

Incremental Maintenance of XML Structural Indexes

Ke Yi^{*†}

Dept. Computer Science
Duke University
yike@cs.duke.edu

Hao He[‡]

Dept. Computer Science
Duke University
haohe@cs.duke.edu

Ioana Stanoi

IBM T. J. Watson
Research Center
irs@us.ibm.com

Jun Yang[‡]

Dept. Computer Science
Duke University
junyang@cs.duke.edu

ABSTRACT

Increasing popularity of XML in recent years has generated much interest in query processing over graph-structured data. To support efficient evaluation of path expressions, many structural indexes have been proposed. The most popular ones are the 1-index, based on the notion of graph bisimilarity, and the recently proposed $A(k)$ -index, based on the notion of local similarity to provide a trade-off between index size and query answering power. For these indexes to be practical, we need effective and efficient incremental maintenance algorithms to keep them consistent with the underlying data. However, existing update algorithms for structural indexes essentially provide no guarantees on the quality of the index; the updated index is usually larger size than necessary, degrading the performance for subsequent queries.

In this paper, we propose update algorithms for the 1-index and the $A(k)$ -index with provable guarantees on the resulting index quality. Our algorithms always maintain a *minimal* index, i.e., merging any two index nodes would result in an incorrect index. For the 1-index, if the data graph is acyclic, our algorithm further ensures that the index is *minimum*, i.e., it has the least number of index nodes possible. For the $A(k)$ -index, we show that the minimal index our algorithm maintains is also the unique minimum $A(k)$ -index, for both acyclic and cyclic data graphs. Finally, through experimental evaluation, we demonstrate that our algorithms bring significant improvement over previous methods, in terms of both index size and update time.

1. INTRODUCTION

Increasing popularity of XML in recent years has gener-

^{*}Part of the work was done while the author was visiting IBM T. J. Watson Research Center.

[†]Supported in part by the National Science Foundation through CAREER grant CCR-9984099 and ITR grant EIA-0112849.

[‡]Supported by a National Science Foundation CAREER Award under grant IIS-0238386.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD 2004 June 13-18, 2004, Paris, France.

Copyright 2004 ACM 1-58113-859-8/04/06 . . . \$5.00.

ated much interest in query processing over graph-structured data. A number of commercial database vendors are making significant efforts to support XML natively, rather than convert it to the traditional relational model. One of the major challenges of this task is to provide support for efficient query processing over XML. To summarize the structure of such data and to support path expression [4] evaluation, novel structural indexes have been proposed [11, 9, 17, 7]. Among the most popular ones are the 1-index [11], based on the notion of graph bisimilarity, and the recently proposed $A(k)$ -index [9], based on the notion of local similarity to provide a trade-off between index size and query answering power. Some structural indexes have also been used as statistical synopses for estimating selectivities of path expressions [3, 16].

Compared with traditional relational indexes, much less attention has been directed to the problem of maintaining structural indexes for XML, with the exception of recent work in [8]. After an XML document is updated, its structural index must be properly maintained so that subsequent queries have a view of the summarized data that is consistent with the updated document. For structural indexes to be practical, we need efficient index maintenance algorithms that guarantee the accuracy and efficiency of these indexes for querying. There are two basic approaches to index maintenance: reconstruction and incremental maintenance. Reconstruction is simple and usually leads to high-quality indexes, but the overhead of reconstruction makes it unattractive even for databases with moderate update rates. The second approach, incremental maintenance, updates the existing index incrementally as soon as the underlying database changes. The cost of computing and applying incremental index updates can be potentially much lower than that of reconstruction.

Designing good maintenance algorithms is challenging because of the delicate balance between *efficacy* and *efficiency*. *Efficacy* means preserving the quality of the structural summary. For the same underlying data, there are many correct structural summaries, but they vary greatly in size and hence in query performance. The algorithm should ensure that the updated index is not expanded unnecessarily. On the other hand, *efficiency* here means that the algorithm itself should be efficient. In some cases, obtaining the smallest possible structural summary is very expensive, so settling on a reasonably small structural summary would be more preferable. Finding the right balance in this trade-off is not trivial. Reconstruction provides perfect efficacy, but severely lacks efficiency. In contrast, although the pre-

viously proposed update algorithms in [9] are efficient, we show that their efficacy is lacking: the updated index usually has a much larger size than necessary, degrading the performance for subsequent query evaluations.

In this paper, we demonstrate that it is possible to achieve high degrees of both efficacy and efficiency in designing incremental maintenance algorithms for structural indexes. We focus on three types of updates: edge insertion, edge deletion, and subgraph addition. Edge insertion and deletion constitute the basic operations upon which other kinds of updates (e.g., node insertion and deletion) can be based. Although subgraph addition can also be processed by inserting edges one at a time, we provide a separate, more efficient algorithm for it since it is such a common operation. We restrict ourselves to the 1-index and the $A(k)$ -index, but we believe our techniques can also be used for other structural indexes based on node partitioning. We develop efficient update algorithms for the 1-index and the $A(k)$ -index that, in contrast to previous algorithms, provide provable guarantees on the resulting index quality. More precisely, we make the following contributions:

1. Our algorithms always maintain a *minimal* index, i.e., merging any two index nodes would result in an incorrect index.
2. If the data graph is acyclic, we show that there is a unique minimal 1-index that is also *minimum*, i.e., it has the least number of index nodes possible. This result further ensures that our algorithm always maintains the minimum 1-index for acyclic data graphs.
3. For cyclic data graphs, where there might be more than one minimal 1-indexes, we show by experiments that the minimal 1-index maintained by our algorithm is always very close to the minimum, if not the same.
4. For any data graph (acyclic or cyclic), we show that there is a unique minimal $A(k)$ -index that is also minimum, which ensures that our algorithm always maintains the minimum $A(k)$ -index for any data graph.
5. Through an extensive experimental study, we demonstrate that our algorithms are not only effective in preserving index quality, but also very efficient in terms of computation cost.

The rest of the paper is organized as follows. We first survey previous work in Section 2. In Section 3, we present the data model of XML and its structural indexes, as well as the basic concepts related to the theory and algorithms. We give a general overview of our algorithms in Section 4, followed by the detailed update algorithms for the 1-index and $A(k)$ -index in Section 5 and 6, respectively. We study their practical performance experimentally in Section 7. Finally, we conclude in Section 8.

2. PREVIOUS WORK

Query optimization for XML has been a popular subject of study [10, 5]. Indexing is essentially used to avoid exhaustive traversal of the documents for query processing. Signature-based techniques have the same goal of reducing the search space. They have been used extensively in information retrieval and have also been adapted for XML data recently in [14, 15]. With this approach, each node of the XML tree is annotated with the bitwise OR of the hash values of its child nodes. Existence of a tag in the subtree of

a node can therefore be estimated by comparing the hashed value of the child tag with the signature of the node. Updates may however lead to recomputation of signatures of all ancestors.

Structural summaries for XML have been used for indexing, query pruning and rewriting, and selectivity estimation. DataGuides [6] was one of the first structural summaries used in XML query processing. The notion of simulation, more commonly used in graph theory, was applied in [5] to schema validation as well as query pruning and rewriting for semistructured data. A number of structural indexes based on simulation followed. The 1-index [11] partitions data nodes into equivalence classes based on bisimilarity. To reduce the size of the 1-index, the $A(k)$ -index was proposed [9]. It uses local similarity for partitioning, thereby compressing the 1-index at the cost of losing some structural information about the underlying data. Very recently, other techniques [17, 7] have been proposed to further improve the flexibility and efficiency of the $A(k)$ -index.

Three important issues need to be considered for any index: construction, query evaluation, and maintenance. Paige and Tarjan [12] gave an iterative splitting algorithm to construct a 1-index in $O(m \log n)$ time, where m is the number of edges and n is the number of nodes in the data graph. In [9], an algorithm based on similar ideas is used to construct an $A(k)$ -index in time $O(km)$. Many different query evaluation strategies use these structural indexes; see [11, 9] for details. In this paper, we only focus on the maintenance issue.

The only known update algorithm for the 1-index is the *propagate* algorithm from [8], which uses Paige and Tarjan’s construction algorithm [12] to handle edge changes. This algorithm essentially provides no guarantee on the quality of the resulting index. In the experiments of [8], the index was shown to have 3%–5% more nodes than the minimum index after a relatively small number of edge insertions (500 in a data graph with about 200,000 nodes); no performance results were reported for deletions. A subgraph addition algorithm based on reconstruction was also given in [8].

Intuitively, because of its locality, the $A(k)$ -index should be easier to maintain than the 1-index. However, no good update algorithm for the $A(k)$ -index has been proposed so far, except for some simple algorithms mentioned in [8, 17]. These approaches all suffer from the same problem of generating too many unnecessary nodes, which undermines the compactness advantage of the $A(k)$ -index. Designing efficient incremental maintenance algorithms for the $A(k)$ -index was left as an interesting area for future research in [8].

3. PRELIMINARIES

Data model. In this paper, we model XML or other semi-structured data as a directed, labeled graph $G = (V, E, root, \Sigma, label, oid, value)$. Each edge in E indicates an object-subobject or IDREF relationship. Each node in V is labeled with a string from Σ via the *label* function and with a unique identifier via the *oid* function. It may also optionally have a value given by the *value* function. There is a single *root* node with the distinguished label **ROOT** with no incoming edges. An example XML document under this model is shown in Figure 1, where object-subobject relations are shown in solid lines, and IDREF relations are shown in dashed lines. A database with multiple XML documents

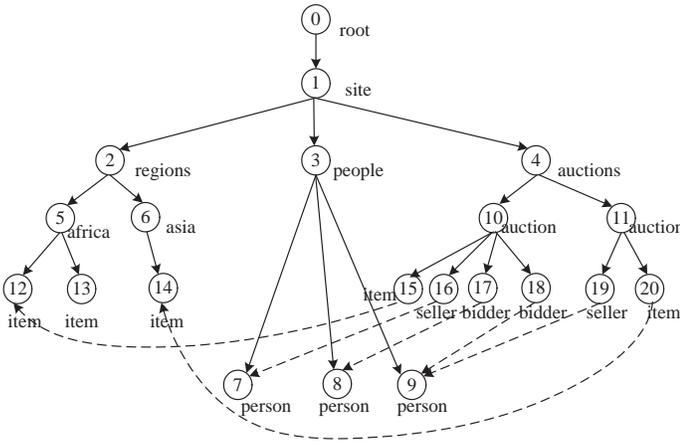


Figure 1: An XML database example.

can be modeled as a single data graph with an artificial root connecting graphs corresponding to the individual files.

We refer to the nodes and edges in V and E as *data nodes* and *data edges*, or *dnodes* and *dedges*, respectively, to differentiate from those in an index graph, to be introduced below. We will use u, v, \dots to denote dnodes, and $Succ(u)$ to denote the set of u 's successors, i.e., $Succ(u) = \{v | (u, v) \in E\}$.

Structural indexes. A structural index (or structure summary) for a data graph takes the form of another labeled directed graph (V_I, E_I) , which is built by the following general procedure: (1) partition the dnodes into classes according to some equivalence relation, (2) make an *index node* (or *inode*) for each equivalence class, with all dnodes in this class being its *extent*, and (3) add an *index edge* (or *iedge*) from inode I to inode J if there is a dedge from some dnode in the extent of I to some dnode in the extent of J . We use $\Phi(G)$ to denote a structural index built for data graph G , and $I[v]$ to denote the inode whose extent contains dnode v . From now on, we will not distinguish between an inode and its extent when there is no confusion. Since a structural index is completely determined by its partition of the dnodes, we also do not distinguish between an index and its dnodes partitions. We define $Succ(I)$ to be $\bigcup_{u \in I} Succ(u)$, the dnodes successors of dnodes in I , and $ISucc(I) = \{J | (I, J) \in E_I\}$, the index successors of I . We use $\mathcal{I}, \mathcal{J}, \dots$ to represent sets of inodes, and define $Succ(\mathcal{I}) = \bigcup_{I \in \mathcal{I}} Succ(I)$.

Evaluation of path expressions can often be made faster with a structural index $\Phi(G)$ by executing the path expression R on $\Phi(G)$, which is often much smaller than the original data graph G . The results of R is contained in the union of the extents of the inodes that match R , because any structural index that is constructed by the procedure above is *safe*. However, not all structural indexes are *precise*, i.e., the result of some query R on $\Phi(G)$ may contain false positives.

Different structural indexes can be obtained by choosing different equivalence relations in step (1) above. The 1-index [11] uses *bisimilarity* [13] to partition the dnodes. For our purpose, we use the following equivalent definition for the 1-index based on the notion of *stability* [12]:

Definition 1. An inode I is *stable* with respect to J if

either $I \subseteq Succ(J)$ or $I \cap Succ(J) = \emptyset$. For a data graph G , an index $\Phi(G)$ is *stable* w.r.t. index $\Phi'(G)$ if for any inode $I \in \Phi(G)$, $I' \in \Phi'(G)$, I is stable w.r.t. I' .

Definition 2. A structural index $\Phi(G)$ is called a *1-index* if (1) all dnodes in any inode of $\Phi(G)$ have the same label, and (2) it is stable with respect to itself. A *minimum 1-index* is the 1-index with the minimum number of inodes.

Note that if I is not stable w.r.t. J , we can make it stable by splitting I into $I \cap Succ(J)$ and $I - I \cap Succ(J)$. This is the basic operation for ensuring the correctness of the index in the construction algorithm of [12] and our algorithms.

There may be more than one 1-index for a given data graph, all of which can be used in the same way for query evaluation. Of course they differ in performance: the smaller the index, the better the performance. The best one is the minimum 1-index, while the worst is the data graph itself (also a valid 1-index) where we do not gain anything from using it. In [12], the following result gives the relationship between the minimum 1-index and other 1-indexes.

Definition 3. For a data graph G , a structural index $\Phi(G)$ is a *refinement* of another index $\Phi'(G)$ if for any inode $I \in \Phi(G)$, there exists an inode $I' \in \Phi'(G)$ such that $I \subseteq I'$.

LEMMA 1. *There is a unique minimum 1-index for any given data graph, and any other 1-index is a refinement of the minimum 1-index.*

However, even the minimum 1-index can sometimes have too many inodes, especially for highly irregular data graphs, resulting in poor query performance. To alleviate the problem, the $A(k)$ -index [9] was proposed to shrink the index size by using *k-bisimilarity* to partition dnodes. We use the following equivalent definition for the $A(k)$ -indexes.

Definition 4. Given any data graph G , the $A(0)$ -index is the structural index obtained by simply partitioning the dnodes of G by their labels. For $1 \leq i \leq k$, a structural index $\Phi(G)$ is called an $A(i)$ -index if there exists an $A(i-1)$ -index $\Phi'(G)$ such that $\Phi(G)$ is a refinement of $\Phi'(G)$ and it is stable with respect to $\Phi'(G)$. A *minimum $A(k)$ -index* is the $A(k)$ -index with the minimum number of inodes.

Note that the $A(k)$ -index is not precise any more, because it only preserves paths of length up to k . For path expressions longer than k , it may generate false positives and we need a *validation* step on the original data graph to eliminate them. Nevertheless, in [9], it was shown by experiments that even with this extra validation step, the total evaluation cost is much less than that of 1-index, due to the small sizes of the $A(k)$ -indexes, for typical values of $k = 2, \dots, 5$.

A result parallel to Lemma 1 holds for the $A(k)$ -index [9]:

LEMMA 2. *For any given data graph G , there is a unique minimum $A(k)$ -index. Any other $A(k)$ -index is a refinement of the minimum $A(k)$ -index.*

Quality of indexes. When there are updates to the data graph G , it is sometimes difficult and costly to maintain the minimum index, but as discussed before, there are many correct indexes and any of them can be used for query processing in the same way as the minimum index. However,

they range from the minimum index to the data graph itself, hence differ greatly in performance [9, 17]. Thus, we would like to keep the index size as small as possible when doing maintenance. To measure the effectiveness of our update algorithms, we define the quality of the index to be

$$\frac{\# \text{ inodes in the index}}{\# \text{ inodes in the minimum index}} - 1,$$

which we would like to keep as close to zero as possible. Note that this is the same metric used by [8] to measure the quality of the index after a sequence of updates.

4. ALGORITHMS OVERVIEW

The basic idea behind our new update algorithms is to iteratively make local improvements after correctness is first ensured. All our algorithms consist of a *split* phase and a *merge* phase. Therefore we will sometimes generally refer to them as split/merge algorithms. The *split* phase uses ideas from the index construction algorithms to first make the index correct by splitting some inodes, while the *merge* phase tries to merge nearby inodes together without violating any constraint, one pair at a time, until no more merges can be made. Both split and merge phases are carried out in an iterative and local manner: we start from the newly inserted (or deleted) edge, and proceed step by step. In each step, we try to split (or merge) the children of some new inode generated from previous splits (or merges). The nice property of our algorithms is that, although these operations are carried out in a local manner, each inode in the resulted index cannot be merged with *any* other inode without violating the stability constraint. We say that such an index is *minimal*. The precise definitions of minimal indexes take slightly different forms for the 1-index and the $A(k)$ -index, so we will defer them to the respective sections. From Lemmas 1 and 2, we know that the minimum index is unique. However, there might be more than one minimal indexes for a given data graph. Nonetheless, in many cases we can prove that there is a unique minimal index, i.e., for acyclic 1-indexes and general $A(k)$ -indexes. In these cases, our algorithms can further guarantee that the minimum index is always maintained.

5. UPDATES FOR THE 1-INDEX

5.1 Edge Insertion and Deletion

The algorithms. We first use a running example to demonstrate how our algorithm updates the 1-index when a dedge is inserted into the data graph. See Figure 2. The data graph is shown in (a), where we use letters to represent labels and numbers to represent dnodes. The new dedge to be inserted is shown with a dashed line. The 1-index before the update is shown in (b), where the inodes' extents are shown in brackets. The split phase first checks if there is an iedge between the two inodes containing the source and sink of the new dedge. In this case there is not, so we split the inode $\{3, 4\}$ in (b) into an inode that contains dnode 4 and one that contains the rest of dnodes (Figure 2(c)). Then, this split triggers the split of inode $\{6, 7\}$ because it now becomes unstable with respect to the two new inodes resulted from the previous split (Figure 2(d)). Now, every inode is stable with respect to every inode and the split phase ends. The

merge phase starts by looking for an inode among the siblings of $\{4\}$, the inode containing the sink of the new dedge, to see if there is an inode that has the same label and the same set of index parents. We find inode $\{5\}$ in this case and then merge inodes $\{4\}$ and $\{5\}$ together (Figure 2(e)). Next, we iteratively consider the possible merges among the children of newly generated inodes from previous merges. In this example, we will merge inodes $\{7\}$ and $\{8\}$ together. The final result of the update is shown in Figure 2(f).

More formally, our algorithm first checks if the new edge (u, v) makes v not bisimilar with the rest of the dnodes in $I[v]$. If yes, we split $I[v]$ into one inode containing v itself and one that contains the rest of the dnodes. A *compound block* is a set of inodes that are the new inodes resulted from a previous split. The split phase basically uses the Paige-Tarjan's 1-index construction algorithm to iteratively split inodes until we get a stable partition with respect to itself (hence correct). We start with only one compound block consisting of the two new inodes. In each of the split steps, we take out a compound block \mathcal{I} , pick an inode $I \in \mathcal{I}$ such that $|I| \leq \frac{1}{2} \sum_{J \in \mathcal{I}} |J|$, and make all other inode stable with respect to $Succ(I)$ and $Succ(\mathcal{I} - \{I\})$. This in turn may split other inodes and generate new compound blocks, which are added to the queue of compound blocks. The split phase ends when queue is empty.

The merge phase starts from $I[v]$ and tries to merge inodes together iteratively until no more merges can be made. We first look for an inode with the same label and index parents as $I[v]$. If one exists, we merge it with $I[v]$, and put the newly merged inode into a queue of merged inodes. In each of the following merge steps, we take out one merged inode I from the queue, and consider the possible merges among the index successors of I . We also add newly merged inodes into the queue. The merge phase ends when the queue is empty.

Our complete 1-index edge insertion algorithm is described in Figure 3. The edge deletion algorithm differs only slightly. For simplicity of presentation, we assume that there is no self-cycles in the 1-index (i.e., an inode that points to itself), which is true for virtually all XML databases. Our algorithms can be modified to take care of self-cycles as well, only that some details get a little messy. Note that in the algorithm description we only specify how the partition of dnodes gets updated; we do not bother to state explicitly how iedges are handled, because they are completely determined from the partition by the definition of structural indexes (Section 3). These iedges can also be easily maintained as we update the inode extents, using techniques similar to those in [8].

Efficacy. Now we give the formal definition for *minimal* 1-indexes.

Definition 5. For a data graph G , a 1-index $\Phi(G)$ is *minimal* if for any two inodes $I, J \in \Phi(G)$, either (1) they have different labels, or (2) there exists an inode $K \in \Phi(G)$ such that $I \cup J$ is not stable with respect to K .

For example, for the data graph in Figure 2(a) after the dedge insertion, the index in 2(f) is a minimal 1-index (and minimum at the same time), the ones in 2(d) and (e) are not minimal, and the one in 2(c) is not even a valid 1-index. Note that a 1-index is minimal if and only if it has no two inodes that have the same label and the same set of index

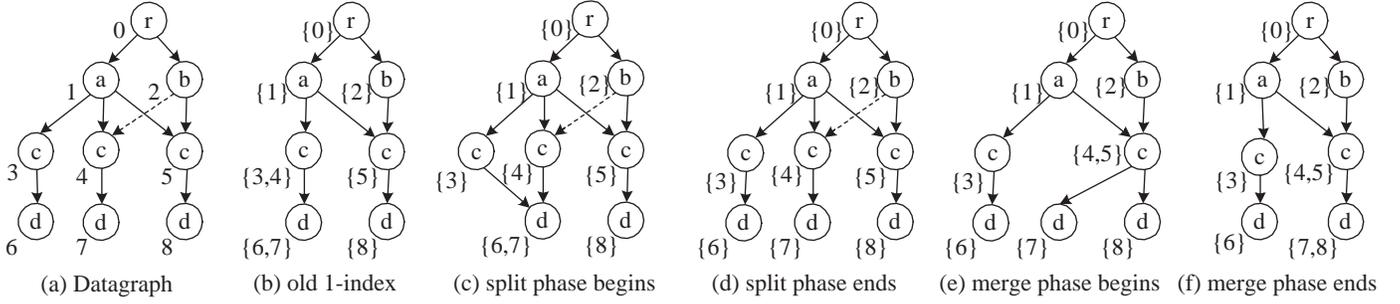


Figure 2: An example of updating the 1-index after a dedge insertion.

```

procedure insert_1_index_edge( $u, v$ )
begin
  add a dedge from  $u$  to  $v$ ;
  if there is an iedge from  $I[u]$  to  $I[v]$  then return;
  /* replace the 2 lines above with the following for deletions:
  delete the dedge from  $(u, v)$ ;
  if there exist  $u' \in I[u], v' \in I[v]$  and there is a dedge
  from  $u'$  to  $v'$  then return; */
  // split phase
   $Q = \emptyset$ ;
  if  $|I[v]| > 1$  then
    split  $I[v]$  into  $I_1 = \{v\}$  and  $I_2 = I - \{v\}$ ;
     $Q = \{\{I_1, I_2\}\}$ ;
  while  $Q \neq \emptyset$  do
    pick any  $\mathcal{I} \in Q$ , remove it from  $Q$ ;
    pick  $I \in \mathcal{I}$  s.t.  $|I| \leq \frac{1}{2} \sum_{J \in \mathcal{I}} |J|$ ;
    if  $|\mathcal{I}| \geq 3$  then insert  $\mathcal{I} - \{I\}$  into  $Q$ ;
    foreach inode  $K \in ISucc(I)$  do
      split  $K$  into  $K_1 = K \cap Succ(I)$  and  $K_2 = K - K_1$ ;
      split  $K_1$  into  $K_{11} = K_1 \cap Succ(\mathcal{I} - \{I\})$  and
       $K_{12} = K_1 - K_{11}$ ;
      let  $\mathcal{K} = \{K_{11}, K_{12}, K_2\} - \{\emptyset\}$ ;
      if  $|\mathcal{K}| \geq 2$  then
        if  $\exists \mathcal{J} \in Q$  s.t.  $K \in \mathcal{J}$  then
          replace  $K$  in  $\mathcal{J}$  with the inodes in  $\mathcal{K}$ ;
        else add  $\mathcal{K}$  to  $Q$ ;
  // merge phase
   $Q = \emptyset$ ;
  look for an inode  $J$  with the same label as  $v$  among  $I[v]$ 's
  siblings that have the same set of index parents as  $I[v]$ ;
  if such an inode  $J$  exists then
    merge  $I[v]$  and  $J$  into  $K = J \cup I[v]$ ;
     $Q = \{K\}$ ;
  while  $Q \neq \emptyset$  do
    pick any  $I \in Q$ , remove  $I$  from  $Q$ ;
    let  $\mathcal{I} = ISucc(I)$ ;
    partition  $\mathcal{I}$  into equivalent classes according to their
    labels and index parents;
    foreach equivalent class  $\mathcal{J} \subseteq \mathcal{I}$  do
      if  $|\mathcal{J}| \geq 2$  then
        merge the inodes in  $\mathcal{J}$  into  $J = \bigcup \mathcal{J}$ ;
         $Q = Q - \mathcal{J}$ ;
        insert  $J$  into  $Q$ ;
end

```

Figure 3: Insert an edge into 1-index

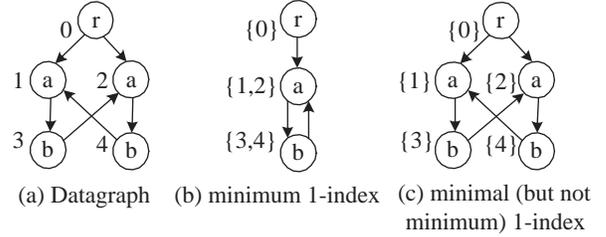


Figure 4: Minimal 1-indexes might not be unique.

parents, which follows directly from the definition of stability. Minimal 1-indexes might not be unique. For example, the indexes in Figure 4(b) and 4(c) are both minimal 1-indexes for the data graph in 4(a), but only the one in 4(b) is minimum.

LEMMA 3. *If the 1-index before the update is minimal, the new index generated by the split/merge algorithm is also a minimal 1-index.*

PROOF. Let (u, v) be the dedge just inserted (or deleted). The algorithm first checks if this edge update changes any index predecessor-successor relations. If no, it simply returns. The resulted index is still a minimal 1-index simply because the index before the update is a minimal 1-index.

Assume now the update indeed causes some changes to the index. Let us call the data graph before the update G_0 , and the one after the update G_2 . Imagine we relabel v of G_2 with a new label that is different from all others, and call this data graph G_1 . We call the 1-index before the update $\Phi_0(G_0)$, the one after the split phase but before the merge phase $\Phi_1(G_1)$ (with relabeled v), and the final 1-index $\Phi_2(G_2)$. We will show that if $\Phi_0(G_0)$ is a minimal 1-index, then $\Phi_1(G_1)$ and $\Phi_2(G_2)$ are both minimal 1-indexes, too.

If v is in an inode by itself in $\Phi_0(G_0)$, the split phase does nothing. In this case, the only inode in $\Phi_1(G_1)$ that may have a different set of index parents than in $\Phi_0(G_0)$ is $I[v]$, which by definition, has a distinguished label in $\Phi_1(G_1)$, therefore it cannot be merged with any other inode. So $\Phi_1(G_1)$ is minimal in this case.

Suppose otherwise that v shares an inode with some other ndnodes in $\Phi_0(G_0)$. After insertion, v has a different set of index parents than these other ndnodes, and is then singled out by the split phase, which afterwards propagates the split using the Paige-Tarjan's algorithm. That $\Phi_1(G_1)$ is indeed a correct 1-index follows from the correctness of the Paige-

Tarjan’s algorithm, which always returns the coarsest self-stable refinement of the starting partition. To see it is also minimal, we define the index parents of a dnode w to be the set of inodes, each of which contains at least one of w ’s parents, i.e., the set $\{I[w'] \mid w \in Succ(w')\}$, and we will show that the split phase maintains the invariant that no two dnodes in different inodes have the same label and the same index parents. Note that if the index is a valid 1-index, the index parents of any dnode w are the same as the index parents of $I[w]$, so this invariant is true in a 1-index if and only if this 1-index is minimal. The invariant is true before the split phase because $\Phi_0(G_0)$ is a minimal 1-index. It is still maintained when we relabel v and single it out as a separate inode, because v ’s label is now different from any others. In each of the following split steps, whenever we split an inode into two, the newly generated two inodes must have at least one different index parent — otherwise they will not get split. Since we already know $\Phi_1(G_1)$ is a 1-index, it is also minimal because the invariant holds.

Next, we need to show that $\Phi_2(G_2)$ is a minimal 1-index, which is the result of labeling v back to its original label and applying the merge phase on $\Phi_1(G_1)$. It is easy to see that it is a 1-index because the merge phase only merges inodes that have the same label and index parents. Since $\Phi_1(G_1)$ is minimal, and the only difference between G_1 and G_2 is v ’s label, the only possible two inodes that may have the same label and the same index parents in $\Phi_1(G_2)$ are $I[v]$ and some other inode. The merge phase exactly starts by looking for this only possible merge. Further notice that after two inodes are merged, it can only trigger new possible merges among the inode successors of the newly merged inode because the index parents of all other inodes remain unchanged. Therefore, when the merge phase completes, no two inodes in $\Phi_2(G_2)$ can be merged, so $\Phi_2(G_2)$ is minimal. \square

Keeping the 1-index minimal is probably the best one can do with reasonable cost, since it is much cheaper to check if the 1-index is minimal, as our algorithms do, than to determine if it is minimum. For example, in order to find out the 1-index in Figure 4(c) is not minimum, we need to be able to detect two merges *simultaneously*, and the number of such simultaneous merges might be as high as $\Theta(n)$. In practice, it is often good enough to keep the 1-index minimal, and in many cases, the minimal 1-index indeed turns out to be the minimum 1-index. Even if we are unlucky to get stuck in a minimal 1-index that is not minimum, our experiments show that the difference between the two is often very small.

Many data graphs are acyclic. For example, in a bibliography database, if we want to model the reference relations with IDREF edges, it is an acyclic graph as a paper can only reference papers that appear earlier in time. Many other XML databases that model hierarchical relations are naturally acyclic, or even trees. For such databases, our algorithms can provide an even stronger guarantee that the minimum 1-index is always maintained, because the minimal 1-index is unique in this case.

LEMMA 4. *For any acyclic data graph G , there is a unique minimal 1-index $\Phi(G)$, which is also minimum.*

PROOF. First we show that any 1-index of G is also acyclic. Suppose there was a cycle in the 1-index. By definition, for any iedge $I \rightarrow J$, any dnode in J has at least one parent

in I . By following iedges backwards in a cycle, we know there exists a path of arbitrary length in G , which could only happen if G is cyclic, too.

Suppose that $\Phi(G)$ is the minimum 1-index and $\Phi'(G)$ is a minimal 1-index different from $\Phi(G)$. We order the inodes in $\Phi(G)$ topologically and pick the first inode I that does not appear in $\Phi'(G)$. By Lemma 1, $\Phi'(G)$ is a refinement of $\Phi(G)$, so there exists at least two inodes $I'_1, I'_2 \in \Phi'(G)$ such that $I'_1 \subset I, I'_2 \subset I$ and I'_1 and I'_2 have the same label. For any index parent J of I , J also appears in $\Phi'(G)$ because J is before I in the topological order. Then J is also an index parent of I'_1 and I'_2 in $\Phi'(G)$ because each dnode in I has at least one parent in J . For any inode J that is not an index parent of I , J cannot be an index parent of I'_1 or I'_2 , either, because J does not contain any parent of any dnode in I , and both I'_1 and I'_2 are subsets of I . So I'_1 and I'_2 have the same set of index parents, which are the same as those of I . This contradicts with the fact that $\Phi'(G)$ is minimal. \square

Combining Lemma 3 and 4, we have:

THEOREM 1. *For acyclic data graphs, the split/merge algorithm always maintains the minimum 1-index during edge insertions and deletions. For cyclic data graphs it always maintains a minimal 1-index.*

Efficiency. Theorem 1 gives a theoretical guarantee on the efficacy of the split/merge algorithm, but how costly it is in terms computation cost? We continue to use $\Phi_0(G_0)$, $\Phi_1(G_2)$, $\Phi_2(G_2)$ to denote the index before the update, between the split and merge phase, and after the update, respectively. It is easy to see that the numbers of split and merge operations are $|\Phi_1(G_2)| - |\Phi_0(G_0)|$ and $|\Phi_1(G_2)| - |\Phi_2(G_2)|$, respectively. The first part is essentially the cost of the *propagate* algorithm, while the second part is the minimum number of merges required to shrink the intermediate result down to minimal. Unfortunately, in the worst case, this intermediate index $\Phi_1(G_2)$ could have much more nodes than the index before or after the update. See for example Figure 5, where the triangles represent two subtrees with the same structure. By arbitrarily enlarging these subtrees, we can have an intermediate index that has $\Omega(n)$ more nodes than the old or the new index. This is also a problem to the *propagate* algorithm and was identified in [8].

Nevertheless, the worst-case example of Figure 5 is rather contrived and is rare in practice. As observed by [8], as well as our own experiments with both real-life and benchmark data, the intermediate index on average only has 0.01% more nodes, which means that the update algorithm is really “incremental”, operating only on a very small fraction of the whole index.

Since we have an additional merge phase, the cost of the split/merge algorithm is certainly higher than the *propagate* algorithm, but we feel the merge phase is always worth doing, not only because it gives us a nice theoretical guarantee on the quality of the resulted index, but also for the following practical considerations: (1) With the merge step, we can effectively keep the index size small, leading to a much lowered reconstruction frequency. (2) The merge phase always makes the index smaller, hence higher query performance. Typically we have more queries than updates, so the effort spent in improving the quality of the index very likely can be paid back by the savings from subsequent query evaluations.

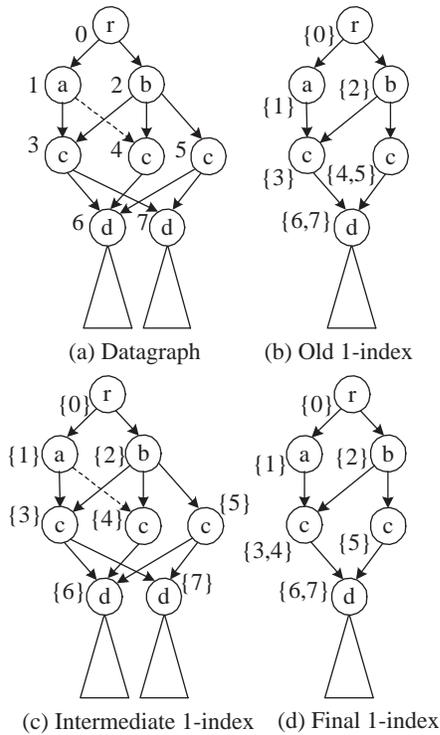


Figure 5: Update cost could be high in the worst case.

Finally, as an implementation note, when we split inodes using $Succ(I)$ (or $Succ(I - \{I\})$), we in fact can split all inodes containing at least one dnode in $Succ(I)$ at the same time by scanning $Succ(I)$ once and creating $K \cap Succ(I)$ for each K . The same technique is used in [12, 8].

5.2 Subgraph Addition

We model a subgraph also as a labeled, rooted directed graph, which can certainly be added by inserting its dnodes and their incident dedges one by one using our edge insertion algorithm. But since subgraph addition occurs so frequently, we design a more efficient algorithm that performs the insertions in a “batched” manner. The basic idea is to build the 1-index first for the new subgraph, and then add all the edges between the new subgraph and the existing data graph using the edge insertion algorithm.

Note that the root of the new subgraph must be in an inode by itself in the 1-index of the subgraph. As an optimization, we can insert all the incoming edges to the root of the subgraph and then perform the merge phase just once. The algorithm $add_1_index_subgraph$ is shown in Figure 6.

The following corollary follows from Theorem 1.

COROLLARY 1. *Algorithm $add_1_index_subgraph$ maintains the minimum 1-index for acyclic data graphs and a minimal 1-index for cyclic data graphs.*

Naturally one would like to delete subgraphs efficiently as well. This is easy, too. Have a special node with a distinguished label DELETE, and add a dedge from this node to the root of the subgraph that we want to delete. This new dedge will single out this subgraph from the rest of index, and then we can just delete it from the index.

procedure $add_1_index_subgraph(G')$

begin

 build the 1-index $\Phi'(G')$ for the new subgraph G' ;

 union $\Phi'(G')$ with the current 1-index $\Phi(G)$;

 add all incoming dedges that go into r , the root of G' ;

 do merge phase of $insert_1_index_edge$ starting at $I[r]$;

foreach other dedge (u, v) between G' and G **do**

$insert_1_index_edge(u, v)$;

end

Figure 6: Add a subgraph in 1-index.

6. UPDATES FOR THE $A(k)$ -INDEX

The algorithm. Our ideas and techniques for updating the 1-index can be extended to handle updates for the $A(k)$ -index as well. As identified in [8], the $A(k)$ -index is difficult to maintain by itself because updating it requires information contained in an $A(k-1)$ -index. Thus, the basic idea in our algorithm is to maintain all the $A(0), A(1), \dots, A(k)$ -indexes together using our 1-index update algorithms. When maintaining the $A(i)$ -index, we use the $A(i-1)$ -index as a reference to make split and merge decisions. We will first describe the algorithm, and then discuss how to implement it in a space- and time-efficient manner. Note that all these $A(i)$ -indexes can be easily built while we build the $A(k)$ -index; in fact, the construction algorithm [9] builds all the $A(0), A(1), \dots, A(k)$ -indexes in order. In the following, we only consider edge insertions and deletions; subgraph addition can be done in a very similar way as we did for the 1-index.

The $A(k)$ -index update algorithm also consists of a split phase to guarantee correctness and a merge phase to ensure minimality. Suppose the new edge to be inserted (or deleted) is (u, v) . We first look for the largest i such that the $A(i)$ -index will not be affected by the edge update. The split phase first creates a new inode containing v itself for each of the $A(i+1)$ to $A(k)$ -indexes. These “initial” splits generate a number of compound blocks (in the 1-index, we have only one compound block at the beginning), and we put them in a queue. Afterwards, we iteratively split other inodes in a way very similar to what we did for the 1-index. The only difference is that, when we stabilize other inodes with respect to a compound block in the $A(i)$ -index, we need to consider all the inodes in the $A(i+1)$ to $A(k)$ -indexes. The merge phase also proceeds similarly as for the 1-index. For each of the affected $A(i)$ -indexes, we first try to merge the inodes containing v with another inode. Next, we iteratively merge other inodes together. In each step, if I is a new inode in the $A(i)$ -index generated from a previous merge, we consider the possible merges among the inodes in the $A(i+1)$ -index that contains at least one dnode with a parent in I .

The detailed edge insertion (and deletion) algorithm for the $A(k)$ -index is shown in Figure 7. We use $\Phi^{(i)}(G)$ to denote the $A(i)$ -index of data graph G ; $I^{(i)}, J^{(i)}$ are some inodes in the $A(i)$ -index; $I^{(i)}[v]$ denotes the inode in the $A(i)$ -index that contains dnode v . We also use $\mathcal{I}^{(i)}, \mathcal{J}^{(i)}$ to denote sets of inodes in the $A(i)$ -index.

```

procedure insert_A(k)_index_edge( $u, v$ )
begin
  find the largest  $i$  s.t.  $v \in Succ(I^{(i)}[u])$ ,
  if no such  $i$  exists, set  $i = -1$ ;
  add a dedge from  $u$  to  $v$ ;
/* replace the three lines above with the following for deletions:
  delete the dedge from  $u$  to  $v$ ;
  find the largest  $i$  s.t.  $v \in Succ(I^{(i)}[u])$ ,
  if no such  $i$  exists, set  $i = -1$ ; */
  // split phase begins
   $Q = \emptyset$ ;
  for  $j = i + 2$  to  $k$ 
    if  $|I^{(j)}[v]| > 1$  then
      split  $I^{(j)}[v]$  into  $I_1^{(j)} = \{v\}$  and  $I_2^{(j)} = I^{(j)}[v] - \{v\}$ ;
      if  $j \leq k - 1$  then
        insert  $\{I_1^{(j)}, I_2^{(j)}\}$  into  $Q$ ;
  // iterate to split others
  while  $Q \neq \emptyset$  do
    pick any  $\mathcal{I}^{(j)} \in Q$  with the smallest  $j$ ;
    remove  $\mathcal{I}^{(j)}$  from  $Q$ ;
    pick  $I^{(j)} \in \mathcal{I}^{(j)}$  s.t.  $|I^{(j)}| \leq \frac{1}{2} \sum_{J^{(j)} \in \mathcal{I}^{(j)}} |J^{(j)}|$ ;
    if  $|\mathcal{I}^{(j)}| \geq 3$  then insert  $\mathcal{I}^{(j)} - \{I^{(j)}\}$  into  $Q$ ;
    foreach inode  $K^{(l)}$ ,  $j + 1 \leq l \leq k$  do
      split  $K^{(l)}$  into  $K_1^{(l)} = K^{(l)} \cap Succ(I^{(j)})$ 
      and  $K_2^{(l)} = K^{(l)} - K_1^{(l)}$ ;
      split  $K_1^{(l)}$  into  $K_{11}^{(l)} = K_1^{(l)} \cap Succ(\mathcal{I}^{(j)} - \{I^{(j)}\})$ 
      and  $K_{12}^{(l)} = K_1^{(l)} - K_{11}^{(l)}$ ;
      if  $l \leq k - 1$  then
        let  $\mathcal{K}^{(l)} = \{K_{11}^{(l)}, K_{12}^{(l)}, K_2^{(l)}\} - \{\emptyset\}$ ;
        if  $|\mathcal{K}^{(l)}| \geq 2$  then
          if  $\exists \mathcal{J}^{(l)} \in Q$  s.t.  $K^{(l)} \in \mathcal{J}^{(l)}$  then
            replace  $K^{(l)}$  in  $\mathcal{J}^{(l)}$  with the inodes in  $\mathcal{K}^{(l)}$ ;
          else add  $\mathcal{K}^{(l)}$  to  $Q$ ;
  // merge phase begins
  for  $j = i + 2$  to  $k$  do
     $Q = \emptyset$ ;
    look for inode  $I^{(j)} \subset I^{(j-1)}[v]$  s.t.  $I^{(j)}$  has the same
    index parents in the  $A(j-1)$ -index as  $I^{(j)}[v]$ ;
    if such a inode  $I^{(j)}$  exists then
      merge  $I^{(j)}[v]$  and  $I^{(j)}$  with  $J^{(j)} = I^{(j)}[v] \cup I^{(j)}$ ;
      if  $j \leq k - 1$  then
        insert  $J^{(j)}$  into  $Q$ ;
  // iterate to merge others
  while  $Q \neq \emptyset$  do
    pick any  $I^{(l)} \in Q$  with the smallest  $l$ ;
    remove  $I^{(l)}$  from  $Q$ ;
    let  $\mathcal{I}^{(l+1)} = \{I^{(l+1)}(w) \mid w \in Succ(w'), w' \in I^{(l)}\}$ ;
    partition  $\mathcal{I}^{(l+1)}$  into equivalent classes according to their
    labels and index parents in the  $A(l)$ -index;
    foreach equivalent class  $\mathcal{J}^{(l+1)} \subseteq \mathcal{I}^{(l+1)}$  do
      if  $|\mathcal{J}^{(l+1)}| \geq 2$  then
        merge the inodes in  $\mathcal{J}^{(l+1)}$  into  $J^{(l+1)} = \bigcup \mathcal{J}^{(l+1)}$ ;
      if  $l \leq k - 2$  then
         $Q = Q - \mathcal{J}^{(l+1)}$ ;
        insert  $J^{(l+1)}$  into  $Q$ ;
end

```

Figure 7: Insert an edge into A(k)-index.

Efficacy. Since we are essentially using our 1-index update algorithm to maintain $\Phi^{(i)}(G)$ with respect to $\Phi^{(i-1)}(G)$ for all $i = 1, \dots, k$, we can show that our algorithm maintains a minimal set of A(i)-indexes in the following sense.

Definition 6. For any data graph G , the set of A(i)-indexes $\Phi^{(0)}(G), \Phi^{(1)}(G), \dots, \Phi^{(k)}(G)$ are *minimal* if for all $1 \leq i \leq k$, merging any two inodes of $\Phi^{(i)}(G)$ will make it unstable with respect to $\Phi^{(i-1)}(G)$.

LEMMA 5. *The split/merge algorithm always maintains a minimal set of A(i)-indexes.*

PROOF. Follow the same lines of reasoning in the proof of Lemma 3. \square

Since the set of A(i)-index is built in a hierarchical manner, which resembles the nature of acyclic 1-indexes, we have the following result for the A(k)-index for *any* general data graph.

LEMMA 6. *For any data graph G, there is a unique minimal set of A(i)-indexes, each of which is also minimum.*

PROOF. Let $\Phi^{(0)}(G), \dots, \Phi^{(k)}(G)$ be the minimum A(i)-indexes of G , and $\Psi^{(0)}(G), \dots, \Psi^{(k)}(G)$ be any minimal set of A(i)-indexes. We will show by induction that $\Phi^{(i)}(G) = \Psi^{(i)}(G)$ for all i . The base case $\Phi^{(0)}(G) = \Psi^{(0)}(G)$ holds by definition. Now suppose $\Phi^{(i)}(G) = \Psi^{(i)}(G)$, then merging any two inodes in $\Psi^{(i+1)}(G)$ will make it unstable with respect to $\Phi^{(i)}(G)$. By Lemma 2, $\Psi^{(i+1)}(G)$ is always a refinement of $\Phi^{(i+1)}(G)$. If $\Phi^{(i+1)}(G) \neq \Psi^{(i+1)}(G)$, we would find at least two inodes in $\Psi^{(i+1)}(G)$ that are contained in the same inode of $\Phi^{(i+1)}(G)$, merging these two inodes would not cause $\Psi^{(i+1)}(G)$ to be unstable with respect to $\Phi^{(i)}(G)$. So we have $\Phi^{(i+1)}(G) = \Psi^{(i+1)}(G)$. \square

Combining Lemma 5 and 6, we have:

THEOREM 2. *For any data graph G, the split/merge algorithm always maintains the minimum A(k)-index.*

Efficiency. As a first impression, maintaining all the A(0) to A(k)-indexes would take a lot of space and increase the update cost. Below we describe a structure called the *refinement tree*, which is designed to exploit the fact that the A(i+1)-index is always a refinement of the A(i)-index. With this tree (a forest in general) we can maintain the A(i)-index on top of the A(i+1)-index, instead of manipulating massive sets of dnodes directly. The refinement tree includes all the nodes in the A(0) to A(k)-indexes. Tree edges are built by linking each inode in the A(i)-index to the inodes in the A(i+1)-index that are contained in this inode (Figure 8). With this tree structure, there is no longer any need to store the extents of the inodes in all the A(i)-indexes for $0 \leq i \leq k-1$, as they can be fully recovered from the extents of A(k)-index nodes.

Let us now see how to use the refinement tree to implement the algorithm *insert_A(k)_index_edge*, or more precisely, the two basic operations split and merge. Merges are easy: If we merge two A(k)-index inodes, we merge their extents as we did for the 1-index. If we merge two A(i)-index inodes for $1 \leq i \leq k-1$, we simply merge them together

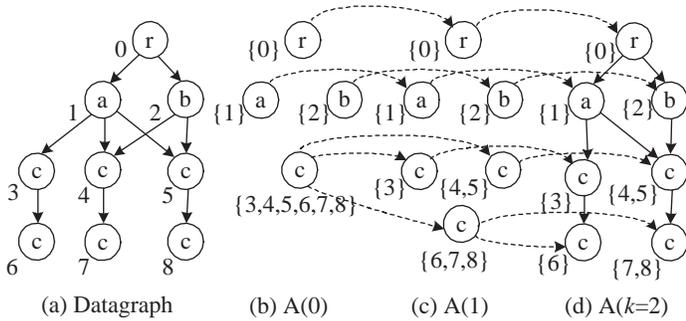


Figure 8: Refinement tree: tree edges are shown in dotted lines.

without any operation on their extents; all $A(i+1)$ -index inodes that were children of the two old nodes in the refinement tree now become the children of the new node.

Splits need more care. There are two kinds of splits: the “initial” splits at the beginning of the split phase and the “normal” splits using $Succ(I^{(j)})$ or $Succ(\mathcal{I}^{(j)} - \{I^{(j)}\})$ (ref. Figure 7). All initial splits together create one new inode containing only v for each of the $A(j)$ -indexes, $j = i+2, \dots, k$, so we just need to split $I^{(k)}[v]$ and then create a new node on each level j of the refinement tree, pointing only to the new tree node on level $j+1$.

For normal splits using, say $Succ(I^{(j)})$, we scan through $Succ(I^{(j)})$ and split all inodes whose extents intersect that of $Succ(I^{(j)})$ at the same time. For each dnode $w \in Succ(I^{(j)})$, there is exactly one inode in the $A(l)$ -index that contains w , for $l = j+1, \dots, k$. These inodes, denoted by $K^{(j+1)}, \dots, K^{(k)}$, form a path in the refinement tree. For the $A(k)$ -index inode $K^{(k)}$, we carry out the same procedure as with the 1-index: create an $A(k)$ -index inode $\hat{K}^{(k)}$ for w if necessary (it might have been created already while processing an earlier dnode that is k -bisimilar to w), and then move w from $K^{(k)}$ to $\hat{K}^{(k)}$. For $l = k-1, \dots, j+1$, we create an $A(l)$ -index inode $\hat{K}^{(l)}$ for w if necessary, and then make $\hat{K}^{(l+1)}$ a child of $\hat{K}^{(l)}$ in the refinement tree. After all dnodes in $Succ(I^{(j)})$ are scanned, we remove any empty inodes from the $A(k)$ -index, and then any $A(l)$ -index inodes with no children in the refinement tree, for $l = k, k-1, \dots, j+1$. After all pairs are processed, all splits with respect to $Succ(I^{(j)})$ are completed. The same procedure applies to $Succ(\mathcal{I}^{(j)} - \{I^{(j)}\})$. Note that in this way we only deal with the dnodes in the $A(k)$ -index; maintenance of the $A(i)$ -index only involves inodes in the $A(i+1)$ -index, and the cost of doing so decreases rapidly as i gets smaller.

Apart from the refinement tree edges, there are two types of iedges we need to maintain: the normal “intra-iedges” inside the $A(k)$ -index, used for query processing, and the “inter-iedges” in the refinement tree that connect inodes in $A(i)$ to their inode successors in $A(i+1)$, which are required in order for the maintenance algorithm to function efficiently. Both types of iedges can be maintained cheaply during the split/merge process. Optionally, one could also maintain the “intra-iedges” inside the $A(i)$ -indexes for $i = 1, \dots, k-1$, which will speed up the evaluation of path expressions of length less than k , but we will not explore this option further in this paper.

Although we store more information than the $A(k)$ -index alone, the extra storage overhead is low. We store each dnode only once (in the extent of an $A(k)$ -index inode), and we use only one hash table for the reverse mapping from the dnodes to the $A(k)$ -index inodes. For the $A(i)$ -indexes where $i < k$, we store only the refinement tree edges and the inter-iedges. Since the number of inodes in the $A(i)$ -index decreases rapidly as i gets smaller, this storage overhead is insignificant compared with the cost of storing extents and the dnode-to-inode mapping, which must be paid by a stand-alone $A(k)$ -index as well.

7. EXPERIMENTS

In this section, we present our experimental study comparing our algorithms with previous methods. All algorithms are implemented in Java. The machine used for experiments is a Dell PowerEdge 2600 with a 2.4GHz Xeon processor and 1GB of RAM, running Linux with JDK 1.4.2. Our machine has enough memory to store everything and no paging is needed during execution. We use the same performance metrics as previous works [8, 17], i.e., we measure efficacy in terms of the quality of the index as defined in Section 3, and efficiency in terms of the wall-clock running time.

We use both benchmark and real-life XML databases in our experiments. The XMark database is generated by the XMark generator from the XML Benchmark Project [2]. It is a highly cyclic and irregular database likely to stress the use of structural indexes. It is 11.7MB in size and consists of 167,865 dnodes and 198,612 dedges, among which 30,747 are IDREF edges. A sample of this database is shown in Figure 1. Cycles in this database are caused by a large number of person-auction edges. To see how our algorithms handle data graphs with cycles, we intentionally remove a portion of those edges to vary the *cyclicality*, which we define to be the fraction of such edges remaining. We name these data sets XMark(c) where c is the cyclicality; e.g., XMark(1) is the original XMark database, and XMark(0) contains no person-auction edges and thus no cycles, although they have the same number of dnodes.

The real-life dataset is extracted from the Internet Movie Database (IMDB) [1] in the following way: First we randomly choose a small subset of movies and all people (actors, directors, etc.) associated with these movies. We then extract all other movies associated with these people, and continue this process until the desired database size is reached. For each movie or person, we also extract a substantial amount of other information (e.g. title, year, genre). This dataset consists of 272,567 dnodes and 285,221 dedges, among which 12,654 are IDREF edges. Overall, it is also a highly cyclic and irregular dataset.

7.1 Experiments on the 1-Index

Edge insertions and deletions. For the 1-index edge insertions and deletions, we compare with the *propagate* algorithm [8]. In this set of experiments we apply a mixed sequence of edge insertions and deletions on both the XMark and IMDB data. For XMark, we select four datasets with cyclicities 1, 0.5, 0.2 and 0, to see how the algorithms perform, since as suggested by Theorem 1, the performance might be affected by cycles.

In order to generate edge insertions in a meaningful way, we first remove 20% of all the IDREF edges from the data

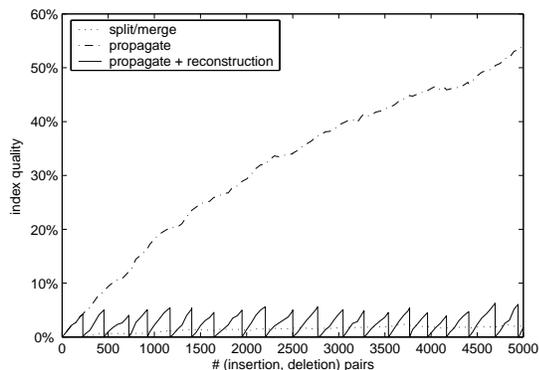


Figure 9: 1-index quality over mixed edge insertions and deletions on IMDB.

graph. These deleted edges then become a “pool” of possible insertions. Using the resulting data graph as the starting point, we perform one edge insertion followed by one edge deletion in each step: first a randomly selected edge is removed from the pool and inserted into the data graph, and then another randomly selected edge is deleted from the data graph and put back into the pool. For each dataset, 5000 pairs of edge insertions and deletions are performed.

Since the *propagate* algorithm does not have any guarantee on the quality of the updated index, the index gets progressively worse over time and it is necessary to reconstruct the index periodically. We used the “index reconstruction” idea of [8], i.e., run the construction algorithm on top of the index graph (treating it as a data graph), and then “blow up” each inode of the new index by replacing each inode of the old index with its extent of dnodes. Since we do not know how big the current minimum index is during the course of a sequence of updates, we use the following simple heuristic to trigger index reconstructions: remember the size of the index when it was last reconstructed, and then perform reconstruction whenever the current index is more than 5% larger than that. Since our split/merge algorithm does not guarantee the minimum 1-index on cyclic data graphs, either, we use the same heuristic to trigger reconstruction.

Results for IMDB are shown in Figure 9. Performance of *propagate* for the first 500 edge updates agrees with the previously reported results [8] very well: around 5% increase in index size. In fact the result reported in [8] was a little better than this, which can be explained by the fact that [8] only did edge insertions, while edge deletions are a little more difficult to handle because the minimum 1-index itself usually shrinks when edges are deleted. After that, we see that its index quality continues to degrade almost linearly with the number of edge updates performed. Thus, reconstruction is triggered once about every 500 updates. On the other hand, our split/merge algorithm maintains the index quality very well, never exceeding 3%. This experiment shows that the minimal 1-index maintained by our algorithm is in fact very close to minimum for this dataset.

Results for XMark are shown in Figure 10. An interesting fact is that on these datasets, our split/merge algorithm performs extremely well: its quality curves virtually remain zero (never exceeding 0.5%). The reason is that the IDREF edges in the XMark datasets are generated more uniformly, while in IMDB they tend to be clustered: related persons

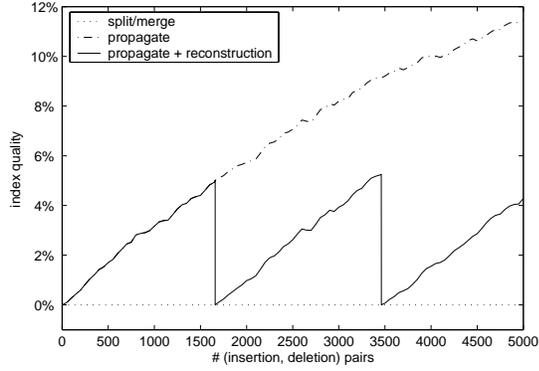
are likely to get involved in related movies, creating shorter cycles that make cases similar to Figure 4 more likely than in XMark. For *propagate*, we see similar trends for all datasets: its quality curves almost always grow linearly, although the rate varies a lot for different cyclicities: on XMark(1), the index quality is still better than 12% after 10000 edge updates, but on XMark(0) it gets worse very quickly. The reason is that XMark(1) is a highly irregular dataset; even the size of its minimum 1-index is more than 40% of its data graph size. For such a big index, there are very few possible merges during updates, so *propagate* algorithm performs relatively well. However, such large 1-indexes usually lead to bad query performance, and we usually turn to other smaller indexes, such as $A(k)$, for these cases. As the cyclicity decreases, the data graph also gets more regular, and the minimum 1-index shrinks. The *propagate* algorithm then has increasing difficulty in keeping the index fit, and has to perform more frequent reconstructions.

We also measured the average running times over the 10000 edge updates for each dataset. From Figure 11 we can see that the split/merge algorithm is more costly than the *propagate* algorithm, due to the extra merge phase, but it becomes much faster if we factor in the amortized reconstruction cost (total reconstruction cost divided by 10000). Notice that cyclicity does not seem to affect the performance of the split/merge algorithm, showing that cases like Figure 5 are not common. Finally, note the index is essentially unusable during the reconstruction, while our split/merge algorithm always responds quickly, thereby making the index more available for queries.

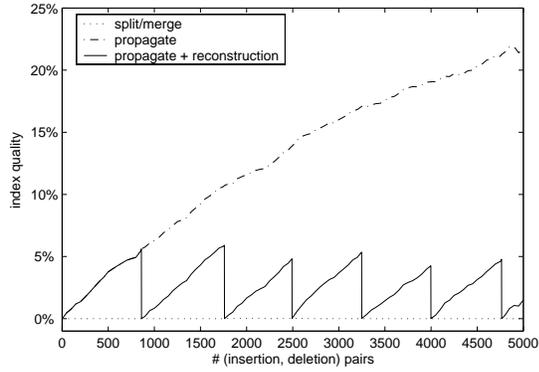
Subgraph additions. We also conduct experiments on subgraph additions with the XMark data. We extract subgraphs in the following manner. First we randomly select an “auction” dnode u , and then perform a traversal down starting from u to extract all descendants of u , which form a subgraph. We do not traverse IDREF edges because we want to avoid cycles, and also because the IDREF edges usually represent inter-object relationships that are not integral parts of the entity of interest. In this way we extract 500 subgraphs, with an average size of 50 dnodes. For each dataset, we first delete all these subgraphs, and then insert them one by one. We compare three alternatives: (1) our algorithm of Section 5.2, (2) same algorithm but using *propagate* instead of *insert_1_index_edge* to insert the edges, and (3) the index reconstruction algorithm of [8], which always maintains the minimum 1-index but is extremely costly. We obtain almost the same results again: Our algorithm keeps the quality of 1-index at 0% almost all the time, while the second alternative keeps increasing the index size and is very sensitive to the structure of the data graph (Figure 12). In terms of running cost, the first two alternatives are both very fast, about 20 msec for each subgraph; the third one is more than 100 times slower because of the costly reconstruction.

7.2 Experiments on the $A(k)$ -Index

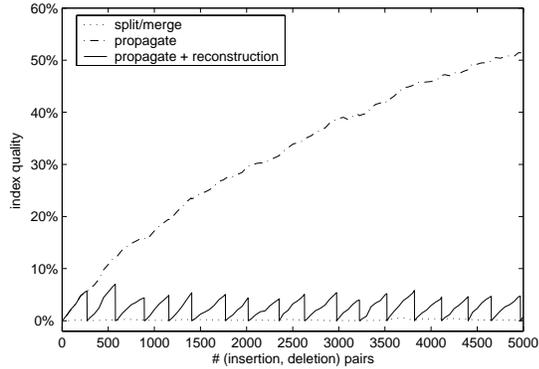
Since we have theoretical guarantee that our split/merge algorithm always maintains the minimum $A(k)$ -index, the experiments on the $A(k)$ -index are mainly aimed at efficiency issues, namely the running cost and addition storage overhead resulted from maintaining all the $A(i)$ -indexes for $0 \leq i \leq k$. In the experiments, we varied k from 2 to 5, covering the range of k 's that give the best performances as



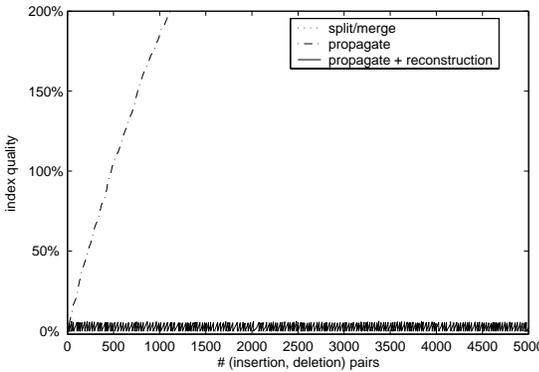
(a) XMark(1)



(b) XMark(0.5)



(c) XMark(0.2)



(d) XMark(0)

Figure 10: 1-index quality over mixed edge insertions and deletions on XMark.

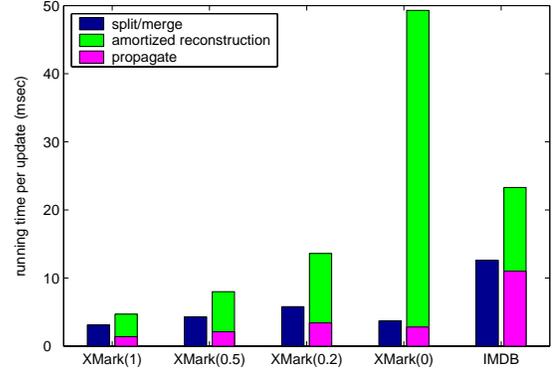


Figure 11: Running times of 1-index algorithms.

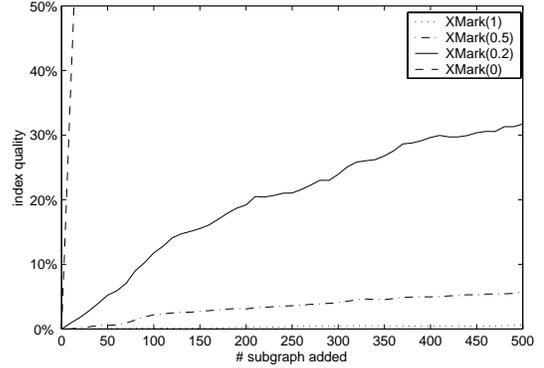


Figure 12: 1-Index quality during a sequence of subgraph additions with the *propagate* algorithm.

reported by [9]. The cyclicities of datasets are not tweaked here because the performance of our algorithms are not affected by cycles. We only present the experiments on edge insertions and deletions for the $A(k)$ -index in this paper.

We compare with the following simple algorithm, obtained by fixing a minor mistake in the one mentioned at the end of [17]. After an edge (u, v) is inserted or deleted, we do a breadth-first search to find all the potentially affected d nodes in the data graph. These d nodes are descendants of v up to a maximum depth of $k-1$. The corresponding inodes containing these d nodes are possibly unstable and need to be partitioned into new inodes according to k -bisimilarity. Since the $A(k)$ -index does not retain enough information to compute k -bisimilarity, we have to go back to the data graph and compute by definition. Notice that the cost of this simple algorithm is exponential in k . Since this algorithm does not provide any guarantee on the index quality, we also consider the option of periodic index reconstructions in the experiments, like what we did with the 1-index.

For the experiments, we only perform 1000 pairs of insertions and deletions since it is already enough to see a clear trend. The simple algorithm, as expected, blows up the index size rapidly without reconstructions, especially for small k 's. The result on the XMark database are shown in Figure 13. The result on the IMDB is similar and omitted. When the reconstruction threshold is set to 5%, this simple algorithm triggers frequent reconstructions, as shown in Table 1.

Running times of our split/merge algorithm and this sim-

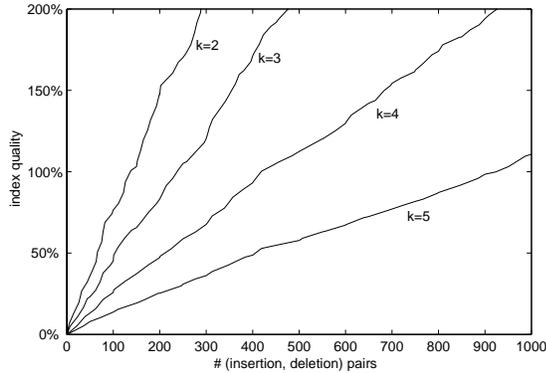


Figure 13: $A(k)$ -index quality of the simple algorithm.

ple algorithm are compared in Table 2, from which we can see that our algorithm is superior in all experiments. When k is large, where the simple algorithm does relatively fine in terms of index quality, its efficiency degrades dramatically. On the other hand, our algorithm is not affected much by k , proving the effectiveness of our implementation techniques in Section 6.

Finally, we also measure the addition storage required by the split/merge algorithm for storing the refinement tree and the inter-edges, which are not part of a stand-alone $A(k)$ -index. In our implementation, each dnode, inode, or pointer takes 4 bytes. Table 3 summarizes the estimated space consumption for newly constructed $A(i)$ -indexes for different configurations. The extra space used in our algorithm is always below 15% of the space occupied by a stand-alone $A(k)$ -index, which is dominated by the storage of inode extents; we avoid storing extents in $A(i)$ -indexes where $i < k$, as discussed in Section 6. We have also observed that this ratio does not change much during updates, since our algorithm always maintains the minimum set of $A(i)$ -indexes.

8. CONCLUSION

In this paper, we present new incremental maintenance algorithms for the 1-index and the $A(k)$ -index for graph-structured databases. We have demonstrated the efficacy (in preserving the minimality of indexes) and efficiency (in update time) of these algorithms both in theory and in experiments. Although we have only looked at these two structural indexes, we believe that these ideas can also be extended to handle other structural indexes as well.

Acknowledgment. We would like to thank Lars Arge, Raghav Kaushik and Sriram Padmanabhan for carefully reading the early drafts of this paper.

9. REFERENCES

- [1] The Internet Movie Database. <http://www.imdb.com>.
- [2] The XML benchmark project, <http://www.xml-benchmark.org>.
- [3] A. Aboulnaga, A. R. Alameldeen, and J. F. Naughton. Estimating the selectivity of XML path expressions for internet scale applications. In *VLDB*, 2001.
- [4] J. Clark and S. DeRose. XML path language XPath <http://www.w3.org/TR/xpath>. 1999.

Dataset	A(2)	A(3)	A(4)	A(5)
XMark	18.6	25.8	46.6	85.2
IMDB	32.2	69	126.4	142.2

Table 1: Average number of updates during two consecutive reconstructions for the simple algorithm over 2000 updates.

k	2	3	4	5
split/merge (XMark)	31	33	34	44
simple+reconstruction (XMark)	42	203	566	675
split/merge (IMDB)	112	115	127	153
simple+reconstruction (IMDB)	176	305	342	1030

Table 2: Average running times over 2000 updates (in msec) of different algorithms.

k	2	3	4	5
stand-alone $A(k)$ (XMark)	2023	2044	2112	2192
$A(0)$ to $A(k)$ (XMark)	2035	2081	2224	2479
Additional storage (%)	.6%	1.8%	5.3%	13%
stand-alone $A(k)$ (IMDB)	3292	3332	3378	3422
$A(0)$ to $A(k)$ (IMDB)	3312	3403	3576	3818
Additional storage (%)	.6%	2.1%	5.9%	11.6%

Table 3: Storage requirement of the split/merge algorithm (in KB).

- [5] M. F. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *ICDE*, 1998.
- [6] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, 1997.
- [7] H. He and J. Yang. Multiresolution indexing of XML for frequent queries. In *ICDE*, 2004.
- [8] R. Kaushik, P. Bohannon, J. F. Naughton, and P. Shenoy. Updates for structure indexes. In *VLDB*, 2002.
- [9] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *ICDE*, 2002.
- [10] J. McHugh and J. Widom. Query optimization for XML. In *The VLDB Journal*, pages 315–326, 1999.
- [11] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, 1999.
- [12] R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6), 1987.
- [13] D. Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science, 5th GI-Conf.*, LNCS 104, pages 167–183. 1981.
- [14] S. Park, Y. Choi, and H.-J. Kim. XML query processing using signature and dtd. In *Proc. of the 3rd Intl. Conf. EC-Web*, 2002.
- [15] S. Park and H.-J. Kim. A new query processing technique for XML based on signature. In *Proc. of the 7th Intl. Conf. on Database Systems for Advanced Applications*, 2001.
- [16] N. Polyzotis and M. Garofalakis. Statistical synopses for graph-structured data. In *SIGMOD*, 2002.
- [17] C. Qun, A. Lim, and K. W. Ong. D(k)-index: An adaptive structural summary for graph-structured data. In *SIGMOD*, 2003.