XML QUERY PROCESSING: INDICES AND HISTOGRAMS

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE

AND THE COMMITTEE ON GRADUATE STUDIES

OF NATIONAL UNIVERSITY OF SINGAPORE

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Qun Chen

September 2004

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Professor +++
(Principal Adviser)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Professor + + +

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Professor + + +

Approved for the University Committee on Graduate Studies.

Acknowledgements

I would first like to thank my mentor and research supervisor, Professor Andrew Lim, for his enlightening guidance and consistent encouragement on my research work. Secondly, I give special thanks to Professor Beng Chin Ooi and Professor Chan Chee Yong for acting as my supervisors concerning amendments of this thesis. Thirdly, I would like to thank the reviewers of this thesis, especially Professor Lee Mong Li; their insightful comments help improve the quality of my work.

I am also owned much gratitude to many colleagues I ever worked with, Ong Kian Win, Tang Ji Qing, Zhu Yi, Xiao Fei and Fu Zhaohui. Without them, my research work and this dissertation could not have been done smoothly.

I also would like to give thanks to my labmates and friends, Wang Gang, Cong Gao, Shi Rui, Zhang Gong, Zhu Xiaotian and others. Their precious friendship and support makes my study an enjoyable experience.

Finally, I thank School of Computing, National University of Singapore for providing me with a world class study and research environment. For faculty members who ever taught me courses and helped me professionally or administratively, I appreciate you much.

Summary

As XML gains unprecedented popularity as the standard format for presenting and exchanging information over the Internet in both the commercial and academic community, the XML database floats as a suitable, semi-structured alternative to store data. The inherent structure of XML documents renders traditional query optimization techniques for relational databases inapplicable or inadequate in the new context. This dissertation investigates two basic tools for query optimization in the XML databases: indices and histograms.

It begins with an adaptive structural summary for general graph structured data, the D(k)-index, which facilitates queries by pruning search space. As its predecessors, 1-index and A(k)-index, D(k)-index is also based on the concept of bisimilarity. However, as a generalization of the 1-index and A(k)-index, it possesses the adaptive ability to adjust its structure according to the 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 work. Experiments are conducted to show the improved performance of search and update operations on D(k)-index over its predecessors.

Existing encoding schemes proposed for XML to enable element-set-based queries mainly target the containment relationship, specifically the *parent-child and ancestor-descendant* relationship. The presence of *preceding-sibling* and *following-sibling* location steps in the XPath specification, which is the *de facto* query language

for XML, makes the horizontal navigation, besides the vertical navigation, among nodes of XML documents a necessity for efficient evaluation of XML queries. Our work enhances the existing range-based or prefix-based encoding schemes such that all structural relationship between XML nodes can be determined from their codes alone. Furthermore, an *external* memory index structure based on the traditional $B+$-tree, $XL+$-tree(XML Location+-tree), is introduced to index element sets such that all defined location steps in the XPath language, *vertical* and *horizontal*, *top-down* and *bottom-up*, can be processed efficiently. The $XL+$-tree under the range or prefix encoding scheme actually share the same structure; but various search operations upon them may be different as a result of the richer information provided by the prefix encoding scheme. Our experiments demonstrate the superior performance of the $XL+$-tree over existing external-memory index structures for XML query processing.

Summary data, or histograms, on XML documents can provide critical information for query optimizers of XML databases. Traditional histograms for relational database fall short, since they do not address path patterns of XML documents. The dissertation also makes contributions in this aspect. It proposes a structural XML histogram, namely *SHiX*, which uses a novel framework for estimating the selectivity of *twig path expression*s on graph-structured XML databases. Instead of exploiting bisimilarity or *divide-and-conquer* strategy, which typify previous approaches, *SHiX* keeps both the numeric relationship(the average number of children) and forward stability information in the summary graph. Efficient algorithms to build *SHiX* histograms are also presented. Extensive experiments on both the real and synthetic XML data validate the effectiveness of the *SHiX* approach.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years, the eXtensible Markup Language(XML)[8] has become the dominant standard for exchanging and querying documents over the World Wide Web. XML is an example of semi-structured data [4, 6]. XML data do not conform to traditional data models, such as relational or object-oriented models. Instead, the underlying data model of XML data is an ordered labeled tree. 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. Thus, XML data, are hierarchically structured and self-describing.

1.1 XML Data Model

An XML document is usually parsed into an ordered labeled tree, with each node in the tree corresponding to an element, an attribute or a text data. Each node is labeled with the element or attribute name. Text data nodes are given a distinguish label, VALUE. Edges between nodes represent element-subelement, element-attribute or element-value relationship. Each node is also assigned a unique *id*.

An example XML data model is shown in Figure 1.1. It is worth noting that references can be established between XML nodes *via* the *ID/IDREF* construct or *Xlink* syntax. An XML database consists of a forest of such trees.

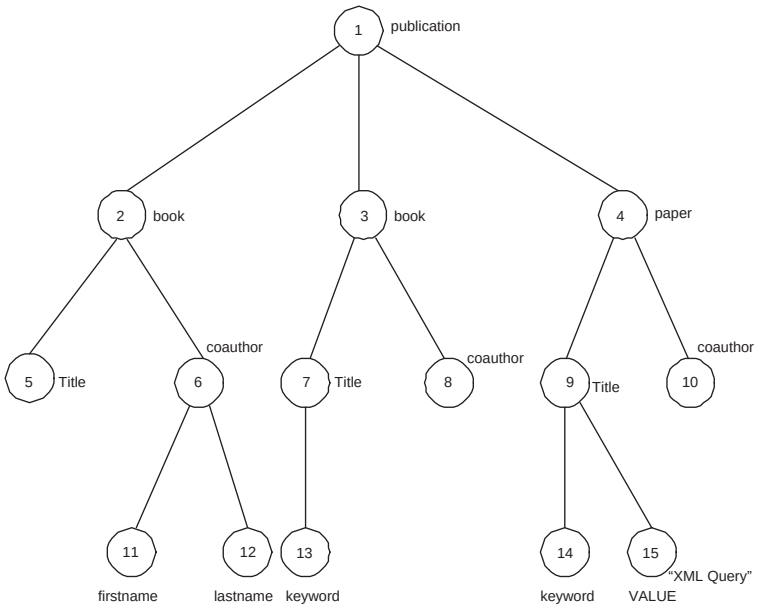


Figure 1.1: An Example XML Data Model

1.2 The XPath Query Language

A variety of query languages [1, 2, 3, 4, 5] have been proposed to query XML data. All of these query languages are built around the XPath specification [7].

The core of Xpath language, the path expression, is used to locate nodes in a XML tree. A path expression begins with a context node(not necessarily the root), which is the starting point of the tree traversal, and consists of a series of location steps. Given a context node, a step's axis establishes the subset of document nodes that are reachable from this context node *via* the specified axis. This set of nodes provides the context nodes for the next location step. There are totally 13 different axes defined in Xpath:namely, *child*, *parent*, *descendant*,

Axis	Results
child	direct child nodes
descendant	recursive closure of *child*
descendant-or-self	*descendant* plus self
parent	direct parent node
ancestor	recursive closure of *parent*
ancestor-or-self	*ancestor* plus self
following-sibling	following nodes in document order, having the same parent
preceding-sibling	preceding nodes in document order, having the same parent
following	following nodes in document order, excluding *descendant* nodes
preceding	preceding nodes in document order, excluding *ancestor* nodes
attribute	attribute node
namespace	namespace node
self	self node

Table 1.1: Semantics of XPath Axes

ancestor, *following-sibling*, *preceding-sibling*, *following*, *proceeding*, *descendant-or-self*, *ancestor-or-self*, *self*, *attribute*, *namespace*. Semantics of XPath axes are described in Table 1.1. The document order in an XML tree orders its nodes corresponding to a sequential read of nodes by a preorder traversal. For instance, in the tree representation of an XML document in Figure 1.1, the evaluation of the path expression P_1: //`publication`/ *child*::`book`/*descendant*::`keyword` returns node {13}; the evaluation of P_2: //`publication`/*descendant*::`title`/*following-sibling*::`coauthor` returns nodes $\{6, 8, 10\}$; and the evaluation of P_3: //`keyword`/ *ancestor*::`paper`/*child*::`coauthor` returns node {10}.

The primitive path pattern of interest to us is *regular path expression*. A node path in an XML tree T 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 label in T. And we denote the regular language specified by R as $L(R)$. We say that R matches a 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 T is the set of nodes in T that match R. For example, the path expression, *publicaion.book.title*, evaluated on the tree in Figure 1.1, will return $\{5, 7\}$; the more general path expression, *publication._.title*, finds titles of all kinds of publication. Here, the optional _ allows the query to ignore the irregularities in the data. This expression matches nodes $\{5, 7, 9\}$.

1.3 Optimization Techniques for XML Query Processing

In this section, we only briefly review existing techniques to facilitate XML query processing. More detailed discussion will be presented in the corresponding chapters later.

Due to the prevalence of relational databases, there have been lots of work on storing and querying XML documents using relational database systems [10, 11, 12, 13, 14, 15, 16, 17]. These techniques deal with how to "shred" XML documents into relations and translate XML queries into SQL queries over those relations. Please note that this appoach of taking advantage of relational query engine to optimize XML queries is beyond the scope of this dissertation. Instead, our work focus on the optimization techniques for querying XML data "naively" stored on the XML data model.

Existing indexing proposals for queries on XML data models can be categorized into two groups. One of them is to build the structural summary of the XML document, which has the form of a labeled directed graph. Typically, each node

in the structural summary corresponds to an equivalence class. Data nodes in the same equivalence class have the same or similar incoming paths. Therefore, path queries on the source data can be instead performed on the structural summary, which can be potentially much smaller depending on regularity of surce data. The structural summary has been shown to be effective in pruning the search space while evaluating non-branching *regular path expression*s. The other approach is based on node encoding. It assigns unique codes to nodes of the XML data model such that structural relationship between nodes can be decided from their codes alone. Such encoding technique enables the element-set-based query processing, which does not involve traversing the data graph. For instance, given a simple *regular path expression* P, $A.B$, suppose that we have element sets $\triangle_1$ and $\triangle_2$ for label A and B respectively; all node elements in $\triangle_1$ have the label A and all node elements in $\triangle_2$ have the label B. Then, all pairs of elements satisfying the parent-child relationship in $\triangle_1$ and $\triangle_2$ can be found by the join operation, namely *structural join* in the literature, since from codes of two elements we can decide whether they are parent and child. *Structural join* has been established to be the building block for more complex XML query processsing.

Another important problem of XML query optimization concerns building effective summary statistics, *histogram*, for XML data. Since XML queries can usually be presented as twig patterns, it is of primary importance to estimate the size of *twig path expression*s on XML data accurately and efficiently.

The remainder of this dissertation is organized as follows. In chapter 2, we propose an adaptive structural summary for XML data, D(k)-Index. Construction and update operations on D(k)-index and experiments results are also presented. We investigate indexing techniques for element-set-based XML query processing in chapter 3. Specifically, enhanced range-based and prefix-based encoding schemes

for XML data are introduced. We also propose the external-memory index structure, XL^+-tree, which indexes element sets such that all location steps specified in the XPath language can be implemented I/O efficiently. Chapter 4 is contributed to building effective histograms for XML data. A new histogram model, *SHiX*, is presented as a robust result estimater of *twig path expression*s over the general graph-structured XML data. Finally, we conclude our work and give a few suggestions for future research in chapter 5.

Chapter 2

Structural Summary

Querying XML document usually means traversing the structured data to locate target part of documents. 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 structure underlying XML is therefore an essential component for querying these data. A naive evaluation of path expressions that scans all data is obviously computationally expensive. A structural summary [18, 19, 20, 21] 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. This chapter considers the problem of building an adaptive structural summary for the more general graph structured data, of which XML tree-structured data is a special case. It was mentioned in the introduction chapter that references can be established between XML tree nodes. If these references are treated as normal edges, the underlying XML data model is actually a graph. In Figure 2.1, a portion of an XML document about movies with references is represented. The solid edges, which are tree edges, represent containment relationships between nodes. Non-tree edges(shown as dashed lines)

represent reference relationships. In this chapter, these two types of edges are not differentiated.

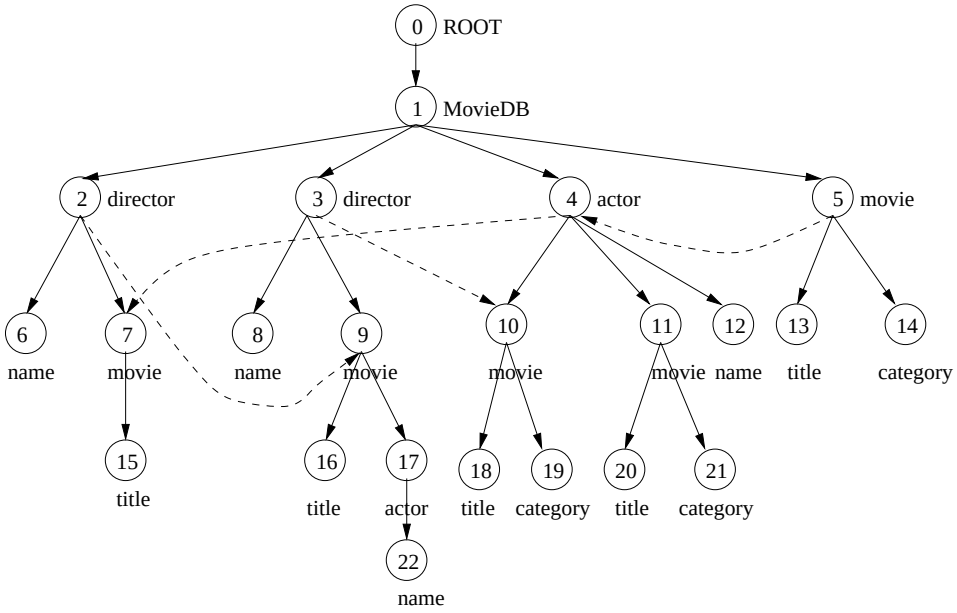


Figure 2.1: An XML Document with Reference Edges

2.1 Introduction

Existing structural summaries for graph-structured data are based on the notion of bisimilarity [24, 25]. 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 [20] 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 [21] 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 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. 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.

2.2 Previous Work on Structural Summary

Three previous summary structures have been proposed for graph-structured data to help evaluate path expressions, the strong DataGuide [18], the 1-index [20], and the A(k)-index [21]. We have already briefly examined the 1-index and the A(k)-index. The strong DataGuide of a graph data is computed by interpreting it as a non-deterministic automation and obtaining an equivalent deterministic automation [33]. 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 [18]. However, because the 1-index, A(k)-index and our new D(k) index, based on graph bisimulation, are non-deterministic if they are treated as antomata, those algorithms can not be generalized to apply in this context. Most recently, update algorithms for 1-index were presented in [26]. 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. Very recently, the update algorithms with provable guarantee on the resulting index quality for 1-index and A(k)-index has been proposed in [40]. It actually involves two phases: splitting and merging, in which the splitting phase is essentially the same as proposed in [26].

Graph schema[27, 28] 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 [29, 32]. 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 [22, 23] 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.

2.3 Bisimilarity

The core idea of building the structural summary 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 or XML data 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 2.1, 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.

2.4 D(k)-Index

2.4.1 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 $\mathbf{O}(mlgn)$ time using Paige and Tarjan's algorithm [25], 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 [21] 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;*
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 [21]:

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 $\mathbf{O}(km)$ 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 2.1, 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.

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 $s = m-1$. 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 a data graph, 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, $k(n_i) \geq m$.

2.4.2 Construction

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 less than its child n_j's local similarity by more than 1. 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 $1 < i < t$, 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 in **Algorithm 2.1**. It takes $\mathbf{O}(m)$ time, in which m is the number of edges in the label-split index graph.

Algorithm 2.1: 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 [21]. For a set of data nodes, A, let $Succ(A)$ denote the set of successors of the nodes in A, i.e., the set $\{v \mid$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,

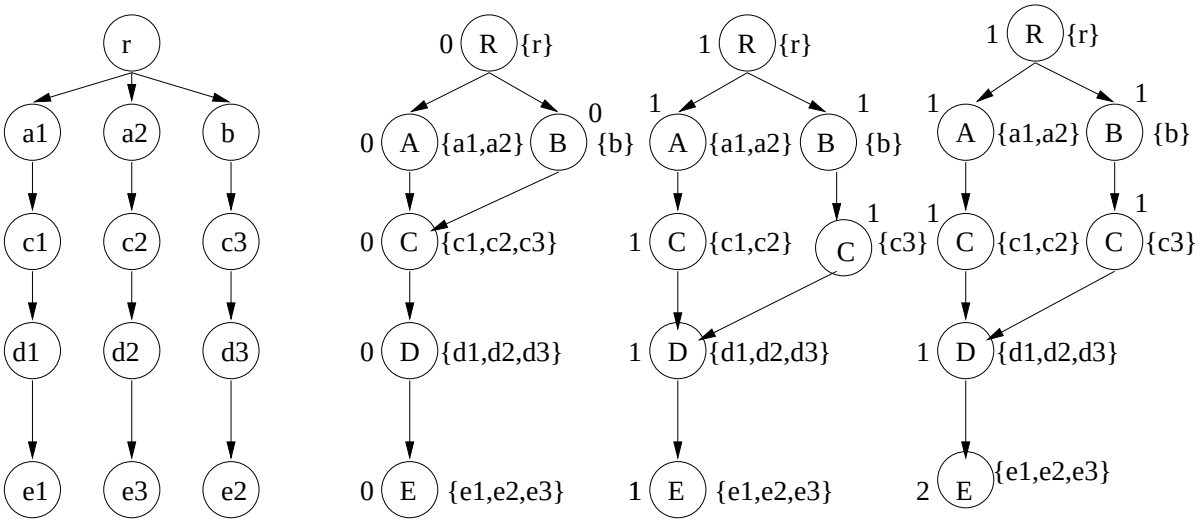
in which all index nodes are 0-bisimulation equivalence classes. Then it proceeds to construct the 1-bisimulation equivalence classes. It repeats this process until the local similarity requirements of all index nodes are satisfied. The D(k)-index construction algorithm is presented in **Algorithm 2.2**. A construction example is shown in Figure 2.2. Please note that:(1) Label E has a local similarity requirement of 2, other labels have a local similarity requirement of 1;(2) the numbers besides the nodes are actual local similarities in the D(k)-Index. It takes $\mathbf{O}(km)$ 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.

Algorithm 2.2: 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 of newly created index nodes by inheritance;
   - Set X to be a copy of the updated I_G;
5. Return the resulting I_G.



(a) The Data Graph G (b) The Label−Split Graph (c) Round 1 (d) Round 2

Figure 2.2: D(K)-Index Construction Example

2.5 D(k)-Index Updating

The paper [39] defines several primitive update operations upon XML documents. We use them as the target operations upon which the D(k)-Index should be adjusted accordingly. As in [39], we use the term *object* to refer to any component of XML, which can be an element, an attribute, an IDREF or a PCDATA content, and assume the presence of tuples of references to the selected objects within XML documents through a path expression matching operation. The defined update operations include: (1) **Delete**(*child*): if the *child* is a member of the target *object*, it is removed;(2)**Insert**(*content*): it inserts a new *content*, which can be element, attribute, reference or PCDATA, into the target *object*; (3)**Rename**(*child*,name): if the *child* is a non-PCDATA member of the target *object*, it is renamed. Note that there are three other update operations presented in [39]. **InsertBefore**(*ref*, content), which is defined only for ordered execution and inserts a new content directly before the target **ref**, poses no difference from the **Insert**(*content*) operation concerning the update operation on D(k)-Index. **Replace**(*child*,content), which is

a replace operation, can be considered to be equivalent to a **Insert**(content) operation followed by a **Delete**(*child*) operation. The **Sub-Update**(*patternMatch, predicates, updateOp*) operation invokes a new path expression matching operation over the target *object*, returns bindings filtered by *predicates* and recursively invokes the update operation *updateOp*. Therefore, it is enough that we address the update operation on D(k)-Index upon the three atomic update operation on XML documents,**Delete**(*child*), **Insert**(*content*) and **Rename**(*child*,name).

In [26], two kinds of update operations upon XML documents are considered for updating the structural index: 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. In this section, we first present efficient update algorithms for the D(k)-index when a new file is inserted or a new edge is added into the data graph. Then, we proceed to demonstrate that our approaches used in these two basic cases are flexible to accommodate other defined operations on XML. Finally, we propose two procedures, *promoting* and *demoting*, to adjust the D(k)-index for a changing query load.

2.5.1 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 [26]. 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 [26].

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 2.3: 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 $I_{G'}$.

2.5.2 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 [26]. 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 2.3. The *propagate* algorithm for the edge addition proposed in [26] 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.

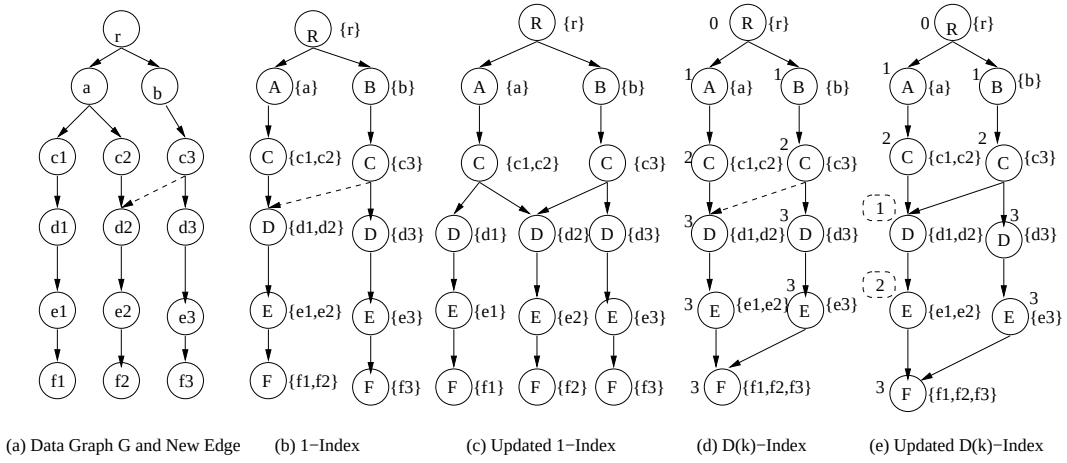


Figure 2.3: 1-Index Update vs D(k)-Index Update

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 2.3, 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 enlarge the set of labels of d_2's parents. Since D's original local similarity before the edge addition is larger than 1, the local similarity of D 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.

Algorithm 2.4: 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. $NLSim$=0,$Stop$=false; // (NLSim denotes V's new local similarity);
3. $NewLabelPathSet(1)$=$\{label(U)\}$, $OldLabelPathSet(1)$=$\{$l|l is the label of some parent(except U) of V in $I_G\}$; And for each label path P in NewLabelPathSet, we keep a set of index nodes in I_G, $S_i(P)$, 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 (NLSim$\leq$Upbound and Stop=false)
   - if ($NewLabelPathSet(NLSim{+}1) \subseteq OldLabelPathSet(NLSim{+}1)$)
     - $NLSim = NLSim + 1$;
     - $OldLabelPathSet(NLSim) = NewLabelPathSet(NLSim)$;
     - Set $UpdatedNewLabelPathSet$ to an empty set;
     - Set $UpdatedOldLabelPathSet$ to an empty set;
     - For (each label path P in $OldLabelPathSet(NLSim)$)
       - for each index node w in $S(P)$
         - for each parent x of w in I_G(excluding $U \rightarrow V$), insert the label path P'=(label(x)+P) to $UpdatedOldLabelPathSet$ and insert x into $S(P')$;
     - $OldLabelPathSet(NLSim+1)$=$UpdatedOldLabelPathSet$;
     - for (each label path P in $NewLabelPathSet(NLSim)$)
       - for each index node w in $S_i(P)$
         - for each parent x of w in I_G, insert the label path P'=(label(x)+P) to $UpdatedNewLabelPathSet$ and insert x into $S_i(P')$;
     - $NewLabelPathSet(NLSim+1)$=$UpdatedNewLabelPathSet$;
   - else Stop=true;
5. Return NewLocalSimilarity.

Algorithm 2.5: 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 propagating the update request from X.

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 have 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. For a 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 $S_i(P)$, which has a path into V through U matching P. Similarly, the set of index nodes, each of which has a label path P into V in the original I_G, is denoted as $S(P)$. We also denote the set of label paths of length k_N into V through U in I_G as $NewLabelPathSet(k_N)$ and the set of label paths of length k_N into V in the original I_G as $OldLabelPathSet(k_N)$. It is clear that if $NewLabelPathSet(k_N) \subseteq OldLabelPathSet(k_N)$, V's local similarity can be reset to k_N in I_G. To proceed

from k_N to $(k_N + 1)$, we need to compute both $NewLabelPathSet(k_N + 1)$ and $OldLabelPathSet(k_N + 1)$. For each label path P in $NewLabelPathSet(k_N)$, labels of parent nodes of each node in $S_i(P)$ should be appended at the head of P; the resulting label paths are of length $(k_N + 1)$ and should be included in $NewLabelPathSet(k_N + 1)$. $OldLabelPathSet(k_N + 1)$ can be computed from $OldLabelPathSet(k_N)$ in a similar way. But be cautious that it is computed in the original I_G with the absence of the edge $U \rightarrow V$. In **Algorithm 2.4**, members of sets $UpdatedNewLabelPathSet$, $UpdatedOldLabelPathSet$, $S_i(P)$ and $S(P)$ are all kept to be distinct.

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.

2.5.3 Other Update Operations upon XML

In this subsection, we first consider the update algorithm on D(k)-Index when an edge is deleted from the original XML document. It is shown to be almost the same as the update algorithm upon an edge insertion. Then we discuss detailedly involved operations on D(k)-Index upon three basic update operations on XML

documents we introduced at the beginning of this section.

Suppose that the edge from u to v is deleted in the original XML data G, and $u \in U$ and $v \in V$ in I_G. If v is still connected with some other data node in $extent(U)$, the local similarity of V remains unchanged. Otherwise, as in the case of the edge insertion, we need to reset V's local similarity in I_G. We have 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 *without* through the edge $U \rightarrow V$, V's local similarity can be reset to k_N. Therefore, a straightforward application of **Algorithm 2.4** can achieve this purpose if we assume the absence of the edge $U \rightarrow V$ in the original I_G. Unlike the case of edge insertion, where the update operation on D(k)-Index does not need to resort to the source data, the update operation on D(k)-Index upon edge deletion needs to check whether U and V remain connected in I_G after the edge $u \rightarrow v$ is deleted from the original data G; thus it involves checking the connectivity between data nodes in $extent(U)$ and in $extent(V)$ after the deletion.

We are now ready to detail the corresponding update operations on D(k)-Index for the defined basic update operations upon XML documents. For the **Delete**(*child*) operation, it amounts to the edge deletion if the *child* is an IDREF. 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*) requires simply removing data nodes corresponding to *child* from *extent*s of index nodes on the D(k)-Index. The local similarities of index nodes on D(k)-Index remain unchanged. The **Insert**(*content*) operation amounts to the edge insertion if the *content* is a reference. Otherwise, a new index node N is created for each inserted *content* in I_G. Its *extent* contains only the new data node and its local similarity is set to be $k_P + 1$, in which k_P is the local similarity of its parent node in I_G. Now we consider the **Rename**(*child*,name) update operation. Suppose that a data node u in $extent(U)$ is renamed as N_{new}. The update

operation on D(k)-Index upon **Rename**(u,N_{new}) consists of two steps: (1) it creates a new index node N labeled N_{new} for the renamed data node u and assigns $1 + min\{k_{P_1}, k_{P_2}, \ldots, k_{P_t}\}$ as its local similarity, in which P_i, for $1 \leq i \leq t$, is the new index node(N)'s ith parent in I_G;(2) Reset local similarities of N's descendant nodes in I_G. In the step (2), as presented in **Algorithm 2.5**, we first reset local similarities of N's child nodes and then broadcast the updates to other affected descendant nodes in I_G. Assume that the connectivity between N and other index nodes has been properly updated in I_G and there is an edge $N \rightarrow V$. Again, we have an observation similar to the one in the case of edge insertion: if all label paths of length $k_N (\leq k_V)$ going into V through U or N in the updated I_G, match V in the original I_G *without* through the edge $U \rightarrow V$, V's local similarity can be reset to k_N. If only resetting V's local similarity is concerned, the *Rename* operation amounts to an edge deletion operation(from u to data nodes in $extent(V)$) followed by an edge insertion operation(from the renamed u in $extent(N)$ to data nodes in $extent(V)$). Therefore, a minor variant of **Algorithm 2.4** can be applied to reset V's local similarity. The difference is that in step 3, the $NewLabelPathSet$ should be initially set to be $\{label(U), label(N)\}$.

2.5.4 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 $V \cap Succ(U)$ and $V - Succ(U)$. 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 **Single-Node_Promoting_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 the local similarities of their close ancestors. The result is that some index node promotions may be saved.

Algorithm 2.6: Single-Node_Promoting_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=**Single-Node_Promoting_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.

In case that a lot of index nodes need to be promoted in the D(k)-index, instead

of promoting them one by one, we propose a more efficient mass promoting algorithm. Suppose that those index nodes requiring promotion have been assigned new local similarities. Note that the new local similarities should satisfy the properties of D(k)-index. We call an index node in the D(k)-index in the *stable* state if all its child index nodes have reached their target local similarities. N_k denotes the set of index nodes that if their child index nodes are partitioned according to them, these child nodes' local similarities are at least promoted to $(k+1)$. Initially, we set the states of index nodes in the D(k)-index to be *stable* or *unstable*. For each *unstable* index node X, if X has at least one child node Y satisfying $lsim(Y) < lsim(X) + 1$, in which $lsim$ represents the index node's current local similarity, we insert X into the set N_{k_i}, in which k_i is the minimal value of $lsim(Y)$ satisfying $lsim(Y) < lsim(X) + 1$. Next, we sequentially consider the set N_k in the increasing order of k's value. For each index node U in N_k, if all its child index nodes have reached their target local similarities, U's state is set to be *stable*;otherwise, consider each U's child index node V, if V's current local similarity is less than its target value and $lsim(U)+1 > lsim(V)$, we split V into $Succ(U)$ and $V - Succ(U)$. Since V may have several parent nodes in N_k, V may be partitioned into multiple sub-nodes, $V_1V_2 \ldots V_t$, in which V_i's new local similarity is set to be *max{lsim(V), k+1}*. And if the original node V is *unstable*, each V_i should be inserted into N_{k+1}. Furthermore, after all the splitting and updating operation, if an index node W has not reached its target local similarity and its current local similarity $lsim(W)$ is less than $(k+1)$, reset its local similarity to be $(k+1)$. Finally, if any node W, whose local similarity has been reset, has any child node X that has not reached its target local similarity and satisfies $lsim(W) + 1 > lsim(X)$, the algorithm inserts W into the set N_{k+1}. The algorithm repeatedly processes $N_{min}N_{min+1} \ldots N_{max-1}$, where min is the smallest local similarity of nodes that has not reached their target local similarities before any promotion and max is

the maximal target local similarity of nodes in the D(k)-index. The whole mass promoting algorithm is described in the **Mass_Promoting_Algorithm**.

Algorithm 2.7: Mass_Promoting_Algorithm(I_G)

Input A D(K)-index I_G and target local similarities for index nodes in I_G

Output A promoted D(K)-index I'_G in which all index nodes possess their target local similarities

1. Initialization;
   (a) (For each node U in I_G)
   - If (all U's children have reached target local similarities)
     - set $stable(U)$=true;
   - Else
     - set $stable(U)$=false;

   (b) For each node V with $stable(U)$==false
   - if (U has at least one child V satisfying $lsim(U)+1 > lsim(V)$)
     - Insert U into N_{k_i}, in which k_i is the minimal local similarity of such children;

2. (For k=min to max-1)
   - (For each node U in N_k)
     - (For each child V of U)
       - Partition V into $Succ(U)$ and $V - Succ(U)$;
       - Set the local similarities of new nodes as $max\{lsim(V), k+1\}$;
   - (For each new node W as a result of splitting)
     - If ($stable(W)$==false)
       - Insert W into the set N_{k+1};
   - (For each node W that has not reached its local similarity and $lsim(W) < (k+1)$)
     - Reset $lsim(W)$ to be $(k+1)$;
     - If (W has any child X that has not reached its target local similarity and $lsim(X) \leq (k+1)$)
       - Insert W into N_{k+1};

To prove its correctness, we have the following lemma:

Lemma 1 *Before processing the set N_k, the local similarity of any index node V that has not reach its target local similarity in the D(k)-index can be reset to k if its current local similarity $lsim(V) < k$.*

Proof: Suppose that k can take values of $k_1, k_1+1, \ldots, k_2$. We prove by induction on the value of k.

Firstly, we consider the base case when $k = k_1$. Suppose that the index node V has not reached its target local similarity $tlsim(V)$ and $lsim(V) < k_1$. If V has any parent index node U in the D(k)-index satisfying $lsim(U) \geq lsim(V)$, the smallest value of k should be $lsim(V)$ but not k_1 according to the algorithm. Therefore, either V has no parent index node or all its parent index nodes have local similarities less than $lsim(V)$(actually should be $lsim(V) - 1$ according to the definition of D(k)-index). In the first case that V has no parent index node, obviously its local similarity can be validly reset to be k_1. In the second case that V's all parent index nodes have local similarities of $(lsim(V)-1)$, we can conclude recursively that any parent node of V's parent nodes, if it exists, should have local similarities of $(lsim(V) - 2)$. Assume that there is any path P with length larger than $lsim(V)$ into node V in the D(k)-index, $U_m U_{m-1} \ldots U_1 V$ with $m > lsim(V)$. We have $lsim(U_i) + 1 = lsim(U_{i-1})$; thus $lsim(U_m) = lsim(V) - m < 0$, which is contradictory to the fact that any index node in the D(k)-index has local similarity of at least 0. Therefore, we can conclude that all paths into V in the D(k)-index graph have length no larger than $lsim(V)$. As a result, V's local similarity can also be validly reset to k_1.

Secondly, we assume that when $k = i$, the lemma is true; now we consider the case $k = i + 1$. Consider index nodes that have not reached their target local similarities before processing N_i. Their local similarities are at least i according to the assumption for the case $k = i$. If it can be proved that after processing N_i, local

similarities of these index nodes or sub-nodes split from them can be promoted to at least $(i+1)$, we complete the whole lemma proof. If any index node has local similarity no less than $(i+1)$ even before processing N_i, obviously local similarities of this node or its sub-nodes remains at least $(i+1)$ after processing N_i. Now we consider any index node V that has not reached its target local similarity and satisfies $lsim(V) = i$ before processing N_i. Any parent node U of V either has reached its target local similarity or has local similarity at least i. Note that since $tlsim(U) \geq tlsim(V) - 1 \geq lsim(V) = i$, in either of the above two cases we have $lsim(U) \geq i$. There are two possibilities:

1. The node U results from the split operation on some index node U_a. Suppose that the splitting happens at the round of processing $N_t (t < i)$. Since U_a has child nodes that have not reached target local similarities, U should be inserted into the set N_{t+1}. While processing N_{t+1}, the algorithm splits any child node W of U into $Succ(U)$ and $W - Succ(U)$ if W has not reached its target local similarity. Any split sub-node W_b of W should also be stable with U. Therefore, the node V is stable with respect to U.

2. The node U exists in the original D(k)-index before any promotion. If U's original local similarity is also $lsim(U)$, U should be in some set $N_s (s \leq i)$ before the first round of processing N_{k_1}. Otherwise, if U's local similarity is promoted to be $lsim(U)$ not through splitting but through resetting; in this case, $lsim(U) = i$ and U should be inserted into N_i according to the algorithm. As in the first case, while processing N_s or N_i, the algorithm splits any child node W of U into $Succ(U)$ and $W - Succ(U)$ if W has not reached its target local similarity. Therefore, the node V is also stable with respect to U.

Therefore, we have the conclusion that after processing N_i, V is stable with

respect to all its parent nodes. Since all V's parent nodes has local similarity of at least i, V's local similarity can be promoted to $(i+1)$. Proof finished. □

Based on **Lemma 1**, we have the following theorem:

Theorem 3 *The* **Mass_Promoting_Algorithm** *correctly promotes local similarities of index nodes in the D(k)-index to their target values.*

Proof: According to **Lemma 1**, after processing N_{max-1}, in which max denotes the maximal target local similarity specified for index nodes in the D(k)-index, if an index nodes has not reached its target local similarity, it should have local similarity of at least max. Since the maximal target local similarity of nodes is max, we can conclude that all index nodes reach their target local similarities. □

2.5.5 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 change. 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'.

In case that only a few labels in D(k)-Index need to be demoted, instead of constructing the new I_G from the current I'_G, we can directly explore the possibilities of merging same-labeled index nodes in I'_G. More specifically, we assume that local similarities of labels, $L_1, L_2, \ldots, L_t$, are supposed to be demoted, and label L_i's local similarity($1 \leq i \leq t$) is lowered to $k_{low}(i)$ from $k_{high}(i)$. The demoting process works in three steps:(1) it assigns the new local similarity $k_{low}(i)$ to each label L_i in the label-split index graph, with other labels' local similarities remaining unchanged, and uses the local similarity broadcast algorithm presented in **Algorithm 2.1** to compute the updated local similarities required by D(k)-Index; (2) For each label $L_i(1 \leq i \leq t)$, and for each L_i-labeled index node N_{L_i} in I'_G, it computes all label paths of length $k'_{low}(i)$(L_i's updated local similarity) into N_{L_i};(3)finally, if two same-labeled index nodes have the same set of incoming label paths, they are merged in the new I_G; and connectivities between index nodes in I_G are updated correspondingly. Be cautious that the merging operation happens in step (3) only after incoming label paths concerning all relevant index nodes have been computed. And note that step (2) can be accomplished using the procedure we presented in **Algorithm 2.4**, where we also need to compute all label paths of some length into index nodes on D(k)-Index.

2.6 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. The purposes of our experimental study include:

1. To investigate the evaluation performance of D(k)-index in comparison to the previous A(k)-index;

2. To evaluate the performance of the update operations on D(k)-index;

3. To demonstrate that the proposed promotion/demotion operations are effective in maintaining the D(K)-index for high evaluation performance.

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 [41], 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 100M in size.

2. Nasa Data. This data set is generated by the IBM data generator using a real DTD file, nasa.dtd [42], 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. The resulting Nasa data is an XML file of about 100M in size.

2.6.1 Evaluation Performance

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 [21]. 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 6 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. We expect that the resulting query load basically simulates query patterns in real XML databases. 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(5). Note that evaluating test paths on the A(5)-index is already *sound*; that is, no validation process is triggered because all test paths are of length less than or equal to 6. Therefore, we do not experiment on A(k) with $k > 5$ because its performance is definitely worse than A(5). The results on the Xmark and Nasa data are presented in Figures 2.4 and 2.5, 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 A(k)-index. Therefore, these results demonstrate the superior performance of the D(k)-index over the A(k)-index.

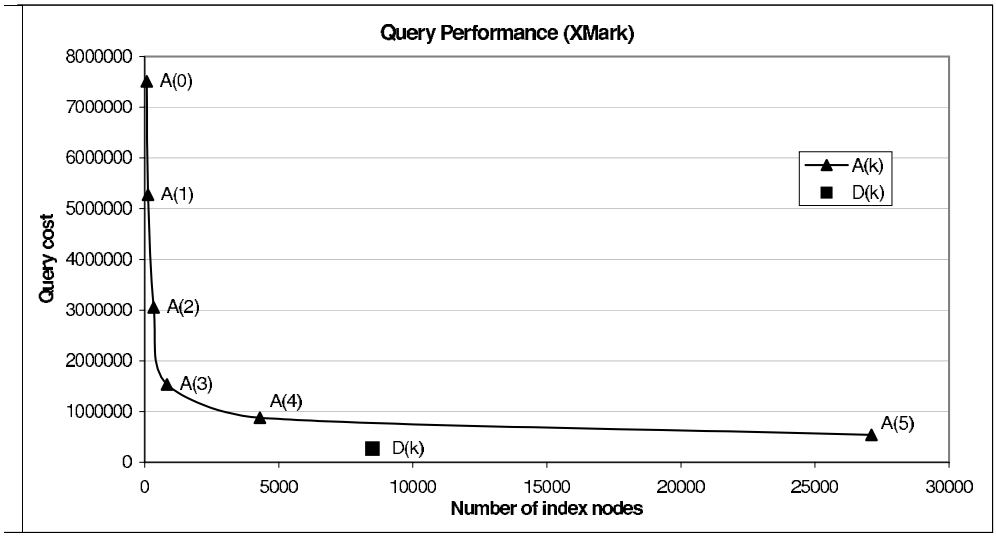


Figure 2.4: Evaluation Performance Comparison between the D(K)-index and the A(k)-index on Xmark Data Before Updating

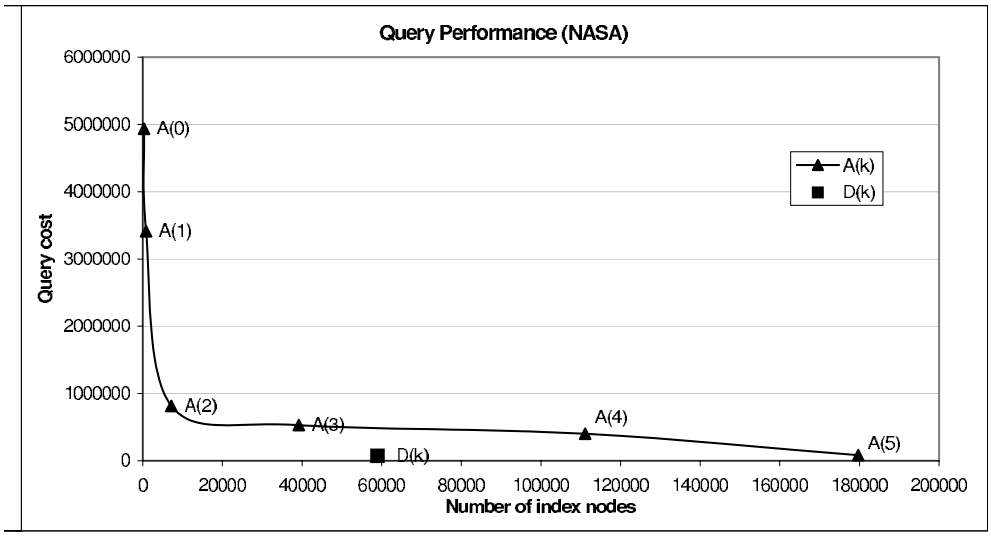


Figure 2.5: Evaluation Performance Comparison between the D(K)-index and the A(k)-index on Nasa Data before Updating

2.6.2 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.

We adopted a variant of the 1-index update algorithm proposed in [26]. Note that very recently, the update algorithms with provable guarantee on the resulting index quality for 1-index and A(k)-index has been proposed in [40]. It is worthy to point out that the new update algorithm actually involves two phases: splitting and merging, in which the splitting phase is essentially the same as proposed in [26]. In our experiments, instead of exploring the merging potential for newly created index nodes whenever an new edge is inserted into the source graph data, the update on A(k)-index only involves the splitting phase; the merging operation is triggered only after a considerable number of updates on source data. The adoption of this update approach for A(k)-index in our experiments is based on

two experimental observations: (1) even though the size of A(k)-index may increase considerably as the result of splitting while updates are conducted on source data, its query performance deteriorates only slightly(below 1%) in most cases in our cost model so long as the index graph resides in main memory; (2) the merging phase can consume a considerable portion of CPU time; for fair comparison, we do not treat the merging phase of the A(k)-index as part of update operation, but as a maintenance operation that is performed only periodically; in the D(k)-index, the maintenance involves both the promotion and merging operations.

Specifically, the update operation on A(k)-index is as follows. When a new edge is added to the A(k)-index graph from U to V as a result of an edge insertion from node u to node v in source data, firstly it determines the maximal local similarity that node V can be reset to; this can be achieved in the same way as described in the updating D(k)-index section. Secondly, if V's reset local similarity k_r is less than k, the algorithm creates a new index node V_n with $extent(V_n) = \{b\}$ and recursively splits the data nodes, whose parents are in the new created index nodes, from their corresponding index nodes. The second process is repeated until the data nodes $(k - k_r)$ distant from the data node v are reached. It is easy to see that each index node of the resulting index graph satisfies k-bisimilarity. Note that this update algorithm is different from the one presented in the conference version of this paper. It does not check the maximal local similarities of newly created index nodes except V_n. Since the checking process is exponential with respect to k, our experiments show that the new update algorithm is significantly more efficient than the old one. Another justification of adopting the new algorithm is that even after a considerable number of updates(for instance 300), sizes of the resulting A(k)-indexes of two algorithms are roughly the same(the difference is no larger than 1% in our experiments). One additional detail concerning the implementation of updating A(k)-index also need to be pointed out. To facilitate maintaining the

connectivity between U and V after some data nodes are split from V, we keep an additional parameter for each edge $U \rightarrow V$ in the A(k)-index graph that records the number of edges between $extent(U)$ and $extent(V)$ in the source graph data. With this parameter, instead of scanning all data nodes in $extent(U)$ or $extent(U)$, we only need to check parents of split data nodes, which are usually much less in number than either $extent(U)$ or $extent(U)$, to maintaining connectivities after splitting. Our experiments show that this additional parameter can speed up the updating process on A(k)-index by up to 5 to 6 times in many tested cases. Note that the results presented in the conference version of our work is based on implementations without such parameter.

We randomly add 300, 600, 900 or 1200 new edges to data graphs, and measure the running time of the update algorithms for A(1) up to A(5), and D(k). Note that updating the A(0) index is trivial since it does not involve any splitting. Due to the unavailability of more accurate cost models for structural summaries, we assume that both the source graph data and index graph are in main memory. In the real application scenario, we can expect that only the index graph is in main memory, but not the source graph data; therefore, the performance advantage of incrementally updating D(k)-index over A(k)-index should be more impressive than what we present here. Our machine features the Linux OS, the Pentium 2.0 Ghz processor and the 512 RAM. The detailed results on the Xmark and Nasa datasets are given in Figure 2.6 and 2.7, in which the running time is the total accumulative time to perform all updates. On both datasets, updating D(k)-index takes roughly the same time as updating A(1) or A(2)-index, but takes less time than updating A(3), A(4) or A(5)-indexes. More specifically, the cost of updating A(k)-index shoot up dramatically as k increases from 4 to 5 on the Xmark data; on the Nasa data, the performance difference between D(k) and A(4) or A(5) is quite significant. Noting the fact that A(5)-index achieves the best evaluation

performance among A(k)-indexes on both datasets(for Xmark, it is about twice the evaluation cost of D(k)-index; for the Nasa data, it is about 1.2 times the evaluation cost of D(k)-index), we demonstrate experimentally that updating the D(k)-index can be accomplished much more efficiently than updating the A(k)-index with comparable evaluation performance.

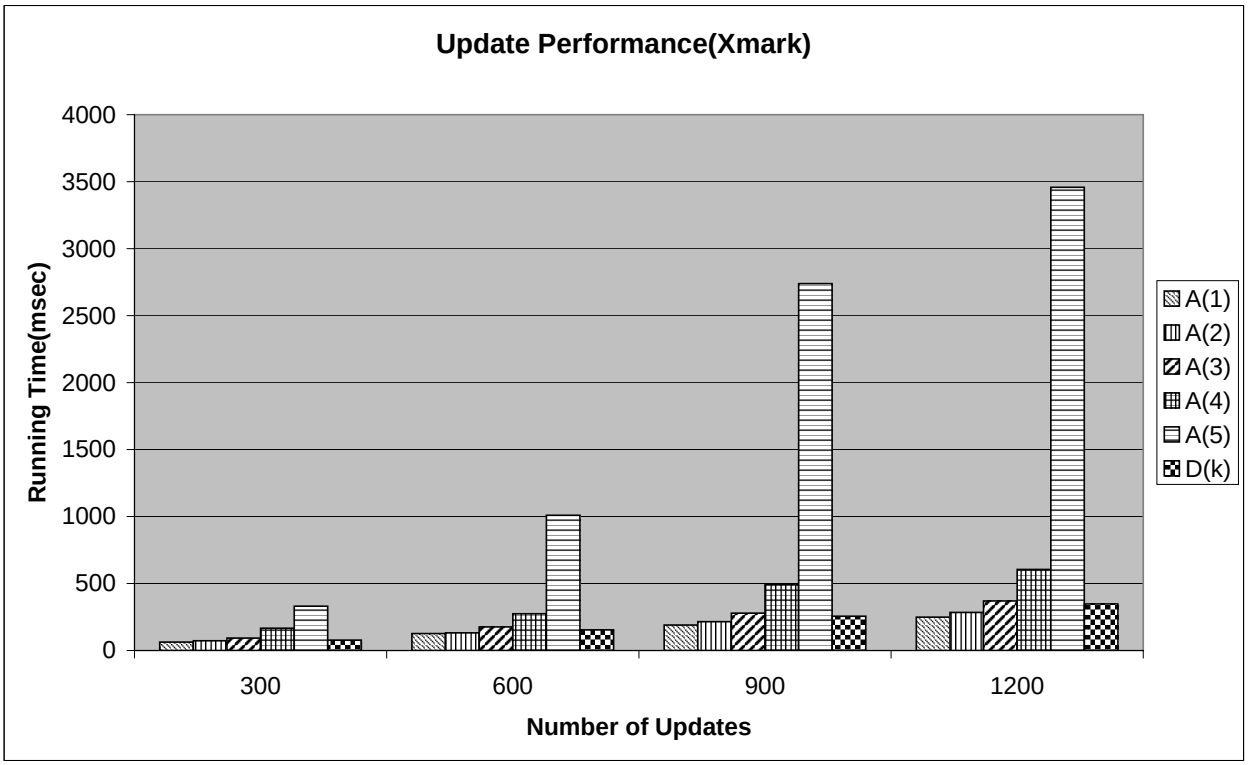


Figure 2.6: Update Performance Comparison Between A(k) and D(k) on Xmark Data

2.6.3 Maintaining A(k) and D(k)-Index

As the A(k)-index is incrementally updated, its evaluation performance may suffer since its size may increase. As for the D(k)-index, its evaluation performance may also deteriorate since its index nodes' local similarities may have been downgraded and the evaluation thus triggers more validations.

We track the size increase of A(k)-index over a sequence of 300 incremental updates(edge insertions) on both datasets. Results are presented in Figure 2.8

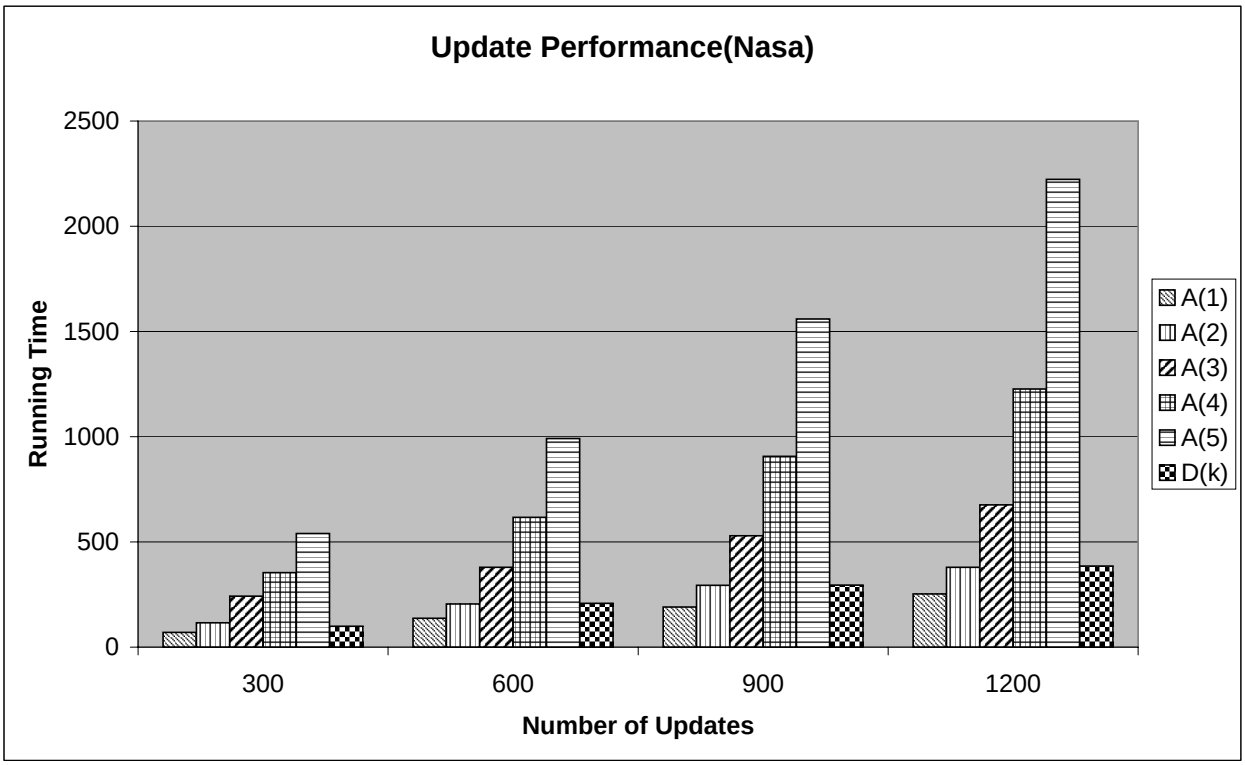


Figure 2.7: Update Performance Comparison Between A(k) and D(k) on Nasa Data

and 2.9. We can see that sizes of A(k)-indexes increase steadily as updates go on and A(k)-indexes with low values of $k(\leq 3)$ have sharper percentage size increase than ones with high values of k(4 or 5). The total A(k) index size increases after 300 updates on Xmark data reach more than 100% when $k = 2or3$; but the size increases are more moderate as k grows larger, 45% for k=4 and 13% for k=5. On the Nasa data, our experiments show that the total A(k)-index size percentage increases are quite moderate for all range values of k. The maximum is roughly 20% when $k = 1or2$; for k=3, 4 or 5, the increases are no larger than 10%.

We also track the evaluation performance of A(k) and D(k)-index over a sequence of 300 incremental updates. Results are presented in Figure 2.10 and 2.11. Note that we only show A(k) for k is between 3 and 5. A(1) and A(2)-index have the much worse evaluation performance, thus are neglected in figures; but their performance degradation follow the same trend as those of A(k)-index with $3 \leq k \leq 5$.

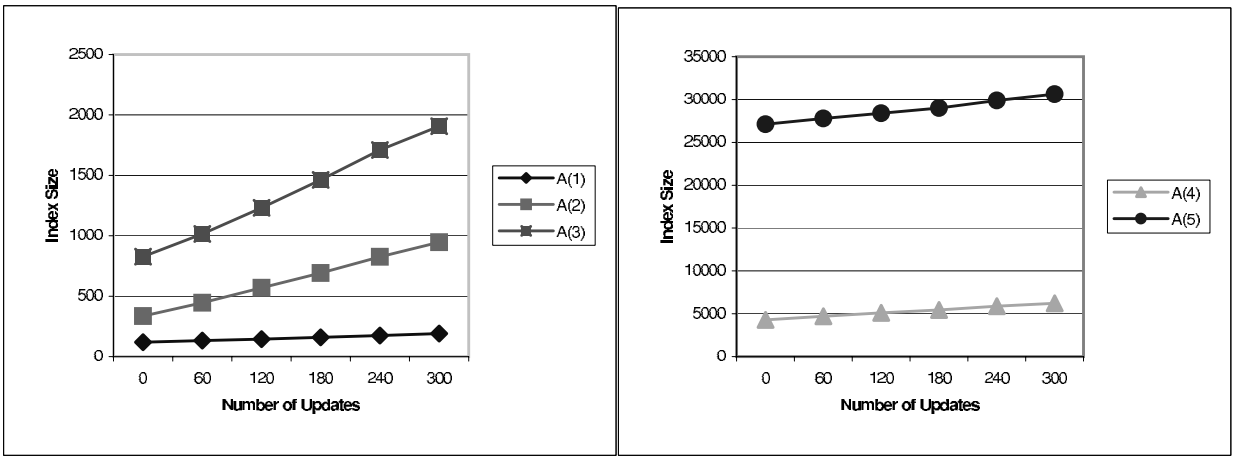


Figure 2.8: Size Increase of A(k)-Index over Incremental Updates on Xmark Data

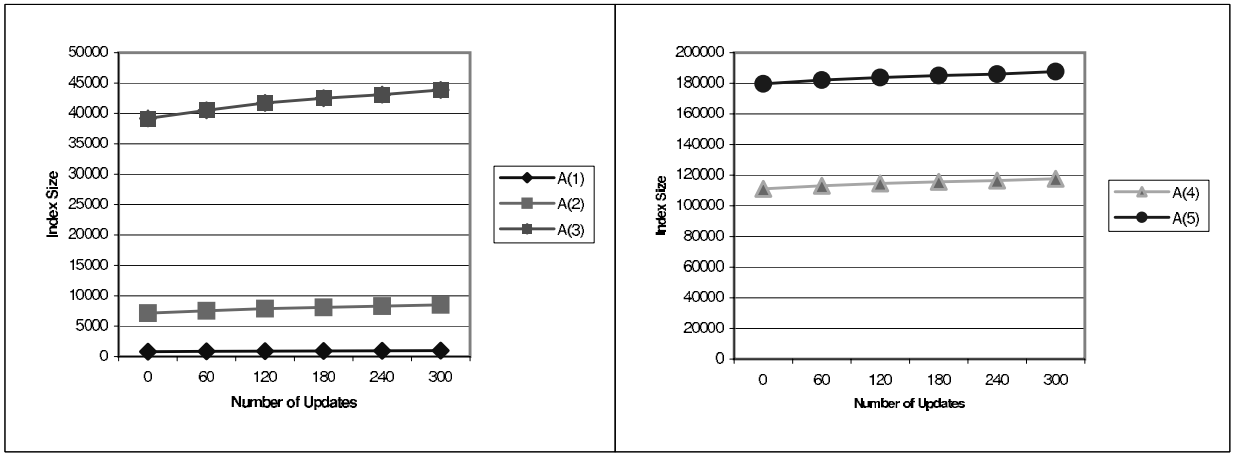


Figure 2.9: Size Increase of A(k)-Index over Incremental Updates on Nasa Data

We have the observation that even though A(k)-index sizes may increase considerably as demonstrated before, its performance degradation is insignificant, less than one percent in our experiments. If k is small, A(k)-index's evaluation cost is dominated by the validation process on the source data; therefore, even though the evaluation cost on the index graph may increase as the index size becomes larger, the overall evaluation cost remains roughly unchanged since the evaluation cost on the index graph represents only quite a small portion. Otherwise, if k is large, the percentage increase of index size becomes small; therefore, its evaluation performance does not fluctuates much either. Compared with A(k)-index, the performance degradation of the D(k)-index is sharper. On the Xmark data, with 60 updates, the D(k)-index underperforms both the A(4) and A(5)-index. Up to 180 updates, the performance of D(k)-index still suffers visibly; after that the degradation gradually flattens out. We have the similar observation on Nasa data. The D(k)-index underperforms A(5)-index after 60 updates; then its performance gradually stabilizes. These observations experimentally verify that the downgrading of local similarities can severely affect D(k)-index's performance; thus justify the necessity of maintaining D(k)-index periodically.

Since sizes of A(k)-index may increase and performance of D(k)-index may suffer as a result of incremental updates, both A(k) and D(k)-index need to be maintained periodically. For the A(k)-index, the maintenance involves merging index nodes with the same label if they satisfy k-bisimilarity in the index graph. Maintaining D(k)-index involves both the promotion and merging processes: index nodes are first promoted to their target local similarities and the merging process is then invoked to shrink the index size. We implement the merging operation by treating the original index graph as a data graph and building the new A(k) or D(k)-index from it. The D(k)-index is promoted through the **Mass_Promoting_Algorithm**.

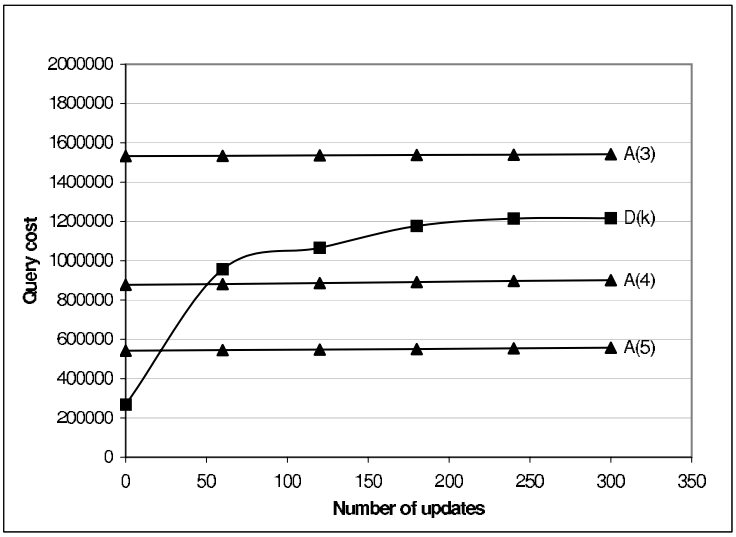


Figure 2.10: Performance Degradation of A(k) and D(k)-index over Incremental Updates on Xmark Data

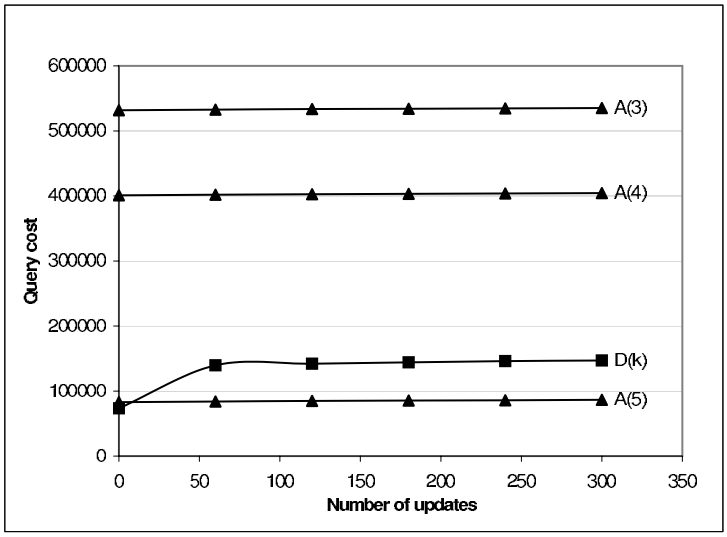


Figure 2.11: Performance Degradation of A(k) and D(k)-index over Incremental Updates on Nasa Data

The maintainance cost comparison between A(k) and D(k)-index are shown in Figure 2.12 and 2.13. Not surprisingly, maintaining D(k)-index is more computationally expensive than maintaining A(k)-index on both datasets. We think that the relative higher maintenance cost of the D(k)-index should not be of much concern because of two reasons: (1) in real applications, updates should be performed much less frequently than queries; and the maintenance process is only invoked after a considerable number of updates; (2) the maintenance operation can significantly improve the evaluation performance of D(k)-index; if we factor the query benefit into the consideration, it is quite a fair price to pay for the improved query performance. The effectiveness of the maintenance operation to improve D(k)-index's evaluation performance are shown in Figure 2.14 and 2.15. For the A(k)-index, even though the index graph can be shrunk to some extent, the overall performance remains roughly the same. In contrast, the performance gain on the D(k)-index is more striking. On the Nasa data, the evaluation cost is actually cut by half after the maintenance; the two dots before and after maintenance appear close because of the large value of Y-axis.

2.7 Summary

In this chapter, we propose the D(k)-index, which 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

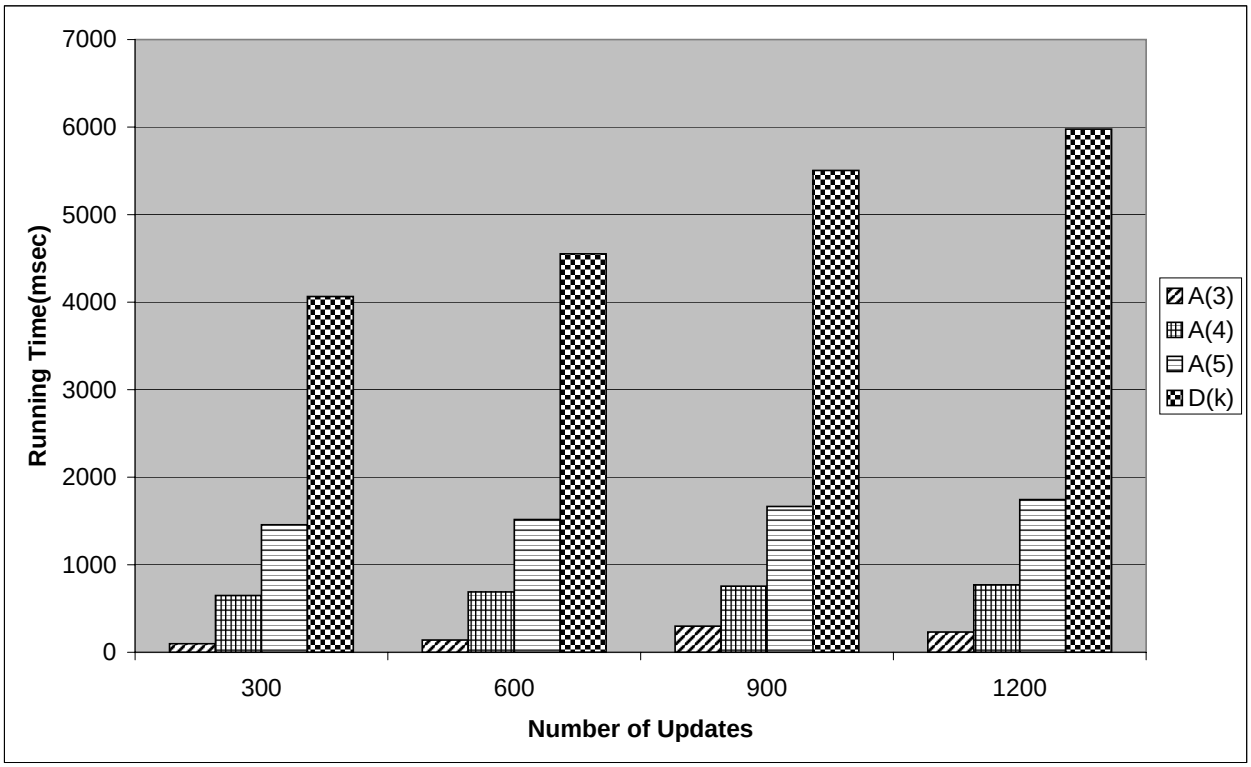


Figure 2.12: Maintenance Cost of A(k) and D(k)-index on Xmark Data

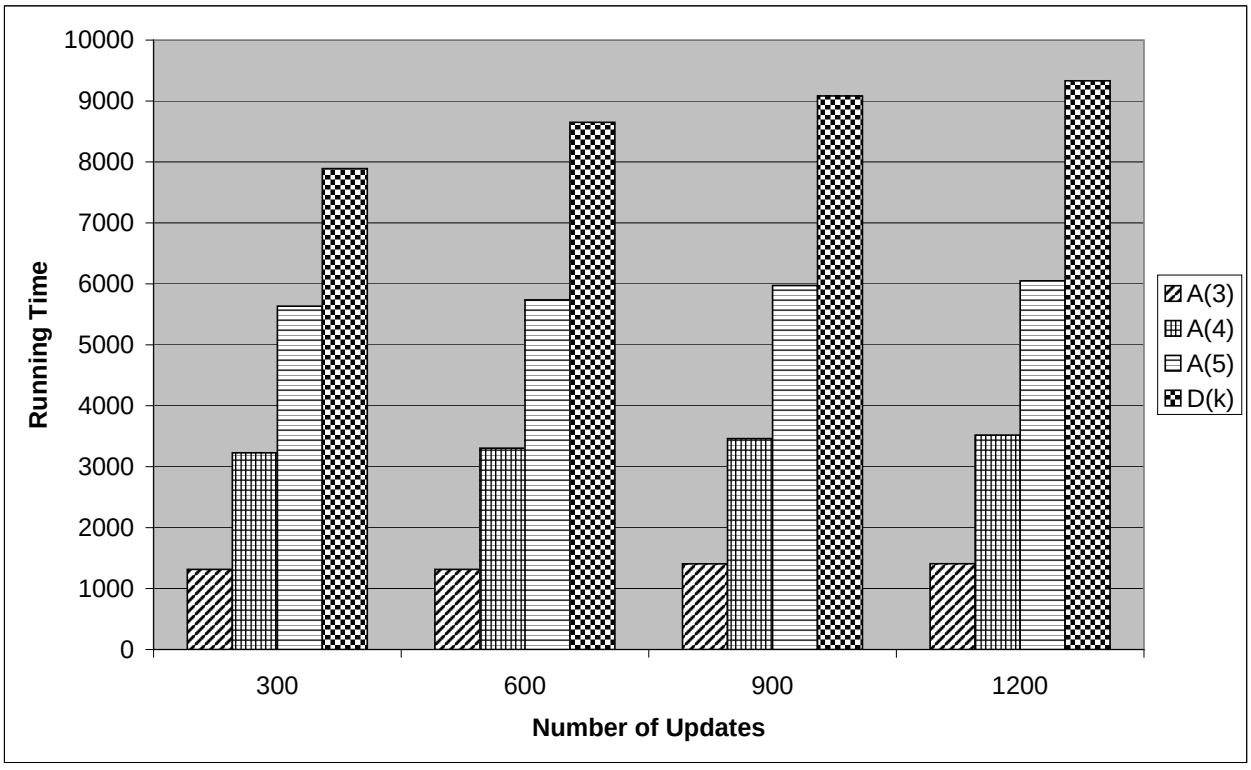


Figure 2.13: Maintenance Cost of A(k) and D(k)-index on Nasa Data

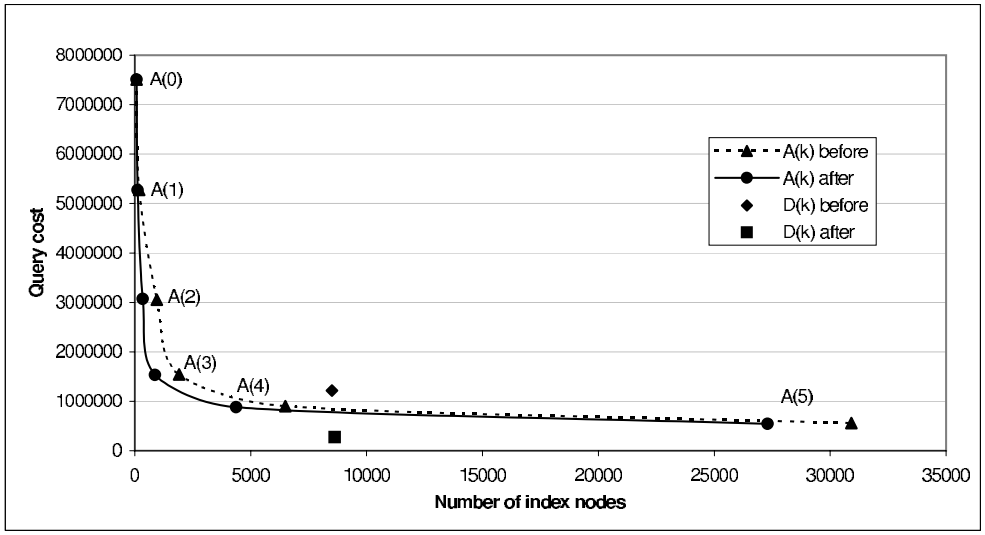


Figure 2.14: Performance Improvement after Maintaining A(k) and D(k)-index on Xmark Data

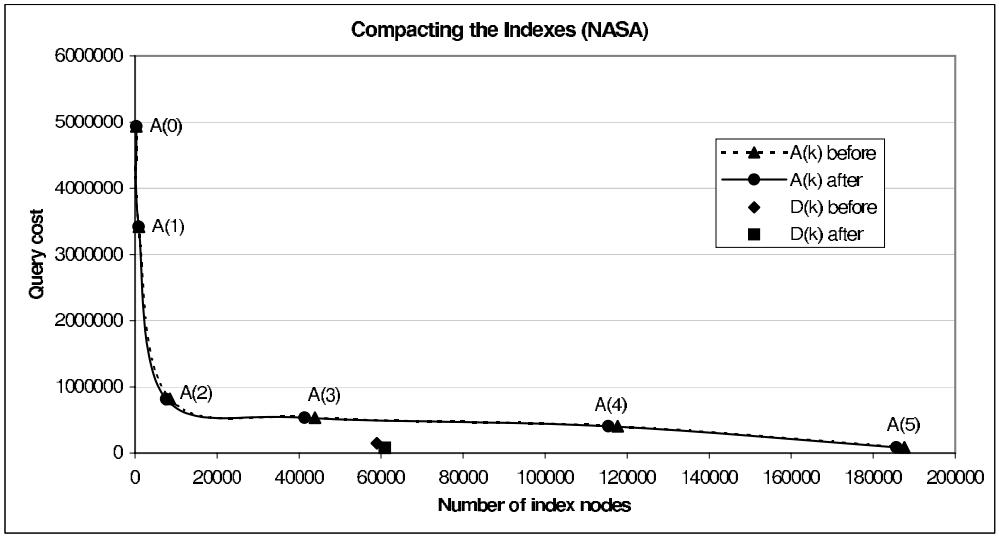


Figure 2.15: Performance Improvement after Maintaining A(k) and D(k)-index on Nasa Data

the superiority of the update operations on the D(k)-index over the update operations proposed for previous summary structures.

Chapter 3

Indexing XML for Xpath Querying in External Memory

One major shortcoming of the structural summary is that it can only be used to evaluate the non-branching *regular path expression*. As the XPath specification shows, possible XML query patterns are beyond that scope. For instance, the presence of branching predicates in an XPath path expression can actually make it correspond to a twig path pattern. Secondly, the evaluation of path expressions on the structural summary still demands the possibly exhaustive traversal of the summary graph. The element-set-based query processing has the advantage that only related elements, whose labels are in the specified query, are involved in the searching process. Nodes in XML tree are usually encoded such that the structural relationship between two nodes can be decided from their codes alone. As a result, pairs of elements satisfying the specified structural relationship can be found through *structural join*. It is worth noting that only the two element sets are required for the structural join between two labels, but not the original XML tree. Thus, the potentially time-consuming traversal of trees or graphs is avoided.

Previous encoding schemes and external-memory index structures proposed

for XML mainly considered the containment relationship between XML elements, specifically parent-child or ancestor-descendant relationship. The presence of *preceding-sibling* and *following-sibling* location steps in the XPath specification makes it clear that the horizontal navigation, besides the vertical navigation, in XML documents, are necessary for efficient evaluation of XPath queries. In this chapter, we enhance the existing two encoding schemes, range-based and prefix-based, such that all possible structural relationship, specified in the XPath language, between two elements can be determined from their codes alone. Next, we propose an external-memory index structure, the $XL+$(XML Location)-Tree, which is based on the $B+$-Tree. It indexes element sets to facilitate all location steps, *vertical* and *horizontal*, *top-down* and *bottom-up*, defined in XPath. The $XL+$-Trees based on the prefix-based or range-based encoding schemes basically share the same structure. We analyze the I/O cost of the search and update operations on the $XL+$-Tree and wrap up this chapter with extensive experiments validating its effectivity. We note that previous works on supporting comprehensive XPath locating steps focused on querying XML documents by taking advantage of the popular relational engines. Therefore, they adopted the *status quo* external memory index structures in relational engines, namely B-Tree and R-Tree, for the query optimization. In contrast, the $XL+$-Tree is an enhanced index structure based on the $B+-Tree$ that specifically supports efficient structural navigations on XML documents as specified in the XPath query language.

3.1 Introduction

One popular type of encoding technique is the range labeling [34, 35, 45, 47], which is inherited from the inverted list widely adopted in information retrieval(IR)[43, 44]. This scheme encodes each element, v, with a pair of integers (L_v, R_v) such

that an element v is an ancestor of u iff $L_v < L_u < R_u < R_v$. The other type is the prefix labeling [46]. It labels each element with a unique string S such that an element v is the ancestor of u iff $S(v)$ is a prefix of $S(u)$. These indexes enable the element-sets-based query on XML documents. Since we can decide the containment relationship between two elements from their labels alone, structural join is usually used to find all pairs of elements satisfying the primitive structural relationship, namely, *parent-child* and *ancestor-descendant* relationships. Equipped with advanced index data structures [47, 48, 49, 50], structural join can be performed quite efficiently, specifically in linear or even sublinear time. We note that these indexes were mainly designed to facilitate the containment relationship evaluation.

Besides the well studied containment relationship, the XPath language also specifies the sibling structural relationship between XML elements. The *preceding-sibling* and *following-sibling* axes enable the horizontal navigation among tree nodes, which we believe is an important query pattern for XML database. Therefore, sibling structural join, as well as the containment structural join, should be dealt with while we build index structures for XML databases. In this chapter, we begin with the enhanced ranged-based and prefix-based encoding schemes for elements in XML trees. Our schemes add additional parameters to the traditional labeling schemes such that all structural relationship specified in the XPath language between two nodes can be determined from their codes alone. Then we proceed to propose an $B+$-Tree based external-memory index structure, $XL+$-Tree, which facilitates comprehensive types of structural navigation on XML trees. The $XL+$-Trees based on the range or prefix encoding schemes factually share the same structure. But their search operations are slightly different because the richer information provided by the prefix encoding is exploited to improve the search performance on $XL+$-Tree.

Our external-memory index structure, $XL+$-tree, is built with *left position*s

of ranges(under the range encoding scheme) or labeling strings(under the prefix encoding scheme) as keys. We note that previous index structures based on the range encoding scheme mainly considered the containment structural relationship. Existing index structures proposed for strings [53, 54, 55, 56] was intended to support two types of search problems: (1) prefix search and range query: prefix search retrieves all strings whose prefix is the given string S; range query retrieves all strings between S_1 and S_2 in lexicographic order; (2) substring search: substring search finds all occurrences of a given string pattern in strings. To support the XPath evaluation, the substring search operation is no longer required for the XL+-tree. However, new string searching operations emerge because of the variety of the XPath location steps. Detailed definitions of search operations under the range or prefix encoding schemes are described in Section 3.3. To cut short, the XL+-tree targets three types of search problems corresponding the *top-down*, *bottom-up* and *horizontal* navigations among XML tree nodes respectively. Let B denotes the disk size. And k denotes the total number of indexed entries. Our major results can be summarized as follows:

1. Analytical Results(regardless of the underlying encoding scheme, range or prefix). The *descendant* search operation takes $\mathbf{O}(log_B k + \frac{rs}{B})$ worst-case disk accesses, where rs is the size of the result; the *children* search operation takes $\mathbf{O}(log_B k + rd)$ worst-case I/Os, where rd is the number of disk pages storing results. Please note that rd may not be equal to $\frac{rs}{B}$ because children may not be stored contiguously in the XL+-tree. The *following-sibling* and *preceding-sibling* search operations both take the $\mathbf{O}(log_B k + rd)$ worst-case I/O cost as well. The *parent* search operation takes $\mathbf{O}(log_B k)$ worst-case I/O cost, while the *ancestor* search takes $\mathbf{O}(lv \times log_B k)$ I/O cost in the worst case, where lv is the level number of the input entry. For the update operations on XL+-Tree, both the insertion and deletion operation take $\mathbf{O}(log_B k)$ amortized I/O

cost.

2. Experimental Evaluation. As far as we know, there is no external-memory index structure specifically designed for handling comprehensive Xpath location steps. The Xpath query accelerator, proposed in [60], encodes each node in an XML tree with a multi-dimensional descriptor and takes advantage of traditional R-Tree and B-Tree to support various Xpath locating processes on a relational engine. Since R-Tree has been shown to outperform B-Tree in their experiments, to validate the effectivity of the $XL+$-tree, we compare its performance with that of R-Tree. Our experiments on both benchmark and synthetic XML data demonstrate that the $XL+$-tree outperforms R-Tree by wide margin in most cases in term of both I/O and CPU cost.

3.2 Enhanced Encoding Schemes

3.2.1 Range-Based Encoding Scheme

In the traditional range encoding scheme, positions of nodes in XML trees are represented by 3-tuple *(DocNo,LeftPos:RightPos, LevelNo)*. *DocNo* is the identifier of document. The pair of *LeftPos* and *RightPos* can be generated by doing a depth-first traversal of the tree and sequentially assigning a number at each visit. Since each no-leaf node is always traversed twice, once before all its children and once after, it has two numbers assigned, while leaf nodes have only one number. *LevelNo* is the nesting depth of nodes in the tree. An instance of the range encoding of an XML tree is shown in Figure 3.1.

With the range encoded representation of an XML tree, the containment structural relationship between tree nodes can be determined easily: (1) containment or ancestor-descendant: a tree node n_1, $(LP_1 : RP_1, lv_1)$, contains a tree node n_2, $(LP_2 : RP_2, lv_2)$, if and only if $LP_1 < LP_2$ and $RP_1 > RP_2$; (2) direct containment

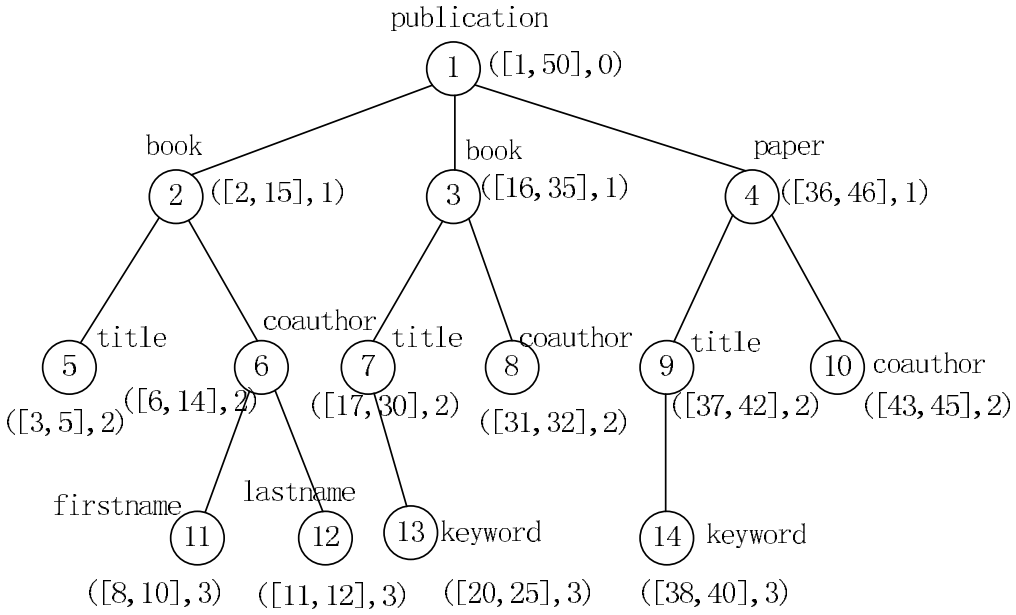


Figure 3.1: The Range Encoding of An XML Tree

or parent-child: a tree node n_1 directly contains n_2 if and only if $LP_1 < LP_2$, $RP_1 > RP_2$ and $lv_1 = lv_2 - 1$. One advantage of this presentation is that checking an ancestor-descendant structural relationship is as easy as checking a parent-child structural relationship.

Now we consider the types of structural navigation required by XPath. Of all 13 types of location steps specified in the XPath language, *attribute* and *namespace* are the same as *child* from the structural point of view since attributes and namespaces can be treated as special types of elements; the axis *self* has no evaluation cost; axes *descendant-or-self* and *ancestor-or-self* are just like axes *descendant* and *ancestor* respectively, plus the context node. The remaining four pairs of exes are of primary interest to us. The pair of axes, *child* and *descendant*, represent the vertical top-down traversal. The pair of axes *parent* and *ancestor* make the vertical bottom-up traversal. The pair of axes *preceding-sibling* and *following-sibling* typifies the horizontal traversal. Finally, axes *preceding* and *following* make the general forward and backward traversal respectively.

While the traditional range encoding of XML trees is sufficient to determine the *parent/child*, *ancestor/descendant*, and *following/preceding* relationships between tree nodes, it does not capture the *preceding-sibling/following-sibling* relationship. The enhanced range encoding scheme represents each node with a three-dimensional descriptor: $< LP : RP, lv, P\_LP >$, in which LP and RP represent its position range, lv is its nesting level and P_LP is its parent node's left position. Note that we assume that nodes are from the same document and ignore the $DocNo$ information from the descriptor. Extending it to handle nodes across multiple documents should be trivial. Based on this encoding scheme, the structural relationship between XML tree nodes can be determined as follows:

1. *descendant*. Node u is a descendant node of v iff $LP(v) < LP(u)$ and $RP(u) < RP(v)$;

2. *child*. Node u is a child node of v iff $LP(v) < LP(u)$, $RP(u) < RP(v)$ and $lv(u) = lv(v) + 1$;

3. *ancestor*. Node u is an ancestor node of v iff $LP(u) < LP(v)$ and $RP(v) < RP(u)$;

4. *parent*. Node u is a parent node of v iff $LP(u) = P\_LP(v)$;

5. *following-sibling*. Node u is the following-sibling node of v iff $LP(u) > LP(v)$ and $P\_LP(u) = P\_LP(v)$;

6. *preceding-sibling*. Node u is the preceding-sibling node of v iff $LP(u) < LP(v)$ and $P\_LP(u) = P\_LP(v)$;

7. *following*. Node u is the following node of v iff $LP(u) > RP(v)$;

8. *preceding*. Node u is the preceding node of v iff $RP(u) < LP(v)$;

Note that the index $XL+$-Tree is built using the LP values of data nodes as keys. Given a context node v, the *following* location step can be accomplished by a simple range search that identifies those data nodes satisfying $LP > RP(v)$. Since node v's *preceding* nodes are defined to be those nodes with $LP < LP(v)$, but excluding v's ancestor nodes, the *preceding* location step can be accomplished by a range search identifying data nodes with $LP < LP(v)$ followed by a *ancestor* search operation. The following three search problems, which corresponds to six basic location steps, are critical to the XPath processing;therefore, they are of primary interest to us. The *R*ange *E*ncoding *S*cheme is denoted by *RES*.

Given a set of node descriptors, $D = \{D_1, D_2, \ldots, D_k\}$, and an input descriptor $D(v) =< LP(v) : RP(v), lv(v), P\_LP(v) >$:

Definition 4 Top-Down Search(RES)*: Search_Descendent(D(v)) retrieves all descriptors in D satisfying $LP(v) < LP < RP(v)$; Search_Children(D(v)) retrieves all descriptors in D satisfying $LP(v) < LP < RP(v)$ and $lv = lv(v) + 1$.*

Definition 5 Bottom-Up Search(RES)*: Search_Ancestors(D(v)) retrieves all descriptors in D satisfying $LP < LP(v) < RP$; Search_Parent(D(v)) retrieves the descriptor satisfying $LP = P\_LP(v)$.*

Definition 6 Horizontal Search(RES)*: Search_Following-Siblings(D(v)) retrieves all descriptors in D satisfying $LP > RP(v)$ and $P\_LP = P\_LP(v)$; Search_Preceding-Siblings(D(v)) retrieves all descriptors in D satisfying $LP < LP(v)$ and $P\_LP = P\_LP(v)$.*

3.2.2 Prefix-Based Encoding Scheme

In the prefix labeling scheme, we encode each node with a unique string S such that: (1) $S(v)$ is before $S(u)$ in lexicographic order iff node v is before node u in the document order; (2) $S(v)$ is a prefix of $S(u)$ iff node v is the ancestor of node

u. Informally, the document order in an XML tree orders its nodes corresponding to a sequential read of nodes by a preorder traversal. One simple example prefix scheme works as follows. We assign to the out-going edges of each node a set of prefix-free binary strings. From left to right, strings assigned to edges are in lexicographic order. Then, starting from the root and going down, we define the label of each node to be the concatenation of its parent's label and the string assigned to its incoming edge. Consider, for example, a node v has two children, v_1 and v_2, and v_1 is before v_2. We can assign string "00" to edge (v, v_1), string "01" to edge (v, v_2). So the label string of v_1, $S(v_1)$=$S(v) \bullet 00$; the label string of v_2, $S(v_2)$=$S(v) \bullet 01$. The labeling of the example XML tree of Figure 1.1 is given in Figure 3.2. Please note that the problem of how to label nodes in the XML tree using the shortest possible string in the static or dynamic setting is beyond the scope of this dissertation. In this chapter, we use the above-mentioned scheme to explain our results. However, our results are valid for any prefix labeling scheme satisfying the above two conditions.

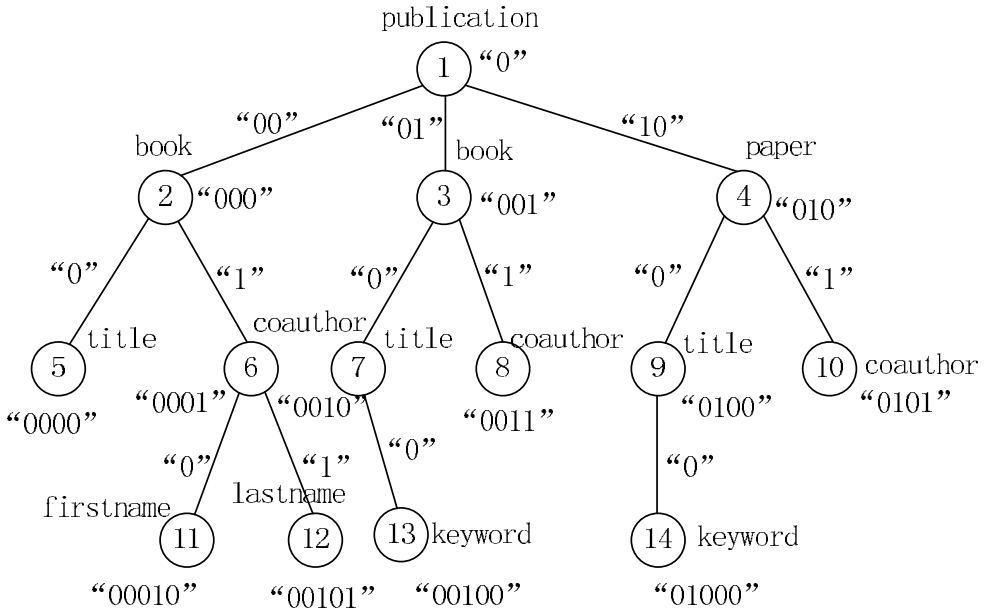


Figure 3.2: The Prefix Encoding of An XML Tree

As in the range-based labeling scheme, we record nodes' level numbers to distinguish the parent-child and ancestor-descendant relationship. Another parameter we keep for each node records the lengths of the strings assigned to its incoming edges. We have the following two observations:

1. The basic component of strings, *character*(referred to as the *char* data type in most programming languages), has up to 256 distinct values; therefore, the strings with maximum length of 5 can represent up to $256^5 (\gg 1$ billion) distinct values. Thus, the length of strings assigned to edges in an XML tree can afford to be small;

2. The maximal level of the XML tree can be expected to be small also. Authors in [46] said that the average depth of XML files collected by a crawler over the web is low; the trees are balanced with relatively high degrees. The popular DBLP document and Xmark benchmark data have the maximal level no larger than 12.

Suppose that the maximal length of labeling strings over edges of XML tree is m. We set the base length value b to be $(m+1)$. A node v at level k(with the root at level 0) has the incoming path of $n_0 n_1 \ldots n_k$, in which node n_0 is the root of the XML tree and $v = n_k$. And the length of the labeling string over edge $n_{i-1} \rightarrow n_i$, for each $1 \leq i \leq k$, is l_{i-1}. We add an integer parameter, the *edge string length* $esl = l_0 \times b^{k-1} + l_1 \times b^{k-2} + \ldots + l_{k-1} \times b^0$, to each node v's descriptor. Obviously, we can determine the values of all l_is from esl's value, specifically $l_i = \lfloor \frac{esl}{b^{((k-1)-i)}} \rfloor$ *%(modula) b*. The *edge string length* parameter will be used to extract a node' ancestors' label strings. This completes our enhanced prefix encoding scheme. Each node v in the XML tree is represented by a three-dimensional descriptor: $< S, lv, esl >$, in which, lv is the nesting level of node v and esl is the *edge string length* parameter defined above.

Under this prefix encoding scheme, the structural relationships between nodes can be determined as follows. Note that the function $prefix(S, i)$ returns the string consisting of the first i characters in string S. And we denote it as $S(v) < S(u)$ iff $S(v)$ is before $S(u)$ in lexicographic order; $S(v) > S(u)$ iff $S(v)$ is after $S(u)$ in lexicographic order. We also denote the lengths of strings over edges of node v's incoming path, in the order from the root to v, as $l_0(v), l_1(v), \ldots, l_k(v)(k{=}lv(v){-}1)$.

1. *descendant*. Node u is a descendant node of v iff $S(v)$ is a prefix of $S(u)$;
2. *child*. Node u is a child node of v iff $S(v)$ is a prefix of $S(u)$, and $lv(u) = lv(v) + 1$;
3. *ancestor*. Node u is an ancestor node of v iff $S(u)$ is a prefix of $S(v)$;
4. *parent*. Node u is a parent node of v iff $S(u) = prefix(S(v), |S(v)| - e_k(v))$;
5. *following-sibling*. Node u is the following-sibling node of v iff $S(u) > S(v)$, $lv(u) = lv(v)$, and $prefix(S(v), |S(v)| - l_k(v))$ is a prefix of $S(u)$;
6. *preceding-sibling*. Node u is the preceding-sibling node of v iff $S(u) < S(v)$, $lv(u) = lv(v)$, and $prefix(S(v), |S(v)| - l_k(v))$ is a prefix of $S(u)$;
7. *following*. Node u is the following node of v iff $S(u) > S(v)$, and $S(v)$ is **NOT** a prefix of $S(u)$;
8. *preceding*. Node u is the preceding node of v iff $S(u) < S(v)$, and $S(u)$ is **NOT** a prefix of $S(v)$.

Similar to the case of the range encoding scheme, nodes *preceding* a given node v are those nodes whose labeling strings are smaller than $S(v)$, but excluding node v's ancestors. Thus, the *preceding* location step can be solved by performing a range string search followed by a string search corresponding to the *ancestor* location step. Nodes *following* a given node v should have a labeling string larger than $S =$

$prefix(S(v), |S(v)|-e_k(v))\bullet\omega$, in which $k = lv(v)-1$, $\bullet$ is a concatenation operator and ω is an imaginary character larger than any other character. Therefore, the *following* location step actually amounts to a range search on strings. The three search problems under the *P*refix *E*ncoding *S*cheme(*PES*), which correspond to six basic XPath location steps, are presented as follows.

Given a set of node descriptors, $D = \{D_1, D_2, \ldots, D_k\}$, and an input descriptor $D(v) =< S(v), lv(v), esl(v) >$:

Definition 7 Top-Down Search(PES)*: Search_Descendent(D(v)) retrieves all descriptors in D satisfying that their labeling strings have S(v) as a prefix; Search_Children(D(v)) retrieves all descriptors in D whose labeling strings have S(v) as prefix, and whose level number is* $(lv(v) + 1)$.

Definition 8 Bottom-Up Search(PES)*: Search_Ancestors(D(v)) retrieves all descriptors in D whose label strings are prefixes of S(v); Search_Parent(D(v)) retrieves the descriptor whose labeling string is* $prefix(S(v), |S(v)|-e_k(v))$*(k=*$lv(v)-1$*).*

Definition 9 Horizontal Search(PES)*: Search_Following-Siblings(D(v)) retrieves all descriptors in D satisfying* $S > S(v)$,$lv = lv(v)$,*and S has* $prefix(S(v), |S(v)|-e_k(v))$ *as a prefix; Search_Preceding-Siblings(D(v)) retrieves all descriptors in D satisfying* $S < S(v)$, $lv = lv(v)$, *and S has* $prefix(S(v), |S(v)| - e_k(v))$ *as a prefix.*

It is interesting to note that while the top-down search problem is similar to the string prefix search problem well studied in previous literature; the bottom-up and horizontal search problems are specific to the XPath evaluation.

3.3 The XL+-Tree for Range Encoding Scheme

The XL+-tree under the range encoding scheme is an extension of the B^+-tree index data structure, in which node descriptors are stored on leaf disk pages and

all leaves are linked sequentially. Entries are sorted according to left positions(LP in our node descriptor). In the XPath specification, each location step usually comes with a node test, specifically an element tag test. Therefore, in our design, an XL+-tree is built for each tag in XML documents. In case that there are so many distinct tags that $XL+-Tree$s may flood the XML query engine, node descriptors with different tags can be actually indexed in a single $XL+-tree$; but it has a composite key, (tag,LP). For convenience of explanation, we will focus on the $XL+-tree$ built for a single tag in this section. Extending it to handle multiple tags should be straightforward.

The overall structure of XL+-tree is shown in Figure 3.3. Each entry in the XL+-tree leaf pages consists of the descriptor and two pointers, one referring to its immediate *preceding sibling* and the other referring to its immediate *following sibling*. The structure of the XL+-Tree's internal page is basically the same as in the B+-tree except that we store two additional integers on each reference to its child page. These two integers record the minimal and maximal level(lv) of entries(node descriptors) in the corresponding subtree respectively. As it will be shown later, the pair of additional integers is for identifying the *first child/sibling* of a given context node; the pair of pointers in each entry is for facilitating the horizontal navigation.

3.3.1 Search Operations on XL+-tree

Given a target node descriptor,$D(v) =< id(v), LP(v) : RP(v), lv(v), ParentId(v) >$, its position in the indexed descriptors is defined to be the position of the leftmost entry whose LP is larger or equal to $LP(v)$. We denote its position by (δ_i, p_j), with δ_i representing the ith leaf disk page and p_j representing the position on this disk page. The procedure, $Find\_Position(D(v))$, which identify $D(v)$'s position, can be implemented by repeatedly performing a binary search in the integers stored

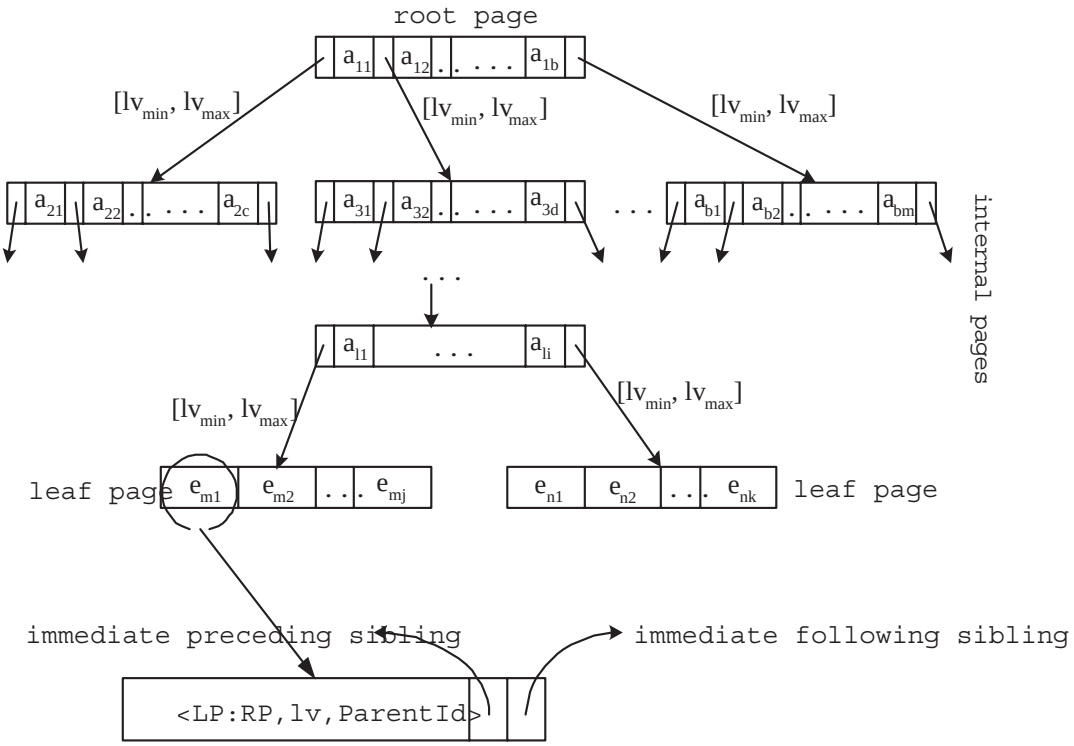


Figure 3.3: The Overall Structure of XL^+-tree

on nodes in the XL+-tree. Its details are omitted here since it is the standard operation on the traditional B^+-tree.

Top-Down Search:Descendant and Children

Since all $D(v)$'s descendent satisfy $LP(v) < LP < RP(v)$, the *Search_Descendant(*$D(v)$*)* operation amounts to the range search operation on the XL+-tree. It can simply be implemented by the $Find\_Position(D(v))$ operation followed by sequentially scanning entries until $LP \geq RP(v)$. Therefore, the $Search\_Descendant(D(v))$ worst case I/O cost is $\mathbf{O}(\frac{rs}{B} + log_B k)$, in which rs is the number of descendant entries.

For the $Search\_Children(D(v))$ operation, we have the observation that, once $D(v)$'s first child is found, its other children can be identified by simply following the *following-sibling* pointers. $D(v)$'s first child can be found through the

$Find\_Position(D(v))$ operation followed by sequentially scanning entries. Unfortunately, this implementation has the same worst case I/O cost as the $Search\_Descendent(D(v))$ operation. In case that there are a lot of $D(v)$'s descendants before its first child, its efficiency suffers. In the scenario of Figure 3.4, it needs to scan B_d pages before finding the first child. Instead, we present a procedure that takes $\mathbf{O}(log_B k)$ I/Os in the worst case to identify the first child. Our approach takes advantage of pairs of integers stored over page references in the XL+-tree. Note that the pair of integers over the reference to page δ keeps the minimal and maximal level(lv) of entries in the subtree rooted at page δ. Firstly, we have the following lemma:

Lemma 2 *$D(v)$'s first child, if it exists, is the first(leftmost) entry satisfying $LP > LP(v)$ and $lv = (lv(v) + 1)$ in the XL+-Tree; and all entries before it but with $LP > LP(v)$ have level $lv > lv(v) + 1$.*

Proof: Since $D(v)$'s first child $D(u)$ satisfies $LP(v) < LP(u) < RP(v)$, all entries before $D(u)$ but with $LP > LP(v)$ also satisfy $LP(v) < LP < RP(v)$; thus they are all $D(v)$'s descendants, but not children because $D(u)$ is $D(v)$'s first child. □

The procedure for identifying $D(v)$'s first child involves two phases:(1) a top-down search; (2) if needed, backtracking and another top-down search. The first top-down search begins with the root page of XL+-tree and recursively advances to the next target page until it reaches a leaf page or the stop criteria is met, which means either the end of the first phase or the non-existence of $D(v)$'s first child. It goes through two steps on each page. Firstly, it chooses the *leftmost* page reference, PR_i, whose corresponding subtree has a range of keys(LPs) $(LP_{min}, LP_{max}]$ satisfying $LP(v) \leq LP_{max}$. Note that the values of LP_{min} and LP_{max} are two delimiting integers of each page reference and this search process can be accomplished through a binary search over delimiting keys as on the traditional B+-tree. Secondly, if the chosen page reference's range of levels $[lv_{min}, lv_{max}]$ contains $lv(v)+1$, which means

that $D(v)$'s first child is *probably* in this subtree, it advances to the next page following this page reference. Otherwise, it reaches the end of the first phase. Note that in the case that $lv_{max} < lv(v)+1$, if it is known that some entry in this subtree has the left position equal to $LP_{max}(LP = LP_{max})$, we can conclude that $D(v)$ has no child in the $XL+-tree$. This condition is satisfied if all delimiting keys stored on internal pages are keys of entries stored on leaf pages. In the following description, as in the traditional $B+$-tree context, we assume such guarantee. If a leaf page is reached, entries stored on it should have a LP range $(LP_{min}, LP_{max}]$ satisfying $LP_{min} \leq LP(v) < LP_{max}$, and a level range $[lv_{min}, lv_{max}]$ containing $(lv(v)+1)$. However, it does not guarantee that the first(leftmost) entry with $LP > LP(v)$ and $lv \leq lv(v)+1$, which according to Lemma 1, either is $D(v)$'s first child or indicates that there is no $D(v)$'s child, is on this leaf page. Therefore, the procedure continues to identify the leftmost entry, $D(w)$, with $LP > LP(v)$ and sequentially scan entries after $D(w)$ on this leaf page. If an entry with $lv \leq lv(v)+1$ is found, either it is $D(v)$'s first child or we can conclude that $D(v)$ has no child in the $XL+$-tree. Otherwise, if all entries after $D(w)$(including $D(w)$) have level of $lv > lv(v)+1$ and the last entry of this leaf page has the left position of $LP < RP(v)$, we invoke the second phase of the procedure.

The second phase involves probably backtracking on the top-down search of the first phase and then another top-down search. If the first phase ends at an internal page, the second phase continues to sequentially consider page references after the current page reference. There are six possible cases:

1. The page reference's maximal level, $lv_{max} < (lv(v)+1)$; in this case, it can be concluded that $D(v)$ has no child entry since according to Lemma 1, all entries before $D(v)$'s first child but with $LP > LP(v)$ should have level larger than $(lv(v)+1)$;

2. The page reference's minimal level, $lv_{min} > lv(v)+1$ but $LP_{max} \geq RP(v)$; in

this case, it can be concluded that no $D(v)$'s child entry exist in the XL+-tree;

3. $lv_{min} > lv(v) + 1$ and $LP_{max} < RP(v)$; in this case, it can be concluded that no $D(v)$'s child is in this subtree. It continues to consider the next page reference;

4. $lv_{min} \leq (lv(v) + 1) \leq lv_{max}$ but $LP_{min} \geq RP(v)$; in this case, again it can be concluded that $D(v)$ has no child entry since any $D(v)$'s child entry should satisfy $LP < RP(v)$;

5. $lv_{min} \leq (lv(v) + 1) \leq lv_{max}$ and $LP_{min} < RP(v)$; in this case, $D(v)$'s first child is *probably* in this subtree; it indicates the end of backtracking and the beginning of the second top-down search;

6. The end of this page is reached; it backtracks to the current page's parent page; if the current page is the root page of XL+-tree, it can be concluded that no $D(v)$'s child exists.

If the first phase ends at a leaf page, the second phase firstly backtracks to the leaf page's parent page. It then continues to consider other page references after the current page reference on the internal page. Possible cases are the same as the six outlined above.

In the second phase, another top-down search is required only when Case 5 occurs. We also have the observation that in the second top-down search, all entries in the subtree, which corresponds to the encountered page reference, satisfy $LP > LP(v)$. Therefore, the procedure always sequentially scans page references or entries on the current page beginning with the leftmost one. The operations upon page references on the internal page are similar to the six cases just described. Cases 1, 2 and 4 indicate that $D(v)$'s first child doesn't exist. If case 5 is encountered, it advances to the next page following the current page reference. Case 6

never occurs. The search operation on the leaf page is also the same as in the first phase except that it again scans from the leftmost entry and there should be some entry with level of $lv \leq lv(v) + 1$.

The procedure for identifying $D(v)$'s first child is described in *Algorithm 3.1.* It is now obvious that the number of page accesses it invokes in the worst case is $\mathbf{O}(log_B k)$(for the first top-down search)+$\mathbf{O}(log_B k)$(for the backtracking)+$\mathbf{O}(log_B k)$(for the second top-down search)=$\mathbf{O}(log_B k)$. By simply following the *following-sibling* pointers, we achieve the claimed $\mathbf{O}(log_B k+rd)$ worst case I/O cost for the $Search\_Children(D(v))$ operation, in which rd is the number of pages where $D(v)$'s children are stored. Note that this result *asymptotically* improves the result of the straightforward solution that takes $\mathbf{O}(log_B k + \frac{rs}{B})$ I/Os in the worst case, in which rs is the number of $D(v)$'s descendants, since $\lceil \frac{rs}{B} \rceil \geq rd$. A working example of this procedure is also provided in Figure 3.4. Note that instead of scanning B_d pages to find $D(v)$'s first child as of the straightforward solution, our proposed algorithm takes only one backtracking step and one additional top-down search, totally two additional pages, to achieve the purpose.

Algorithm 3.1: Identify $D(v)$'s first child, $D(v) =< LP(v) : PR(v), lv(v), P\_LP(v) >$

1. the first top-down search;
   - end-of-first-phase=false;
   - beginning with the root page of XL+-tree, do {
     - If (the current page is an internal page)
       (a) Identify the leftmost page reference, PR_i, with $LP(v) \leq LP_{max}$;
       (b) If (PR_i's range level $[lv_{min}, lv_{max}]$ covers $(lv(v)+1)$)
           * advance to next page following PR_i;
       (c) Else
           * end-of-first-phase=true;
     - Elseif (the current page is a leaf page)
       (a) identify the leftmost entry, $D(w)$, with $lv(w) > LP(v)$;
       (b) sequentially scan entries after $D(w)$
           * if (the current entry $D(u)$'s LP satisfies $LP(u) < RP(v)$)
             · If $(lv(u) = lv(v)+1)$
               terminate this algorithm; $D(u)$ is $D(v)$'s first child;
           * Elseif $(LP(u) > RP(v))$
             · terminate this algorithm; No $D(v)$'s child exists;
           * end-of-first-phase=true;
   - } until (end-of-phase==true)
2. backtracking;
   - if (the first phase ends at an internal page)
     - sequentially consider other page references after PR_i on the current page
       (a) If $(lv_{max} < (lv(v)+1))$
           terminate this algorithm; no $D(v)$'s child exists;
       (b) Elseif $(lv_{min} > lv(v)+1$ & $LP_{max} \geq RP(v))$
           terminate this algorithm; no $D(v)$'s child exists;
       (c) Elseif $(lv_{min} > lv(v)+1$ & $LP_{max} \leq RP(v))$
           continue to consider next page reference;
       (d) Elseif $(LP_{min} \geq RP(v))$
           terminate this algorithm; no $D(v)$'s child exists;
       (e) Elseif $(LP_{min} < RP(v))$
           it indicates the end of backtracking;
       (f) Elseif (the end of page is reached)
           backtrack to the current page's parent page; if the current page is the root page of XL+-tree, terminate this algorithm; no $D(v)$'s child exists;
   - elseif (the first phase ends at a leaf page)
     - it firstly backtracks to the leaf page's parent page and then continues to consider other page references after the current page reference on the internal page. Possible cases are the same as the six outlined above.
3. the second top-down search;
   the second top-down search sequentially scans page references or entries from the leftmost one on the current page. The operations upon page references on the internal page are similar to the six cases just described except that case 6 should never occur. Cases 1, 2 and 4 indicate that $D(v)$'s first child doesn't exist. If case 5 is encountered, it advances to the next page following the current page reference. The search operation on the leaf page is also the same as in the first phase except that it scans from the leftmost entry and there should be some entry with $lv \leq lv(v)+1$.

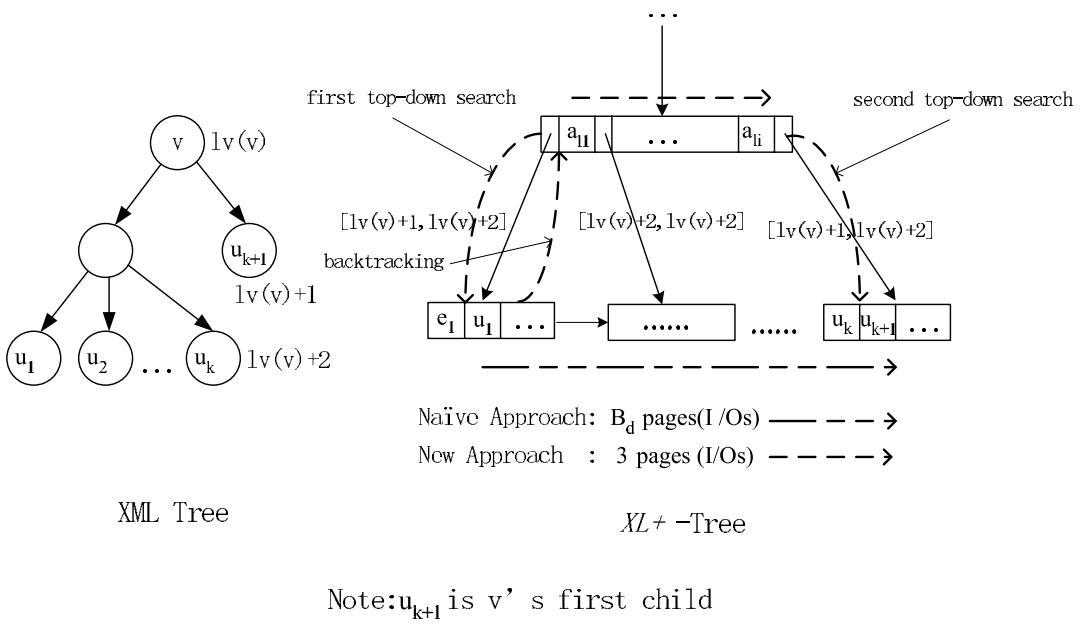


Figure 3.4: A working instance of searching $D(v)$'s first child

Horizontal Search: Preceding and Following Sibling

To search $D(v)$'s preceding(or following) siblings, we have the observation that once $D(v)$'s *first* preceding(or following) sibling is identified, its other preceding(or following) siblings can be tracked through entries' *preceding-sibling*(or *following-sibling*) pointers. we have the following Lemma which is similar to Lemma 1.

Lemma 3 *$D(v)$'s first following sibling $D(u)$, if it exists, is the leftmost entry satisfying $LP > RP(v)$ and $lv = lv(v)$ in the XL+-Tree; and all entries with $LP > RP(v)$ but before $D(u)$ have level of $lv > lv(v)$. Similarly, $D(v)$'s first preceding sibling $D(w)$, if it exists, is the rightmost entry satisfying $LP > LP(v)$ and $lv = lv(v)$ in the XL+-Tree; and all entries with $LP > LP(v)$ but after $D(w)$ have level of $lv > lv(v)$.*

Proof: Consider the leftmost entry $D(u)$ with $LP > RP(v)$ and $lv \leq lv(v)$. If $D(u)$ is $D(v)$'s following sibling, the lemma is true. Otherwise, assuming that v's parent is v_p, since $lv(u) \leq lv(v)$, u is NOT v_p's descendant; from the fact that u

is after v, we have $LP(u) > RP(v_p)$. Therefore, none of entries after $D(u)$ can be v_p's child or v's following sibling.

Similarly, we consider the rightmost entry $D(w)$ with $LP < LP(v)$ and $lv \leq lv(v)$. If $D(w)$ is $D(v)$'s preceding sibling, the lemma is true. Otherwise, assuming that v's parent is v_p, since $lv(w) \leq lv(v)$, w is NOT v_p's descendant; from the fact that w is before v, we have $LP(w) > LP(v_p)$. Therefore, none of entries before $D(w)$ can be v_p's child or v's preceding sibling. □

The strategy of efficiently searching $D(v)$'s first preceding or following siblings in $XL+-tree$ is similar to the operation of searching $D(v)$'s first child. It also involves two phases: the first phase of a top-down search, and if necessary, the second phase of backtracking and another top-down search.

Consider the procedure for identifying $D(v)$'s first following sibling. The first top-down search find the *leftmost* page reference with $LP_{max} > RP(v)$, PR_i, on each internal page. If PR_i's level range contains $lv(v)$, the search advances to the page of next level. Otherwise, it invokes the second phase and sequentially scans other page references after PR_i. There are totally four possible cases:

1. $lv_{max} < lv(v)$. It can be concluded that $D(v)$ has no following sibling;
2. $lv_{min} > lv(v)$. $D(v)$'s first following sibling can not be in this subtree, continue to next page reference;
3. $lv_{min} \leq lv(v) \leq lv_{max}$. $D(v)$'s first following sibling is probably in this subtree; this case indicates the end of backtracking.
4. The end of page is reached. It backtracks to the current page's parent page;if the current page is the root page of $XL+$-tree, no $D(v)$'s following sibling exists;

As in the procedure of identifying the first child, after a leaf page is reached, it determines whether the leftmost entry with $LP > RP(v)$ and $lv \leq lv(v)$ is in

this page. If yes, either $D(v)$'s first following sibling is found or it is concluded that no $D(v)$'s following sibling exists. Otherwise, it backtracks to the leaf page's parent page. The operations upon page references on internal pages are basically the same as described above. The second top-down search is also the same as the first one except that it always begins with the leftmost page reference(or entry) on each page and case 4 should never occur.

The procedure of identifying $D(v)$'s first preceding sibling should be straightforward since it is actually symmetric to the procedure of identifying $D(v)$'s first following sibling. The first top-down search find the *rightmost* page reference with $LP_{min} < LP(v)$, PR_i, on each internal page. If PR_i's level range contains $lv(v)$, the search advances to the page of next level. Otherwise, it invokes the second phase and sequentially scans other page references *before* PR_i in the *backward* manner. Four possible cases are as follows:

1. $lv_{max} < lv(v)$. It can be concluded that $D(v)$ has no preceding sibling;
2. $lv_{min} > lv(v)$. $D(v)$'s first preceding sibling can not be in this subtree, continue to previous page reference;
3. $lv_{min} \leq lv(v) \leq lv_{max}$. $D(v)$'s first preceding sibling is probably in this subtree; this case indicates the end of backtracking.
4. The end of page is reached. It backtracks to the current page's parent page;if the current page is the root page of $XL+$-tree, no $D(v)$'s preceding sibling exists;

The backtracking and second top-down search can also be accomplished in the similar way. We do not describe further details since they are obvious.

From the above descriptions, procedures for identifying $D(v)$'s first preceding or following sibling take $\mathbf{O}(log_B k)$ I/Os in the worst case. Therefore, both

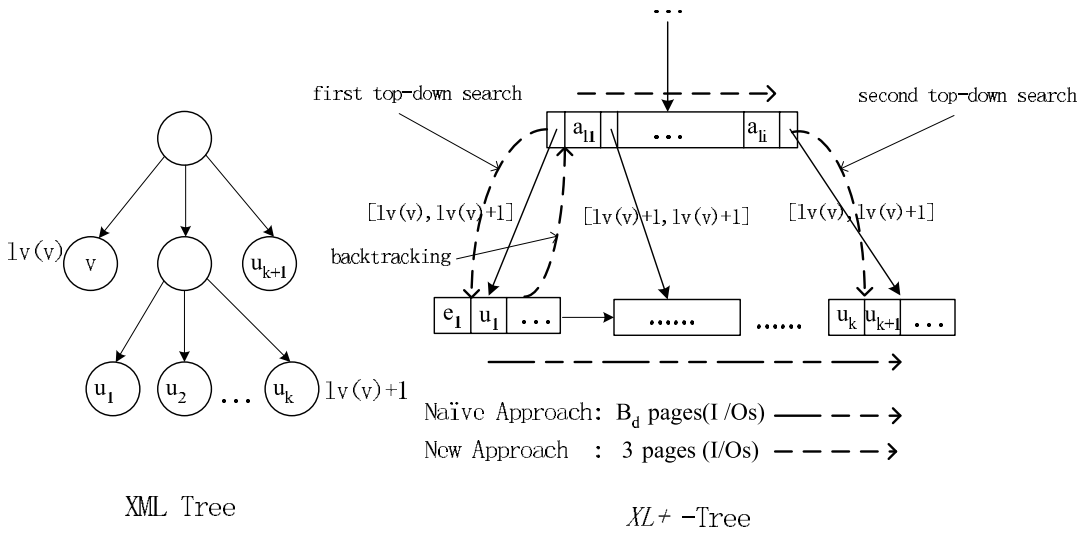


Figure 3.5: A working instance of searching $D(v)$'s first following sibling

Search_Preceding-Sibling($D(v)$) and *Search_Following-Sibling*($D(v)$) operations can be accomplished consuming only $\mathbf{O}(log_B k+rd)$ I/O cost in the worst case, in which rd is the number of pages storing $D(v)$'s preceding or following siblings. A working example of the procedure for identifying $D(v)$ first following sibling is presented in Figure 3.5. It takes $h = \mathbf{O}(log_B k)$ I/Os, in which h is the height of $XL+-tree$. Note that a naive solution, which searches the leftmost entry with $LP > RP(v)$ and then scan sequentially to find $D(v)$'s first following sibling, takes $(h+B_d)$ I/Os in this instance.

Bottom-Up Search:Ancestor and Parent

Concerning the *Search_Parent*($D(v)$) and *Search_Ancestors*($D(v)$) operations, we have the following lemma:

Lemma 4 *$D(v)$'s ancestor at level $lv_a \leq (lv(v)-1)$, if it exists, is the rightmost entry at level lv_a and with $LP < LP(v)$ in the XL+-tree; and all entries after it but before $D(v)$ have level of $lv > lv_a$.*

Proof: Consider the *rightmost* entry, $D(u)$, with $LP(u) < LP(v)$ and $lv(u) \leq lv_a$ in the XL+-Tree. If it is $D(v)$'s ancestor of level lv_a, the lemma is true; otherwise, there are two possible cases:

1. $D(u)$ is $D(v)$'s ancestor, but has level $lv(u) < lv_a$; since $D(v)$'s ancestor of level lv_a should be $D(u)$'s descendant, obviously no entry before $D(u)$ can be $D(v)$'s ancestor of level lv_a;

2. $D(u)$ is not $D(v)$'s ancestor; in this case, ranges $[LP(u), RP(u)]$ and $[LP(v), RP(v)]$ do not overlap. Note that any entry before $D(u)$ should have a range $[LP, RP]$ which either contains $[LP(u), RP(u)]$ or does not overlaps with $[LP(u), RP(u)]$. If its range does contain $[LP(u), RP(u)]$, its level satisfies $lv < lv(u) \leq lv_a$; therefore, it can not be $D(v)$'s ancestor of level lv_a. If its range does not overlap with $[LP(u), RP(u)]$, it neither overlaps with $[LP(v), RP(v)]$; thus it can not be $D(v)$'s ancestor.

Therefore, we have the conclusion that if $D(u)$ is not $D(v)$'s ancestor of level lv_a, no ancestor of level lv_a exists in the XL+-tree. □

Obviously, the *Search_Parent(D(v))* operation amounts to the key(equal to $P\_LP(v)$) search operation on the XL+-tree.

To facilitate the *Search_Ancestor(*$D(v)$*)* operation, we record all distinct levels of entries indexed by an XL+-tree. As claimed in section 3.2, XML trees' maximal depth can be expected to be small; thus number of distinct levels in an XL+-tree is also small. The overall idea of conducting the *Search_Ancestor*$(D(v))$ operation is similar to that of other search operations. Intuitively, it repeatedly searches, in the decreasing order of lv_a, the *rightmost* entry of level $lv_a (lv_a < lv(v))$ before $D(v)$. It involves multiple repetitions of the top-down search followed by the backtracking.

Its first top-down search recursively identifies the *rightmost* page reference satisfying $LP_{min} < LP(v)$ on internal pages. If it reaches a leaf page, all entries with

$LP < LP(v)$ or before $D(v)$ are sequentially scanned in the *backward* manner; if the minimal level of these entries is $lv_m < lv(v)$, according to Lemma 3, we have the conclusion that $D(v)$'s ancestors of level $lv_m \leq lv < lv(v)$ should be among them if they exist. The procedure continues to identify $D(v)$'s ancestors of level $lv < lv_m$. It backtracks to the current leaf page's parent page and scans page references sequentially in the *backward* manner before the current page reference. In the case that the first top-down search ends at some internal page because $lv_{min} \geq lv(v)$, it simply continues to consider previous page reference sequentially. If the encountered page reference's lv_{min} is less than lv_m, it stops the backtracking and begins the second top-down search. The second top-down search similarly identifies the rightmost page reference with $lv_{min} < lv_m$ on internal pages. Note that beginning with the second top-down search, all entries in the corresponding subtree have $LP < LP(v)$. Therefore, the top-down search should reach a leaf page and the minimal level(lv) of entries on this leaf page should be $lv_{min} < lv_m$. According to Lemma 3, we have the conclusion that all $D(v)$'s ancestors of level $lv \in [lv_{min}, lv_m)$ should be on this leaf page. Therefore, the procedure scans all entries on this leaf page in the *backward* manner. If it encounters an entry with level of $lv' < lv_m$, we have the conclusion that either this entry is $D(v)$'s ancestor of level lv' or $D(v)$ has no ancestor of level lv'. If an entry of level lv_{min} is encountered, the scanning process on this leaf page stops. The value of lv_m is now reset to be lv_{min} and another round of backtracking and top-down search begins. The procedure continues this process until the value of lv_m reaches the smallest level of entries indexed by the $XL+$-tree. The whole procedure of the *Search_Ancestor*($D(v)$) operation is described in **Algorithm 3.2**. Since each round of backtracking and top-down search reduces the value of lv_m by at least one, the maximal number of rounds required by the operation is $(lv(v) - 1)$. Therefore, the *Search_Ancestor*($D(v)$) operation takes $\mathbf{O}(lv(v) \times log_B k)$ I/O cost in the worst case. A working instance

of the *Search_Ancestor*($D(v)$) operation is also provided in Figure 3.6.

Algorithm 3.2: the *Search_Ancestor*($D(v)$) operation, $D(v) =< LP(v) : PR(v), lv(v), P\_LP(v) >$

1. lv_m=$lv(v)$; set lv_x to be the minimal level of entries in the XL+-tree;

2. end-of-first-search=false;

3. beginning with the root page of XL+-tree, do {

   - if (the current page is an internal page)
     (a) Identify the *rightmost* page reference, PR_i, with $LP_{min} < LP(v)$;
     (b) If (PR_i's level range $lv_{min} < lv_m$)
         – advance to next page following PR_i;
     (c) Else
         – end-of-first-search=true;
   - elseif (the current page is a leaf page)
     (a) identify the rightmost entry, $D(w)$, with $lv < LP(v)$;
     (b) sequentially scan entries before $D(w)$(including $D(w)$) on the current leaf page in *backward* manner;
         – if (the current entry $D(u)$'s $lv(u)$ satisfies $lv(u) < lv_m$)
           * output $D(u)$ if it is $D(v)$'s ancestor of level $lv(u)$;
           * lv_m=$lv(u)$;
         – elseif (the end of leaf page is reached)
           * end-of-first-search=true;

4. } until (end-of-first-search=true)

5. while ($lv_m > lv_x$)

   (a) backtracking;
   - if (the last top-down search ends at an internal page)
     – sequentially scan page references before the current PR_i in the backward manner;
       i. If ($lv_{min} < lv_m$)
          this case indicates the end of backtracking;
       ii. Elseif ($lv_{min} \geq lv_m$)
          continue to consider the previous page reference;
       iii. Elseif (the head of page is reached)
          backtrack to the current page's parent page;
   - elseif (the last top-down search ends at a leaf page)
     – backtrack to the current leaf page's parent page and sequentially scan page references before the current page reference; all possible cases are the same as presented above;

   (b) repeated top-down search;
   - The repeated top-down search operation always sequentially scan page references or entries on each page in the backward manner beginning with the rightmost one. Operations on the internal pages are the same as described in the backtracking part except that it begins with the rightmost page reference and case (iii) should never occur. Operations on the leaf page are the same as described in the first top-down search. Also note that lv_m's value will be reduced after each round of backtracking and top-down search.

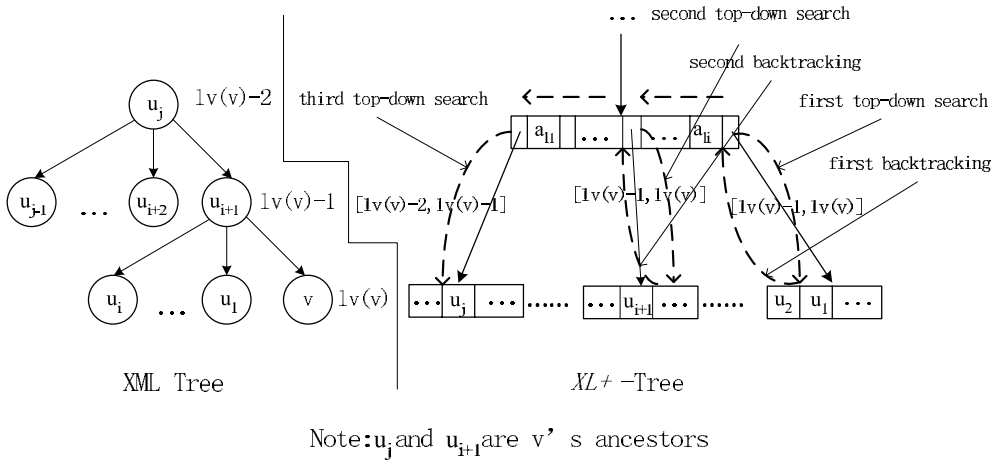


Figure 3.6: A working instance of searching $D(v)$'s ancestors

3.3.2 Update Operations on Range-Based $XL+$-tree

When the entry of a new element is inserted or deleted from the $XL+$-tree, the pointers of related entries and the level ranges stored over page references need be maintained effectively. It turns out that both deletion and insertion operations upon $XL+$-tree take the amortized I/O cost of $\mathbf{O}(log_B k)$. We first present the insertion operation and then the deletion operation.

As described in [31], we use *slot*s to store entries's positions on leaf pages. The advantage of implementing *slot*s is that when a new entry is inserted into a leaf page and positions of all entries after it in this page are shifted forward, we only need to update position values of shifted entries stored in slots; since the *preceding-sibling* or *following-sibling* pointers actually refer to slots, they do not need to be updated upon such shifting.

A new entry $D(w)$ can be inserted at the right position on $XL+$-tree just as on a typical $B+$-tree. $D(w)$'s immediate preceding and following siblings in the $XL+$-Tree can also be identified using presented search operations with the worst-case I/O cost of $\mathbf{O}(log_B k)$. To maintain the level ranges over page references, we check the level range over the page reference pointing to the leaf page where

$D(w)$ is inserted. If the level range $[lv_{min}, lv_{max}]$ contains $D(w)$'s level $lv(w)$, it remains unchanged and no other level range on the $XL+$-tree needs to be updated; otherwise, either lv_{min} or lv_{max} should be updated to accommodate $lv(w)$ and such update should be recursively propagated to the current page' parent page. Note that only level ranges of page references on the path from the root page to the target leaf page can be affected by the insertion operation. Therefore, in the worst case, the required I/O cost to maintain level ranges is $\mathbf{O}(log_B k)$. Finally, if an insertion operation results in the overcapacity of a leaf page, this leaf page needs to be split into two. For each moved entry, pointers to it should be updated properly. Since each entry is only referred by its immediate preceding sibling or following sibling entry, only constant I/Os are required to update pointers referring to each shifted entry. Additionally, only page references on the paths from the root page to two new sub-page can be affected by such splitting. Therefore, the amortized I/O cost of the insertion operation is $\mathbf{O}(log_B k)$.

Next we turn to the deletion operation. After an entry $D(w)$ is deleted from $XL+$-tree, its immediate preceding sibling entry's *following-sibling* pointer should be redirected to its immediate following sibling;and its immediate following sibling entry's *preceding-sibling* pointer should be redirected to its immediate preceding sibling. Maintaining the level ranges over page references is similar to what was described in the insertion operation. If the deleted entry $D(w)$'s level satisfies $lv_{min} < lv(w) < lv_{max}$, in which lv_{min} and lv_{max} are minimal and maximal levels recorded over the page reference pointing to the leaf page where $D(w)$ is stored before deletion, the level range $[lv_{min}, lv_{max}]$ over this page reference does not need to be changed; thus no other level ranges in $XL+$-tree needs to be updated. Otherwise, $lv(w)$ is equal to lv_{min} or lv_{max};in this case, we need to sequentially scan all remaining entries on this leaf page to determine if there is any one with level of $lv(w)$. If there is an entry of level $lv(w)$, the level range over this page

reference again does not need to be updated and it also indicates the end of the process to maintain level ranges. Otherwise, the target level range should be updated correspondingly and such update is propagated up one level. On each internal page, the level range over the page reference pointing to it is set to be $[lv_i, lv_a]$, where lv_i is the minimal of all lv_{min}s over page references initiating from this page and lv_a is the maximal of all lv_{max}s. This process is continued until the level range of the target page reference remains unchanged or the root page of $XL+$-tree is reached. It is not hard to see that the I/O cost of this procedure to maintain level ranges on $XL+$-tree is $\mathbf{O}(log_B k)$. If a deletion operation results in the undercapacity of a leaf page, it should be merged with another leaf page or some entries from another leaf page should be moved onto this leaf page. For each moved entry, it takes only constant I/Os to maintain pointers. Note that only page references over paths from the root page to the affected pages(at most two) need to be updated in the worst case. Therefore, the amortized I/O cost of the deletion operation on $XL+$-tree is $\mathbf{O}(log_B k)$.

We conclude this subsection with the following theorem, whose proof is straightforward from our above analysis.

Theorem 4 *The amortized I/O cost of the insertion and deletion operation on the $XL+$-tree are both $\mathbf{O}(log_B k)$.*

3.4 The $XL+$-Tree for Prefix Encoding Scheme

The $XL+$-tree based on the prefix encoding scheme has exactly the same structure as the one based on the range encoding scheme. The only difference is that entries on $XL+$-tree are represented by descriptors of format, $< S, lv, esl >$. Keys on the $XL+$-tree are label strings(S) instead of left positions(LP) and entries on leaf pages are sorted in the increasing lexicographic order of label strings. All the

analytical results of I/O cost concerning the search and update operations under the range encoding scheme also apply under the prefix encoding scheme.

In this section, we focus on the potential improvements of search operations on the XL+-tree as a result of the richer information provided by the prefix encoding scheme. Note that even though these improvements are not so significant to lead to an analytically improved *big O* results of I/O cost, they do reduce the I/O and CPU cost of search operations on XL+-tree. Our claim will also be verified by experimental results presented in the next section.

Under the prefix encoding scheme, the *Search_Ancestor*($D(v)$) operation can be accomplished by conducting multiple key searches since label strings of $D(v)$'s ancestors can be extracted from $D(v)$. Since the XL+-tree also stores the level range over each page reference, an additional requirement is enforced while advancing from one page to the next-level page: $[lv_{min}, lv_{max}]$ should contain the level(lv) of target entry; otherwise, it can be concluded that no such entry exists in the XL+-tree. The potential improvement of the new approach can be illustrated by the example in Figure 3.7. The previous approach requires to read the leaf page P_1 into main memory and then scan the entries before $D(v)$ on this page; next, it backtracks to P_1's parent page and reads the second leaf page P_2 into main memory; finally it scans all entries on P_2 in the backward manner to identify $D(v)$'s ancestor $D(u_1)$. In contrast, the new approach only requires to search the label string of $D(u_1)$ in the XL+-tree. Its first advantage is that, it don't need to read P_1 into main memory, but directly reads P_2, which has the result $D(u_1)$, into main memory. Secondly, searching on the P_2 page can be accomplished through the binary search, which is more CPU efficient than the linear scanning search.

The second potential improvement is on the *Search_Following-Sibling*($D(v)$) and *Search_Preceding-Sibling*($D(v)$) operation. Note that under the range encoding scheme, two nodes's non-overlapping intervals $[LP, RP]$ gives no clue about their

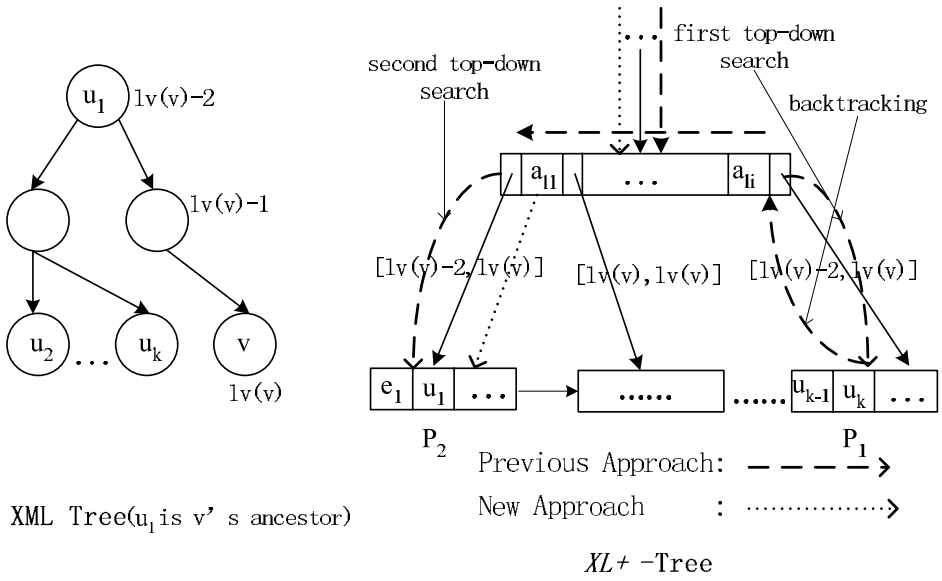


Figure 3.7: The new approach of searching $D(v)$'s ancestor under the prefix encoding scheme

sibling relationship, which can only be determined by checking their P_LPs. Under the prefix encoding scheme, label strings of $D(v)$'s following or preceding siblings should have the label string of $D(v)$'s parent as their prefix. We denote the label string of $D(v)$'s parent as $S(v_p)$. Any entry with the label string larger than $S(v_p)\bullet\omega$ thus can not be $D(v)$'s following sibling. This observation can be exploited to further prune the search space. Note that the similar strategy has been used under the range encoding scheme to prune search space while searching $D(v)$'s first child. Over there, any entry with $LP > RP(v)$ can not be $D(v)$'s first child. Therefore, while searching $D(v)$'s first following sibling, if we encounter some page reference with $S_{min} \geq S(v_p) \bullet \omega$, it can be concluded that no $D(v)$'s following sibling exists in the $XL+$-tree.

3.5 Experimental Results

In this section, we experimentally evaluate the performance of the XL+-tree on both the benchmark and synthetic XML data. The two datasets we use are:

1. Xmark Benchmark Data.

2. Synthetic XML Data. We use the IBM XML data generator to generate this synthetic data of size $20MB$ according to the DTD definition in Figure 3.8. Note that same-label nodes represent the same element definition. The asterisk(*) at the right-top of label nodes specifies the zero-or-more numerical relationship. This DTD is deliberately designed such that the resulting XML data has the following properties:(1) the first D-labeled child of an A-labeled data node a_i may not be right after the position of a_i on the XL+-tree T_D indexing D-labeled data nodes;(2)the first D-labeled following-sibling of an E-labeled data node e_i may not be right after the position of e_i in T_D;(3) the A-labeled ancestors of a H-labeled data node h_i may not be right before the position of h_i on the XL+-tree T_A.

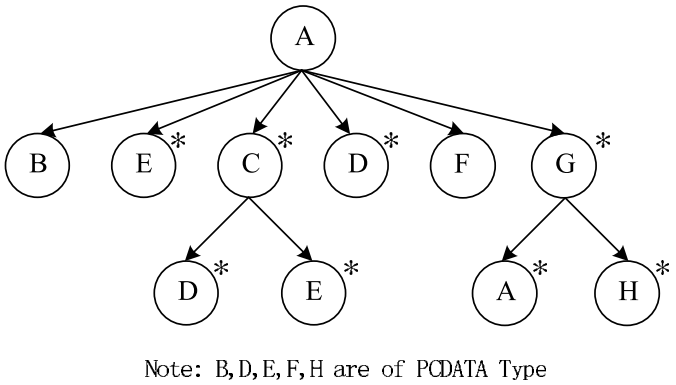


Figure 3.8: The DTD Definition of Synthetic Data

We compare the performance of the XL+-tree with that of the R-tree approach used in [60]. In [60], data nodes in an XML tree are represented as multidimensional

data points based on their *pre* and *post* positions. Specifically, we represent each data node v by $(pre(v), pos(v), par(v))$, in which $par(v)$ is v's parent node's *pre* position. Note that, since we do not differentiate attribute nodes from element nodes and the $XL+$-tree or R-tree indexes same-label nodes, we do not include the additional two dimensions used in [60], $att(v)$ and $tag(v)$, in our representation. We implement both structures on the TPIE platform(written in C++) [72], which is a software environment for external-memory algorithms. To fully explore the potential of R-tree approach, we also run the queries in the *batch* mode on R-trees. Instead of searching next location nodes from the current context nodes one by one, we bound a group of data points in a multidimensional box, which is then run on the R-tree. As a result, in the batch running mode, the query process involves one additional step: validating returned entries from R-tree. Depending on the type of locating axis, we also optimize the validation algorithm accordingly. We first sort the data points of current context nodes by some appropriate dimension in the increasing order and then validate the returned entries one by one. Suppose that we want to validate the returned entry u,

1. *child*: data points are sorted by *pre*; the validation is accomplished through the binary search of $par(u)$.

2. *parent*: data points are sorted by *par*; the validation is accomplished through the binary search of $pre(u)$.

3. $preceding - sibling$: data points are sorted by *par*; the validation is accomplished through the binary search of $par(u)$;

4. $following - sibling$: data points are sorted by *par*; the validation is accomplished through the binary search of $par(u)$;

5. *descendant*: data points are sorted by *pre*; the validation linearly scans data points until $pre > pre(u)$;

Top-Down Patterns	//A/*child*::D, //A/*descendant*::D
Bottom-Up Patterns	//D/*parent*::A, //H/*ancestor*::A
Horizontal Patterns	//B/*following-sibling*::D, //F/*preceding-sibling*::E

Table 3.1: Query Loads on Synthetic Data

6. *ancestor*: data points are sorted by *pre*; the validation identifies the first data point with $pre > pre(u)$ and then linearly scans the list until $pos > pos(v)$.

Our machine features the OS of Linux 2.4 and a Pentium 2.2 Ghz processor. Three query loads, which correspond to the top-down, bottom-up and horizontal navigations respectively, are tested on each dataset. The query loads for Xmark data are randomly generated from its DTD definition and each consists of 10 binary patterns. The query loads for the synthetic data are presented in Table 3.1.

3.5.1 XL+-Tree vs R-Tree

Since it is observed in our experiments that additional cache above $1MB$ has little effect on the overall performance of the $XL+-tree$ and R-tree on both datasets, all presented results of I/Os and running time are virtually independent of the size of available cache. Their comparative I/O performance and running time on both datasets are presented in Figure 3.9, 3.10, 3.11 and 3.12. In them, R-tree(k) represents the batch query mode on the R-tree with the size of batch set to be the total capacity of k pages. Since our experiments show that on the R-tree, the batch approach performs significantly better than the one-node-at-a-time approach, only results of the batch R-tree approach are presented. Note that the vertical axes of all figures follow a **logarithmic** scale, since there are marked differences in performance.

Our experiments show that compared with the range-based XL+-tree, the prefix-based XL+-tree has a worse performance in term of either the I/O cost or

the running time on all settings. Note that since the prefix-based XL+-tree actually shares the same structure as the range-based one, the observed performance differences result from the underlying encoding schemes instead of the index structure design. Between the range-based XL+-tree and the R-tree, it is clear that on either dataset, the XL+-tree performs considerably better than the R-tree approach in term of both I/O and running time. On both datasets, increasing the batch size of the R-Tree approach consistently results in the reduced I/O cost; however, the overall running time may increase as the validation may consume more CPU time. On the Xmark data, the running time of the *up* and *down* queryloads increase as the batch size is increased from 64-pages to 128-pages. On the synthetic data, the running time of the *up* and *down* queryloads also increase as the batch size is increased from 64-pages to 128-pages, the running time of the *horizontal* queryload increase as the batch size reaches 256-pages. Finally, we have the observation that the prefix XL+-tree also performs better than the R-tree approach in term of both the I/O cost and running time in most cases.

3.6 More Related Work

The range labeling scheme was first used to index XML tree nodes in [34]. We note that the later proposed durable numbering scheme [35, 51, 52], which is more friendly to update operations, is also range-based. Given two range labeled element sets, it has been shown that the containment structural join can be performed in the linear I/O and CPU cost [45]. Later on, with the help of the advanced B^+-tree [48, 49], its performance was improved to be sublinear because unrelated descendent or ancestors can be skipped. Similar approaches have also been successfully applied in the more complicated twig pattern XML queries [47, 50]. The B^+-tree based external memory index structures have also been proposed for XML under the

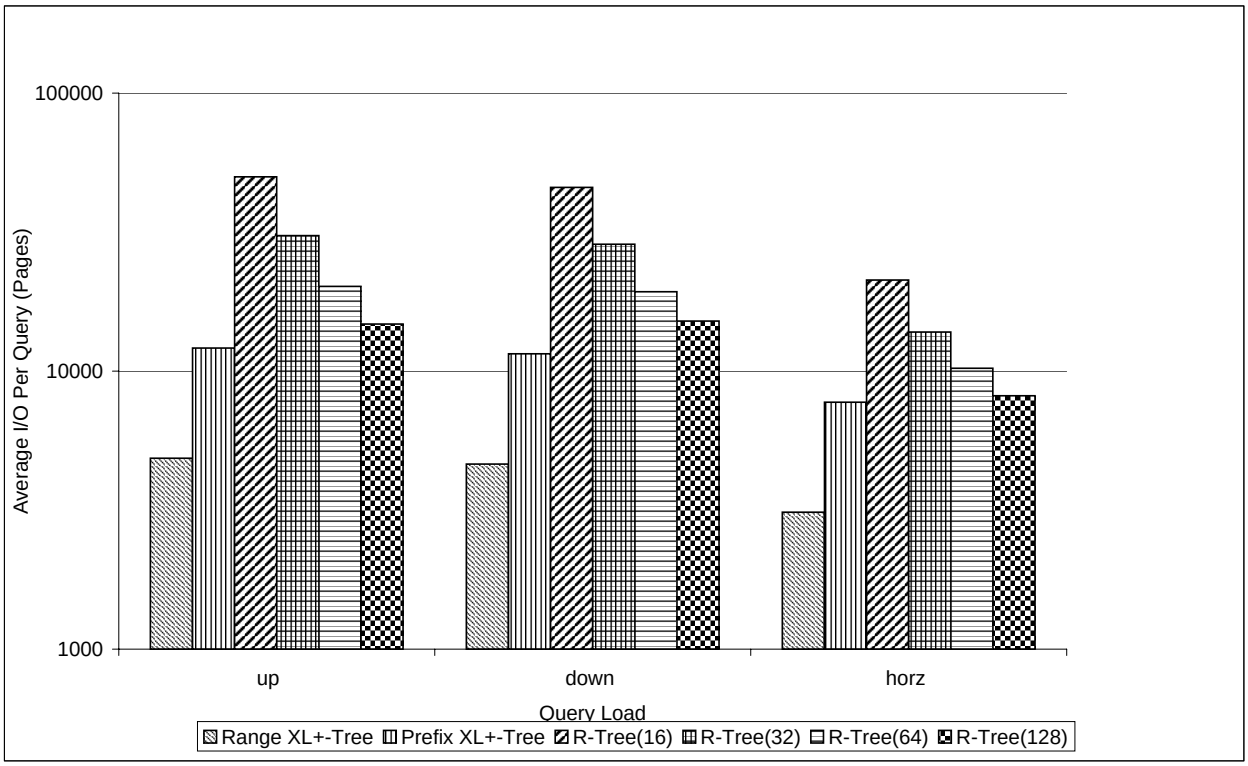


Figure 3.9: I/O Performance on Xmark Data

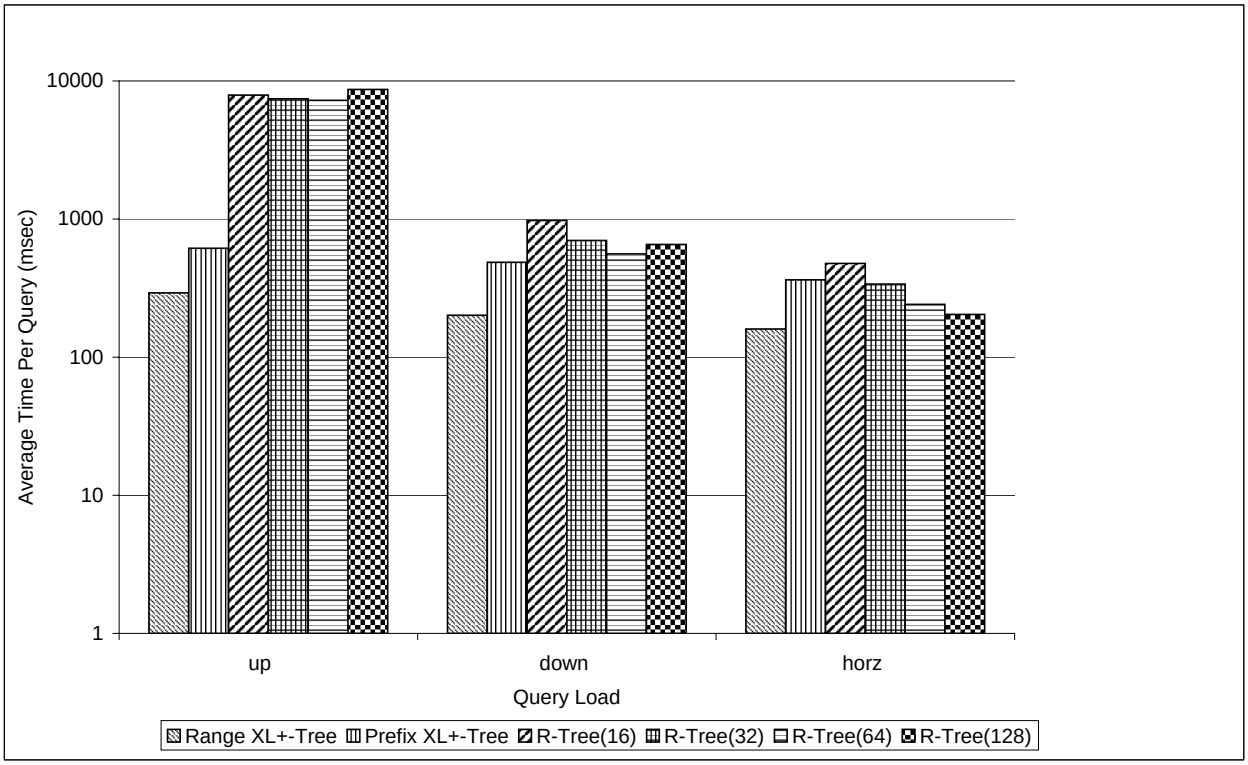


Figure 3.10: Combined I/O and CPU Performance on Xmark Data

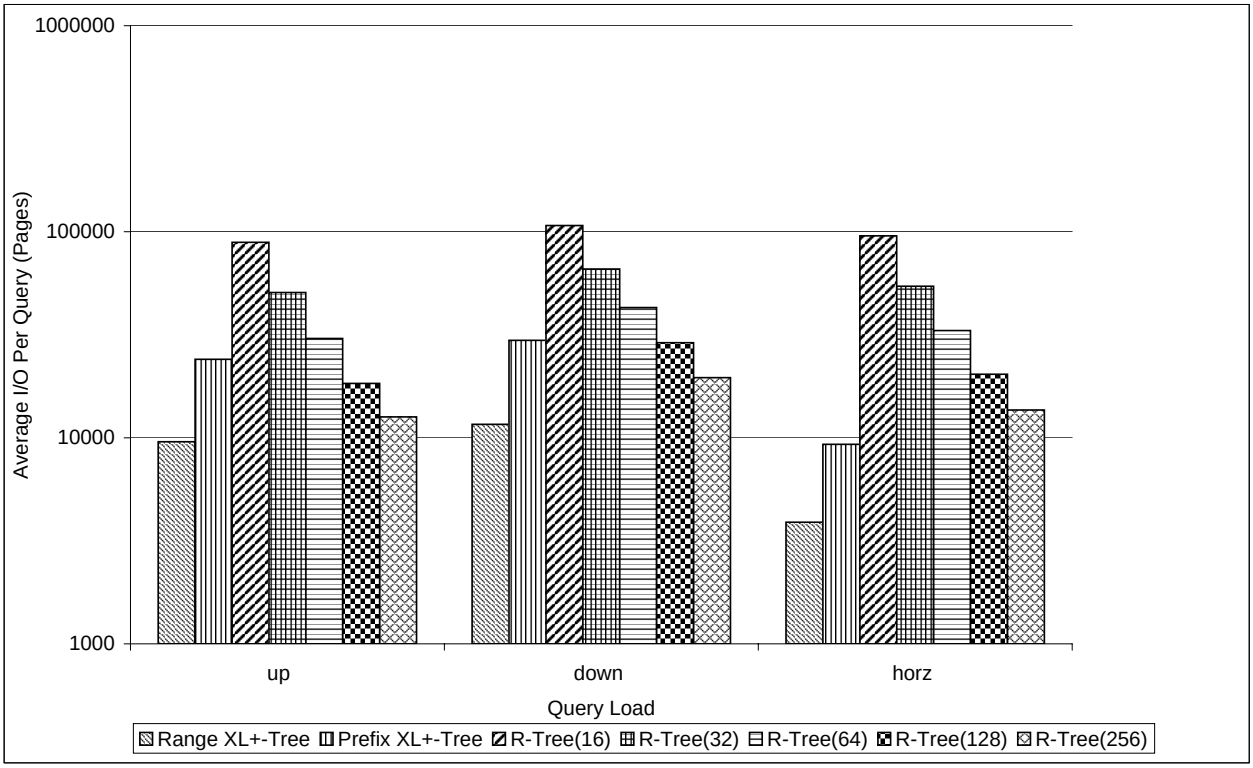


Figure 3.11: I/O Performance on Synthetic Data

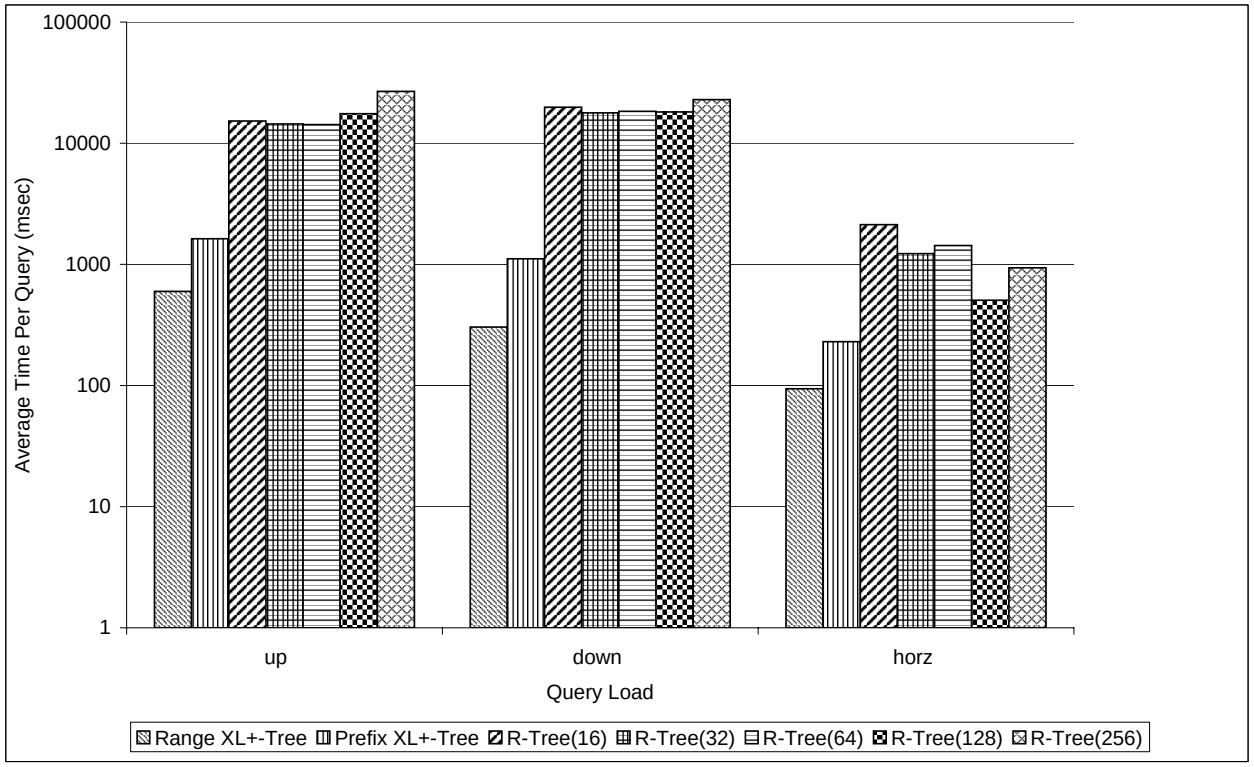


Figure 3.12: Combined I/O and CPU Performance on Synthetic Data

range labeling scheme [48, 49]. Unfortunately, only the containment relationship was considered.

We note that there are much less work on the prefix-based labeling scheme. The solution proposed in [36] encodes paths as strings and inserts them into a special index called Index Fabric. It is worth pointing out that the Index Fabric is actually a compact path summary of the XML tree. Its indexing technique is not the type of the prefix-based encoding we discussed in this paper.

Our idea of storing level ranges over page references on the XL+-tree was inspired by [30], where authors proposed a succinct XML physical storage for efficiently matching *next-of-kin*(NoK) patterns with only *parent/child* and *preceding-/following-sibling* relationships. Their technique represents an XML tree as a string on the external memory, and stores the minimal and maximal levels of nodes on each page.

Theoretical aspects of labeling the tree-structured data in the static or dynamic settings were studied in [46, 61, 62, 63, 67, 68]. Specifically, they considered how to encode nodes in the tree using the shortest labels such that we can decide the ancestor-descendant relationship between two nodes from their labels only.

The existing index structures to manipulate the external-memory strings, such as inverted files [64], B-tree [53, 55] and its variants (prefix B-trees [54] and string B-tree [56]), mainly target the prefix search and substring search problems. Especially, [56] assumed that strings are arbitrarily long and addressed the string search problems on B-tree where strings are represented by their logical pointers in external memory; their proposed technique can also be applied on the XL+-tree based on the prefix encoding scheme. There are also effective techniques to index strings in main memory with the aim of perform string matching, such as compacted tries [65], suffix trees [66, 69, 70] and suffix arrays [66, 71]. Note that these data structures did not consider new string search problems specific to Xpath

query processing.

3.7 Summary

In this chapter, we enhance the traditional range-based and prefix-based encoding schemes for XML documents and based on them, propose an external-memory index structure, the XL+-tree, which efficiently implements the comprehensive location steps specified in the Xpath query language. We analyze the I/O performance of both the search and update operations on XL+-tree. Finally, our experimental evaluation results on the benchmark and synthetic data validate the effectiveness of the XL+-tree proposal.

Chapter 4

SHiX: A *S*tructural *Hi*stogram for *X*ML Databases

Histograms are by far the most popular summary data structures used for approximating the result size of selection operations in relational databases [82, 83, 78, 80]. They usually divide the value range into several buckets based on collected statistical information, such as minimal value, maximal value and numbers of unique values. Assuming the uniform distribution within buckets, histograms were experimentally shown to be able to achieve high accuracy to support effective query optimization. Applied in the XML context, these techniques can actually be used to estimate the number of nodes satisfying a specified p redicate. However, the histogram for XML demands more if it could be useful. Most XML queries can be expected to combine content and structural searches. We need to find those nodes not only satisfying the specified predicate, but their position in documents matching some kind of path pattern. In this chapter, we introduce *SHiX*, a novel **S**tructural **Hi**stogram for the general graph-structured **X**ML databases. SHiX serves as a robust size estimator of XML twig patterns by exploring the numeric relationship information between node groups in the summary graph. *SHiX* also

possesses the adaptivity upon a typical update operation to XML databases, inserting new documents. Our extensive experiments on both real and benchmark XML data demonstrate its effectivity for approximating the result size of XML twig patterns.

4.1 Introduction

We note that proposals for XML result estimators have appeared in the literature. Even though most of previous work focus on simplified versions of the problem targeted by this chapter since they either assume the tree data model or only consider *non-branching* path expressions, the recently proposed *Xsketch* synopsis supports the branching path expression selectivity estimation on the general graph-structured XML data model; thus is closely related to ours. It is worthy to point out that, compared with *XSketch*, our proposed SHiX targets a slight richer class of path expressions, termed *twig pattern expression*, and is based on a different framework for estimating selectivity. SHiX has also an attractive property, being adaptive to a typical update operation that can cause a major change to XML databases: inserting new documents. As far as we know, how to adjust the *XSketch* synopsis for accommodating newly inserted XML documents without building it up from scratch remains unaddressed and seems not to be an easy task.

We propose *SHiX*, a novel **S**tructural **Hi**stogram for the general graph-structured **X**ML databases. We have the finding pointed out by [53] that the average depth of XML documents collected by a crawler over web is low; the parsed trees are balanced with relatively high degrees. Intuitively, our approach is motivated by the observation that the selectivity of a path expression can be estimated through the numeric relationship between neighboring labels in the path. As an example, in an XML document about some university's publications, there may be many books in

the directory of publication and each book may have several authors. Given that each *publication* averagely has six *book* elements and each *book* element averagely has three *author* elements and that there are totally two *publication* elements in the document, the selectivity of the path expression *publication/book/author* can be estimated as $2 \times 6 \times 3 = 36$. To handle branching predicates, besides the average numeric information on each directed edge, for instance $A \rightarrow B$, in *SHiX*, we also keep the forward-stability percentage information recording how many percent of nodes in group A have *at least* one child node in group B.

Our major contributions can be summarized as follows:

1. We propose a structural histogram, *SHiX*, for estimating the selectivity of path expressions with twig pattern on graph-structured XML databases. Compared with previous works, *SHiX* is based on a novel selectivity estimation framework. It records the average numeric relationship and forward-stability percentage information between summary nodes in the histogram graph.

2. With the problem of building the optimal *SHiX* given a limited memory trivially shown to be NP-hard, we present a greedy algorithm for efficiently building an effective *SHiX*. It consists of a sequence of refinement operations on the coarse *SHiX* structure. We also provide an effective algorithm for updating *SHiX* without building it up from scratch in the case that new documents are inserted into the XML database.

3. We conduct an extensive experimental study, using both real and benchmark XML data, to validate the effectivity of our new approach. Our comparative experiments demonstrate its superior performance over the previous *XSKETCH* proposal. Finally, we also verify experimentally that *SHiX* adapts well to the update operation of inserting new documents to XML databases.

4.2 Background

In this chapter, as in the second chapter, we assume the general graph model of XML documents. Reference links between elements are treated as normal edges. We use the XML data model of Figure 4.1 to illustrate SHiX throughout the whole chapter.

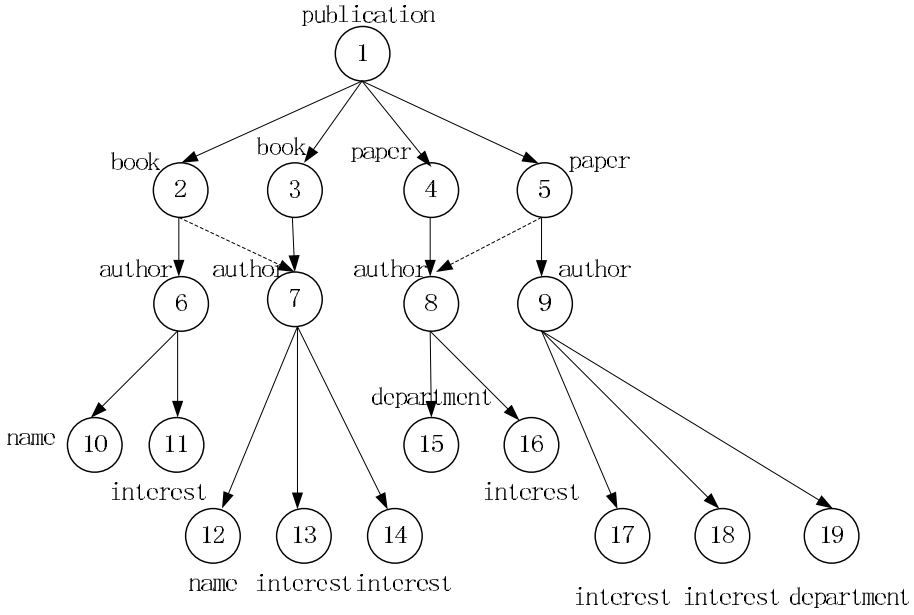


Figure 4.1: A Graph-Structured XML Data Model

The common feature among query languages proposed for XML is the use of path expressions for navigation in the XML document structure. In [22], a *simple path expression* is defined to be $L_1/L_2/\ldots/L_n$, where each L_i is a document label. The focus of *Xsketch* [22] is the *branching path expression* with the form $P = L_1[B_1]/L_2[B_2]/\ldots L_n[B_n]$, where each B_i is a simple path expression or ϵ. Our $SHiX$ proposal targets a slightly wider class of path expressions, termed *twig pattern expression*. Generally speaking, *twig pattern expression*s are those path expressions that can be presented as twig patterns with *parent-child* edges. It represents a more versatile class of query patterns than the *branching path expression* because it allows:(1) predicates themselves be a twig pattern; (2) each

label in the navigation path have more than one predicate. As an example, the XPath query Q_1 : $//paper[author[interest][department]]$ returns elements *paper* that have *author* as his child, which in turn has both *interest* and *department* as its child. If it is run on data in Figure 4.1, nodes {4, 5} will be returned. The query pattern Q_1 can be represented as a *twig pattern expression*, but not *branching path expression*.

Formally, the *twig pattern expression*s are the path expressions with the form of $L_1[B_{11}]\cdots[B_{1b_1}]/L_2[B_{21}]\cdots[B_{2b_2}]/\ldots L_n[B_{n1}]\cdots[B_{nb_n}]$, in which L_i is the document label and has b_i predicates, each being a twig pattern. Given a *twig pattern expression* TP, its selectivity is defined to be the total number of distinct matches on a graph data G. Each match of TP is a node path $u_1u_2\ldots u_n$ on G satisfying that each data node u_i has the label L_i and from u_i there exists b_i node paths on G matching patterns $B_{i1}B_{i2}\ldots B_{ib_i}$ respectively. Note that in XSketch paper, a path expression P's selectivity on G is defined to be the number of nodes that can be reached through P. Even though these two definitions of the path expression selectivity converge on tree-structured data, the results on graph-structured data are probably different since two parent nodes may have the same child node. The number of distinct matches was also used as the selectivity criteria in previous work, such as [22].

4.3 SHiX Framework

In this section, we present the *SHiX* structural model and the estimation framework based on it.

4.3.1 SHiX Summary Model

The *SHiX* summary model shares some similarities with *Xsketch* in that it is a node-labeled directed graph G_H where each node corresponds to a subset of identically-labeled data nodes of the original graph data G and an edge (u', v') in G should be represented in G_H by an edge between the summary nodes whose extents contain them. But, instead of the backward-stability or forward-stability indicators, two types of relationship information between summary nodes are stored over each edge, $A \rightarrow B$, in *SHiX*: (1) the average number of child data nodes in B for each data node in A, which is exploited to estimate the selectivity of the pattern $label(A)/label(B)$ along the navigation path; (2)the percentage of data nodes in A that have at least one child data node in B, which is exploited to estimate the selectivity of the pattern $label(A)[label(B)]$ along the branching predicate.

Definition 10 (The SHiX Summary Model) *A SHiX structural histogram for an XML graph data $G = (V_G, E_G)$ is a node-labeled directed graph, $G_H = (V_H, E_H)$, where each node $v \in V_H$ corresponds to a set $extent(v) \subseteq V_G$ such that: (1) all elements in $extent(v)$ have the same label; (2) $\cup_{v \in V_H} extent(v) = V_G$ and $extent(u) \cap extent(v) = \emptyset$ for each $u, v \in V_H$; (3) $(u, v) \in E_H$ if and only if there exist $u' \in extent(u)$ and $v' \in extent(v)$ such that $(u', v') \in E_G$; (4) each node $v \in V_H$ stores a field $|extent(v)|$, which is the total number of data nodes in v; (5) for each edge $(u, v) \in E_H$, there store two fields $aver(u, v)$ and $pert(u, v)$, which records the average number of children in $extent(v)$ for each data node in $extent(u)$ and the percentage of data nodes in u that have at least one child in v.*

An example *SHiX* structure for the date model of Figure4.1 is provided in Figure 4.2. The numbers inside circles represents the $|extent|$s of summary nodes; the pair over edges represents $(aver, pert)$. It actually represents the *coarsest SHiX*, the label-split graph.

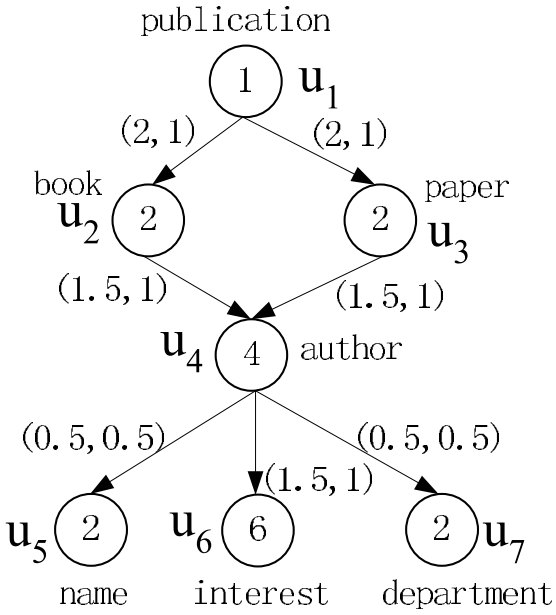


Figure 4.2: An Example SHiX Model

4.3.2 SHiX Estimation Framework

The following theorem explains the mechanism of *SHiX* serving as a size estimator.

Theorem 5 *Suppose that G_H is a SHiX structural histogram for an XML graph data G, and for each edge (u, v) in G_H, each data node in u has the same number of children in v and $pert(u, v) = 100\%$, then the estimation result of any twig pattern expression on G_H is exact.*

In practice, the *SHiX* with such uniform structure may not be very helpful because of its large size. Instead, we present the estimation framework to approximate the selectivity of *twig pattern expression*s on *SHiX* without the structural uniformity requirement. As previous proposals, our approach relies on several statistical independence and uniformity assumptions to compensate for the lack of detailed distribution information.

Estimation over Navigation Path

Consider the match of a simple path expression $P = L_1L_2 \dots L_n$ on the *SHiX* G_H, $M = u_1u_2 \dots u_n$. P's selectivity on the data graph G according to M can be *exactly* calculated as $|extent(u_1)| \times aver'(u_1, u_2) \dots \times aver'(u_{n-1}u_n)$, where $aver'(u_i, u_{i+1})$ is the average number of children in u_{i+1} for those data nodes in u_i that can be reached through some node path $u'_1u'_2 \dots u'_i$ in G, each u'_j being a data node in u_j for $1 \leq j \leq i$. For the estimation purpose, we replace $aver'(u_i, u_{i+1})$ with $aver(u_i, u_{i+1})$ stored over the edge (u_i, u_{i+1}) in G_H by exploiting the following two statistical assumptions.

Estimation Assumption 1 (`Frequency Uniformity`) EA 1: *Given any incoming path $u_1u_2 \dots u_i$ into u_i on G_H, for data nodes in u_i, their frequencies in distinct matches of $L_1L_2 \dots L_i$ on G, $u'_1u'_2 \dots u'_i$(each u'_j being a data node in u_j for $1 \leq j \leq i$), are uniformly distributed.*

Estimation Assumption 2 (`Path Independence`) EA 2*: The distribution of data nodes in u_i, concerning either the number of children or the forward-stability with respect to any child of u_i, is independent of any incoming path into u_i in G_H.*

The `Frequency Uniformity` assumption says that for each matching data node u'_i in u_i, an equal number of matches of $L_1L_2 \dots L_i$ end with it; the `Path Independence` assumption guarantees that for these data nodes in u_i, their average number of children in u_{i+1} is actually the same as $aver(u_i, u_{i+1})$, which is the average information for all nodes in u_i. Thus, the selectivity of P is estimated as $|extent(u_1)| \times aver(u_1, u_2) \dots aver(u_{n-1}u_n)$. In the case that one of these two assumptions fails, the estimation accuracy may be sacrificed as a result.

Take the example of estimating the selectivity of the pattern *book/author/name* on the *SHiX* presented in Figure 4.1. Its embedding is $u_2/u_4/u_5$. The data nodes in u_4 with incoming paths from u_2 actually have an average number of children in

u_5 of 1, $aver'(u_4, u_5)$=1; instead the estimation uses $aver(u_4, u_5)$=0.5; these two values are different because the `Path Independence` assumption is not satisfied. In another instance, if we estimate the selectivity of *book/author/interest*, its embedding on the *SHiX* is $u_2/u_4/u_6$. Even though $aver(u_4, u_6) = aver'(u_4, u_6) = 2$, the estimation result is *not* exact because data node $id = 7$ happens twice in matches of *book/author* and distribution of numbers of children in u_6 among nodes $id = 7$ and $id = 6$ is not uniform. It does not satisfy the `Frequency Uniformity` assumption. The estimation value is $2 \times 1.5 \times 1.5 = 4.5$; but the accurate selectivity is 5.

Estimation over Predicate Branch

Consider the path expression of $P = L_i[[L_{i1}][L_{i2}]]$, and its embedding on *SHiX* $M = u_i[[u_{i1}][u_{i2}]]$. The exact selectivity of P is S_P=$|extent(u_i)| \times prob(u_i, u_{i1}) \times prob(u_i[u_{i1}], u_{i2})$, where $prob(u_i, u_{i1})$ is the probability of data nodes in u_i having at least one child in u_{i1}, and $prob(u_i[u_{i1}], u_{i2})$ is the probability that the data nodes having at least one child in u_{ii} have at least one child in u_{i2}. Even though $prob(u_i, u_{i1})$=$pert(u_i, u_{i1})$ on *SHiX*, the value of $prob(u_i[u_{i1}], u_{i2})$ is unknown. By exploiting the following statistical assumption, we estimate the value of $prob(u_i[u_{i1}], u_{i2})$ with $pert(u_i, u_{i2})$.

Estimation Assumption 3 (`Branch Independence`) EA 3: *The distribution of data nodes in u_i, concerning either the number of children or the forward-stability with respect to any child of u_i, is independent of the existence of other outgoing paths from u_i in G_H.*

As long as the `Branch Independence` assumption is valid, $prob(u_i[u_{i1}], u_{i2})$=$prob(u_i, u_{i2})$=$pert(u_i$
But the estimation inaccuracy may occur if this assumption is not satisfied. As an instance, consider the pattern of $author[[name][department]]$ on the *SHiX* of Figure 4.2. Its mapping is $u_4[[u_5][u_7]]$; thus its selectivity is estimated as $4 \times 0.5 \times 0.5 = 1$

. But actually no *author* element has both the *name* and *department* element as its predicates in the graph data of Figure 4.1. The estimation on the G_H is not exact because the forward-stability percentage distribution of data nodes in u_4 over edge $u_4 \rightarrow u_7$ is not independent of the existence of the edge $u_4 \rightarrow u_5$ in G_H.

Note that in *twig pattern expression*s, each predicate branch B_i itself may be a twig pattern. For a pattern match $u_i[B_j]$ in G_H, where B_j is a simple path expression $u_j/u_{j+1}\ldots/u_k$, the probability of data nodes in u_i matching the branch B_j in G is estimated to be $pert(u_i, u_j) \times pert(u_j, u_{j+1}) \times \cdots pert(u_{k-1}, u_k)$ by assuming the `Path Independence` concerning the forward-stability percentage along the path B_j. Generally, if B_j is a twig pattern, $u_j[B_{j1}][B_{j2}]\ldots[B_{jk}]$, assuming both the `Path Independence` and `Branch Independence` assumptions, we recursively estimate the probability of data nodes in u_i matching B_j in G by $prob(u_i[B_j])$=$pert(u_i, u_j) \times prob(u_j[B_{j1}]) \times prob(u_j[B_{j2}]) \cdots prob(u_j[B_{jk}])$.

Summary: Estimation over Twig Pattern Expression

Summarizing the above analysis, we estimate the selectivity of a *twig pattern expression* embedding on *SHiX*,

$$u_1[B_{11}]\cdots[B_{1b_1}]/u_2[B_{21}]\cdots[B_{2b_2}]/\ldots u_n[B_{n1}]\cdots[B_{nb_n}]$$

as

$$|extent(u_1)| \times \prod_{k=1}^{b_1} prob(u_1[B_{1k}]) \times aver(u_1, u_2) \times \prod_{k=1}^{b_2} prob(u_2[B_{2k}]) \cdots \times$$
$$aver(u_{n-1}, u_n) \times \prod_{k=1}^{b_n} prob(u_n[B_{nk}])$$

by assuming **EA 1**, **2** and **3**. As an example, on the *SHiX* of Figure 4.2, the selectivity of $paper/author[department]/interest$, whose embedding is $u_3/u_4[u_7]/u_6$, is estimated as $extent(u_3) \times aver(u_3, u_4) \times pert(u_4, u_7) \times aver(u_4, u_6) = 2 \times 1.5 \times 0.5 \times 1.5 = 2.25$.

4.4 Constructing Effective SHiX

In this section, we describe *SHiX* construction algorithms. Note that given a graph data, its corresponding label-split graph represents the coarsest *SHiX* structure; therefore is used as the starting point of the construction process. The construction algorithm consists of a sequence of refinement operations on *SHiX*, with the purpose of minimizing the dependence on statistical assumptions, thus achieving higher estimation accuracy. The result of **Theorem 1** provides us with an unified approach for this, since it implies that the estimation accuracy depends on the extent of distribution uniformity over edges in the *SHiX* G_H, concerning the number of children and the forward-stability.

4.4.1 Optimal SHiX

Before proceeding to describe the refinement operation, we present a metric, *independent* of the workload, to measure the effectiveness of *SHiX* as a *twig pattern expression* size estimator. We use the *Sum Squared Error(SSE)* metric, proposed in [22] to evaluate the accuracy of a histogram in relational databases, to capture the skewedness of distributions of numbers of children and forward stability over edges on G_H. The backward SSE over an edge (u, v) is defined to be $SSE_b = \sum_{u' \in u}(f(u') - aver(u, v))^2$, where $f(u')$ is the number of children in v of the data node u'. The forward SSE is defined to be $SSE_f = \sum_{u' \in u}(st(u') - pert(u, v))^2$, where $st(u')$ indicates whether u' has at least one child in v, 1 for $true$ and 0 for $false$.

Definition 11 *Given a graph data G and a limited memory size, the optimal $SHiX$ is a $SHiX$ graph G_H satisfying that: (1)it takes the memory with size no more than available; (2) it has the minimal value of the* **S***HiX* **E***rror* **M***etric* $SEM = \sum_{(u,v) \in G_H}(SSE_b(u, v) + aver(u, v)^2 SSE_f(u, v))$.

Note that in Definition 2, we normalize the SSE_f with the factor $aver(u,v)^2$. Since the standard deviation $\delta_f = \sqrt{SSE_f}$, it amounts to normalizing the standard deviation of the forward stability with $aver(u,v)$. The intuition is that the forward stability percentage is a *relative* estimation parameter; thus the normalized δ_f more accurately reflects the relative importance of the backward and forward relationship on the edge (u,v).

The above definition of optimality of *SHiX* also allows a flexible way to take the query load into consideration. We can normalize $SSE_b(u,v)$ and $SSE_f(u,v)$ with weights $w_b(u,v)$ and $w_f(u,v)$ respectively; the value of $w_b(u,v)$ reflects the frequency of the backward binary pattern $label(u)/label(v)$ in the query load and the value of $w_f(u,v)$ reflects the frequency of the forward pattern $label(u)[label(v)]$. Thus, SEM is adjusted to be $\sum_{(u,v)\in G_H}(w_b(u,v) \times SSE_b(u,v) + w_f \times aver(u,v)^2 \times SSE_f(u,v))$.

It can be trivially shown that the problem of building the optimal *SHiX* is even harder than building the V-optimal multidimensional histogram in the relational database context. Consider a simple *SHiX* model consisting of only one parent and k children; and the parent node's forward-stability distribution is uniform according to any of its child nodes. It is obvious that the problem of building the optimal *SHiX* is equal to building the V-optimal k-dimensional histogram. Since the later problem has been shown to be NP-hard [84], constructing the optimal *SHiX* is also NP-hard.

4.4.2 A Greedy Approach

Since the intractability of building the optimal *SHiX*, we introduce a greedy approach to *efficiently* refine summary nodes in G_H with the target of reducing SEM, thus improving the estimation accuracy. It repeats the following two steps until the memory size limit is reached: (1) choose the summary node in G_H that is

in the most critical need of being refined; (2) partition the chosen node according its children.

The criticality of a summary node u in G_H is measured by SEM over all its outgoing edges. We define $SEM(u) = \sum_{(u,v)\in G_H}(SSE_b(u,v)+aver(u,v)^2 SSE_f(u,v))$. At each step, the node u with the maximal $SEM(u)$ is chosen as the refinement node.

The second step is closely related to the problem of building the V-optimal histogram for multidimensional data. The MHIST technique, proposed in [79] for building the multi-dimensional histogram without the attribute value independence assumption, was shown to be superior to other approaches in various experiments. MHIST works in two steps repeatedly: 1) from the m dimensions of data points, we choose one dimension whose distribution is the most in need of partitioning; 2) Next, data points are split along this dimension into a small number of buckets, t. By picking a dimension based on its criticality to the partition constraint at each step, MHIST-2($t = 2$) was shown to result in desirable results in most cases [79]. Our partitioning operation is a variant of the MHIST-2 technique. Instead of choosing the most critical edge to partition along at each step, the refinement operation considers all outgoing edges from u in G_H. On each such edge (u, v), data nodes in u are sorted by their numbers of children in v and are partitioned into two continuous sets in all possible ways. The one resulting in the minimal value of $SEM_{new} = SEM(u_1) + SEM(u_2)$, where u_1 and u_2 are two new summary nodes resulting from the partition of u, are accepted as the candidate partition along this edge. Finally, among all candidate partitions over edges, the one with the minimal SEM_{new} is adopted to refine the summary node u.

It is worthy to point out that usually the structures of documents in XML databases are not totally random; instead, they may conform to some DTD or schema definition. In the DTD specification, the number of same-label sub-element

nodes that an element node may have, can be defined as: *optional* (?), *one-or-more* (+), or *zero-or-more* (*). In the XML schema definition, it is represented by two parameters, *minOccurs* and *maxOccurs*, which mean the minimum and maximum respectively;the default values of both parameters are one. Analyzing both the real and benchmark XML DTDs or schemas [89], we expect that in most element definitions, the {?,+,*} operators or *minOccurs* and *maxOccurs* parameters are only present in a small subset of all defined sub-elements. Therefore, the structural irregularities, concerning the numbers of children and forward stability, are actually limited to some part of XML data. As a result, even though, as most of proposed heuristics for constructing the optimal multidimensional histogram, analytically the greedy construction algorithm has no quality guarantee for the resulting *SHiX*, empirically it is quite effective in constructing *SHiX* with good performance for XML data. This claim will be verified by our experimental study.

4.5 More Discussion on SHiX: Estimating and Updating

4.5.1 Estimation on SHiX

Given a *twig pattern expression* P, the sum of estimations of P's embeddings on the *SHiX* G_H with *distinct* navigation paths is the selectivity of P on the data graph G. Note that estimating the forward-stability percentage on a summary node with multiple embeddings of a predicate branch on G_H is not so straightforward. Consider an embedding $u[v_1][v_2](label(v_1) = label(v_2))$ on G_H of the pattern $label(u)[label(v)]$. Estimating the percentage of data nodes in u having at least one child with label $label(v)$ with $(pert(u, v_1) + pert(u, v_2))$ results in the *double counting* error: data nodes in u having children in both v_1 and v_2 are counted

twice. Instead, we present an approach estimating the percentage of data nodes in u having children in both v_1 and v_2, which is denoted as $pert_b$. As a result, the percentage of data nodes in u with children in either v_1 or v_2 can be estimated to be $(pert(u, v_1) + pert(u, v_2) - pert_b)$.

We denote the set of data nodes in v_1 and v_2 as S_v. A data node in u has averagely $av_u = aver(u, v_1) + aver(u, v_2)$ children in S_v. Assuming that the probabilities of these children being in v_1 or v_2 are proportional to the numbers of connectivities, (u, v_1) and (u, v_2), we estimate them to be $prob(u, v_1) = \frac{aver(u,v_1)}{av_u}$ and $prob(u, v_2) = \frac{aver(u,v_2)}{av_u}$ respectively. Therefore, for each data node in u, the probability of all its children being in v_1(or v_2) is $prob(u, v_1)^{av_u}$(or $prob(u, v_2)^{av_u}$). Thus, the percentage of data nodes in u with children in both v_1 and v_2, $pert_b$, is estimated to be $prob_b = (1 - prob(u, v_1)^{av_u} - prob(u, v_2)^{av_u})$. Since the estimation of $pert_b$ should not be less than 0, nor be larger than $pert(u, v_1)$ or $pert(u, v_2)$, $pert_b$ is normalized to be $min(pert(u, v_1), pert(u, v_2), max(prob_b, 0))$. Finally, if the number of embeddings are larger than two, the estimation of the percentage is calculated in a recursive way by considering two embeddings at each step. An instance of two embeddings is provided in Figure 4.3. Note that the estimation result of the new approach, 50%, is more accurate than the naive one which estimates the percentage as $(50\% + 50\% = 100\%)$.

It's interesting to mention that *SHiX* always estimates the selectivity of direct containment binary patterns (L_1/L_2) *exactly*, no matter how skewed the distributions of numbers of children or forward-stability of data nodes inside summary nodes are. This is because to estimate such binary pattern, *SHiX* does not depend on the validity of any statistical assumption. But be cautious that for the predicate binary pattern, $L_1[L_2]$, *SHiX* may incur estimation error because of the presence of multiple embeddings of L_2 on the *SHiX* structure.

Theoretically, *SHiX* can be also applied to estimate the selectivity of *binary*

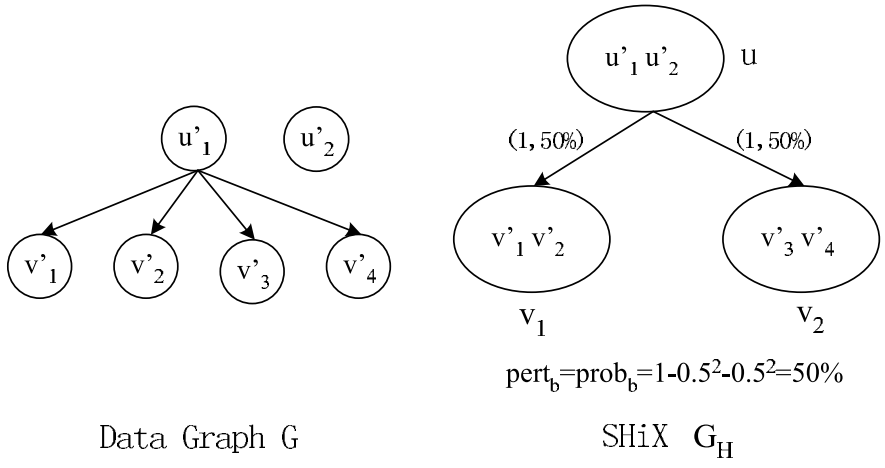


Figure 4.3: Computing $pert_b$ on Multiple Embedding of A Predicate

containment patterns, which have two labels connected through parent-child or ancestor-descendant axes. The problem of estimating the containment join size was first put forward by [85]. Authors assumed the tree structure of the underlying XML data model and only considered binary patterns. However, the effectiveness of *SHiX* to estimate the selectivity of path expressions with ancestor-descendant axes need to be further explored both analytically and empirically in the future research. This is beyond the scope of this dissertation.

4.5.2 Updating SHiX upon Insertion of New Documents

In this subsection, we present the updating operation for *SHiX* without building it up from scratch in the case that new documents are inserted into the XML database. Note that such updates may cause dramatic change to the selectivity of path expressions.

Our solution is to build a separate *SHiX* G_{H_2} for the newly inserted documents and then merge it with the existing *SHiX* G_{H_1}. Because of lack of detailed distribution information of data nodes inside summary nodes in *SHiX*, the updating

procedure assumes the uniform distribution among data nodes in the same summary node in G_{H_1} or G_{H_2}. It begins with a label-split *SHiX* G_H of combining G_{H_1} with G_{H_2}, and consists of a sequence of partitioning operations to refine summary nodes in G_H until the memory size limit is reached.

Beginning with the label-split *SHiX* G_H , the refinement operations partition summary nodes in G_H in a greedy way similar to the approach presented in Section 4.2. But there are some subtle differences between them. Firstly, data nodes in each summary node of the original G_{H_1} or G_{H_2} are considered as a *unit*, therefore would never be partitioned in the refinement process. Because of the uniform distribution assumption, data nodes in a unit have the same number of children and forward-stability with respect to any of its child unit. Secondly, if several units are grouped in the same summary node in the new G_H, the *aver* and *pert* information of this summary node with respect to its child or parent in G_H should be calculated properly. Suppose that two separate units v_1 and v_2 have the same label. Consider the summary node v consisting of v_1 and v_2, and the summary node w consisting of v_1 and v_2's common child unit w_1 on the new G_H. Obviously,

$$aver(v,w) = \frac{|extent(v_1)|\times aver(v_1,w_1)+|extent(v_2)|\times aver(v_2,w_1)}{|extent(v_1)|+|extent(v_2)|}$$

$$pert(v,w) = \frac{|extent(v_1)|\times pert(v_1,w_1)+|extent(v_2)|\times pert(v_2,w_1)}{|extent(v_1)|+|extent(v_2)|}$$

As for the relationship between v and the summary node u consisting of v_1 and v_2's common parent unit u_1, we have $aver(u,v)$=$aver(u_1,v_1) + aver(u_2,v_2)$. However, the exact value of $pert(u,v)$ is not available; therefore, it is estimated to be $(pert(u_1,v_1) + pert(u_1,v_2) - pert_b)$ as described in the last subsection. For the general case that summary nodes consist of multiple(maybe > 2) same-label units on G_H, their *aver* and *pert* information with respect to child or parent summary nodes can be straightforwardly calculated from the above two basic cases.

4.6 Experimental Study

In this section, we experimentally evaluate the performance of *SHiX* on both real and benchmark XML data. The datasets we use are:

1. DBLP Data. This is a very popular real XML dataset used in numerous experiments. It contains bibliographic data from the DBLP database. It features a relatively simple tree structure. The XML document is a $130MB$ file.

2. Xmark Data [41]. We generate Xmark documents of size $50MB$ through the provided data generator.

3. Bibliography HyperText(BHT) Data [90]. This is a real XML data describing the hypertext used in DBLP papers' bibliographies. Since there is no reference defined in its DTD file, its underlying data model is a tree. Compared with DBLP and Xmark data, it has a less regular structure;in its DTD definition, the zero-or-more(*) is specified over many sub-elements. We conduct experiments on a document with the size of $47MB$.

4.6.1 Quality Metric of Estimation

As in the *Xsketch* proposal [22], we measure the performance of *SHiX* by the *average absolute relative error* between the estimated and real selectivity over all path expressions in a workload, D. Specifically, the *average absolute relative error* is defined to be

$$AverError(D) = \frac{1}{|D|} \times \sum_{p_i \in D} \left(\frac{|count_{G_H}(p_i) - count_G(p_i)|}{count_G(p_i)}\right)$$

where $count_{G_H}(p_i)$ is the path expression p_i's estimated selectivity on the *SHiX* G_H and $count_G(p_i)$ is p_i's exact selectivity on the data graph G. Since the zero or low-count path expressions may contribute disproportionally high estimation

error percentage to $AverError(D)$, a "sanity" bound was also introduced in [22] to equate all zero or low-selectivity of path expressions with a default value s, which is usually set to be a small percentile of the exact selectivity distribution of the path expressions in D. Therefore, our estimation quality metric is defined as:

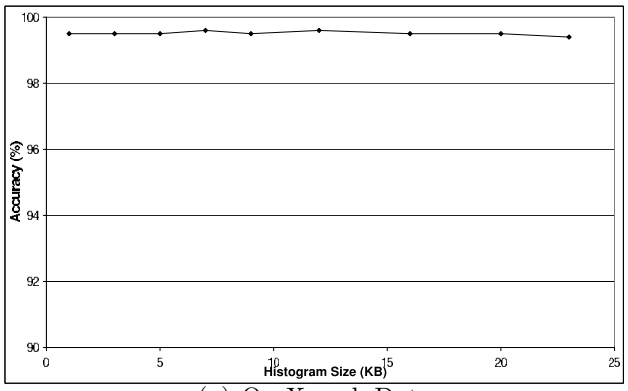
$Accuracy(D) = 1 - \frac{1}{|D|} \times \sum_{p_i \in D} (\frac{|count_{G_H}(p_i) - count_G(p_i)|}{max(count_G(p_i), s)})$

In our experiments, s is set to be the 10-percentile of the selectivity distribution; in other words, 90% of paths' selectivity are larger than s.

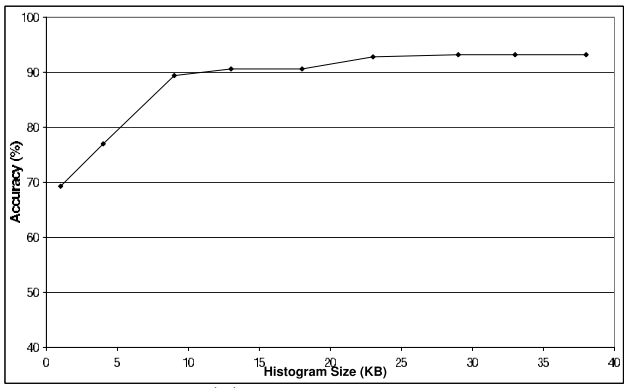
4.6.2 SHiX Estimation Performance

On each dataset, we randomly generate two workloads from the source graph data, one(D_s) consisting of only *s*imple path expressions and the other(D_t) consisting of *t*wig pattern expressions. Because of the simplicity of structures in the DBLP data, the generated D_t barely contains any path with predicates;therefore, we only evaluate the performance of the simple path query load on it. We set the range of simple paths' lengths to be $[1, 3]$; note that the binary path A/B are considered to be of length 1. In the workload of twig pattern expressions, lengths of navigation paths are also randomly between 1 and 3; each label along the navigation path may have $0 - 2$ predicate branches and their total size is maximally 3. On all datasets, we use the workload D_s with 300 test paths and D_t with 500. Our experiments show that if we varies the number of test paths between 100 and 1000 in either workload, the overall performance of *SHiX* only fluctuates slightly. Therefore, we believe that our chosen workloads effectively capture the overall structures present in tested XML data.

The *SHiX* G_H on all datasets are constructed using the unweighted version of SEM in our experiments. We track the histogram G_H's estimation performance as its size in memory increase gradually. Note that all label names are hashed and stored as 2-Byte integer numbers in G_H. We think that the 2-Byte integer type



(a) On Xmark Data



(b) On BHT Data

Figure 4.4: Performance of SHiX on Simple Path Expressions

should satisfy requirements in most XML databases since it can represent up to 65536 distinct values. As for the pair (*aver*,*pert*) over edges in *SHiX*, their values are represented by the 4-Byte integer type(int_{av}) and the 2-Byte integer type(int_{pe}) respectively. Specifically, $aver = \frac{int_{av}}{1000}$ and $pert = \frac{int_{pe}}{10000}$, where int_{av} and int_{pe} are the integer values stored over edges in G_H. Our representation of *aver*'s value is accurate by the measurement of $\frac{1}{1000}$ and allows the maximum of more than 4 millions. The representation of *pert*'s value is accurate by the measurement of $\frac{1}{10000}$.

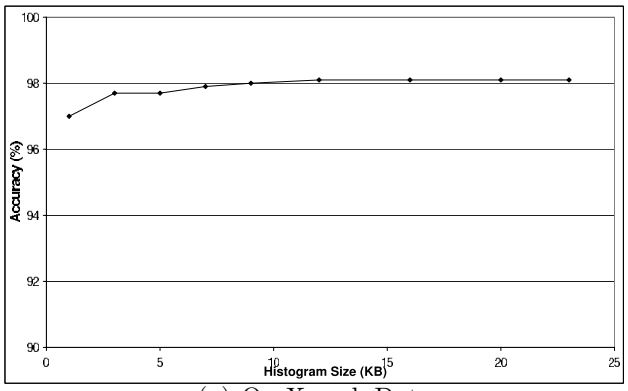
Performance on Simple Path Expressions

The estimation performance of *SHiX* on simple path expressions is shown in Figure 4.4. Note that the result on DBLP data is presented in Figure 4.6(a),

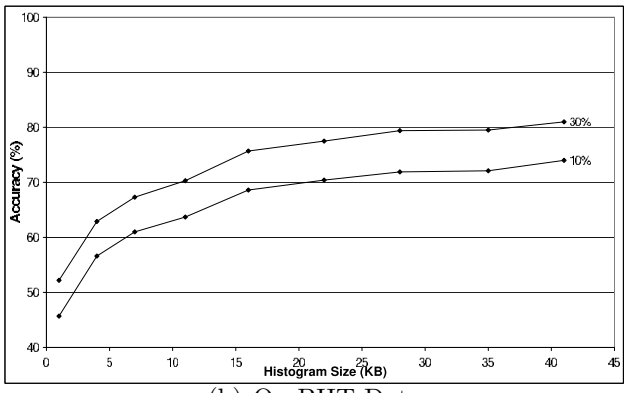
along with *Xsketch*'s result. On the DBLP and BHT data, the initial label-split G_H achieves only about 82% and 70% estimation accuracy respectively. Its performance improves steadily as the size grows. On the DBLP data, its accuracy approaches 90% with only $20KB$ size. On the BHT data, it similarly achieves the 93% accuracy at size $25KB$. This fact demonstrates that our proposed partitioning operation is quite effective in refining summary nodes for better estimation results. On the Xmark dataset, even the initial label-split G_H achieves the extremely high estimation accuracy, about 99%. This observation validates the effectiveness of our new approach of estimating path selectivity through the numerical relationship between summary nodes. We can expect that, in many XML databases, the numerical relationship between two types of elements is more or less regular. *SHiX* is shown to be quite effective in exploiting such regularity. The following refinement operations on Xmark data are shown to have little effect on the performance.

Performance on Twig Pattern Expressions

The performance of *SHiX* on twig pattern expressions is presented in Figure 4.5. Even though the overall performance of *SHiX* is not as good as that on simple path expressions, our results demonstrate that refinement operations on G_H steadily improve its estimation accuracy on both datasets. On the Xmark data, similar to the result on the simple path query load, even the label-split G_H achieves the 97% accuracy; further refinements improve its performance to 98% within $20KB$ size. On the BHT data, the original label-split G_H only achieves the 45% accuracy; its accuracy is improved to 75% at the size of $40KB$. Our experiments show that further refinements after that, up to $100KB$, do not result in considerable improvement. Noting that at 10-percentile, the sanity bound is only 23, a very small count. If we set the sanity bound to 30-percentile(367), which is still a reasonably small number, the accuracy percentage reaches 82%. Considering that



(a) On Xmark Data



(b) On BHT Data

Figure 4.5: Performance of SHiX on Twig Pattern Expressions

SHiX is estimating selectivity of the general twig pattern expressions on highly irregular data, we believe that this performance is reasonably good.

4.6.3 Comparison with *Xsketch*

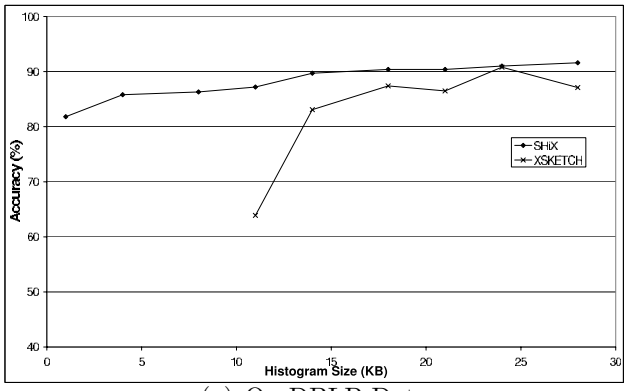
Since the *Xsketch* proposal for graph-structured XML databases was shown to be superior to other estimation techniques in performance and memory requirement, we compare our *SHiX* approach with the *Xsketch* synopses on the BHT and Xmark datasets in this subsection. Note that in the *Xsketch* proposal, (1) authors targeted the *branching path expression*s, where each label in the navigation path has at most one predicate branch and each branch should be a simple path; (2) the selectivity of a path expression p_i was defined to be the number of nodes that

can be reached through the node path matching p_i in G. Their defined selectivity actually converges with ours on tree-structured data.

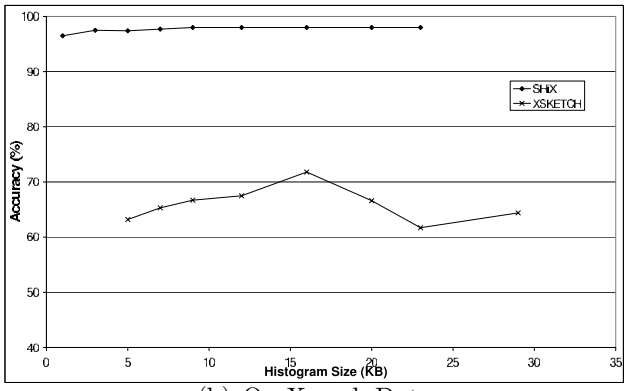
We randomly generate workloads consisting of 300 *branching path expression*s on the DBLP, Xmark and BHT datasets;the range of navigation paths' lengths is set to be $[1, 3]$ and the range of branching paths' length is set to be $[1, 3]$. As noted before, the query load on DBLP actually consists of only simple path expressions. We use the *Xsketch* binary code from its original authors to run the experiments. A path sample of size 200 is used to construct the full bisimulation(forward and backward) *Xsketch* on all datasets. The results are presented in Figure 4.6. Our experiments show that the performance of both *SHiX* and *Xsketch* flatten out after the $30KB$ size. We have the observation that *SHiX* clearly outperforms *Xsketch* on all three datasets. Although *Xsketch* also achieves the relative high estimation accuracy(90%) after the first iterations of refinements on DBLP data, further refinements do not yield higher performance. On Xmark data, its performance stabilizes at about 72%; on the BHT, it is at about 65%. As for *SHiX*, it achieves the high estimation accuracy of 91% and 98% on the DBLP and Xmark data respectively. On the BHT data, similar to the results presented on *twig pattern expressions*, refinements on *SHiX* steadily result in higher estimation accuracy, up to 72% within $25KB$ size. After that, its performance also stabilizes.

4.6.4 SHiX Updating

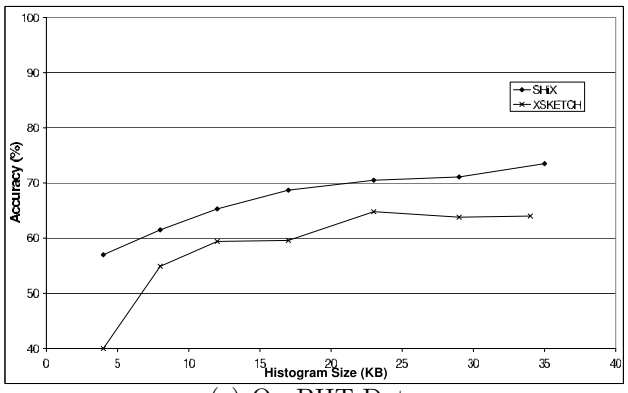
In this subsection, we investigate the performance of our proposed updating operation on *SHiX* upon the insertion of new documents into XML databases. On the Xmark, we sequentially insert new documents of sizes, $25MB$, $20MB$, $15MB$, $10MB$ and $5MB$, which are generated through the data generator. On the BHT, since we do not have other real data, we divide the original data into parts of sizes, $20MB$, $15MB$, $10MB$ and $5MB$; they are then sequentially inserted



(a) On DBLP Data

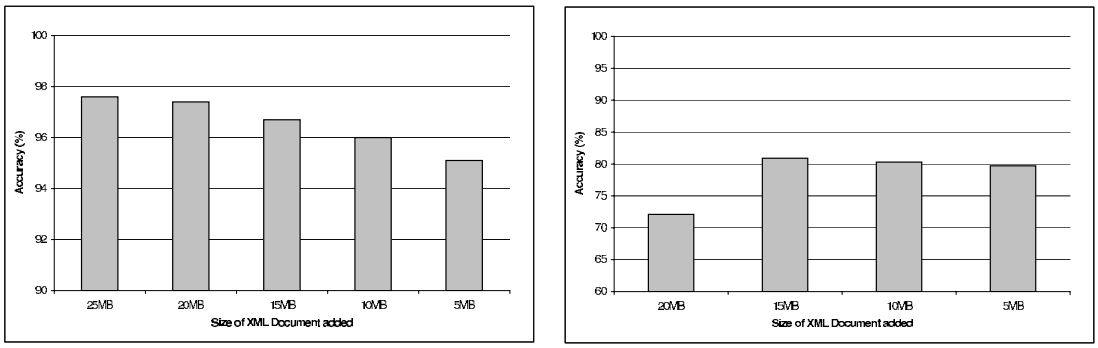


(b) On Xmark Data



(c) On BHT Data

Figure 4.6: SHiX vs Xsketch



(a) On Xmark Data (b) On BHT Data

Figure 4.7: SHiX Update Performance upon Insertion of New Document

into databases. Suppose that the original data is empty, and Xmark's histogram memory limit is $20KB$, BHT's is $40KB$. Upon the insertion of a new document, we construct its own $G_{H_{new}}$ of the maximal size and then merge $G_{H_{new}}$ with the existing $G_{H_{old}}$. Beginning with the label-split G_H as a result of merging $G_{H_{new}}$ and $G_{H_{old}}$, we refine G_H iteratively until its size reaches the limit. The results are presented in Figure 4.7. The Y-axis represents the new estimation accuracy on the twig pattern expression query load after each insertion. We can see that on both datasets, the overall performance of *SHiX* only fluctuates slightly. This observation experimentally testify that *SHiX* adapts well to the insertion update operation on XML databases. Note that on the BHT data, upon the second insertion, the G_H even achieve a higher estimation accuracy. This phenomenon results from the fact that the second BHT file's inherent structure is more regular than the first one's.

4.7 Related Work

Most of previous estimating proposals for XML focus on the tree-structured data, such as the path tree, the Markov Table [73], correlated subpath tree [74], the position histogram [76] and StatiX [75]. The path tree and Markov Table further limit the estimated path expression to be *simple*, or non-branching. The path

tree is based on the concept of bisimilarity [20, 21]. Since the path tree is usually larger than the available memory, it needs to be summarized using a special tag name "*", which can be matched to any tag. The selectivity of a path expression is estimated through navigating the summary data structure to find a set of matching summary nodes. The total frequency of these summary nodes is the selectivity. The correlated subpath tree and the position histogram proposals take a *divide-and-conquer* approach. They store statistics of short and simple path patterns and the correlation information between them. To estimate a long and complex path query, it first decomposes the query into a set of subquery pieces and estimates the size of each piece using the summary structure; and then finally, taking their correlations into consideration, "stitch"s them together. Statix also supports the estimation of twig query patterns by summarizing the structure and values in an XML document through one-dimensional histograms. The beneficial difference is that it is scheme-aware, leveraging XML schema validators for gathering statistics. More recently, a novel bloom histogram was proposed for estimating simple path selectivity over tree XML data in [86]. It has the advantages of possessing an analytical upper bound on estimation error and being sensitive to the incremental updates(for instance, inserting of deleting nodes) on XML data.

As mentioned in the introduction, the work most related to us is the *Xsketch* synopsis [22]. It exploits the localized graph stability to approximate the path and branching distribution on a graph-structured data. In their follow-up work [23], authors also proposed an extended version of *Xsketch* to incorporate the value selection on predicates by capturing the correlation pattern between the path structure and values elements in the graph data. Over the tree-structured data model, the *Xsketch* synopsis augmented with a summarization method for approximating the cardinality of structural joins was experimentally shown to be effective in estimating the selectivity of twig pattern queries [87].

The MHIST technique for constructing the multi-dimensional histogram was mainly the work of [79]. Probably the operation on histograms most similar in purpose to our updating on *SHiX* is building dynamic multidimensional histograms for continuous data stream [88]. It actually maintains a dynamic summary structure approximating the distribution of underlying continuous streams. The histogram is derived from this dynamic structure.

4.8 Summary

In this chapter, we propose a novel framework, *SHiX*, for estimating the selectivity of twig pattern expressions on graph-structured XML databases. The *SHiX* histogram captures the inherent structures present in XML data through the numerical relationship and forward-stability percentage information between two summary nodes. With the NP-hardness result of constructing the optimal *SHiX*, we present a greedy approach of refining summary nodes gradually to achieve an effective *SHiX* within a small memory requirement. We also show that when new documents are inserted into XML databases, the *SHiX* can be updated accordingly without building it up from scratch. Our extensive experiments on XML data demonstrate that *SHiX* is an effective selectivity estimator of twig pattern expressions, and adapts well to the insertion of new documents into XML databases.

Chapter 5

Conclusion and Future Research

XML, an example of semi-structured data, poses many new challenges to database communities, which include designing indexing techniques and histograms specifically for semi-structured data. In this dissertation, we push forward the research on XML query processing on several fronts.

First, we propose an adaptive structural summary for XML data, D(k)-Index. D(k)-Index is a clean generalization of the previous 1-index and A(k)-index. It has clear advantages over them because of its dynamism. It can adjust its structure accordingly, subject to the changing query load. We have shown by experiments that it achieves improved evaluation performance over previous static structural summaries. Equally significantly, the D(k)-index has more flexible and efficient update algorithms, which are crucial to such structural summary's application. Our experiments also demonstrate the superiority of the update operations on D(k)-index over update operations proposed for previous structural summary.

Secondly, we introduce the enhanced range-based and prefix-based encoding schemes for XML data and an external-memory index structure, XL^+-tree, which efficiently implements the various location steps specified by the XPath query language. We define all search problems required by the XPath locating process under

both schemes and present their corresponding search operations on the $XL+$-trees. The worst case I/O cost of all search operations are analyzed, along with the amortized I/O cost of the insertion and deletion operations on the $XL+$-tree. We also experimentally investigate the performance of the proposed $XL+$-tree by comparing it with existing indexing techniques for XML data. Results show that $XL+$-tree outperforms by a wide margin.

Thirdly, we propose a novel framework for estimating the size of twig path expressions over XML data. The *SHiX* structual histogram keeps the information of numeric relationship and forward stability between summary nodes. We define the problem of building the optimal *SHiX* and, because of its intractability, present a greedy approach to construct effective *SHiX* efficiently. It is also shown that *SHiX* possesses the adaptivity upon a typical update operation upon XML database, inserting a new document. Our comparative experiments with previous proposals validate the effectiveness of the *SHiX* framework.

As for the future research, there are lots of interesting problems on indices and histograms for XML that need to be further explored. Here we list a few that are considered important and related to our work.

1. How the structural summary can handle branching path queries effectively, or more generally how a structural summary can be incorporated into an XML query engine to facilitate more complex XML queries, remains unclear. The work of [37] is the first effort of this direction. But Authors reminded that intriguing questions remained, for instance, how to select an optimal set of indices given a query workload and how to update indices efficiently.

2. we expect that there are better techniques to process an XPath expression based on XL^+-tree than the naive approach, which simply locates the context nodes step by step. Furthermore, the $XL+$-tree only considers the structural navigation among XML data. The Xpath language, or the full-blown XQuery

language, defines various syntax beyond location steps; for instance, it also involves value predicates. How to incorporate these definitions into the $XL+$-tree framework remains a interesting question.

3. Since *SHiX* is proposed to estimate sizes of structural twig path expressions, how to extend it to handle the twig expressions with value predicates remains unaddressed. The second interesting question about *SHiX* is how to make it adaptive to the changing query load. Given the fact that XML queries are possibly posed in the big stock of XML documents over the Internet, it becomes important that *SHiX*, which should be accomodated in limited memory space, stores only staticstics of query patterns in the recent query load.

Bibliography

[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.

[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.Ceri, S.Comai, E.Damiani, P.Fraternali, S.Paraboschi and L.Tanca, *XML-GL: A Graphical Language for Querying and Restructuring XML*, in Proceedings of WWW, 1999.

[6] S.Abiteboul, *Query Semi-structured Data*, in Proceedings of ICDT, 1997.

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

[8] 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.

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

[10] P.Bohannon, J.Freire, P.Roy, and J.Simeon, *From XML Schema to Relations: A Cost-based Approach to XML storage*, in Proceedings of ICDE, 2002.

[11] A.Deutsch, M.Fernandez, and D.Suciu, *Storing Semistructured Data with STORED*, in Proceedings of ACM SIGMOD, 1999.

[12] D.Florescu and D.Kossmann, *Storing and Querying XML Data Using an RDBMS*, IEEE Data Engineering Bulletin 22(3), 1999.

[13] J.Shanmugasundaram et al. *Relational Databases for Querying XML Documents: Limitations and Opportunites*, in Proceedings of VLDB, 1999.

[14] J.Shanmugasundaram et al. *A General Technique for Querying XML Documents using a Relational Database System*, SIGMOD Record, September 2001.

[15] T.Shimura, M.Toshikawa, and S.Uemura, *Storage and Retrieval of XML Documents using Object-Relational Databases*, in Proceedings of DEXA, 1999.

[16] M.Yoshikawa et al., *XREL:A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases*, in ACM Transactions on Internet Technology, August 2001.

[17] I.Tatarinov and S.D.Viglas, *Storing and Querying Ordered XML Using a Relational Database System*, in Proceedings of ACM SIGMOD, 2002.

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

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

[20] T.Milo and D.Suciu, *Index Structures for Path Expressions*, in Proceedings of ICDT, 1999.

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

[22] N.Polyzotis, M.Garofalakis, *Statistical Synopses for Graph-Structured XML Databases*, in Proceedings of ACM SIGMOD, 2002.

[23] N.Polyzotis, M.Garofalakis, *Structure and Value Synopses for XML Data Graphs*, in Proceedings of VLDB, 2002.

[24] M.Henzinger, T.Henzinger, and P.Kopke, *Computing Simulations on Finite and Infinite Graphs*, in Proceedings of FOCS, 1995.

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

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

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

[28] T.Milo and D.Suciu, *Optimizing Regular Path Expressions Using Graph Schemas*, in Proceedings of ICDE, 1998.

[29] 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.

[30] N.Zhang, V.Kacholia and M.T.Ozsu, *A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML*, ICDE 2004.

[31] R.Ramakrishnan and J.Gehrke, *Database Management Systems(Third Edition)*, McGraw-Hill, 2002.

[32] D.Lee and M.Yannakakis, *Online Minimization of Transition Systems (extended abstract)*, in Proceedings of ACM Symposium on the Thoery of Computing(STOC), 1992.

[33] S.Abiteboul, P.Buneman and D.Suciu, *Data on the Web: From Relations to Semistructured Data and XML*, Morgan Kaufmann Publishers, 1999.

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

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

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

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

[38] C.W.Chung, J.K.Min and K.Shim, *APEX:An Adaptive Path Index for XML Data*, in Proceedings of ACM SIGMOD, 2002.

[39] I.Tatarinov, Z.G.Ives, A.Y.Halevy and D.S.Weld, *Updating XML*, SIGMOD, 2001.

[40] K.Yi, H.He, I.Stanoi, and J.Yang, *Incremental Maintenance of XML Structural Indexes*, ACM SIGMOD, 2004.

[41] 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.

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

[43] M.P.Consens and T.Milo, *Optimizing Queries on Files*, in Proceedings of ACM SIGMOD, 1994.

[44] M.P.Consens and T.Milo, *Algebras for Querying Text Regions*, in Proceedings of ACM PODS , 1995.

[45] D.Srivastava, S.Al-Khalifa, H.V.Jagadish, N.Koudas, J.M.Patel, and Y.Wu, *Structural Joins: A Primitive for Efficient XML Query Pattern Matching*, in Proceedings of ICDE, 2002.

[46] E.Cohen, H.Kaplan and T.Milo,*Labeling Dynamic XML Trees*, in Proceedings of ACM PODS 2002.

[47] N.Bruno,N.Koudas,and D.Srivastava,*Holistic Twig Joins: Optimal XML Pattern Matching*, in Proceedingws of ACM SIGMOD, 2002.

[48] S-Y.Chien, Z.Vagena, D.Zhang, V.Tsotras, and C.Zaniolo, *Efficient Structural Joins on Indexed XML Documents*, in Proceedings of VLDB, 2002.

[49] H.F.Jiang, H.J.Lu, W.Wang and B.C.Ooi, *XR-Tree: Indexing XML Data For Efficient Structural Joins*, in Proceedings of ICDE, 2003.

[50] H.F.Jiang, W.Wang and H.J.Lu, *Holistic Twig Joins on Indexed XML Documents*, in Proceedings of VLDB, 2003.

[51] S-Y.Chien, V.J.Tsotras and C.Zaniolo, *Efficient Management of Multiversion Documents by Object Referencing*, in Proceedings of VLDB, 2001.

[52] S-Y.Chien, V.J.Tsotras, C.Zaniolo and D.Zhang, *Efficient Complex Query Support for Multiversion XML Documents*, in Proceedings of EDBT, 2002.

[53] R.Bayer, and C.McCreight, *Organization and Maintenance of Large Ordered Indexes*, *Acta Informatica 1*, 3(1972).

[54] R.Bayer, and K.Unterauer, *Prefix B-trees*, ACM Transactions on Database Systems 2,1(1977).

[55] D.Comer, *The Ubiquitous B-Tree*, Computing Survey 11(1979),121-137.

[56] P.Ferragina and R.Grossi, *The String B-Tree: A New Data Structure for String Search in External Memory and Its Applications*, Journal of ACM 46(2), 1999.

[57] A.Guttman, *R-trees: A Dynamic Index Structure for Spatial Searching*, in Proceedings of ACM SIGMOD, 1984.

[58] N.Beckmann, H.P.Kriegel, R.Schneider and B.Seeger, *The R*-tree: An Efficient and Robust Access Method for Points and Rectangles*, in Proceedings of ACM SIGMOD, 1990.

[59] Q.Chen, A.Lim and O.K.Win, *D(k)-Index:An Adaptive Structural Summary for Graph-Structured Data*, in Proceedings of ACM SIGMOD, 2003.

[60] T.Grust, *Accelerating XPath Location Steps*, In Proceedings of ACM SIGMOD, 2002.

[61] S.Abiteboul, H.Kaplan and T.Milo, *Compact Labeling Schemes for Ancestor Queries*, in Proceedings of SODA, 2001.

[62] S.Alstrup and T.Rauhe, *Improved Labeling Scheme for Ancestor Queries*, in Proceedings of SODA, 2002.

[63] H.Kaplan, T.Milo and R.Shabo, *A Comparison of Labeling Schemes for Ancestor Queries*, in Proceedings of SODA, 2002.

[64] N.S.Prywes and H.J.Gray, *The Organization of a Multilist-Type Associative Memory*, IEEE Transactions on Communication and Electronics 68, 1963.

[65] G.H.Gonnet, R.A.Baeza-Yates and T.Snider, *Information Retrieval: Data Structures and Algorithms*, Charpter 5: New Indices for Text, Prentice-Hall, 1992.

[66] A.Amir, M.Farach, Z.Galil, R.Giancarlo and K.Park, *Dynamic Dictionary Matching*, Journal of Computer and System Science 49, 1994.

[67] T.C.Hu and C.Tucker, *Optimum Computer Search Trees*, SIAM Journal of Applied Mathematics, 21:514-532, 1971.

[68] P.Buneman, S.Davidson, G.Hillebrand and D.Suciu, *A Query Language and Optimization Techniques for Unstructured Data*, in Proceedings of ACM SIGMOD, 1996.

[69] D.Gusfield, G.M.Landau and B.Schieber, *An Efficient Algorithm for All Pairs Suffix-Prefix Problem*, Information Processing Letter 41, 1994.

[70] E.M.McCreight, *A Space-Economical Suffix Tree Construction Algorithm*, Journal of ACM 23(2), 1976.

[71] U.Manber and G.Myers, *Suffix Arrays: A New Method for On-Line String Searches*, SIAM Journal on Computing 22(5), 1993.

[72] The TPIE project is available at http://www.cs.duke.edu/ tpie/.

[73] A. Aboulnaga, A.R.Alameldeen, and J.F.Naughton, *Estimating The Selectivity of XML Path Expressions for Internet Scale Applications*, in Proceedings of VLDB, 2001.

[74] Z.Chen, H.V.Jagadish, F.Korn, N.Koudas, S.Muthukrishnan, R.Ng, and D.Srivastava, *Counting Twig Matches in A Tree*, in Proceedings of ICDE, 2001.

[75] J. Freire, J.R.Haritsa, M.Ramanath, P.Roy, *StatiX:Making XML Count*, in Proceedings of ACM SIGMOD, 2002.

[76] W.Yuqing, J.M.Patel, H.V.Jagadish, *Estimating Answer Sizes for XML Queries*, in Proceedings of EDBT, 2002.

[77] H.V.Jagadish, *Linear Clustering of Objects with Multiple Attributes*, in Proceedings of ACM SIGMOD, 1990.

[78] M.Muralikrishna, D.J.Dewitt, *Equi-depth Histograms for Estimating Selectivity Factors for Multi-dimensional Queries*, in Proceedings of ACM SIGMOD, 1988.

[79] V.Poosala, Y.E.Ioannidis, *Selectivity Estimation Without The Attribute Value Independence Assumption*, in Proceedings of VLDB, 1997.

[80] Y.E.Ioannidis, V.Poosala, *Balancing Histogram Optimality and Practicality for Query Result Size Estimation*, in Proceedings of ACM SIGMOD, 1995.

[81] Y.E.Ioannidis, *Universality of Serial Histograms*, in Proceedings of VLDB, 1993.

[82] G.P.Shapiro, C.Connell, *Accurate Estimation of The Number of Tuples Satisfying a condition*, in Proceedings of ACM SIGMOD, 1984.

[83] M.V.Mannino, P.Chu, T.Sager, *Statistical Profile Estimation in Database Systems*, ACM Computing Surveys, 20(3):192-221, September 1988.

[84] S.Muthukrishnan, V.Poosala, and T.Suel, *On Rectangular Partitionings in Two Dimensions: Algorithms, Complexity, and Applications*, ICDT, 1999.

[85] W.Wang, H.F.Jiang, H.J.Lu, and J.X.Yu, *Containment Join Size Estimation: Models and Methods*, SIGMOD, 2003.

[86] W.Wang, H.F.Jiang, H.J.Lu, and J.X.Yu, *Bloom Histogram: Path Selectivity Estimation for XML Data with Updates*, VLDB 2004.

[87] N.Polyzotis, M.Garofalakis, and Y.Ioannidis, *Selectivity Estimation for XML Twigs*, ICDE 2004.

[88] L.Qiao, D.Agrawal, and A.E.Abbadi, *RHist: Adaptive Summarization over Continuous Data Streams*, CIKM 2002.

[89] XML Data Repository, http://www.cs.washington.edu/research/xmldatasets /www/repository.html.

[90] The DBLP BHT file is available at http://dblp.uni-trier.de/xml/.

[91] J.Naughton, C.Jianjun, D. DeWitt, C.Zhang, *The Niagara Internet Query System*, Technical Report. Available at http://www.cs.wisc.edu/niagara/.