Adaptive Index Utilization in Memory-Resident Structural Joins

Bingsheng He, Qiong Luo, and Byron Choi

Abstract—We consider adaptive index utilization as a fine-grained problem in autonomic databases in which an existing index is dynamically determined to be used or not in query processing. As a special case, we study this problem for structural joins, the core operator in XML query processing, in the main memory. We find that index utilization is beneficial for structural joins only under certain join selectivity and distribution of matching elements. Therefore, we propose adaptive algorithms to decide whether to use an index probe or a data scan for each step of matching during the processing of a structural join operator. Our adaptive algorithms are based on the history, the look-ahead information, or both. We have developed a cost model to facilitate this adaptation and have conducted experiments with both synthetic and real-world data sets. Our results show that adaptively utilizing indexes in a structural join improves the performance by taking advantage of both sequential scans and index probes.

Index Terms—Adaptive query processing, memory-resident systems, structural joins, index utilization.

1 Introduction

Adaptive query processing has long roots in the relational world [5], [6], [7], [19], [21], [31] and is important in improving the performance of autonomic databases [17], [33], [34]. The essence of adaptation is to gracefully adjust to the dynamic environment with some self-learning mechanism. As a result, adaptive algorithms may not always achieve the peak performance that can be obtained through careful tuning. However, they are most likely to maintain good performance in the presence of unexpected changes with little a priori information or manual tuning. In this paper, we consider adaptive index utilization as a fine-grained problem in autonomic databases which dynamically determines whether or not to use an existing index during query processing.

Structural joins [2], [10], [12], [13], [15], [23], [24], [27] or containment joins [37], have been a core operator in XML query processing. For example, a query "find all SECTION elements that contain FIGURE elements" involves a structural join between the SECTION elements and the FIGURE elements, which produces element pairs that satisfy the element nesting relationship in an XML document. Although previous work [2], [12], [15] deals with disk-resident structural joins, an increasing number of applications manipulate XML data in the main memory [18], [32]. In these applications, main-memory indexes can be built or loaded together with the data on the fly. However, it is questionable whether utilizing these indexes always has performance benefits.

- *B. He and Q. Luo are with the Computer Science and Engineering Department, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong. E-mail: {saven, luo}@cse.ust.hk.*
- *B. Choi is with the School of Computer Engineering, Nanyang Technological University, Block N4, Nanyang Avenue, Singapore 639798. E-mail: kkchoi@ntu.edu.sg.*

Manuscript received 4 May 2006; revised 28 Sept. 2006; accepted 22 Jan. 2007; published online 30 Jan, 2007.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0231-0506.
Digital Object Identifier no. 10.1109/TKDE.2007.1024.

The state-of-the-art structural join algorithms include nonholistic algorithms [2], [15], [23], [37] and holistic algorithms [10], [12], [13], [24], [27]. Holistic algorithms treat a query as a whole for processing, whereas nonholistic algorithms decompose a query into binary subqueries and evaluate the subqueries one by one [2], [15], [23], [37]. Since the basic idea of index utilization in binary join algorithms is similar to that in holistic join algorithms, we start with the algorithms of adaptive index utilization on binary joins and extend our algorithms to holistic joins.

A binary structural join compares two lists of XML elements and finds out the ancestor-descendant relationship in an XML document between the elements in the two lists. Moreover, the XML elements are represented in some encoding scheme. One basic encoding scheme is to use the start and the end positions of elements in a document. This is often referred to as the region encoding [2], [12], [13], [15], [23], [37]. With this encoding, the ancestor-descendant relationship checking becomes range predicates on the start and the end positions of the elements in the two lists. As both input lists are sorted on the (start, end) positions of the elements, tree indexes (for example, the B+-tree [15] and the XR-tree [23], [24]) can be built on both lists to facilitate range search and to skip some elements unnecessary for matching. In addition, an in-memory stack [2] is often used to store elements from the ancestor list to be compared with the current element in the descendant list so that no backtrack on either list is needed.

Examining the operations on the lists in a structural join, we define two execution modes for it: *scan* and *probe*. In the scan mode, the elements in one list are sequentially read one by one to compare with the elements in the other list. In the probe mode, indexes are used to locate the next-to-be-processed element in either list so that a number of elements may be skipped. By this definition, existing algorithms are all static—either always in the scan mode, for example, the *Stack-Tree* [2], or always in the probe mode, for example, the *Anc_Des_B+* [15] with B+-tree indexes [16].

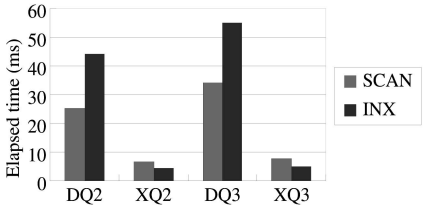


Fig. 1. Execution time of *Stack-Tree* (SCAN) and *Anc_Des_B+* (INX).

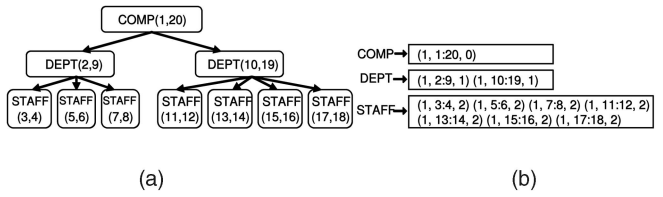


Fig. 2. (a) Region encoding. (b) Label indexes.

To study the performance of memory-resident structural joins, we implemented these two representative structural join algorithms. The *Stack-Tree* algorithm uses no index and our modified *Anc_Des_B+* algorithm uses the Cache-Sensitive B+-Tree (CSB+-Tree) [30] instead of the B+-tree for the main-memory performance.

To determine whether to use indexes for a structural join, common wisdom holds that indexes should be used for low join selectivities and scans otherwise. However, the boundary between low and high selectivities is a variable for different platforms. Moreover, given a fixed join selectivity, it is possible that using indexes improves the performance in some cases and degrades the performance in others, as shown in our study (Fig. 1). The detailed experimental setup of Fig. 1 is described in Section 4. Differently from the relational join selectivity, we use the ancestor join selectivity (the percentage of distinct matching elements in the ancestor list) and the descendant join selectivity (the percentage of distinct matching elements in the descendant list) as in recent studies on structural joins [23]. At this point, it is sufficient to know that both queries, DQ2 and XQ2 have a low descendant join selectivity of around 8 percent and that both DQ3 and XQ3 have a relatively high descendant join selectivity of around 22 percent. All four of these queries have an ancestor join selectivity of 100 percent.

Since neither of the static algorithms are a guaranteed winner, we propose improving the performance of memory-resident structural joins by adaptively determining whether to use indexes for each step of matching during join processing. On one hand, this adaptation should respond to the environment well; on the other hand, the overhead of the adaptation should be low.

First, we develop a cost model to compare index probes with data scans. Intuitively, an index probe may skip elements but have the overhead of going down the index tree to reach the data leaves. The cost-based decision of index utilization boils down to the number of elements skipped in an index probe. Additionally, our cost model can serve as the basis for comparing the relative performance of different structural join algorithms.

Next, we design our adaptive schemes for index utilization. They are based on the history, the look-ahead information, or a mix of the two. Since our model can tell if the previous or the upcoming index probes are cost efficient, the decision on the adaptation for the next move is made in a cost-based manner. The history-based adaptation is analogous to the processor branch prediction [22] in that it uses one or two bits to record the correctness of previous decisions on whether to choose an index probe or a data scan. The look-ahead-based adaptation checks if the next index probe can skip a sufficient number of elements to compensate for its cost. Finally, the hybrid adaptation scheme makes decisions considering both the history and the look-ahead information.

We have tested these adaptive algorithms in comparison with static algorithms on synthetic data sets, as well as on real-world data sets. Taking advantage of both the scan and the probe, our adaptive algorithms consistently outperform the static algorithms and the hybrid scheme performs the best among the three adaptive schemes.

The remainder of this paper is organized as follows: Section 2 introduces the preliminaries of this work and discusses the related work. We present our proposed adaptive algorithms for binary and holistic structural joins in Section 3. In Section 4, we experimentally evaluate the performance of our approaches. Finally, we conclude in Section 5.

2 Preliminaries and Related Work

In this section, we introduce the preliminaries of this work and discuss the related work. We first review the region encoding scheme and structural join algorithms. We then illustrate the two execution modes of a generic structural join algorithm. Finally, we review existing techniques on autonomic databases and adaptive query processing.

2.1 Region Encoding and Structural Joins

The structural relationship of two elements in an XML document can be efficiently determined using a region encoding scheme [2], [12], [13], [15], [23], [24], [37]. This scheme encodes each element with a 4-ary tuple: $< docID, start : end, level >$, where $docID$ is the document identifier, $start$ and end are the start and the end positions (or the preorder and the postorder ranks) of the element, and $level$ is the depth of the element in the document tree. The region encoding of a sample XML document is shown in Fig. 2, with $docID$ and $level$ omitted. Using this encoding scheme, the structural relationship of elements a and d is determined as follows: 1) a is an ancestor of d if and only if $a.docID = d.docID$, $a.start < d.start$, and $d.end < a.end$ and 2) a is the parent of d if and only if a is an ancestor of d and $a.level = d.level - 1$.

With this region encoding, XML documents are represented as *label indexes*, which are lists of encoded elements sorted by $(docID, start)$. Fig. 2b shows the label indexes of the sample document. We refer to these label indexes as *lists of elements* (or element lists) to distinguish them from the tree indexes built on top. For simplicity of presentation, we omit the $docID$ and $level$ and use a tuple $< start : end >$ to represent an element in the remainder of the paper.

TABLE 1
Representatives of State-of-the-Art Structural Join Algorithms

	No index	Index
Non-holistic	*MPMGJN* [37], *Stack-Tree* [2], staircase [18]	*Anc_Des_B+* [15], *XR-stack* [23]
Holistic	*PathStack,TwigStack* [10], *TGGeneric* [24], *iTwigJoin* [13], *TJFast* [27], *Twig^{2}Stack* [12]	*TwigStackXB* [10], *TGGenericXR* [24]

With two lists of elements as input, a *binary structural join* outputs all pairs of elements that satisfy the structural relationship. In this paper, we focus on the ancestor-descendant relationship. We call an element in the resulting pairs a *matching element*. If a matching element is from the ancestor list, it is an *AYElement*; otherwise, it is a *DYElement*. Similarly, an unmatching element, which has no matching element in the other list, is either an *ANElement* from the ancestor list or a *DNElement* from the descendant list. We call a block of consecutive matching elements in either list a *matching block* and a block of consecutive unmatching elements an *unmatching block*.

Table 1 shows a few representatives of the existing structural join algorithms. We categorize them along two dimensions: nonholistic versus holistic processing and using indexes on the lists or not. All algorithms except $Twig^2Stack$ [12] evaluate the query through accessing the documents from the top down, whereas $Twig^2Stack$ utilizes a hybrid approach combining both top-down and bottom-up evaluations. *PathStack* [10] is based on a novel stack encoding and is optimal for evaluating linear path queries and $Twig^2Stack$ outperforms existing holistic algorithms for twig queries (i.e., [10], [27]). All of these algorithms except the staircase join [18] are designed for processing disk-resident data. In comparison, we focus on memory-resident data. With the same focus on memory-resident data, the staircase join utilizes the consecutiveness of an XML encoding scheme and can skip the unmatching elements without indexes. In contrast, we focus on the adaptive utilization of index probes for documents with a common, possibly nonconsecutive region encoding scheme.

A number of cache-optimized tree index structures have been proposed for main-memory databases, such as CSB+-trees [30], CR-trees (Cache-conscious R-Trees) [25], and T-trees [26]. Rao and Ross showed that the search performance of CSB+-trees is better than that of T-trees [30]. We have implemented CR-trees and tested them in the structural join. Similarly to the previous study on structural joins for disk-based data [15], our experimental results showed that structural joins with CR-trees in the main memory were slower than those with CSB+-trees. Therefore, we choose the CSB+-tree as our index structure in the main memory and develop a model to estimate the access cost of CSB+-trees. Note that some compression schemes, such as the partial key technique [9], can be applied to CSB+-trees to improve their search performance. For simplicity, we used CSB+-trees without compression in our model and experiments. Nevertheless, our model can be extended to consider CSB+-trees with compression.

2.2 The Two Execution Modes of Structural Joins

Our adaptive techniques work around the *scan* and *probe* modes of a structural join in the presence of tree indexes. Let us illustrate these two modes with a generic static binary structural join algorithm in the presence of B+-trees (Algorithm 1), which is slightly modified from the original *Anc_Des_B+* algorithm [15] by adding the mode definition. The variables used in the join algorithms throughout this paper are shown in Table 2. We will describe the variables in the last four rows in more detail when we present our adaptive algorithms in Section 3.

Algorithm 1. *Anc_Des_B+*$(AList, DList)$ [Modified]
Procedure: *Anc_Des_B+*()
1: $a = AList.first$, $d = DList.first$
2: **while** $a \neq AList.end$ and $d \neq DList.end$ **do**
3: **if** a is an ancestor of d **then**
4: $stack.push(a_i), \forall a_i \in AList$ and a_i is an ancestor of d;
5: a is set to be the last element pushed;
6: *Output* (a_i, d), $\forall a_i \in stack$;
7: $d = DList.next()$;
8: **else**
9: **if** $a.end < d.start$ **then**
10: **while** $stack.topEnd() < d.start$ **do**
11: *stack.pop()*;
12: *NextAnc*(); /*Locate the next ancestor*/
13: **else**
14: *Output* (a_i, d), $\forall a_i \in stack$;

TABLE 2
Terminology Used throughout This Paper

Variables	*Description*
$AList$, $DList$	ancestor and descendant lists in the join
I_A, I_D	tree indexes built on $AList$ and $DList$
a, d	cursors for $AList$ and $DList$
$AMode$, $DMode$	execution modes on $AList$ and $DList$ (**N** means to *scan* and **I** means to *probe* the index.)
$ACount$, $DCount$	numbers of *ANElement* or *DNElement* scanned since d or a was last advanced
$ASkip$, $DSkip$	threshold values to determine $AMode$ and $DMode$
$ALook$, $DLook$	on-off flags of the look-ahead operation on $AList$ and $DList$ (**N** and **L** mean to determine the execution mode without and after a look ahead, respectively.)

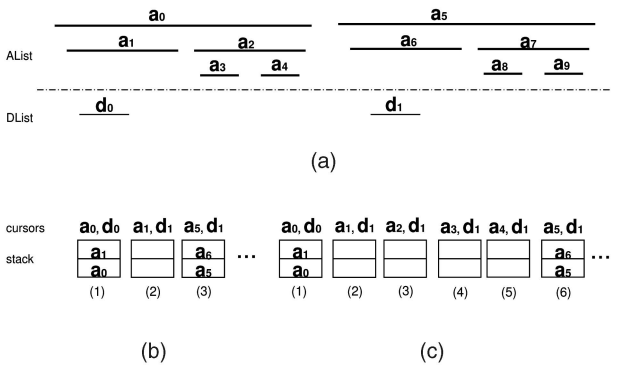


Fig. 3. An example of the probe and scan modes. (a) AList and DList. (b) Probe mode. (c) Scan mode.

```
15:    NextDes(); /*Locate the next descendant*/
#IFDEF PROBE_MODE
Procedure: NextAnc()
 1: l is set to be the last element popped;
 2: a = I_A.SearchLarger(l.end);
Procedure: NextDes()
 1: if stack.empty() then
 2:    d = I_D.SearchLarger(a.start);
 3: else
 4:    d = DList.next();
#ELSE
Procedure: NextAnc()
 1: a = AList.next();
Procedure: NextDes()
 1: d = DList.next();
#ENDIF
```

There are three procedures in Algorithm 1, namely, *Anc_Des_B+*, *NextAnc*, and *NextDes*. The main procedure *Anc_Des_B+* performs the matching of elements between the two lists. It maintains one cursor on each list and calls *NextAnc* and *NextDes* to advance the cursor on the corresponding list. The algorithm maintains an in-memory stack, which keeps all ancestors of the element d for processing. The operations over the stack are *empty*, *pop*, *push*, and *topEnd*. The first three operations are standard stack operations and the last one returns the *end* value of the element on the top of the stack.

The flag **PROBE_MODE** indicates the execution mode of the join, which determines if a scan (sequentially advancing the cursor to the next element) or an index probe (using the index to advance the cursor to the next element, which might have skipped several unmatching elements in the list) is used in *NextAnc* and *NextDes*. If the flag is set to *probe*, it is the original *Anc_Des_B+* algorithm [15]; if the flag is set to *scan*, it is similar to the *Stack-Tree* algorithm [2] with minor differences in the elements pushed into the stack. In our experiments, we found that there was no significant performance difference between *Stack-Tree* and the scan mode *Anc_Des_B+*.

Let us illustrate the differences between the two modes with an example. Given $AList = [a_0, a_1, \ldots, a_9]$ and $DList = [d_0, d_1]$, their positions in a document are shown in Fig. 3a. The cursor movement and the stack content during the join processing in two execution modes are shown in Figs. 3b and 3c, respectively. In the probe mode, the elements a_2, a_3, and a_4 are skipped, whereas, in the scan mode, no element is skipped. Even though the probe mode may skip some unmatching elements, the overhead of the index access may offset the performance gain from skipping.

The idea of skipping unmatching elements using the B+-tree index has been exploited in Zigzag skips [3], [4], [14]. Cheng et al. proposed applying an ascending traversal from the current tree node through its parent pointer and, then, a descending traversal to locate the next matching element [14]. It needs extra space to store an auxiliary parent pointer in each tree node. Antoshenkov proposed scanning the index leaf node where the last matching element was found, prior to executing an index probe [3]. However, in our study, the node size of the CSB+-Tree is small and the benefit of scanning the remainder of the current leaf node is limited. Closely related to our work, an adaptive algorithm was proposed to switch between the *skip* mode and the *no skip* mode [4]. It switches to the no skip mode when the number of skip attempts without actual skipping exceeds some small threshold and switches to the skip mode at the end of scanning the current leaf node in the index. In comparison, we develop a cost model to quantify the switching conditions and the frequency of our adaptation is per tuple.

2.3 Autonomic Databases and Adaptive Query Processing

The database community has already made many significant contributions to autonomic query processing [17], [33], [34]. The index advisor [33] recommends the best index among multiple candidate indexes. In comparison, we assume that the tree index has been built on top of each element list and we consider adaptive utilization for these indexes. The LEO (Learning Optimizer) [34] learns from prior executions and uses actual cardinalities for later executions of queries with similar predicates. Similarly, our history-based algorithm predicts the next execution mode based on the correctness of the previous one or two decisions.

Adaptive query processing is self-optimizing and provides good performance for autonomic databases with little manual tuning [17]. The state-of-the-art adaptive query processing techniques [5], [6], [19], [31] are compared and categorized in two surveys [7], [21]. Under their categorization, our work belongs to intraoperator and per-tuple adaptation. Early work on intraoperator adaptation used either competition [5] or sampling [31] techniques. In contrast, our look-ahead-based algorithm compares the cost of the upcoming index probe with a threshold value at each step of matching without competition and our history-based algorithm is based on a short most recent observation, not sampling. Compared with the per-tuple adaptive eddies [6], our work deals with the CPU cost of tuple matching as opposed to routing tuples through a pool of pipelined operators.

Our focus in this work has been to study the in-memory cost of data scans versus index probes for tuple matching and to propose lightweight high-frequency adaptation

schemes for a structural join in the main memory. To the best of our knowledge, this work is the first on applying adaptive schemes to the index utilization in the memory-resident structural join.

3 Adaptive Structural Joins

In this section, we first discuss the challenges in adaptively choosing the correct execution mode for each step of matching. Next, we present the adaptive algorithms for binary structural joins, including the history-based, the look-ahead-based, and the hybrid ones. We then describe our cost model for the adaptation. We also apply our adaptive schemes to two holistic algorithms, *PathStack* [10] and $Twig^2Stack$ [12]. Finally, we discuss a few issues about our adaptive algorithms.

3.1 Challenges

As shown in Fig. 1, it is hard to predict the best static algorithm even when the join selectivity is known. Therefore, we consider choosing a correct execution mode for each step of matching during the join processing. The first challenge is to define the correctness of an execution mode. As we deal with memory-resident structural joins in the presence of indexes, we need a model to estimate the access cost of the main-memory hierarchy for both data scans and index probes.

One observation on the index probes in a structural join is that their search-key values are ascending. Due to data reuse, this sortedness of index probes has a significant performance impact on in-memory indexed structural joins, as we observed in our experiments. To the best of our knowledge, there is no existing model to estimate the access cost for these kinds of index probes in the main memory. Specifically, the existing cost models for in-memory databases [20], [28] considered the cost of random probes on a tree index structure.

Having the cost model in hand, the next challenge is to apply the correct execution modes. A static approach containing precomputed execution modes for each step of matching is undesirable due to the space overhead or may even be infeasible in many situations. Furthermore, evaluating joins on different element lists may need different plans. Thus, we consider applying the execution modes adaptively at runtime. As this adaptation happens at a high frequency and the structural join is memory resident, our adaptive schemes must be lightweight.

3.2 Algorithms for Binary Joins

We develop our adaptive algorithms by modifying the generic *Anc_Des_B+* algorithm (shown in Section 2.2). The modifications include removing the static execution mode and changing *NextAnc* and *NextDes* to adaptively choose the execution mode according to our adaptive schemes.

3.2.1 The History-Based Algorithm

A natural way of adaptation is through observations on the history. Our history-based algorithm belongs to this category. Inspired by the correspondence between branch prediction [22] and the selection of execution modes, we develop a history-based adaptive algorithm using the n-bit scheme. The n-bit scheme uses n bits to record the correctness of the previous n decisions and makes the next decision based on the n-bit history. Considering the trade-off between the overhead of maintaining the history and the potential gain from adaptivity, we tested the 1-bit and the 2-bit schemes that have been known to be practical [22]. Since the performance of the 2-bit scheme was similar to that of the 1-bit scheme in our experiments, we focus on the 1-bit scheme.

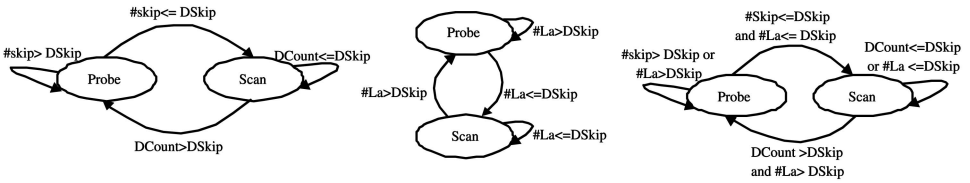
Procedures *NextAnc* and *NextDes* in the 1-bit adaptive structural join algorithm *1bit-Adaptive* are shown in Algorithm 2. Variables $ACount$ and $DCount$ are both initialized with zero and $AMode$ and $DMode$ both with **I**, that is, the algorithm starts with the probe mode. We present the estimation of $ASkip$ and $DSkip$ in our cost model in Section 3.3. At this point, it is sufficient to know that $ASkip$ and $DSkip$ are constants.

Algorithm 2. Procedures *NextAnc()* and *NextDes()* of *1bit-Adaptive*

Procedure: *NextAnc*()
1: **if** $AMode = N$ **then**
2: $a = AList.next()$;
3: $ACount = ACount + 1$;
4: **if** $ACount > ASkip$ **then**
5: $AMode = \mathbf{I}$;
6: l is set to be the last element popped;
7: $a = I_A.SearchLarger(l.end)$;
8: **else**
9: l is set to be the last element popped;
10: $olda = a$;
11: $a = I_A.SearchLarger(l.end)$;
12: If $(a - olda - 1) \leq ASkip$, then $AMode = \mathbf{N}$; /*Skip too little*/
13: $DCount = 0$;

Procedure: *NextDes*()
1: **if** $stack.empty()$ **then**
2: **if** $DMode = \mathbf{I}$ **then**
3: $oldd = d$;
4: $d = I_D.SearchLarger(a.start)$;
5: If $(d - oldd - 1) \leq DSkip$, then $DMode = \mathbf{N}$; /*Skip too little*/
6: **else**
7: $d = DList.next()$;
8: $DCount = DCount + 1$;
9: **if** $DCount > DSkip$ **then**
10: $DMode = \mathbf{I}$;
11: $d = I_D.SearchLarger(a.start)$;
12: **else**
13: $d = DList.next()$;
14: $ACount = 0$;

Let us describe the adaptive algorithm in more detail using the *NextDes* procedure. Suppose the previous mode was *probe*. If the number of unmatching elements skipped was smaller than the threshold value $(DSkip)$, the previous mode was incorrect and the algorithm changes to the scan mode (Line 5). Similarly, if the previous mode was *scan* and the number of unmatching elements scanned since the cursor of the ancestor list was last advanced was larger than



(a) (b) (c)

Fig. 4. Execution mode switching in our adaptive schemes. $\#skip$ represents the number of unmatching elements that was skipped in the previous probe and $\#La$ represents the number of unmatching elements that may be skipped in the current index probe (the comparison of $\#La$ and $DSkip$ is examined through a look-ahead operation). (a) One-bit history-based. (b) Look-ahead-based. (c) Hybrid.

$DSkip$, the previous mode was incorrect and the algorithm changes to the probe mode (Lines 9-11). This mode switching is illustrated in Fig. 4a.

Since the history-based algorithm maintains only one or two previous decisions, it has a low misprediction rate when there are consecutive long unmatching blocks (the length of each block is larger than the threshold value) or consecutive short ones (the length of each block is smaller than the threshold value). The 1-bit scheme used in the history-based algorithm is lightweight as the bookkeeping overhead is low. As shown in our experiments, the 1-bit scheme is a good trade-off between the efficiency and the overhead of the adaptation.

3.2.2 The Look-Ahead-Based Algorithm

The history-based algorithm predicts the execution mode based on the history and may make incorrect decisions. As we assume the two lists are static during the join processing, it is possible to make decisions based on the look-ahead information.

The state diagram of the look-ahead-based scheme is shown in Fig. 4b. It decides the execution mode using the look-ahead operation. We discuss the look-ahead-based algorithm with the *NextDes* procedure (Algorithm 3). The main idea is to look ahead at the element offset by $DSkip$ in the descendant list and to examine whether the length of the current unmatching block is larger than $DSkip$. If so, an index probe is chosen to skip this unmatching block. Otherwise, the execution mode should be *scan*, and the block is scanned.

Algorithm 3. Procedure *NextDes()* of *Look-ahead-Adaptive*

```
1: if stack.empty() then
2:   if (d + DSkip).start < a.start then
3:     d = I_D.SearchLarger(a.start); /*DMode = N*/
4:   else
5:     while d.start < a.start do
6:       d = DList.next(); /*DMode = N*/
7: else
8:   d = DList.next();
```

Differently from the history-based algorithm, the look-ahead-based algorithm always makes a correct decision. However, each look-ahead operation induces the possible overhead of scanning one unmatching element. Since an element in the region encoding uses 16 bytes and is smaller than an L2 cache line in our experiments, this overhead can be an L2 cache miss in the worst case. This overhead is negligible when the total number of elements skipped is large and can otherwise be considerable.

3.2.3 The Hybrid Algorithm

With the history-based and the look-ahead-based algorithms, we consider if it is possible to decide the execution mode based on both the history and the look-ahead information. With the history information, some unnecessary look-ahead operations can be eliminated. With the look-ahead information, some incorrect decisions that would have been made based solely on the history may be avoided.

We propose a hybrid algorithm utilizing the history and the look-ahead information, namely, *Hybrid-Adaptive*. We integrate the history-based and the look-ahead-based adaptation components into the *NextDes* procedure, as shown in Algorithm 4. Flag $DLook$ indicates whether the algorithm applies the look-ahead-based component (Lines 3-11) or the history-based component (Lines 14-23). The algorithm starts with a look-ahead operation, that is, $DLook$ is **L**, initially. When it finds the execution mode that needs to be changed (Lines 18 and 23), it performs a look-ahead operation and decides the next execution mode based on the look-ahead information. Otherwise, the execution mode is unchanged. The state diagram of the algorithm is sketched in Fig. 4c.

Algorithm 4 Procedure *NextDes()* of *Hybrid-Adaptive*

```
 1: if stack.empty() then
 2:   /*the look-ahead-based component*/
 3:   if DLook = L then
 4:     DLook = N;
 5:     if (d + DSkip).start < a.start then
 6:       d = I_D.SearchLarger(a.start);
 7:       DMode = I;
 8:     else
 9:       while d.start < a.start do
10:         d = DList.next();
11:       DMode = N;
12:   else
13:     /*the history-based component*/
14:     if DMode = I then
15:       oldd = d;
16:       d = I_D.SearchLarger(a.start);
17:       if (d − oldd − 1) ≤ DSkip then
18:         DLook = L, go to Line 3;
19:     else
20:       d = DList.next();
21:       DCount = DCount + 1;
22:       if DCount > DSkip then
```

TABLE 3
Input Parameters for the Cost Model

Parameter	*Description*
C_L	the average cost of accessing one cache line
C_{AN}	the average cost of scanning one *ANElement*
C_{DN}	the average cost of scanning one *DNElement*
C_{AI}	the average cost of one probe on I_A
C_{DI}	the average cost of one probe on I_D
C_R	the average cost of accessing an index node that resides in the cache
C_{NR}	the average cost of accessing an index node that does not reside in the cache
N_{AN}	the number of *ANElement* scanned
N_{DN}	the number of *DNElement* scanned
N_{AI}	the number of probes on I_A
N_{DI}	the number of probes on I_D

23: $DLook = L$, go to Line 3;
24: **else**
25: $d = DList.next()$;
26: $ACount = 0$;

3.3 The Cost Model

All of our adaptive structural join algorithms need a cost model to determine the threshold values, $ASkip$ and $DSkip$. Furthermore, it is desirable to quantitatively compare the relative performance of the algorithms without running them on target machines. Therefore, we developed a cost model for performance estimation, part of which estimates $ASkip$ and $DSkip$.

Let us go through the cost model from the top down. The total cost of a structural join includes three components: the cost of scanning the matching elements, the cost of scanning the unmatching elements, and the cost of index probes. Our cost metric is the execution time. For a given workload (a structural join on two element lists), the cost of scanning the matching elements is fixed among the stack-based algorithms (either the static ones or the adaptive ones), because their numbers of stack operations and matching elements scanned are fixed. To compare the relative performance of these algorithms, we can exclude the fixed cost of scanning the matching elements and only compare the costs of scanning the unmatching elements and index probes.

We define *variable cost* VC to be the total cost of scanning the unmatching elements and index probes for a given workload. Define SC_{AN} and SC_{DN} to be the total cost of scanning the *ANElement*s and *DNElement*s, respectively. Also, define SC_{AI} and SC_{DI} to be the total cost of index probes on indexes I_A and I_D, respectively. Then, the top-level equation for the variable cost is the following:

$$VC = SC_{AN} + SC_{DN} + SC_{AI} + SC_{DI}. \quad (1)$$

The larger VC an algorithm has, the less efficient it is.

To estimate SC_{AN}, SC_{DN}, SC_{AI}, and SC_{DI}, we need the parameters listed in Table 3. Each cost component in (1) is computed as the unit cost of an operation multiplying the number of operations. That is, SC_{AN}, SC_{DN}, SC_{AI}, and SC_{DI} are estimated to be $(C_{AN} \times N_{AN})$, $(C_{DN} \times N_{DN})$, $(C_{AI} \times N_{AI})$, and $(C_{DI} \times N_{DI})$, respectively. In addition, we can estimate $ASkip$ and $DSkip$ in (2) and (3), respectively.

$$ASkip = \left\lceil \frac{C_{AI}}{C_{AN}} \right\rceil, \quad (2)$$

$$DSkip = \left\lceil \frac{C_{DI}}{C_{DN}} \right\rceil. \quad (3)$$

We obtain the values of C_{AN} and C_{DN} through calibration on target machines. The values of C_{AI} and C_{DI} are estimated using the values of C_R and C_{NR}, which are also obtained through calibration on target machines. The detailed experimental setup of the calibration is described in Section 4. In our experiments, the values obtained through calibration were stable and sufficiently accurate for our cost model. This accuracy contributes to the improved performance of our adaptive algorithms.

We now focus on the estimation of C_{AI} and C_{DI} using C_R and C_{NR}. One critical factor for estimating C_{AI} and C_{DI} is cache reuse among index probes. Note that the index search keys in a structural join are in ascending order. We describe cache reuse among the probes with ascending search keys using a proposition and a corollary. We make an observation about the proposition, the corollary, and our estimations: If an index node does not reside in the cache, its descendant nodes are not likely to reside in the cache either. In addition, let P_k and $P_{k'}$ be the paths accessed by two probes with search-key values k and k', respectively; we denote $LP(P_k, P_{k'})$ to be the longest common prefix of P_k and $P_{k'}$.

Proposition 1. *Let P_k, $P_{k'}$, and $P_{k''}$ be the paths accessed by three index probes with search-key values k, k', and k'', respectively. If $k \leq k' \leq k''$, then $LP(P_{k''}, P_k)$ is a prefix of $LP(P_{k''}, P_{k'})$.*

Proof. Suppose $n_1 n_2 \ldots n_h$ are the index nodes contained in P_k from the top down, where k is the search-key value and h is the height of the index tree. We define a *key sequence*, $a_1 a_2 \ldots a_{h-1}$, for each probe so that a_i $(1 \leq i < h)$ is the maximum of the key values that are less than k in the nonleaf node n_i.

Given any two probes with search-key values k_x and k_y, where $k_y \geq k_x$ and their key sequences are $a_1 a_2 \ldots a_{h-1}$ and $b_1 b_2 \ldots b_{h-1}$, respectively, if $a_i = b_i$ $(1 \leq i < m, m \leq (h-1))$ and $a_i \neq b_i$ $(i \geq m)$, we have $LP(P_{k_x}, P_{k_y}) = n_1 n_2 \ldots n_m$ and vice versa. As a special case, when $m = 1$, $LP(P_{k_x}, P_{k_y}) = n_1$, that is, P_{k_x} and P_{k_y} only have the root node in common.

Let P_k, $P_{k'}$, and $P_{k''}$ be $n_1 n_2 \dots n_h$, $n'_1 n'_2 \dots n'_h$, and $n''_1 n''_2 \dots n''_h$, respectively. Suppose

$$a_1 a_2 \dots a_{h-1},\ a'_1 a'_2 \dots a'_{h-1},\ \text{and}\ a''_1 a''_2 \dots a''_{h-1}$$

are the key sequences of the probes with search-key values k, k', and k'', respectively. Since $k \le k' \le k''$, we have $a_i \le a'_i \le a''_i$ $(1 \le i < h)$.

We give the relationship of $LP(P_{k''}, P_k)$ and $LP(P_{k''}, P_{k'})$ according to the differences among the key sequences:

1. $a_i = a'_i$ $(1 \le i < h)$ indicates $P_k = P_{k'}$. We have $LP(P_{k''}, P_k) = LP(P_{k''}, P_{k'})$.
2. $a_i = a'_i$ $(1 \le i < m, m \le (h-1))$ and
$$a_i \ne a'_i\ (i \ge m).$$
We have two cases based on the differences of the key sequences of the probes with search-key values k' and k'':
   a. $a'_i = a''_i$ $(1 \le i < h)$ indicates $P_{k'} = P_{k''}$. We have $LP(P_{k''}, P_{k'}) = P_{k''}$. Hence, $LP(P_{k''}, P_k)$ is a prefix of $LP(P_{k''}, P_{k'})$.
   b. $a'_i = a''_i$ $(1 \le i < m', m' \le (h-1))$ and
$$a'_i \ne a''_i\ (i \ge m').$$
We have $LP(P_{k'}, P_{k''}) = n''_1 \dots n''_{m'}$. Similarly, we have $a_i = a''_i$ $(1 \le i < m'', m'' \le (h-1))$ and $a_i \ne a''_i$ $(i \ge m'')$ and obtain
$$LP(P_k, P_{k''}) = n''_1 \dots n''_{m''}.$$
Since
$$a_i \le a'_i \le a''_i\ (1 \le i < h),$$
we have $m'' \le m'$ and $LP(P_{k''}, P_k)$ is a prefix of $LP(P_{k''}, P_{k'})$.

In all cases, we have that $LP(P_{k''}, P_k)$ is a prefix of $LP(P_{k''}, P_{k'})$. □

Corollary 1. *Let P_{k_i} $(1 \le i \le n, n \ge 3)$ be the path accessed by a probe with search-key value k_i. If $k_1 \le k_2 \dots \le k_n$, then $LP(P_{k_n}, P_{k_j})$ is a prefix of $LP(P_{k_n}, P_{k_{n-1}})$, $1 \le j \le n-2$.*

Proof. Since $k_j \le k_{n-1} \le k_n$ $(1 \le j \le n-2)$, $LP(P_{k_n}, P_{k_j})$ is a prefix of $LP(P_{k_n}, P_{k_{n-1}})$ according to Proposition 1. □

According to Corollary 1, we only need to consider the cache reuse between two consecutive probes in a structural join. Given two consecutive index probes p_i and p_{i+1} in a structural join, p_{i+1} may access an index node in the path of p_i (the node is likely to reside in the cache) or may access an index node that is not in the path of p_i (the node does not reside in the cache). For simplicity, we assume that the probability of accessing an index node in the path of p_i is equal to that of accessing an index node that is not in the path of p_i, that is, the probability is $\frac{1}{2}$.

The estimation of the unit cost of one probe on a CSB+-Tree index with a height of h is shown in (4). $C_I(h)$ is either C_{AI} or C_{DI} in Table 3. Since the root node is likely to reside in the cache, we estimate $C_I(h)$ to be the total cost of accessing the root node C_R and accessing the lower levels $C_L(h-1)$. The partial cost $C_L(h)$ is estimated using (5). If a node at height h resides in the cache, the cost of accessing this node is C_R and we recursively estimate the cost of accessing the lower levels of the index. Otherwise, this node and the nodes at its lower levels accessed by the probe do not reside in the cache and the cost of accessing these nodes is hC_{NR}.

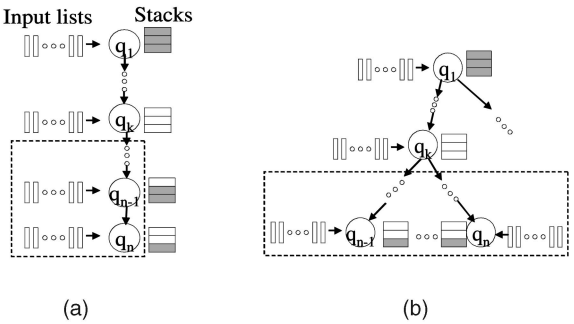


Fig. 5. Skipping in *PathStack* and $Twig^2Stack$. The top-down stack of q_k is empty. Unmatching elements in the element lists $q_{k+1}, q_{k+2}, \dots$, and q_n can be skipped. (a) *PathStack*. (b) $Twig^2Stack$.

$$C_I(h) = C_R + C_L(h-1), \tag{4}$$

$$C_L(h) = \begin{cases} \frac{1}{2}(C_L(h-1) + C_R) + \frac{1}{2}hC_{NR}, & h \ge 1 \\ 0, & h = 0. \end{cases} \tag{5}$$

In summary, our cost model estimates the cost of index probes with sorted keys considering the cache reuse among these probes. This consideration improves the accuracy of our estimation. Specifically, without this consideration, the threshold values obtained by the random probe model [20] are larger than those by our model. These larger threshold values may result in missing the opportunities of skipping some unmatching blocks and, in turn, reduce the performance improvement of our adaptive algorithms, as we observed in our experiments.

3.4 Algorithms for Holistic Joins

After presenting the adaptive schemes on binary joins, we apply our adaptive schemes to two holistic joins, *PathStack* [10] and $Twig^2Stack$ [12]. Since $Twig^2Stack$ subsumes $TwigStack$ [10], we focus our discussion on $Twig^2Stack$. *PathStack* is optimal for path queries and $Twig^2Stack$ is the state-of-the-art holistic algorithm for twig queries. Since both algorithms originally scan all involved element lists and do not utilize tree indexes, we first need to study how to use an index in these algorithms and then design adaptive index utilization schemes for them. We only present the adaptive algorithms with the look-ahead-based scheme. Similarly, we can apply the other two adaptive schemes to these join algorithms.

3.4.1 *Adaptive* Pathstack

PathStack is an efficient holistic algorithm for the evaluation of a path query. In the presence of tree indexes on element lists, we can skip the unmatching elements. The basic idea of the skipping process is illustrated in Fig. 5a. We denote Q as the set of element lists involved in the query. Given a path query $q_1//q_2 \dots //q_n$, Q consists of n element lists q_i $(1 \le i \le n)$. If the stack of the element list q_k $(k \ne n)$ is empty, unmatching elements in the descendant element lists of q_k (that is, $q_{k+1}, q_{k+2}, \dots, q_n$) can be skipped.

Algorithm 5 describes the modified *PathStack* algorithm with the look-ahead-based scheme. It maintains an in-memory stack S_{q_i} for each element list q_i which stores potential results for the join. Differently from the stack entry of the binary structural join algorithm, an entry in S_{q_i} consists of a tuple $< e, p >$, where e is an element from q_i and p is the pointer to an entry in $S_{q_{i-1}}$ ($p = nil$ if $i = 0$).

Algorithm 5. *LA_PathStack*: the modified *PathStack* [10] with the look-ahead-based scheme

1: **while** $\neg end(Q)$ **do**
2: $q_{min} = getMinSource(Q)$; /*the element list with the minimum $start$ value*/
3: $oldc = c_{q_{min}}$;
4: $key = -1$;
5: **for** $i = 1; i \leq n; i++$ **do**
6: **while** $\neg S_{q_i}.empty()$ and $S_{q_i}.topEnd() < c_{q_{min}}.start$ **do**
7: $S_{q_i}.pop()$;
8: **if** $S_{q_i}.empty()$ and $i < min$ **then**
9: $key = c_{q_i}.start$; /*$key \neq -1$ indicates some unmatching elements can be skipped.*/
10: **if** $key \neq -1$ and $i \neq n$ **then**
11: **if** $(c_{q_{i+1}} + DSkip_{q_{i+1}}).start < key$ **then**
12: $c_{q_{i+1}} = I_{q_{i+1}}.SearchLarger(key)$; /*Probe mode on q_{i+1}*/
13: **else**
14: **while** $c_{q_{i+1}}.start < key$ **do**
15: $c_{q_{i+1}} = q_{i+1}.next()$; /*Scan mode on q_{i+1}*/
16: **if** $key = -1$ **then**
17: $S_{q_{min}}.push(c_{q_{min}},$ pointer to $S_{q_{min-1}}.top())$;
18: **if** $min = n$ **then**
19: $showSolutions(S_{q_{min}}, 1)$;
20: $S_{q_{min}}.pop()$;
21: **if** $oldc = c_{q_{min}}$ **then**
22: $c_{q_{min}} = q_{min}.next()$;

Procedure: $end(Q)$

1: $c_q \neq q.end, \forall q \in Q$;

Procedure: $getMinSource(Q)$

1: return q_{min} so that $c_{q_{min}}.start$ is minimal among c_q, $\forall q \in Q$;

For each iteration (Lines 2-22), the algorithm starts with the element list whose cursor has the minimum $start$ value among the n element lists. Suppose this element list is q_{min}. If there exists an element list, q_k, in the path from q_1 to q_{min} whose stack becomes empty, we may skip the unmatching elements in q_k's descendant lists. The decision on the execution mode of each element list is according to our look-ahead-based algorithm (Lines 11-15). Note that the threshold value $DSkip_{q_i}$ is the threshold value for q_i, which is estimated using our model. When $min = n$, the algorithm outputs the join result using Procedure $showSolutions$ [10].

3.4.2 *Adaptive* Twig^{2}Stack

$Twig^2Stack$ [12] utilizes a hybrid approach combining both top-down and bottom-up evaluation processes. Since both processes use stacks, we call these stacks *top-down stacks* and *bottom-up stacks*, correspondingly. The top-down evaluation is similar to *PathStack* except that its evaluation is performed on a subtree of element lists as opposed to a single path in *PathStack*. In the bottom-up evaluation, $Twig^2Stack$ pushes the partial results into bottom-up stacks and enumerates the final results on these stacks. This bottom-up process ensures that the partial results in bottom-up stacks match part of the twig query.

We apply our adaptive schemes to the top-down evaluation of $Twig^2Stack$. The basic idea of skipping the unmatching elements is illustrated in Fig. 5b. For each label l in the twig query and its element list q_l, we refer the element lists of l's child and the parent labels in the twig query as q_l's child and parent element lists, respectively. If the top-down stack of the element list q_k is empty, unmatching elements in the descendant element lists of q_k can be skipped in the depth-first order.

The modified top-down evaluation of $Twig^2Stack$ with the look-ahead-based scheme is shown in Algorithm 6. In each iteration, we call Procedure $getMinSource(Q)$ to determine the element list q_{min} with the smallest $start$ value. Based on this $start$ value, we start to remove elements from the top-down stacks. The removed elements are pushed into the bottom-up stacks for further processing [12]. This removal is performed in two phases. In Phase 1 (Lines 6-12), the algorithm handles the path from the root element list (the element list corresponding to the root label of the twig query) to q_{min}. Meanwhile, it determines the element list with an empty top-down stack. If multiple empty top-down stacks are found, it uses the one which is nearest to the root element list. In Phase 2 (Lines 14-21), if an element list with an empty top-down stack is found in Phase 1 (denoted as $emptyList$), it calls Procedure $skipSubtree$ to skip the unmatching elements in the descendant element lists of $emptyList$ according to the look-ahead-based scheme. Otherwise, it performs a depth-first search (DFS) traversal on the subtree rooted at q_{min}. During the traversal, elements in the top-down stack are removed.

Algorithm 6. LA_Twig^2Stack: the modified top-down evaluation of $Twig^2Stack$ [12] with the look-ahead-based scheme

1: **while** $\neg end(Q)$ **do**
2: $q_{min} = getMinSource(Q)$; /*Procedures $end(Q)$ and $getMinSource(Q)$ are shown in Algorithm 5.*/
3: $oldc = c_{q_{min}}$;
4: $key = -1$, $emptyList = nil$;
5: /*Phase 1*/
6: $q = q_{min}$;
7: **while** $q \neq nil$ **do**
8: **while** $\neg S_q.empty()$ and $S_q.topEnd() < c_{q_{min}}.start$ **do**
9: $S_q.pop()$; /*The popped elements are pushed into the bottom-up stacks for further processing.*/
10: **if** $S_q.empty()$ **then**
11: $emptyList = q$;
12: $q = q.parent$;
13: /*Phase 2*/
14: **if** $emptyList = nil$ **then**
15: **for** each descendant element list of q_{min}, q, accessed in the DFS traversal on the subtree rooted at q_{min} **do**

TABLE 4
Machine Characteristics

Name	*P4*	*P3*
OS	Linux 2.4.18	Linux 2.4.18
Processor	Intel P4 2.8GHz with hardware prefetching	Intel P3 1.0GHz without hardware prefetching
L1 D-Cache	<8K, 64, 4>	<16K, 64, 4>
L2 cache	<512K, 128, 8>	<256K, 64, 8>
DTLB	64	64
Memory (bytes)	2.0G	512M

```
16:          while ¬S_q.empty() and
             S_q.topEnd() < c_qmin.start do
17:            S_q.pop();
18:      else
19:          skipSubtree(emptyList);
20:      if oldc = c_qmin then
21:          c_qmin = q_min.next();
Procedure: skipSubtree(emptyList)
 1:  for each descendant element list of emptyList, q,
     accessed in the DFS traversal on the subtree rooted at
     emptyList do
 2:      while ¬S_q.empty() and S_q.topEnd() < c_qmin.start do
 3:          S_q.pop();
 4:      key = c_q.parent.start;
 5:      if (c_q + DSkip_q).start < key then
 6:          c_q = I_q.SearchLarger(key); /*Probe mode on q*/
 7:      else
 8:          while c_q.start < key do
 9:              c_q = q.next(); /*Scan mode on q*/
```

3.5 Discussion

Our adaptive structural joins are proposed to improve the performance of memory-resident XML query processing. The strengths of our approach lie in its simplicity and its capability of deciding the execution modes as the join proceeds. This adaptive approach does not assume a priori information of join selectivity or data distribution. In addition, adding these adaptation mechanisms to existing static algorithms requires little modification.

Following the discussion by Babu and Bizarro on the performance issues of adaptive query processing [7], we discuss the following performance issues regarding our adaptations.

Quality of adaptation. All three adaptive algorithms can quickly detect the changes in the length of unmatching blocks and efficiently switch to the other execution mode. Compared on their ability to find a correct execution mode, the look-ahead-based algorithm is the best, the hybrid algorithm is the second, and the history-based algorithm is the last.

Runtime overhead. Our adaptive algorithms are light-weight. That is, the cost of switching between the execution modes is low and our algorithms do not need to reexecute an operation on a misprediction. Additionally, the overhead for the bookkeeping information in our adaptive schemes is low.

Thrashing. Thrashing in the execution modes (that is, frequently switching between scans and probes) is incurred with a high rate of changes in the lengths of unmatching blocks. The look-ahead-based algorithm can always detect the changes in the length of unmatching blocks through the look-ahead operation and is insensitive to thrashing. In contrast, thrashing makes the misprediction rate of the history-based algorithm high since it knows nothing about the future. The hybrid algorithm eases the thrashing effect on the history-based algorithm by utilizing the look-ahead information when detecting the need for changing the execution mode.

Finally, we note that the misprediction rate in our adaptive algorithms cannot fully determine the performance of our algorithms due to the different performance gains (respectively, penalties) of correct (respectively, incorrect) decisions. That is, a low misprediction rate does not mean a high performance of our adaptive algorithm. Therefore, we choose the variable cost as an indicator for the quality of our adaptation. The lower the variable cost, the better the quality of our adaptation.

4 Experimental Evaluation

In this section, we present a set of evaluation results for our proposed adaptive algorithms on binary and holistic structural joins. The measurement results on synthetic and real-world data sets are to demonstrate the end-to-end performance of our algorithms.

To compare the measurement results of adaptive algorithms with static algorithms, we use the elapsed time as the performance metric and also examine the speedup (the ratio of elapsed time) between two algorithms. In addition, we have compared the measurement results with the variable costs from our model. To make this comparison clearer, we normalize the performance result (either the elapsed time or the variable cost) of an algorithm by dividing it by the performance result of the corresponding static SCAN algorithm. This normalization is fair because the variable costs for a static SCAN algorithm are the same for a given join selectivity on a fixed hardware platform and the measurement results for a static SCAN algorithm are also nearly constant.

4.1 Experimental Setup

All experiments were run on two machines, P3 and P4. Some features of these machines are listed in Table 4. We define the *cache configuration* as a three-element tuple $< C, B, A >$, where C is the cache capacity in bytes, B is the cache line size in bytes, and A is the set associativity. Both the L1 and L2 caches are nonblocking and the L2 cache is unified.

TABLE 5
Unit Costs in Our Model (Cycles)

Platform	C_{AN}	C_{DN}	C_R	C_{NR}
P4	85	70	100	360
P3	130	122	120	244
P3_sp	70	56	800	1652

In our experiments, we first parsed the XML documents into a number of element lists using a Simple API for XML (SAX) parser [29]. Next, we constructed a CSB+-Tree index for each element list. The join algorithms were then executed on these element lists and CSB+-Tree indexes and output the number of join results. The cost of materializing join results was not included in our measurements as it was the same among all join algorithms.

We have implemented the static algorithms, including the ones with scan and index modes (denoted as SCAN and INX, respectively), and our adaptive algorithms, including *1bit-Adaptive* (1bit), *Look-ahead-Adaptive* (LA), and *Hybrid-Adaptive* (H). These algorithms were implemented in C++ and were compiled using g++ 3.2.2 with optimization flags (*O3*, *foptimize-sibling-calls*, and *finline-functions*). As we studied memory-resident structural joins, the element lists and the CSB+-Tree indexes in all experiments were always memory-resident and the memory usage never exceeded 80 percent.

Since we consider the main-memory performance of the join algorithms, we investigate whether software prefetching can improve the overall performance. The Intel platform provides two kinds of software prefetching instructions [1], including 1) *prefetchnta*, the nontemporal instruction that fetches the data into one way of the L2 cache to minimize the cache pollution, and 2) *prefetcht0/prefetcht1/prefetcht2*, the temporal instructions that fetch data into the L2 cache. Since the element lists were scanned once in structural joins, we used the nontemporal instruction *prefetchnta*. Additionally, we applied the temporal instruction *prefetcht2* to index probes using an existing prefetching scheme [11]. Note that hardware prefetching is enabled on P4. Since software prefetching had little performance impact on P4, we do not report the results of software prefetching on P4.

We used a hardware profiling tool, Performance Counter Library (PCL) [8], to measure the unit costs in our model. The unit costs calibrated from P3 and P4 are shown in Table 5. The values of C_{AN} and C_{DN} were obtained by running Algorithm 1 in the scan mode, that is, the **PROBE_MODE** is off. The value of C_R (respectively, C_{NR}) was measured by simulating index probes with ascending search keys when their paths resided (respecively, did not reside) in the cache. The values in the row of *P3_sp* were calibrated on P3 with software prefetching. The values of C_R and C_{NR} of *P3_sp* were larger than those of P3. This is because the tree index with the prefetching mechanism has a larger node size (the node size was eight cache lines for *P3_sp*).

We have conducted experiments with both synthetic and real-world data sets. We used two real-world data sets obtained from the XML data repository [36] (Digital Bibliography Library Project (DBLP) and the Protein Sequence Database), a benchmark data set, namely, XMark [35], and our own synthetic data sets. We mainly present our results on the DBLP and XMark data sets as the results on the Protein sequence database are very similar to those on DBLP.

TABLE 6
Characteristics of DQ0 on DBLP: (a) Descendant Join Selectivity with $year$ Varied and (b) the Number of Elements in the Ancestor List

$year$	88	89	90	91	92	93	94	95
author sel.(%)	1.0	1.2	1.4	1.8	2.0	2.7	3.1	3.4
$year$	96	97	98	99	00	01	02	03
author sel.(%)	3.0	4.0	4.7	5.5	6.2	6.3	7.2	7.4

(a)

author sel.(%)	1	2	3	4	5
Inp. length(10^3)	5.7	10.0	14.7	17.9	22.9
author sel.(%)	10	20	30	40	50
Inp. length(10^3)	43.1	86.7	130.7	180.1	226.0

(b)

We first chose a query DQ0 on DBLP, *inproceedings*[*@key like* "%year"]*//author*, to verify the effectiveness of our adaptive algorithms when the join selectivity changed. Table 6a shows the descendant join selectivity (denoted as author sel.) when the $year$ value in the predicate changes from 88 to 03. Since the ancestor and the descendant join selectivities have similar performance effects, we fixed the ancestor join selectivity to be 100 percent and varied the descendant join selectivity. Through carefully choosing the disjunction of the predicates [*@key like* "%year"] from Table 6a, we varied the descendant join selectivity into two ranges: 1) low selectivity: 1 percent to 5 percent and 2) high selectivity: 10 percent to 50 percent. Table 6b shows the number of elements in the ancestor list for these ranges of join selectivity. The number of elements in the descendant list of DQ0 is always 1,075,100.

We then chose four pairs of queries to examine the effect of different distributions of matching elements on our adaptive algorithms. Each pair of queries, one for DBLP $(\mathrm{DQ}i)$ and the other for XMark $(\mathrm{XQ}i)$, has a similar descendant join selectivity. The statistics of these four pairs of queries are shown in Table 7. The number in "()" following the tag name is the number of elements in thousands in the list. The performance comparison of the

TABLE 7
Queries on DBLP and XMark

Query	*Ancestor*(10^3)	*Descendant*(10^3)	*D. sel. (%)*
DQ1	inproceedings[p_1] (7.6)	author (1,075.1)	1.4
XQ1	open_auction[p_2] (0.8)	text (105.1)	1.5
DQ2	inproceedings[p_3] (34.1)	author (1,075.1)	8.4
XQ2	open_auction[p_4] (4.6)	text (105.1)	8.4
DQ3	inproceedings[p_5] (49.3)	author (1,075.1)	21.8
XQ3	open_auction (12.0)	text (105.1)	21.8
DQ4	article (173.7)	year (479.4)	36.2
XQ4	listitem (60.5)	parlist (20.8)	36.9

Note: $p_1 = \mathrm{@key}$ *like "%90",* $p_2 = \mathrm{reserve} \geq 500$ *and reserve* $< 1{,}000$, $p_3 = \mathrm{@key}$ *like "%88" or @key like "%03,"* $p_4 = \mathrm{reserve} < 500$, $p_5 = \mathrm{@key}$ *like "%88" or @key like "%93" or @key like "%94" or @key like "%95."*

TABLE 8
Path Queries and Twig Queries on DBLP and XMark

Query name	*Data set*	*Path query*
DBLP-P1	DBLP	//article//title//sub
DBLP-P2	DBLP	//article//title//i
DBLP-P3	DBLP	//dblp//article//author
XMark-P1	XMark	//site//open_auctions//text
XMark-P2	XMark	//people//person//text
XMark-P3	XMark	//parlist//listitem//text
Query name	*Data set*	*Twig query*
DBLP-T1	DBLP	//dblp//inproceedings[title]//author
DBLP-T2	DBLP	//dblp//article[author][.//title]//year
DBLP-T3	DBLP	//inproceedings[author][.//title]//booktitle
XMark-T1	XMark	//site/open_auctions[.//bidder/personref]//reserve
XMark-T2	XMark	//people//person[.//address/zipcode]/profile//education
XMark-T3	XMark	//item[location]/description//keyword

static algorithms for DQ2 and DQ3 on DBLP and XQ2 and XQ3 on XMark is shown in Fig. 1.

The path queries and twig queries on DBLP and XMark are shown in Table 8. These queries are selected with different combinations of total sizes of element lists and join selectivities. The twig queries are the same as those used in previous studies on twig joins [12].

In addition to these third-party data sets, we generated our own synthetic data sets for better control on the join selectivity and the distribution of the matching elements. We generated the data set RANDOM, which consists of random trees generated using three parameters: depth, fanout, and the number of labels. For simplicity, we set the maximum depth and fanout to be the number of labels. The node labels in the trees were uniformly distributed. Since we obtained similar results from data sets with different numbers of labels and numbers of nodes, we present the results for the data set with five million nodes and six different labels: $A_0, A_1, \ldots, A_5$. The queries used on the synthetic data sets are shown in Table 9. Among twig queries SYN-Ti ($1 \leq i \leq 4$), a query with a larger i has a more complex twig structure (for example, more element lists and more branches involved in the twig query). For these queries, we randomly removed elements from the element list to obtain a certain join selectivity. Since all of these queries have a common join $A_0//A_1$, for simplicity, we fixed the join selectivity on A_0 to be 100 percent and varied the join selectivity on A_1.

We further evaluated our adaptive schemes with data sets of different distributions. In particular, we generated these data sets in two steps. First, we generated a RANDOM data set. Second, we removed the elements from A_0 such that unmatching elements of A_1 in the join $A_0//A_1$ conform to a certain distribution. Figs. 6a, 6b, 6c, and 6d show the four distributions used in our experiments. Suppose the descendant list A_1 has L unmatching

TABLE 9
Path Queries and Twig Queries on Synthetic Data Sets

Query name	*Query*
SYN-Pn	$//A_0//A_1...//A_n$
SYN-T1	$//A_0[.//A_1]//A_2$
SYN-T2	$//A_0[.//A_1][.//A_2]//A_3$
SYN-T3	$//A_0[.//A_1//A_2]//A_3//A_4$
SYN-T4	$//A_0//A_1[.//A_2]//A_3[.//A_4]//A_5$

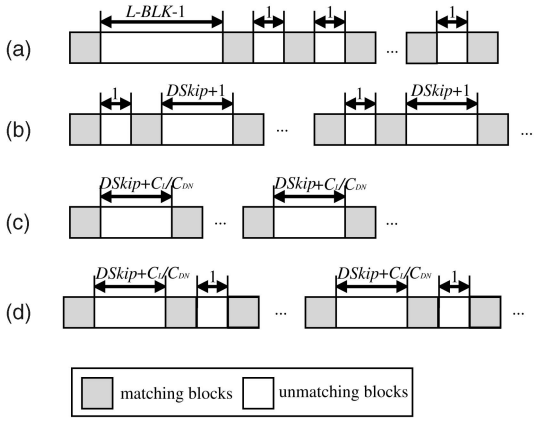


Fig. 6. Different distributions of unmatching elements in A_1 in the join $A_0//A_1$. (a) BEST. (b) 1bit_WORST. (c) LA_WORST.(d) H_WORST.

elements and these unmatching elements are distributed into BLK blocks ($BLK \leq L$). Thus, we generated the following four data sets:

- BEST. This data set evaluates the advantage of our adaptive algorithms over the static algorithms. The length of the first unmatching block is $(L - BLK - 1)$ (INX is likely to be efficient on this unmatching block) and the remaining blocks consist of one element each (SCAN is efficient on these unmatching blocks). All of our adaptive algorithms choose the probe mode for the first or second unmatching block. The remaining blocks are scanned.
- 1bit_WORST. This data set tests the thrashing in the length of the unmatching blocks over our adaptive algorithms. In particular, the history-based algorithm makes a wrong decision on each unmatching block. That is, it chooses a probe for an unmatching block consisting of one element and chooses scans for an unmatching block with a length $(DSkip + 1)$.
- LA_WORST. This data set tests the overhead of look-ahead operations. Suppose C_L is the average cost of one cache miss. Thus, C_L/C_{DN} is the number of unmatching elements whose total scanning cost equals to one cache miss stall. In our experiments, C_L is set to the cache stall of one L2 cache miss. The length of unmatching blocks is $(DSkip + C_L/C_{DN})$ such that the look-ahead-based algorithm accesses one extra unmatching element in each look-ahead operation. This access incurs one cache miss in the worst case.
- H_WORST. This data set tests the overhead of both look-ahead operations and incorrect decisions made based on history (due to thrashing in the length of the unmatching blocks). In particular, the hybrid algorithm chooses a probe through a look-ahead operation for unmatching block with a length $(DSkip + C_L/C_{DN})$ and chooses a probe on an unmatching block consisting of one element.

In our experiments, we varied both L and BLK. Specifically, for each L value, we varied the BLK value to evaluate the effect of BLK. The queries on these data sets are the same as the ones on the data set RANDOM (Table 9).

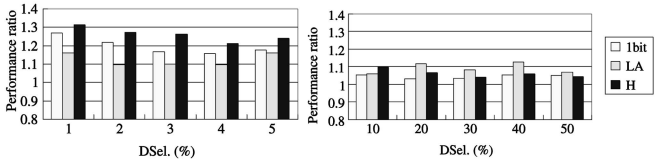


Fig. 7. Ratio of the execution time of the adaptive algorithms with the random probe model and those with our model on P4. A ratio larger than one means that the adaptive algorithms with our model are faster than those with the random probe model. (a) Low DSel. (b) High DSel.

4.2 Evaluation on Binary Joins

4.2.1 Third-Party Data Sets

Once of our adaptive algorithms with our model and those with the random probe model [20] to examine the impact of modeling the sortedness of index probes. We varied the join selectivity of DQ0 on DBLP. Fig. 7 shows the ratio of the execution time of the adaptive algorithms with the random probe model and those with our model on P4. We do not show the ratios on P3 since the results on P3 are similar to those on P4. The *DSkip* value was 21 and 35 by our model and by the random probe model, respectively. The adaptive algorithms with our model are considerably faster than the ones with the random probe model. When the join selectivity is low, the number of index probes in our adaptive algorithms is large. As a result, there is a large amount of cache reuse between these ascending index probes. Our model improves the adaptation much better than the random probe model. In comparison, when the join selectivity is high, there are few index probes in the adaptive algorithms and the performance difference between the two models is small.

We then evaluated our adaptive algorithms on binary joins with the join selectivity of DQ0 on DBLP varied. Fig. 8 shows the elapsed time of the static and the adaptive algorithms on P3 and P4 with the join selectivities between 1 percent and 5 percent on the left and between 10 percent and 50 percent on the right. These measurements were obtained without software prefetching on either platform. Since the performance trend on P3 is similar to that on P4, we mainly present the results on P4 in the remainder of this paper.

Our algorithms improve the join performance on both platforms regardless of the join selectivity. When the join selectivity is low, our adaptive algorithms consistently outperform the best static ones with an average speedup of 1.8. In particular, our adaptive algorithms have an average speedup of 3.1 and 1.9 over the SCAN (*Stack-Tree*) and INX (*Anc_Des_B+*) algorithms, respectively. This performance improvement is mainly because our adaptive algorithms avoid most of the unworthy index probes. When the join selectivity is high, our adaptive algorithms (except for the look-ahead-based algorithm) consistently outperform the best static ones with an average speedup of 1.35. The performance improvement is because our adaptive algorithms can utilize index probes to skip large unmatching blocks even when the overall join selectivity is high. Note that the join selectivity alone does not determine the relative performance of the two static algorithms, as shown on the left of Fig. 8. In contrast, our adaptive algorithms, which do not require knowledge of the join selectivity, consistently achieve better performance than the best static ones.

To further investigate the performance improvement of our adaptive algorithms, we examine the statistics on probes and scans of static algorithms. We choose the DQ0 with the descendant join selectivity of 3 percent and the other with 30 percent as examples, denoted as $\mathrm{DQ0}_L$ and $\mathrm{DQ0}_H$. The numbers of probes in $\mathrm{DQ0}_L$ and $\mathrm{DQ0}_H$ are around 12,100 and 50,500, respectively. The INX algorithm overuses probes for $\mathrm{DQ0}_H$ and has a performance penalty because the number of unmatching elements in $\mathrm{DQ0}_H$ is less than that in $\mathrm{DQ0}_L$. This causes a performance penalty.

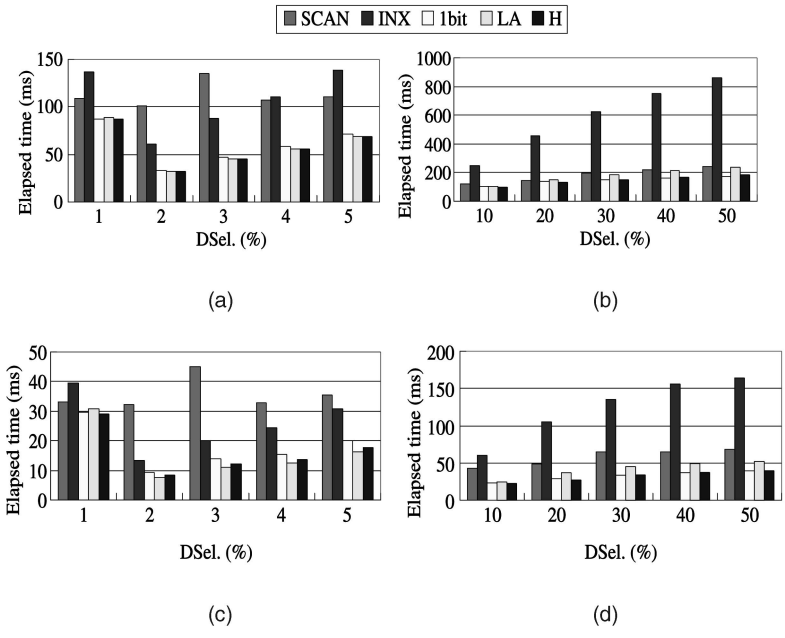


Fig. 8. Binary joins: DQ0 execution time on P3 and P4 with different descendant join selectivities. (a) P3. (b) P4.

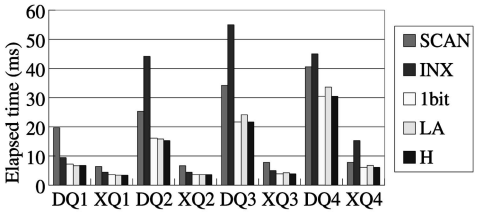


Fig. 9. Binary joins: the execution time on DBLP and XMark on P4.

None of the static algorithms is optimal for either query: 1) Using INX for $DQ0_L$, the number of probes skipping fewer than $DSkip$ unmatching elements is about 10,600 (88.0 percent of the total number of probes). The history-based and hybrid algorithms eliminate around 80 percent of these unworthy probes. 2) Using SCAN for $DQ0_H$, the total number of elements in the unmatching blocks with a size larger than $DSkip$ is 329,000 (53.8 percent of the total number of unmatching elements). The history-based and hybrid algorithms use index probes to skip 95.4 percent of these unmatching elements.

Comparing the performance of individual adaptive algorithms, we find that all three algorithms have similar performance when the join selectivity is low. However, the performance of the look-ahead-based algorithm is similar to that of the best static one or, sometimes, slightly worse when the join selectivity is high. This downside of the look-ahead-based algorithm is due to the increased look-ahead overhead when the join selectivity is high. As expected, the hybrid algorithm is consistently close to the best of the history-based and the look-ahead-based algorithms.

Additionally, we studied the performance impact of software prefetching on both P3 and P4. Software prefetching on P4 improves the performance of both algorithms slightly or even causes performance degradation. On P3, software prefetching greatly improves the performance of SCAN and slightly improves the performance of INX and our adaptive algorithms. The performance comparison among the static and adaptive algorithms of P3 with software prefetching is similar to that of P4 without software prefetching. This indicates that software prefetching with careful tuning can achieve a similar performance improvement to hardware prefetching. In the remainder of this section, we only focus on the results on P4 without software prefetching.

We further examine the performance comparison of the four pairs of queries on DBLP and XMark in Fig. 9. The distributions of the matching elements in the data sets contribute to the performance difference of the algorithms on them. For instance, SCAN outperforms INX in DQ2 and DQ3 but underperforms in XQ2 and XQ3. Regardless of the join selectivity and the distribution in the two data sets, our adaptive algorithms have performance better than or similar to the best static algorithm. The average speedup over the best static algorithm with our adaptive algorithms is 1.45 and 1.35 on DBLP and XMark, respectively.

4.2.2 Synthetic Data Sets

Fig. 10 shows the execution time of our algorithms on RANDOM. Regardless of join selectivities, the performance of our adaptive algorithms is always similar to or better than that of the best static algorithm.

Finally, we evaluated our binary joins on the synthetic data sets with different distributions. Figs. 11a, 11b, 11c, and 11d show the normalized execution time and variable cost of our algorithms on synthetic data sets with different distributions. These results were obtained when the descendant join selectivity was 3 percent and similar results were obtained when the descendant join selectivity was varied.

We summarize our findings on Fig. 11 on five aspects. First, the normalized costs on all data sets increase as the BLK value increases. This is because the number of probes becomes larger and the advantage of INX and our adaptive algorithms over SCAN becomes smaller.

Second, the performance comparison between the static schemes depends on both the join selectivity and the distribution of unmatching elements. Specifically, given a fixed join selectivity, the performance of INX varies with the distribution of unmatching elements. For example, given a join with a join selectivity of 3 percent, INX is over six times faster than SCAN under the distribution 1bit_WORST, whereas INX is 20 percent slower than SCAN under the uniform distribution.

Third, our adaptive algorithms outperform the static algorithms on these data sets. In particular, the hybrid scheme is nearly three times faster than the best static algorithm on BEST.

Fourth, among the three adaptive schemes, the hybrid scheme has a good and robust performance, which is always similar to or better than the best performance of the history-based and the look-ahead-based schemes. On data sets BEST, 1bit_WORST, and LA_WORST, the hybrid scheme is the best among the static and the adaptive algorithms and it is second (slightly slower than the look-ahead-based algorithm) on H_WORST.

Finally, our cost model accurately predicts the relative performance of algorithms. For example, the model correctly predicts the best algorithm on each data set.

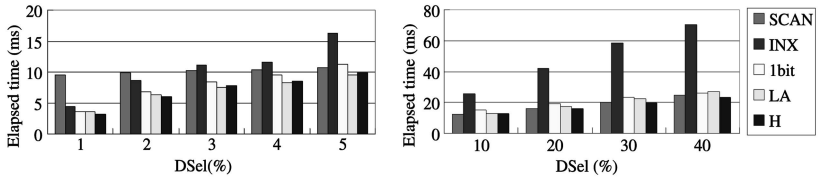


Fig. 10. Binary joins: execution time on RANDOM on P4. (a) Low DSel. (b) High DSel.

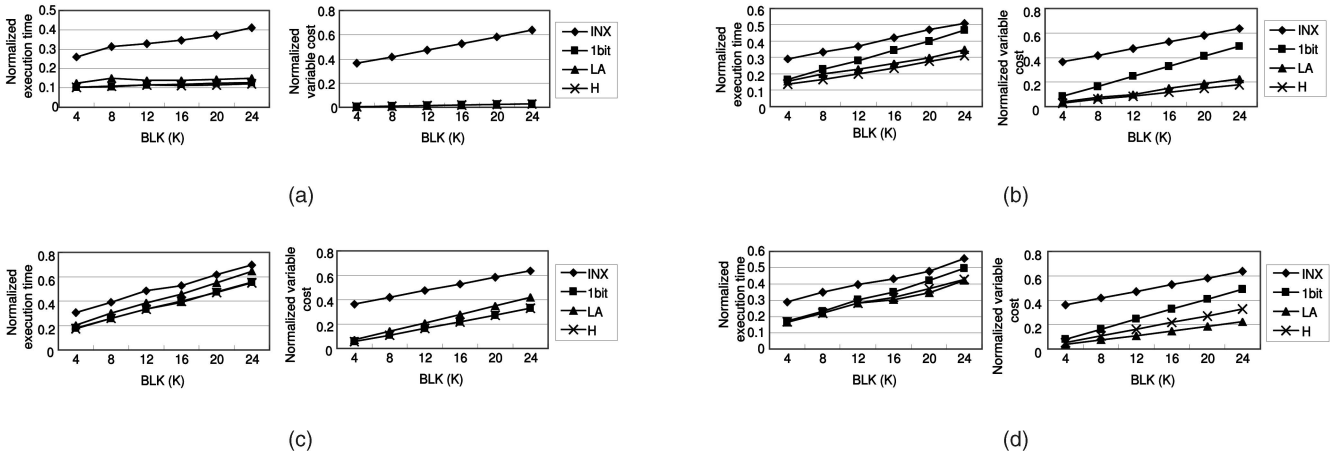


Fig. 11. Normalized execution time and variable cost of binary joins on P4. (a) BEST. (b) 1bit_WORST. (c) LA_WORST. (d) H_WORST.

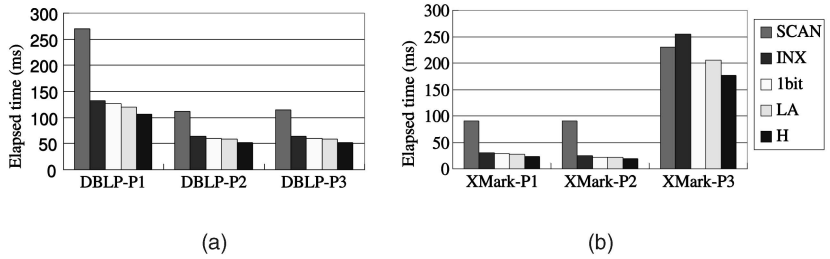


Fig. 12. Execution time of *PathStack* on P4: (a) DBLP and (b) XMark.

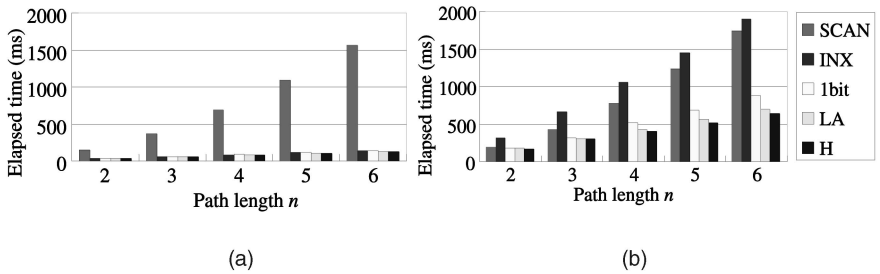


Fig. 13. Execution time of *PathStack* on P4: $A_1//A_2 \ldots //A_n$ on the data set RANDOM. (a) $\mathrm{DSel.} = 3\%$. (b) $\mathrm{DSel.} = 30\%$.

4.3 Evaluation on Path Queries

After studying the binary structural join, we evaluated our adaptive algorithms on path queries. Fig. 12 shows the performance comparison of our algorithms on the path queries on DBLP and XMark. The history-based and the look-ahead-based schemes have similar performance on these queries. The hybrid scheme is the best among all of the algorithms, which has an average speedup of 1.2 and 1.3 over the best static algorithm on DBLP and XMark, respectively.

Fig. 13 shows the performance comparison of our algorithms on the data set RANDOM with the length of the path query varied. When the descendant join selectivity is low (3 percent), the gap between the performance of the SCAN and the INX schemes increases as the length of the path query increases. In contrast, when the descendant join selectivity is high (30 percent), this performance gap becomes smaller as the length of the path query increases. The complexity of a path query increases the difficulty in determining the correct static algorithm on the join.

Our adaptive algorithms greatly improve the performance of holistic structural joins on path queries. Among them, the hybrid approach provides a good and robust performance. It is 15 percent faster than the best static algorithm when the join selectivity is low and has an average speedup of 1.9 over the best static algorithm when the join selectivity is high. The performance improvement is mainly due to the elimination of the unnecessary index probes on all lists.

Fig. 14 shows the normalized execution time and variable cost of *PathStack* on synthetic data sets with different distributions. We present the results for the path

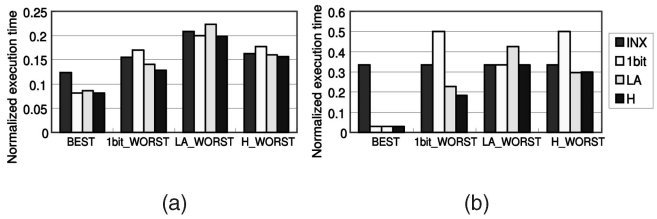
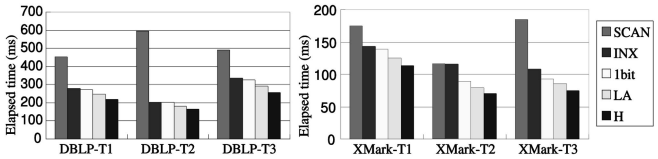


Fig. 14. *PathStack*: normalized execution time and variable cost on P4 $(\mathrm{DSel.} = 3\%)$.



(a) (b)

Fig. 15. Execution time of $Twig^2Stack$ on P4: (a) DBLP and (b) XMark.

query $//A_0//A_1//A_2$ only, because we obtained similar results for other path queries. Regardless of the data distributions, our adaptive algorithms outperform the static algorithms. The hybrid approach is typically a winner among the three adaptive algorithms. Additionally, our cost model is accurate in predicting the relative performance of the static and adaptive algorithms.

4.4 Evaluation on Twig Queries

After studying the path queries, we evaluated our adaptive algorithms on twig queries. Fig. 15 shows the performance comparison of our algorithms on the twig queries on DBLP and XMark. The history-based and the look-ahead-based schemes have similar performance on these twig queries. The hybrid scheme is the best among all the algorithms, which has an average speedup of 1.3 and 1.4 over the best static algorithm for DBLP and XMark, respectively.

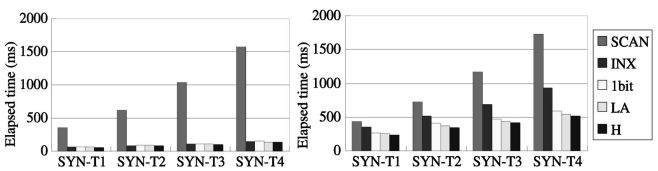
Fig. 16 shows the performance comparison of our algorithms on the data set RANDOM. One observation is that INX is preferred over SCAN as the twig query becomes more complex. This is mainly because the condition for being a matching element is more stringent and more unmatching elements can be skipped as the query becomes more complex. Regardless of the complexity of the twig query, our adaptive algorithms consistently outperform the static algorithms.

Fig. 17 shows the normalized execution time and variable cost of $Twig^2Stack$ on synthetic data sets with different distributions. Similarly to the results on path queries, our adaptive algorithms outperform the static algorithms on the data sets with different distributions. Additionally, our cost model accurately predicts the relative performance of the static and adaptive algorithms.

4.5 Summary

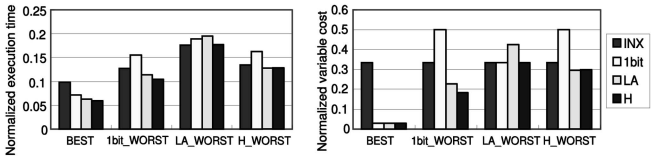
We have studied the performance of our adaptive algorithms in comparison with the state-of-the-art static algorithms for binary structural joins [2], [15], linear path queries [10], and twig queries [12]. Our measurement results show that our adaptive algorithms consistently achieve good performance regardless of changes in the join selectivity and data distribution. The speedup of our adaptive algorithms over the best static algorithm for a given workload is up to three times. Additionally, the hybrid adaptive algorithm usually outperforms the 1-bit-history and the look-ahead ones.

The success of our adaptation schemes is mainly due to our accurate estimation of the access costs in the structural join. Specifically, our model accurately estimates the number of elements to skip to make an index probe cost-effective at every step of matching. This estimation helps the history-based branch prediction, the look-ahead-based algorithm, and the hybrid algorithm to make correct decisions most of the time. Consequently, the adaptive algorithms achieve a good performance consistently without the knowledge of the join selectivity or the data distribution.



(a) (b)

Fig. 16. Execution time of $Twig^2Stack$ on P4: twig queries of different complexities on RANDOM. (a) $\mathrm{DSel.} = 3\%$. (b) $\mathrm{DSel.} = 30\%$.



(a) (b)

Fig. 17. $Twig^2Stack$: normalized execution time and variable cost of SYN-T1 on P4 $(\mathrm{DSel.} = 3\%)$.

5 Conclusion

We have investigated the adaptive index utilization in structural joins and found that index utilization is beneficial only under certain join selectivity and distribution of matching elements. To address this problem, we propose adaptive algorithms for index utilization in memory-resident structural joins and develop a cost model to facilitate the decision making for the adaptation. We have experimentally evaluated our adaptive structural joins with synthetic data sets, as well as real-world data sets. Our experimental results indicate that our adaptive algorithms improve the performance of memory-resident structural joins without 1) a priori knowledge of the join selectivity and data distribution or 2) human intervention.

Acknowledgments

The authors thank the anonymous reviewers for their comments on the earlier versions of the draft. This work was supported by Grants DAG05/06.EG11, HKUST6263/04E, and 617206, all from the Hong Kong Research Grants Council.

References

[1] *IA-32 Intel Architecture Optimization Reference Manual,* ftp://download.intel.com/design/Pentium4/manuals/24896611.pdf, 2006.

[2] S. Al-Khalifa, H. Jagadish, N. Koudas, J. Patel, D. Srivastava, and Y. Wu, "Structural Joins: A Primitive for Efficient XML Query Pattern Matching," *Proc. 18th Int'l Conf. Data Eng. (ICDE '02),* 2002.

[3] G. Antoshenkov, "Query Processing in DEC Rdb: Major Issues and Future Challenges," *IEEE Data Eng. Bull.*, vol. 16, no. 4, pp. 42-52, 1993.
[4] G. Antoshenkov, "Dynamic Optimization of Index Scans Restricted by Booleans," *Proc. 12th Int'l Conf. Data Eng. (ICDE '96)*, 1996.
[5] G. Antoshenkov and M. Ziauddin, "Query Processing and Optimization in Oracle RDB," *VLDB J.*, vol. 5, no. 4, pp. 229-237, 1996.
[6] R. Avnur and J.M. Hellerstein, "Eddies: Continuously Adaptive Query Processing," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '00)*, 2000.
[7] S. Babu and P. Bizarro, "Adaptive Query Processing in the Looking Glass," *Proc. Second Biennial Conf. Innovative Data Systems Research (CIDR '05)*, 2005.
[8] R. Berrendorf, H. Ziegler, and B. Mohr, "PCL: Performance Counter Library," http://www.fz-juelich.de/zam/PCL/, 2006.
[9] P. Bohannon, P. McIlroy, and R. Rastogi, "Main-Memory Index Structures with Fixed-Size Partial Keys," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '01)*, 2001.
[10] N. Bruno, N. Koudas, and D. Srivastava, "Holistic Twig Joins: Optimal XML Pattern Matching," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '02)*, 2002.
[11] S. Chen, P.B. Gibbons, and T.C. Mowry, "Improving Index Performance through Prefetching," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '01)*, 2001.
[12] S. Chen, H. Li, J. Tatemura1, W. Hsiung, D. Agrawal, and K. Selcuk Candan, "*Twig*^{2}*Stack*: Bottom-Up Processing of Generalized Tree Pattern Queries over XML Documents," *Proc. 32nd Int'l Conf. Very Large Data Bases (VLDB '06)*, 2006.
[13] T. Chen, J. Lu, and T.W. Ling, "On Boosting Holism in XML Twig Pattern Matching Using Structural Indexing Techniques," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '05)*, 2005.
[14] J.M. Cheng, D.J. Haderle, R. Hedges, B.R. Iyer, T. Messinger, C. Mohan, and Y. Wang, "An Efficient Hybrid Join Algorithm: A DB2 Prototype," *Proc. Seventh Int'l Conf. Data Eng. (ICDE '91)*, 1991.
[15] S.-Y. Chien, V.J. Tsotras, C. Zaniolo, and D. Zhang, "Efficient Structural Joins on Indexed XML Documents," *Proc. 28th Int'l Conf. Very Large Data Bases (VLDB '02)*, 2002.
[16] D. Comer, "Ubiquitous B-Tree," *ACM Computing Surveys*, vol. 11, no. 2, pp. 121-137, 1979.
[17] S. Elnaffar, W. Powley, D. Benoit, and P. Martin, "Today's DBMSs: How Autonomic Are They?" *Proc. 14th Int'l Conf. Database and Expert Systems Applications (DEXA '03)*, 2003.
[18] T. Grust, M. van Keulen, and J. Teubner, "Staircase Join: Teach a Relational DBMS to Watch Its Axis Steps," *Proc. 29th Int'l Conf. Very Large Data Bases (VLDB '03)*, 2003.
[19] P.J. Haas and J.M. Hellerstein, "Ripple Joins for Online Aggregation," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '99)*, 1999.
[20] R.A. Hankins and J.M. Patel, "Effect of Node Size on the Performance of Cache-Conscious B+-Trees," *Proc. ACM Int'l Conf. Measurement and Modeling of Computer Systems (SIGMETRICS '03)*, 2003.
[21] J. Hellerstein, M. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. Shah, "Adaptive Query Processing: Technology in Evolution," *IEEE Data Eng. Bull.*, vol. 23, no. 2, pp. 7-18, 2000.
[22] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, second ed. Morgan Kaufman, 1996.
[23] H. Jiang, H. Lu, W. Wang, and B. Ooi, "XR-Tree: Indexing XML Data for Efficient Structural Join," *Proc. 19th Int'l Conf. Data Eng. (ICDE '03)*, 2003.
[24] H. Jiang, W. Wang, H. Lu, and J.X. Yu, "Holistic Twig Joins on Indexed XML Documents," *Proc. 29th Int'l Conf. Very Large Data Bases (VLDB '03)*, 2003.
[25] K. Kim, S.K. Cha, and K. Kwon, "Optimizing Multidimensional Index Trees for Main Memory Access," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '01)*, 2001.
[26] T.J. Lehman and M.J. Carey, "Query Processing in Main Memory Database Management Systems," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '86)*, 1986.
[27] J. Lu, T.W. Ling, C.-Y. Chan, and T. Chen, "From Region Encoding to Extended Dewey: On Efficient Processing of XML Twig Pattern Matching," *Proc. 31st Int'l Conf. Very Large Data Bases (VLDB '05)*, 2005.
[28] S. Manegold, P. Boncz, and M. Kersten, "Generic Database Cost Models for Hierarchical Memory Systems," *Proc. 28th Int'l Conf. Very Large Data Bases (VLDB '02)*, 2002.
[29] T.S.P. Organization, "SAX: Simple API for XML," http://www.saxproject.org, 2006.
[30] J. Rao and K.A. Ross, "Making B$^+$-Trees Cache Conscious in Main Memory," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '00)*, 2000.
[31] A. Shatdal and J.F. Naughton, "Adaptive Parallel Aggregation Algorithms," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '95)*, 1995.
[32] S. Sipani, K. Verma, J.A. Miller, and B. Aleman-Meza, "Designing a High-Performance Database Engine for the 'Db4XML' Native XML Database System," *J. System and Software*, vol. 69, nos. 1-2, pp. 87-104, 2004.
[33] A. Skelley, "DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes," *Proc. 16th Int'l Conf. Data Eng. (ICDE '00)*, 2000.
[34] M. Stillger, G.M. Lohman, V. Markl, and M. Kandil, "LEO—DB2's LEarning Optimizer," *VLDB J.*, pp. 19-28, 2001.
[35] "XMark—An XML Benchmark Project," http://monetdb.cwi.nl/xml/index.html, 2006.
[36] "XML Data Repository," http://www.cs.washington.edu/research/xmldatasets/, 2006.
[37] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman, "On Supporting Containment Queries in Relational Database Management Systems," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '00)*, 2001.

Bingsheng He received the bachelor's degree in computer science from Shanghai Jiao Tong University (1999-2003). He is a PhD student in the Computer Science and Engineering Department at the Hong Kong University of Science and Technology (HKUST). His research interests are in database systems, especially cache-centric query processing techniques.

Qiong Luo received the BS and MS degrees in computer sciences from Beijing (Peking) University, China, in 1992 and 1997, respectively, and the PhD degree in computer sciences from the University of Wisconsin-Madison in 2002. She is an assistant professor in the Computer Science and Engineering Department, Hong Kong University of Science and Technology (HKUST). Her research interests are database systems, with focus on data management and analysis techniques related to network applications.

Byron Choi received the BEng degree in computer engineering from the Hong Kong University of Science and Technology (HKUST) in 1999 and the MSE and PhD degrees in computer and information science from the University of Pennsylvania in 2002 and 2006, respectively. He is currently an assistant professor in the School of Computer Engineering, Nanyang Technological University, Singapore.