Multiresolution Indexing of XML for Frequent Queries*

Hao He Jun Yang
Department of Computer Science, Duke University, Durham, NC 27708, USA
{haohe,junyang}@cs.duke.edu

Abstract

XML and other types of semi-structured data are typically represented by a labeled directed graph. To speed up path expression queries over the graph, a variety of indexes, e.g., 1-index, A(k)-index, and D(k)-index, have been proposed. They usually work by partitioning the nodes in the data graph into equivalence classes and storing these equivalence classes as index nodes. A(k)-index introduces the concept of local bisimilarity for partitioning. It accurately supports path expressions of length up to some tunable constant k, allowing the trade-off between index size and query answering power. However, all index nodes in A(k)-index have the same local similarity value k, which cannot take advantage of the fact that a workload may contain path expressions of different lengths, or that different parts of the data graph may have different local similarity requirements.

To overcome these limitations, we propose M(k)- and M(k)-indexes. The basic M(k)-index is workload-aware: Like the previously proposed D(k)-index, it allows different index nodes to have different local similarity requirements, providing finer partitioning for parts of the data graph targeted by longer path expressions, and coarser partitioning for those targeted by shorter path expressions. Unlike D(k)-index, however, M(k)-index is never over-refined for irrelevant index or data nodes. However, the workload-aware feature of M(k)-index incurs overrefinement due to overqualified ancestors. To solve this problem, we further propose the M*(k)-index. An M*(k)-index consists of a collection of indexes whose nodes are organized in a partition hierarchy, allowing successively coarser partitioning information to co-exist with the finest partitioning information required. Experiments show that our indexes are superior to previously proposed indexes in terms of index size and query performance.*

* This work was supported by a National Science Foundation CAREER Award under grant IIS-0238386.

1. Introduction

XML has become a popular standard for exchanging and querying data over the Internet. An XML document consists of *tagged* (or labeled) *elements* (or nodes) that are nested hierarchically. A reference can be made from one element to another using an ID/IDREF pair. Therefore, in general, an XML document can be represented by a labeled directed-graph. XML is a *semi-structured* data model [1], which means that XML data might be irregular or incomplete. Thus, it is expensive to maintain all structural information of XML data.

Several query languages for XML and semi-structured data have been proposed [3, 4, 2]. *Path expressions* are the basic building blocks of XML queries. To speed up query processing, we can construct a structural index to summarize the structure of a data graph. Then, we can process path expressions using the index without referring to the original data graph, which may be much bigger than the index. A number of structural indexes for XML data have been proposed [8, 15, 10, 6, 11, 18]. Usually, a structural index is a graph defined using a specific equivalence relation on the nodes of the data graph. Each index node corresponds to an equivalence class of data nodes.

The *1-index* [15] is based on the notion of *bisimulation* [16]. If two nodes are *bisimilar*, they are reachable by the same set of label paths. Therefore, the 1-index can be used to evaluate any path expression accurately without accessing the data graph. However, the size of 1-index can be quite large for irregular XML data. Moreover, not all structures are interesting and most queries probably only involve short path expressions.

Based on the above observations, the *A(k)-index* [11] introduces the notion of *k-bisimilarity* to capture the local structures of a data graph. The A(k)-index can accurately support all path expressions of length up to k. For a path expression longer than k, the index may return some false positives, so they must be validated in the data graph. Taking advantage of local similarity, the A(k)-index can be substantially smaller than the 1-index. The parameter k controls the "resolution" of the entire A(k)-index; all index nodes have the same local similarity of k. If k is too small, the in-

dex cannot support long path expressions accurately. If k is too large, the index may become so large that evaluating any path expression over this index will be expensive. Furthermore, not all path expressions of length k are equally common. The A(k)-index lacks the ability to make certain parts of the index have higher resolution than others, so it cannot be optimized for common path expressions.

The *D(k)-index*, proposed recently in [5], allows different index nodes to have different local similarity requirements that can be tailored to support a given set of frequently used path expressions (or *FUP*'s for short). For parts of the data graph targeted by longer path expressions, a larger k can be used for finer partitioning. For parts targeted only by shorter path expressions, a smaller k can be used for coarser partitioning. The values of k can be adjusted dynamically to adapt to changing query workloads. The general approach of the D(k)-index is flexible and powerful, but the index design still has several limitations that need to be overcome. These limitations are outlined below, with detailed discussions to follow in Section 2.

- *Over-refinement of irrelevant index nodes.* The construction procedure of the D(k)-index forces all index nodes with the same label to have the same local similarity, which is unnecessary and restrictive. For example, a FUP //branch/dept/employee/name/lastname would cause all index nodes labeled lastname to acquire similarity values of at least 4, even if there exists such an index node, say, in fact for data nodes targeted by a rarely queried path expression //forum/support/message/from/name/lastname, which cannot be reached by the path expression that we are interested in. In general, over-refinement causes the size of the index to increase unnecessarily, with adverse effects on query performance.
- *Over-refinement for irrelevant data nodes.* The D(k)-index also proposes a *promoting procedure* that incrementally refines the index to support a given FUP. This procedure increases the local similarity of an index node if it can be reached by the given FUP in the index graph. This index node will be partitioned into smaller nodes, all with the same increased local similarity. However, the problem is that in general the index node to be refined also points to data nodes that are irrelevant to the given FUP. For example, we start with a D(k)-index containing an index node that corresponds to //name/lastname data nodes (including last names of employees as well as support forum posters), and we wish to refine the index to support the path expression //branch/dept/employee/name/lastname. The D(k)-index promoting procedure correctly refines the index for the last names of employees, but in doing so it also unnecessarily refines the index for the last names of support forum posters.
- *Over-refinement due to overqualified parents.* To increase the local similarity of an index node to k, the D(k)-index promoting procedure uses the information about the node's parents in the index graph. If any parent is "overqualified," i.e., its local similarity is greater than $k-1$, the algorithm will over-refine the index node, creating more partitions than necessary. In general, there is no guarantee that the local similarity is less than k for all parents.
- *Single resolution per node.* Once the D(k)-index has been refined to support a long path expression, e.g., //branch/dept/employee/name/lastname, the index nodes corresponding to the targeted data nodes will have a large local similarity value. Such a fine partitioning can potentially result in many index nodes. Now, a shorter path expression targeting the same data nodes (among others), e.g., //name/lastname, will be more expensive to evaluate, because it has to examine potentially many more index nodes. In general, if we insist that each data node can only be indexed at particular resolution, we will inevitably run into this problem for workloads containing both short and long path expressions targeting the same data nodes.

To overcome the first two of the above limitations, we propose the *M(k)-index* (for "mixed-k"). Like the D(k)-index, the M(k)-index uses the k-bisimilarity equivalence relation but allows different k values for different nodes; it is also incrementally refined to support new FUP's extracted from the query workload. Unlike the D(k)-index, however, M(k)-index is never over-refined for irrelevant index or data nodes. Thus, the M(k)-index has a smaller size without sacrificing support for any FUP's. Unfortunately, like the previously proposed indexes, the M(k)-index still can be over-refined due to overqualified parents, and does not work well for workloads that require multiple index resolutions per node.

To overcome the last two limitations, we further introduce the *M*(k)-index*, which consists of a collection of M(k)-indexes whose nodes are organized in a partition hierarchy, allowing successively coarser partitioning information to co-exist with the finest partitioning information required. The M*(k)-index maintains k-bisimilarity information for all k up to some desired maximum, which can be different across nodes and adjusted dynamically according to the query workload. This feature allows the M*(k)-index to avoid over-refinement due to overqualified parents and support both short and long path expression queries over the same data nodes at the same time.

An understandable concern about the M*(k)-index is its size, since this index essentially trades off space for efficiency of queries and quality of refinement operations. Our experiments on XMark and NASA datasets indicate that over-refinement due to overqualified parents is a significant

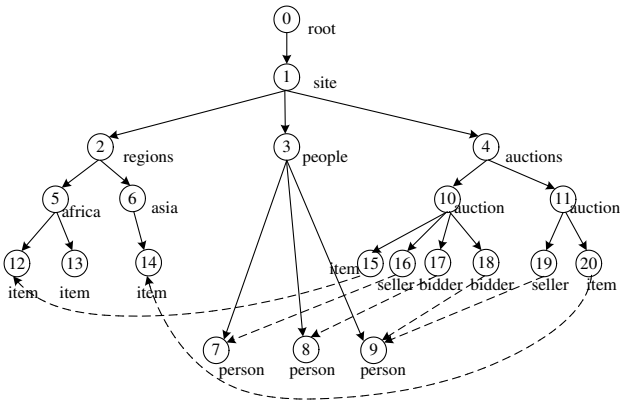


Figure 1. Example graph-structured data.

cause for index size inflation, so by avoiding this type of over-refinement, the M*(k)-index actually has size comparable to (and in many cases much smaller than) that of the M(k)-index.

The rest of this paper is organized as follows. Section 2 covers background and related work, focusing on discussion and critique of the three seminal papers on bisimilarity-based XML indexing [15, 11, 5]. We introduce the M(k)-index in Section 3 and the M*(k)-index in Section 4. Section 5 presents the performance results of experiments for M(k)- and M*(k)-indexes, in comparison with A(k)- and D(k)-indexes. Finally, Section 6 summarizes the paper and discusses future work.

2. Background and Related Work

An XML document is generally presented by a labeled directed graph $G = (V_G, E_G, root_G, \Sigma_G)$. Each node in the vertex set V_G is uniquely identified by its oid and has a string-literal $label$ from the alphabet Σ_G. The root node is denoted $root_G$. There are two types of edges in the edge set E_G. The *regular* edges represent parent-child relationships between elements in the XML document. The *reference* edges represent reference relationships defined using ID/IDREF attributes. A small example data graph is shown in Figure 1. The dashed lines represent reference edges.

A *label path* is a sequence of labels $l_0 l_1 \cdots l_n$. A *node path* is a sequence of nodes, $v_0 v_1 \cdots v_n$, such that an edge exists from v_{i-1} to v_i, for $1 \leq i \leq n$. A node path $p_n (= v_0 v_1 \cdots v_n)$ is an *instance* of a label path $p_l (= l_0 l_1 \cdots l_n)$ if $label(v_i) = l_i$ for each i. There are usually multiple node paths in G that are instances of a given label path p_l. The set of end nodes (v_n) of these node paths is called the *target set* of p_l. In this paper, we define $length(p_l)$, the length of a path $p_l = l_0 l_1 \cdots l_n$, to be n. Several query languages [3, 4, 2] for XML and semi-structured data define the notion of *path expressions*. For example, in XPath [3] syntax, the path expression /site/people/person returns the target set $\{7, 8, 9\}$ for the data in Figure 1; a slightly more complicated path expression involving a wildcard, /site/regions/*/item, returns the target set $\{12, 13, 14\}$. In this paper, we focus on *simple* path expressions, which are basically label paths.

Lore [13] supports path expressions using a structural summary called the *DataGuide* [8]. The basic idea is to build a summary, or an index graph, from the data graph, which preserves all label paths in the data graph but has far fewer nodes and edges. Following this approach, a variety of structural indexes for data have been proposed [15, 10, 6, 11]. In general, an structural index for a data graph G is a labeled directed graph $I_G = (V_{I(G)}, E_{I(G)}, root_{I(G)}, \Sigma_G)$, defined using an equivalence relation on V_G (specific to the index). Each index node in $V_{I(G)}$ represents a set of data nodes equivalent under this relation. The $extent$ of an index node is defined as the set of data nodes associated with this index node. There is an index edge (u_i, v_i) in $E_{I(G)}$ if and only if a data edge (u_d, v_d) exists in G and $u_d \in u_i.extent$, $v_d \in v_i.extent$.

1-index The 1-index [15] is based on the notions of bisimulation [16] and bisimilarity defined below. Only bisimilar data nodes are grouped into the same 1-index node.

Definition 1 (Bisimulation) *A symmetric, binary relation $\approx$ on V_G is called a* bisimulation *if the following holds: For any two data nodes u and v, $u \approx v$ if and only if*

1. *u and v have the same label;*
2. *If u' is a parent of u, that is, (u', u) is an edge and u' is closer to the root, then there exists a parent v' of v such that $u' \approx v'$, and vice versa.*

Two data nodes u and v are *bisimilar* if there exists some bisimulation $\approx$ such that $u \approx v$. This relation implies that if two nodes are bisimilar, the sets of label paths into them are the same. However, the converse is not true: Even if the sets of label paths into two nodes are the same, these nodes may not be bisimilar. An example is shown in Figure 2. The two nodes labeled d have the same incoming label paths: r/a/c/d and r/b/c/d. However, these two nodes are not bisimilar because their parents (labeled c) are not.

A(k)-index The A(k)-index [11] introduces the notion of k-bisimilarity defined below, which captures the local structures of a data graph.

Definition 2 (k-bisimilarity $\approx^k$) *k-bisimilarity is defined inductively:*

1. *For any two nodes, u and v, $u \approx^0 v$ if and only if u and v have the same label.*

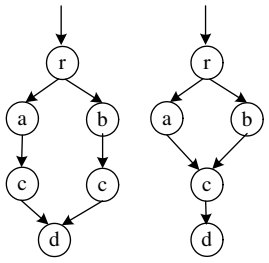


Figure 2. Two d nodes that are not bisimilar.

2. *$u \approx^k v$ iff $u \approx^{k-1} v$ and for every parent u' of u, there is a parent v' of v such that $u' \approx^{k-1} v'$, and vice versa.*

The A(k)-index uses k-bisimilarity as the equivalence relation to partition data nodes. The parameter k controls the resolution of the entire A(k)-index, providing a trade-off between index size and query answering power. The A(k)-index has the following properties:

1. If nodes u and v are k-bisimilar, then the sets of label paths of length up to k into them are the same.
2. The set of label paths of length up to k into an A(k)-index node is the set of label paths of length up to k into any data node in its extent.
3. The A(k)-index is precise for any simple path expression of length up to k.
4. The A(k)-index is safe, i.e., the result of evaluating any simple path expression on the index graph always contains the result of evaluating the same expression on the data graph.
5. The (k+1)-bisimulation is either equal to or a refinement of the k-bisimulation.

We note here that the definition of k-bisimilarity can be simplified by the following lemma. Besides the $(k-1)$-bisimilarity requirement on their parents, we only require the two nodes to have the same label (as opposed to being $(k-1)$-bisimilar). In Section 3, we will exploit this lemma to refine nodes in the M(k)-index, raising their local similarity to a large value in one step instead of increasing it one at a time.

Lemma 1 *$u \approx^k v$ if and only if $u \approx^0 v$ and for every parent u' of u, there is a parent v' of v such that $u' \approx^{k-1} v'$, and vice versa.*

Proof: $\Rightarrow$: If $u \approx^k v$, then by Defintion 2, $u \approx^i v$ for $i = k-1, \ldots, 0$, and the requirement on parents is also obviously met. $\Leftarrow$: Since for every parent u' of u, there is a parent v' of v such that $u' \approx^{k-1} v'$, which implies $u' \approx^0 v'$, we can conclude $u \approx^1 v$ from $u \approx^0 v$ according to Defintion 2. From $u \approx^1 v$, by noting that $u' \approx^1 v'$, we can further conclude that $u \approx^2 v$. Using the same argument repeatedly we can prove that $u \approx^i v$ for $i = 3, \ldots, k-1, k$. □

D(k)-index The D(k)-index [5] is an adaptive structural summary that supports different local similarity requirements on different index nodes. For each index node v, let $v.k$ denote the local similarity requirement on v, i.e., all data nodes in $v.extent$ must be $v.k$-bisimilar. The D(k)-index has the property that a parent's local similarity requirement cannot be lower than that of a child by more than 1. More precisely, for any two index nodes v and v', $v.k \geq v'.k - 1$ if there is an edge from v to v'.

We have mentioned earlier in Section 1 the construction and promoting procedures of the D(k)-index. The construction procedure is used to construct a D(k)-index from scratch to support a given set of FUP's. As we have seen in Section 1, this procedure over-refines irrelevant index nodes. Here, we focus more on the PROMOTE procedure, which refines an existing D(k)-index incrementally to support a given FUP.

```
PROMOTE(v, k_v, I_G)
1: if v.k ≥ k_v then
2:     Return I_G
3: for each parent u of v in I_G do
4:     I_G = PROMOTE(u, k_v − 1, I_G)
5: for each parent u of v in I_G do
6:     Split v.extent into v.extent ∩ Succ(u.extent) and v.extent − Succ(u.extent)
7: Return the final I_G
```

Here, v is the index node to be refined and k_v is the required new local similarity. $Succ(s)$ returns all data nodes that are children of some data nodes in a set s. In general, PROMOTE is first invoked on an index node v reachable by the given FUP. If v does not have the required local similarity to support the FUP, $v.extent$ may contain *irrelevant* data nodes that do not belong to the target set of the FUP. PROMOTE then refines v by recursively promoting all of its parents (lines 3–4) and then partitioning $v.extent$ to meet the new local similarity requirement (lines 5–6). In doing so, PROMOTE has effectively refined the index to support not only the given FUP, but also all incoming paths to v of length up to k_v. These paths may include those that do not lead to the target set of the given FUP but instead to the irrelevant data nodes in $v.extent$. Consequently, PROMOTE has over-refined the D(k)-index.

A concrete example of over-refinement is shown in Figure 3. The data graph is shown in part (a). The oid for each data node is shown on its left. Part (b) is the index graph before PROMOTE. The extent of each index node is shown on its left and the local similarity is shown on its right. At this point, all local similarities are 0, which means that all data nodes with the same label belong to the same index node.

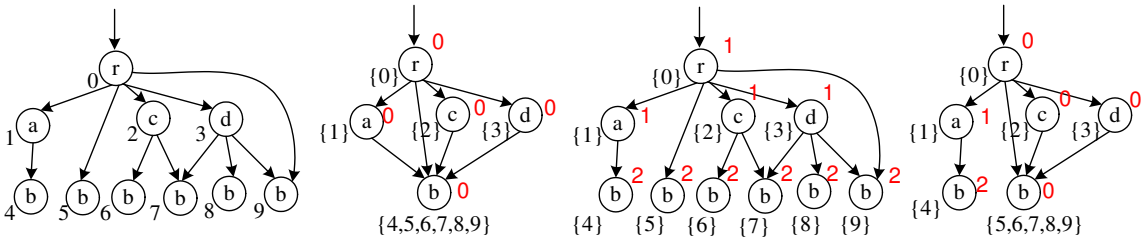


Figure 3. Comparison of D(k)- and M(k)-index refinement.

Suppose that r/a/b is the FUP to be supported. The refined D(k)-index after PROMOTE is shown in part (c). As we can see from the figure, the refined index becomes essentially a copy of the data graph, with irrelevant data nodes 5–9 belonging to different index nodes. In contrast, part (d) shows the M(k)-index after our refinement procedure, which is significantly more compact and groups all irrelevant data nodes into one index node.

Another source of over-refinement is overqualified parents, mentioned briefly in Section 1. A concrete example is presented in Figure 4. Again, the data graph is shown in part (a), and the index graph before PROMOTE is shown in part (b). Suppose we need to increase the local similarity of the index node c from 0 to 1. The refined D(k)-index after PROMOTE is shown in part (c), where the index node c has been split into two. However, their corresponding data nodes 4 and 5 are actually 1-bisimilar, and should have stayed together in one index node with local similarity of 1, as shown in part (d). The source of the problem is that PROMOTE uses the 2-bisimilarity information of the parents, but instead it should use the 0-bisimilarity information, which is not directly available. Our M*(k)-index will be able to avoid this problem since it keeps track of such information also.

Other indexes APEX [6] is another adaptive path index for XML tuned for supporting FUP's. Our work is similar to APEX in the workload-aware aspect. However, our indexes can capture significantly more structural information from the data. APEX maintains two structures, a graph and a hash tree. The graph represents a structural summary of the data. However, except for the FUP's with entries in the hash tree, APEX cannot directly answer other path expressions of length more than one. In some sense, APEX behaves more like an efficiently organized cache of answers to FUP's.

UD(k, l)-index [18] generalizes the A(k)-index by extending local bisimilarity to up-bisimulation and down-bisimulation, corresponding to upward and downward paths respectively. The index is especially efficient for branching

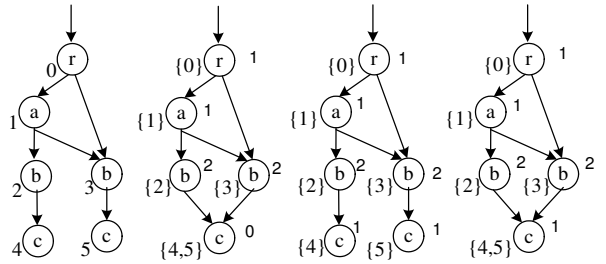


Figure 4. Over-refinement due to overqualified parents.

path expressions. However, it also inherits the static nature of the A(k)-index.

Many other XML indexing techniques have also been developed. In [7], paths in the data graph are viewed as strings and stored in a multilevel Patricia trie. The inverted index [19] and the numbering scheme [12] support efficient ancestor queries. However, they all focus mostly on tree-structured data.

3. The M(k)-Index

In this section we describe the M(k)-index, which achieves the same goal of adaptively and selectively supporting FUP's as the D(k)-index, but without the problems of over-refinement for irrelevant index and data nodes. Formally, the M(k)-index graph of a data graph G is a labeled directed graph $I_G = (V_{I(G)}, E_{I(G)}, root_{I(G)}, \Sigma_G)$. Each node in $V_{I(G)}$ has two attributes: $v.k$, the local similarity, and $v.extent$, the set of data nodes associated with v. The M(k)-index has the following three basic properties that are preserved by all index operations.

Property 1 All data nodes in $v.extent$ are $v.k$-bisimilar in G.

Property 2 $(v, v') \in E_{I(G)}$ if and only if $\exists o \in v.extent$ and $\exists o' \in v'.extent$, such that $(o, o') \in E_G$.

Property 3 For all parent v_p of v in $V_{I(G)}$, $v_p.k \geq v.k - 1$.

These three properties are identical to those of the D(k)-index, meaning that it is functionally equivalent to our M(k)-index. However, as we will see later in this section, our approach to maintaining these properties is quite different from the D(k)-index, and can potentially produce a much smaller index than the D(k)-index.

The following lemma follows from the above three basic properties of the M(k)-index.

Lemma 2 *For any index node v and any data node $o \in v.extent$, the set of label paths of length up to $v.k$ going into v in I_G is the same as the set of label paths of length up to $v.k$ going into o in G.*

Proof: According to Property 2 of the M(k)-index, any node path in G can be mapped to a node path in I_G (by mapping each edge along the path to a corresponding edge in I_G). Therefore, any label path going into o in G must also go into v in I_G. The converse remains to be proven. Since the data nodes in $v.extent$ are $v.k$-bisimilar, they have the same of set of incoming label paths of length up to $v.k$ (Section 2). Thus, it suffices to show that for any label path of length up to $k \leq v.k$ going into v in I_G, there exists some data node in $v.extent$ with this incoming path in G.

We use induction on the value of k. The claim is obviously true for $k = 0$, because there is only one label path of length 0 into v, which is the label of all data nodes in $v.extent$. Assume that the claim is true for $k = k' - 1$. Consider any label path $l_0 l_1 \cdots l_i$ going into v, where $i \leq k' \leq v.k$. Let u be the parent of v in the instance of $l_0 l_1 \cdots l_i$ going into v. There exist m and n, such that $m \in u.extent$, $n \in v.extent$, and (m, n) is an edge in G. According to Property 3 of the M(k)-index, $u.k \geq v.k - 1 \geq k' - 1$. By the induction hypothesis, since u has $l_0 l_1 \cdots l_{i-1}$ as an incoming path, where $i - 1 \leq k' - 1 \leq u.k$, m must also have $l_0 l_1 \cdots l_{i-1}$ as an incoming path in G. Because (m, n) is an edge in G, we know that n has $l_0 l_1 \cdots l_i$ as an incoming path in G. The proof is complete. □

From the basic properties of the M(k)-index and the lemma above, we can show that the M(k)-index has the following safety and precision properties for answering path expression queries:

- The M(k)-index is safe, i.e., it produces no false negatives. To be specific, the result of evaluating any path expression on I_G always contains the result of evaluating the same expression on G. This property follows from Properties 1 and 2 of the M(k)-index.

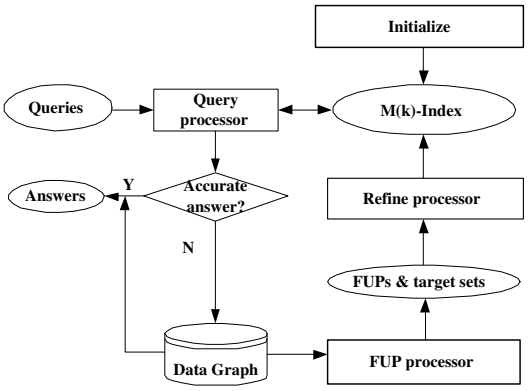


Figure 5. Overview of M(k)-index operations.

- The M(k)-index is precise for a label path expression of length k if each node v in its target set in I_G satisfies $v.k \geq k$. That is, the result contains no false positives. This property follows from Lemma 2.

We now give an overview of how the M(k)-index is constructed, queried, and refined dynamically at runtime. The process is outlined below and illustrated in Figure 5:

1. Initialize the index with $k = 0$ for all index nodes, resulting essentially in an A(0)-index. This index can only precisely answer path expressions of length 0, i.e., those with a single label.
2. Answer incoming queries using the index graph. If the answer is not guaranteed to be precise, validate it against the data graph.
3. Extract FUP's (frequently used path expressions) from queries.
4. Refine the index to support the FUP's.
5. Go to step 2.

Next, we present the details of the query algorithm in Section 3.1 and the refinement algorithm in Section 3.2.

3.1. Query Algorithm

To answer a simple path expression l using the M(k)-index, we take the following steps:

1. Find the target set of l in the index graph, i.e., the set of index nodes with l as an incoming path.
2. For each index node v in the target set, if $v.k \geq length(l)$, we return all of $v.extent$ to the user; otherwise, we must validate the nodes in $v.extent$ and only return to the user those that really do have l as an incoming path in the data graph.

The check in the second step reflects the precision property of the M(k)-index discussed earlier. As a result of validation, we know which data nodes in $v.extent$ are true answers to the query. This information about "relevant"

data nodes will be passed on to the refinement algorithm if the query is a FUP to be supported. As we will see in the next section, this information is vital in avoiding over-refinement.

3.2. Refinement Algorithm

The goal of refinement is to increase the index resolution for selected parts of the index, so that it would be able to answer a given FUP precisely. As discussed in the previous section, the target set of the FUP in the data graph has been acquired by the query algorithm before refinement. The single most important difference between our refinement algorithm and the previous algorithms is our use of this target set to avoid over-refinement.

The refinement algorithm, REFINE, is presented below. The input to REFINE includes l, the FUP to be supported; S, the target set of l in the index graph; and T, the target set of l in the data graph.

```
REFINE(l, S, T)
 1: for each v in S do
 2:     REFINENODE(v, length(l), v.extent ∩ T)
 3: while ∃v ∈ I_G such that v has l as an incoming path and v.k <
    length(l) do
 4:     PROMOTE'(v, length(l))
```

The algorithm attempts to promote the local similarity values to at least $length(l)$ for the index nodes in S. To maintain Property 3 of the M(k)-index, the ancestors of S nodes may need to be promoted too. These tasks are accomplished by a recursive procedure REFINENODE. To avoid over-refinement, we pass in the relevant data nodes in $v.extent$ to REFINENODE.

Procedure REFINENODE is shown below. Here, $Pred(s)$ returns all data nodes that are parents of some data nodes in a set s. To understand how REFINENODE works and how it differs from the D(k)-index, we can compare it side by side with the PROMOTE algorithm used by the D(k)-index discussed in Section 2. Lines 2–7 of REFINENODE correspond to lines 1–4 of PROMOTE; both recursively refine the parents. Lines 9–17 of REFINENODE correspond to lines 5–6 of PROMOTE; both split the index node according to the $Succ$ set of parents. REFINENODE only processes a parent if its extent contains some parents of the relevant data nodes, while PROMOTE blindly processes all parents recursively. This optimization translates into significant size reduction for the M(k)-index.

```
REFINENODE(v, k, relevantData)
 1: // Lines 2–7: recursively refine parent nodes:
 2: if v.k ≥ k then
 3:     Return
 4: for each parent u of v in I_G do
 5:     predData = Pred(relevantData) ∩ u.extent
 6:     if predData ≠ ∅ then
 7:         REFINENODE(u, k − 1, predData)
```

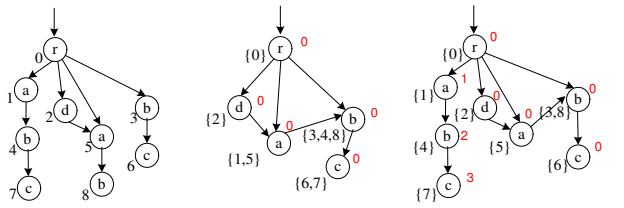


Figure 6. False positive created by RefineNode.

```
 8: // Lines 9–17: split v:
 9: k_old = v.k
10: V = {v}
11: for each parent u of v in I_G do
12:     if Pred(relevantData) ∩ u.extent ≠ ∅ then
13:         for each w in V do
14:             Replace w (in both V and I_G) with w_1 and w_2, where:
15:             w_1.extent = w.extent ∩ Succ(u),
16:             w_2.extent = w.extent − Succ(u), and
17:             w_1.k = w_2.k = k
18: // Lines 19-26: merge unnecessary splits
19: remainderExtent = ∅
20: for each w in V do
21:     if relevantData ∩ w.extent = ∅ then
22:         remainderExtent = remainderExtent ∪ w.extent
23:         Remove w from I_G
24: Add v_rest to I_G, where:
25: v_rest.k = k_old, and
26: v_rest.extent = remainderExtent
```

As a concrete example, consider again Figure 3, where r/a/b, with its target set of $\{4\}$, is the FUP to be supported. Among index node b's parents, only a needs to be considered since it is the only parent whose extent contains parents of the data nodes in the target set. It turns out that a already has the required local similarity, so we proceed to split b. Instead of forcing all partitions of b to have local similarity of 2, we only require it for those partitions that contain data nodes in the target set. All irrelevant data nodes are grouped into one index node, which retains the old local similarity value. The refined M(k)-index is shown in part (d), compared with the result of refining the D(k)-index in (c).

After all REFINENODE calls have completed, we would like every instance of l in the index graph to lead to an index node that contains no false positives. Indeed, at this point, we can guarantee that all data nodes in T, the target set of l, belong to index nodes with local similarity of at least $length(l)$. However, there is still a very small possibility that, after refinement, a new instance of l is created in the index graph that does not lead to any data nodes in T. A concrete example is shown in Figure 6. Suppose we refine the index to support the path expression r/a/b/c. After all REFINENODE calls have completed, we have the index graph shown on the right. Note that a second instance of r/a/b/c has been created, which leads only to false positives.

The last loop of REFINE guards against this possibility.

As long as there still exists an instance of l that leads to false positives, we use PROMOTE', a modified version of PROMOTE, to "break" this instance by splitting the index nodes along the instance. Since the purpose of PROMOTE' is not refinement per se, but to break a false instance of l by refinement, we do not need to carry out the promoting algorithm to completion. Instead, we simply add a check between lines 6 and 7 of PROMOTE: If no more false instances of l are found, just do a long jump out of the procedure (and terminate all recursions).

4. The M*(k)-Index

Although the M(k)-index avoids over-refinement of irrelevant index and data nodes, it still has two remaining problems: over-refinement due to overqualified parents, and expensive short path expression queries. These problems expose a key deficiency of the M(k)-index, which is that each index node can have only one single local similarity value. Once the local similarity of an index node increases from k to $k' > k$, it is difficult to tell which k-bisimilarity partition this node belongs to.

To solve the above problems, we propose the M*(k)-index. Conceptually, the M*(k)-index consists of a sequence of component indexes $I_0, I_1, \ldots, I_k$ with different resolutions. Each component can be regarded as an M(k)-index that supports the FUP's as much as possible, abiding by the restriction that the maximum local similarity in component I_i is i. Hence, I_0 maintains the coarsest (0-bisimilarity) partitioning information and is able to process single-label path expressions efficiently, whereas I_k maintains the finest (up to k-bisimilarity) partitioning information and is able to answer all FUP's accurately. Each index node in component I_i is possibly partitioned by the next component I_{i+1} further into a set of index nodes, with a local similarity value one greater. Formally, we say that an index node v in I_i is the *supernode* of an index node v' in I_{i+1} (and v' is a *subnode* of v) if $v.extent \supseteq v'.extent$. The M*($k$)-index uses special links to connect a supernode with its subnodes, thereby forming a partitioning hierarchy across the components. A concrete example of an M*(k)-index is illustrated in Figure 7, where the dashed lines represent the special links across components. The properties of the M*(k)-index are summarized below:

Property 1 Each component index I_i has all properties of the M(k)-index.

Property 2 The maximum local similarity value for the index nodes in I_i is i.

Property 3 Component I_{i+1} is a refinement of component I_i. More precisely, the extent of every index node in I_i is the disjoint union of the extents of its subnodes in I_{i+1}, and every index node in I_{i+1} has exactly one supernode in I_i.

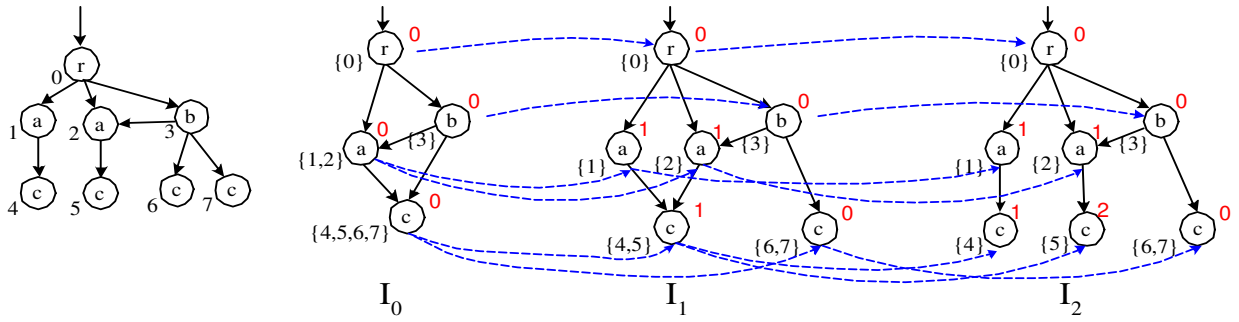
Property 4 If v is the supernode of v', then $v.k \leq v'.k \leq v.k + 1$. That is, the local similarity value of an index node cannot increase by more than one from one component index (I_i) to the next (I_{i+1}). This property ensures that the M*(k)-index maintains partitioning information for all resolutions from 0 up to the finest local similarity value required. (Technically, this property follows from Properties 2 and 5.)

Property 5 If v is an index node in I_i and $v.k < i$, then for any subnode u of v, $u.k = v.k$. That is, once the local similarity stops "growing" from one component index to the next, it stays the same in all subsequent component indexes.

In the example of Figure 7, the index nodes labeled r and b keep the same local similarity value of 0 across all index components. The index node labeled a in I_0 is partitioned into two nodes in I_1 with local similarity 1, which stays the same in I_2. The index node labeled c in I_0 is partitioned into two nodes in I_1 with local similarity values of 0 and 1, respectively; the node with local similarity 0 cannot be refined further in I_2 (Property 5), while the node with local similarity 1 is further partitioned into two nodes in I_2.

The M*(k)-index inherits the dynamic and adaptive nature of the M(k)-index. Component indexes are created and refined only when it is necessary to support additional FUP's. The unique feature of the M*(k)-index is that it also maintains successively coarser partitioning information in addition to the finest partitioning information required. This feature enables a much more efficient query algorithm and a refinement algorithm that avoids over-refinement due to overqualified parents. These two algorithms are described in detail in the remainder of this section.

It may appear that the M*(k)-index needs more space than the M(k)-index in order to maintain partitioning information at multiple resolutions. However, the implementation of the M*(k)-index can be made much more space-efficient than its logical representation by exploiting the fact that many index nodes and edges remain unchanged from one component index to the next. A supernode in I_i does not need to be duplicated in I_{i+1} if this node has only one subnode. An edge in I_i does not need to be duplicated in I_{i+1} if the two nodes connected by this edge each have only one subnode. For simplicity of presentation, the algorithms discussed in this section still assume the less efficient logical representation illustrated in Figure 7. In practice, our experiments reveal that the M*(k)-index is actually smaller than the M(k)-index in most cases because the M*(k)-index completely avoids over-refinement due to overqualified parents. The performance advantage of the M*(k)-index will be quantified in Section 5.



(a) Datagraph

(b) The M*(k) (supporting //b/a/c)

Figure 7. An example of the M*(k)-index.

4.1. Query Algorithm

The coarser partitioning information available in the M*(k)-index obviously improves the performance of short path expression queries. For example, in Figure 7, evaluation of the path expression //a/c can be done in component index I_1 without accessing the larger component index I_2. In contrast, for the M(k)-index, evaluation of any path expression must use the finest (and the only) partitioning information available.

Evaluation of long path expressions also benefits from the M*(k)-index, because long path expressions can be evaluated in steps starting with short path expressions. For a very simple example, to evaluate the path expression //b/a/c using the M*(k)-index in Figure 7, we can evaluate //b in I_0 first. The target set contains the only index node labeled b, and we follow the dashed link to find its subnode in I_1. Starting from this node, we continue to evaluate the path expression //b/a in I_1, obtaining an index node labeled a (the one on the right), which leads us to the subnode in I_2. Starting from that subnode, we can finish evaluating //b/a/c in I_2. This toy example does not really show the performance advantage of this evaluation strategy. However, consider a larger M*(k)-index where b nodes are partitioned into many index nodes in the finest component index. In this case, evaluating //b in I_0 would be much more efficient in comparison.

In general, the rich structure of the M*(k)-index supports many strategies for evaluating path expressions. We briefly discuss three strategies below. The decision of which strategy to use is an interesting query optimization problem, but it would be beyond the scope of this paper.

Naive evaluation Suppose the length of the path expression is l. Simply go to component index I_l and process the path expression using the query algorithm of the M(k)-index.

Top-down evaluation The top-down strategy processes a path expression by evaluating its prefixes in increasing order of length. A prefix path expression of length l can be evaluated in I_l, starting with the subnodes of the results of evaluating the prefix of length $l-1$. The M*(k)-index allows us to evaluate each prefix using the coarsest (thus smallest) component index possible, without using the finest (thus largest) component index.

An example of using this strategy to evaluate //b/a/c in Figure 7 was given earlier in this section. Here we present the detailed algorithm below. For simplicity, this algorithm performs validation at the very end. In practice, it would be more efficient to validate after evaluating each prefix.

QUERYTOPDOWN($l_0 l_1 \cdots l_j$)
1: $Q = \{v \in I_0 \mid label(v) = l_0\}$
2: **for** $i = 1$ to j **do**
3: $S = \{u \in I_i \mid supernode(u) \in Q\}$
4: $Q = \{v \in I_i \mid label(v) = l_i \wedge (\exists u \in S : (u, v)$ is an edge in $I_i)\}$
5: $A = \varnothing$
6: **for** each $v \in Q$ **do**
7: **if** $v.k = j$ **then**
8: $A = A \cup v.extent$
9: **else**
10: Validate the data nodes in $v.extent$
11: Add those that pass the validation to A
12: Return A

Subpath pre-filtering This strategy exploits the fact that a subsequence of the labels in a path expression may be highly selective. Instead of evaluating the original path expression of length l in the component index I_l, we evaluate a subpath of length $l' < l$ in the coarser and smaller component index $I_{l'}$. From the result set, we follow the cross-component links to find the corresponding index nodes in I_l. Then, we evaluate the rest of the original path expression in I_l starting from these nodes.

For example, to evaluate the path expression

//branch/dept/employee/name/lastname, we can start by evaluating a short subpath employee/name in I_1 and find the corresponding name nodes in I_4. This pre-filtering step can potentially eliminate many name nodes in I_4 from further consideration. Furthermore, it is likely that most of the name nodes that passed the filter indeed lead to the target set of //branch/dept/employee/name/lastname. The remaining task is to process branch/dept/employee/name and name/lastname in I_4 starting from these names nodes, and validate the final answer if necessary. This strategy might outperform the top-down strategy if there are many possible branches coming out from branch, dept, and employee nodes, but there are few branches going into them.

Other approaches In addition to top-down evaluation, bottom-up and hybrid (combining top-down and bottom-up) approaches have also been proposed in the XML query processing literature [14]. With the M*(k)-index, the idea would be to evaluate progressively longer suffixes of a path expression in progressively finer component indexes. Specifically, we can find the nodes in I_l with an outgoing path of length l, and then look for the parents of their subnodes in I_{l+1} with an outgoing path of length $l+1$. Unfortunately, indexes based on k-bisimilarity only guarantee that the data nodes belonging to the same index node have the same set of *incoming* paths of length up to k; there is no guarantee on the *outgoing* paths. Therefore, in the M*(k)-index, a subnode may have fewer outgoing paths than its supernode even if the supernode has a high enough local similarity value. That means whenever we move to a finer component index, we need to check downwards to ensure that the suffix path still exists. This overhead makes bottom-up and hybrid approaches less efficient than the top-down approach, which does not need to check upwards for the existence of the prefix path. Note that if we incorporate the feature of the recently proposed UD(k, l)-index [18], the M*(k)-index would also be able to efficiently support the bottom-up or hybrid evaluation approaches.

4.2. Refinement Algorithm

The M*(k)-index is initialized with a single component index I_0, which is identical to the A(0)-index (or a newly initialized M(k)-index). To refine the M*(k)-index, we use the REFINE* procedure presented below. The input to REFINE* includes l, the FUP to be supported; S, the target set of l in the finest component index; and T, the target set of l in the data graph. To support a FUP of length k, we need at least $k+1$ component indexes (from I_0 to I_k). If there are fewer, new component indexes will be created by copying the last existing component index. The remainder of REFINE* has the same outline as REFINE discussed in Section 3.2.

```
REFINE*(l, S, T)
 1: for i = 1 to length(l) do
 2:    if I_i does not exist then
 3:       Construct a new component I_i by copying I_{i-1}
 4: S = S nodes (or copies thereof) in I_{length(l)}
 5: for each v in S do
 6:    REFINENODE*(v, k, v.extent ∩ T)
 7: while ∃v ∈ I_{length(l)} such that v has l as an incoming path and
    v.k < length(l) do
 8:    PROMOTE*(v, length(l))
```

Procedure REFINENODE*, presented below, is called by REFINE* to increase the local similarity of an index node $v \in I_k$ to the desired value k. To avoid over-refinement, we also pass in the relevant data nodes in $v.extent$.

```
REFINENODE*(v, k, relevantData)
 1: // Lines 2–7: recursively refine parent nodes:
 2: if v.k ≥ k then
 3:    Return
 4: for each parent u of supernode(v) in I_{k-1} do
 5:    predData = Pred(relevantData) ∩ u.extent
 6:    if predData ≠ ∅ then
 7:       REFINENODE*(u, k − 1, predData)
 8: // Lines 9–13: refine v and its ancestor supernodes:
 9: i_start = min{i | supernode*(v, I_i).k < i}
10: for i = I_start to k do
11:    p = supernode*(v, I_i)
12:    SPLITNODE*(p, i, p.extent ∩ relevantData)
13:    Propagate changes made to I_i to all subsequent component indexes
```

The structure of REFINENODE* is similar to REFINENODE, its counterpart for the M(k)-index. Lines 2–7 correspond to those of REFINENODE, serving the same purpose of recursively refining parent index nodes. The rest of the procedure is less similar to REFINENODE, because the M*(k)-index needs to maintain partitioning information at all resolutions. The "ancestor supernode" of v in I_i, denoted by $supernode^*(v, I_i)$, is computed by following the cross-component links from v to I_i. Before refining v in I_k, REFINENODE* first refines v's ancestor supernodes in all applicable component indexes, starting from the first component index in which the local similarity value of v's ancestor supernode is lower than the maximum permissible value. Once we finish refining a component index I_i, we propagate all changes to all subsequent component indexes.

Note that it is necessary to propagate the changes to subsequent component indexes immediately after each node is refined (Line 13). If not, Properties 3 and 4 might be violated in the following case. Suppose there are two irrelevant data nodes d and d', both of which are not i-bisimilar to any relevant data node. However, d is $(i-1)$-bisimilar to some relevant data node while d' is not. In component index I_{i-1}, d should belong to the same index node as the $(i-1)$-bisimilar relevant data node, whereas d' should belong to a different index node. On the other hand, if we re-

fine I_i independently without immediately propagating the changes from I_{i-1}, we would group d and d' into one index node together with all other irrelevant data nodes that are not i-bisimilar to any relevant data node. That means I_i is not a refinement of I_{i-1}. Furthermore, the index node containing d in I_i may have a lower local similarity value than the index node containing d in I_{i-1}. Therefore, we must propagate the changes on I_{i-1} to I_i before refining I_i.

The actual partitioning of p (v's ancestor supernode) in I_i is handled by procedure SPLITNODE* detailed below, which corresponds to Lines 8–26 of REFINENODE. As with the M(k)-index, SPLITNODE* uses the information about relevant data nodes to avoid over-refinement. A seemingly minor but very important difference is that SPLITNODE* uses the parents of $supernode(v)$ in I_{k-1} to split v, instead of using v's parents directly. From Property 2 of the M*(k)-index, we know that the local similarity values of the parents in I_{k-1} cannot be more than $k-1$. Furthermore, REFINENODE* refines these parents before calling SPLITNODE*, so their local similarity values must be exactly $k-1$, meaning that they are "perfectly qualified" for splitting v. In contrast, the M(k)-index and the D(k)-index might over-split v because v's parents in these indexes might have local similarity values higher than $k-1$, i.e., they are overqualified.

SPLITNODE*($v, k, relevantData$)
1: // Split v:
2: $k_{old} = v.k$
3: $V = \{v\}$
4: **for** each parent u of $supernode(v)$ in I_{k-1} **do**
5: **if** $Pred(relevantData) \cap u.extent \neq \varnothing$ **then**
6: **for** each w in V **do**
7: Replace w (in both V and I_k) with w_1 and w_2, where:
8: $w_1.extent = w.extent \cap Succ(u)$,
9: $w_2.extent = w.extend - Succ(u)$, and
10: $w_1.k = w_2.k = k$
11: // Merge unnecessary splits:
12: $remainderExtent = \varnothing$
13: **for** each w in V **do**
14: **if** $relevantData \cap w.extent = \varnothing$ **then**
15: $remainderExtent = remainderExtent \cup w.extent$
16: Remove w from I_k
17: Add v_{rest} to I_k, where:
18: $v_{rest}.k = k_{old}$, and
19: $v_{rest}.extent = remainderExtent$

Finally, procedure PROMOTE* is used by REFINE* to break false positives that might have been introduced during refinement, analogous to procedure PROMOTE' discussed in Section 3.2. PROMOTE* is basically the same as REFINENODE*, except it does not take relevant data as input and therefore promotes all data nodes. It also makes a long jump out of itself as in PROMOTE' once all false positives are removed.

5. Experiments

The experiments in this section are aimed at comparing query performance and space consumption for M(k)- and M*(k)-indexes as well as previously proposed A(k)- and D(k)-indexes. We have implemented all four indexes in Java as main-memory data structures.

Datasets We use two XML datasets in our experiments: XMark and NASA. The XMark generator from the ongoing XML Benchmark Project [17] generates synthetic data about the activities of an auction Web site. The synthetic document we use is about 11MB in size and contains about $120,000$ nodes. The NASA dataset contains synthetic data generated by the IBM XML generator for a real DTD from NASA [9]. The synthetic document we use is about 11MB in size and contains about $90,000$ nodes. The NASA DTD is deeper, broader, has a more irregular structure, and contains more references than the XMark DTD.

The same two data generators were used in [5] to evaluate the D(k)-index. In [5], more than half of the references were removed from the NASA dataset in order to keep the index size manageable. We do not modify the generated datasets in any way for our experiments.

Cost metrics To measure query performance, we adopt the same main-memory cost metric similar to those used for evaluating the A(k)-index [11] and the D(k)-index [5]. The cost of a query consists of two parts: (1) the cost of evaluating the query on the index graph, and (2) the cost of validating the answer on data graph (to remove false positives when necessary). We measure the first part by the number of index nodes visited during query evaluation, and the second part by the number of data nodes visited during validation. Note that we do not count the data nodes in the extents of index nodes in the target set, unless they are visited in the data graph during validation.

One measure of the index size commonly used in previous work is the total number of index nodes. For the M*(k)-index, we count the total number of nodes across all component indexes. However, we do not count duplicate nodes such as those labeled r and b in I_1 and I_2 in Figure 7, who are the only subnode of their respective supernodes. These duplicate nodes exist only in the logical representation of the M*(k)-index and do not need to be stored by an implementation.

A second measure of the index size is the number of edges in the index graph. For the M*(k)-index, we count the total number of edges in all component indexes as well as cross-component links. Duplicates edges that connect duplicate nodes are not counted because they do not need to be stored.

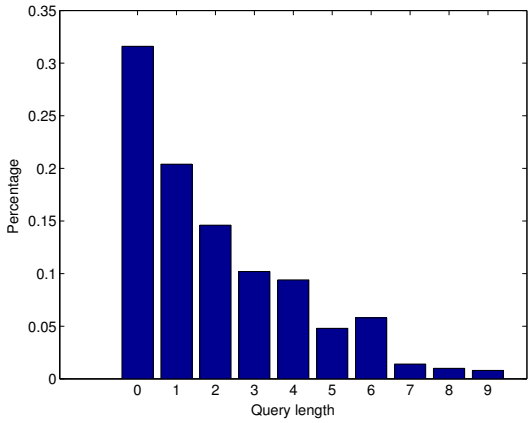


Figure 8. Query distribution on NASA dataset (max path length: 9)

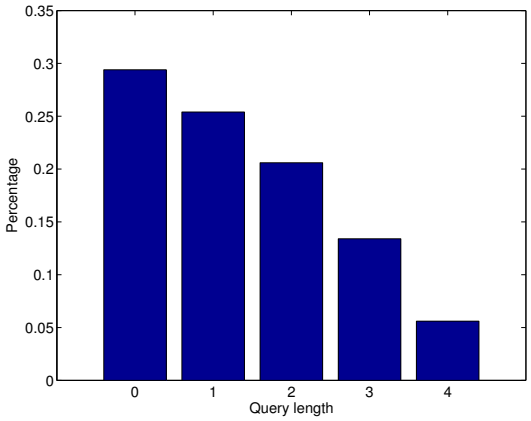


Figure 9. Query distribution on NASA dataset (max path length: 4)

Query workload We use a synthetic query workload generated in a similar fashion as in previous work. First, we generate all possible label paths of length up to 9 in the data graph. The length limit prevents paths containing infinite loops from being generated. Next, to generate a path expression query, we select a label path at random and extract a subsequence with random start position and length. The self-or-descendent axis ($//$) is placed in front of the subsequence to form a path expression query. Given a label path of length no more than 9, since we randomly choose the start position of the query, the possible length of the query is restricted. Thus, short queries are more likely to be generated than long ones, which captures the observation that short path expressions are more common than long ones in reality. The cumulative distribution of queries by length is showing in Figure 8. Our workload consists of 500 queries for each dataset as FUP's.

We have also experimented with two query workloads where the maximum length of path expressions is 4 instead of 9. Recall that in our context, the path length is defined by the edge number of a path, which is one less than the convention of using the node number as the path length in [5]. The cumulative distribution of queries by length is shown in Figure 9. These workloads are similar to those used in [5] and allow us to reproduce similar performance results for A(k)- and D(k)-indexes.

5.1. Performance Results

Maximum query length of 9 Figures 10–13 summarize the performance results of various indexes for XMark and NASA datasets, respectively. In Figures 10 and 12, the horizontal axes show the index size in terms of the number of index nodes. In Figures 11 and 13, the horizontal axes show the number of index edges. For the adaptive indexes (D(k), M(k), and M*(k)), we show their final sizes after they have been refined to support all queries in the query workload. The vertical axes show the average performance of queries in the workload (in the number of nodes accessed during query evaluation). For the adaptive indexes, we rerun the workload to measure the average performance, after the indexes have been refined to support all workload queries; therefore, no validation cost is reflected by the average performance of rerun queries. For the A(k) family of indexes, however, queries in general do incur validation costs.

We plot the results for the family of A(k)-indexes, where $k = 0, 1, \ldots, 7$. For the D(k)-index, we experiment with two options. The first option, *D(k)-construct*, uses the D(k)-index construction procedure to construct an index from scratch to support all queries in the workload. For the second option, *D(k)-promote*, we start with an A(0)-index and incrementally refine it using the D(k)-index promoting procedure for each query in the workload. The final D(k)-index obtained for the second option can be quite different from the one constructed for the first option. For the M(k)- and M*(k)-indexes, we also start from an A(0)-index and incrementally refine it using the M(k)- and M*(k) refinement algorithms for each query in the workload. We use the top-down strategy to evaluate queries on the M*(k)-index.

From the figures, we see that the average query cost of the A(k)-index drops dramatically as k increases from 0 to 4 (Figures 10 and 11) or 2 (Figures 12 and 13), because a finer index significantly reduces the need for validation. However, the improvement quickly diminishes as k reaches 5 (Figures 10 and 11) or 3 (Figures 12 and 13), and the cost tends to rise afterwards. The reason is that the index has grown so large that it becomes expensive to evaluate path expressions on the index itself.

Both D(k)-construct and D(k)-promote offer good query performance. In terms of index size, D(k)-construct is

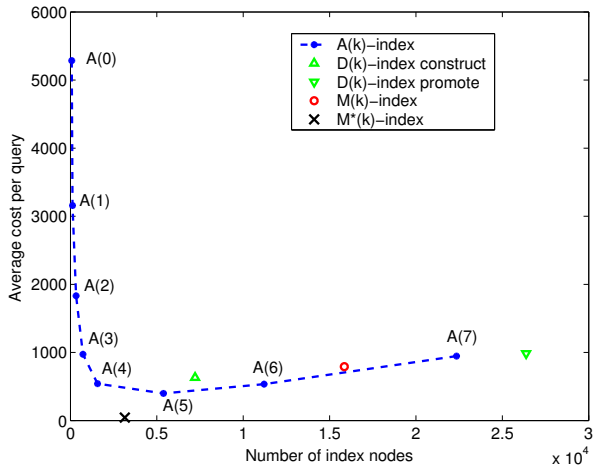


Figure 10. Query cost vs. number of index nodes on XMark dataset (max path length: 9)

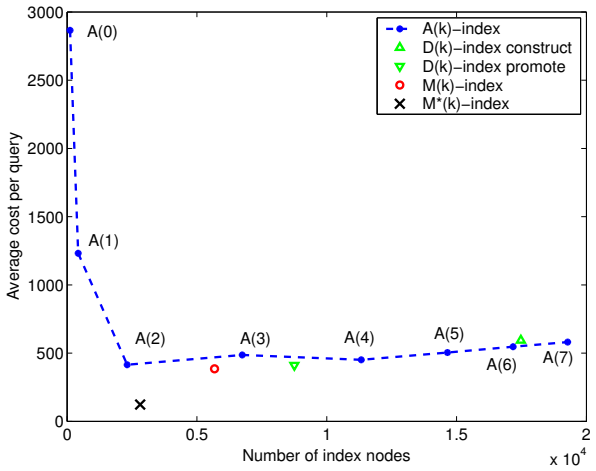


Figure 12. Query cost vs. number of index nodes on NASA dataset (max path length: 9)

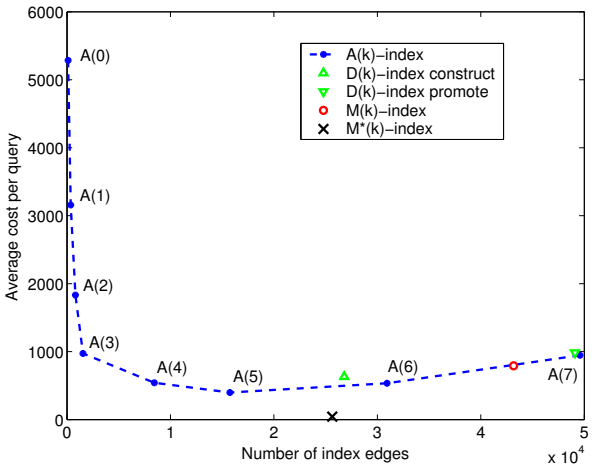


Figure 11. Query cost vs. number of index edges on XMark dataset (max path length: 9)

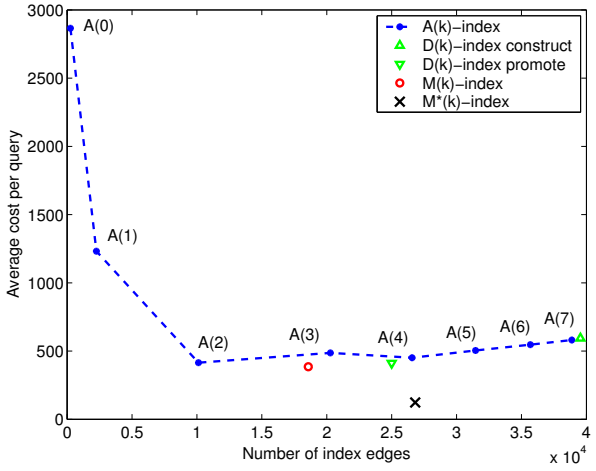


Figure 13. Query cost vs. number of index edges on NASA dataset (max path length: 9)

smaller for XMark, while D(k)-promote is smaller for NASA. This discrepancy can be explained by the different types of over-refinement possible under these two approaches. Both approaches suffer from over-refinement for irrelevant data nodes. In addition, D(k)-construct has over-refinement of irrelevant index nodes, while D(k)-promote has over-refinement due to overqualified parents. Irrelevant index nodes occur quite frequently in the NASA experiment, because many elements are used multiple times in different parts of the NASA DTD (e.g., name is used in seven different contexts); XMark reuses elements much less often. On the other hand, compared with NASA, XMark has a much simpler DTD, so it is very easy for workload queries to "collide," thereby creating more opportunities for the parents to be overqualified due to previous queries.

The M(k)-index outperforms D(k)-promote in both experiments, achieving lower query cost with much smaller index size, primarily because the M(k)-index avoids over-refinement for irrelevant data nodes. However, like D(k)-promote, the M(k)-index suffers from over-refinement due to overqualified parents. Therefore, while it outperforms D(k)-construct for NASA, it underperforms D(k)-construct for XMark, for the same reason discussed in the previous paragraph.

Compared with all other indexes, the M*(k)-index pro-

vides substantially lower query cost for both datasets. Interestingly, the M*(k)-index is also smaller than D(k)-promote, D(k)-construct, and the M(k)-index in terms of number of index nodes. The reason is that the M*(k)-index eliminates all three types of over-refinement: over-refinement of irrelevant index nodes (possible for D(k)-construct), over-refinement for irrelevant data nodes (possible for D(k)-construct and D(k)-promote), and over-refinement due to overqualified parents (possible for D(k)-promote and M(k)). Although the M*(k)-index maintains additional partitioning information for lower resolutions, the overhead is small compared with savings achieved by avoiding over-refinement.

In Figures 11 and 13 we show results from the same two experiments, this time using the number of index edges instead of index nodes as the horizontal axis. The M*(k)-index could potentially have a higher edge-to-node ratio than other indexes because we count cross-component links as edges. The relative size comparison of the various adaptive indexes is only slightly different from Figures 10 and 12. In terms of index edges, the M*(k)-index also remains to be competitive to other adaptive indexes, and beats them in the case of XMark.

Finally, for incrementally refined indexes (D(k)-promote, M(k), and M*(k)), we show how they grow in size as more queries are added to the FUP set. For every 50 queries added, we measure index size in terms of both number of nodes and number of edges. The results are shown in Figures 14–17. Note that all incrementally refined indexes exhibit similar growth patterns. The first batch of 50 FUP's tend to have the largest impact on index size, although several later batches also result in sudden size increases for certain indexes. The relative ordering of index sizes does not change during the course of refinement except for the NASA dataset when index size is measured in number of edges. Such an exception highly depends on the query set and the order of queries. Since queries are randomly generated, the relation among indexes with regard to their edge sizes are not determined. Our experiment shows that the number of edges in the M*(k)-index increases much faster than other indexes, as well as its own node size during the same period, in this exceptional case. This phenomenon can be explained by the conjunct effect of large fan-in or fan-out and multiple resolution. If an index node connects to many nodes, when it is split to several index nodes, the increase of new edges over that of new nodes is approximately factored by its degree. Meanwhile, when a target set of refinement is given, the M*(k)-index may incur much more nodes to split in multiple component indexes, since all local similarity information has to be maintained. Therefore, the increase of new edges in the M*(k)-index is also

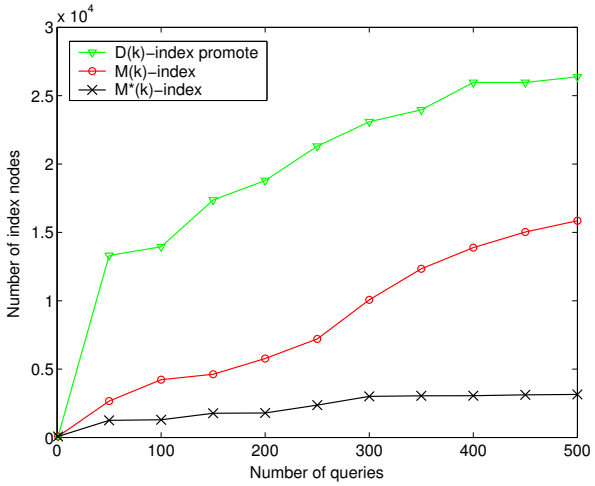


Figure 14. Index node size growth over queries on XMark dataset (max path length: 9)

more than those in other indexes. However, even in this case, its edge size is just slightly larger than others'. Considering the much smaller size in terms of index nodes, the M*(k)-index is the most space-efficient index in general.

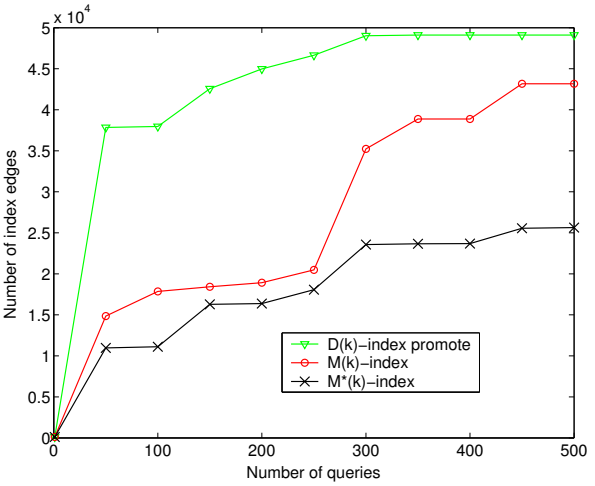


Figure 15. Index edge size growth over queries on XMark dataset (max path length: 9)

Maximum query length 4 This second set of experiments compares the performance of various indexes for relatively shorter queries. Since the maximum length of path expressions is 4, we only show the results of the A(k)-index

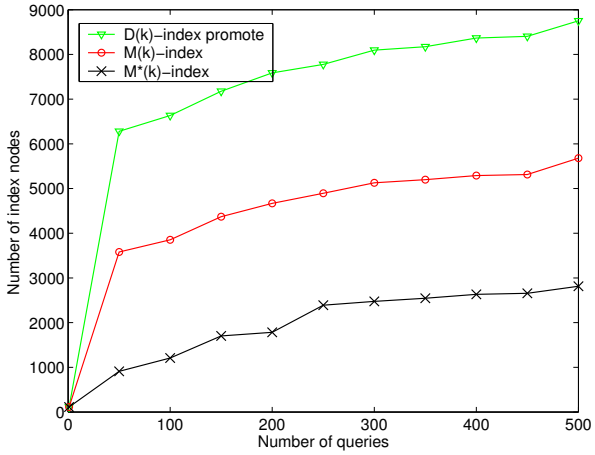


Figure 16. Index node size growth over queries on NASA dataset (max path length: 9)

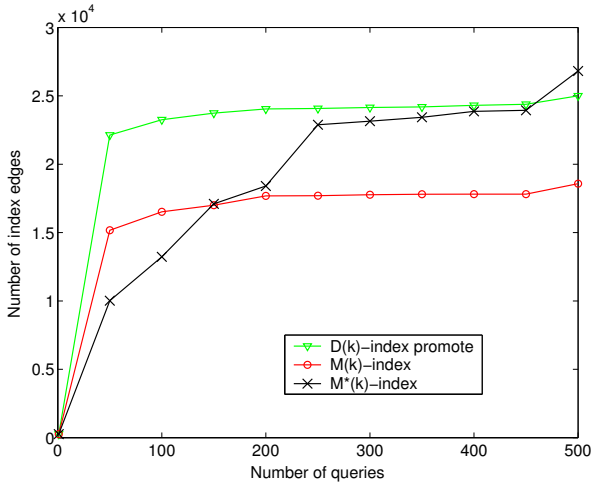


Figure 17. Index edge size growth over queries on NASA dataset (max path length: 9)

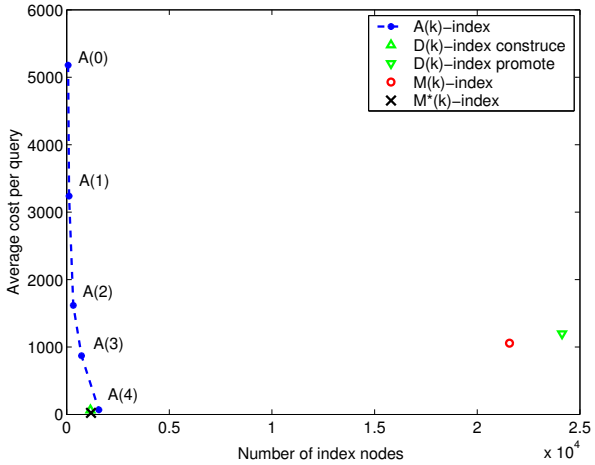


Figure 18. Query cost vs. number of index nodes on XMark dataset (max path length: 4)

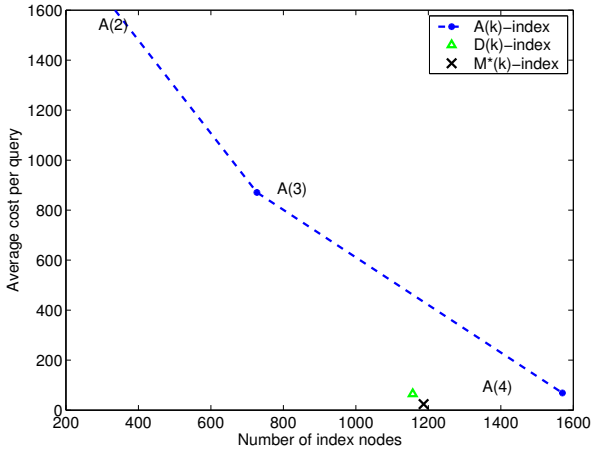


Figure 19. Query cost vs. number of index nodes on XMark dataset without D(k)-promote and M(k) (max path length: 4)

for $k \leq 4$. For A(k)-indexes and D(k)-construct, the results are consistent with those reported in [5].

Figures 18–22 plot the average cost of a workload query against the index size, both in terms of the number of nodes and in terms of the number of edges. Note that for XMark, both D(k)-promote and the M(k)-index suffer heavily from over-refinement due to overqualified parents. To visualize the rest of Figure 18 better, we plot Figure 19 without A(0), A(1), D(k)-promote, and M(k). In this figure, we see that both D(k)-construct and the M*(k)-index work clearly better than the A(k)-indexes. Furthermore, the M*(k)-index has much lower query cost than D(k)-construct, while using only slightly more space. The other figures lead to observations and conclusions similar to those for experiments on longer queries. In Figures 23–26, we also show how the incrementally refined indexes grow in size as more FUP's are supported. Again, we see that the M*(k)-index is almost always superior to the others.

6. Conclusion

We have introduced the M(k)- and M*(k)-indexes, which are incrementally refined structural indexes geared toward supporting a set of frequent path expression queries. Compared with previous proposals, these indexes avoid various types of over-refinement that ad-

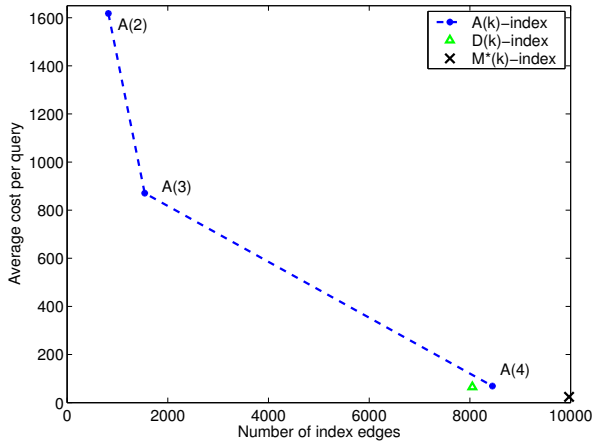


Figure 20. Query cost vs. number of index edges on XMark dataset without D(k)-promote and M(k) (max path length: 4)

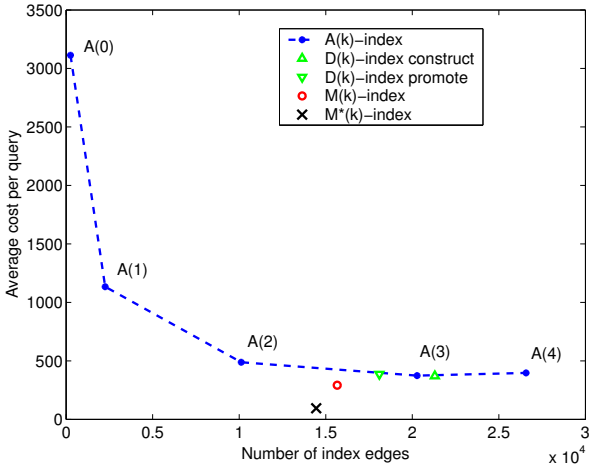


Figure 22. Query cost vs. number of index edges on NASA dataset (max path length: 4)

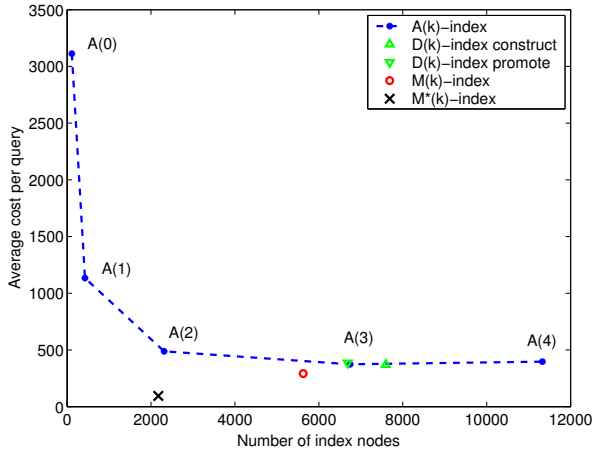


Figure 21. Query cost vs. number of index nodes on NASA dataset (max path length: 4)

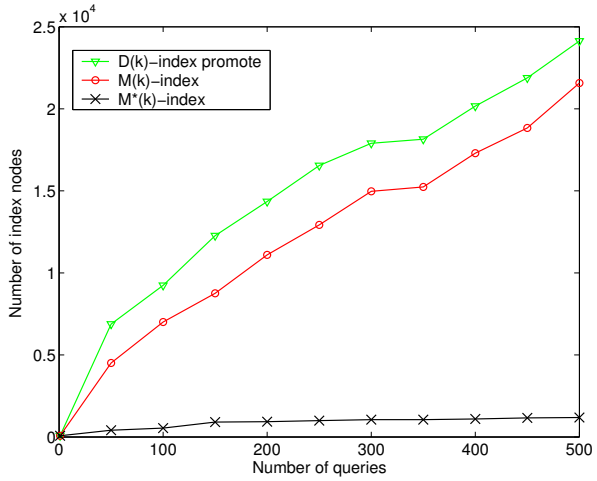


Figure 23. Index node size growth over queries on XMark dataset (max path length: 4)

versely affect index performance. The M(k)-index avoids over-refinement by targeting only the data nodes relevant to frequent queries. The M*(k)-index further eliminates over-refinement due to overqualified parents by maintaining partitioning information at multiple resolutions. The M*(k)-index is truly multiresolution, in the sense that it supports not only multiple resolutions across different parts of the data graph, but also multiple resolutions for the same part of the data graph. Besides avoiding over-refinement, the M*(k)-index is much more efficient for querying than previously proposed structural indexes, as the multiresolution partitioning information also serves as a multilevel index. Our experiments have demonstrated that the overhead of storing partitioning information at different resolutions is small compared with savings achieved by avoiding over-refinement, and well worthwhile because of the substantial improvement to query performance. We are currently studying how to make the M*(k)-index I/O-efficient by turning it into a disk-resident structure that can be loaded into memory selectively and incrementally during query processing.

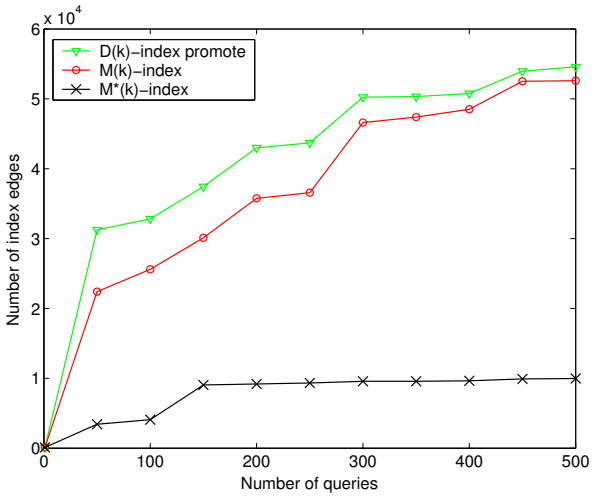


Figure 24. Index edge size growth over queries on XMark dataset (max path length: 4)

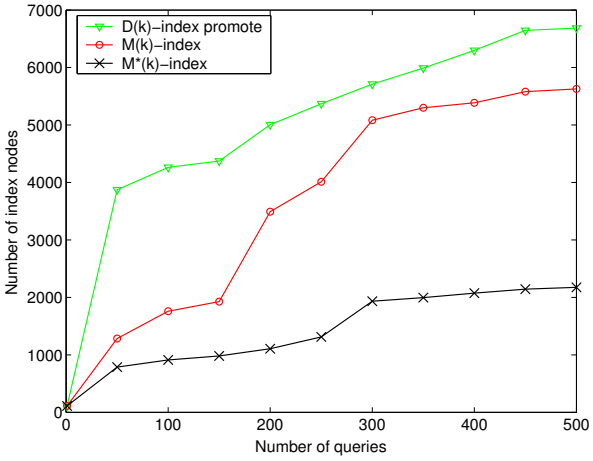


Figure 25. Index node size growth over queries on NASA dataset (max path length: 4)

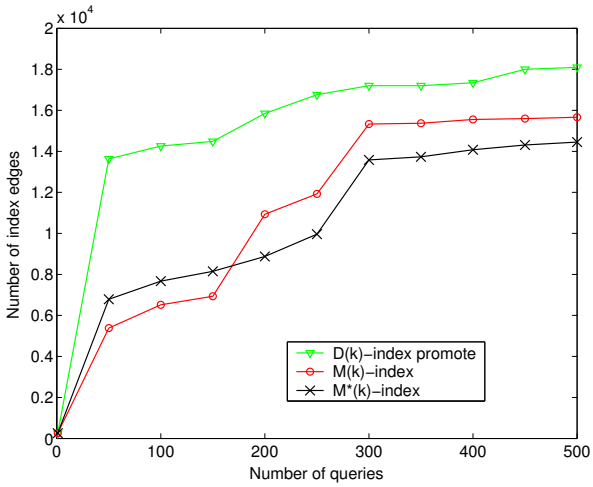


Figure 26. Index edge size growth over queries on NASA dataset (max path length: 4)

References

[1] S. Abiteboul. Querying semi-structured data. In *Proc. of the 1997 Intl. Conf. on Database Theory*, 1997.

[2] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The lorel query language for semistructured data. *Journal of Digital Libraries*, November 1996.

[3] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Siméon. Xml path language (xpath) 2.0. http://www.w3.org/TR/xpath20, August 2002.

[4] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Siméon. Xquery 1.0: An xml query language. http://www.w3.org/TR/xquery, August 2002.

[5] Q. Chen, A. Lim, and K. W. Ong. D(k)-index: An adaptive structural summary for graph-structured data. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, June 2003.

[6] C. Chung, J. Min, and K. Shim. Apex: An adaptive path index for xml data. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, June 2002.

[7] B. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A fast index for semistructured data. In *Proc. of the 2001 Intl. Conf. on Very Large Data Bases*, January 2001.

[8] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proc. of the 1997 Intl. Conf. on Very Large Data Bases*, pages 436–445, August 1997.

[9] NASA XML Group. available at http://xml.gsfc.nasa.gov/.

[10] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, June 2002.

[11] R. Kaushik, P. Sheony, P. Bohannon, and E. Gudes. Exploiting local similarity for efficient indexing of paths in graph structured data. In *Proc. of the 2002 Intl. Conf. on Data Engineering*, February 2002.

[12] Q. Li and B. Moon. Indexing and querying xml data for regular path expressions. In *Proc. of the 2001 Intl. Conf. on Very Large Data Bases*, 2001.

[13] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. In *SIGMOD Record 26(3)*, 1997.

[14] J. McHugh and J. Widom. Query optimization for xml. In *Proc. of the 1999 Intl. Conf. on Very Large Data Bases*, 1999.

[15] T. Milo and D. Suciu. Index structures for path expressions. In *Proc. of the 1999 Intl. Conf. on Database Theory*, pages 277–295, January 1999.

[16] R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16:973–988, 1987.

[17] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. The xml benchmark project. Technical report, CWI, 2001.

[18] H. Wu, Q. Wang, J. X. Yu, A. Zhou, and S. Zhou. Ud(k,l)-index: An efficient approximate index for xml data. In *Proc. of the 2003Intl. Conf. on Web-Age Information Management*, August 2003.

[19] C. Zhang, J. Naughton, D. Dewitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. In *Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, 2001.