D(K)-Index: An Adaptive Structural Summary for Graph-Structured Data

Qun Chen
Department of Computer Science
National University of Singapore
Lower Kent Ridge Road, Singapore 119260
chenqun@comp.nus.edu.sg

Andrew Lim
Department of Computer Science
National University of Singapore
Lower Kent Ridge Road, Singapore 119260
alim@comp.nus.edu.sg

Kian Win Ong
Department of Computer Science
National University of Singapore
Lower Kent Ridge Road, Singapore 119260
ongkianw@comp.nus.edu.sg

ABSTRACT

To facilitate queries over semi-structured data, various structural summaries have been proposed. Structural summaries are derived directly from the data and serve as indices for evaluating path expressions on semi-structured or XML data. We introduce the D(k) index, an adaptive structural summary for general graph structured documents. Building on previous work, 1-index and A(k) index, the D(k)-index is also based on the concept of bisimilarity. However, as a generalization of the 1-index and A(k)-index, the D(k) index possesses the adaptive ability to adjust its structure according to the current query load. This dynamism also facilitates efficient update algorithms, which are crucial to practical applications of structural indices, but have not been adequately addressed in previous index proposals. Our experiments show that the D(k) index is a more effective structural summary than previous static ones, as a result of its query load sensitivity. In addition, update operations on the D(k) index can be performed more efficiently than on its predecessors.

1. INTRODUCTION

In recent years, the eXtensible Markup Language(XML)[7] has become the dominant standard for exchanging and querying documents over the World Wide Web. XML is an example of semi-structured data [4, 5]. Semi-structured data do not conform to traditional data models, such as relational or object-oriented models. Instead, the underlying data model of semi-structured data is a labeled graph. XML documents consist of hierarchically nested elements, which can be either atomic, for instance raw character data, or composite, for instance a sequence of nested subelements. Tags stored with the elements describe the semantics of the data. References between elements can be established by using *id/idref* attribute or *Xlink* constructs [7, 8]. Thus, XML data, like semi-structured data, are hierarchically structured and self-describing.

A variety of query languages [1, 2, 3, 4] have been proposed to query semi-structured data or XML. All of these query languages are built around path expressions [6], which are used to traverse graph-structured data. Typically, a data node is selected by a path expression if some path to the node has a sequence of labels matched by the expression. The navigation of the graph structure underlying the semi-structured data and XML is therefore an essential component for querying these data. A naive evaluation of path expressions that scans all data is obviously very computationally expensive. We would like to build an index structure to speed up the evaluation process. A structural summary [9, 10, 11, 12] can be used to prune the search space significantly, thus improving the evaluation performance. Alternatively, an index graph, consisting of a structural summary along with stored mapping from index nodes to data nodes, may be directly used to evaluate such path expressions.

Existing structural summaries for graph-structured data are based on the notion of bisimilarity [15, 16]. Two nodes are bisimilar if all label paths into them are the same. Structural summaries consist of the collection of equivalence classes. Nodes in each equivalence class are bisimilar. The 1-index [11] is an accurate structural summary that considers incoming paths up to the root of the whole graph. The 1-index summary is *safe* and *sound*. Path expressions can be directly evaluated in the index graph and can retrieve label-matching nodes without referring to the original data graph. Unfortunately, 1-index structural summaries are usually quite large and are considered not efficient enough to speed up the evaluation. Exploiting the observation that long and complex paths tend to contribute disproportionately to the complexity of an accurate summary structure, the A(k)-Index [12] relaxes the equivalence condition and considers only incoming paths whose lengths are no longer than k. By taking advantage of the similarity of short paths, the A(k)-Index has been experimentally shown to have a substantially reduced index size. However, the A(k)-Index becomes only approximate for paths longer than k. Therefore, a validation process was introduced to extract exact answers from approximate index graphs.

The performance of the A(k)-Index largely depends on

SIGMOD 2003, June 9-12, 2003, San Diego, CA.

how to choose the parameter k. If k is large, the resulting index graph tends to remain large. The big size is a severe disadvantage for structural summaries. If we choose to use a small k, the index graph's size can be substantially reduced; but more queries should involve validation process, which is very inefficient because it requires traversing the source data. The key observation exploited by our new index proposal is that **not** all structures are of *equivalent significance*. Some nodes in the source data may be only traversing nodes, which aid in label path matching, but are never returned by queries. There is obviously no gain in refining index equivalence classes consisting of traversing nodes. Even for those nodes, which should be returned by query processing, the complexity of their structures that matters in query processing may differ. Depending on the actual query load, some type of nodes may be accessed using short paths most of the time; the other type of nodes may be frequently queried by long paths. Both 1-Index and A(k)-Index fail to adjust their index graphs according to the different structure complexity of the equivalence classes required by the query load, because of their static nature. In this paper, we introduce D(k)-Index, an adaptive structural summary for graph-structured data, which can be tuned efficiently for specific query loads to achieve reduced index size and improved performance. Instead of specifying the same local similarity, k, for every equivalence class in the index graph, the D(k)-Index uses possibly different, but the most effective local similarities for equivalence classes according to the current query load. As the query load changes incrementally, the D(k)-Index can be efficiently adjusted accordingly to maintain its high performance. And, not surprisingly, the inherent dynamism of the D(k)-Index also results in efficient update operations, which are crucial to any practical application of structural summaries, but were not adequately addressed in the previous literature. Our major contributions can be summarized as follows:

1. We propose the D(k)-index, an adaptive summary structure for the general graph-structured data and present an efficient construction algorithm. Unlike previous index structures that are regardless of the query load, our proposal takes advantage of query load information to optimize the D(k)-index structure accordingly.

2. We present efficient algorithms to update the D(k)-Index with changes in the source data and the query load. Believing that the update operation in the index resulting from a small change to the source data should be done very efficiently, we avoid the *propagate* partitioning strategy proposed for updating 1-index, which refers to the source data and thus can be potentially expensive. Instead, the D(k) index accommodates changes by adjusting the local bisimilarities of the affected index nodes, thus achieving high efficiency. Efficient algorithms to tune the D(k)-index as the query load changes are also presented.

3. We show by extensive experiments that the D(k)-index is a more effective summary structure than other static summary structures. It has a reduced index size and an improved performance. Updates on the D(k)-index can be executed more efficiently.

The remainder of this paper is organized as follows. In Section 2, we review related work about summary structures and path expression evaluation techniques for semistructured data and XML. In the third section, we present the data model and related concepts, i.e., path expression and local similarity. We proceed to describe the D(k)-index and its construction algorithm in Section 4. Updating the D(k)-Index is discussed in Section 5. And finally, in Section 6, experimental results are presented to show the improved performance of the D(k)-index over previous index structures.

2. RELATED WORK

Three previous summary structures have been proposed for graph-structured data to help evaluate path expressions, the strong DataGuide [9], the 1-index [11], and the A(k)-index [12]. We have already briefly examined the 1-index and the A(k)-index in the introduction section. The strong DataGuide of a graph data is computed by interpreting it as a non-deterministic automation and obtaining an equivalent deterministic automation. Thus, the path expression with k nodes is evaluated by matching a sequence of exactly k nodes in the strong DataGuide. Because of this, a data node may appear in extents of more than one index node. In the worst case, the number of index nodes in the strong DataGuide can be exponential related to the size of the data graph. This exponential behavior makes the strong DataGuide inappropriate for complex graph-structured data.

Update algorithms were proposed to maintain the strong DataGuide [9]. However, because the 1-index, A(k)-index and our new D(k) index, based on graph bisimulation, are non-deterministic when thought of antomata, those algorithms can not be generalized to apply in this context. Most recently, update algorithms for 1-index were presented in [17]. The authors considered the 1-index update algorithms for the insertion of a new document and edge addition. The *propagate* refinement strategy was adopted to update the 1-index incrementally. Although the 1-index update algorithm for document insertion can be easily generalized to apply in the A(k)-index context, the generalization of the update algorithm for edge addition was shown not to be clean.

Graph schema[18, 19] are also summary structures. However, construction and update algorithms were not discussed by the authors. Instead, they focused on structures of different schemas and explored possible applications of graph schemas to query optimization.

The bisimulation technique comes from the verification research community [20]. It is used to compress the state space graph in a manner that preserves some properties and behaviors of the state space. The compressed graph could then be analyzed with higher efficiency than the original state-space graph. A similar concept of local bisimilarity, localized stability, is also exploited to build the XSketch statistical synopses [13, 14] for graph structured data. The XSketch synopses takes advantage of different localized degrees of stability , demonstrated by the presence of backward-stable or forward-stable sub-paths with possibly different lengths, to achieve concise and effective summaries. Adopting the similar strategy that different portions of the data require different degrees of refinement, the D(k)-Index assigns higher bisimilarities to those nodes that are frequently accessed through long query paths.

Other indexing strategies have also been proposed to evaluate XML documents. The inverted index in [21] and the

numbering scheme in [22] enabled ancestor queries to be answered in constant time. In [23], every path in the data graph is viewed as a string and stored in a multi-level Patricia trie. Unfortunately, these indexing techniques were supposed to handle tree data. Extension of these structures to the context of graph data could be very difficult because of the possibly exponential number of paths in a graph. In [25], a workload-aware path index, termed APEX, was introduced for XML data. APEX enhances a summary structure with a hash tree such that frequently used paths can be queried more efficiently. An incremental update algorithm was presented to adjust APEX due to the change of query workloads. However, we note that no algorithm was provided to update APEX due to the change of the source data. In constrast, the D(k)-index, serving as a robust summary structure, can be incrementally adjusted according to changes of both the query load and the source data.

3. PRELIMINARIES

In this section, we describe our presentation of XML or other semi-structured data, path expression evaluation, and the concept of bisimilarity.

We model XML or other semi-structured data as a directed, labeled graph. Each edge in G indicates an object-subobject, or object-value relationship. Each node in G has a label and a unique identifier, with simple objects given a distinguished label, VALUE. There is also a single root element with the distinguished label, ROOT. The structure of XML documents is basically a tree, with edges representing element-subelement, element-attribute, or element-value relationships between nodes. The reference edges, which can be established between XML nodes using the *ID/IDREF* construct or *Xlink* syntax, make the model for XML documents a graph. In Figure 1, a portion of an XML document about movies is represented as a data graph. The solid edges, which are tree edges, represent containment relationships between nodes. Non-tree edges(shown as dashed lines) represent reference relationships. In our data model, we do not differentiate between these two kinds of edges, but treat both as normal edges.

We now introduce terminologies for paths and path expressions. A node path in the data graph G is a sequence of nodes, $n_1 n_2 \cdots n_p$, such that an edge exists between nodes n_i and n_{i+1} , for $1 \leq i \leq p-1$. A label path is a sequence of labels $l_1 l_2 \cdots l_p$. A node path matches a label path if $label(n_i) = l_i$, for $1 \leq i \leq p$. A label path, $l_1 l_2 \cdots l_p$ matches a node n if there is some node path ending in node n that matches $l_1 l_2 \cdots l_p$. A regular path expression, R, is defined in the usual way in terms of sequence(.), alternation(|), repetition(*) and optional expression(?), as follows:

$$R = \sum_G |\_|R.R|R|R|(R)|R?|R*$$

in which the symbol _ matches any l_i in G. And we denote the regular language specified by R as $L(R)$. We say that R matches a data graph node, n, if the label path for some word in $L(R)$ matches a node path ending in n. The result of evaluating R on G is the set of nodes in G that match R. For example, the path expression, *director.movie.title*, evaluated on the graph in Figure 1, will return $\{15, 16, 18\}$; the more complicated path expression, $movieDB.(\_)?.movie.actor.name$, finds names of actors in movies. Here, the optional _ allows the query to ignore the irregularities in the data graph. The *movie* node can appear directly after *movieDB*, or it can be a child of any labeled node, whose parent is *movieDB*. This expression matches nodes $\{12, 22\}$.

Structural summaries have been proposed to prune the searching space while evaluating path expressions. The idea is to preserve paths of the data graph in the summary graph, but with far fewer nodes and edges. If we associate an *extent*, which is a set of data nodes in the data graph, with a single node in the summary graph, it is possible for us to evaluate the path expression on the summary graph instead of the much larger data graph. We denote the index graph for data graph, G, as I_G. The result of executing a path expression, R, on I_G is the union of the extents of the index nodes in I_G that match R. We require the mapping from the data nodes to index nodes to be $safe$: if $l_1 l_2 \cdots l_m$ is a label path that matches node v in G, then this label path also matches some node A in I_G for which $v \in extent(A)$. This guarantees that the evaluation result of any path expression, R, on G is contained in the result of evaluating R on the index graph, I_G. An index graph, I_G, is said to be *sound* if the converse holds; that is, if the label path, P, $l_1 l_2 \cdots l_m$ matches node A in I_G, then it also matches every data node in $extent(A)$ in G.

Existing index structures for semi-structured data or XML are based on the notion of bisimulation.

Definition 1. (*Bisimulation*) Let G be a data graph in which the symmetric, binary relation $\approx$, the bisimulation, is defined as : we say that two data nodes u and v are bisimilar($u \approx v$), if

1. u and v have the same label;
2. if u' is a parent of u, then there is a parent v' of v such that $u' \approx v'$, and vice versa;

Two nodes u and v in the data graph G are bisimilar, denoted as $u \approx_b v$, if there is some bisimulation such that $u \approx v$. For example, in Figure 1, nodes 7 and 10 (*movie*) are bisimilar, while nodes 7 and 9 are not bisimilar, because node 7 has a parent labeled *actor*; but node 9 does not have any parent labeled *actor*. We can easily come to the conclusion by induction that if two nodes are bisimilar, the set of paths coming into them is the same.

4. D(K)-INDEX

4.1 Introduction to the D(k)-Index

We can obtain an index graph, I_G, by creating an index node for each equivalence class in the data graph, G. Data nodes in each equivalence class are mutually bisimilar. An edge is added from index nodes A to B in I_G if an edge exists in G between some data nodes, $v \in extent(A)$ and $u \in extent(B)$. Such an index graph is referred to as the 1-index structure. In the worst case, the 1-index graph can never be larger than the data graph. It can be constructed in $O(mlgn)$ time using Paige and Tarjan's algorithm [16], in which n is the number of nodes and m is the number of edges in the data graph.

Because of the big size of the 1-index and the rarity of long queries in practice, the A(k)-index proposal [12] takes advantage of local similarity to reduce the size of index graph.

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

1. For any two nodes, u and v, $u \approx^0 v$ iff u and v have the same label;

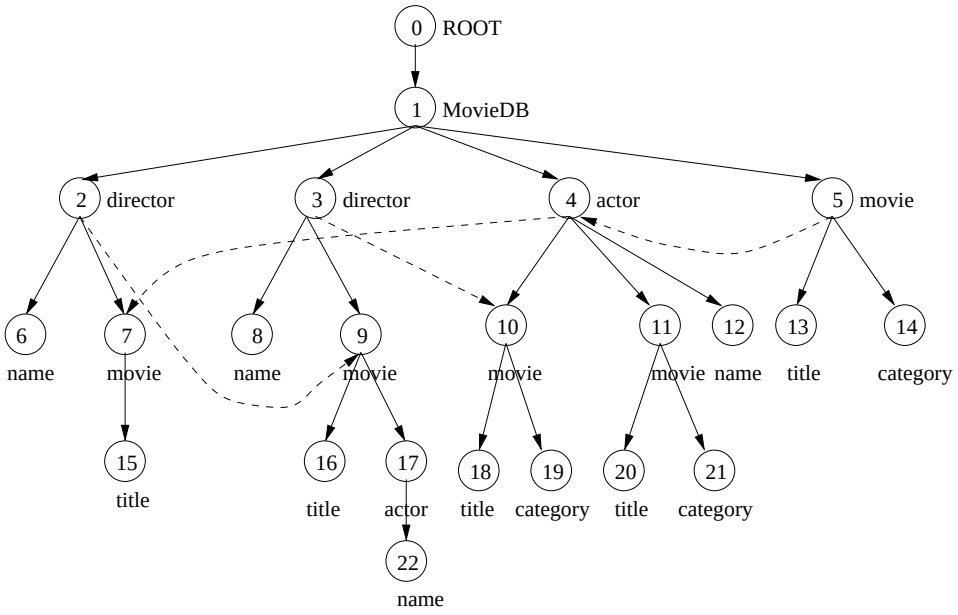

Figure 1: An Example Graph-Structured Data

2. Node $u \approx^k v$ 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 has the following properties [12]:

1. If nodes u and v are k-bisimilar, then the set of label paths of length $\leq k$ into them is the same.
2. The set of label-paths of length $m(m \leq k)$ into an A(k)-index node is the set of label paths of length m into any data node in its extent.
3. The A(k)-index is safe, i.e , its results on a path expression always contain the data graph results for that query.
4. The A(k)-index is sound for any path expression of length less than or equal to k.

The A(k)-index can be constructed in $O(km)$ time, where m is the number of edges in the data graph G. The evaluation result of the A(k)-index is accurate if the length of a path expression is less than or equal to k. Otherwise, the index results should be validated by referring to the data graph to return the final query results.

Our adaptive D(k)-index is also based on local similarity. Furthermore, it takes irregularity of query patterns into consideration. Different types of nodes in the data graph may be queried using different query patterns. In particular, since we expect the majority of path queries will be partial matching queries with the self-or-descendant axis('//'), the complexity of the relevant label paths entering different types of data nodes may differ. For example, in the data graph in Figure 1, if queries are only concerned with the names of actors or directors, regardless of movies they direct or act in, the index node for *name* nodes satisfying 1-bisimilarity would be sufficient to answer these queries accurately. But the index nodes for *title* nodes are required to comply with 2-bisimilarity to answer such queries that ask for the titles of movies directed by a specific director. Therefore, the local similarities of different types of data nodes required by the query load may vary. The A(k)-index fails to adapt to the query load, because it assumes the uniformity of query patterns. In contrast, by assigning different bisimilarity requirements to different types of data nodes according to the query load, the D(k)-index can adjust its structure optimally to achieve reduced index size and improved evaluation performance, a some specific query load.

For a given index node, A, in some index graph, I_G, we assume that the local similarity of A required by queries is k_A. The value of k_A can be obtained by mining the current query load. The choice of k_A should guarantee that the majority of queries accessing A are less than or equal to k_A in length. Thus, most queries on A can be directly performed on the index graph without the validation process, which is potentially inefficient because of reference to the data graph. Now we are ready to prove the theorem that lays the foundation for the correctness of the D(k)-index as a summary structure for graph-structured data. This theorem demonstrates that given a path P of length k in an index graph, I_G, $n_1 n_2 \cdots n_{k+1}$, if the index node n_i is of at least $(i-1)-bisimilarity$, for each $1 \leq i \leq (k+1)$, then the label path along P matches all data nodes in the $extent(n_{k+1})$.

THEOREM 1. *Given an index graph, I_G, and a path, P, $n_1 n_2 \cdots n_s$, in I_G. Assume that Label(n_i)=l_i, for each $1 \leq i \leq s$. If data nodes in the extent(n_i) are at least $(i-1)-$bisimilar, for each $1 \leq i \leq s$, then the label path, $l_1 l_2 \cdots l_s$, matches each data node in the extent(n_s).*

Proof: We prove by induction on the length of path P, s. The basic case when s=0 is obviously true. Assume that the result is true for $s = m-1$. When $s = m$, and $P = n_1 n_2 \cdots n_m n_{m+1}$, the label path $l_1 l_2 \cdots l_m$ matches all data nodes in $extent(n_m)$ according to the assumption of case

s=m. Because there is an edge between n_m and n_{m+1} in the index graph I_G, there exists some node u in $extent(n_{m+1})$, whose parents include some node v in $extent(n_m)$. Since the label path $l_1 l_2 \cdots l_m$ matches v, one of the nodes in $extent(n_m)$, the label path $l_1 l_2 \cdots l_m l_{m+1}$ matches node u. Finally, nodes in $extent(n_{m+1})$ are at least $m - bisimilar$, so the label path $l_1 l_2 \cdots l_m l_{m+1}$, whose length is equal to m, matches all data nodes in $extent(n_{m+1})$.

According to theorem 1, given an index graph, I_G, if for any two directly connected index nodes $n_i \rightarrow n_j$ in I_G, $k(n_i) \geq k(n_j) - 1$, in which $k(n_i)$ and $k(n_j)$ are local similarities of n_i and n_j, respectively, then the query result of a path expression of length s on I_G, $n_1 n_2 \cdots n_{s+1}$, is accurate so long as $k(n_{s+1}) \geq s$. We call this index graph I_G the D(k)-index.

Definition 3. The D(k)-index is the index graph based on local bisimilarity that satisfies the condition that for any two nodes n_i and n_j, $k(n_i) \geq k(n_j) - 1$ if there is an edge from n_i to n_j, in which $k(n_i)$ and $k(n_j)$ are n_i and n_j's local similarities, respectively.

According to this definition, the 1-index and A(k)-index are both special cases of the D(k)-index. In the D(k)-index, the local similarity of the parent plus one can not be less than the local similarity of its child. Note that given graph data, G, the simplest index graph constructed by label splitting is a D(k)-index with the local similarity of each index node equal to 0.

Some important properties of the D(k)-index are given as follows. Their proofs should be obvious from the D(k)-index definition and theorem 1.

1. The set of label paths of length $s(\leq k(n_i))$ into a node n_i in the D(k)-index is the set of label paths of length s into any data node in its extent;
2. The D(k)-index is safe, i.e , its result on a path expression always contains the data graph result for that query;
3. The D(k)-index is *sound* for a path expression P of length m, $l_1 l_2 \cdots l_{m+1}$, if, for each matching index node n_i of P, $k(n_i) \geq m$.

4.2 Construction Algorithm

We now present the D(k)-index construction algorithm. We begin with the simplest index graph, the label-split graph. The local similarity requirement for each label can be obtained from the query load. The default local similarity requirements of those labels that never appear in the query load are set to zero. The resulting D(k)-index should satisfy the requirement that for each label, all nodes in the D(k)-index with such a label have a local similarity larger than or equal to the required one.

Besides requirements by query load, local similarities of index nodes may also be constrained by the structure requirement of the D(k)-index. For example, for two directly connected nodes, n_i and n_j ($n_i \rightarrow n_j$), in the label-split index graph, if the local similarities of n_i and n_j specified by the query load are 0 and 2, respectively, the local similarity of n_i should be reset to 1 because the local similarity of the parent, n_i, can not be > 1 less than its child n_j's local similarity. Therefore, we use a broadcast algorithm to compute the actual local similarities of labels in the D(k)-index. First, we specify a local similarity for each label in the index graph according to the current query load. Assume there are t different local similarities, and $k_1 > k_2 > \cdots > k_t$. For each local similarity k_i, for $1 < i < m$, a list of labels with local similarity requirement k_i is attached to it. Second, beginning with the largest local similarity k_1, the algorithm "*broadcasts*" the local similarity requirements to all parents of labels in its list. Then it continues with the second largest local similarity and goes on until all local similarities are processed. The detailed algorithm is described below. It takes $O(m)$ time, in which m is the number of edges in the label-split index graph.

Algorithm 1: The Local Similarity Broadcast Algorithm

Input The label-split index graph, G, with initial local similarities for label nodes in G.

Output The index graph, G, with updated local similarities for label nodes in G, as required by the D(k)-index

1. Sort all local similarities in G, $k_1 > k_2 > \cdots > k_t$, and for each local similarity k_i, a list of label nodes with local similarity k_i is attached to it;
2. Beginning with the largest local similarity, k_1, for each k_i, repeat the following process:
   - For each label node, n_j, in the list for k_i, update the local similarities of all parents of n_j in G such that their new local similarities are no less than $(k_i - 1)$. That is, if the original local similarity is no less than $(k_i - 1)$, the node remains unchanged; otherwise, its local similarity should be set to $(k_i - 1)$;
   - Update the local similarity list and their attached label nodes list;
   - Select the next largest local similarity and repeat Step 2;

With local similarities for label nodes in the label-split index graph, our D(k)-index can be constructed using a similar algorithm as the A(k)-index construction algorithm [12]. For a set of data nodes, A, let $Succ(A)$ denote the set of successors of the nodes in A, i.e., the set $\{v|$there is a node $u \in A$ with an edge from u to $v\}$. And given two set of data nodes, A and B, we say that B is stable with respect to A if B is a subset of $Succ(A)$ or B and $Succ(A)$ are disjoint. If we have two node sets, A and B, and we want to make B stable with respect to A, we split B into $B \cap Succ(A)$ and $B - Succ(A)$. As in the A(k)-index construction, we compute the $(k+1)$-bisimulation equivalence classes from the k-bisimulation equivalence classes. We make a copy of the k-bisimulation equivalence classes and then split them until they are stable with respect to the equivalence classes of k-bisimulation. The D(k)-index construction algorithm also begins with the label-split index graph, in which all index nodes are 0-bisimulation equivalence classes. Then it proceeds to construct the 1-bisimulation equivalence classes. It repeats this process until the local similarity requirements of all index nodes are satisfied. The D(k)-index construction algorithm is presented in Algorithm 2. A construction example is shown in Figure 2. It takes O(km) time in the worst case, in which m is the number of edges in the data graph G and k is the maximal local similarity requirement.

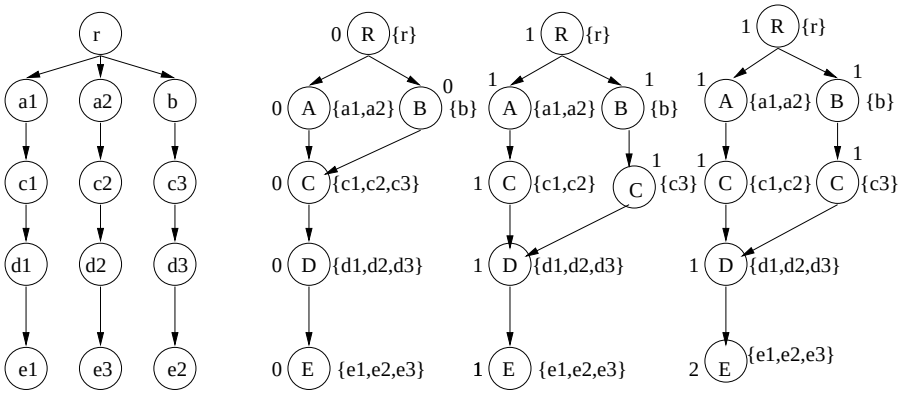

Figure 2: D(K)-Index Construction Example: (1) Label E has a local similarity requirement , 2, other labels have a local similarity requirement, 1; (2) the numbers besides the nodes are actual local similarities.

Algorithm 2: The D(k)-index Construction Algorithm

Input The data graph G, and local similarity requirements of label nodes specified by the query load.

Output The D(k)-index graph I_G.

1. Build the label-split index graph I_G from G;
2. Use the **The Local Similarity Broadcast Algorithm** to update the local similarities of index nodes in I_G;
3. X is a copy of I_G;
4. For $k = 1$ to k_{max} (k_{max} is the maximal local similarity requirement in I_G)
   - For each index node n_i in X
     - If (its local similarity requirement $\geq k$)
       - For each parent n_j of n_i in X
         - Replace the node n_i in I_G with $n_i \cap Succ(n_j)$ and $n_i - Succ(n_j)$;
         - Update the edges in I_G;
   - Set the local similarity requirements to newly created index nodes by inheritance;
   - Set X to be a copy of the updated I_G;
5. Return the resulting I_G.

5. D(K)-INDEX UPDATING

As in [17], we study two kinds of updates: the addition of a subgraph and the addition of a new edge. The addition of a subgraph represents the insertion of a new file into the database; the addition of a new edge represents a small incremental change. All other update operations on the D(k)-index can be built on these two basic cases. In this section, we present efficient update algorithms for the D(k)-index. First, we give algorithms to update the D(k)-index when a new file is inserted or a new edge is added into the data graph. Then, we proceed to propose two procedures, *promoting* and *demoting*, to adjust the D(k)-index for a changing query load.

5.1 Subgraph Addition

The update algorithm on the D(k)-index for a subgraph addition is a variant of the update algorithm for the 1-index [17]. Suppose that a new subgraph, H, is inserted under the root of the original data graph, G. We can compute the D(k)-index, I_H, on the new subgraph and add I_H as a subgraph under the root of I_G. Then, simply treating the new I_G as a data graph, we compute the D(k)-index for the new data graph. Note that the index nodes with the same label in the original I_G and I_H should have the same local similarity. The correctness of this procedure is established through the following theorem. It is essentially a variant of theorem 1 in [17].

THEOREM 2. *Let G be a data graph. Let I_G be the D(k)-index for G and I'_G be an index graph constructed from any refinement of I_G. Then, the D(k)-index graph for I'_G is the same as the D(k)-index for G, I_G.*

Algorithm 3: Subgraph_Addition_Update_Algorithm

Input A D(K)-Index graph I_G for G and a new subgraph H.

Output A D(K)-index $I_{G'}$ for the new data graph G' consisting of G and H.

1. Construct the D(k)-index, I_H, for the new subgraph H;
2. Add I_H as a subgraph under the root of the original D(k)-index, I_G;
3. Treat the new I_G as a data graph and compute its D(k)-index, $I_{G'}$;
4. Set the extents of nodes of $I_{G'}$ by merging the nodes' extents in I_G;
5. Return the resulting $I_{G'}$.

5.2 Edge Addition

It has been shown that a small change in a graph can trigger large changes in the 1-index and A(k)-index [17]. An edge insertion in the original data graph may affect all its descendants in the 1-index or all descendants within distance k in the A(k)-index. This is demonstrated in the example in Figure 3. The *propagate* algorithm for the edge addition proposed in [17] essentially refines all descendant index nodes. In the worst case, it needs to touch $\mathbf{O}(n+m)$ nodes and edges in the data graph. In contrast, the D(k)-index update algorithm for edge addition is more efficient. Instead of referring to the data graph to partition the index nodes, the update operation on the D(k)-index simply lowers the local similarities of the affected index nodes. When a new edge, from A to B, is added to the index graph I_G, we can simply bring B's local similarity down to 0 and update the local similarities of its neighbor index nodes accordingly. That is, all B's children's local similarities should be reset to 1 if their original local similarities are larger than 1. Generally, an index node , k distant from B in I_G, should be updated such that its local similarity is no larger than k.

When a new edge is added to the D(k)-index graph, the local similarity of the end index node would be lowered to 0 only in the worst case. There is some possibility that its local similarity can be updated to a higher value. In the example in Figure 3, the end index node, D, has a parent index node, C, in the original D(k)-index. This means that all data nodes in D have some parent labeled c in the old data graph. Thus, the new edge from c_3 to d_2 doesn't change label parents of d_2. Since D's original local similarity before the edge addition is larger than 1, the local similarity of B after the edge addition can at least remain at 1. We therefore reset D's local similarity to 1 and its child E's local similarity to 2.

Generally, the update operation for the edge addition on the D(k)-index can be conducted in two steps. Suppose that a new edge is added to the D(k)-index, I_G, from U to V and V's original local similarity is k_V. We make the observation that if all label paths of length $k_N (\leq k_V)$ going into V, through U, match V in the original I_G, V's updated local similarity can be reset to k_N. Therefore, at the first step, the update operation decides the maximal k_N, such that all label paths of length k_N into V, through U, match V in the original I_G. This algorithm is presented below as the algorithm **The Update_Local_Similarity**. Beginning with $k_N = 0$, which is obviously true, it repeatedly checks if all label paths of length $k_N = k_N + 1$ into V through U match V in the original I_G. Suppose that all label paths of length k_N into V through U match V in the original I_G. Then, for each such label path P, $l_{k_N} \cdots l_2 l_1 (l_2 = U$ and $l_1 = V)$, we denote the set of those index nodes in I_G as $S'_{k_N}(P)$, which has a path into V through U matching P. Similarly, for the label path P, the set of index nodes is denoted as $S_{k_N}(P)$, which have a path into V in the original I_G matching P. If for each label path of length k_N P matching V through U, labels of parents of all nodes in $S'_{k_N}(P)$ are included in labels of parents of all nodes in $S_{k_N}(P)$, we can conclude that all label paths of length $k_N + 1$ into V, through U, match V in the original I_G. At the second step, the algorithm updates V's local similarity to k_N. Simply using the breadth-first search, it broadcasts this update to V's neighboring nodes in I_G. An index node, which is r distant from V in the breadth-first search, should lower its local similarity to $(k_N + r)$ if its original local similarity is larger than $(k_N + r)$; otherwise, its local similarity remains unchanged and the algorithm stops propagating the update request from this node. The whole algorithm is sketched in the update algorithm **Edge_Addition_Update_Algorithm**. Note that in the worst case, the update algorithm for edge addition with the D(k)-index can touch nodes and edges within distance k_V in the index graph I_G, which has much fewer nodes and edges than the data graph G. Thus, it can be expected to be much more efficient than the update operation on the 1-index and A(k)-index. We validate our claims by experiments in the experimental evaluation section.

Algorithm 4: Update_Local_Similarity

Input A D(K) index I_G and a new edge from node U to node V in I_G;

Output The new local similarity for node V.

1. Upbound=min{$K_U + 1$,k_V}; // V's new local similarity can not be larger than $K_U + 1$ or k_V.
2. NewLocalSimilarity=0,Stop=false;
3. NewLabelPathSet={$label(U)$}, OldLabelPathSet={l|l is the label of some parent of V in I_G}; And for each label path P in NewLabelPathSet, we keep a set of index nodes in I_G, $S'(P)$, which are starting nodes of matching node paths into V through U; Similarly, for each label path P in OldLabelPathSet, we keep a set of index nodes, $S(P)$, that are starting nodes of matching node paths in the original I_G;
4. While (NewLocalSimilarity≤Upbound and Stop=false)
   - if (NewLabelPathSet ⊆ OldLabelPathSet)
     - NewLocalSimilarity = NewLocalSimilarity + 1;
     - OldLabelPathSet=NewLabelPathSet;
     - Set UpdatedNewLabelPathSet to an empty set;
     - Set UpdatedOldLabelPathSet to an empty set;
     - For (each label path P in OldLabelPathSet)
       - for each index node w in S(P)
         - for each parent x of w in I_G, insert the label path P'=(label(x)+P) to UpdatedOldLabelPathSet and set S(P')={x};
       - delete the redundant label paths and merging their index nodes sets in UpdatedOldLabelPathSet such that label paths are unique;
     - OldLabelPathSet = UpdatedOldLabelPathSet;
     - for (each label path P in NewLabelPathSet)
       - for each index node w in S'(P)
         - for each parent x of w in I_G, insert the label path P'=(label(x)+P) to UpdatedNewLabelPathSet;and set S'(P')={x};
       - delete the redundant label paths and merging their index nodes sets in UpdatedNewLabelPathSet such that label paths are unique;

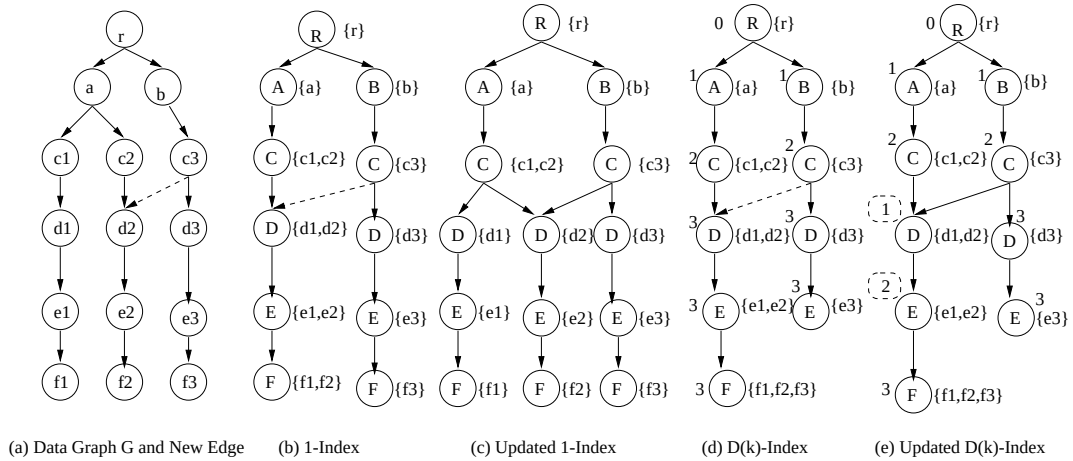

Figure 3: 1-index Update vs D(k)-index Update

- NewLabelPathSet = UpdatedNewLabelPathSet;

- else Stop=true;

5. Return NewLocalSimilarity.

Algorithm 5: Edge_Addition_Update_Algorithm

Input A D(K)-Index graph I_G for G and an new edge from U to V

Output An updated D(K)-index $I_{G'}$

1. k_N=Update_Local_Similarity(I_G,(U,V));
2. Set V's local similarity to k_N;
3. Beginning with the index node V, it traverses the nodes in I_G in breadth-first order. Suppose the edge from W to X is being considered, the updated local similarity of W is k_1, the old local similarity of X is k_2. If $(k_1 + 1 < k_2)$, it updates X's local similarity to $(k_1 + 1)$; otherwise, X's local similarity remains unchanged and the algorithm stops propogating the update request from X.

5.3 The Promoting Process

As more new edges are added to the D(k)-index graph, we can expect that local similarities of index nodes will decrease gradually. As the query load changes, higher local similarities may be required for some index nodes. If we do not upgrade related index nodes' local similarities, more queries will trigger validations. Since the validation process involves referring to the data graph to check the correctness of the answers on the D(k)-index, it can bring down the performance of the query processing significantly. Therefore, in this subsection, we propose a promoting procedure to upgrade local similarities of the index nodes in the D(k)-index. The promoting procedure should be executed *periodically* to tune the D(k)-index and keep its high performance.

To upgrade the local similarity of an index node V in the D(k)-index I_G, from k_1 to k_2, we adopt the same strategy as the D(k)-index construction algorithm. We first upgrade V's parents' local similarities to $(k_2 - 1)$ and then split the extent of V according to their parents. Specifically, for each parent U of V in I_G, the algorithm splits $extent(V)$ into $A \cap Succ(U)$ and $A - Succ(U)$. The local similarity upgrading on V's parents can be accomplished recursively. When the algorithm reaches the index nodes with local similarities no less than the required value, it begins the partitioning operation. The recursive promoting procedure is given in the **Promoting_Procedure_Algorithm**. In practical applications, there is usually a batch of index nodes that need to be promoted. Then, we choose first to promote index nodes with higher new local similarities, because upgrading them involves upgrading their close ancestors. The result is that some index node promotions may be saved.

Algorithm 6:
Promoting_Procedure_Algorithm(V, k_n, I_G)

Input A D(K)-Index I_G, an index node V in I_G and the new local similarity for V, k_n

Output An updated D(K)-index I'_G

1. If $(k_v \geq k_n)$ return I_G;//k_v is V's original local similarity in I_G
2. For each parent W of V in I_G
   - I_G=**Promoting_Procedure_Algorithm**$(W, k_n - 1, I_G)$;
3. For each parent W of V in I_G
   - split $extent(V)$ into $V \cap Succ(W)$ and $V - Succ(W)$;
4. Return the final I_G.

5.4 The Demoting Process

As updates on the D(k)-index proceeds, we can expect it to become larger gradually because of the refinements conducted on its index nodes. The query pattern may also changes. So it is important that the D(k) index be shrunk to a smaller size when its size becomes a disadvantage. A smaller size means less accuracy in the structural summary. For the D(k)-index, smaller size can be achieved by lowering

the local similarities of the index nodes, thus making it possible to merge some index nodes with the same label. This is why the shrinking procedure is called the demoting process. It actually downgrades the local similarities of index nodes in the D(k)-index. Like the promoting process, the demoting process is executed only *periodically*. Theorem 2 in the subsection **Subgraph Addition** states that from any refinement of a D(k)-index I_G, we can construct the original D(k)-index I_G. Therefore, given lower local similarities for labels in G, we do not need to reconstruct the D(k)-index I_G from scratch, which is obviously very time consuming. Instead, since the current D(k)-index I_G' is actually a refinement of I_G, we can just treat I_G' as a data graph and construct the new D(k)-index I_G from I_G'.

6. EXPERIMENTAL STUDY

In this section, we will validate the effectiveness and efficiency of our new D(k)-index through extensive experiments. We will compare our D(k)-index with the previous structural index A(k)-index, since the A(k)-index has been shown to outperform the 1-index. Our experiments show that:

1. The D(k)-index achieves the higher evaluation performance than the best A(k)-index, given specific query loads;
2. The update operations on the D(k)-index can be done more efficiently than on the A(k)-index;
3. The D(K)-index, after a considerable number of update operations, can still keep its better evaluation performance than the best A(k) index.

We use two datasets in our experiments: one benchmark data and one synthetic data.

1. Xmark Data. This is a synthetic XML data set from an XML benchmark [26], which simulates information about activities of an auction site. It features a regular structure. We use the benchmark data generator to generate an Xmark file of about 10M in size.
2. Nasa Data. This data set is generated by the IBM data generator using a real DTD file, nasa.dtd [27], which is a markup language for the data and metadata at the astronomical data center at NASA/GSFC. It has a broader, deeper and less regular structure than the Xmark data. It also has more references. To make the index size smaller and more manageable, we delete 12 of its original 20 references. The resulting Nasa data is an XML file of about 15M in size.

6.1 Evaluation Performance Before Updating

Because no standard storage scheme and query cost model exists for graph-structured data, we adopt the simple in-memory cost model used in evaluating the $A(k)$-index [12]. The cost of a query is defined to be the number of nodes visited in the index or data graph during path expression evaluation. Note that data nodes in the extent of a matched index node are not counted as visited; but the data nodes visited during the validating process are counted.

We randomly generate 100 test paths with lengths between 2 and 5 for the Xmark and Nasa data. First, the program randomly chooses some long query paths; then, from these long paths, many shorter branching paths are generated. These basically simulate query patterns in real XML databases. We can expect that most real XML queries will be posed with these structures. In the D(k)-index, we set a label's local similarity requirement to be the longest length of test path queries less one such that no validation will be needed for evaluation on it. And we compare D(k)-index's performance with A(0), A(1), up to A(4). Note that evaluating test paths on the A(4)-index is already *sound*; that is, no validation process is triggered because all test paths are of length less than or equal to 5. Therefore, we do not experiment on A(k) with $k > 4$ because its performance is definitely worse than A(4). The results on the Xmark and Nasa data are presented in Figures 4 and 5, respectively. The X-axis denotes the number of nodes in the index graph; the Y-axis denotes the evaluation cost measured by the average number of nodes visited over all test paths. In both figures, the D(k)-index result is well below the curve of the D(k)-index. Therefore, these results demonstrate the superior performance of the D(k)-index over the A(k)-index.

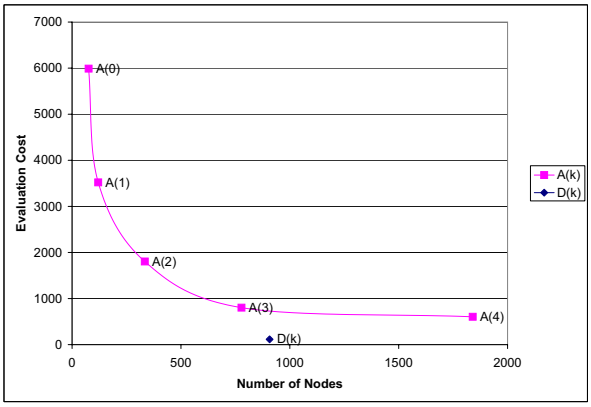

Figure 4: Evaluation Performance Comparison between the D(K)-index and the A(k)-index on Xmark Data Before Updating

6.2 Updating Performance

To evaluate the updating performance, we randomly choose a pair of $ID/IDREF$ labels in the DTD file and one data node from each label group; then, a new edge is added between these two data nodes. Since 1-index is a special case of the A(k)-index, we compare our D(k)-index's updating performance with the A(k)-index's performance. Unfortunately, so far as we know, no update algorithm has been proposed for the A(k)-index, so we adopt a variant of the 1-index update algorithm [17]. When a new edge is added to the A(k)-index graph, it creates a new index node. Next, it recursively checks if the newly created index nodes' child index node satisfies k local similarity. If yes, it stops; otherwise it partitions the extent of the target index node such that the data nodes in the extents are actually k-bisimilar. The update is propagated to index nodes up to $(k-1)$ dis-

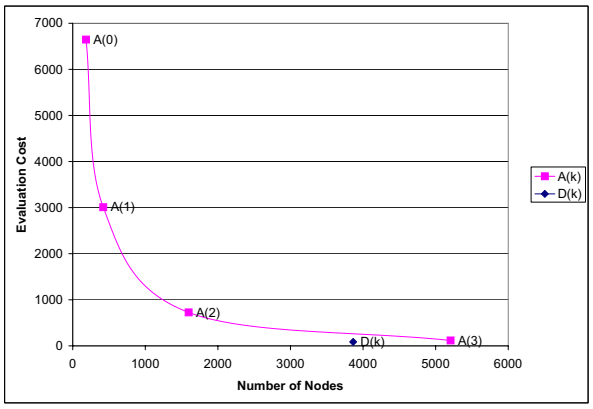

Figure 5: Evaluation Performance Comparison between the D(K)-index and the A(k)-index on Nasa Data before Updating

	Running Time(msec)	
	Xmark	Nasa
A(1)	1,022	3,863
A(2)	3,322	11,126
A(3)	5,196	31,992
A(4)	23,262	53,090
D(k)	2	1377

Table 1: Update Efficiecy Comparison Between D(k) and A(k)

tant from the first new index node. We randomly add 100 new edges to data graphs, and measure the running time of the update algorithms for A(1), up to A(4), and D(k). In case of the A(0) index, the index graph remains unchanged. Our machine features Linux OS, a Pentium 41.8 Ghz processor and a 512 RAM. Our machines memory is large enough that the data resides in the primary memory during the update execution. The detailed results are given in Table 1, in which the running time is the total accumulative time to perform all updates. As the value of k increases, the update cost on the A(k) index shoots up dramatically. Not surprisingly, the updating on the D(k) is significantly faster than on the A(k)-index.

6.3 The Effect of Updating on Evaluation Performance

After updating on the D(k)-index, its evaluation performance suffers, because its index nodes' local similarities have been decreased and the evaluation triggers more validation. As for the A(k)-index, the evaluation cost increases more moderately. However, while the D(k)-index size remains unchanged after updating, the size of the A(k) index increases dramatically. Factoring both the size and evaluation cost, our experiments show that the D(k) index still has better than or roughly the same performance as the best A(k) index. The evaluation performance comparisons after updating are presented in Figures 6 and 7. The promoting process proposed in the last section can improve the D(k)-index's performance after updating. This part of experiments will be included only in the full version of this paper.

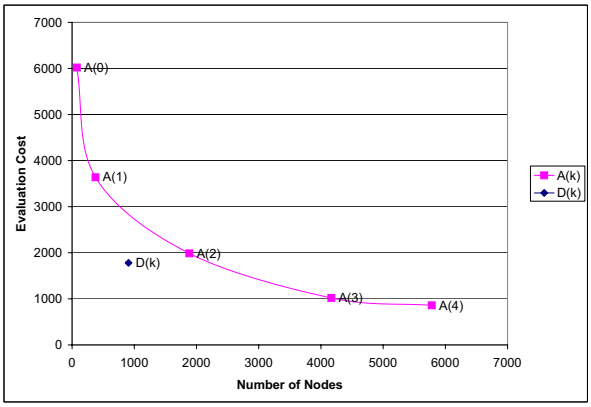

Figure 6: Evaluation Performance Comparison between the D(K)-index and the A(k)-index on Xmark Data after Updating

7. CONCLUSION AND FUTURE RESEARCH

The D(k)-index is a clean generalization of the previous 1-index and A(k)-index structures. It has clear advantages over them because of its dynamism. Subject to the changing query load, it can adjust its structure accordingly. We have shown by experiments that it achieves a higher evaluation performance than previous static index structures. Equally significantly, the D(k)-index also has more flexible and efficient update algorithms, which are crucial to such summary structure's applications. Our experiments demonstrate the superiority of the update operations on the D(k)-index over the update operations proposed for previous summary structures.

As for the future work, the research on summary structures for graph-structured data can be pushed on two fronts. One direction is to mine query patterns on query loads. An effective query pattern mining technique is not only important to the D(k)-index structure's performance, but is closely related to other index structures proposed for semi-structured data, for instance, the F&B index [24]. Another direction we will pursue is to tune and improve the update and evaluation efficiency of the D(k)-index on a real system. Currently, the update and evaluation processes are executed independently. Potentially, they can be combined to speed up the D(k)-index's processing of path queries.

8. REFERENCES

[1] D.Chamberlin, D.Florescu, J. Robie, J.Simeon, and M.Stefanescu. *XQuery: A Query Language for XML*. World Wide Web Consortium, http://www.w3.org/TR/xquery.

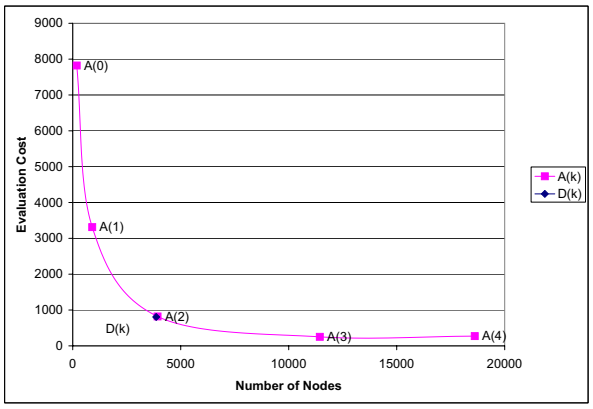

Figure 7: Evaluation Performance Comparison between the D(K)-index and the A(k)-index on Nasa Data After Updating

[2] A.Deutsch, M. Fernandez, D.Florescu, A.Levy, and D.Suciu, *A Query Language for XML*, Proceedings of the Eighth World Wide Web Conference, 1999.

[3] D.Chamberlin, D.Florescu, and J.Robie, *Quilt: An XML Query Language for Heterogeneous Data Sources*, Proceedings of WebDB, 2000.

[4] S.Abiteboul, D.Quass, J.McHugh, J.Widom, and J.Wiener, *The Lorel Query Language for Semistructured Data*, International Journal on Digital Libraries, 1(1):68-88, April 1997.

[5] S.Abiteboul, *Query Semi-structured Data*, ICDT, 1997.

[6] J.Clark and S.Derose, *XML Path Language(XPath) Version 1.0*, World Wide Web Consortium, http://www.w3.org/TR/xpath.

[7] T.Bray, J.Paoli, C.M.Sperberg-McQueen, and E.Maler, *Extensible Markup Language(XML) 1.0(Second Edition)* ,W3C Recommendation, http://www.w3.org/TR/REC-xml.

[8] S.Derose, E.Maler, and D.Orchard, *XML Linking Language(XLink), version 1.0*, W3C Recommendatio, http://www.w3.org/TR/xlink.

[9] R.Goldman and J.Widom, *Dataguides: Enabling Query Formulation and Optimization in Semistructured Databases*, VLDB, 1997.

[10] J.McHugh, J.Widom, S.Abiteboul, Q.Luo and A.Rajamaran, *Indexing Semistructured Data*, Technical Report, Stanford University, January 1998.

[11] T.Milo and D.Suciu, *Index Structures for Path Expressions*, ICDT, 1999.

[12] R.Kaushik, P.Shenoy, P.Bohannon and Ehud Gudes, *Exploiting Local Similarity for Efficient Indexing of Paths in Graph Structured Data*,ICDE, 2002.

[13] N.Polyzotis, M.Garofalakis, *Statistical Synopses for Graph-Structured XML Databases*, SIGMOD, 2002.

[14] N.Polyzotis, M.Garofalakis, *Structure and Value Synopses for XML Data Graphs*, VLDB, 2002.

[15] M.Henzinger, T.Henzinger, and P.Kopke, *Computing Simulations on Finite and Infinite Graphs*, FOCS, 1995.

[16] R.Paige and R.Tarjan, *Three Partition Refinement Algorithms*, SIAM Journal of Computing, 16:973-988, 1987.

[17] R.Kaushik, P.Bohannon, J.F.Naughton, and P.Shenoy, *Updates for Structure Indexes*, VLDB, 2002.

[18] P.Buneman, S.B.Davidson, M.F.Fernandez, and D.Suciu, *Adding Structure to Unstructured Data*, ICDT, 1997.

[19] T.Milo and D.Suciu, *Optimizing Regular Path Expressions Using Graph Schemas*, ICDE, 1998.

[20] M.Roggenbach and M.Majster-Cederbaum, *Towards A Unified View of Bisimulation: A Comparative Study*, Theoretical Computer Science, 238(1-2):81-130, May 2000.

[21] C.Zhang, J.Naughton, D.Dewitt, Q.Luo, and G.Lohman, *On Supporting Containment Queries in Relational Database Management Systems*,ACM SIGMOD, 2001.

[22] Q.Li and B.Moon, *Indexing and Querying XML Data for Regular Path Expressions*, VLDB, 2001.

[23] B.Cooper, N.Sample, M.J.Franklin, G.R.Hjaltason, and M.Shadmon, *A Fast Index for Semistructured Data*, VLDB, 2001.

[24] R.Kaushik, P.Bohannon, J.F.Naughton and H.F.Korth, *Covering Indexes for Branching Path Queries*, ACM SIGMOD 2002.

[25] Chin-Wan Chung, Jun-Ki Min and Kyuseok Shim, *APEX:An Adaptive Path Index for XML Data*, SIGMOD, 2002.

[26] R.Busse, M.Carey, D.Florescu, M.Kersten, A.Schmidt, I.Mauolescu,and F.Waas, *The XML Benchmark Project*, Available at http://monetdb.cwi.nl/xml/index.html.

[27] NASA is available at http://xml.gsfc.nasa.gov/.