JOURNAL OF KOREA MULTIMEDIA SOCIETY, VOL. 8, NO. 6, JUNE 2005(pp. 729-743)

MPI: A Practical Index Scheme for XML Data in Object Databases

Ha-Joo Song[†]

ABSTRACT

In order to access XML data stored in object databases, an efficient index scheme is inevitable. There have been several index schemes that can be used to efficiently retrieve XML data stored in object databases, but they are all the single path indexes that support indexing along a single schema path. Hence, if a query contains an extended path which is denoted by wild character ('*'), a query processor has to examine multiple index objects, resulting in poor performance and inconsistent index management. In this paper, we propose MPI (Multi-Path Index) scheme as a new index scheme that provides the functionality of multiple path indexes more efficiently, while it uses only one index structure. The proposed scheme is easy to manage since it considers the extended path as a logically single schema path. It is also practical since it can be implemented by little modification of the B^+-tree index structure.

Keywords: XML, index, object database

1. INTRODUCTION

As XML (eXtensible Mark-up Language) is emerging as a standard format for data exchange and storage, the research on efficiently storing and retrieving XML data in object database management systems (ODBMS) or relational database management systems (RDBMSs) is becoming more prevalent[3,1,6,4,5]. Especially ODBMS are getting popular as a storage system for XML data, since their data model is similar to that of XML, and XML queries can be supported by the existing object query language (OQL) processors. XML queries, however, have some features that are not directly supported by the existing OQL processors, for instance, the use of more extensible path expressions. Following is an example of this.

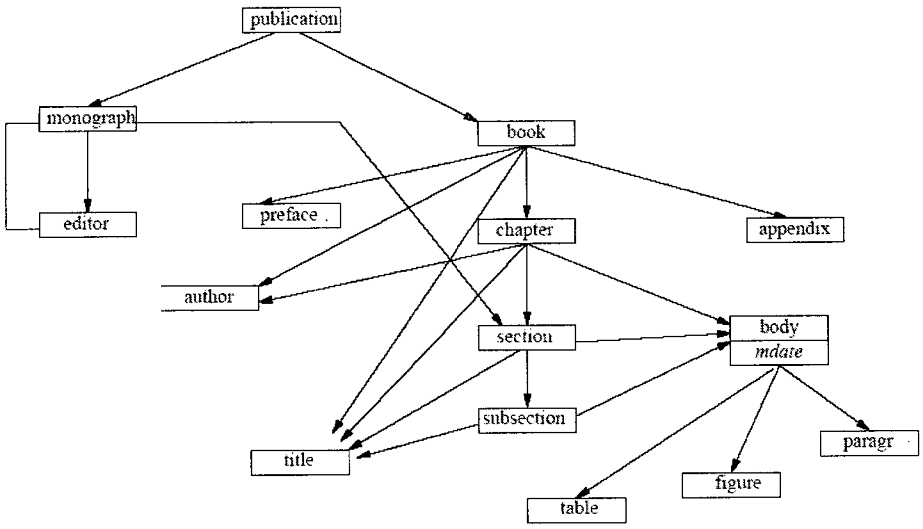
Fig. 1-(a) shows a part of an XML DTD (document type definition) which defines the structure of an XML data, and corresponds to a schema of a database. Fig. 1-(b) illustrates a part of ODB schema, mapped from the XML DTD in Fig. 1-(a). By converting the XML DTD into an ODB schema, XML data can be stored in an ODBMS. In Fig. 1-(b), rectangles and strings in italic style denote classes and attributes, respectively. Usually elements and attributes of XML DTD are mapped to classes and attributes of the ODB schema. The details about the conversion from XML DTD to ODB schema can be found in[4]. Assuming that the attribute 'mdate' denotes the date when the corresponding part of XML data has been last modified, XML data stored in an ODBMS can be queried as follows.

- XML query example 1:

Retrieve all books of which subsection body have been last modified on Oct. 1, 2004.

for $b in doc ("http://...") /publications/publication/book
let $mdate := $b/chapter/section/subsection/body/mdate["2004:10:01"]
where count($mdate) > 0
return $b

※ Corresponding Author : Ha-Joo Song, Address : (608-737) 559-1 Daeyon-dong, Nam-gu, Busan, Korea, TEL : +82-51-620-6492, FAX : +82-51-620-6450
E-mail : hajusong@pknu.ac.kr
Receipt date : Oct. 29, 2004, Approval date : Jan. 27, 2005
[†] Pukyong National University.
※ This work was supported by Pukyong National University Research Fund in 2003.



(b) ODB schema

Fig. 1. Example XML DTD and corresponding ODB schema.

```
<!DOCTYPE publication [
<!ELEMENT publication (monograph | book)>
<!ELEMENT monograph (editor+,section+)>
<!ELEMENT editor (monograph+)>
<!ELEMENT book (title,author+,preface,
  chapter+,appendix)>
<!ELEMENT chapter (title,author*,body*,section+)>
<!ELEMENT section (title,body*,subsection+)>
<!ELEMENT subsection (title,body+)>
<!ELEMENT body (paragr|figure|table)>
<!ATTLIST body mdate #CDATA>
...
```

(a) XML DTD

To process queries like the above which include path expressions, a query processor has to visit a set of objects that references one the other. This kind of an operation is termed as the object navigation (or pointer chasing). Since the query processor has to visit lots of objects during the object navigation, it incurs a lot of overheads. The overheads increase when the objects to be visited are dispersed over a disk[15]. Hence, index schemes such as nested index, path index[2], multi index, ASR(access support relation)[10], and hierarchical join indexes[15] have been proposed to improve the performance of the object navigation. Meanwhile, the following type of a query which includes a wild character '*' in its path expression is also possible to denote multiple paths concisely.

• XML query example 2:

Retrieve all publications of which the body has been last modified on Oct. 1, 2004.

for $b in doc ("http://...") /publications/publication/book
let $mdate := $b//body/mdate["2004:10:01"]*
where count($mdate) > 0
return $b

The character '*' is a wild card that can be substituted by any path or element as long as the resulting path expression complies with the given XML DTD or ODB schema. We define paths (or path expressions) with a '*' character inside as extended paths (or extended path expressions) to differentiate them from the normal single paths (or single path expressions). A clear definition of an extended path will be given in Section 2. Hence, in query example 2, the extended path b/*/body/mdate stands for a set of single paths-b/chapter/body/mdate, b/chapter/section/body/mdate and b/chapter/section/subsection/body/mdate. An extended path can also be used to denote paths that are otherwise impossible to denote. Following is an example.

• XML query example 3:

Retrieve all monographs of which the body has been last modified on Oct. 1, 2004.

for $m in doc ("http://...") /publications/publication/monograph
let $mdate := $m//body/mdate["2004:10:01"]*
where count($mdate) > 0
return $m

The extended path in above query denotes several paths like m/section/body/mdate, m/editor/monograph/section/body/mdate, m/editor/monograph/editor/monograph/section/body/mdate, and so on. Since there is a cycle in the path from monograph to editor, above extended path can denote arbitrary long paths. In this case, it is almost impossible to support indexing with the previously proposed index schemes.

We categorize the previously proposed index schemes mentioned in the above paragraphs as 'single path index scheme', since they support the indexing along a single schema path. The use of single path indexes to enhance the performance of extended paths incurs some problems. Following describes two major cases of them.

• The first problem:

It is difficult to manage the indexes, since the database administrator has to find out the individual paths of an extended path to build and to remove single path indexes for it. Therefore, it is likely that single path indexes are allocated on some of the single paths, while others are not. This results in an inconsistent index allocation. Moreover, some single path indexes can be even built on the wrong paths by mistake, resulting in the overall performance degradation caused by the management cost of the wrong indexes. The management difficulty is expected to increase as the number of single paths in an extended path increases.

• The second problem:

Even though the management of single path indexes is done automatically by the database systems, the number of indexes increases as the number of single paths consisting of an extended path does. Hence, the storage cost increases drastically, while the retrieval performance degrades. Especially, the retrieval performance for the point queries (or exact match queries) significantly degrades.

In this paper, we propose 'multi-path index (MPI) scheme' as a new index scheme for the extended path expressions. The proposed scheme provides relatively good performance over the single path indexes, and is easy to manage. The rest of the paper is organized as follows: Assumptions for the proposed index scheme and definitions of terms used in this paper are explained in Section 2. In Section 3, we introduce our new index scheme, multi-path index, for indexing extended paths. In Section 3.5, we briefly summarize related work. Section 4 presents the results of the performance evaluation. Finally, conclusions from our study and areas for future research are given in Section 5.

2. PRELIMINARIES

In this section we briefly explain the assumptions based on which it is possible to use proposed index scheme, and next describe definitions of terms used throughout this paper.

2.1 Assumptions

We assume the following characteristics of XML DTD for which the proposed scheme will be used.

First, we regard an XML DTD as a unit of name space. Hence, elements and attributes that have identical names but are defined in different DTDs are considered to be different from one another. This assumption is also applied to ODB schema converted from the XML DTD.

Second, the proposed scheme does not support indexing along a path specified by IDREF type attributes. This is due to the fact that IDREF type attributes can reference objects of all types, making it intractable to find all the possible paths in advance by examining the XML DTD or ODB schema.

2.2 Definitions

We use the term 'element' and 'class' interchangeably, since elements of XML DTDs are converted to classes of ODB schema. A 'schema path' or simply a 'path' is defined as a series of elements or attributes separated by '.' character. A path used in a query is called a 'query path'. A set of objects which references one another is termed as a 'path instance' or an 'object path'. We define traditional path expressions as single path expressions, and path expressions with '*' inside them as 'extended path expressions'. Following is a general form of an extended path expression.

$$p_f \;/\; * \;/\; p_r,$$

where p_f and p_r are called as a pre-path and a post-path, respectively and both are single path expressions without '*' inside them. Therefore, path expressions like book/chapter/section and section/subsection/title are single path expressions, while book/*/title and book/chapter/*/body/mdate are extended path expressions. book and book/chapter are the pre-paths, and title and body/mdate are the post-paths. An extended path expression may contain cycles inside them as it can be seen from publication/*/section/title.

3. MULTI-PATH INDEX SCHEME

In this section, we describe how the multi-path index is implemented. For a given extended path, a multi-path index is built in three steps:

1. The first step is to find out all paths that are compliant with the given extended path expression, and allocate a path identifier (PID) for each paths found.
2. The second step is to collect index data for all the paths found in first step and to store the result in temporary files.
3. The third step is to build the multi-path index structure.

Since the second step is processed by the query processor, we do not explain it in details, but we elaborate on

1. allocating PIDs to possible schema paths in the first step,
2. storage structure of the multi-path index, and
3. its retrieval and update methods.

3.1 Allocating Path Identifiers

For an extended path p, we define 'path extent of p' as a set of all paths covered by path p and denote it as $\mathcal{E}(p)$. The paths which constitute the $\mathcal{E}(p)$ are termed as 'member paths of p'. Algorithm 1 illustrates the algorithm that searches the path extent of a given extended path p (an input of Algorithm1) and then allocates PIDs to member paths of p. Another input to the Algorithm 1 is the XML DTD used to store XML data in the ODBMS. As path expressions can be possible between parent and its child elements or between an element and its attributes of an XML DTD, Algorithm 1 makes use of these relationships to find out all the possible member paths of the given extended path. Before we explain details of Algorithm 1, we define *head(p)* and *tail(p)* for a given extended path p as the first element and the last element (or attribute) of path p, respectively. We also define *children(e)* as the set of all child elements and attributes of an element e and a *node* means either an element or an attribute.

Algorithm 1 first initializes all variables and checks the validity of pre-path (p_f) and post-path (p_r) of path p. It then calls search_single_paths to look for all the member paths from *tail*(p_f) to *head*(p_r). search_single_paths stops if given element e is a leaf element which does not have any child. In case e has children, search_single_paths pushes e into the stack which keeps track of current search path.

Algorithm 1. Algorithm for allocating path identifiers

```
Input: XML DTD, $p$ : extended path expression
Output: $\mathcal{G}(p)$ and PIDs for member paths in $\mathcal{G}(p)$

$\mathcal{G}(p) \leftarrow \phi$, $p_f \leftarrow$ pre-path($p$), $p_r \leftarrow$ post-path($p$), $cp(p) \leftarrow \phi$;
check validity of path $p_f$ and $p_r$;
from $\leftarrow$ tail($p_f$), to $\leftarrow$ head($p_r$);
initialize stack
search_single_paths(from);
search_cycle_paths();
assign serial numbers for every member paths in $\mathcal{G}(p)$;

procedure search_single_paths($e$)
if $e$ is a leaf element then
    return;
end if
if $e$ already exists in stack then
    return;
end if
push $e$ into stack;
for all $c \in$ children($e$) do
    if $c$ = to then
        $\mathcal{G}(p) \leftarrow \mathcal{G}(p) \cup$ concatenate_path (path in stack, $c$) ;
    else if $c$ is an element then
        search_single_paths ($c$);
    end if
end for
pop stack;
end procedure

procedure search_cycle_paths()
for each path $p$ in $\mathcal{G}(p)$ do
    for each $n$ in elements($p$) do
        find_cycle_pattern($n$);
    end for
end for
pattern_generalization();
compose_cycle_pattern();
end procedure

procedure find_cycle_pattern($n$)
if $n$ is a leaf element then
    return;
end if

if $n$ already exists in the stack but not at the bottom then
    return;
else if $n$ is the bottom node in stack then
    $cp(p) \leftarrow cp(p) \cup$ concatenate (path in stack, $n$);
    return;
end if
push $n$ into the stack;
```

Algorithm 1. Continued

```
52: for all $c \in child(n)$ do
53:     if $c$ is an element then
54:         search_cycle_path ($c$);
55:     end if
56: end for
57: pop stack;
58: end procedure
59:
60: procedure pattern_generalization()
61: $cp(p) \leftarrow$ sort $cp(p)$ by the number of nodes and alphabetical order in a pattern
62: for each $s$ in $cp(p)$ in reverse order do
63:     for each $t \in$ all the paths before $s$ in cp(p) do
64:         if $s$ contains all nodes in $t$ then
65:             change the matching part of $s$ with $t$;
66:         end if
67:     end for
68: end for
69: end procedure
70:
71: procedure compose_cycle_pattern()
72: for all $s \in \mathcal{E}(p)$ do
73:     for all $c \in cp(p)$ do
74:         if $c$ has a sub-path $t$ of $s$ then
75:             compose $s$ with $c$: $(\mathcal{E}(p)) \leftarrow$ replace $t$ with $c$ from $s$;
76:         end if
77:     end for
78: end for
79: remove duplicated cycle paths from $\mathcal{E}(p)$
80: end procedure
```

Example 1 For the XML DTD in Figure 1-(a), the member paths of the extended path publication/ */mdate and their PIDs are as follows:

Then, for each child *c* of *e*, search_single_paths checks if *c* is identical with to (=*head*(p_r)). If so, search_single_paths concatenates the current path stored in the stack and *c*, and inserts the concatenated path into $\mathcal{E}(p)$. For the other cases, search single path either continues the search by recursive call or probes next child *c* of *e*.

After all the single paths of *p* have been searched out, it calls search_cycle_paths to extract cycle patterns and generate cycle paths. search_cycle_paths first calls find_cycle_pattern to get all cycles related to distinct nodes in single paths using a graph traversal algorithm, and then calls pattern_generalization to make the cycle patterns more general. Therefore, we use only the most general cycle pattern to make cycle paths. The generalized cycle patterns (*cp(c)*) are then merged to $\mathcal{E}$(p). During the merge, redundant paths that can be denoted by other expressions are removed. All the generated cycle paths are inserted into $\mathcal{E}(p)$. After all the member paths inside the given extended path have been identified, PIDs are allocated to them.

publication/book/chapter/body/mdate - PID : 1,
publication/book/chapter/section/body/mdate - PID : 2,
publication/book/chapter/section/subsection/body/mdate - PID : 3
publication/(monograph/editor)*/monograph/se-

ction/body/mdate - PID : 4
publication/(monograph/editor)*/monograph/se
ction/subsection/body/mdate - PID : 5

3.2 Storage Structure of the Multi-Path Index

Index data are stored in a multi-path index of which the structure is same as a B⁺-tree except that the leaf record structure is modified from that of a B⁺-tree. A leaf record of a multi-path index contains following information.

- *record header* - information about the record such as record length and key size
- *key value* - index key value
- *# paths* - number of member paths in OID list
- *PID* - PID of the member path
- *#OIDs* - number of OIDs of corresponding member path
- *OIDs* - list of OIDs

The PIDs used in leaf record are allocated using the algorithm described in the previous section. The structure of internal nodes of the multi-path index is the same as that of B+-tree internal nodes. Fig. 2 illustrates the structure of a leaf record in a multi-path index. Example 2 constructs an example muti-path index.

Example 2 Fig. 3 illustrates XML data stored in an object database that are compliant with the XML DTD in Fig. 1. Circles represent objects. Labels inside the circles and the underlined strings denote OIDs and the values of mdate attribute of the corresponding objects, respectively. The object o0 labeled 'publications' denotes the extent object that keeps the collection of OIDs of the instances that belong to the 'publication' class, and the objects, o1, o2, o3, and o4 are the root objects (instances) of 'publication' classes. Followings illustrate the structure and information kept in a multi-path index *i* along the path publication/*/mdate.

- Multi-path index along the path: publication/*/mdate
- P_i : *publication/*/mdate*
- D_i : *backward*

record header	key value	# paths	(PID1, #OIDs, OIDs)	···	(PIDn, #OIDs, OIDs)

Fig. 2. The leaf record structure of a multi-path index. A variant of leaf node in a B+-tree index.

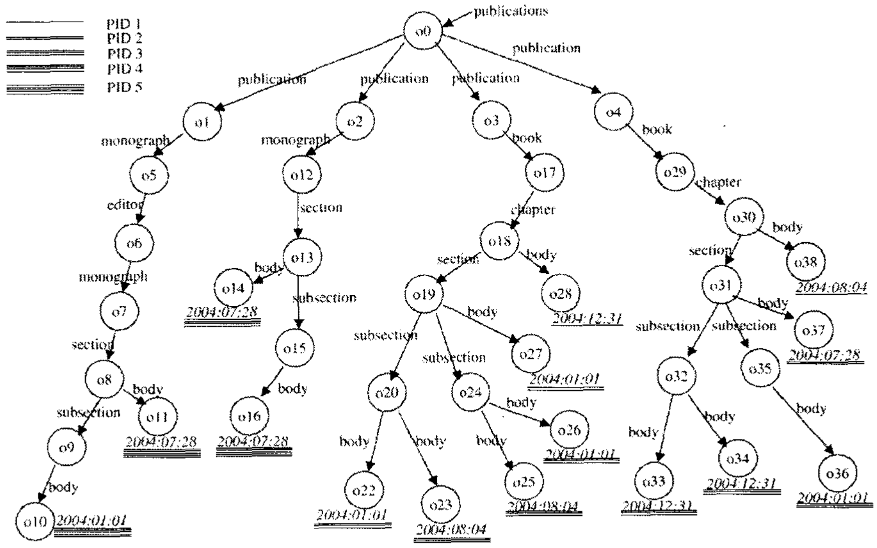


Fig. 3. An example of XML data stored in ODBMS.

- *H_i : publication*
- *T_i : mdate*
- *E_i : publication/book/chapter/body/mdate, publication/book/chapter/section/body/mdate, publication/book/chapter/section/subsection/body/mdate publication/(monograph/editor)*/monograph/section/body/mdate publication/(monograph/editor)*/monograph/section/subsection/body/mdate*
- *Book-keeping information along the member paths*
  publication/book/chapter/body/mdate - PID : 1,
  publication/book/chapter/section/body/mdate - PID : 2,
  publication/book/chapter/section/subsection/body/mdate - PID : 3,
  publication/(monograph/editor)/monograph/section/body/mdate - PID : 4,*
  publication/(monograph/editor)/monograph/section/subsection/body/mdate - PID : 5,*

After the member paths have been identified and allocated with PIDs, collect the pairs of key values and pointers (OIDs) along every member paths, and then insert them into the index. Followings illustrate how this process is done.

1. Search all the path instances of member path identified by PID 1, and then collect pairs of key values and OIDs
   ('2004:08:04', o4),
   ('2004:12:31', o3)
2. After insert above data into index, the leaf records will be as follows.
   ('2004:08:04', 1, (1, 1, o4)),
   ('2004:12:31', 1, (1, 1, o3))
3. The leaf records after we have done the same processing for path instances of PID 2
   ('2004:01:01', 1, (2, 1, o3)),
   ('2004:07:28', 1, (2, 1, o4)),
   ('2004:08:04', 1, (1, 1, o4)),
   ('2004:12:31', 1, (1, 1, o3))
4. The leaf records after we have done the same processing for path instances of PID 3
   ('2004:01:01', 2, (2, 1, o3) (3, 4, o3, o3, o3, o4)),
   ('2004:07:28', 1, (2, 1, o4)),
   ('2004:08:04', 2, (1, 1, o4) (3, 2, o3, o3)),
   ('2004:12:31', 2, (1, 1, o3) (3, 2, o4, o4))
5. The leaf records after we have done the same processing for path instances of PID 4[1]
   ('2004:01:01', 2, (2, 1, o3) (3, 4, o3, o3, o3, o4)),
   ('2004:07:28', 2, (2, 1, o4) (4, 1, o2)),
   ('2004:08:04', 2, (1, 1, o4) (3, 2, o3, o3)),
   ('2004:12:31', 2, (1, 1, o3) (3, 2, o4, o4))
6. The leaf records after we have done the same processing for path instances of PID5 (final). Fig. 4 shows physical structure of a multi-path index after inserting the following leaf records. In this figure again, we can find that the structure of internal nodes in a multi-path index is the same as that of a conventional B+-tree, while the structure of leaves are little bit different. Hence the traversal, insertion and deletion algorithms of a B+-tree can be used for a multi-path index with slight modification, which will be explained in later sections.
   ('2004:01:01', 3, (2, 1, o3) (3, 4, o3, o3, o3, o4) (5, 1, o1)),
   ('2004:07:28', 3, (2, 1, o4) (4, 1, o2) (5, 2, o1, o2)),
   ('2004:08:04', 2, (1, 1, o4) (3, 2, o3, o3)),
   ('2004:12:31', 2, (1, 1, o3) (3, 2, o4, o4))

3.3 Retrieval and Update Method of the Multi-path Index

Retrieval method of a multi-path index is different from that of a B+-tree only at the leaf records. Before starting to traverse B+-tree nodes, the path

1) Record headers have been omitted in leaf records for space reasons.

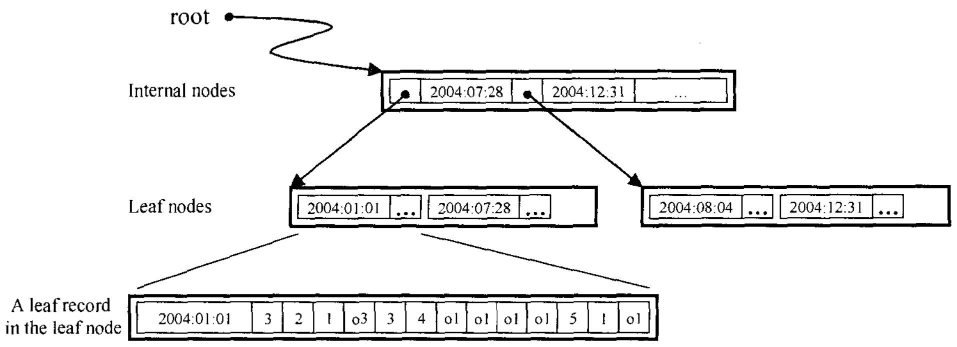


Fig. 4. The internal and leaf node structure of a multi-path index.

Algorithm 2. Algorithm for traversing a multi-path index

```
1: Input: key(key value), pids(set of PIDs)
2:
3: traverse index nodes using key in the same manner in a B+ tree traversal algorithm
4: if key not found then
5:       return;
6: end if
7:
8: r ← leaf node where the key resides
7: s ← φ
10: for all PID in r do
11:       if PID exists in pids then
12:             s ← s ∪ OIDs corresponding to this PID
13:       end if
14: end for
15: return s
```

extent of the search path is first calculated, and their PIDs are determined. Traversing from the root to the leaves is the same as in a B+-tree, but the multi-path index retrieves objects with the specified PIDs from the leaf records. Algorithm 2 shows the traversal algorithm used in a multi-path index.

Example 3 This example illustrates the query processing using the multi-path index shown in Example 2 for the following queries.

1. *for $p in doc ("http://...") /publications/ publication*
   let $mdate := $p/book/chapter/section// mdate["2004:12:31"]*
   where count($mdate) > 0 return $b
2. *for $p in doc ("http://...") /publications/ publication*
   let $mdate := $p/book/chapter//mdate ["2004:08:04"]*
   where count($mdate) > 0 return $p
3. *for $p in doc ("http://...") /publications/ publication*
   let $mdate := $p/monograph//mdate ["2004:01:01"]*
   where count($mdate) > 0 return $p

In the first query, the path extent of query path consists of publication/book/chapter/section/body/ mdate and publication/book/chapter/section/sub-section/body/mdate. Corresponding PIDs can be found using the information kept in the multi-path index, and they are 2 and 3. Traversal from root node to leaf is done using the key '2004:12:31', and

Algorithm 3. Algorithm for insertion to a multi-path index

```
1: Input: key(key value), objid(OID), pid(PID)
2: traverse index nodes using key
3: if key found then
4:       append objid in OID list of corresponding PID
5: else
6:       allocate a new leaf node, and insert key, pid, objid
7: end if
8: adjust tree structure in the same manner as in a B+ tree
```

Algorithm 4. Algorithm for deletion to a multi-path index

```
1: Input: key(key value), objid(OID), pid(PID)
2: traverse index nodes using key
3: r ← the leaf node found
4: remove (pid, objid) pair from r
5: if given pid has no OID then
6:       remove corresponding (PID, OID) pairs
7: end if
8: if given r has no OID then
9:       remove r from the index
10: end if
11: adjust tree structure in the same manner as in a B+ tree
```

then, OIDs with PID 2 or PID 3 are chosen. Therefore, the result is o4. In the second query, PID 1, PID 2, and PID 3 meet the query path. Hence, the objects with key '2004:08:04' and with PID 1, PID 2 or PID 3 are chosen, resulting in a query result of o3 and o4. The query path of the last query corresponds to PID 4 and PID 5 and the index key is '2004:01:01'. Hence the query result is o1.

The multi-path index scheme is the same as the nested index scheme[2] in that both schemes keep the key values and OIDs of index path ends. Therefore, in the case where a path instance is updated (deleted or inserted), the path instance is traversed before and after the actual update (deletion or insertion) happens. Thereby the changed object pairs are identified and reflected in the multi-path index. A detailed description on the update method can be found in[2]. Algorithm 3 and Algorithm 4 elaborate on the insertion and traversal algorithm of a multi-path index respectively.

3.4 Related Work

Index schemes such as nested index, path index, multi-index, ASR, and object skeleton[9] have been proposed to support object navigation.along a path. These schemes differ in retrieval performance, update cost and storage cost, and have their own merits and demerits. However, as they are originally designed for indexing along a single path, they are all inadequate for indexing along an extended path as described in Section 1

Uniform indexing scheme[8] allocates a unique path identifier for every class and its sub-classes that exist along a given index path, and keeps the path identifier for every pointer or object identifier (OID) in the leaf records of the index. With the use of path identifier, a uniform index can support both functionality of a class hierarchy index and a path index. The uniform scheme is similar to the proposed multi-path index scheme in that both schemes use path identifier and are very practical because both can be implemented by little modification of B+-tree structure. But, the uniform scheme differs from multi-path index scheme in that it is logically an index scheme for a single path, although it can selectively retrieve a path that consists of the specified classes along a path.

[13] proposed a group of index schemes - 1-index, 2-index and T-index - for the path indexing of a semi-structured data. When we call all these index schemes as Milo's index scheme, it can be regarded as a composite structure of groups of object identifier pairs and automata which guides the selection of target groups of OID pairs. In other words, Milo's index scheme builds an ASR[11] for every possible paths and selects a set of ASRs for a given regular path expression using the automata. The ASR used in Milo's index scheme keeps only OIDs of specified classes along a path. Hence Milo's index scheme can be viewed as a group of single indexes which still has the second problem described in Section 1.

DataGuide[7] used in Lore[14] have been proposed as an efficient index scheme for a storage system for the semistructured data. DataGuide, however, can be regarded as an extent object that collects OIDs of objects residing on the same path from a root object, and hence is a path index for the root object. Index Fabric[16] is similar to DataGuide in that it also keeps all the paths from the root objects, but is not applicable for path queries including parent-child relationships among elements. APEX [17] and D(k) - index[18] introduced an adaptive path index scheme that can dynamically update its structure according to the changes of query workloads. Graph Indexing[19] proposed a generalized indexing technique that retrieves sub-graphs quickly from a large graph data. Graph Indexing not only provides an elegant solution to the graph indexing, but also demonstrates how database indexing and query processing can benefit from data mining. Shasha et al.[20] extends the pattern-matching based algorithms for fast search in trees and graphs. It could be used for direct support of queries on the data types, or could be used as a preprocessor for join-like algorithms.

4. PERFORMANCE TEST

In this section, we compare the performance of the proposed scheme with a single path index scheme. We used the nested index scheme[2] as a single path index scheme to be compared with the proposed scheme, since both schemes keep the pairs of key values and OIDs of corresponding objects at the ends of a path, and the nested index has the best retrieval performance among the single path indexes. We use MPI and GNI to denote the proposed scheme and the nested index scheme, respectively.

4.1 Test Environment

Fig. 5 shows the path expression to be used in the test in ODB schema. In this figure, C_t and C_p are the starting class and the ending class of the extended path, respectively. There are N distinct member paths between the two classes. *kattr* denotes the index key attribute. Table 1 illustrates the parameters used in this test. Since a nested in dex can be built along a schema path, we used N nested indexes so that each nested index cover a member path. In case of multi-path index, only one index structure is used regardless of the number of member paths.

c denotes the number of objects (instances) of C_t. f represents the fan-out which means the number of C_p objects referenced by a C_t object. For

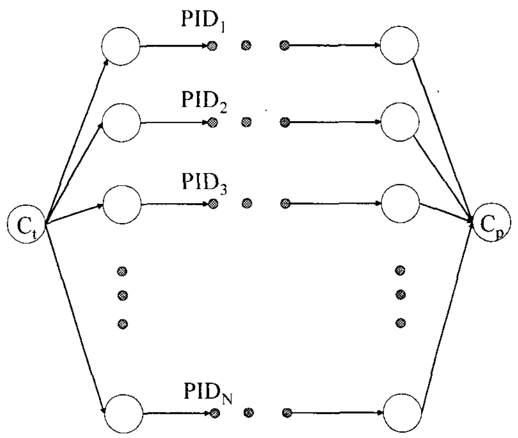


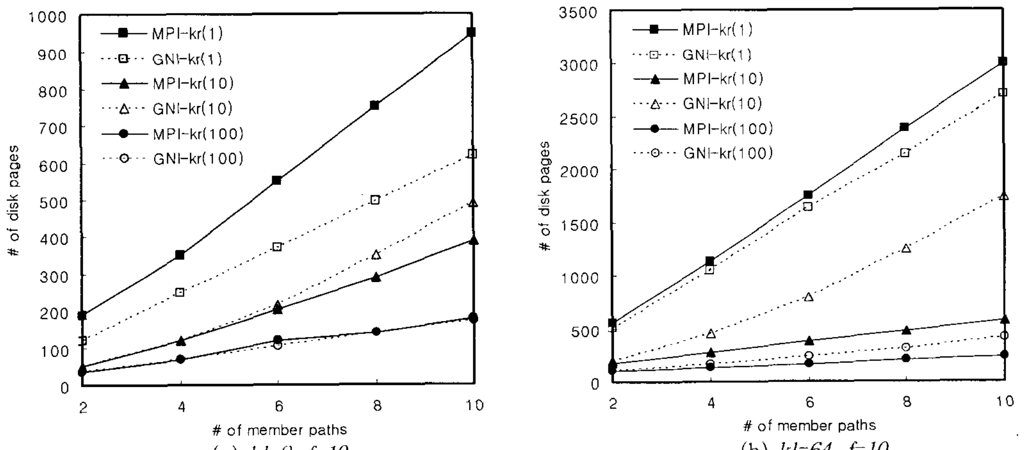
Fig. 5. ODB Schema to be used in the Test.

Table 1. Test parameters

Parameter	Test values
c	1000
kl	8, 64
N	2, 4, 6, 8, 10
f	1, 5, 10
kr	1, 10, 100
rqr	1%, 5%

instance, when we set $N = 8$ and $f = 10$, it means that a C_t object directly or indirectly references 10 distinct C_p objects along each member path, and there are 8 distinct member paths between C_t and C_p. Hence a C_t object is directly or indirectly connected to 80 C_p objects. If *c* is set to 1000 with the above configuration of *N* and *f*, there are 80,000 C_p objects. *kl* and *kr* denote the size of the index key and the number of the C_p objects with the same key value, respectively. *rqr* is used only in key range retrievals and denotes the portion of key range to be retrieved compared with the total key value range. We have compared the storage cost and the retrieval cost of MPI and GNI. The storage cost is measured by the number of disk pages used to store the indexes. Retrieval cost is categorized into two types: exact match retrieval cost and range retrieval cost. The exact match retrieval is to find an object whose attribute value exactly matches the given value, while the range retrieval is to find objects with a given key range. The retrieval cost is measured by the time to process a retrieval operation.

MPI and GNI are built as follows: We build *N* nested indexes along the *N* member paths from C_t to $C_{p:kattr}$. MPI allocates unique PIDs to every member path using the algorithm 1 as shown in Fig. 5. All indexes are configured to have its own disk segment (a contiguous set of disk pages) so that index pages of an index are clustered together. Fig. 6 illustrates the storage cost of indexes with varying the number of member paths (*N*) and size of key (*kl*). MPI uses more storage space than GNI does in case when $kr = 1$. This is due to the fact that the leaf records of MPI are larger than those of GNI and therefore MPI uses more leaf nodes than GNI does. However, the difference in storage cost is reduced as the size of key (*kl*) increases, since when *kl* is large, the size difference between the leaf records of MPI and GNI is not so big as it is when *kl* is small. When *kr* is set to 10 or 100, MPI uses less storage space than GNI does since MPI stores more pointers with the same key value in the a leaf record than GNI does, and hence uses less leaf records. Above effect is more significant when *kl* increases as shown in Fig. 6-(b). Fig. 7 shows the result of exact match retrieval cost



(a) *kl=8, f=10* (b) *kl=64, f=10*

Fig. 6. Storage cost comparison.

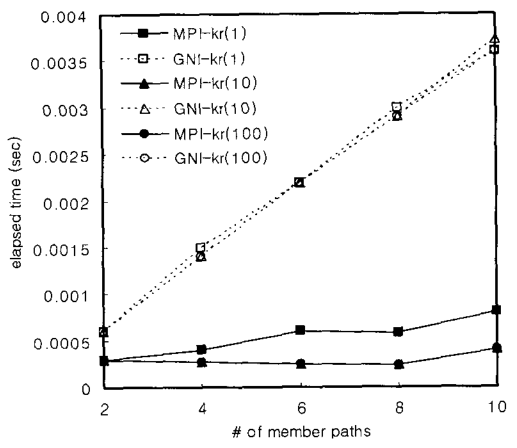


Fig. 7. Exact match retrieval cost comparison (*kl*=8, *f*=10).

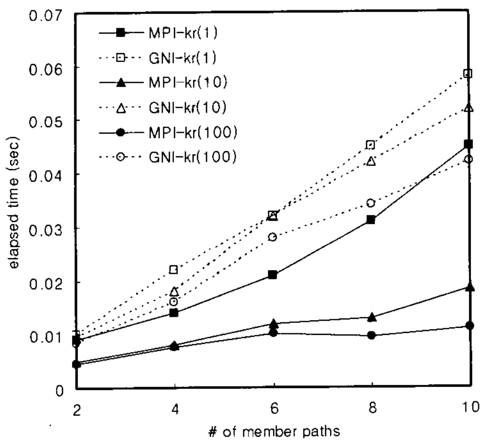
comparison when *kl* = 8 and *f* = 10. Since we have obtained the similar results in tests with other configurations, we do not present other test results.

From Fig. 7, MPI provides better exact match retrieval performance regardless of key size and replication of key values. The performance gain becomes larger as the number of member paths increases. Exact match retrieval cost is mainly determined by the height of indexes used. Since GNI has to examine *N* indexes to process a retrieval on an extended path, its exact match retrieval cost is equal to $\sum_{i}^{N} H_i$ where H_i denotes the height of the index *i*. MPI, however, uses only one index structure and its index height does not vary so much as the *N* increases (mostly 3~5). Therefore its exact match retrieval cost is almost constant regardless of *N*.

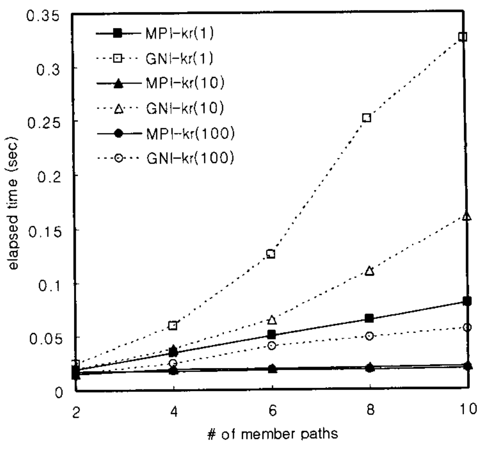
Fig. 8 shows some results of range retrieval comparison. We have observed that MPI provides better performance as more key values are replicated. It can be explained as follows: As more key values are replicated, MPI uses smaller number of leaf nodes and hence reads less number of disk pages for the same key range retrieval than GNI does. When key values are not replicated (*kr* = 1), though MPI uses more leaf nodes, the performance of MPI is better than that of GNI except for the test configuration with *kl* = 8 and *rqr* = 5%. In these cases, MPI has to visit more disk pages than GNI does, but the leaf nodes of MPI are clustered better than those of GNI so that MPI takes less time to retrieve the leaf nodes. Clustering effect, however, diminishes as the retrieval range (*rqr*) increases, since MPI has to visit a lot more disk pages than GNI does. But, in case of GNI, we have not considered the time needed to merge the retrieval results from individual indexes, which is sometimes needed for the range retrieval with the key value order. Test results shown in Fig. 8 doe not includes time for the merge. Since MPI does not need such merging step, MPI is more efficient for above cases. Hence, MPI is more useful to process index retrievals with key order.

5. CONCLUSION

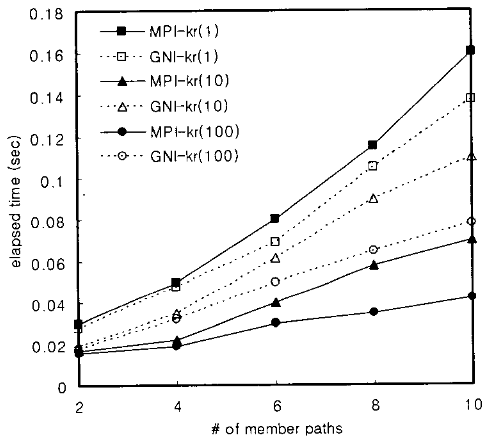
ODBMS are emerging as a prominent storage for XML data, and queries on XML data can be expressed using extended path expressions. In this paper, we propose a multi-path index scheme as a new index scheme for extended path expressions. Multi-path index scheme is implemented by allocating a unique path identifier for each member path, single path and cycle path, in a given extended path, and storing both OID and PID in the modified leaf records of a B^+-tree . Multi-path index provides a good indexing performance, since queries can be supported by looking-up only one index, not by scanning multiple indexes and merging the results from each indexes. Multi-path index is easier to manage than the approaches based on multiple single path indexes. It is also practical since it can be implemented by slightly modifying the leaf records of a B^+-tree, and thereby, does not need any new concurrency control or recovery scheme that are usually required for a new type of indexes. Using path identifiers can also be applied to existing index schemes such as path index, ASR and join index hierarchies by a minor modification on them.



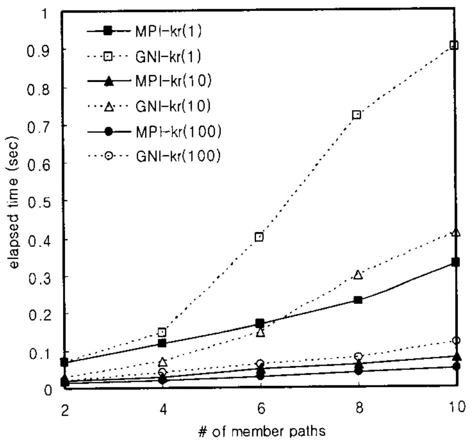
(a) *kl=8, f=10, rqr=1%*



(b) *kl=64, f=10, rqr=1%*



(c) *kl=8, f=10, rqr=5%*



(d) *kl=64, f=10, rqr=5%*

Fig. 8. Range retrieval cost comparison.

6. REFERENCES

[1] Serge Abiteboul, Peter Buneman, and Dan Suciu, *Data on The Web*, Morgan Kaufmann Publishers, San Fransico Calif., 2000.

[2] Elisa Bertino and Won Kim, "Indexing techniques for queries on nested objects," *IEEE Trans. on Knowledge and Database Eng.*, Vol. 1, No. 2, pp. 196-214, 1989.

[3] Neil Bradley, *The XML Companion*, Third Edition, Addison-Wesley, New York, 2001.

[4] Vassilis Christophides, Serge Abiteboul, Sophie Cluet, and Michel Scholl, "From structured documents to novel query facilities," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 313-324, 1994.

[5] Poet Software Corp, XML - The Foundation for the Future, http://www.poet.com/products/cms/white papers/xml/index.html, 2004.

[6] Alin Deutsch, Mary F. Fernandez, and Dan Suciu, "Storing semistructured data with stored," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 431-442, 1999.

[7] Roy Goldman and Jennifer Widom, "Dataguides: Enabling query formulation and optimization in semistructured databases," *Proceedings of the International Conference on Very Large Data Bases*, pp. 436-445, 1997.

[8] Ehud Gudes, "A uniform indexing scheme for

object-oriented databases," *Proceedings of the International Conference on Data Engineering*, pp. 238-246, 1996.

[9] Kien A. Hua, "Object skeletons: An efficient navigation structure for object-oriented database systems," *Proceedings of the International Conference on Data Engineering*, pp. 508-517, 1994.

[10] Alfons Kemper and Guido Moerkotte, "Access support in object bases," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 364-374, 1990.

[11] Alfons Kemper and Guido Moerkotte, "Access support relations: An indexing method for object bases," *Information Systems*, Vol. 17, No. 2, pp. 117-145, 1992.

[12] David Maier and Jacob Stein, "Indexing in an object-oriented dbms," *Int. Workshop Object-Oriented Database Systems*, pp. 171-182, 1986.

[13] Tova Milo and Dan Suciu, "Index strucutres for path expressions," *Proceedings of the International Conference on Database Theory*, pp. 277-295, 1999.

[14] Jason McHugh, Serge Arbiteboul, Roy Goldman, Dallan Quass, and Jennifer Widom , "Lore: A Database Management System for Semi-structured Data," *ACM SIGMOD Record*, Vol. 26, No. 3, pp. 54-66, 1997.

[15] Zhaohui Xie and Jiawei Han, "Join index hierarchies for supporting efficient navigations in object-oriented databases," *Proceedings of the International Conference on Very Large Data Bases*, pp. 522-533, 1994.

[16] B. Cooper, Neal Sample, Michael J. Franklin, Gisli R. Hjaltason, and Moshe Shadmon, "A Fast Index for Semistructured Data," *Proceedings of the International Conference on Very Large Data Bases*, pp. 341-350, 2001.

[17] Chin-Wan Chung, Jun-Ki Min, and Kyuseok Shim, "APEX: an adaptive path index for XML data," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 121-132, 2002.

[18] Q. Chen, A. Lim, and K.W. Ong. "D(k)-index: An adaptive structural summary for graph-structured data," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 134-144, 2003.

[19] Xifeng Yan, Philip S. Yu, and Jiawei Han, Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 335-346, 2004.

[20] D. Shasha, J.T-L Wang, and R. Guigno, "Algorithmics and applications of tree and graph searching," *In Proceedings of ACM Sym. Principles of Data Systems*, pp. 39-52, 2002.

Ha-Joo Song

He received his B.S., M.S., Ph.D., degree in computer engineering from Seoul National University, Seoul, Korea, in 1993, 1995, and 2001, respectively. He was a principal technical staff of LimeTV Inc. and is currently a full time lecturer in the division of electronic, computer and telecommunication. His research interests include object-oriented databases, transaction processing, and web-services.