only need to operate on the elements pointed by these 3 pairs of pointers. When compacting process has finished scanning the signature filters, the filtering process is also finished.

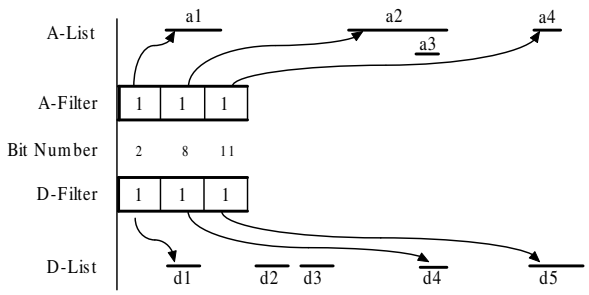


Figure 7. Compacting Pointer-based signature filter

The structural join algorithm with compacted signature filter is similar to Algorithm 4. The only difference is that the filtering operation has to be performed before structural join operation. The details are left and not discussed here.

5 Performance Evaluation

In this section, we present the performance evaluation of various structural join algorithms with different lengths of signatures, different queries and different sizes of documents on the same platform. We consider the following algorithms: stack-tree algorithm(STJ), structural join algorithm with range-based signature filter(S-Filter), structural

join algorithm with pointer-based signature filter(PS-Filter) and structural join algorithm with compacted pointer-based signature filter(CPS-Filter).

All experiments are run on a PC with 2.6GHz CPU, 512M memory and 80G hard disk. The operating system is WindowsXP. And we use Berkeley DB [15] to store XML index, programming with Microsoft Visual C++ 6.0.

We adopt DBLP [16] data set with five sizes (20M, 40M, 60M, 80M and 100M). Firstly we parse these documents and represent every position of element in code. Then we parse the documents again to get the candidate element set for each element. We generate signatures with different sizes (128 bits, 256 bits, 512 bits, 1024 bits, 2048 bits) for each element set. At last we design four query expressions, Q_1 (inproceedings//cite), Q_2(article//url), Q_3(article//ee) and Q_4(inproceedings//number), to examine the impact of different document sizes and different signature lengths on performance. The features of these queries are described as follows: only a few ancestor elements of Q_1 can produce join results while the majority of the descendant elements can produce join results; only a few descendant elements of Q_2 can produce join results while the majority of the ancestor elements can produce join results; Q_3 represents the usual cases, whose join results are output from moderate numbers of both ancestor and descendant elements; Q_4 is a particular case that only a few elements of ancestor and descendant can produce join results.

5.1 Performance vs Different Queries

Figure 8 shows the execution time of the four queries. In this experiment we adopt 100M DBLP data set and 1024-bit signature. We can see from Q_1 and Q_2 that *S-Filter* greatly enhances the performance when filtering ancestor elements because it reduces the stack operation in STJ algorithm; while it does not as good when filtering descendant elements because they need signature generating instead of stack operation. Therefore the performance of *S-Filter* is affected by the character of query. *PS-Filter* shows good performance and *CPS-Filter* performs a bit better than *PS-Filter*. And characters of queries have no influence on these two methods. For Q_3 and Q_4, the algorithms with filter can improve the performance in common cases, and in some extreme cases with only a few elements producing join results, the algorithm with filter shows great superiority.

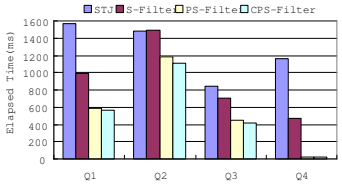


Figure 8. Elapsed Time for Q_1 to Q_4

5.2 Filtered-out Rate and False Pass Rate

The filtering performance of signature filter can be evaluated by filtered-out rate and false pass rate (see Definition 1 in Section2.3). Figure 9 shows the filtered-out rates and false pass rates of Q_3 with different signature lengths on 100M test document. We can see that the filtered-out rates of *S-Filter* and *PS-Filter* become higher when the signature lengths become longer while the false pass rates decrease, because the longer the length of signature, the more precise the range of element set presented by signature filter (the range represented by each bit is more narrow).

With different signature lengths, the filtered-out rate of *PS-Filter* is higher than that of *S-Filter* while the false pass rate is lower than that of *S-Filter*, so the performance of *PS-Filter* is better than that of *S-Filter*. Since we use 100M document, the code range is very large and the precision of the signature filter is not good. So the false pass rate is a little bit higher. However, with a smaller document, we can get a lower false pass rate. What is more, false pass rate can be deceased by increasing the signature length. But we did not use longer signature filter for the sake of saving space.

5.3 Performance vs Different Signature Length

Figure 10 illustrates different elapsed time of 4 queries with different signature filter lengths. For Q_1,Q_3 and Q_4, the filtering capability of signature filter improves with the increase of signature length. There are more filtered-out elements while the false passing elements become less. So

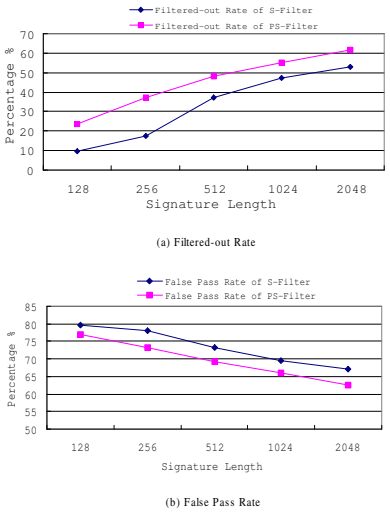


Figure 9. Filtered-out rate and false pass rate with different signature filter lengths

the elements that take part in join operation are reduced and the elapsed time are shortened. For Q_2, few elements can be filtered out. So the filtering capability cannot be improved by increasing the length of filter.

5.4 Performance vs Different Document Size

Figure 11 illustrates different elapsed time of 4 queries with different document sizes. The time complexity of STJ algorithm is $O(n)$, in which n is the number of elements in input list. With the document size growing, we can see that the elapsed time of STJ algorithm shows linear increase proportionately. Since *S-Filter*, *PS-Filter* and *CPS-Filter* are based on STJ algorithm, the elapsed time also increases linearly. While the three filtering algorithms reduce the number of input elements, thus cut the curve slope of STJ algorithm. And the filtered-out rates of *PS-Filter* and *CPS-Filter* are much higher because their curve slopes are the lowest.

The experiment results show that *S-Filter*,*PS-Filter* and *CPS-Filter* enhance the performance of structural join algorithm effectively. *S-Filter* algorithm needs to scan the input list once and cost some time on computing the signature of the filtered elements. *PS-Filter* algorithm reduces the accesses to unwanted elements and performs better. *CPS-Filter* has better performance than *PS-Filter*.

6 Conclusions

In this paper, we propose three signature-based algorithms for XML structural joins. Range-based signature filter accelerates the joins by reducing push and pop operations of unwanted elements. What is needed is only a small memory to store signature filter. Pointer-based signature filter adds an array of pointers, with which it can skip unwanted elements and locate the potential elements directly. With a little extra memory, we achieve much greater join efficiency. The optimized one further curtails memory cost and expedites it a little.

Acknowledgments This research was partially supported by the National Natural Science Foundation of China (Grant No. 60273079 and 60473074) and Specialized Research Fund for the Doctoral Program of Higher Education (SRFDP).

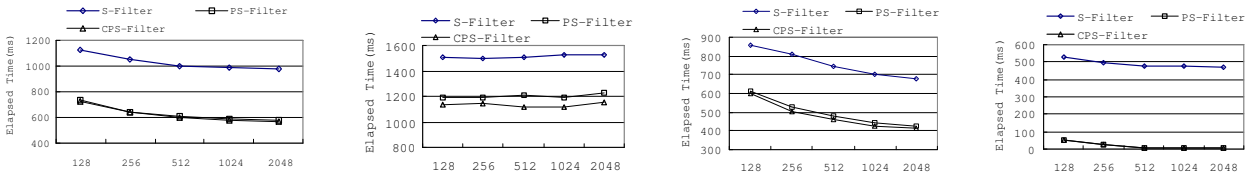
References

[1] T Bray, J Paoli, C M Sperberg-McQueen et al. Extensible markup language (XML) 1.0(Second Edition) W3C recommendation. World Wide Web Consortium, Tech Rep: RECxml-20001006, 2000. http://www.w3.org/TR/ 2000/REC-XML-20001006.

[2] D Chamberlin, D Florescu, J Robie et al. XQuery 1.0: An XML query language W3C working draft. World Wide Web Consortium, Tech Rep: WD-xquery-20010607, 2001.

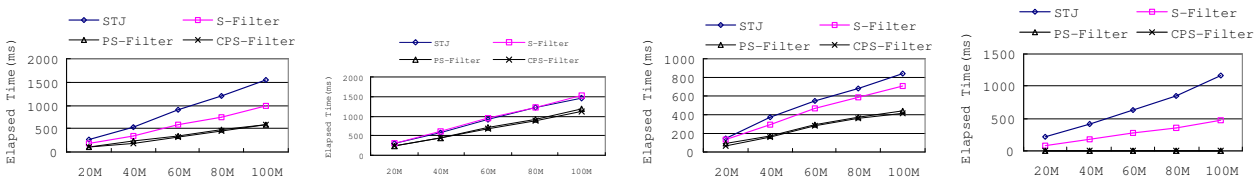
[3] J Clark, S DeRose. XML path language (XPath) version 1.0 W3C recommendation. World Wide Web Consortium, Tech Rep: REC-xpath-19991116, 1999.

[4] D Srivastava, S Al-Khalifa, H V Jagadish, N Koudas, J M Patel, W Yuqing. Structural joins: A primitive for efficient XML query pattern matching. In: Proc. of ICDE, 2002.



(a) Elapsed time of Q1 (b) Elapsed time of Q2 (c) Elapsed time of Q3 (d) Elapsed time of Q4

Figure 10. Elapsed Time of $Q_1 \sim Q_4$ with different signature filter lengths



(a)Elapsed time of Q1 (b)Elapsed time of Q2 (c)Elapsed time of Q3 (d)Elapsed time of Q4

Figure 11. Elapsed Time of $Q_1 \sim Q_4$ with different document sizes

[5] Q Li, B Moon. Indexing and querying XML data for regular path expressions. In: Proc. of VLDB, 2001.

[6] C Zhang, J F Naughton, D J DeWitt, Q Luo, G M Lohman. On supporting containment queries in relational database management systems. In: Proc. of SIGMOD, 2001.

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

[8] H Jiang, H Lu, W Wang, B C Ooi. XR-Tree: Indexing XML data for efficient structural joins. In: Proc. of VLDB, 2003.

[9] G. Yu, G. Wang, K. Kaneko, A. Makinouchi. Applying Signature Filtering Technique to Join Algorithms. DEXA Workshop. pp.928-932.

[10] B Sun, J Lv, G Wang, G Yu, B Zhou. Efficient Evaluation of XML Path Queries with Automata. In: Proc. of WAIM, 2003.

[11] D DeWitt, J Naughton. An evaluation of non-equijoin algorithms. In: Proc. of VLDB, 1991.

[12] G Graefe. Query evaluation techniques for large databases. ACM Computing Surveys, 1993, 25(2):73-170.

[13] Y Ishikawa, H Kitagawa, N Ohbo. Evaluation of signature files as set access facilities in OODBs. In: Proc. of SIGMOD, 1993.

[14] W Lee, D L Lee. Signature file methods for indexing object-oriented database systems. In: Proc. of ICIC, 1992.

[15] Sleepycat Software. Berkeley DB. http://www.sleepycat.com/download/db/index.shtml.

[16] DBLP Bibliography in XML. http://dblp.uni-trier.de/xml/dblp.xml.