Adaptive Partitioned Indexes for Efficient XML Keyword Search

Sung Jin Kim

Department of Computer Science, University of California Los Angeles (UCLA), Los Angeles, California, USA
sjkim@cs.ucla.edu

Hyungdong Lee

School of Computer Science and Engineering, Seoul National University, Seoul, Korea
hdlee@idb.snu.ac.kr

Hyoung-Joo Kim

School of Computer Science and Engineering, Seoul National University, Seoul, Korea
hjk@idb.snu.ac.kr

A query result of an XML keyword search is usually defined as a set of the most specific elements containing all query keywords. Search systems find the query result by considering the combinations of all elements in the inverted indexes of the query keywords. However, we conclude that it is not necessary to consider the combinations of all the elements, when an "effective result depth" (which represents how deeply nested elements are eligible for the query result) is given. This paper describes a way to construct partitioned indexes on the effective result depth, guaranteeing that the combinations of elements in different partitions never produce result elements. Therefore, search systems can find query results by considering only combinations of elements in the same partitions. Partitioned indexes are adaptable; when an effective result depth is changed, partitioned indexes constructed on the original depth can be used efficiently without being reconstructed physically on the changed depth. The experimental results show that our approach worked quite well in most cases.

ACM Classification: H.3.1 (Information Storage and Retrieval – Content Analysis and Indexing – Indexing methods); I.7.2 (Document and Text Processing – Document Preparation – Index generation); H.2.4 (Database Management – Systems – Textual databases); H.3.3 (Information Storage and Retrieval – Information Search and Retrieval – Information filtering)

1. INTRODUCTION

Keyword search, which is extensively used for searches over flat HTML documents on the web, is a simple and effective paradigm for information discovery. Many researchers (Carmel *et al*, 2003; Cohen *et al*, 2003; Florescu *et al*, 2000; Fuhr and Grobjohann, 2004; Guo *et al*, 2003; Hritidis *et al*, 2003; Li *et al*, 2004; Xu and Papakonstantinou, 2005) have studied how to effectively apply this useful paradigm to searches over XML documents. XML Keyword search makes it possible for users to obtain relevant information without having to know complex query syntaxes (Robie *et al*, 1998; Clark and DeRose, 1999; Deutsch *et al*, 1998) and structures (e.g., DTDs) of XML documents.

Manuscript received: 10 October 2005
Communicating Editor: John Yearwood

One of the main features of an XML keyword search is that a query result is a set of XML elements, instead of entire documents. In general, the query result is composed of the most specific elements containing all query keywords. For example, let us consider the XML document shown in Figure 1, and suppose a query is composed of keywords "Schmidt" and "XML". Then, the query result is a set of the first <collection> element and the fifth <paper> element. The second <collection> element cannot be a result element, even though it contains all query keywords, because it is not the most specific (i.e., it includes the fifth <paper> element that already contains both query keywords).

Using inverted indexes for efficient keyword searches are already a proven, accepted technique. The granularity of inverted indexes for XML keyword searches is an element. There has been a lot of research on how to efficiently handle element-based inverted indexes. Guo *et al* (2003) proposed the DIL (Dewey Inverted List) index structure, where the elements are sorted by their identifiers, called Dewey Identifier. They obtained query results by reading the elements in each of the indexes sequentially. The indexes are never read more than once under the structure. Xu and Papakonstantinou (2005) used a B-tree based index structure for XML elements, where the non-leaf nodes are cached in main memory, and successfully reduced the number of disk accesses.

Previous research focused on how to efficiently organize element-based inverted indexes and how to quickly obtain query results with their indexes. Search systems consider all elements containing query keywords and find which combinations of the elements produce the least common ancestors appropriate for their result elements. Search systems try to read all the elements (thereafter, physical disk accesses can be reduced if elements are cached) in the inverted indexes of the query keywords.

In this paper, we conclude that it is not necessary to consider the combinations of all elements containing query keywords to obtain query results, when the "effective result depth" is given. An "effective result depth" represents how deeply nested elements are eligible as query result elements.

```
<data>
    <collection no="1">
        <paper no="1">      <author>  Y. Wu  </author>
                            <title>  Using Histograms to Estimate Answer Sizes For XML Queries </title>
        </paper>
        <paper no="2">      <author>  A. Schmidt  </author>
                            <title>  Priority in DBMS resource scheduling  </title>
        </paper>
        <paper no="3">      <author>  L. Mignet  </author>
                            <title>  The XML Web: a first study  </title>
        </paper>
        <paper no="4">      <author>  S. Cohen  </author>
                            <title>  A Semantic Search Engine for XML  </title>
        </paper>
    </collection>
    <collection no="2">
        <paper no="1">      <author>  A. Schmidt  </author>
                            <title>  Why and How to Benchmark XML Databases  </title>
        </paper>
    </collection>
    <collection no="3">
    </collection>
</data>
```

Figure 1: An example of an XML document

For example, in Figure 1, let us suppose that an administrator does not want <data> and <collection> elements to be returned as query result elements, simply because those elements give too broad, general information to users. Then, the "effective result depth" is 2 because the administrator wants the result elements to be nested at least twice. Let e_1 and e_2 be the elements that are nested in different <paper> elements. Then, for any query to the document, even though all of the query keywords are contained in either e_1 or e_2, the least common ancestor of e_1 and e_2 never becomes a result element. That is because it is certain that the least common ancestor is either a <data> or a <collection> element. Therefore, we do not need to consider the combination of e_1 and e_2 for any queries if the effective result depth is 2. Our research is motivated by this simple, yet promising idea.

Recently, Botev and Shanmugasundaram (2005) were interested in how to avoid reading unnecessary elements while finding query results. Their search system first interpreted the context of queries that users issue, and obtained query results by considering the combinations of only elements belonging to the context. Our approach differs from theirs in terms of using only topology information of elements, without the step of interpreting the context of a query.

This paper describes a way to construct partitioned inverted indexes on an effective result depth. The partitioned indexes are composed of a number of partitions, and guarantee that the combinations of elements in different partitions never produce least common ancestors eligible for result elements. Therefore, we only need to selectively read the elements in the same partitions and try to find which least common ancestors of those elements are eligible as result elements. In our approach, the number of disk accesses can be reduced significantly because we often do not read whole inverted indexes of query keywords. When there is no element in a partition of an index, the elements belonging to the same partitions of the other indexes are not read.

An effective result depth is sometimes apparent. For example, the DBLP site (DBLP, 2005) provides users with an XML keyword search service to find either paper or author information. The nodes containing paper or author information are nested in other elements at least once. Therefore, the effective result depth is certainly 1. On the other hand, when an effective result depth is unclear, administrators may use a possible best value for the effective result depth. Even though the partitioned indexes on the value have been constructed, the value can be changed to another value if necessary. The partitioned indexes are adaptable, which means that partitioned indexes constructed on the original value can be used efficiently without being reconstructed physically on the changed value.

Our approach was experimentally tested in two collections of XML documents. The first collection is composed of one XML document that has been used practically in the DBLP site. The second collection is composed of the XML documents that were provided by the INEX2003 (INEX, 2003), which was established for the purpose of testing XML search systems. The experimental results showed that our approach worked quite well in most cases.

The paper is organized as follows. Section 2 presents basic concepts and requirements of the partitioned index structure. Section 3 presents a way to organize partitioned indexes and search query results. Section 4 shows evaluation results in two test collections. Finally, Section 5 contains the conclusion.

2. PRELIMINARY

An XML document is regarded as an ordered, node labeled tree. Figure 2 represents the tree for the document shown in Figure 1. XML elements correspond with tree nodes. The relation between the child and parent nodes is denoted as a directed line. Attributes of an element can be regarded as child elements of the element. For simplicity of our explanation, attributes will be ignored in this

paper. Data strings nested in an element are linked to each other by a straight line. Node identifiers are denoted in parentheses. The depth of the root node (n_1) is zero. When the depth of a node is *d*, the depths of its child nodes are (*d+1*).

Consider a query composed of several keywords. There are two possible semantics for keyword search queries: conjunctive and disjunctive keyword query semantics. In the query result, the former semantic returns nodes that contain all of the query keywords, and the latter semantic returns nodes that contain any of the query keywords. In this paper, we have chosen to use the conjunctive keyword query semantic. For example, when a user wants to find papers written by Schmidt on XML-related topics, the user can issue two keywords "XML" and "Schmidt", simultaneously, as a query.

Definition 1. Let us suppose that there is an XML tree. The *effective result depth* of the tree is a depth of *d*. For any query to the tree, depths of the result nodes should be greater than or equal to *d*.

For example, given the XML tree in Figure 2, let us suppose that an administrator wants result nodes to be composed of <paper>, <author>, and <title> nodes, simply because she thinks <data> and <collection> nodes give too broad information to users. Then, the effective result depth of the tree is 2. For another example, the DBLP site provides users with an XML keyword search service to find either paper or author information. The depths of the nodes containing paper information are 1. The depths of the nodes containing author information are 2. That is, the depths of result nodes are either 1 or 2, and the effective result depth is 1.

Definition 2. Let there be a node in an XML tree. If the depth of the node is greater than or equal to the effective result depth of the tree, then the node is called an *effective* node to the depth (and we also say that the node is *effective* to the depth).

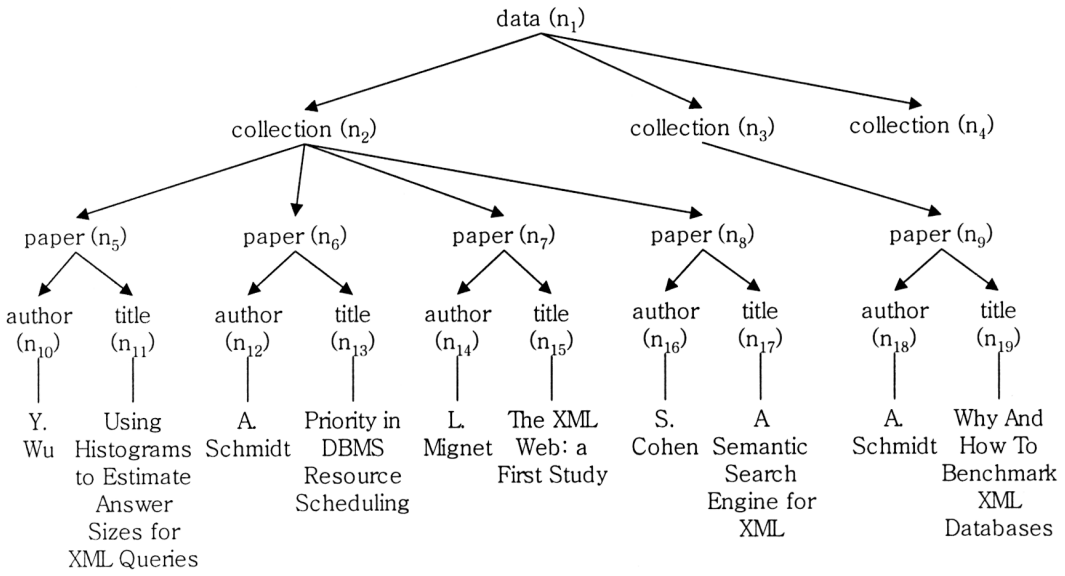


Figure 2: An XML tree of the document shown in Figure 1

The search result of a query is defined as the set of the nodes satisfying the two conditions. First, the nodes should be effective. Second, the nodes should contain at least one occurrence of all of the query keywords, after excluding the occurrences of the keywords in child nodes that already contain all of the query keywords. The second condition is the same as the query result condition of Guo *et al* (2003). For example, let us suppose that the query keywords are "XML" and "Schmidt", and the effective result depth is 2. Then, n_8 is a query result node because it is effective, contains all the query keywords, and does not have any child node containing all the query keywords. On the other hand, n_2 is not a query result node because it is not effective, although it satisfies the second condition.

A partition of a keyword is an inverted index that indexes all or some of the nodes containing the keyword. A partitioned index of a keyword is a set of partitions of the keyword, where there is no node indexed in more than one partition. Let us suppose there are keywords x and y. A partitioned index of x on a depth d, composed of M partitions ${}_xP_0$, ${}_xP_1$, …, ${}_xP_{M-1}$, and a partitioned index of y on the depth d, composed of M partitions ${}_yP_1$, …, ${}_yP_{M-1}$, should satisfy the following partitioning and merging requirements, where $\in$ means that the left side node is indexed in the right side partition; $\cup$ produces a partition that indexes all the nodes indexed in the partitions of both the sides; = means that the nodes indexed in the left side partition are the same as those indexed in the right side partition.

- Partitioning requirement:
  Let us suppose that the effective result depth is d. Let us suppose $n_x \in {}_xP_i$ $(0 \le i < M)$ and $n_y \in {}_yP_j$ $(0 \le j < M)$. If the least common ancestor of n_x and n_y is effective to d, then i is equal to j.
- Merging requirements:
  For any non-negative integer d' less than d $(0 \le d' < d)$, let us suppose that the effective result depth is changed from d to d'. Let us suppose a partitioned index of keyword x on depth d', is composed of M' partitions ${}_xP'_0$, ${}_xP'_1$, …, ${}_xP'_{M'-1}$, and a partitioned index of keyword y on depth d', is composed of M' partitions ${}_yP'_0$, ${}_yP'_1$, …, ${}_yP'_{M'-1}$ (both the partitioned indexes on depth d' satisfy the partitioning requirement). Then,

  (1) ${}_xP_0 \cup {}_xP_1 \cup \ldots {}_xP_{M-1} = {}_xP'_0 \cup {}_xP'_1 \cup \ldots {}_xP'_{M'-1}$, and ${}_yP_0 \cup {}_yP_1 \cup \ldots {}_yP_{M-1} = {}_yP'_0 \cup {}_yP'_1 \cup \ldots {}_yP'_{M'-1}$
  (2) $\forall n_x \in {}_xP_m$ $(0 \le m < M)$, $n_x \in {}_xP'_{m'}$ $(0 \le m' < M')$, and $\forall n_y \in {}_yP_m$, $n_y \in {}_yP'_{m'}$

The partitioning requirement calls for the elements whose least common ancestor is effective to be indexed in the partitions with the same partition number. The partitions with the same partition number are said to be the same partitions. For example, ${}_xP_1$ and ${}_yP_1$ are the same partition. The partitioning requirement makes it possible that a search system can obtain query results by considering the combinations of elements in the same partition. Let us suppose that the query keywords are "XML" and "Schmidt", and the effective result depth is 2. Because n_{19}, containing "XML" and n_{18}, containing "Schmidt" have an effective least common ancestor, n_{19} and n_{18} are indexed in the same partitions. On the other hand, the partitioning requirement does not necessitate that the nodes whose least common ancestor is not effective should be indexed in different partitions. Therefore, n_{12} containing "Schmidt" and n_{19} could be indexed in the same partitions even though their least common ancestor is not effective. Search systems need to find which combinations of the elements in the same partitions of query keywords produce least common ancestors satisfying both the query result conditions. However, it is certain that search systems do not need to consider the combinations of the elements in the different partitions.

When the effective result depth is changed from d to d', there are two cases: $d < d'$ and $d > d'$. The first case, i.e., $d < d'$, means that administrators give users more specific query result elements. The solution for the first case is very simple. After obtaining query result nodes that are effective to depth d, with the partitioned index on depth d, search systems give users only the nodes that are effective to d'. The second case, i.e., $d > d'$, means that administrators allow more general nodes to be included in the query result if the nodes satisfy the second query result condition. The problem with the second case is the fact that the combinations of two nodes in different partitions of partitioned indexes on depth d should be considered for the new effective result depth d'. That is because nodes whose least common ancestor is effective to d' not d could be indexed in different partitions of partitioned indexes on d. The solution for the second case is to construct the partitioned indexes on the depth d' virtually without reconstructing partitioned indexes on depth d' physically.

Merging requirements are established to make it efficient for search systems to obtain query result nodes that are effective to d' ($d' < d$) with the partitioned indexes on depth d. The merging requirement (1) requires that all nodes indexed in partitioned indexes on d' should be indexed in partitioned indexes on d. If a node is not effective to a depth d, then the combinations of the node and any other nodes will not produce the effective least common ancestor. However, the node could be effective to a depth d' and join the combinations producing the query result nodes effective to d'. If we do not index the node in the partitioned index on depth d, then we lose the node that is necessary for constructing partitioned indexes on depth d' virtually.

The merging requirement (2) requires that all nodes in a partition (i.e., ${}_xP_m$ or ${}_yP_m$) of the partitioned index on d should be indexed in only one partition (i.e., ${}_xP'_{m'}$ or ${}_yP'_{m'}$) of the partitioned index on d'. If two nodes in a partition of the index on depth d are indexed in two partitions of the index on d', then reading two virtual partitions could require the search systems to read the same physical partition twice. The merging requirement (2) makes it possible that all physical partitions are read only once while search systems scan all partitions of the virtual indexes.

3. CONSTRUCTING PARTITIONED INDEXES AND SEARCHING QUERY RESULTS

In this section, we describe a way to organize partitioned indexes satisfying the partitioning and the merging requirements, suggested in the previous section, with a partition function. We also describe how to search query results with the partitioned indexes.

3.1 Partition Function

A partition function receives an effective result depth and a node to be indexed, and returns a partition number indicating the partition where the node should be indexed. Through using the partition function, search systems can promptly determine an appropriate partition for a node while reading XML documents sequentially. We will first explain a basic idea of how to design a partition function. In this subsection, let us assume that there are two sibling nodes n_α and n_β whose depths are d, and let us assume that there is also a node n_χ, the least common ancestor of n_α and n_β, whose depth is d'.

For the partitioning requirement, with an effective result depth of d, a partition function must return the same partition numbers for n_α (or n_β) and all its descendents because the least common ancestors produced by the combinations of some (or all) of those nodes, i.e., n_α and all its descendents, are always effective. When the effective result depth is d', the partition function must return the same partition numbers for n_χ and its descendants, as before. On the other hand, we may force descendants of n_α and n_β not to be indexed in the same partitions, because any combinations of n_α's descendants and n_β's descendants do not produce the least common ancestors effective to

depth d. However, given thousands or millions of sibling nodes of n_α, indexing all sibling nodes in different partitions could result in large management costs.

A partition factor δ is the number of partitions allowed to the sibling nodes of n_α. When the number of the sibling nodes is greater than δ, we allow the indexing of n_α and n_β, as well as their descendants, in the same partition. Note that the partitioning requirement does not demand the two nodes that have no least common ancestor to be indexed in different partitions. When the total number of the sibling nodes is 0 or less than δ, as many as δ partitions are also created for the sibling nodes. Suppose the effective result depth is (d+1). Then, the child nodes of n_α, which are indexed in the same partition when the effective result depth is d, could be divided into as many as δ partitions. If n_α and n_β are indexed in the same partitions of the partitioned indexes on d, only δ (not 2δ) partitions are allowed for the child nodes of both nα and n_β.

Given the partition factor δ and the effective result depth d, the total number of partitions of an index is δ^d. When d is 0, there is only one partition because the root node and its all descendants are indexed in the same partition. The total number of partitions is δ itself when d is 1, because all nodes on depth d are the siblings that have the same parent (i.e., the root). When d is 2, the total number of partitions is δ^2, because each of the δ partitions is divided to as many as δ partitions. As a partition factor δ becomes smaller, the total number of partitions decreases and descendants of n_α and descendants of n_β are likely to be indexed in the same partition. On the other hand, as the partition factor δ becomes bigger, the total number of partitions increases, and descendants of both n_α and n_β tend to be indexed in different partitions.

For the merging requirement (1), n_χ should be indexed in partitioned indexes on d even though n_χ never joins the combinations producing result nodes effective to d. For the merging requirement (2), a partition function assigns n_χ into the partition where n_χ's first descendant node on depth d is indexed. If the indexes on d' satisfy the partitioning requirement, n_χ and all its descendants (including n_α and n_β) are indexed in the same partitions of the indexes. Then, all nodes of the partition, where n_α (or n_β) is indexed, of the index on depth d are indexed in the partition, where n_χ is indexed, of the index on depth d'.

$OR_d(n)$ is defined by equation (1). OR(n) is a function that returns the order of node n among its siblings. The order starts with zero. For example, in Figure 2, $OR(n_8)$ is 3 because n_8 is indexed in the fourth position among the siblings (i.e., n_5, n_6, n_7, and n_8). $A_d(n)$ is defined for only the nodes whose depths are greater than d. $A_d(n)$ returns the ancestor node of n on depth d. For example, $A_2(n_{14})$ returns n_7, and $OR_2(n_{14})$ is $OR(A_2(n_{14})) = OR(n_7) = 2$. $D_d(n)$ is defined for only the nodes whose depths are less than d. $D_d(n)$ returns the first descendant node of n on depth d. For example, $D_2(n_2)$ returns n_5. $OR_2(n_2)$ is $OR(D_2(n_2)) = OR(n_5) = 0$. If there is no descendant node of node n on depth d, $D_d(n)$ returns a virtual node assuming that the virtual node is the first node among the siblings on depth d. For example, $OR_2(n_4)$ is $OR(D_2(n_4)) = 0$. $OR_d(n)$ always returns 0 if the depth of n is smaller than d.

$$
\begin{aligned}
OR_d(n) &= OR(A_d(n)) && \text{(when the depth of n} > d\text{)} \\
&= OR(n) && \text{(when the depth of n} = d\text{)} \\
&= OR(D_d(n)) = 0 && \text{(when the depth of n} < d\text{)}
\end{aligned} \qquad (1)
$$

$PF_d(n)$ is a partition function that returns the number of partition where node n should be indexed, when the effective result depth is d. $PF_d(n)$ is defined by equation (2). When the depth of n is greater than d, $OR_d(n)$ returns the same partition numbers for the ancestor node of n on depth d. Therefore, a node whose depth is d and all its descendants have the same partition number. ($OR_d(n)$ mod δ) makes the siblings on depth d are divided into as many as δ partitions.

$$\begin{aligned} PF_d(n) &= (OR_d(n) \bmod \delta) + (PF_{d-1}(n) * \delta) && \text{(when } d > 0\text{)} \\ &= 0 && \text{(when } d = 0\text{)} \end{aligned} \quad (2)$$

$PF_d(n)$ can be expressed as the right side of equation (3).

$$\begin{aligned} PF_0(n) &= 0 \\ PF_1(n) &= (OR_1(n) \bmod \delta) \\ PF_2(n) &= (OR_2(n) \bmod \delta) + (OR_1(n) \bmod \delta) * \delta \\ PF_3(n) &= (OR_3(n) \bmod \delta) + (OR_2(n) \bmod \delta) * \delta + (OR_1(n) \bmod \delta) * \delta^2 \\ \ldots &= \qquad\qquad \ldots \\ PF_d(n) &= \sum_{i=1}^{d} ((OR_i(n) \bmod \delta) * \delta^{d-i}) \end{aligned} \quad (3)$$

Theorem 1. Suppose that the effective result depth is d. If all nodes having the same partition number by $PF_d(n)$ are indexed in the same partitions of partitioned indexes, then the indexes satisfy the partitioning requirement.

Proof. Let us suppose that we select some nodes from the nodes having the same partition number. First, if the least common ancestor of the selected nodes is not effective, it is orthogonal to the partitioning requirement whether $PF_d(n)$ returns the same partition numbers for the selected nodes or not. That is because the partitioning requirement does not put any constraints on the nodes whose least common ancestor is not effective. Second, when the least common ancestor of the selected nodes is effective, we will prove that $PF_d(n)$ always returns the same partition numbers for the selected nodes. Let n_x and n_y be the selected nodes. Let n_e be the least common ancestor of n_x and n_y. Let n_E be an ancestor node of n_e on depth d (n_E and n_e are identical when the depth of n_e is d). Then, $PF_d(n_x) = PF_d(n_E)$ because $OR_i(n_x)$ is always the same as $OR_i(n_E)$ for i values between 1 and d, in equation (3). $PF_d(n_y) = PF_d(n_E)$ as before. As a result, $PF_d(n_x) = PF_d(n_y)$

Lemma 1. Let us suppose two nodes n_a and n_b. If $PF_d(n_a) = PF_d(n_b)$, then $(OR_i(n_a) \bmod \delta) = (OR_i(n_b) \bmod \delta)$ for any i value that is an integer between 1 and d.

Proof. We prove Lemma 1 with mathematical induction. First, we show that the proposition holds for a base case (i.e., $d = 0$) as follows:

$PF_0(n_a) = PF_0(n_b) \rightarrow (OR_0(n_a) \bmod \delta) = (OR_0(n_b) \bmod \delta)$

When $d = 0$, only 0 is considered as the value of i. $PF_0(n_a)$, $PF_0(n_b)$, $OR_0(n_a)$, and $OR_0(n_b)$ are 0. Second, we show that whenever the proposition holds for some value of d, it must hold for the next number as well. So, we'll start with the original formula and show that when it is true for some $(d - 1)$, then the formula must also work for d. We can consider the value of i as 1, 2, …, $(d - 1)$.

$$\begin{aligned} PF_{d-1}(n_a) = PF_{d-1}(n_b) \quad &\rightarrow (OR_1(n_a) \bmod \delta) = (OR_1(n_b) \bmod \delta), \\ &\rightarrow (OR_2(n_a) \bmod \delta) = (OR_2(n_b) \bmod \delta), \\ &\qquad \ldots \\ &\rightarrow (OR_{d-1}(n_a) \bmod \delta) = (OR_{d-1}(n_b) \bmod \delta) \end{aligned}$$

Now, we do an expansion on $PF_d(n_a)$ and $PF_d(n_b)$.

$$PF_d(n_a) = PF_d(n_b)$$

$$(OR_d(n_a) \bmod \delta) + (PF_{d-1}(n_a) * \delta) = (OR_d(n_b) \bmod \delta) + (PF_{d-1}(n_b) * \delta)$$

where $(OR_d(n_a) \bmod \delta) = (OR_d(n_b) \bmod \delta)$ because $(PF_{d-1}(n_a) * \delta) = (PF_{d-1}(n_b) * \delta)$. Therefore, the proposition will still hold for d as follows:

$PF_d(n_a) = PF_d(n_b)$
$\rightarrow (OR_1(n_a) \bmod \delta) = (OR_1(n_b) \bmod \delta)$,
$\rightarrow (OR_2(n_a) \bmod \delta) = (OR_2(n_b) \bmod \delta)$,
…
$\rightarrow (OR_{d-1}(n_a) \bmod \delta) = (OR_{d-1}(n_b) \bmod \delta)$
$\rightarrow (OR_d(n_a) \bmod \delta) = (OR_d(n_b) \bmod \delta)$

Theorem 2. Suppose that the effective result depth is changed from d to d' and that d' is smaller than d. There is a partitioned index where the nodes that have the same partition number from $PF_d(n)$ are indexed in the same partition. There is another partitioned index where the nodes that have the same partition number from $PF_{d'}(n)$ are indexed in the same partition. Both the indexes satisfy the merging requirements.

Proof. It is trivial that both the indexes satisfy the first merging requirement because all nodes have partition numbers for $PF_d(n)$ and $PF_{d'}(n)$. Now, to show both the indexes satisfy the second merging requirement, we prove the proposition that if $PF_d(n_a) = PF_d(n_b)$ for any two nodes n_a and n_b, then $PF_{d'}(n_a) = PF_{d'}(n_b)$. $PF_d(n_a)$ and $PF_d(n_b)$ can be expressed as follows.

$$PF_d(n_a) = \sum_{i=1}^{d'}((OR_i(n_a) \bmod \delta) * \delta^{d-i}) + \sum_{i=(d'+1)}^{d}((OR_i(n_a) \bmod \delta) * \delta^{d-i})$$
$$= PF_{d'}(n_a) + \sum_{i=(d'+1)}^{d}((OR_i(n_a) \bmod \delta) * \delta^{d-i})$$
$$PF_d(n_b) = \sum_{i=1}^{d'}((OR_i(n_b) \bmod \delta) * \delta^{d-i}) + \sum_{i=(d'+1)}^{d}((OR_i(n_b) \bmod \delta) * \delta^{d-i})$$
$$= PF_{d'}(n_b) + \sum_{i=(d'+1)}^{d}((OR_i(n_b) \bmod \delta) * \delta^{d-i})$$

$PF_d(n_a) = PF_d(n_b)$ can be expressed as follows:

$$PF_{d'}(n_a) + \sum_{i=(d'+1)}^{d}((OR_i(n_a) \bmod \delta) * \delta^{d-i}) = PF_{d'}(n_b) + \sum_{i=(d'+1)}^{d}((OR_i(n_b) \bmod \delta) * \delta^{d-i})$$

$PF_{d'}(n_a) = PF_{d'}(n_b)$ because $\sum_{i=(d'+1)}^{d}((OR_i(n_a) \bmod \delta) * \delta^{d-i})$ equals to $\sum_{i=(d'+1)}^{d}((OR_i(n_b) \bmod \delta) * \delta^{d-i})$

according to Lemma 1. Therefore, if $PF_d(n_a) = PF_d(n_b)$, then $PF_{d'}(n_a) = PF_{d'}(n_b)$.

Figure 3 shows partition numbers for each node of the tree shown in Figure 2 according to changes in d and δ. The dotted lines represent the effective result depths. When δ=1 or d=0, partition numbers for all nodes are zero and all nodes are indexed in only one partition, which means that indexes are not partitioned at all.

Let us see the case where δ is 3 and d=2. Three partitions are allowed for the sibling nodes (i.e., n_5, n_6, n_7, and n_8) whose parent node is n_2. $PF_2(n_5) = PF_2(n_8) = 0$, $PF_2(n_6) = 1$, and $PF_2(n_7) = 2$. Nodes n_5, n_6, and n_7 are indexed in the partitions 0, 1, and 2, respectively. Node n_8 is indexed in the partition 0 where node n_5 is indexed. Descendants of n_5, n_6, n_7, and n_8 are indexed in the same partition where n_5, n_6, n_7, and n_8 are indexed, respectively. Nodes n_1 and n_2 are indexed in the partition 0 where their first descendant node (n_5) on depth 2 is indexed. Three additional partitions

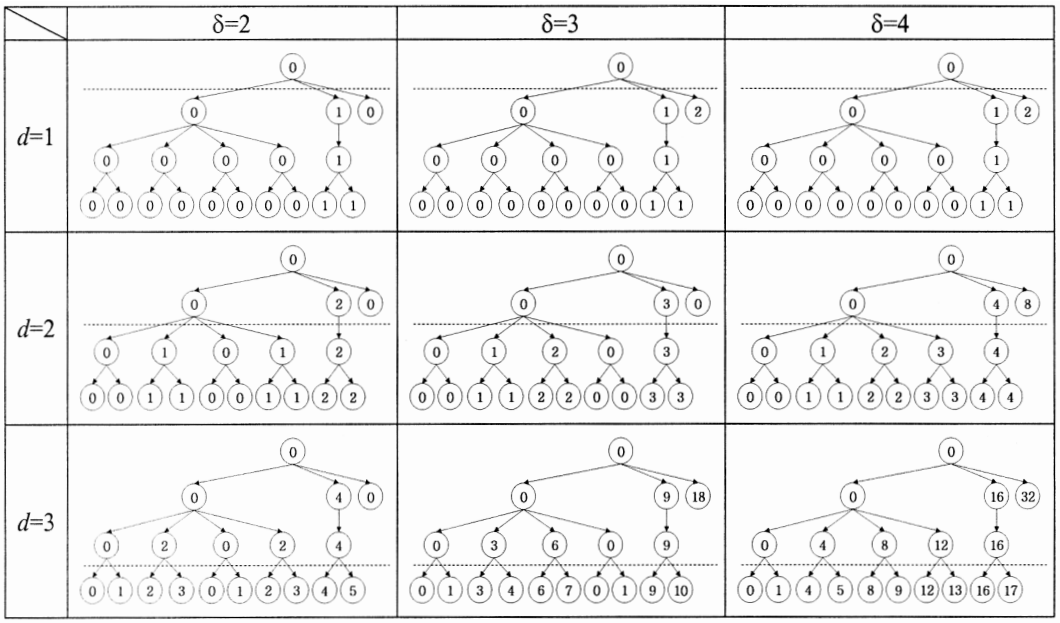


Figure 3: Partition number computed by $PF_d(n)$ for each node of the tree shown in Figure 2

are allowed for the sibling nodes whose parent is n_3. Node n_9 is indexed in partition 4. There is no node indexed in partitions 5 and 6. Node n_3 is indexed in partition 4, where its first descendant (n_9) on depth 2 is indexed. The other three partitions (partitions 6, 7, and 8) are allowed for the siblings whose parent is n_4. Node n_4 is indexed in partition 6 because the first descendant node of n_4 (if the descendant existed) is indexed in partition 6. There is no node indexed in partitions 7 and 8.

Figure 4 conceptually shows partitioned indexes of keywords "XML" and "Schmidt". A partition indexing no nodes is referred to as an empty partition. Empty partitions are not represented in Figure 4. Figure 4(a) shows the indexes that are not partitioned. When users issue "Schmidt" and "XML" as a query, search systems might need to consider the eight combinations between the four nodes containing "XML" and the two nodes containing "Schmidt". Note that the DIL index structure makes it possible for search systems to read the nodes in each of the indexes sequentially to obtain query results. If the indexes were organized with the DIL structure, there would be five required combinations, namely n_{11} and n_{12}, n_{15} and n_{12}, n_{15} and n_{18}, n_{17} and n_{18}, and n_{19} and n_{18}.

Figure 4(b) shows the indexes on depth 2 with partition factor 3. Partitions 0, 2, and 3 of keyword "XML" include two, one, and one node(s), respectively. The partitions 1, 4, 5, 6, 7, and 8 are empty partitions. There are two nodes containing keyword "Schmidt", one of which is indexed in partition 0; the other is indexed in partition 3. Let us suppose that an effective result depth is 1. With the partitioned indexes in Figure 4(b), search systems only need to consider three combinations, namely n_{11} and n_{12}, n_{17} and n_{12}, and n_{19} and n_{18}. In addition, since partition 2 of the keyword "Schmidt" is an empty partition, we can avoid reading partition 2 of keyword "XML". However, if using the conventional indexes as in Figure 4(a), search systems still have to consider the five combinations.

Figure 4(c) shows the partitioned indexes on depth 2 with partition factor 4. As the partition factor is greater than that of Figure 4(b) by one, the indexes have 16 partitions. Since only partitions

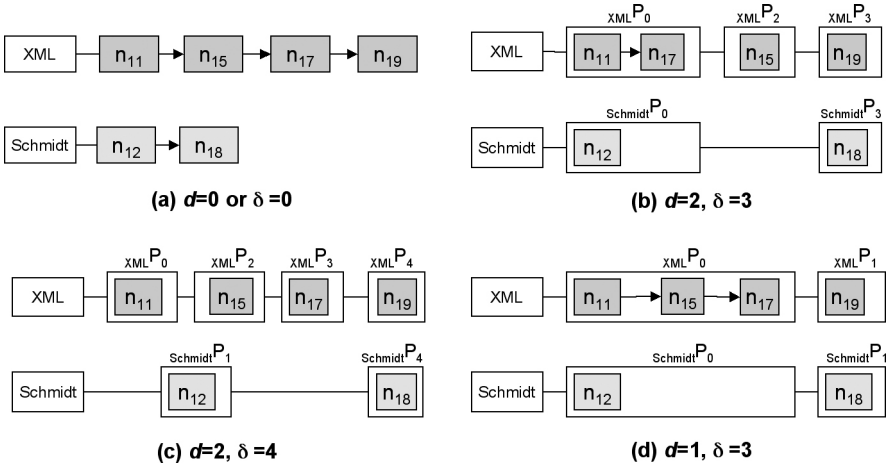


Figure 4: Inverted indexes of keywords "XML" and "Schmidt"

4 of both the indexes are not empty partitions, we consider one combination between nodes n_{19} and n_{18} in order to obtain query results. It is unnecessary to continue to search for query results if the combination does not produce the least common ancestor corresponding to the search result definition.

Figure 4(d) shows the partitioned indexes on depth 1 with partition factor 3. Suppose that an effective result depth is 1. Then, we can use the partitioned indexes on depth 1 as in Figure 4(d), and obtain query results by considering four combinations, namely n_{11} and n_{12}, n_{15} and n_{12}, and n_{17} and n_{12} in partitions 0, and n_{19} and n_{18} in partitions 1. If we have already constructed the partitioned indexes on depth 2 as in Figure 4(b), we can also use those indexes without physically reconstructing the indexes on depth 1. The union of partitions 0, 1, and 2 in Figure 4(b) is the same as the partition 0 in Figure 4(d). That is, considering the combinations between the nodes in the partition 0 in Figure 4(d) is the same as considering the combinations between the nodes in the partitions 0, 1, and 2 in Figure 4(b).

3.2 Searching Query Results with Partitioned Indexes

The process of searching query results with the partitioned indexes is described in Figure 5. *M* is the number of partitions including empty partitions, and *K* is the number of query keywords that users issue. On the 2nd line, a two-dimensional array of "pheaders" is defined. "pheaders" holds the addresses of all partitions of query keywords. If there is at least one node indexed in the i^{th} partition of the k^{th} query keyword, then pheader[*k*][*i*] contains the physical address of the partition. If there is no node indexed in the partition, the value of pheader[*k*][*i*] is null. On the 3rd line, "load_pheaders" function loads physical addresses of all partitions. On the 5th line, the "check_empty_partitions" function returns "true" if there is an empty partition among the i^{th} partitions. If the function returns "true", we do not try to search for query results for the i^{th} partitions, and do not read all nodes in the i^{th} partitions. If all i^{th} partitions have at least one node, we try to find query results from the i^{th} partitions. On the 8th line, the "find_results" function finds query result nodes. Note that each partition of a partitioned index can be regarded as a non-partitioned index. After obtaining the nodes satisfying the second search result condition with the existing efficient methods (Guo *et al*, 2003; Xu and Papakonstantinou, 2005), we checked if obtained nodes are effective or not to *d*.

```
1  :  // M: number of partitions, K: number of query keywords

2  :  Declare pheaders [K][M];
3  :  pheaders = load_pheaders(keywords);

4  :  for  i  =  0  to  M
5  :        if  check_empty_partitions (pheaders, i)  then
6  :            continue;
7  :        else
8  :            find_results_in_partitions (pheaders, i, d);
9  :  end loop
```

Figure 5: Keyword search with partitioned indexes I

Figure 6 shows an algorithm for searching query result nodes that are effective to *d'* with the partitioned indexes on *d*. We make a virtual partitioned index on *d'* and search query result by considering only the combinations of the nodes in the same virtual partitions. *N* represents the number of physical partitions to be merged to a virtual partition. *N* is $\delta^{(d-d')}$, because as many as δ partitions are merged into a partition as the effective result depth decrease by one. On the 4th line, the "vpheaders" is a two-dimensional array containing the addresses of physical partitions to be merged. On the 7th line, the "get_partitions_to_merge" function returns the addresses of physical partitions to be merged for the i^{th} virtual partition. The "check_empty_virtual_partitions" function returns "true" if there is an empty virtual partition (in other words, if all physical partitions consisting of the virtual partition are empty partitions). The "find_results_in_virtual_partitions" function finds query result nodes with the virtual indexes.

```
1  :  // M: number of partitions, K: number of query keywords
2  :  // N: number of partitions to merge, M': number of virtual partitions

3  :  Declare pheaders[K][M];
4  :  Declare vpheaders[K][N];
5  :  pheaders = load_pheaders (keywords);

6  :  for  i  =  0  to  M'
7  :       vpheaders = get_partitions_to_merge(pheaders, i);
8  :       if  check_empty_virtual_partitions (vpheaders)  then
9  :           continue;
10 :       else
11 :           find_results_in_virtual_partitions (vpheaders, d);
12 :  End loop;
```

Figure 6: Keyword search with partitioned indexes II

4. EVALUATION

The goal of our experiment is to show how much time can be saved in searching for query results, when the effective result depth is given and partitioned indexes are used. We implemented an XML keyword search system in C++. BerkelyDB (BerkelyDB, 2005) was used for storing partitioned indexes. Each partition was constructed in the structure of the DIL. A partitioned index had a header that includes offsets of the partitions to access each of the partitions directly. The experiments were performed on a PentiumIII 993MHz PC machine with 1GB memory. We used the DBLP XML document and the INEX test collection (INEX, 2003) for the experiments.

The DBLP XML document included more than half a million titles of papers. The volume of the document was approximately 210 mega bytes. The height of the XML tree was 2. When we consider the search for paper information or author information, the effective result depth was 1. The testing query was composed of four keywords "computer", "system", "architecture", and "2002". Figure 7 shows the times for searching query results with non-partitioned indexes and partitioned indexes. The total number of partitions of the partitioned indexes is the same as the partition factor δ because the effective result depth is 1. When we used the partitioned indexes with δ=5,000 and δ=10,000, the times for searching query results were reduced by 25% and 45%. Even though we conducted the experiment with larger delta values, such as 20,000, 30,000, and so on, there was no further reduction in time.. That was because, for the query, increasing the partition factor simply increased the number of empty partitions, and did not reduce the number of the same partitions to be considered.

The INEX evaluation initiative is part of a large-scale effort to encourage research in information retrieval and digital libraries. The main goal of INEX is to promote the evaluation of content-oriented XML retrieval. They provide 125 XML documents, each of which contains papers published in IEEE journals. The volume of total documents is approximately 500 mega bytes. In our study, we assumed that the averages of the effective result depths were 2, 4, and 6, respectively. The testing queries were classified into two parts: CO (content only) and CAS (content and structure). Because CAS queries had been made for testing how efficiently search systems can handle structural queries, we selected six testing queries (called Q1 to Q6) from the CO part. The testing queries are shown in Table 1.

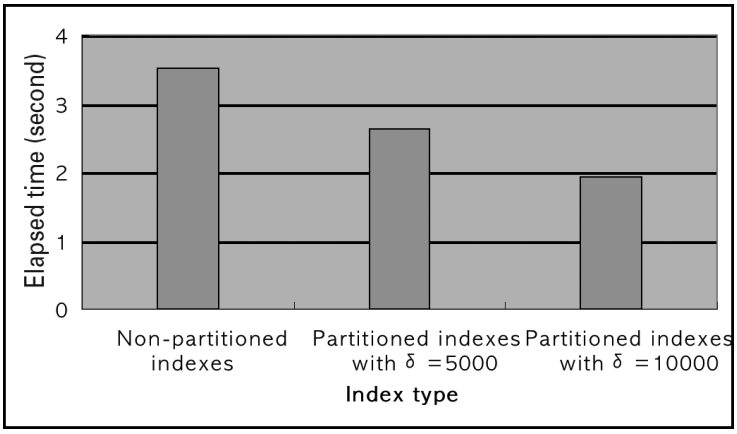


Figure 7: Elapsed times for searching query results on the DBLP XML document

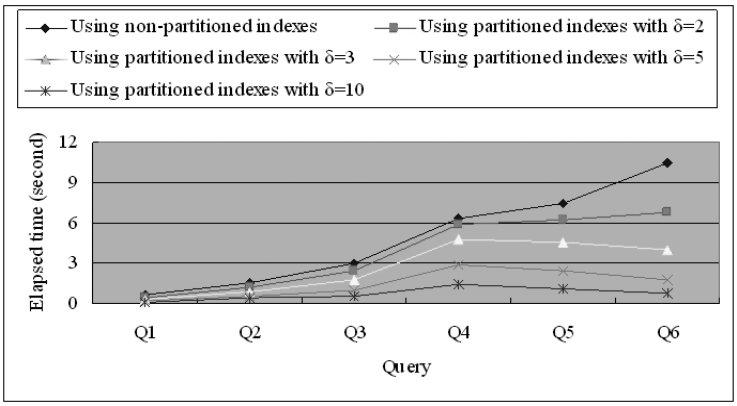
Query	Keywords	Query	Keywords
Q1	singular value decomposition	Q4	information data visualization technique hierarchy space
Q2	wireless security application	Q5	concurrency control semantic transaction management application performance benefit
Q3	recommender system agent	Q6	machine learning adaptive algorithm probabilistic model neural network support vector machine

Table 1: Selected queries

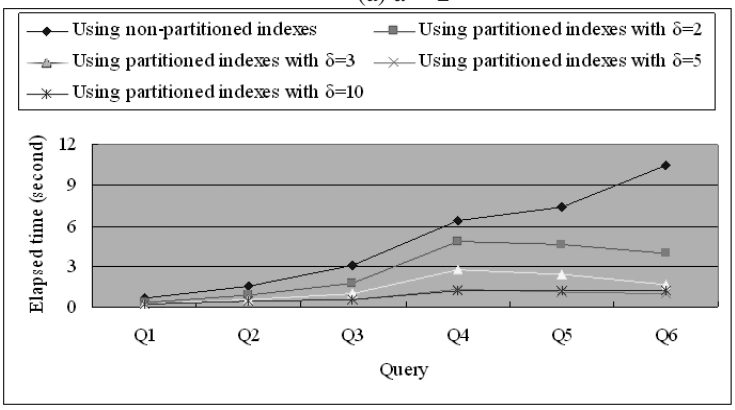
Figure 8 shows the elapsed times for searching query results when we used non-partitioned indexes and partitioned indexes with partition factors 2, 3, 5, and 10, respectively. Figure 8(a) shows the elapsed times, when the effective result depth was 2 and the partitioned indexes on depth 2 were used. The elapsed times under the partitioned index structures were less than those under the non-partitioned ones. Under the non-partitioned index structure, all combinations of the nodes in each of the indexes for the query keywords were candidates for query results. Under the partitioned index structure, only combinations of the nodes indexed in the same partitions became candidates for query results. A number of partitions of partitioned indexes were not read, which reduced the times significantly. When partitioned indexes with δ=10 were used, the elapsed times for processing queries with non-partitioned indexes were reduced by at least 77% (Q2) and by at most 92% (Q6). On average, 81% of the elapsed time was reduced. When the partitioned indexes with δ=5, δ=3, and δ=2 were used, the average elapsed times were reduced by 67%, 44%, and 22%, respectively. As partition factors increased, the elapsed times for searching query results decreased, because the probability that the nodes having no effective least common ancestor are indexed in the same partitions decreased.

Figure 8(b) shows the elapsed times for searching query results, when the effective result depth was 4 and when the partitioned indexes on depth 4 were used. Similar to the above case, searching with partitioned indexes was faster than searching with non-partitioned indexes for all the testing queries. Furthermore, searching with partitioned indexes with a large partition factor was faster than searching with the partitioned indexes with small partition factors. When we used partitioned indexes with δ=10, the elapsed times for searching query results with non-partitioned indexes were reduced by at least 70% (Q2), at most 88% (Q6), and 79% on average. When the partitioned indexes with δ=5, δ=3, and δ=2 were used, the average elapsed times, on average, were reduced by 81%, 67%, and 43%, respectively.

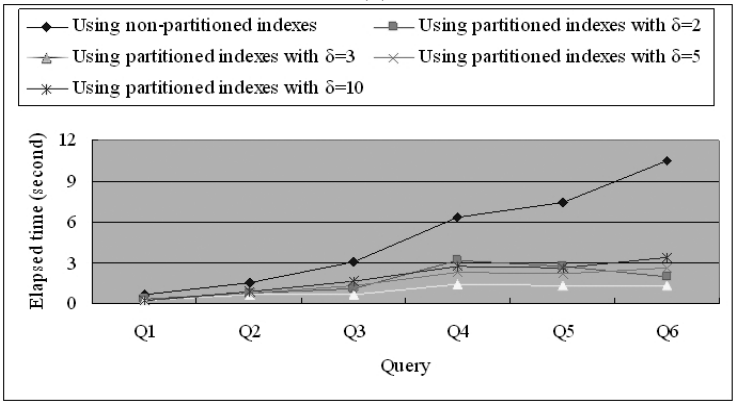
Figure 8(c) shows the elapsed times when the effective result depth was 6 and we used the partitioned indexes on depth 6. When we used partitioned indexes with δ=10, δ=5, δ=3, and δ=2, the elapsed times for searching query results with non-partitioned indexes were reduced by 59%, 66%, 75%, and 60%, respectively. Searching with the partitioned indexes with δ=10 was slower than that with the partitioned indexes with δ=5. Also, searching with the partitioned indexes with δ=5 was slower than that with the partitioned indexes with δ=3. When the effective result depth was deep (which requires a high δ value), a partition factor δ with a high value could degrade the system



(a) $d = 2$



(b) $d = 4$



(c) $d = 6$

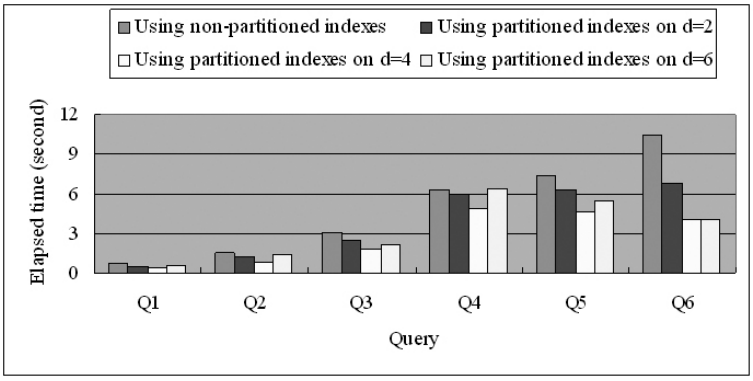
Figure 8: Elapsed times for searching query results I

performance more than that with a low value. That was because high partition factor increased the total number of partitions significantly when the effective result depth is deep. Given d=2, d=3, d=5, and d=10, the total number of partitions are 6^2=36, 6^3=216, 6^5=7,776, and 6^{10}=60,466,176,

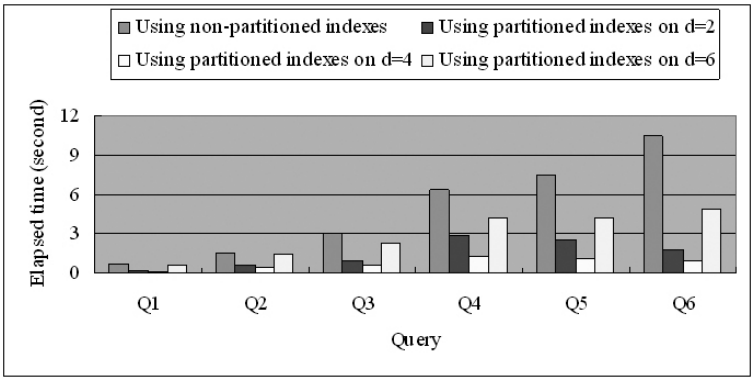
respectively. When there were too many partitions, the number of indexed nodes in a partition was too small. Therefore, the number of nodes that were read by one disk access was also too small.

Now, we show the elapsed times for searching query result nodes that is effective to *d'*, with the partitioned indexes physically constructed on *d*. We constructed non-partitioned indexes and partitioned indexes on depths 2, 4, and 6 with partition factors 2 and 5, respectively (that is, seven indexes were constructed physically). We assumed that the changed effective result depth is 4.

Figure 9(a) shows the elapsed times when we used the non-partitioned indexes and partitioned indexes on depth 2, 4, and 6, with δ=2. When the non-partitioned indexes were used, our system searched candidates for query results by considering all combinations of the nodes in each index of query keywords, and returned only the nodes that are effective to 4 among the candidates. When the partitioned indexes on depth 2 and 4 were used, the systems searched candidates for query results by considering only the combinations of the nodes belonging to the same partition. The elapsed times for searching query results with non-partitioned indexes were reduced by 22% (indexes on depth 2) and 47% (indexes on depth 4). Note that the effective result depth was 4. The number of combinations to be considered for query results was the smallest when partitioned indexes on depth 4 were used.



(a) δ = 2



(b) δ = 5

Figure 9: Elapsed times for searching query results II

While finding query result nodes that are effective to depth 4 with the partitioned indexes on depth 6, our search system virtually constructed partitioned indexes on depth 4. One virtual partition is composed of 4, i.e., $2^{(6-4)}$, partitions of the physical indexes on depth 6. Compared with the partitioned indexes on depth 4, partitioned indexes on depth 6 caused the searching times to increase by 46%, 58%, 25%, 31%, 19%, and 19% (Q1 to Q6), respectively. Nevertheless, when our system used the partitioned indexes on depth 6, the searching times with non-partitioned indexes were reduced by 23%, 10%, 29%, 0%, 26%, and 39%, respectively.

Figure 9(b) shows the searching times when we used the non-partitioned indexes and partitioned indexes on depth 2, 4, and 6, with δ=5. The searching times with the indexes on depth 6 were greater than those with partitioned indexes on depth 4 by 3.8 times, on average. However, the time for merging and searching query results was faster than searching with non-partitioned indexes. As the partition factor increases, the number of physical partitions to be merged into a virtual partition increases. Therefore, when we used the partitioned indexes with δ=5, the time for merging was more burdensome.

5. CONCLUSION

The effective result depths of XML documents are useful information. Under the conventional non-partitioned index structures, the information has not been utilized at all. We presented how to construct partitioned indexes on the effective result depths with a partition function. We also presented the specific partition function $PF_d(n)$. Search systems can obtain query results by considering only the combinations of the nodes in the same partitions of query keywords.

The times for searching query results were reduced significantly when the effective result depths were greater than 0 and partitioned indexes were used. From our experiments, we learned the following. First, as the number of partitions increased (within a limited level), the searching time was reduced more. Second, as the number of query keywords increased, the searching time increased under the non-partitioned index structure, but the time often decreased under our partitioned index structure. That was because an empty partition of one partitioned index caused the same partitions of many other indexes not to be read. Finally, when an effective result depth was changed, using the partitioned indexes virtually constructed on the depth was less efficient using the partitioned indexes physically constructed on the depth. However, using the partitioned indexes virtually constructed on the depth is more efficient than using conventional non-partitioned indexes.

ACKNOWLEDGEMENTS

This research was supported by the Ministry of Information and Communication, Korea, under the College Information Technology Research Center Support Program, grant number IITA-2005-C1090-0502-0016.

This work was supported by the Brain Korea 21 Project in 2006.

REFERENCES

ANH, V., KRESTER, O. and MOFFAT, A. (2001): Vector-space ranking with effective early termination. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieva*l: 35–42.

BERKELYDB. (2005): The BerkelyDB System. http://www.sleepycat.com.

BOTEV, C. and SHANMUGASUNDARAM, J. (2005): Context-sensitive keyword search and ranking for XML. In *Proceedings of International Workshop on the Web and Databases*: 115–120.

CARMEL, D., MAAREK, Y.S., MANDELBROD, M., MASS, Y. and SOFFER, A. (2003): Searching XML documents via XML fragments. In *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in information retrieval*: 151–158.

CLARK, J. and DEROSE, S. (1999): XML path language (XPATH) Version 1.0, http://www.w3.org/TR/1999/REC-xpath-19991116.
COHEN, S., MAMON, J., KANZA, Y. and SAGIV, Y. (2003): XSEarch: A semantic search engine for XML. In *Proceedings of 29th International Conference on Very Large Data Bases*: 45–46.
DBLP (2006): Computer Science Bibliography, http://www.informatik.uni-trier.de/~ley/db.
DEUTSCH, A., FERNANDEZ, M., FLORESCU, D., LEVY, A. and SUCIU, D. (1998): XML-QL: A query language for XML, http://www.w3.org/TR/NOTE-xml-ql/
FLORESCU, D., KOSSMANN, D. and MANOLESCU, L. (2000): Integrating keyword search into XML Query Processing. *Computer Networks* 33(1–6): 119–135.
FUHR, N. and GROBJOHANN, K. (2004): XIRQL: An XML query language based on information retrieval concepts. *ACM Transactions on Information Systems* 22(2):313–356.
GUO, L., SHAO, F., BOTEV, C. and SHANMUGASUNDARAM, J. (2003): XRANK: Ranked keyword search over XML documents. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*: 16–27.
HRITIDIS, V., PAPAKONSTANTINOU, P. and BALMIN, A. (2003): Keyword proximity search on XML graph. In *Proceedings of the 19th International Conference on Data Engineering*: 367–378.
INEX (2003): INitiative for the evaluation of XML retrieval, http://inex.is.informatik.uni-duisburg.de:2003/.
LI, Y., YU, C. and JAGADISH, H.V. (2004): Schema-free XQuery. In *Proceedings of the 30th International Conference on Very Large Data Bases*: 72–83.
MOFFAT, A. and ZOBEL, J. (1996): Self-indexing inverted files for fast text retrieval. *ACM Transactions on Database Systems* 14(4): 349–379.
ROBIE, J., LAPP, J. and SCHACH, D. (1998): XML query language (XQL), http://www.w3.org/TandS/QL/QL98/pp/xql.html.
THEOBALD, A. and WEIKUM, G. (2002): The index-based XXL search engine for querying XML data with relevance ranking. In *Proceedings of the 8th International Conference on Extending Database Technology*: 477–495.
XU, Y. and PAPAKONSTANTINOU, Y. (2005): Efficient keyword search for smallest LCAs in XML databases. In *Proceedings of the ACM SIGMOD international conference on Management of data*: 527–538.

BIOGRAPHICAL NOTES


Sung Jin Kim

Sung Jin Kim is a post doctoral research fellow in the Department of Computer Science at the University of California, Los Angeles, USA. He received a PhD degree in computer science from Soongsil University, Korea, in 2004. The title of his PhD thesis is "Efficient construction and maintenance of web databases". He was a post doctoral research fellow in the Department of Computer Engineering at Seoul National University, Korea (2004–2006). So far, he has developed a number of applications, such as a web robot, a meta-search engine, an XML keyword search system, and so on. His research interests include web databases, XML, and information retrieval.


Hyungdong Lee

Hyungdong Lee is a senior engineer at Samsung Electronics in Korea. He received a PhD degree in computer engineering from Seoul National University, Korea, in 2005. His research interests include XML data processing, knowledge discovery, information retrieval, and databases.


Hyoung-Joo Kim

Hyoung-Joo Kim is a professor in the Department of Computer Engineering at Seoul National University, Korea. He received a PhD degree in computer science from the University of Texas at Austin, USA. During his research, he has studied the object-oriented database. Currently, his research interests include XML, semantic web, and bioinformatics.