Journal of Information Technology and Engineering
Vol. 1 No. 2 (December, 2016)

A New Structural Summary for Graph-Structured XML Data

Jia-Dong REN and Rui MA
College of Information Science and Engineering, Yanshan University, Qinhuangdao 066004, China
E-mail: jdren@ysu.edu.cn ma_rui508@163.com

Received: 23rd July 2016 Revised: 14th October 2016 Accepted: 15th November 2016

Abstract: *With the growing popularity of XML data, research on query processing over XML data is a hot topic. Therefore, many methods based on structural summary have been proposed, which only contain all the path information from the data graph and have less nodes and edges than the original data graph. However, to answer all path queries accurately, the existing structural summaries have large size which blocks the query performance. In this paper, we introduce S(k)-index. Building on the previous works such as D(k)-index and M*(k)-index, our approach is also based on the concept of bisimilarity and allows different index nodes to have different local similarity requirements. S(k)-index also avoids over-refinement and overqualified parent problems from the D(k)-index, and reforms the sequence structure of M*(k)-index with a single structure which avoids not only storage of nodes and edges in components from sequence, but also links between components. Furthermore, efficient update algorithms are also proposed. Experiment results show better performance on size and the query efficiency than the previous structural summaries. In addition, update operations on the S(k)-index can be performed more efficiently.*

Keywords: *XML data, S(k)-index, D(k)-index*

1. INTRODUCTION

With the rapidly increasing popularity of XML [1] for data exchanging and representation, there is a lot of interest in query processing over XML data that conforms to a labeled tree or labeled graph model.

Standard query languages such as Xpath[2] and Xquery[3] for XML and semistructured data have been proposed. Path expressions are the basic building blocks of XML queries. To speed up query processing, structural summary is constructed to summarize the structure of a data graph. Then, we can process path expressions without referring to the original data graph, which may be much bigger than the index structure.

Existing structural summaries are based on the notion of bisimilarity[8]. 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[4] is an accurate structural summary that considers incoming paths up to the root of the whole graph. 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 structure is usually quite large and is considered not efficient enough to speed up the evaluation. The A(k)-index[5] 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 shown to have a substantially small index size. However, the A(k)-index becomes only approximate for paths longer than k and a validation process on original data graph is introduced to extract exact answers.

The D(k)-index[6] gives us a new view. It can adjust the structure according to the different queries, and allows different nodes to have different local similarity. The dynamic property not only controls the size, but also keeps the accuracy. However, there is over-refinement problem which makes the size of index to increase unnecessarily and has an adverse effect on query performance.

M*(k)-index[7] overcomes the limitation above from D(k)-index. It consists of a sequence of component indexes $I_0, I_1 \dots, I_k$ with different similarity, where I_0 is the simplest index graph constructed by label splitting, and I_K maintains the finest partitioning information and is able to answer the relevant query of length up to k accurately. Each index node in component is possibly partitioned in the next component I_{i+1} further into a set of index nodes.

The M*(*k*)-index uses special links to connect relevant nodes in the components in order to evaluate the short path expressions. Figure 1 is an example of M*(*k*)-index, where the extents of index nodes are shown in brackets, and the dashed lines represent the special links across components. M*(*k*)-index has fewer nodes and edges than D(k)-index. But, because of sequence structure, we not only have to store the nodes and edges from each of the components, but also

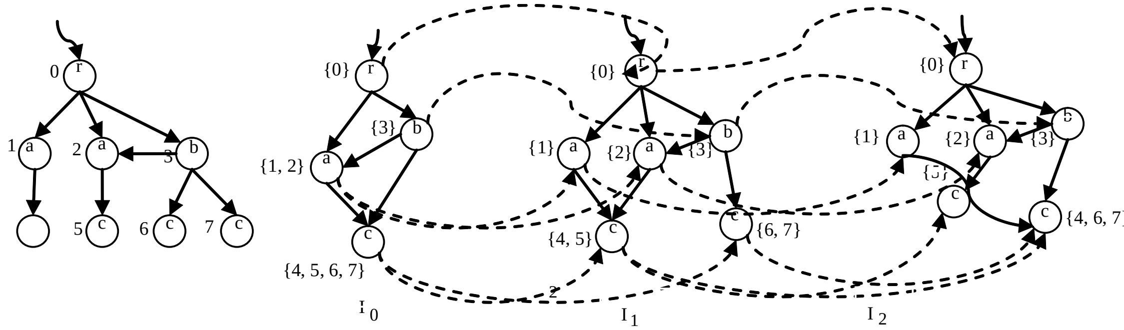

Figure 1: An Example of the M*(k)-index.(//b/a/c)

the links between two components. These storage requirements reduce query performance.

To overcome the problems caused by M*(*k*)-index, S(k)-index is proposed. As an improvement of M*(k)-index, it has a single structure which not only avoids over-refinement and overqualified parent, but also has small storage requirement. We also present efficient algorithms to process the update operations including addition and deletion of edge and subtree.

The rest of this paper is organized as follows. In Section 2, we review some background knowledge. In Section 3, we represent S(k)-index and its construction algorithm. Update operation occurs in Section 4. In Section 5, we report our experiment result. Finally, we conclude the paper in the section 6.

2. BACKGROUND

An XML document can be represented as a labeled direct graph G = { $V_G, E_G, root_G, \Sigma_G$ }. The node from the vertex set V_G has an unique identifier *id* and a literal *label* from Σ_G. The root node is denoted $root_G$. The edge from graph is in the edge set E_G.

A path is a sequence of nodes, such that an edge exists between adjacent nodes. There are two families of paths, label path and node path. A node path $P_V = (V_0 V_1 \cdots V_n)$ is unique, and is an instance of a label path $P_l = (l_0 l_1 \cdots l_n)$, if label ($V_i$) = l_i for each *i*. There are usually multiple node paths that correspond to a given label path. The set of last nodes of the node paths is target set of corresponding label path. For example, the path expression syntax r/a/c from Figure 1returns target set {4, 5}. A complicated path expression such as //b/* involving wildcards * and //, returns target set {2, 6, 7}. Based on the notion of bisimulation[8], summary structure is constructed. The structure is also a labeled directed graph, $I_G = (V_{I(G)}, E_I(G), root_{I(G)}, \Sigma_G)$, which preserves all the label paths in the data graph, and has much fewer nodes and edges. The node in the index graph represents a set of data nodes being bisimilar, which is denoted by the extent of an index node. There is an index edge (u_i, v_i) in $E_{I(G)}$ if and only if a data edge (u_d, v_d) exists in G and $u_d \in u_i$. exent, $v_d \in v_i$. exent.

Definition 1. (Bisimulation) Let G be a data graph in which the symmetric, binary relation ≈, the bisimulation, is defined as: we say that two data nodes u and v are bisimilar (u ≈ 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' ≈ v', and vice versa;

3. S (K)-INDEX

In this section, we present the S(k)-index structure, which has a single structure and also supports different local similarity requirements on different index nodes. The S(k)-index has the same basic property as pervious structures, where v.k is the local similarity, and v.extent is the set of data nodes associated with v.

Property 1 All data nodes in *v.extent* are *v.k*-bisimilar.
Property 2 (*v*, *v'*) $\in E_{I(G)}$ if and only if $\exists o \in$ *v.extent* and $\exists o' \in$ *v'.extent*, such that (*o*, *o'*) $\in E_G$.
Property 3 For all parent v_p of *v* in $V_{I(G)}$, $v_p.k \geq v.k-1$.

The properties above guarantee S(k)-index is precise for a label path expression of length *k*

3.1 Construction Algorithm

Now, we present the S(k)-index construction algorithm. We also begin with the coarsest index structure I_0, built by label splitting. The input *l*, the path expression to be supported; *S*, the target set of *l* in the index structure; and *T*, the target set of *l* in the data graph. *Succ*(*s*) returns all data nodes that are children of data nodes in a set *s*. Unlike the M*(*k*)-index, in the REFINE**procedure, we only build a single structure.

REFINE** (*l*, *S*, *T*)

1: Create a new structure I_0
2: **for** each *v* in *S* **do**
3: REFINENODE** (*v*, *k*, *v.extent* $\cap$ *T*)
4: **while** $\exists v \in I_{length(l)}$ such that *v* has *l* as an incoming path and $v.k <$ *length*(*l*) **do**
5: PROMOTE*(*v*, *length* (*l*))

Similar to REFIENODE* of M*(k)-index, in REFIENODE**procedure, we use the relevant data nodes in *v.extent* to avoid over-refinement, and recursively refine relevant parent nodes.

REFIENODE** (*v, k, relevantData)*

1: // Lines 2–7: recursively refine parent nodes:
2: **if** $v.k \geq k$ **then**
3: Return
4. **for** each parent u of v **do**
5: $predData = Pred(relevantData) \cap u.extent$
6: **if** $predData \neq \phi$ **then**
7: REFINENODE** (u, $k - 1$, *predData*)
8: SPLITNODE** (v, k, *relevantData*)

SPLITNODE**procedure is the key of our algorithm. Instead of *Succ*(*u*), we use *relevantData* to partition the index node v. We reduce the partitioning scope with relevant nodes from query expression. When constructing I_1, we extract the relevant nodes from index nodes in I_0, and get the desired structure on I_1, which appears on I_2 in M*(k)-index. Figure 2 shows the change of Figure 1 based on the procedure.

After the partitioning operation, in order to limit the increase of size caused by continuously querying the XML document, we need merge the relevant nodes. For example, in Figure 3, when path expression *"// a / b"* is evaluated, a_1 、 a_2 and c_4、c_5 in $I_2^{'}$ are two pairs of relevant nodes respectively, so we can merge them into two nodes $a_{\{1,2\}}$ and $c_{\{4,5\}}$ in $I_2^{''}$.

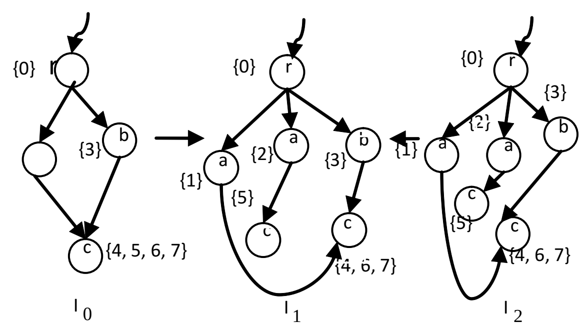

Figure 2: The Result of Partition

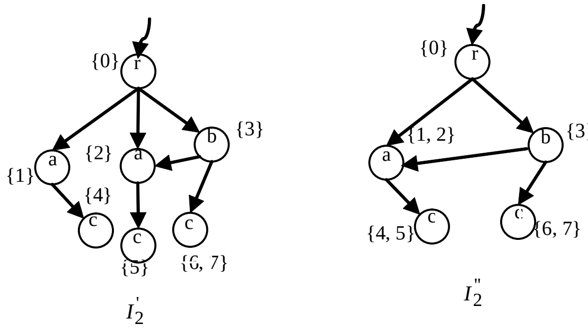

Figure 3: Merge of Relevant Nodes

SPLITNODE** (*v, k, relevantData*)

1: // split *v*:
2: $k_{old} = v.k$
3: $V = \{v\}$
4: **for** each w in V **do**
5: Replace w (in both V and I_1) with w_1 and w_2, where:
6: $w_1.extent = w.extent \cap relevantData$,
7: $w_2.extent = w.extent - relevantData$,
8: $w_1.k = k$
9: $w_2.k = k_{old}$
10: // merge relevant nodes:
11: $remainderExtent = \phi$
12: **for** each w in V **do**
12: **if** $relevantData \cap w.extent \neq \phi$ **then**
13: $remainderExtent = remainderExtent \cup w.extent$
14: Remove w
15: **Add** v_{new} *rest* to I_k, where:
16: $v_{new}.k = k$, and
17: $v_{new}.extent = remainderExtent$

After partition operation in SPLITNODE**procedure, the sequence structure of M*(k)-index only contains two components, and the single structure produced by merge operation can efficiently process the short path expressions. So, we can use the single structure instead of original sequence structure to finish the query processing. Figure 4 shows the new structure, which is improvement of Figure 1.

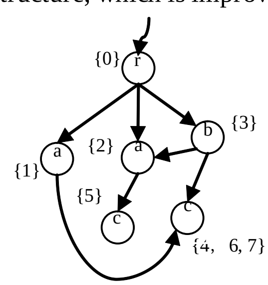

Figure 4: S(k)-Index of Data Graph in Figure 1

3.2 Processing Overqualified Parent Problem

The S(k)-index preserves the properties of the M*(k)-index, which allows different index nodes to have different local similarity requirements, and ensure query processing precise. It also avoids overqualified parent problem, which makes the size large.

In Figure 5, I_1 is a resulting index structure by processing some queries in data graph, where the similariy of b_2 and b_3 are 2. When we continue evaluating query *"// b / c"*, we need to increase the local similarity of the index node *c* from 0 to 1 and get the structure I_2 where similarity of c is 1. However, in I_1, their corresponding data nodes c_4 and c_5 are actually 1-bisimilar, and should have stayed together in one index node with local similarity of 1.

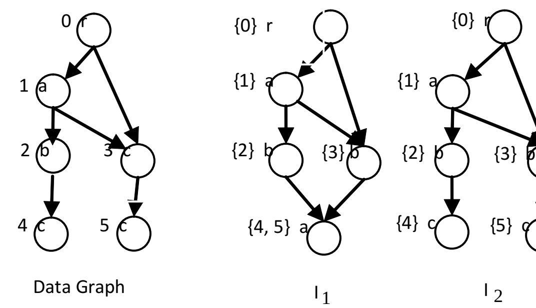

Figure 5: Overqualified Parent Problem

In fact, in I_2 from Figure 5, based on the similarity, b_2、b_3 and c_4、c_5 are two pairs of relevant nodes in the query above. So, we can merge them respectively, and get a smaller size. The structure after merging operation avoids Figure 6 shows the result.

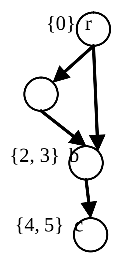

Figure 6: Processing Overqualified Parent

4. S(K)-INDEX UPDATING

Similar to previous theory in [9, 10], we research two kinds of updates, the additions and deletions of a subtree and a new edge. The addition of a subtree represents the insertion of a new file into the XML document, and the addition of a new edge represents the insertion of a new element. All other update operations on the S(k)-index can be built on these two basic cases.

As in [10], update operation consists of a split phase and a merge phase. However, in our algorithms, we only process the relevant nodes from the current query, and guarantee the structure minimal and precise.

4.1 Edge Addition and Deletion

We use a running example to demonstrate the process of update. Firstly, we describe the case of edge addition.

Figure 7(a) is the data graph, where the new edge to be inserted is shown with a dashed line. The index structure before the update is shown in Figure 7 (b). The split phase first checks if there is an index edge in Figure 7 (b) between the two index nodes containing source and end of new data edge. Although there is not the index edge, {1}、{2}in Figure 7(b) are two single nodes, and we don't need to split any index node and only add an edge between {1}and {3} (Figure 7(c)). Because of the unbisimulation caused by addition of edge, we have to spilt node {4, 6, 7} into {4} and {6, 7} (Figure 7(d)). Now, after the split, any two data nodes in extent of an index node are besimilar.

Merge phase begins in Figure 7(e) by looking for an index nodes among the siblings of {1}, which have the same label and the same set of index parents. Here, we find {2}, and then merge {1}、{2}together. Next, we iteratively consider the possible merges among the children of newly generated index nodes from previous merges. In this example, we will merge index nodes {4} and {5} together. The final result of the update is shown in Figure 7(f).

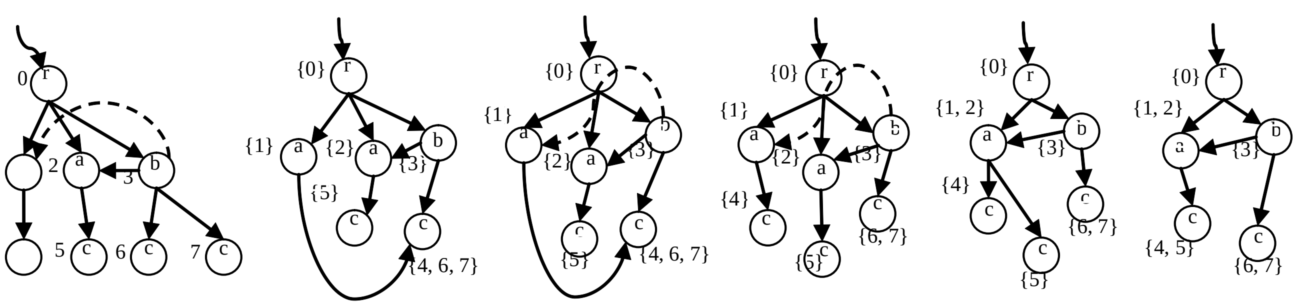

Figure 7: An Example of Edge Insertion

In general, we first checks if the new edge (u, v) makes v not bisimilar with the rest of the data nodes in v.extent. If yes, we split v.extent into one index node only containing v itself and the other that contains the rest of the data nodes. Then, we continue split other unbisimilar index nodes containing relevate nodes in the query.

The merge phase starts from I[v]. We first look for an index node with the same label and index parents as I[v] and merge them together iteratively until no more merges can be made, then tries to process the descendants iteratively with the same method.

Our algorithm is described below. In this algorithm, we use I[v] to denote the index node whose extent contains data node v, and define $ISucc[I]=\{J|(I, J)\in E_I\}$, the index successors of I.

Algorithm Edge Insertion and Deletion

insert_edge(u, v)

1: //our structure is defined by path expression Q;
2: add an edge from u to v in data graph;
3: **if** there is an index edge from I[u] to I[v]
   then return;
4: /*replace the 2 lines above with the following for deletions:
   delete the data edge from(u, v)
   if there exist u' $\in$ I[u], v' $\in$ I[v] and there is a data edge from u to v or the edge in irrelevant to Q
   then return; */
5: //Split phase;
6: **if** |I[v]|>1 **then**
7: split I[v] into I_1={v} and I_2=I-{v}:
8: **SplitNode**(I_1)

SplitNode(I)

1: **For** each index node K$\in$ISucc[I] do
2: **if** K $\cap$ Q $\neq \emptyset$
3: split K into K_1= K $\cap$ Succ[I] and K_2=K- K_1;
4: **SplitNode**(K_1);

// Merge phase;

MergeNode(I[v])

1: Look for an index node J with the same label as v among I[v]'s siblings that have the same set of index parents as I[v];
2: **if** such an index node J exists **then**
3: merge I[v] and J into M=J $\cup$ I[v];
4: M.k =J.k;
5: **for** each index node K $\in$ ISucc[I]
6: **if** K $\cap$ Q $\neq \emptyset$ **then**
7: **MergeNode**(K);

For edge deletion, we also have split phase and merge phase. We only need modify the edge insertion algorithm slightly.

In the algorithm above, split phase guarantees the structure correct, and merge phase makes the index structure minimal, which brings in high query performance.

4.2 Subtree Insertion

Because addition of subtree is tantamount to consecutive insertions of all the nodes and edges in the subtree, addition of subtree can be processed in the similar way as addition of edge. Here, because addition of new node, we don't need to split any index node and only carry out merge operation.

We give a simple example in the Figure 8, which shows the process of subtree addition. We begin this process by adding the root of the subtree and corresponding edge. The algorithm is briefly shown below. Based on the analysis of the case of edge addition, we still guarantee the structure minimal and precise.

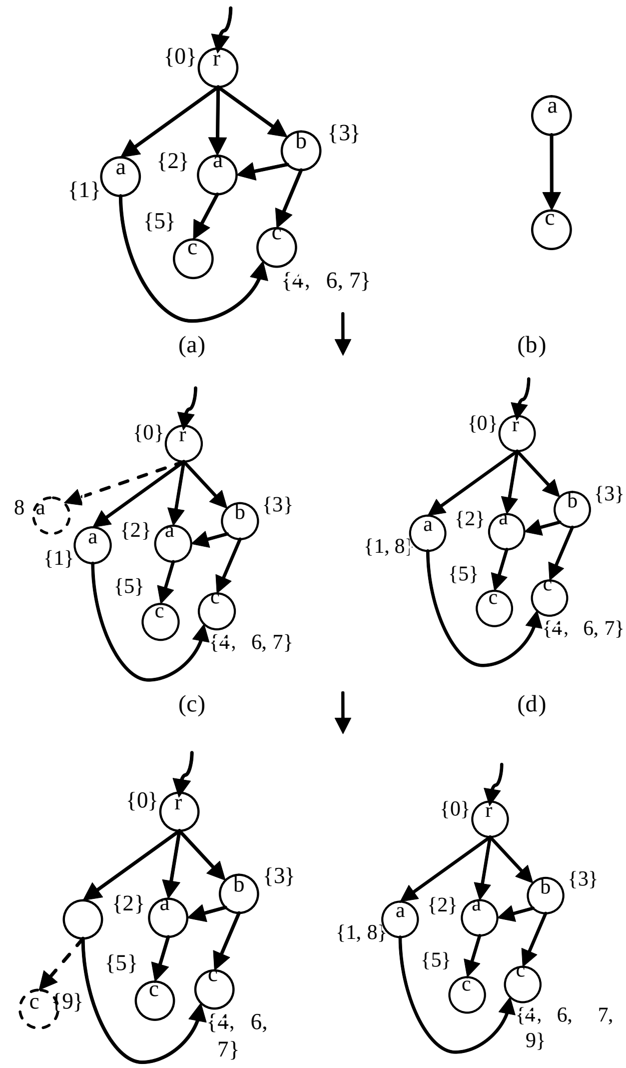

Figure 8: An Example of Subtree Insertion

Algorithm Subtree Addition

add_subtree(r)

1: Add r, the root of the subtree T and the corresponding edge to the index graph;
2: **MergeNode***(r)
3: **for** each index node K$\in$Succ[r]
4: **add_subtree** (K);

MergeNode*(I)

1: Look for an index node J with the same label as I among I's siblings that have the same set of index parents as I;
2: **if** such an index node J exists **then**
3: merge I and J into M = J $\cup$ I;
4: M.k =J.k;

For subtree deletion, this is the process of consecutive deletions of all the edges in the subtree, we can process it in the same way as the case of node deletion.

5. ANALYSIS AND EXPERIMENT

In this section, extensive experiments are conducted to demonstrate effectiveness of the S(k)-index, and we describe these experiments and present the experiment results.

5.1 Experiment Setup

Experiments are implemented in Java with JDK 1.5.0. We conduct the experiments on the AMD Athlon XP 1.83G with 256MB main memory running on Windows XP (sp2) with 80G hard disk.

We carried out our experiments on real XML database DBLP (*http://uni-trier.de/XML*). DBLP is a popular computer science bibliography database and each record of DBLP accords to a publication. The part we downloaded has size of 11M with tree structure of maximum depth 4. The document trees in the DBLP dataset have good similarity in structure.

5.2 Index Size

We measure the size of index structure with the number of edges and nodes from index structure. Similar to [7], we show the different numbers of edges and nodes on size after a large number of queries are carried out.

For M*(k)-index, we count the total number of nodes across all component indexes except duplicate nodes such as those labeled r and b in I_1 and I_2 in Figure 1. The edges in all component indexes and across-component links are counted except duplicates edges that connect duplicate nodes.

Because of the single structure, of S(k)-index, we avoid storing not only the nodes and edges from each of the components, but also the links between two components. Moreover, the merging operation of relevant nodes does well in controlling numbers of nodes and edges.

The dataset we selected from DBLP contains about 18,000 nodes. Table 1 and 2 show the number of nodes and edges based on the two index structures respectively. For this dataset, the number of nodes in S(k)-index is about 60% of that in M*(k)-index, and the number of edges in S(k)-index is about 20% of that in M*(k)-index where there are a lot of links.

Table 1
The Number of Nodes

Num. of query	*M*(k)-index*	*S(k)-index*
50	4250	2550
100	4750	2802
200	5250	3150

Table 2
The Number of Edges

Num. of query	*M*(k)-index*	*S(k)-index*
50	8700	4422
100	9052	4500
200	10032	5809

5.3 Query Performance

In this section, we investigate the query performances of the two different index structures. Table 3 lists four queries with ascending complexity. These queries from XML documents in DBLP represent four kinds of queries respectively and have different characteristics in terms of selectivity, presence of values and twig structure. Q_1 is a single path expression query with two nodes and there is no attribute values involved. Q_2 has one branch structure which involves an attribute value. Q_3 and Q_4 are twig patterns with three nodes and one branch respectively, and use wild cards which increase query scope.

Table 3
Sample Queries over DBLP

Query	*Path Expressions*	*Dataset*
Q_1	/mastersthesis/title	DBLP
Q_2	/article/Journal/IWBS Report	DBLP
Q_3	/*/author/Frank Manola	DBLP
Q_4	//author/Klaues Jansen	DBLP

Figure 9 shows the performance results of the two different index structures. The S(k)-index outperforms the M*(k)-index. For M*(k)-index, the queries are processed by traversing components from the sequence structure. The process needs visit plenty of nodes and edges, so it takes much more time to complete the queries. S(k)-index which is a single structure has smaller size, and processes the queries more efficiently. For query Q_1 whose length is very short, the slight difference between the two index structures makes their running times close. The length of Q_2 is 2, so a sequence including three components should be built for M*(k)-index to finish the query. Because of the single structure, S(k)-index has better performance on query Q_2. For Q_3 and Q_4, wildcards, "//" and "*" are involved, which enlarge the search scope. For M*(k)-index, in order to process the queries, components in the sequence must be traversed through the "links", and the cost is expensive. In construction of S(k)-index, merging relevant nodes makes the index structure succinct, and the structure can process the queries more efficiently.

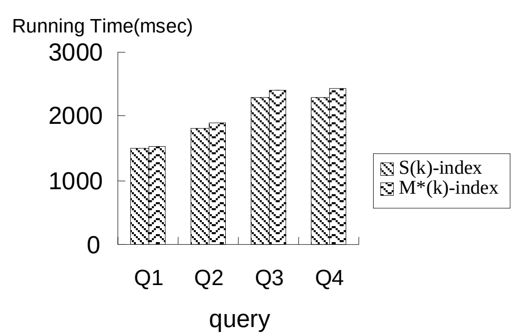

Figure 9: S(k)-index VS M*(k)-index

5.4 Updating Performance

The idea of update operation is from [10], which contains two phases, splitting and merging. But our approach is more flexible, which only processes the nodes from the current query. In this section, we analyze the impact on size of index structure after update operations.

5.4.1 Edges Insertion and Deletion

In order to generate edge insertions in a meaningful way, we first remove a sequence of edges from the data graph. These deleted edges become the "source" of insertion. Using the resulting data graph as the starting point, we perform one edge insertion followed by one edge deletion in each step: first a randomly selected edge is removed from the source and inserted into the data graph, and then another randomly selected edge is deleted from the data graph and put back into the source.

The split/merge algorithm [10] guarantees the index structure minimal and precise, but it processes all the relevant nodes in the structure. Since our insertion algorithm (I-split/merge algorithm) only processes the nodes from the current query, it takes less time to construct the resulting index structure, and the storage requirement of the resulting index structure is smaller. Figure 10 shows the ratio of augment of document after the insertion of nodes. In Figure 10, the I-split/merge algorithm maintains the size very well, never exceeding 2%, while the split/merge algorithm makes the ratio of augment much larger.

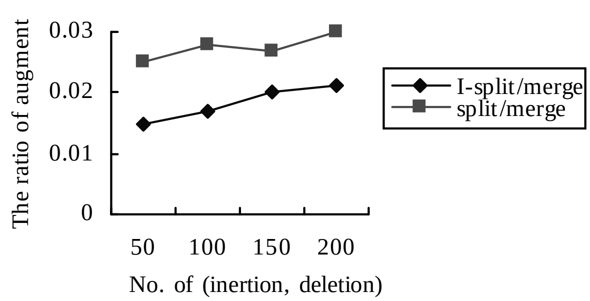

Figure 10: The Different Ratio of Augment on Edges Insertion and Deletion

5.4.2 Subtree Addition

We also construct the experiments on subtree addition. According to DBLP DTD[11], we build 50 XML documents, with an average size of 50 nodes. Addition of subtree is tantamount to consecutive insertions of all the nodes and edges in the subtree, addition of subtree can be processed in the similar way as addition of edge.

10 documents are added into original document tree once a time. Similar to the case of insertion and deletion of edges, in Figure 11, the I-split/merge algorithm keeps the better performance all the time. In terms of running cost, the I-split/merge algorithm is very fast, about 5sec for each subtree, which is 2 times faster than the split/merge algorithm.

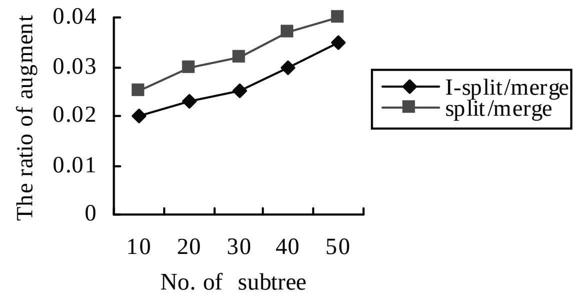

Figure 11: The Different Ratio of Augment During a Sequence of Subgraph Additions

6. CONCLUSION

In this paper, we introduce a new index structure, S (k)-index. As an improvement of M*(k)-index, it reserves the adaptive property that allows different index nodes to have different local similarity requirements, and its single structure avoids not only storage of nodes and edges in components from sequence structure, but also links between components. Furthermore, we process the update operations on S (k)-index, including additions and deletions of edge and subtree. The operations only process the relevant nodes in the query, and the structure after updating is fit for the current query process. An experimental studies of the storage requirements of S (k)-index and M*(k)-index indicate the newly proposed index structure is more compact than M*(k)-index. The experiments of performance on query and update show better efficiency than M*(k)-index.

REFERENCES

[1] Editors: T. Bray, Janguag. Paoli, and C. Sperberg-McQueen. Extensible Markup l e (XML) 1.0, February 1998. *W3C Recommendation available at http:// www.w3.org/TR/1998/REC-xml-19980210.*

[2] J. Clark and S.Derose, XML Path Language(XPath) Version 1.0, *World Wide Web Consortium, http:// www. W3. org/TR/xpath*, November 1999.

[3] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Sim´eon. XQuery 1.0: An XML Query Language. *http://www.w3.org/ TR/xquery*, August 2002.

[4] T. Milo and D. Suciu. Index Structures for Path Expressions In ICDT: *7th International Conference on Database Theory*, pages 277-295, 1999.

[5] R. Kaushik, P. Sheony, P. Bohannon, and E. Gudes. Exploiting Local Similarity for Efficient Indexing of Paths in Graph Structured Data, In Proc. of the 2002 Intl. *Conf. on Data Engineering*, pages 129-140, February 2002.

[6] Q. Chen, A. Lim, and K. W.Ong. D(k)-Index: An Adaptive Structural Summary for Graph-Structured Data.

In Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data, pages 134-144, June, 2003.

[7] H. He and J. Yang. Multiresolution Indexing of XML for Frequent Queries. *In Proc. of the 20th Intl Conf. on Data Engineering*, pages 683-694, 2004.

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

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

[10] K. Yi, H. He, I. Stanoi, J. Yang, Incremental Maintenance of XML Structural Indexes, in: *Proceedings of the 23th ACM SIGMOD*, pages 491–502, 2004.

[11] DBLP database web site. *http://uni-trier.de/X*