Available online at www.sciencedirect.com
SCIENCE DIRECT®
Data & Knowledge Engineering 51 (2004) 349–368
DATA & KNOWLEDGE ENGINEERING
www.elsevier.com/locate/datak

A partition index for XML and semi-structured data ☆,☆☆

Jongik Kim *, Hyoung-Joo Kim

School of Computer Science and Engineering, Seoul National University, San 56-1, Shillim-dong, Kwanak-gu, Seoul 151-742, Republic of Korea

Received 18 September 2003; received in revised form 3 March 2004; accepted 3 June 2004
Available online 23 June 2004

Abstract

XML and other semi-structured data can be represented by a graph model. The paths in a data graph are used as a basic constructor of a query. Especially, by using patterns on paths, a user can formulate more expressive queries. Patterns in a path enlarge the search space of a data graph and current research for indexing semi-structured data focuses on reducing the search space. However, the existing indexes cannot reduce the search space when a data graph has some references.

In this paper, we introduce a partitioning technique for all paths in a data graph and an index graph which can effectively find appropriate path partitions for a path query with patterns.

Keywords: XML; Semi-structured data; Path query; Path partition

1. Introduction

As XML [3] has become an emerging standard for data representation and information exchange on the Web, managing XML data is one of the most intensely dealt research issues within the database community. XML has no fixed structure and contains structure information in itself (*self-describing*). These characteristics transform XML into an instance of semi-structured data.

☆ This work was supported in part by the Brain Korea 21 project.
☆☆ This work was supported in part by the Ministry of Information and Communications, Korea, under the Information Technology Research Center (ITRC) Support Program.
* Corresponding author.
E-mail addresses: jikim@oopsla.snu.ac.kr (J. Kim), hjk@oopsla.snu.ac.kr (H.-J. Kim).

0169-023X/$ - see front matter
doi:10.1016/j.datak.2004.06.001

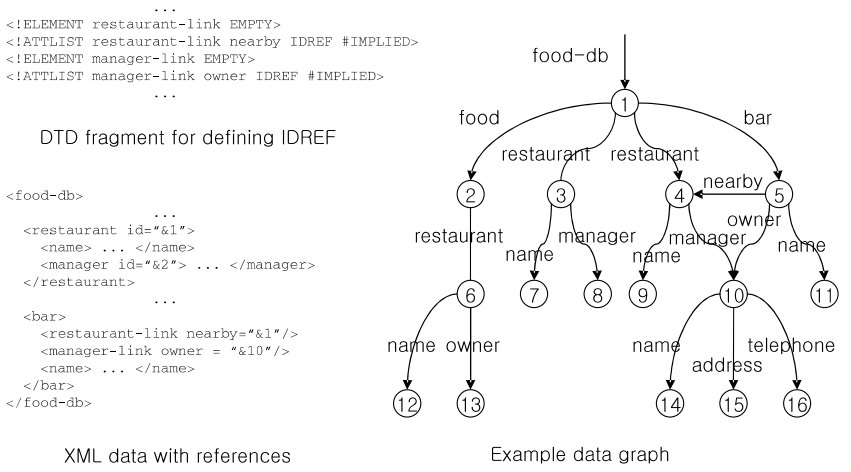


Fig. 1. An XML document and data graph.

XML and semi-structured data are modeled by a labeled graph to represent their arbitrary structures. Fig. 1 shows an XML document and the corresponding data graph. In this paper, we use XML on behalf of semi-structured data, because XML is the most representative example of semi-structured data.

Queries for XML are expressed using paths (sequences of labels) in a data graph and query processing cost depends on the overhead of traversals of a data graph. Due to the irregularity of XML, it is difficult to know the whole structure of an XML data. To query an XML data with the knowledge of its partial structure, query languages use patterns on paths. Patterns are also useful for representing arbitrary paths in a data graph (there can be infinite paths due to cycles). Regular expressions [2,5,10] and XPath expressions [6,8] are the most widely accepted patterns for paths. Although patterns are useful and necessary for formulating a query, they enlarge the search space of a data graph and degrade the performance. For example, consider a path query, _*.restaurant.owner, which finds all owners of restaurants. Here, the wild card (_) matches any labels and the Kleene star (*) means repetition of the context label._* in the query can match every path fragment that start from the root node in the data graph. So, the query should traverse every node in the data graph to produce the results.

To prevent degradation of the performance due to patterns, an index for paths is indispensable to query processing. A data graph of an XML data consists of many similar structured sub-graphs even though an XML data has no fixed structure. In other words, there exist many identical paths in a data graph. The existing index techniques (e.g. [13,15]) for XML data take advantage of these identical paths. Basically these techniques merge many identical paths in a data graph into one path. Therefore, it is expected for an index to form a graph of small size, which is called an *index graph*. A query can be evaluated through index graph traversals. It is obvious that the performance of an index is extremely dependent of the size of the index. However, we found that these indexes can grow as large as a source data graph for some reference edges in the source data graph, and cannot decrease the search space of a query with patterns. (Details are presented in Section 2.)

In this paper, we propose a technique for partitioning all paths in a data graph and an index graph which is capable of finding appropriate partitions for a path query with patterns. Unlike existing index graphs, we can adjust the size of the proposed index graph and keep it in a fixed size regardless of the structure of the source data graph. If we fix the size of the index graph into the size of the available memory for an index, there would be no I/O overhead for the index graph traversals. The proposed indexing technique is named a *partition index*. The following are the key contributions of this paper.

1. We address the problems of the existing index graphs. We show the case where the sizes of existing index graphs can increase too large to function as indexes.
2. We propose an index graph that partitions all possible paths in a data graph. Each node in the proposed index graph has a partition of paths in the source data graph. We propose a technique to adjust the size of the index graph and the size of partitions. By doing so, it is always possible to obtain a steady performance for any data with the proposed index.
3. We exploit existing techniques [9,18] to accelerate processing a partition.
4. We implemented our index technique and conducted an extensive experimental study with both real-world and synthetic data sets. Experimental results show that the partition index improves the performance much better than the existing index.

The remainder of paper is organized as follows. In Section 2, we describe basic concepts such as data models and queries and present some prior work related to our technique. In Section 3, we present the details of the partition index. In Section 4, we evaluate the performance of our index technique. We conclude the paper and provide some discussions in Section 5.

2. Background and previous work

2.1. Data model and path query

Despite some differences, XML is considered to be an instance of semi-structured data [19] and data models proposed for semi-structured data (e.g. [17]) can be used for representing XML data. Basically XML data have a tree structure, but ID/IDREF constructor in XML implements reference edges and XML must be modeled by a graph for supporting reference (non-tree) edges.

In this paper, we model XML and semi-structured data as a directed, labeled graph $G = (V, E, root, S_{oid}, C)$. Each node in V is either a leaf node or a non-leaf node. There is a special non-leaf node, *root*, that is distinguished from other nodes. $E \subseteq V \times C \times V$ indicates a set of possible edges in a data graph where C is an infinite set of labels. Every node is assigned a unique identifier by the skolem function, S_{oid} (see Fig. 1).

We refer to a sequence of labels in a data graph as a *path*. For example, the sequence of labels, (food-db, restaurant, manager), is a path in Fig. 1. For simplicity, we denote a path as a dot-separated sequence of labels (e.g. food-db.restaurant.manager). In this paper, we only consider a path starting from the root node, or the first label of a path should be the label of the root node. We say that a node, N, *satisfies* a path, P, when we traverse a data graph along P and can reach

the node N. For example, the node whose identifier is 8 satisfies food-db.restaurant.manager in Fig. 1. Also, the node with identifier 10 satisfies the path.

A path is a basic constructor of a query for XML and semi-structured data [2,5,6,8,10], where some subsequences of labels in a path can be expressed by patterns. We use regular expressions for patterns of a path. A *regular path* is a path represented using regular expressions. A regular path, R, includes labels in C and a wild card(_) and R is recursively constructed by concatenation(.), alternation (|), and repetition($*$), as follows:

$$R ::= C|\_|R.R|(R|R)|R^*$$

The wild card symbol, _, matches any label in C. We say that a node, N, *satisfies* a regular path, R, when a path from the root to N is an element of regular language defined by R. For example, _$*$.restaurant.owner which finds all owners of restaurants is a regular path in Fig. 1. Node 10 and node 13 satisfy the regular path.

We assume that every path expression ends up with a normal symbol in C. That is, a path that ends with the wild card symbol or Kleene star($*$) of the wild card is not a valid path. Consider a path _$*$.restaurant.owner._. This path query finds all children of each owner nodes and this query is equivalent to find all owners semantically. Therefore, the wild card that appears in the last of a path is useless and this proves that our assumption makes sense.

To explicitly distinguish a path from a regular path, we use a simple path. In most cases, we just use a path for both a regular path and a simple path. A path query means a regular path because a regular path is the core of a query for XML and semi-structured data.

2.2. The existing index techniques

A structural summary (or data guide) [1] for semi-structured data plays the role of a schema for semi-structured data. Because semi-structured data contains structural information in it, a structural summary can be extracted from the data. Various schema extraction techniques for semi-structured data and XML can be found in [4,12,13,16]. In [11], it is shown that the graph schemas [4] can be used as an index for path queries.

Most techniques for indexing semi-structured data are derived from schema extraction techniques. The Dataguide [13] and the 1-index [15] are most remarkable indexes. In this section, we review the Dataguide and the 1-index, address the problem with these index techniques, and provide some other variants of these techniques.

The Dataguide is an index graph created from a source data graph by interpreting a source data graph as a non-deterministic automaton and converting it into a deterministic one. Generally, converting a non-deterministic automaton into a deterministic one tends to increase the size of an automaton, but an XML data graph has many identical paths in it and such a conversion may decrease the size of a data graph. Basic strategy of constructing a Dataguide is as follows. Initially, we merge data nodes with the same label among the children of the root node into a new index node. For an index node, we merge data nodes with the same label, which are child nodes of the data nodes in the index node. When we merge a set of data nodes, if there already exists an index node that contains the set of data nodes, we simply use the index node and do not create a new index node. Fig. 2(a) and (b) shows an example of a Dataguide. For index node 1 in Fig. 2(b), data nodes 1 and 2 with the label, *restaurant*, in Fig. 2(a) are merged into a new index node $(2, 3)$

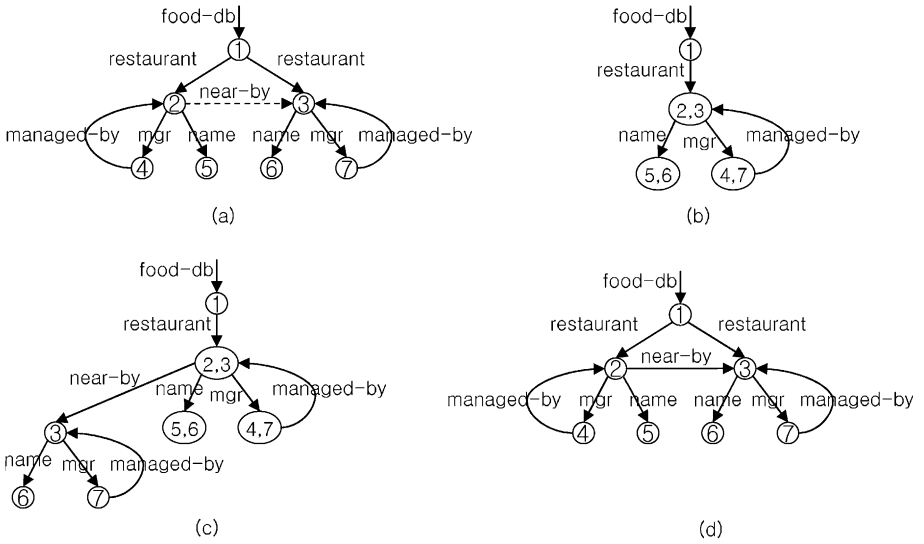


Fig. 2. A reference and index graph: (a) source data graph; (b) dataguide and 1-index without near-by edge; (c) dataguide with near by edge and (d) 1-index with near-by edge.

in Fig. 2(b). For *managed-by* edge in Fig. 2(a), because the set of data nodes, $\{2, 3\}$, is already exists in the index graph, we simply use the index node.

The 1-index also takes advantage of many identical paths in a data graph. The 1-index uses language equivalent classes in a data graph, where the language is a set of paths. For a set of paths, S, all data nodes, each of which satisfies all paths in S, are merged into a new index node. In Fig. 2(a), each node, n_1 in the subgraph of the first *restaurant* edge can find a node in the subgraph of the second *restaurant* which satisfies all paths from the root to n_1. Therefore, the two subgraphs are merged into one. to make 1-index shown in Fig. 2(b).

For graph structured indexes such as the Dataguide and the 1-index, the size of an index graph is critical to performance because indexes are also another kind of semi-structured data, that is, a query should traverse the index graph instead of the data graph. Thus, it is important that an index graph maintain a size small. The Dataguide and the 1-index assume that the size of the index graph is small considering that basically, they merge the same paths into one. But these indexes can grow to be as large as the size of a source data graph for some references.

Consider a data graph in Fig. 2(a). A reference edge like *near-by* spoils the structure of an index graph. In Fig. 2(a), there is not any language equivalent node between two subgraphs of *restaurant* edges for *near-by*. Consequently, the 1-index depicted in Fig. 2(d) has the same structure as the source data graph. In the case of the Dataguide, the two subgraphs are merged into one and then, a graph with the same structure as the sub-graph of the second *restaurant* edge should be connected to the index graph, like in Fig. 2(c). In general, for two subgraphs that have the same structure, an asymmetric reference edge from a node in one subgraph to the root of the other subgraph spoils the structure of an index graph.

As was pointed out, some references can spoil the structure of an index graph. Therefore, existing index graphs would be able to play a role of an index only under the condition that an XML data has a tree structure. However, many practical data sets have references either explicitly or implicitly, and existing techniques are unable to guarantee an efficient index for such data sets.

There are other indexes such as $A(k)$-index[14], APEX [7], and Index Fabric [9]. Each technique is a variant of either the 1-index or the Dataguide and potentially has the aforementioned problems. $A(k)$-index is a variant of the 1-index. It builds an approximate index to reduce the size of an index graph. They used an approximate index graph to retrieve a set of nodes which contains the result, and they traverse a data graph using each node in the retrieved set in order to obtain the final result. APEX an index for frequently used paths was proposed. Basically, it is 1-RO [16], which is the foundation of the Dataguide. From the information of a query workload, APEX updates its structure to be adaptive to the frequently used queries. Index Fabric is another path index which is similar to the Dataguide and the 1-index in that it indexes all paths from the root node into a sort of graph structures. However, in the case of the Index Fabric, the target of indexing is a set of path strings and not a data graph. Index Fabric indexes each path using a variant of the Patricia Trie; Index Fabric uses a multi-level index for the Partricia Trie.

3. Partition index

In Section 3.1, we will present formal definitions of the partition index, examples for a partition index and algorithms for building a partition index. In Section 3.2, we propose a technique to adjust the size of a partition index. Finally, we show how to exploit the existing techniques for the purpose of accelerating partition processing, in Section 3.3.

3.1. Partition index

To reduce the search space of a query on a database, partitioning techniques are widely used in traditional database systems. Traditional partitioning techniques use a function (e.g. hash function) to partition a set of values into several disjoint subsets. To find a value (or other attributes associated with the value), we can just hash the value with the same hash function and look up an appropriate subset. In XML data, the target of a query is a path and the query retrieve data nodes associated with the path. To apply traditional partitioning techniques to XML data, we should partition all simple paths in a data graph into disjoint subsets with a hash function. To find a simple path (or data nodes associated with the simple path), we hash the simple path with the same hash function and look up a subset associated with the hash value. However, for a regular path, the hash function used for partitioning paths cannot find an appropriate subset. For example, consider a hash function divides the number of characters in a path by 10 and returns the remainder. For a simple path, restaurant.owner.name, the hash function will return 9 (dots are ignored). We can find the simple path in the 9th subset. However, in case of a regular path, restaurant._*.name, the hash function will return 6 and we cannot find the appropriate subset. Obviously, there can be many subsets that contains paths matched with restaurant._*.name, and the traditional hash function cannot find all appropriate subsets.

In this section, we propose a partitioning technique for paths in a data graph and an index graph named a *partition index* that make it possible to find appropriate partitions for a regular path. First, we describe an initial partition index (IPI). Next, we present some properties of the index to show the correctness of our index technique. Finally, we complete our partition index by modifying IPI at the end of this section.

We begin with the formal definitions for IPI, demonstrate IPI with an example, and provide some properties of IPI. Definitions 1 and 2 are useful definitions for describing the partition index. Definitions 3 and 4 define IPI.

Definition 1. A *label equivalent set* is a set of nodes that have the same label on an incoming edge. We denote a label equivalent set for a label, L as L_{eq}. The outgoing set of a label equivalent set, a L_{eq}, is a set of labels, which are labeled on outgoing edges of each node in L_{eq}.

For example, the label equivalent set for *employee* is {1,2} in Fig. 3(a), and the outgoing set is {*name*, *age*, *supervisor*}. A node in a data graph can participate in more than one label equivalent sets, because it can have more than one incoming edge that has distinct labels. For example, node 1 in Fig. 3(a) is an element of both $employee_{eq}$ and $supervisor_{eq}$.

Definition 2. A *path partition* is a set of simple paths (not regular paths). A path in a partition is represented in a character string. Each path in a partition is associated with data nodes that satisfy the path.

An IPI is made from a data graph, and each node of an IPI keeps a partition for paths in a data graph. We do not distinguish a node from the partition in the node and treat as the same thing. The formal definitions of the IPI are as follows.

Definition 3. A node of an IPI is defined with respect to each distinct label in a data graph. For a label, L, in a data graph, let P be a partition of paths whose last label is L. A node of an IPI for L consists of (L, P, O), where O is the outgoing set of L_{eq}. We call L the *identifying label* of the index node.

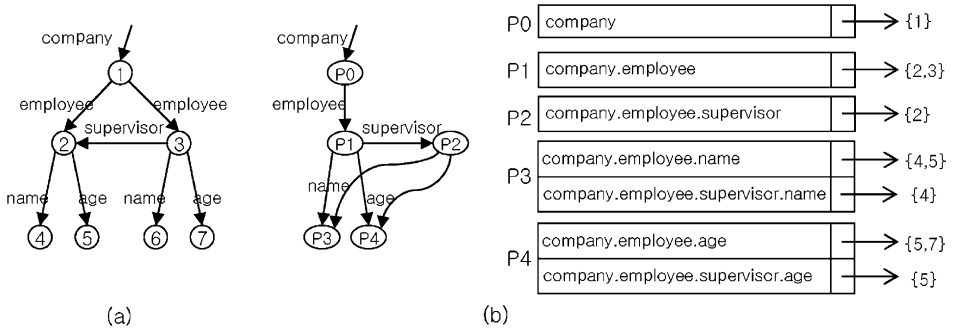


Fig. 3. Example of an initial partition index: (a) source data graph and (b) partition index and path partitions.

Definition 4. Nodes in an IPI are connected to each other as follows. For an index node, N, let the identifying label of N be L. N has an incoming edge from every node in an IPI who has L in its outgoing-set. All incoming edges of an index node are labeled by the identifying label of that node.

Fig. 3 shows an example of an IPI. There are six distinct labels in the data graph in Fig. 3(a) and there are six index nodes in the IPI corresponding to the distinct labels in Fig. 3(b). For the index node P1, the identifying label of P1 is *employee* and the label equivalent set of *employee* is $\{1,2\}$ and the outgoing set of P1 contains all outgoing edges of node 1 and 2, which is {*name*, *edge*, *supervisor*}. According to Definition 4, P1 is connected with the three index nodes: P2, P3, and P4, where identifying labels of which are *name*, *edge*, and *supervisor*, respectively. Each node in the IPI is associated with a path partition, where paths in the partition end up with the identifying label of the index node. Fig. 3(b) shows each path partition associated with each index node. All nodes and edges of the IPI in Fig. 3 is created just as Definitions 3 and 4.

The following lemma and claims describe the properties of the IPI and prove that the IPI is a correct index for a data graph. These properties can be easily derived from the definitions of an IPI.

Lemma 1. *Each node in an IPI has a different identifying label.*

Proof. Assume that two nodes, (L, P_1, O_1) and (L, P_2, O_2), exists, which have the same identifying label. And then, P_1 and P_2 must be the same set according to Definition 3, and O_1 and O_2 must be the same set according to the Definition 2. So the assumption fails. □

By Lemma 1 and Definition 3, it is easily seen that the size of an IPI is always the same as the number of distinct labels in a data graph.

Property 1. *Different partitions in an IPI are disjoint.*

Property 2. *For each path*, path, *in a data graph*, *there exist a partition*, partition, *such that* partition *contains* path.

Claim 1. *An IPI creates partitions that satisfy the properties described above.*

Proof. Let the partition of two node, N_1 and N_2, in an IPI be P_1 and P_2. By Lemma 1, N_1 and N_2 have distinct labels, L_1 and L_2. By the Definition 3, the last label of all paths in P_1 is L_1 and the last label of all paths in P_2 is L_2. This means that there cannot be any intersection between P_1 and P_2. For any path, P, in a data graph, let the last label of P be L, and let a node that P points to be N. By Definition 3, an index node exists whose identifying label is L, and P is stored in the partition of this index node. □

Claim 2. *If a path exists in a data graph*, *then the path also exists in the IPI graph.*

Proof. By Definition 3, paths of length 1 in a data graph continue to exist in the IPI. Assume that paths of length k in a data graph also exist in the IPI. Consider a path, P, of length k in a data graph. Let the last label of P be L, and there exist an index node, N, whose identifying label is L by Definition 3. Following this assumption, there would exist a path, P, from the root node of the IPI to the node, N. For all nodes in the data graph that satisfy path P, the outgoing set of N includes all outgoing labels of the nodes by Definition 3. Therefore, by Definition 4, all paths of length $k+1$ which has a prefix, P in a data graph would also exist in the IPI graph. □

Algorithm 1. Building an initial partition index

```
1: // Input: root, the root node of a source data graph
2: path: global stack of labels
3: targetHash: global hash table, to map a label to an index node
4:
5: buildIPI(root){
6:    index_root = createIndexNode( );
7:    index_root.label = root.label;
8:    path.pushLabel(index_root.label);
9:    append (path, {root}) to the partition of index_root
10:   recursiveMake(index_root, {root});
11: }
12:
13: recursiveMake(node, extent){
14:   label_set = ∅;
15:   foreach object o in extent do
16:      label_set = label_set ∪ {l|l is an outgoing label(edge)from o};
17:   foreach distinct label l in label_set do{
18:      nxt_node = targetHash.Lookup(l);
19:      if(nxt_node = = nil){
20:         nxt_node = createIndexNode( );
21:         nxt_node.label = l;
22:         targetHash.Insert(l, nxt_node);
23:      }
24:      make nxt_node be a child of node;
25:      eset1 = a set of objects reached from the objects in extent along l;
26:      eset2 = all objects in the partition of nxt_node;
27:      path.pushLabel(l);
28:      append (path, eset1) to the partition of nxt_node;
29:      recursiveMake(nxt_node, eset1 − eset2);
30:      path.popLabel( );
31:   }
32: }
```

Algorithm 1 is an algorithm for building an IPI. We traverse data graph in the depth first fashion. We, first, visit the root node of a data graph and create the root node of the IPI in the

function, *buildIPI*. Each node in an IPI keeps a partition. A partition has a set of paths having the same last label. The partition of an index node is represented as a set of the pairs, (path, extent), where the extent is (a pointer to) a set of nodes that satisfy the path. The union of all extents in a partition is a label equivalent set for the identifying label of the partition.

Each call of the *recursiveMake* function creates an index node if necessary, and makes an edge between two index nodes. The first parameter of *recursiveMake* is the currently visited index node and the second is the currently visited data nodes. We group a set of nodes, which can be reached by traversing all outgoing edges of each node in *extent*, by distinct labels. For each distinct label, *l*, if an index node, *nxt_node*, whose identifying label is *l* exists, we add an edge from *node* to *next_node*. Otherwise, we create a new index node for the label *l*, and then add an edge from *node* to the newly created node. In line 28 of Algorithm 1, if the path, *path*, already exists in the partition of *nxt_node*, we simply append each element in $eset_1$ into the set of nodes corresponding to *path*.

An IPI is similar to 1-Representative Object (1-RO) [16], but an IPI has a partition for paths in each node. Given an IPI index graph with a partition in each index node, we can process a regular path query as follows. We traverse the index graph of an IPI to find appropriate a partition for a regular path query. For each partition that satisfies the regular path, we investigate paths in the partition and output the nodes associated with paths that satisfy the regular path query.

We have two different issues, which are related. One is that the index graph of an IPI is not necessary. That is, by looking up the last label of the regular path, we can find appropriate partitions directly, and it is not necessary to traverse an IPI. For example, consider a query, _*.supervisor._*.name in Fig. 3. Because the last label of the path query is *name*, we can directly find *P*3 and verify if each path in the partition satisfies the regular path. The other is that paths in a data graph can be skewed. That is, some partitions may have a large amount of paths while others have small number of paths. For example, the partition whose identifying label is *name* may a large amount of paths in it because there can be many names in a database; name of employee, name of supervisor, name of company, and so on. For the query, _*.supervisor._*.name, we should check many paths and it could be very inefficient.

To prevent degradation of performance for processing a big partition, we should divide a big partition into two or more partitions. When a partition is divided into two partitions, we cannot determine it only with the last label of a regular path query which one of them satisfies the regular path query. Splitting a partition is extremely dependent of the index graph of an IPI, because an index node represents a partition and a partition split causes a node split of the index graph. In the case of splitting partitions, we should exploit the index graph to find appropriate partitions. In the following, we discuss partition splitting of an IPI and provide how to split nodes in an IPI. Actually, an initial partition index is constructed for splitting partitions, which cause the structure of the IPI graph to be changed. A partition index is the index made from an IPI by splitting index nodes (or partitions).

To determine if partitions in an IPI should be split, we can use the deviation of sizes of partitions, where the size of a partition means the number of paths in the partition. If the number of partitions is desirable and the deviation of sizes of partitions is low, an IPI is definitely useful because it can respond to a query, directly. However if the deviation of sizes of partitions is high, the processing cost of some partitions would be very expensive. To guarantee the steady per-

formance of the partition index, we can adjust the size of partitions in an IPI by splitting both the partitions and the nodes containing the partitions to be split. We do not distinguish a split of a partition from a split of a node because if a partition is split then a node corresponding to the partition should be split. The following definitions are useful for describing the node split.

Definition 5. For a label, L, in a data graph, $S(L)$ denotes the number of paths whose last label is L and $P(L)$ denotes the partition whose identifying label is L. For a sequence of labels, L_S, $S(L_S)$ denotes the number of paths that end up with the sequence.

Definition 6. For a path of length k, $l_1.l_2.\ldots.l_k$, *anchor*(m) denotes the label, l_{k-m} $(1 \leqslant m \leqslant k)$. In particular, *anchor*(1) (the label, l_{k-1}) is called *anchor* or *anchor label* of the path. For an anchor label, a, in a partition, the size of a means the number of paths in the partition whose anchor label is a.

Suppose there are n distinct labels in a data graph and let them be $L_1, \ldots, L_n$. Then the number of partitions in the IPI is n and the size of partitions would be denoted as $S(L_1), \ldots, S(L_n)$. Let the mean value of $S(L_1), \ldots, (L_n)$ be E. Then each partition (node) in the set of partitions, $S = \{P(L_m) \,|\, E - S(L_m) > \delta, 1 \leqslant m \leqslant n\}$, would be the target for splitting. We can adjust the threshold value (δ) to determine number of nodes to be split. To minimize the deviation of the sizes of partitions, we can perform node split procedure repeatedly. We use *split*(n) to refer to splitting a partition index n times.

When a node is split, the incoming edges of the node must be distributed to newly created nodes. Fig. 4(a) shows a fraction of an IPI. Node t in Fig. 4(a) is split to be the two nodes, t_1 and t_2, in Fig. 4(d). It is important to make the total number of incoming edges of t_1 and t_2 be the same as the number of incoming edges of node t. Otherwise, a query could traverse more edges in an index graph and retrieve more partitions. Fig. 4(b) shows the partition for node t in Fig. 4(a). If we split a partition across paths with the same anchor label like *restaurant* in Fig. 4(b), the outgoing edge from the node whose identifying label *restaurant* is distributed both of two split like the dashed edge in Fig. 4(d). Therefore, we must split a partition not to divide any paths with the same anchor into different partitions. There is another thing to be considered when a partition is split. We must split a partition such that the difference of the size between two split partitions is minimized. Obviously, this is why we split a partition. To do this, we use a set, N, of the number of paths per each anchor, like in Fig. 4(c). It is an NP problem to divide a set of numbers, N, into two

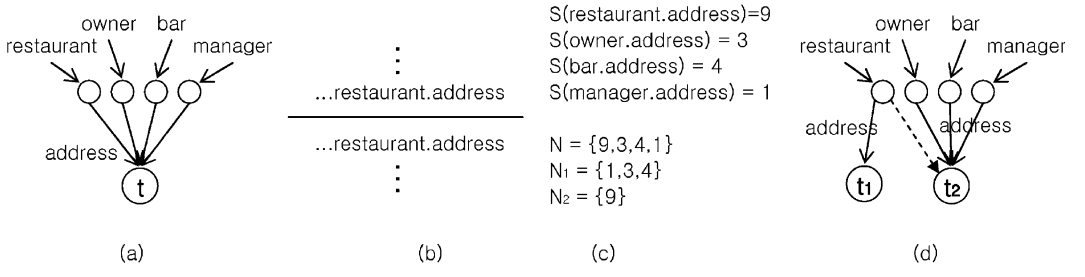


Fig. 4. Incoming edge distribution: (a) a fraction of IP; (b) partition of node t; (c) number of paths per each anchor and (d) incoimg edge distribution.

sets, N_1 and N_2, such that $N_1 \cup N_2 = N$, $N_1 \cap N_2 = \emptyset$, and $abs(|N_1| - |N_2|)$ is minimized, where $|S|$ denotes the sum of all elements in a set, S, and $abs(n)$ denotes the absolute value of a number, n. We use an approximate algorithm to split a set, N, of numbers of paths per each anchor. We sort N in descending order and then add the elements from the first element of the sorted set to the point where the sum is nearest to the half of $|N|$. Then, we divide the set into two sets at the point.

When a node is split, the outgoing edges of the node are also distributed to the newly created nodes. For two split nodes, t_1 and t_2, let a_1 be the set of anchor labels of paths in the partition of t_1 and let a_2 be the set of anchor labels of paths in the partition of t_2. Now, we can distribute the outgoing edges of t to t_1 and t_2 as follows. For a child, c, of t, if there exist a path, p, in the partition of c such that anchor(2) of $p \in a_1$, then we can add an edge from t_1 to c. Likewise, if there exist a path, p, in the partition of c such that anchor(2) of $p \in a_1$, then we can add an edge from t_2 to c. For example, the set of anchor labels of paths in the partition of t_1 is {restaurant} and the set of anchor labels of paths in the partition of t_2 is {owner, bar, manager} in Fig. 5. In Fig. 5(b), the bold and underlined labels are anchor(2) of the paths. Because the partition of c_1 does not have any path whose anchor(2) is in {owner, bar, manager}, we do not add an edge from t_2 to c_1 in the figure. Likewise, we do not add an edge from t_1 to c_3, because the partition c_3 does not have any path whose anchor(2) is *restaurant* in the figure. Obviously, a partition index after the node split still preserves the properties of an IPI.

Algorithm 2 is an algorithm for the node split. This algorithm consists of three logical blocks. The first block is splitting the partition of an index node into two partitions. For this purpose, we retrieve distinct anchors of paths in the partition of the index node to be split. We split the set of anchors into two sets using the aforementioned approximate technique. From line 5 to line 11 of Algorithm 2 shows this approximate technique. Based on the two anchor sets, the first *foreach* loop divides paths in the partition of the node to be split into two partitions. The second block is distributing the incoming edges of the node to be split. The second *foreach* loop implements the incoming edge distribution. As is seen in the algorithm, an incoming edge is distributed only one split node, and not both. The third block is distributing the outgoing edges of the node to be split. The third *foreach* loop distributes the outgoing edges. Note that an outgoing edge can be distributed both of the split nodes.

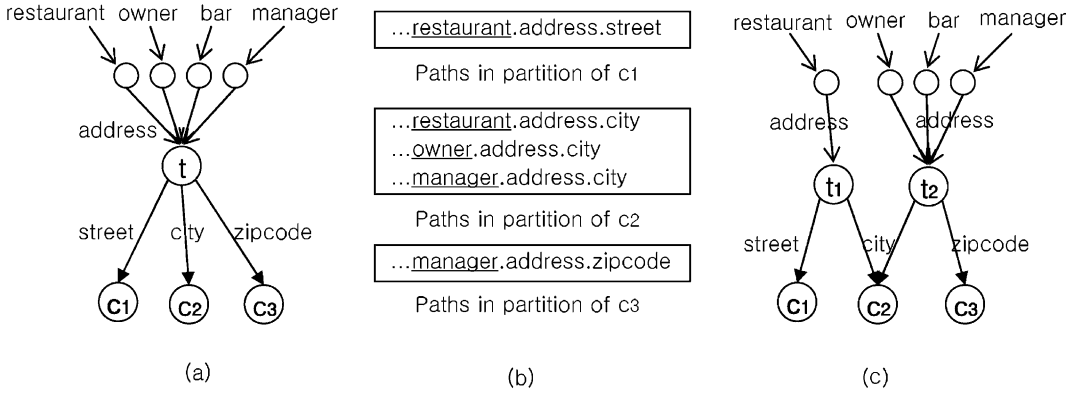


Fig. 5. Outgoing edge distribution: (a) a fraction of IP; (b) partition of child nodes; (c) outgoing edge distribution.

Algorithm 2. Split a node (a partition) of a partition index

```
1: splitNode(IndexNode node){
2:    a_set = set of anchor labels of paths in node.partition;
3:    sort a_set in the ascending order of sizes of anchors;
4:
5:    size = 0;
6:    a1 = a2 = ∅;
7:    foreach anchor a in a_set do{
8:       size += size of a;
9:       if(size ⩽ node.partition.size) insert a into a1;
10;      else insert a into a2;
11:   }
12:
13:   node1 = createIndexNode( );
14:   node2 = createIndexNode( );
15:
16:   foreach path p in node.partition do{
17:      anchor = the anchor of p;
18:      if(anchor ∈ a1) insert p into node1.partition;
19:      else insert p into node2.partition;
20:   }
21:
22:   foreach parent p of node do{
23:      remove edge from p to node;
24:      if(p.label ∈ a1) add an edge from p to node1;
25:      else add an edge from p to node1;
26:   }
27:
28:   foreach child c of node do{
29:      if(there exists a path in c.partition whose anchor(2) ∈ a1)
30:         add an edge from node1 to c
31:      if(there exists a path in c.partition whose anchor(2) ∈ a2)
32:         add an edge from node2 to c
33:   }
34:   delete node from the index;
35: }
```

3.2. Adjust the size of a partition index

In this section, we present a technique to adjust the size of a partition index. The size of a partition index is determined by the number of distinct labels in a data graph and the number of split nodes in the IPI. Here, we adjust the size of a partition index by modifying the number of distinct labels.

Definition 7. The function, $m^k(l)$ maps a label, l into a number between 1 and k.

We can adjust the size of a partition index using the function, $m^k(l)$ as follows:

1. Make an index graph from a data graph using Algorithm 1.
2. Convert each label in the index graph using $m^k(l)$.
3. Merge the nodes with the same label in the converted index graph.
4. Split nodes of the converted index graph using Algorithm 2.

Let us consider the size of a partition index. In step 3 of the procedure described above, the size of the index graph is k or less. We can determine the number of nodes to be split by modifying the threshold value, δ, and if we determine the number of nodes to be split as $k/2$, then the size of the index graph will be $k + k/2$. Let the size of the available memory for an index be M. We can always load an index graph into the memory by setting k to $2M/3$, and consequently, there is no I/O overhead to traverse the index graph.

To process a query, each label in a query must be converted using $m^k(l)$ and then, the index graph must be traversed to find appropriate partitions. We can design $m^k(l)$ as a translation table to minimize the deviation for the index of size k. If there are too many distinct labels to look up the translation table, we can use a hash function for $m^k(l)$.

3.3. Partition processing

To obtain the results of a query using a partition index, we must follow the following three steps. First, translate each label in a query to traverse the index graph. Second, traverse the index graph to find appropriate partitions. Third, process the partitions to obtain the final results of the query. In this section, we will discuss the third topic, *partition processing*.

A regular path can be represented by an automaton. We say a *query automaton* to refer to a regular path in a query that is represented in an automaton. For example, consider a query, _*.(restaurant| bar)._*.address. The query can be represented by an automaton depicted in Fig. 6. A simple way to process a partition is to evaluate each path in the partition using the query automaton. For example, food-db.restaurant.location.address can be accepted in the automaton in Fig. 6. The set of state transitions for the query are $q1 \rightarrow q1 \rightarrow q2 \rightarrow q2 \rightarrow q3$, where the first $q1$ denotes the initial state. Therefore, the path is an answer of the regular path query. food-db.bar.name cannot be an answer of the query, because this path cannot reach the final state, $q3$, of the query automaton. However, some partitions may exist whose size is too large to process a

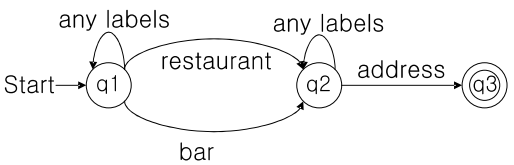


Fig. 6. A query automaton.

query efficiently, even though we split nodes in a partition index and adjust the size of a partition index. To process such partitions, we can exploit two existing techniques.

The first is the signature technique [18]. We can make a signature for each path in a partition using the technique in [18] and filter a partition using signatures to reduce the size of the partition to be processed by a query automaton. Of course, the signatures for paths in a partition are stored with the partition when the partition is created. If paths in a partition are stored sequentially, then the signature technique would not likely reduce the number of I/Os. As noted in [13], however, it is difficult to guarantee clustering, so we believe the signature technique can reduce the cost of processing a partition.

The second technique to be exploited is the Index Fabric [9]. We store paths in each partition and the target of the Index Fabric is a set of paths, so we can exploit the Index Fabric for indexing a partition whose size is too large to use a query automaton or to exploit the signature technique.

We can mix the techniques explained above in a partition index according to the size of each partition.

4. Experimental results

In this section, we will provide the experimental results of our index technique. We begin with the data sets and the platform used for the experiments. Next, we briefly describe the path queries. We compared our index technique with the strong Dataguide. Our main target is the performance on reference edges which can spoil structural summaries as described in Section 4.1. After comparing with the Dataguide, we investigate the performance of the partition processing, varying the size of the index by splitting nodes. For a node split in the experiments, we use zero as δ value (see Section 4.2).

4.1. Platform and data sets

We implemented a simple object repository system for the experiments. A page of the system has a 4Kbyte size. And the system has 50 memory buffers for pages. We stored data used in the experiments into our system and nothing is resident in-memory except system memory buffers. We did not use other optimization techniques such as prefetching and clustering. Here, the performance measure is the number of I/Os to load pages into memory buffers.

We used two data sets. One is XMark [21] data sets and the other is the Internet movie database (IMDB) [20]. XMark data set is a synthetic data set for auction data, and it has many reference edges which spoil the structure of the strong Dataguide. Using the program provided in [21], we could produce an auction data set of 10M size. IMDB data set consists of many small files for movies and actors. We merged these files into one large semi-structured data, and we connected reference edges between movie elements and actor elements using the titles of movies and the names of actors. We removed all dangling pointers in IMDB data. IMDB has lots of cycles in it; each actor has a cycle, which has the path, *Actor._*.Movie._*.Actor*. Though IMDB has many cycles, the Dataguide for IMDB is constructed in a small size. This is because the reference edges of IMDB are too simple and regular, that is, every actor is pointed by some movies with the label, "Actor", and every movies is pointed by some actors with the label, "Movie".

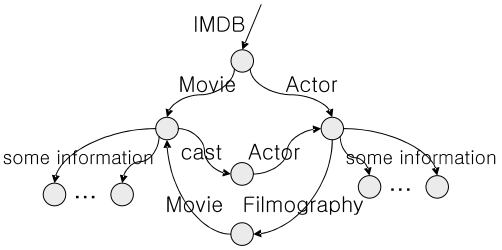


Fig. 7. Dataguide for IMDB.

Fig. 7 shows the Dataguide for IMDB. We chose IMDB data set to show that our index technique keeps a steady performance for data which are suitable for the existing indexes.

4.2. Path queries

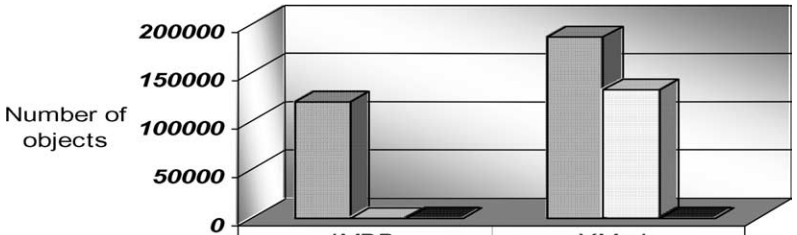
We divide path queries into two groups. One group contains simple path expressions which do not use regular expressions, and the other contains regular path expressions which use only '_*'. We use '_*' for regular paths, because it enlarges the search space of an index. We generate simple path expressions by performing random walks on a data graph. By replacing a sequence of a few labels in a simple path with '_*', we can make a regular path. The length of a path is randomly decided. The length of simple path is from 5 to 9, and the length of a regular path is from 3 to 9.

4.3. The sizes of data and indexes

Fig. 8 shows sizes of data sets, strong Dataguide, and partition indexes. We use partitions indexes with *split*(1) in the experiments.

Both the Dataguide and the partition index are created in small sizes for the IMDB data set. For the XMark data set, however, the size of the Dataguide is about 70% of the data graph.

We can see that the size of the partition index is not influenced by the structure (e.g. reference structure) of a data graph from Fig. 8. As was pointed out, the size of a partition index is



	IMDB	XMark
Source data	119797	185905
Dataguide	37	131461
Partition index	20	92

Fig. 8. Sizes of data and indexes.

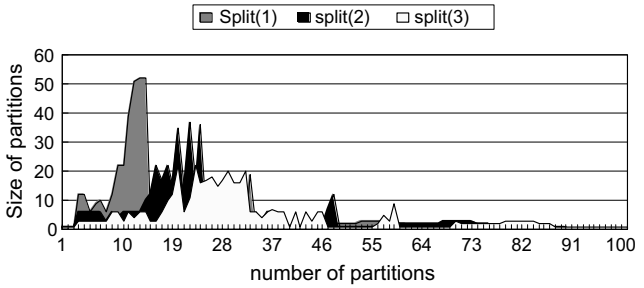


Fig. 9. Partition sizes for XMark data.

dependent only on the number of distinct tags and the number of split nodes, so that the size of partition index can be small even though the data is unstructured. The worst case for the size of a partition index is when each partition has only one path, that is, each data node has a unique tag. We can handle the worst case by using a well designed function to merge partitions, as described in Section 4.3.

The size of a partition will affect the performance of the partition index. We reduced the deviation of sizes by splitting nodes. Fig. 9 shows the results of the node splits. We split nodes of the partition index for the XMark data set. The maximum size of partitions decrease about 60% and the number of partitions increase about 23% after splitting nodes three times. The size of partitions decrease significantly and the number of nodes is still small enough to be suitable for an index graph after splitting nodes as shown in Fig. 9. We provide the influence of splitting partitions on the performance in the next subsection.

4.4. Query evaluation results

We present a comparison of our results on query evaluation costs for the partition index, the Dataguide and the data graph. We use the partition index with *split*(1) in the experiments.

Fig. 10 shows the results of query evaluation costs which are normalized to the cost of the Dataguide. The evaluation cost in each group is averaged over 50 queries (which is randomly selected) and measured by the number of I/Os. The first two groups show the costs for the XMark

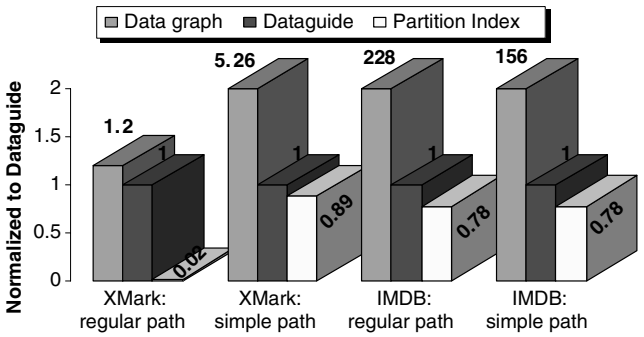


Fig. 10. Query evaluation results.

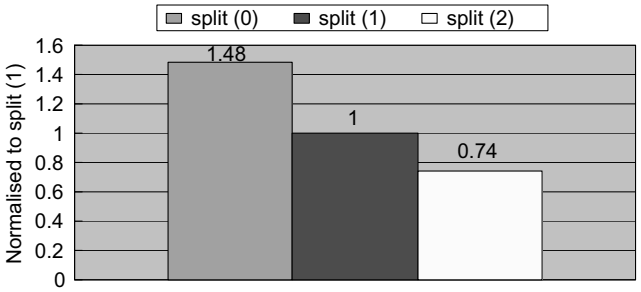


Fig. 11. Cost of partition processing.

data set and the second two groups for the IMDB data set. The first and the third groups show the costs for the regular path queries and the second and the fourth groups for the simple path queries.

For the simple path queries of the XMark data set, whenever processing each label in a path, a large amount of search space is reduced because of the deterministic feature of the Dataguide. So the evaluation cost of the Dataguide is much better than that of the data graph, even though the size of the Dataguide is almost as large as the size of data graph. Thus, it would be desirable to use the Dataguide as an index for simple path queries.

For the regular path queries of the XMark data set, the partition index performs much better than the Dataguide and the data graph. As described earlier, the partition index can keep a steady performance because the size of the partition index is independent of the structure of a data graph. The evaluation cost for the Dataguide is as expensive as that of the data graph. Regular path enlarge the search space of a graph and the size of the graph is crucial to the evaluation cost. In particular, the initial '_$*$' in a query cause the query to traverse every node of the graph.

For the regular path queries of the IMDB data set, the size of the Dataguide is very small compared with that of the data graph. It is a matter of course that the Dataguide has good performance on the IMDB data, and the same is true for the simple path queries of the IMDB data set. Actually, the IMDB data set is as regular as data on a relational table and this is why the Dataguide can be a good index for the IMDB data set. Nevertheless, We can see that the evaluation cost of the partition index is still better than that of the Dataguide from the figure.

The size of a partition is another fact that can increase the cost of query evaluation. By splitting the partitions, we can reduce the size of skewed partitions. Fig. 11 shows the cost of the partition processing. The processing cost is averaged over 50 queries which consist of regular paths mixed with simple paths. As shown in the figure, *split*(2) improve the processing cost 50% compared with *split*(0).

5. Conclusions and discussions

The partition index partitions all possible paths in a data graph and finds partitions for a query efficiently. By using a partition index we can always find the results of a query without traversing a data graph. The partition index guarantees the steady performance by adjusting the size of an

index graph and partitions. Partitions in a partition index are useful for the second level optimization of the index. We have introduced two partition processing techniques that exploit existing techniques. A partition in a partition index is not a semi-structured data but a set of values, and this feature of a partition gives great opportunities to use traditional data processing techniques.

Applications of XML include data representation on the Web, data exchange on business to business applications, and data integration for heterogeneous data sources. As a large amount of data are being populated in XML format, it is important to obtain necessary information from XML data. We proposed the partition index technique for efficient retrieving of necessary fraction of data from large XML data. Our technique can be used for an index of query processor in XML database management systems. A query optimizer can determine if a partition index should be used for a query.

Although we propose an efficient index technique, our technique has some limitations. We assume that a path start from the root node. Sometimes, a path does not start from the root node and our index cannot be applied to such a path query directly. We provided a simple technique to split a node whose partition is large enough to be split. We also provided a technique to merge nodes by using the function, $m^k(l)$. However, there can be occasions where some partitions still have large sizes. We believe that we can make optimized partitions using node splits with $anchor(k)$ instead of $anchor(1)$. We plan to extend our index technique to deal with a path query which may contain paths starting from arbitrary nodes. We also plan to improve our node split algorithm for building optimized partitions.

References

[1] S. Abiteboul. Querying semi-structured data, in: Proceedings of the International Conference on Database Theory, 1997.
[2] S. Abiteboul, D. Quass, J. McHugh, J. Widom, J. Wiener, The lorel query language for semistructured data, International Journal on Digital Libraries (1996).
[3] T. Bray, J. Paoli, C.M. Sperberg-McQueen. Extensible markup language (XML) 1.0, Technical Report, W3C Recommendation, 1998.
[4] P. Buneman, S. Davidson, G. Hillebrand, D. Suciu, A query language and optimization techniques for unstructured data, in: Proceedings of the ACM SIGMOD International Conference on the Management of Data, 1996.
[5] P. Buneman, M.F. Fernandez, D. Suciu, Unql: a query language and algebra for semistructured data based on structural recursion, VLDB Journal: Very Large Data Bases 9 (1) (2000) 76–110.
[6] D. Chamberlin, D. Florescu, J. Robie, J. Simeon, M. Stefanescu, XQuery: a query language for XML, Technical Report, W3C Working Draft, February 2001.
[7] C.-W. Chung, J.-K. Min, K. Shim. APEX: an adaptive path index for XML data, in: Proceedings of the ACM SIGMOD International Conference on the Management of Data, 2002.
[8] J. Clark, S. DeRose. XML Path Language (XPath) 1.0, Technical report, W3C Recommendation, 1999.
[9] B. Cooper, N. Sample, M.J. Franlin, G.R. Hjaltason, M. Shadmon. A fast index for semistructured data, in: Proceedings of the Conference on Very Large Data Bases, 2001.
[10] A. Deutsch, M.F. Fernandez, D. Florescu, A.Y. Levy, D. Suciu. A query language for XML, in: Proceedings of Eighth International World Wide Web Conference, 1999.
[11] M.F. Fernandez, D. Suciu, Optimizing regular path expressions using graph schemas, in: IEEE International Conference on Data Engineering, 1998.

[12] M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, K. Shim. XTRACT: a system for extracting document type descriptors from XML documents, in: Proceedings of the ACM SIGMOD International Conference on the Management of Data, 2000.
[13] R. Goldman, J. Widom. Dataguides: enabling query formulation and optimization in semistructured databases, in: Proceedings of the Conference on Very Large Data Bases, 1997.
[14] R. Kaushik, P. Shenoy, P. Bohannon, E. Gudes, Exploiting local similarity for indexing paths in graph-structured data, in: IEEE International Conference on Data Engineering, 2002.
[15] T. Milo, D. Suciu, Index structures for path expressions, in: Proceedings of the International Conference on Database Theory, 1999.
[16] S. Nestorov, J. Ullman, J. Wiener, S. Chawathe. Representative objects: concise representations of semistructured, hierarchical data, in: IEEE International Conference on Data Engineering, 1997.
[17] Y. Papakonstantinou, S. Abiteboul, H. Garcia-Molina. Object exchange across heterogeneous information source, in: IEEE International Conference on Data Engineering, 1995.
[18] S. Park, H.-J. Kim. A new query processing technique for XML based on signature, in: Proceedings of International Conference on Database Systems for Advanced Applications, 2001.
[19] D. Suciu, Semistructured data and XML, in: Proceedings of International Conference on Foundations of Data Organization, 1998.
[20] The Internet Movie Database Ltd. Internet movie database. Available from <http://www.imdb.com>.
[21] XMark, The xml benchmark project. Available from <http://monetdb.cwi.nl/xml/index.html>.

Jongik Kim received his BS and MS degree in computer science from KAIST (Korea Advanced Institute of Science and Technology), Taejon, Korea, in 1998 and 2000, respectively, and his Ph.D. degree in computer engineering from Seoul National University, Seoul, Korea, in 2004. His research interests include semistructured and XML databases, object-oriented databases, and spatial databases.

Hyoung-Joo Kim received his BS degree in computer engineering from Seoul National University, Seoul, Korea, in 1982 and his MS and Ph.D. in computer engineering from University of Texas at Austin in 1985 and 1988, respectively. He was an assistant professor of Georgia Institute of Technology, and is currently a professor in the Department of Computer Engineering at Seoul National University. His research interests include object-oriented databases, multimedia databases, HCI, and computer-aided software engineering.