Enabling structural summaries for efficient update and workload adaptation

Qun Chen a,*, Andrew Lim b, Kian Win Ong c

a School of Computer, Northwestern Polytechnical University, Xi’an, China
b Department of Industrial Engineering and Engineering Management, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong
c Department of Computer Science and Engineering, University of California at San Diego, 9500 Gilman Drive, La Jolla, CA 92093-0114, United States

Received 20 September 2006; received in revised form 4 September 2007; accepted 11 September 2007

Available online 7 October 2007

Abstract

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

© 2007 Elsevier B.V. All rights reserved.

Keywords: Semi-structured data; XML index; Query language; Structural summary; Bisimilarity

1. Introduction

In recent years, eXtensible Markup Language (XML) [1] has become the dominant standard for exchanging and querying documents over Internet. XML is an example of semi-structured data [2,3]. It does not conform to the traditional relational or object-oriented model. Instead, its underlying data model is a labeled graph. An XML document consists of hierarchically nested elements, which can be either atomic (for instance, raw character data) or composite (for instance, a sequence of nested subelements). The tags stored with elements describe the semantics of data. The references between elements can be established through id/idref attributes or Xlink constructs [1,4].

* Corresponding author.

E-mail addresses: chenbenben@nwpu.edu.cn (Q. Chen), iealim@ust.hk (A. Lim), okwin@ucsd.edu (K.W. Ong).
A variety of query languages [2,5–7] have been proposed to query XML and semi-structured data. All these query languages adopt path expressions [8] to traverse data. Note that there have been various techniques proposed for path expression evaluation, which include the Index Fabric approach [33], the holistic join method [31,32] and APEX [35]. The Index Fabric and holistic join methods target only tree-structured data. APEX is a workload-aware index that can adapt its structure to query pattern changes. It makes use of a graph structural summary. Since the naive navigation that scans the whole data is obviously very expensive in term of computational cost, the concept of structural summary [9–12] has been proposed to prune the search space. An index graph consisting of a structural summary along with a mapping from data nodes to index nodes, instead of source data, is used to evaluate path expressions. Similar to APEX, we also employ a structural summary. However, in this paper we make use of the concept of local similarity to make structural summary adaptive to query pattern changes and also study the structural summary updates upon source data changes.

Structural summary is based on the notion of bisimilarity [13,14]. On the graph data model, two nodes are bisimilar iff all label paths into them are the same. Mutually bisimilar nodes form an equivalence class. Structural summary is a collection of equivalence classes. 1-index [11] is an accurate structural summary in that it is both safe and sound. Path expressions can be evaluated on 1-index and evaluation results can be directly retrieved without referring to source data. Unfortunately, 1-index summaries usually remain big, hence can not achieve satisfying evaluation performance in most cases. Exploiting the observation that long and complex paths tend to contribute disproportionately to the complexity of 1-index, A(k)-index [12] relaxes the equivalence condition and considers only the incoming paths of length \( \leq k \). Experiments have demonstrated that A(k)-index achieves substantially reduced index sizes. However, A(k)-index is only an approximate summary. For the path expressions longer than \( k \), an additional validation process on source data is required to filter out false positives.

The performance of A(k)-index depends on the value of \( k \). A large \( k \) tends to result in a big index size. A smaller \( k \) may substantially reduce the index size; but at the same time, more queries trigger the computationally expensive validation process. The key observation exploited by our new index proposal is that not all structures are of equivalent significance. Some elements may be never or seldom queried. Refining their equivalence classes obviously results in no visible improvement in querying performance. Even for frequently queried elements, their structural complexities differ. Some may be accessed by short path expressions, others by longer ones. Because of their static nature, neither 1-index nor A(k)-index can handle such diverse access patterns optimally.

In this paper, we propose D(k)-index, an adaptive structural summary. Instead of setting a fixed local similarity for all equivalence classes, D(k)-index adopts different but effective local similarities for them according to their specific access patterns. Compared with its predecessors, D(k)-index achieves higher query performance because of its query load sensitivity. It also enables efficient update operations, which are crucial to real applications but have not been adequately addressed in previous literatures. Our major contributions are summarized as follows:

1. We propose D(k)-index and present its efficient construction algorithms. Unlike previous proposals, D(k)-index takes advantage of query load information to optimize its index structure accordingly.
2. We present efficient algorithms to update D(k)-index upon changes in source data and query load. D(k)-index accommodates these changes through adjusting the local similarities of affected index nodes, thus avoids the potentially expensive propagate partitioning method proposed for updating 1-index and A(k)-index.
3. We demonstrate through extensive experiments that D(k)-index is a more effective structural summary than previous static ones. It achieves reduced index size and improved evaluation performance. Various update operations on it can also be executed more efficiently.

Note that the core idea of D(k)-index has been published in [15]. The major technical extensions of this paper include:

1. Besides the basic index construction algorithm, we present a more sophisticated, greedy method to optimize D(k)-index according to a specific query load.
Besides the subgraph and edge additions, we consider updating $D(k)$-index upon a comprehensive set of update operations supposed to be applied on XML documents.

The experimental evaluation is more elaborate. Besides comparing the query and update performance of $D(k)$-index and $A(k)$-index, we study their query performance degradations upon a sequence of incremental updates and analyze their maintaining processes.

The remainder of this paper is organized as follows. In Section 2, we introduce the data model and some basic concepts, i.e. path expression and local similarity. We proceed to describe $D(k)$-index and its construction algorithms in Section 3. Updating $D(k)$-index is discussed in Section 4. Section 5 presents the promoting and demoting processes that adjust $D(k)$-index to changes in query load. Experimental evaluations are presented in Section 6. Section 7 reviews the related work on structural summary and path expression evaluation techniques. Finally, we conclude this paper with some future research suggestions in Section 8.

2. Preliminaries

We model XML and other semi-structured data as a directed and labeled graph. Each node has a label and a unique identifier, with text elements given a distinguished label, VALUE. There is also a single root element with the distinguished label, ROOT. The edges indicate element–subelement or element–value relationships. The structure of an XML document is basically a tree. The presence of references makes it a graph. In Fig. 1, a portion of an XML document about movies is represented as a data graph. Tree edges, shown as solid lines, represent containment relationships between nodes. Non-tree edges, shown as dashed lines, represent reference relationships. In this paper, we do not differentiate between these two types of edges but treat both as normal edges.

We now introduce the terminologies of paths and path expressions. A node path in a data graph $G$ is a sequence of nodes $v_1v_2\cdots v_p$ such that an edge exists between nodes $v_i$ and $v_{i+1}$ for $1 \leq i \leq p - 1$. A label path is a sequence of labels $L_1L_2\cdots L_p$. A node path matches a label path if $\text{Label}(v_i) = L_i$ for $1 \leq i \leq p$. A label path $P$ matches a node $v$ in $G$ if there exists a node path into node $v$ matching $P$.

A regular path expression, $R$, is defined in terms of sequence($.$), alternation($\mid$), repetition($\ast$) and optional expression($?$) as follows:
in which the symbol _ matches any label in G. We denote the regular language specified by R as L(R). R matches a node v if a label path for some word in L(R) matches v. The result of evaluating R in G is the set of nodes matching R. For example, the path expression, director.movie.title, matches nodes \{15,16,18\} in Fig. 1. The path expression, movie\_DB._\_\_?.movie.actor.name, which identifies the names of actors in movies, is more complicated. The optional _ allows the query to ignore irregularities in G. As a result, node movie can appear directly after movie\_DB or be a child of any node whose parent is movie\_DB. The expression matches nodes \{12,22\} in Fig. 1.

The idea of structural summary is to preserve paths of a data graph in a summary graph but with a far smaller size. If we associate an index node in a summary graph with a set of data nodes, it is possible to evaluate path expressions on summary graph instead of data graph. We denote the index graph for G as IG. The result of evaluating a path expression P in IG is the union of the extents of the index nodes matching P. It is required that the mapping from data nodes to index nodes be safe: if P matches node v in G, it must also match index node V in IG for which v ∈ extent(V). This condition guarantees that the result of evaluating P in G be contained in the result of evaluating P in IG. IG is said to be sound if the converse holds: if P matches index node V in IG, it matches every data node in extent(V).

Structural summary is based on the notion of bisimilarity [11,16].

**Definition 1.** Two data nodes u and v in G are bisimilar(u \(\approx\) v) iff

1. u and v have the same label;
2. if u has a parent u′, then v has a parent v′ such that v′ \(\approx\) u′, and symmetrically for v.

According to the above definition, bisimilarity is a symmetric binary relation. It is easy to prove by induction that two data nodes are bisimilar iff all label paths matching them in G are the same. In Fig. 1, nodes 7 and 10 are bisimilar. But nodes 7 and 9 are not bisimilar because node 7 has a parent labeled actor but node 9 has not.

3. **D\((k)\)-index**

3.1. **Introduction to D\((k)\)-index**

An accurate structural summary is constructed through creating an index node for each equivalence class. An edge is inserted from index node V to U in IG if there is an edge v → u in G with v ∈ extent(V) and u ∈ extent(U). Such index structure is referred as 1-index. Its size never exceeds that of its data graph. 1-index can be constructed in O(mlgn) time using Paige and Tarjan’s algorithm [14], in which n is the number of nodes and m is the number of edges in G.

Based on 1-index, A\((k)\)-index [12] takes advantage of the local similarity concept to reduce its index size.

**Definition 2.** k-bisimilarity(\(\approx^k\)) is defined inductively: for two data nodes u and v:

1. u \(\approx^0\) v iff u and v have the same label;
2. u \(\approx^k\) v iff
   a. u \(\approx^{k-1}\) v;
   b. For every parent u′ of u, there is a parent v′ of v such that v′ \(\approx^{k-1}\) u′;
   c. For every parent v′ of v, there is a parent u′ of u such that u′ \(\approx^{k-1}\) v′.

A\((k)\)-index possesses the following properties:

1. If u \(\approx^k\) v, the set of label paths of length p (p \(\leq k\)) matching u or v in G is the same.
2. A label path of length p (p \(\leq k\)) matching an index node V in IG matches all data nodes in extent(V).
(3) $A(k)$-index is safe.
(4) $A(k)$-index is sound for path expressions of length $p$ ($p \leq k$).

Built on $A(k)$-index, $D(k)$-index takes query pattern irregularities into consideration. Given a query load, the local similarity requirements of different data nodes may differ. In Fig. 1, if users are only interested in the names of actors and directors, regardless of the movies they direct or act in, 1-bisimilarity is sufficient for the name-labeled index nodes. If users want to ask for the titles of movies directed by specific directors, the title-labeled index nodes are required to comply with 2-bisimilarity. Assuming the uniformity of access patterns, $A(k)$-index fails to adapt to this scenario. The core improvement of $D(k)$-index over $A(k)$-index is that its index nodes can have different local similarities. As a result, it can flexibly tailor its structure to different structural patterns present in a query load.

We first prove a theorem establishing $D(k)$-index as a correct structural summary.

**Theorem 1.** Given a node path $N_1 N_2 \cdots N_p$ in $I_G$ and $\text{Label}(N_i) = L_i$ ($1 \leq i \leq p$), if index node $N_i$ (for $1 \leq i \leq p$) is at least $(i - 1)$-bisimilar, the label path $L_1 L_2 \cdots L_p$ matches all data nodes in $\text{extent}(N_p)$ in $G$.

**Proof.** We prove by induction on the length of $P, p$. The basic case when $p = 1$ is obviously true. Assume that the result is true for $p = t$. Let us now consider the case when $p = t + 1$ and $P = N_1 N_2 \cdots N_t N_{t+1}$. The label path $L_1 L_2 \cdots L_t$ matches all data nodes in $\text{extent}(N_i)$ according to the assumption. Since there is an edge $N_i \rightarrow N_{t+1}$ in $I_G$, there should be some data node $v$ in $\text{extent}(N_{t+1})$ in $G$ whose parent is some data node $u$ in $\text{extent}(N_i)$. Because the label path $L_1 L_2 \cdots L_t$ matches data node $u$, the label path $L_1 L_2 \cdots L_t L_{t+1}$ matches data node $v$. Since the data nodes in $\text{extent}(N_{t+1})$ are at least $t$-bisimilar, the label path $L_1 L_2 \cdots L_t L_{t+1}$ matches them all.

Let $K(N_i)$ denote index node $N_i$’s local similarity in $I_G$. According to Theorem 1, if $I_G$ satisfies $K(N_i) \geq K(N_j) - 1$ for any two directly connected index nodes $N_i \rightarrow N_j$, a path expression $P$’s evaluation result in $I_G$ $N_1 N_2 \cdots N_p$ is sound as long as $K(N_p) \geq p - 1$. We call such index structure $D(k)$-index.

**Definition 3.** $D(k)$-index is a structural summary $I_G$ satisfying $K(N_i) \geq K(N_j) - 1$ for any two directly connected index nodes $N_i \rightarrow N_j$.

According to Definition 3, 1-index and $A(k)$-index are both special cases of $D(k)$-index. Two important properties of $D(k)$-index are presented below. Their proofs should be obvious from $D(k)$-index definition and Theorem 1.

(1) $D(k)$-index is safe.
(2) $D(k)$-index is sound for a path expression $P$ of length $p$ if every matching node path of $P$ in $I_G$, $N_1 N_2 \cdots N_{p+1}$, satisfies $K(N_{p+1}) \geq p$.

**Algorithm 1.** Target local similarity broadcast algorithm

(1) Sort the target local similarities of all labels in $I_G$, $k_1 > k_2 > \cdots > k_t$, and for each $k_i$, store the list of labels with target local similarity of $k_i$.
(2) For each $k_i$, $1 \leq i \leq t$, repeat:
   - For each label in the list of $k_i$, update the target local similarities of its parents in $I_G$: if it is less than $(k_i - 1)$, reset to $(k_i - 1)$;
   - Update the label list of $(k_i - 1)$.

3.2. Construction

3.2.1. Basic approach

The basic algorithm assumes that every label in an index graph have been assigned a target local similarity. It begins with the simplest structural summary, the label-split graph, and repeatedly refines its index nodes.
until their local similarities reach target values. Note that the label-split graph is a special \( D(k) \)-index with all index nodes’ local similarities equal to 0.

We use \( k_{\text{max}} \) to denote the maximal local similarity any index node can have on a \( D(k) \)-index. If the maximal length, \( p_{\text{max}} \), of path expressions querying a label is less than \( k_{\text{max}} \), this label’s target local similarity is set to be \( p_{\text{max}} \). Otherwise, it is set to be \( k_{\text{max}} \). The target local similarities of index nodes are also constrained by \( D(k) \)-index structural requirements. Suppose that two directly connected nodes \( N_i \rightarrow N_j \) in \( I_G \) have their target local similarities, \( K(N_i) = 0 \) and \( K(N_j) = 2 \), as determined by a query load. According to the \( D(k) \)-index definition, \( K(N_i) \) should be reset to 1. We propose a broadcast algorithm to correctly reset all target local similarities. Its details are described in Algorithm 1. It takes \( O(m) \) time, in which \( m \) is the number of edges in \( I_G \).

The index nodes on a \( D(k) \)-index are refined using the approach proposed in [12]. For an index node \( U \) in \( I_G \), let \( \text{Succ}(U) \) denote the set of successors of \( U \)’s data nodes in \( G \), i.e. the set \{v| there is an edge \( u \rightarrow v \) in \( G \) with \( u \in \text{extent}(U) \}). Given two index nodes \( U \) and \( V \) in \( I_G \), \( V \) is stable with respect to \( U \) iff \( \text{extent}(V) \) is a subset of \( \text{Succ}(U) \) or \( \text{extent}(V) \) and \( \text{Succ}(U) \) are disjoint. Note that \( V \) can be made stable with respect to \( U \) through splitting \( V \) into \( V \cap \text{Succ}(U) \) and \( V - \text{Succ}(U) \). The basic construction algorithm is described in Algorithm 2. Beginning with \( k = 1 \), it repeatedly computes \( k \)-bisimulation equivalence classes from \( (k/0) \)-bisimulation equivalence classes. It takes \( O(k_{\text{max}}m) \) time in the worst case, in which \( m \) is the number of edges in \( G \) and \( k_{\text{max}} \) is the maximal target local similarity. A construction example is also shown in Fig. 2.

\[ \text{Algorithm 2. The basic construction algorithm} \]

1. Build the label-split index graph \( I_G \) from \( G \);
2. Use the query load and Algorithm 1 to set target local similarities;
3. For \( k = 1 \) to \( k_{\text{max}} \)
   - For each index node \( N_j \) in \( I_G \)
     - If (its target local similarity is \( \geq k \))
       - For each parent \( N_i \) of \( N_j \) in \( I_G \)
         - Replace \( N_j \) with \( N_j \cap \text{Succ}(N_i) \) and \( N_j - \text{Succ}(N_i) \);
       - Set local similarities of split index nodes to \( k \);
       - Set target local similarities of split index nodes to \( N_j \)’s;

3.2.2. Greedy approach

The basic construction approach suffers two drawbacks. First, it does not relate a label’s target local similarity to its query path’s evaluation cost. The resulting local similarities on \( D(k) \)-index may result in subop-

---

![Fig. 2. \( D(k) \)-index construction example: (1) Label E has a target local similarity of 2, other labels have a target local similarity of 1; (2) Numbers beside the nodes are real local similarities.](image-url)
timal evaluation performance. Secondly, we expect that \( D(k) \)-index size is constrained by the available main memory in real applications. It is difficult to effectively control \( D(k) \)-index size under the basic approach. To overcome these shortcomings, we propose a more sophisticated, greedy approach to construct \( D(k) \)-index.

Instead of fixing labels’ target local similarities from the onset, the greedy approach incrementally refines index nodes in the most cost-effective way until the index size reaches its limit. The cost-effectiveness of a refinement is defined to be the ratio between the evaluation performance improvement and the index size increase. At each step, the algorithm selects the most cost-effective path expression in a query load to be supported by \( D(k) \)-index.

**Algorithm 3.** The greedy construction algorithm

1. For each path \( P_i \) in a query load \( Q \)
   - Record \((L_i, p_i)\), where \( L_i \) is \( P_i \)'s end label and \( p_i \) is the length of \( P_i \);

2. Repeat until \( I_G \) reaches its size limit:
   a. For each \((L_i, p_i)\) satisfying \( L_i \)'s local similarity in \( I_G \)
      - Calculate \( \text{UnitBenefit}(L_i, p_i) = \frac{\text{Benefit}(L_i, p_i)}{\text{Cost}(L_i, p_i)} \), where \( \text{Benefit}(L_i, p_i) \) denotes the estimated evaluation performance improvement as a result of upgrading label \( L_i \)'s local similarity to \( p_i \) and \( \text{Cost}(L_i, p_i) \) denotes the estimated index size increase.

   b. Select the \((L_i, p_i)\) with the maximal \( \text{UnitBenefit} \);
      - For each \( L_i \)-labeled index node in \( I_G \)
        - Upgrade its local similarity to \((p_i - 1)\) (refer to Algorithm 4);

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 \( A(k) \)-index \[12\]. The cost of a query is defined to be the number of nodes visited in the index and data graph. Supposing that the evaluation cost in the index graph is negligible, we measure the evaluation performance improvement by the decrease in the number of data nodes to be validated times the length of query path.

We measure the index size increase by the number of newly created index nodes. If \( G \) is tree-structured, the estimation is straightforward. Suppose that an index node \( V \) has \( t \) parents in \( I_G \). Refining \( V \) according to its parents results in \((t - 1)\) new index nodes in \( I_G \) in the worst case. Now we consider the general case of graph-structured data. For every two labels \( U \rightarrow V \) in the initial label-split index, we record \( V \)'s average number of \( U \)-labeled parents, denoted as \( \text{aver}(U, V) \). Suppose that a \( W \)-labeled index node \( W_1 \) has \( t_1 \) \( U \)-labeled parents and \( t_2 \) \( V \)-labeled parents in \( I_G \). Since each data node in \( \text{extent}(W_1) \) has averagely \( \text{aver}(U, W) \) \( U \)-labeled parents in \( G \) and the maximal number of distribution probabilities of \( \text{aver}(U, W) \) data nodes among \( t_1 \) index nodes is \( C_{t_1}^{\text{aver}(U,W)} \), where \( C \) is the combination function \text{choose}, the total number of index nodes resulting from partitioning \( W_1 \) according to its \( t_1 \) \( U \)-labeled parents is estimated to be \( C_{t_1}^{\text{aver}(U,W)} \) in the worst case. Similarly, the total number of index nodes resulting from partitioning \( W_1 \) according to its \( t_2 \) \( V \)-labeled parents is estimated to be \( C_{t_2}^{\text{aver}(V,W)} \) in the worst case. Assuming the independence between these two distributions, we estimate the number of new index nodes as a result of partitioning \( W_1 \) according to its parents to be \( C_{t_1}^{\text{aver}(U,W)} \times C_{t_2}^{\text{aver}(V,W)} - 1 \).

Extending this estimation method to the case of multiple parent labels is straightforward.

**Algorithm 4.** The single refinement algorithm: Single_Refinement(\( V, k, I_G \))

1. If \( (K(V) \geq k) \) return \( I_G \);
2. For each parent \( U_i \) of \( V \) in \( I_G \)
   - \( I_G = \text{Single_Refinement}(U_i, k - 1, I_G) \);
3. For each parent \( U_i \) of \( V \) in \( I_G \)
   - Split \( \text{extent}(V) \) into \( V \cap \text{Succ}(U_i) \) and \( V \setminus \text{Succ}(U_i) \);
4. Return \( I_G \).
The greedy approach is sketched in Algorithm 3. Note that selecting the most cost-effective query path only refers to \( D(k) \)-index \( I_G \) but not \( G \). This is crucial for the algorithm’s efficiency.

The algorithm of upgrading a single index node \( V \)’s local similarity to \( k \) is sketched in Algorithm 4. It runs in a recursive way.

### 4. Updating \( D(k) \)-index

In [16], two types of update operations upon source data were considered for updating an index: subgraph addition and new edge addition. Subgraph addition represents the insertion of a new document into database. New edge addition represents a small incremental change. In this section, we first present the \( D(k) \)-index update algorithms upon these two basic cases. Then we proceed to demonstrate that our approaches can flexibly accommodate other defined update operations on XML documents.

#### 4.1. Subgraph addition

The \( D(k) \)-index update algorithm for subgraph addition is a variant of the 1-index update algorithm proposed in [16]. Suppose that a new subgraph \( H \) is inserted under the root of a data graph \( G \). It first constructs a \( D(k) \)-index \( I_H \) for \( H \) and adds \( I_H \) as an index subgraph under the root of \( I_G \). Simply treating the new \( I_G \) as a data graph, it then constructs \( I_G \)’s \( D(k) \)-index, which is also \( D(k) \)-index of the updated \( G \). This algorithm’s correctness is established by the following theorem. It is essentially a variant of Theorem 1 in [16].

**Theorem 2.** If \( I_G \) is \( G \)’s \( D(k) \)-index and \( I_0 \) is a refinement of \( I_G \), \( I_G \) is also the \( D(k) \)-index of \( I_0 \).

#### 4.2. Edge addition

It has been shown that a small update in source data may trigger dramatic change in 1-index and \( A(k) \)-index [16]. An edge insertion in \( I_G \) may affect all its descendants in 1-index or its descendants within distance of \( k \) in \( A(k) \)-index. This phenomenon is illustrated in Fig. 3: the propagate method need refine all descendants of the affected index node in \( I_G \). The refinement operation need access relevant data nodes and their edges, is thus computationally expensive. It also involves expensive I/O operations if source data is stored in disk instead of main memory. Instead of refining index nodes, \( D(k) \)-index accommodates small updates in source data through adjusting the affected index nodes’ local similarities. This flexibility enables more efficient update algorithms on \( D(k) \)-index than on its predecessors.

![Fig. 3. 1-index update vs \( D(k) \)-index update.](image-url)
When a new edge from $A$ to $B$ is inserted in $I_G$, the simplest option is to reset $B$’s local similarity to 0. But it can probably be reset to a higher value. For instance, in Fig. 3, the affected index node $D$ has a parent labeled $C$ in the original $I_G$. It means that each data node in $D$ has a $C$-labeled parent in the original data graph. The new edge from $c_3$ to $d_2$ does not change the label set of $d_2$’s parents. Thus $D$’s new local similarity can be set to 1 and its child $E$’s be set to 2. Suppose that a new edge from $U_1$ to $V_1$ is inserted in $I_G$ and $K(V_1)$ is $V_1$’s original local similarity. It is observed that if all label paths of length $k$ ($\leq K(V_1)$) into $V_1$ through edge $U_1 \rightarrow V_1$ match $V_1$ in the original $I_G$, $V_1$’s local similarity can be reset to $k$. The algorithm of determining a maximal local similarity for $V_1$ is described in Algorithm 5.

**Algorithm 5.** The single local similarity resetting algorithm

\[ k_{\text{max}} \]  
the upperbound of $V_1$’s new local similarity

\[ k \]  
$V_1$’s current local similarity

\[ \text{NewPaths}(i) \]  
the set of label paths of length $i$ into $V_1$ through new edge $U_1 \rightarrow V_1$ in $I_G$

\[ \text{OldPaths}(i) \]  
the set of label paths of length $i$ into $V_1$ in the original $I_G$

\[ S_d(P) \]  
the set of starting nodes of node paths into $V_1$ matching $P$ through $U_1 \rightarrow V_1$ in $I_G$

\[ S_o(P) \]  
the set of starting nodes of node paths into $V_1$ matching $P$ in the original $I_G$

1. \( k_{\text{max}} = K(V_1), \quad k = 0 \);
2. \( \text{NewPaths}(1) = \{\text{Label}(U_1)\}, \quad S_o(\text{Label}(U_1)) = \{U_1\} \);
3. \( \text{OldPaths}(1) = \{L|L \text{ is the label of } V_1\’\text{’s parent in the original } I_G\}, \quad S_d(\text{Label}(U_1)) = \{U_i | U_i \rightarrow V_1 \text{ and Label}(U_i) = \text{Label}(U_1) \text{ in the original } I_G\} \)
4. While \( (k \leq k_{\text{max}}) \)
   - if \( (\text{NewPaths}(k + 1) \subseteq \text{OldPaths}(k + 1)) \)
     - \( k = k + 1 \);
     - \( \text{OldPaths}(k) = \text{NewPaths}(k) \);
     - For (each \( P \) in \( \text{NewPaths}(k) \))
       - For each \( W_i \) in \( S_o(P) \)
         - For each parent \( X_i \) of \( W_i \) in \( I_G \), insert \( P’ = \text{Label}(X_i) + P \) into \( \text{NewPaths}(k + 1) \) and insert \( X_i \) into \( S_d(P’) \);
       - For each \( W_i \) in \( S_o(P) \)
         - For each parent \( X_i \) of \( W_i \) in the original \( I_G \), insert \( P’ = \text{Label}(X_i) + P \) into \( \text{OldPaths}(k + 1) \) and insert \( X_i \) into \( S_o(P’) \);
   - Else Return \( k \);

After resetting $V_1$’s local similarity, the update algorithm need broadcast this update in the breadth-first order to $V_1$’s descendants in $I_G$. An index node with distance $d$ from $V_1$ should lower its local similarity to $(k + d)$ if it is originally larger than $(k + d)$. In the worst case, Algorithm 5 involves all nodes within distance $k$ to $V_1$ and the following broadcast algorithm involves all nodes within distance $(K(V_1) - k)$ from $V_1$ in $I_G$. Since the update on $D(k)$-index does not refer to source data, it is expected to be much more efficient than its counterparts on 1-index and $A(k)$-index. We will validate our claims empirically in the experimental evaluation section.

### 4.3. Update operations upon XML

In recent years, there were a few proposals for updating XML documents, such as XUpdate [18], the work in [17], XML-RL Update Language [19] and XQuery Update Facility [20]. Even though these proposals differ in details to some extent, they share more similarities and all include three basic functionalities: insert, delete and replace. In this paper, we choose the primitive update operations defined in [17] as our example operations applied on XML documents because in term of structural summary updating, they represent currently defined XML update operations well.
The paper [17] uses the term object to refer to an XML component, which can be an element, an attribute, an IDREF or a PCDATA content, and assumes the presence of tuples of references to selected objects within XML documents through a path expression matching operation. Six primitive XML update operations are defined as

1. **Delete(child)**: if child is a member of the target object, it is removed;
2. **Insert(content)**: it inserts a new content into the target object;
3. **Rename(child, name)**: if child is a non-PCDATA member of the target object, it is renamed.
4. **InsertBefore(ref, content)**: it is defined for ordered execution and inserts a new content directly before the target ref;
5. **Replace(child, content)**: it replaces child with content;
6. **Sub-Update(patternMatch, predicates, updateOp)**: it uses a path expression to select target objects, returns bindings filtered by predicates and recursively invokes the update operation updateOp.

Note that **InsertBefore(ref, content)** is not different from **Insert(content)** operation in term of D(k)-index update. **Replace(child, content)** can be considered to be equivalent to an **Insert(content)** operation followed by a **Delete(child)** operation. Therefore, it is sufficient to address the D(k)-index update upon three operations: **Delete, Insert and Rename**.

Before that, we consider the D(k)-index update upon another type of incremental change: edge deletion. Suppose that the edge \( u \rightarrow v \) is deleted from the original data graph \( G \) and \( u \) and \( v \)'s corresponding index nodes are \( U_1 \) and \( V_1 \) in \( I_G \) respectively. If \( v \) is still connected with some data node in \( extent(U_1) \), \( V_1 \)'s local similarity remains unchanged. Otherwise, it needs to be reset. It is observed that if all label paths of length \( k \) (\( k \leq K(V_1) \)) into \( V_1 \) through \( U_1 \rightarrow V_1 \) match \( V_1 \) in \( I_G \) without edge \( U_1 \rightarrow V_1 \), \( V_1 \)'s local similarity can be reset to \( k \). A minor variant of **Algorithm 5** can be used to determine \( V_1 \)'s maximal new local similarity.

### 4.3.1. Delete(child)

It amounts to the edge deletion operation if child is a reference. Otherwise, since it is assumed that a single element can only be deleted after all its attributes, nested subelements and edges initiating from it are deleted, **Delete(child)** only requires removing child's data nodes from the extents of their corresponding index nodes. The affected index nodes’ local similarities remain unchanged.

### 4.3.2. Insert(content)

It amounts to the edge insertion operation if content is a reference. Otherwise, a new index node \( V \) is created for content in \( I_G \). Its extent contains the only data node and its local similarity is set to be \( K(V_p) + 1 \), in which \( V_p \) is \( V \)'s parent in \( I_G \).

### 4.3.3. Rename(child, name)

Suppose that data node \( v \) in \( extent(V_i) \) is renamed as \( L \). The D(k)-index update operation consists of two steps:

1. It creates a new index node \( L \) labeled \( L \) for \( v \) and assigns \( 1 + \min\{K(U_1), K(U_2), \ldots, K(U_t)\} \) as its local similarity, in which \( U_j (1 \leq j \leq t) \) is \( L \)'s parent in \( I_G \);
2. Reset the local similarities of \( L \)'s descendants in \( I_G \).

At step two, we first reset the local similarities of \( L \)'s child nodes and then broadcast the updates to its other descendants. Suppose that \( X_i \) is \( L_i \)'s child. It is observed that if all label paths of length \( k \) (\( k \leq K(X_i) \)) into \( X_i \) through \( V_i \) or \( L_i \) in \( I_G \) match \( X_i \) in the original \( I_G \), \( X_i \)'s local similarity can be reset to \( k \). If only resetting \( X_i \)'s local similarity is concerned, the **Rename** operation amounts to an edge deletion operation (from \( v \) to some data node in \( extent(X_i) \)) followed by an edge insertion operation (from renamed \( v \) in \( extent(L_i) \) to some data node in \( extent(X_i) \)). Similarly, a minor variant of **Algorithm 5** can be applied to determine \( X_i \)'s maximal new local similarity.
5. Adjusting $D(k)$-index

As more update operations are performed on $D(k)$-index, its index nodes’ local similarities are supposed to decrease gradually. At the same time, changes in query load may demand higher local similarities for some index nodes. Both cases can result in significant degradation of $D(k)$-index query performance. On the other hand, changes in query load may render high local similarities unnecessary for some index nodes. It is important that $D(k)$-index adjust itself accordingly to update operations and changing query load. In this section, we propose promoting/demoting procedures to upgrade/degrade index nodes’ local similarities in $D(k)$-index. They are supposed to be executed periodically to tune $D(k)$-index structure for high query performance.

5.1. Promoting process

The process to promote a single index node’s local similarity has been described in Algorithm 4. In real applications, there are usually a batch of index nodes that need to be promoted. In such case, the promotion is conducted in an orderly way:

1. Record each to-be-promoted index node’s target local similarity, and set its ancestors’ target local similarities correspondingly;
2. Beginning with the lowest target local similarity, iteratively promote the corresponding index nodes until their local similarities reach target values.

For instance, suppose that there is a path $A_1 \rightarrow B_1 \rightarrow C_1$ in $I_G$ and the three nodes have the same local similarity of 3. $B_1$ and $C_1$ are to be promoted to 5. At the first step, we record $(A_1, 4), (B_1, 5), (B_1, 4)$ and $(C_1, 5)$. At the second step, $A_1$ and $B_1$ are promoted to 4. The resulting index nodes $B_1$ and $C_1$ are then promoted to 5. Compared with random one-by-one promotion, the ordered promotion reduces over-promotion cases: if $B_1$ is first promoted to 5, $C_1$ would be unnecessarily partitioned according to $B_1$’s sub-nodes with the local similarity of 5. Over-promotion is undesirable because it increases index size.

The promotion can also be performed in the greedy approach described in Section 3.2.2. At each step, it always chooses the most cost-effective label to promote. Its details should be the same as previously presented, thus omitted here.

5.2. Demoting process

The demoting process downgrades index nodes’ local similarities. Lower local similarities make it possible to merge index nodes of the same label, thus achieve smaller index size.

Theorem 2 in Section 4.1 states that a $D(k)$-index $I_G$ can be reconstructed from its refinement. Given lower local similarities, since the current $D(k)$-index $I'_G$ is a refinement of the target $I_G$, $I_G$ can be constructed from $I'_G$ by treating $I'_G$ as a data graph.

6. Experimental study

In this section, we perform extensive experiments to validate the efficiency of $D(k)$-index, as well as investigate how its performance varies with changes to source data and query workload. For comparative purposes, we choose to measure the relative performance of other indexes based on bisimilarity which can support the same class of queries on graph-structured data. We compare $D(k)$-index with the previously proposed $A(k)$-index, since the latter has been shown to outperform 1-index. Our experiments illustrate that $D(k)$-index can:

1. Achieve comparable or higher query performance with a smaller index size;
2. Be updated efficiently by deferring propagation to a maintenance procedure;
3. Achieve reasonable query performance, even when propagation is deferred for data updates, or when there are changes in the query workload the index specializes for;
4. Be periodically maintained to preserve its query performance.
We perform our experiments on two datasets:

1. **XMark**: This is a synthetic dataset from an XML benchmark suite [21], which produces XML documents modeling the activities of an auction site. Using the benchmark suite, we generate an XML file of about 100 MB.

2. **NASA**: This dataset is based on a DTD designed by the XML Group at the NASA Goddard Space Flight Center, originally intended to publish data and metadata collated by the Astronomical Data Center. To obtain a sizable dataset, we use the IBM XML Generator to generate a synthetic XML document of about 100 MB.

All experiments are implemented in C++, and executed under Linux 2.6 kernel. The machine used has a Pentium 4 2.8 GHz processor and 512 MB RAM.

### 6.1. Query performance

Due to lack of an external memory cost model for graph-structured data querying, we assume that both index graph and source data reside in primary memory during execution. Previously [15], we used the number of nodes traversed on data and index graphs to measure query performance. We choose to present our results here using elapsed CPU time instead, so that relative comparisons can be made between query, update and maintenance procedures. At any rate, since the previous metric is an accurate measure of elapsed time, there is no difference in the characteristics of the graphs.

For each dataset, we randomly generate a query workload consisting of 100 distinct paths with lengths evenly distributed between 1 and 5. A random label is first chosen among all possible labels in the DTD, a random data node of that label is then selected. From the data node, we obtain a random path in the data graph of the specified length. We thus avoid a bias on frequently occurring labels, obtaining a query workload that is a representative mixture of both selective and unselective paths.

We present two flavors of \textit{D}(k)-index. The \textit{max}-\textit{k} variant has a parameter \textit{k} analogous to that of \textit{A}(\textit{k})-index. For a given workload, the maximal position of each label within the query paths is noted. Within the index, each label is assigned a local similarity equal to this maximal position or \textit{k}, whichever is smaller. The \textit{max}-\textit{k} variant with \textit{k} = 5, denoted \textit{D}(5), for example, does not require any validation since paths have maximum length 5. On the other hand, the index support level, or \textit{isl}, variant is constructed with the greedy heuristic. The index is iteratively refined until a certain percentage of the query paths can be evaluated without validation. \textit{D}[70\%], for example, can evaluate at least 70% of paths without accessing the data graph.

For both \textit{A}(\textit{k})-index and the \textit{max}-\textit{k} variant, we let \textit{k} range between 0 and 5. The \textit{isl} variant has its percentage ranging 65–95% in 5% increments. Query performance on the XMark and NASA datasets are presented in Fig. 4.

The time needed to evaluate queries is plotted against the size of index graph. This explicitly represents the space vs time trade-off for the choice of a suitable index. As we can see, this trade-off is not linear – there are diminishing returns on performance when we allow more information to be stored in the indexes. For \textit{A}(\textit{k})-index, intuitively there is more commonality in the sub-structures when \textit{k} is small. As \textit{k} increases, the sub-structures diverge, thereby requiring more space to be indexed. Unsurprisingly, for any value of \textit{k}, the \textit{max}-\textit{k} variant will always equal or better the corresponding \textit{A}(\textit{k})-index. Whereas \textit{A}(\textit{k})-index only makes use of the (estimated) path lengths in the workload, \textit{D}(\textit{k})-index further notes the maximal position of each label in the query load. For a given workload, this allows us to avoid allocating extraneous space for higher local similarities that would not be taken advantage of during query evaluation.

In comparison, the \textit{isl} variant achieves an even faster convergence behavior by recognizing that not all query paths are equally cost-effective to be supported. At each iteration during construction, the heuristic estimates the most cost-effective path in terms of the potential savings in validation over potential increases in size. This strives for maximum bang for the buck in a fairly straightforward manner. Unlike the \textit{max}-\textit{k} variant which requires the user to have some knowledge of the nature of the workload (e.g. typical length of query paths), the \textit{isl} variant also has the benefit of admitting a more automatic, rule of thumb administration (e.g. 80–20 rule).
It is possible to milk even more performance by further noting within the workload the parent labels which result in the maximal position of each label, as presented in the $M(k)$-index structure [22]. Such an evaluation of the query performance under unchanging source data and a static workload represents the best case scenario for $D(k)$-index. In general, query performance varies when data updates or workload changes occur, as we will explore in the following subsections. Regardless, it is likely that the static scenario is a common case for certain classes of applications.

Fig. 4. Query performance.
6.2. Update performance

To measure update performance, we insert new edges between the pair of data nodes whose labels compose an ID/IDREF pair in the DTD file. We choose nine ID/IDREF label pairs and generate 3000 edges that are uniformly distributed among these pairs.

We use a variant of the 1-index update algorithm [16] to update \( A(k) \)-index. Suppose that a new edge \( U \to V \) is inserted into \( I_G \) because of an edge insertion \( u \to v \) in \( G \). The maximal new local similarity \( k_n \) of \( V \) can be determined in the same way as on \( D(k) \)-index. If \( k_n < K(V) \), the algorithm partitions \( v \) and its ancestors within distance \( (k - k_n) \) into new index nodes in \( I_G \). It is straightforward that the \( k \)-bisimilarity property will be preserved on the resulting index.

In addition to what has been presented in [15], we introduce several optimization techniques in this paper for \( A(k) \)-index update such that a fairer comparison can be made between \( A(k) \) and \( D(k) \). Note that the CPU time of determining an index node’s maximal local similarity is linear with the number of edges in \( I_G \) in the worst case. Our experiments show that the process of checking both \( V \)'s and all its ancestors’ maximal local similarities is computationally expensive. Instead, the algorithm chooses to only check \( V \)'s while partitioning all other index nodes regardless. This option is justified by the fact that in our experiments on XMark data, the \( A(k) \)-index update performance is improved by an order of magnitude. Secondly, the algorithm associates a counter, which represents the number of edges between data nodes in \( \text{extent}(U) \) and \( \text{extent}(V) \), with each edge \( U \to V \) in \( I_G \). While updating the connectivity between \( U \) and \( V \), the algorithm decrements the counter correspondingly once a data node is split off. It saves the process of rescanning the edges between data nodes in \( \text{extent}(U) \) and \( \text{extent}(V) \). Our experiments show that this technique speeds up \( A(k) \)-index update by a factor of 5.

Fig. 5 plots the update time against the query time after update. It represents a classical trade-off between query and update. \( A(0) \), with the worst query performance, also takes the least time to update. The \( A(k) \)-index update cost increases significantly as \( k \) becomes larger. \( D(k) \)-index obviously performs better than \( A(k) \)-index. Moreover, its update performance is independent of parameters and remains roughly unchanged.

In our experiments, it is assumed that source data resides entirely in main memory. It is worthy to emphasize that unlike \( A(k) \)-index, \( D(k) \)-index can be updated without access to source data. For larger data that has to be stored in external memory, the performance difference is supposed to be even starker.

An update algorithm for \( A(k) \)-index with a provable guarantee on the resultant index’s quality has also been proposed in [23]. The guarantee arises from an additional merging phase following index nodes partitioning. Our algorithm instead separates the merging phase from the update operation. This choice is based on two experimental observations: (1) the query cost is dominated by the number of data nodes validated, not the index size; (2) the merging phase is much more expensive than the simple update operation, thus delaying it reduces more update time than the decreased index size it may engender. Our algorithm includes the merging operation as part of the maintaining process, which is only executed after a considerable number of updates.

6.3. Evaluation performance variation with updates

As \( A(k) \)-index is incrementally updated, its query performance may suffer because refinements increase its size. \( D(k) \)-index query performance may deteriorate with updates as well since local similarity adjustments make evaluations trigger more validations.

Our experiment tracks the size increase of \( A(k) \)-index over a sequence of 300 incremental updates, \textit{edge insertion}. Detailed results are presented in Fig. 6. It shows that the index size increases steadily with updates and \( A(k) \)-index with low value of \( k (k \leq 3) \) has the sharper percentage increase than with higher value of \( k (k > 3) \). On XMark data, the size increase after 300 updates amounts to more than 100% when \( k = 2 \) or 3; it is more moderate while \( k \) is larger, 45% for \( k = 4 \) and 13% for \( k = 5 \). On NASA data, the size increase is quite moderate for all values of \( k \). It is roughly between 10% and 20%.

Our experiments also track the evaluation performance variation of \( A(k) \) and \( D(k) \)-index over a sequence of 300 incremental updates. Detailed results are presented in Fig. 7. Note that only \( A(k) \)-indexes with \( 3 \leq k \leq 5 \) are shown in figure. Results on \( A(k) \)-index with \( k = 1 \) and 2 follow the same trend as presented here. Two \( D(k) \)-indexes with \( k_{\text{max}} = 5 \) and \( \text{isl} = 80\% \) respectively are chosen for comparison. It is observed that
$A(k)$-index performance degradations is negligible (less than 5%). While $k$ is small, $A(k)$-index evaluation cost is dominated by validations; even though the index size may increase considerably, the overall cost is roughly the same since the validation cost remains unchanged. While $k$ is large, its size increase percentage becomes smaller; as a result, its evaluation performance does not fluctuate much either. Compared with $A(k)$-index, $D(k)$-index performance degradation is sharper on both data. $D(k)$-index performance on XMark data is visibly worse than that of $A(4)$ or $A(5)$-index after updates. On NASA data, $D(k)$-index performance
deterioration is less dramatic. Since the majority of paths in NASA data do not contain ID/IDREF labels, updates do not affect the evaluation cost of most queries in the workload. Our experiments verify the claim that local similarity downgrading may severely affect $D(k)$-index performance.

6.4. Evaluation performance variation with changing query load

We study $A(k)$ and $D(k)$-index query performance variation upon the changing workload through adding new random queries of length between 1 and 5. Detailed results are presented in Fig. 8. The lines of $D(5)$ Optimal and $D(80\%)$ Optimal denote the achieved query performance if $D(k)$-index is reconstructed from scratch. Since local similarities on $A(k)$-index are uniform, its performance fluctuates up and down randomly with the changing workload. Instead, $D(k)$-index is optimized for a given workload, thus its query performance deteriorates consistently as new queries are added. It is also observed that new $D(k)$-indexes constructed from scratch easily outperform original ones.

6.5. Maintaining indexes

Our study on $D(k)$-index performance variation with incremental updates and changing workload justifies the necessity to maintain $D(k)$-index periodically. $A(k)$-index maintenance process simply merges index nodes of the same label if they satisfy $k$-bisimilarity. $D(k)$-index maintenance process consists of two phases: first promote index nodes and then merge them if possible.

We compare the maintenance performance of $A(k)$-index with $3 \leq k \leq 5$ and $D(k)$ with $k_{max} = 5$. The results are presented in Fig. 9. Not surprisingly, maintaining $D(k)$-index is more computationally expensive than maintaining $A(k)$-index. The relatively higher maintenance cost of $D(k)$-index should not be of much concern because: (1) in real applications, updates should be performed much less frequently than queries; and the maintenance process is invoked only after a considerable number of updates; (2) as demonstrated later, the maintenance operation can significantly improve the evaluation performance of $D(k)$-index; it is quite a fair price to pay.

Fig. 6. Size increase of $A(k)$-index over incremental updates.
The maintenance operation’s effectiveness to improve $D(k)$-index evaluation performance is demonstrated in Fig. 10. On $A(k)$-index, even though the index can be shrunk to some extent, its overall performance remains roughly the same. In contrast, the performance improvement on $D(k)$-index is much more dramatic. Note that on NASA data, the evaluation cost is actually cut by half. The two dots before and after maintenance appear close because of the large value of $Y$-axis.
7. Related work

Three structural summaries have been proposed for graph-structured data, strong DataGuide [9], 1-index [11] and $A(k)$-index [12]. We have already examined 1-index and $A(k)$-index in Section 1. Strong DataGuide interprets a graph data as a non-deterministic automation and obtains an equivalent deterministic automaton. A path expression is then evaluated by matching an exact sequence of nodes in DataGuide. Since a data node
may appear in extents of more than one index node, the size of a strong DataGuide may be exponential with respect to data graph size in the worst case. This exponential behavior makes strong DataGuide inappropriate for complex graph-structured data. It is worth pointing out that $M(k)$-index proposed in [22] is actually a type of $D(k)$-index according to our Definition 3 in Section 3.1. Its major improvement over what we present in this paper is that it further optimizes the index nodes' local similarities by considering not only the labels being queried, but the whole access patterns present in path expressions. Under the $M(k)$-index proposal, index nodes of the same label may have different local similarities. As pointed out in Section 6, $M(k)$-index presents an improved way to construct index structure and can achieve ever higher evaluation performance.

Update algorithms have been proposed to maintain strong DataGuide [9]. However, because $D(k)$-index, based on graph bisimulation, is non-deterministic when translated to automata, these algorithms cannot be generalized to apply in our context. The update algorithms for 1-index were presented in [16]. The authors considered two update cases: document insertion and edge addition. The propagate refinement strategy was proposed to update 1-index incrementally. Although the algorithm for document insertion can be easily generalized to apply in $A(k)$-index, the generalization of the algorithm for edge addition was shown not to be clean. The update algorithm with a provable guarantee on the size of 1-index and $A(k)$-index was also proposed in [23]. It consists of two phases: splitting and merging, in which the splitting phase is essentially the same as what was proposed in [16].

The bisimulation technique comes from the verification research community [24]. It was used to compress the state space graph in a manner that preserves some properties and behaviors of the state space. A similar concept of local bisimilarity, localized stability, has been exploited to build XSketch statistical synopses [25, 26] for graph-structured data. Graph schema [27, 28] is also a structural summary. However, its construction and update algorithms were not discussed. Instead, the authors focused on the structures of different schema and explored their applications on query optimization.

There have been lots of other indexing strategies proposed for evaluating XML documents. Here we review some typical proposals related to structural summaries. The inverted index [29] and the numbering scheme [30] enabled ancestor queries to be answered in constant time. Based on these schemes, efficient algorithms to process path expression queries have also been proposed in [31, 32]. Note that these indexing techniques were designed to handle tree-structured data. Extending them to accommodate graph-structured data is not straightforward and has not been studied. It’s interesting to note that in [34], Kaushik proposed a strategy that combines inverted lists with structural summary to optimize XML path expression evaluation on the tree model. The Index Fabric approach [33] represents every root-to-leaf path on XML tree as a string and indexes them using multiple layers of Patricia Tries. Its shortcoming, as pointed out by [35], is that it is inefficient to process the partial path queries beginning with self-or-descendant axis: they should be first rewritten to simple path expressions beginning at root. Compared with $D(k)$-index, it is based on the assumption of tree data model and does not take advantage of query load to optimize the index structure. In [35], a workload-aware path index, termed APEX, was introduced for XML data. APEX enhances a structural summary with a hash tree to speed up the processing of frequently queried paths. The evaluation algorithm looks up the entries on a hash tree and retrieves the extents of the matching nodes on a summary graph. An incremental algorithm was also presented to adjust APEX upon changing query load. However, the authors did not address the problem.
of updating APEX upon source data change. The APEX work is complementary to our work. $D(k)$-index, as a robust structural summary, can be flexibly adjusted to the changes on both query load and source data. The hash tree approach can be used to speed up the lookup on $D(k)$-index.

8. Conclusion and future work

$D(k)$-index is a clean generalization of previous 1-index and $A(k)$-index structures. It has clear advantages over them. It can adjust its structure to a changing query load. The updates on $D(k)$-index can also be pro-
cessed more flexibly and efficiently. We have experimentally demonstrated that it achieves higher evaluation and update performance.

The future work can be charted on two research fronts. Firstly, we currently assume that both the source data and its index can stay in main memory. The more practical assumption should be that only the index graph is in main memory, but the source data resides in external memory. It is important to investigate the structural summary proposal based on the external-memory cost model. Secondly, the actual queries on XML or other graph-structured databases may involve values and more complicated query patterns (for instance, branch predicates). Extending structural summary to accommodate more general path expressions is, therefore, another interesting research direction. The FleXPath [36] and XFT algebra [37] approaches for integrating the structure and text queries on XML may be helpful to address this problem.

References


Q. Chen et al. / Data & Knowledge Engineering 64 (2008) 558–579


Qun Chen obtained his bachelor degree in Management Information Systems from Tsinghua University of Beijing, China, in 1998. After a brief period working as a software engineer, he began his postgraduate study in Computer Science at National University of Singapore in 1999 and received his Ph.D. degree in 2004. He is currently a professor of School of Computer at Northwestern Polytechnical University, Xi’an, China. His research interests include XML databases, RFID data management and web service technologies.

Andrew Lim obtained his Ph.D. degree in 1992 from the University of Minnesota. From 1992 to 1997, he was a system developer, project leader and consultant to many large private and government organizations in Singapore. From 1997 to 2002, he was an associate professor of Computer Science at the National University of Singapore. At the present moment, he is an associate professor at the Department of Industrial Engineering and Engineering Management in Hong Kong University of Science and Technology. Andrew’s interests include algorithms, software components, framework and architecture for Global Supply Chain management. He has published more than 185 papers in premier international conferences and journals. Professor Lim is also the Director of the HKUST Logistics and Supply Chain forum, and the founding director of the Logistics and Supply Chain Institute (China). He has consulted to a large number of companies in Hong Kong, Singapore and United States.

Kian Win Ong received his bachelors in Computer Engineering from the National University of Singapore in 2003. After working on various XML indexing structures in the Hong Kong University of Science and Technology, he is currently a Computer Science postgraduate student at University of California at San Diego. His research interests are in databases, including optimizations for semi-structured data, novel application frameworks and schema evolution.