A New Structure for Accelerating XPath Location Steps

Yaokai Feng, and Akifumi Makinouchi

Abstract—**Indexing technology is one of the kernel technologies in database management systems, especially for large datasets. XML (eXtensible Markup Language) has been successfully adopted as a universal data exchange format, particularly in the WWW environment. It is an important and basic task to efficiently implement XPath axes on XML documents. Using R*-tree, T. Grust proposed an interesting method to support all XPath axes. In this method, all of the nodes of an XML document are mapped to a point set in a five-dimensional space. T. Grust clarified the fact that each of the XPath axes can be implemented by a range query in the abovementioned five-dimensional space. Thus, R*-tree (one of the popular multidimensional indices) was used to improve the query performance for XPath axes. However, according to our investigations, most of the range queries for the XPath axes are partially-dimensional range queries. If the existing multidimensional indices are used for such range queries, a great deal of information that is irrelevant to the queries must also be read from disk. Based on this observation, a new multidimensional index structure, called Adaptive R*-tree (AR*-tree), is proposed herein to support the XPath axes more efficiently.**

Index Terms—**databases, multidimensional range queries, multidimensional index, XML data.**

I. INTRODUCTION

XML has been successfully adopted as a universal data exchange format, particularly in the World Wide Web, the problem of managing and querying XML documents poses challenges to database researchers. Although XML documents may have rather complex internal structures, they share the same data type underlying the XML paradigm: the ordered tree. Tree nodes represent document elements, attributes, or text data, while edges represent the element-subelement (or parent-child, ancestor-descendant) relationship.

For the purpose of retrieving such tree-shaped data, several XML query languages have been proposed in the literature. Examples include XPath [2] and XQuery [3]. XQuery is being standardized as a major XML query language, and the main building block of XQuery is XPath, which addresses part of XML documents for retrieval [16]. For example, "paragraph//section" is used to find all *section*s that are contained in each *paragraph*. Here, the double slash "//" represents the *ancestor-descendant* relationship. A single slash "/" in an XPath represents a *parent-child* relationship, for example "section/figure".

In line with the tree-centric nature of XML, XPath provides operators to describe path traversals in a tree-shaped document. Path traversals evaluate a collection of subtrees (forests), which may then, recursively, be subject to further traversal. Starting from a context node, an XPath query traverses its input document using a number of location steps. For each step, an axis describes which document nodes (and the subtrees below these nodes) form the intermediate result forest for this step. The XPath specification [2] lists a family of 13 axes (among these the children and descendant-or-self axes, which may be more widely known by their abbreviations / and //, respectively).

Generally, XPath expressions specify a tree traversal via two parameters: (1) a context node (not necessarily for the root) which is the starting point of the traversal, (2) and a sequence of location steps syntactically separated by /, evaluated from left to right. Given a context node, a step's axis (only one step of a regular XPath expression) establishes a subset of document nodes. This set of nodes, or forest, provides the context nodes for the next step, which is in turn evaluated for each node of the forest. The results are combined and sorted in document order. To illustrate the semantics of the XPath axes, Fig. 1 depicts the result forests for three steps along different axes taken from context node *e* (note that the preceding axis does not include the ancestors of the context node). Table I lists all XPath axes.

It is an important and basic task to efficiently implement XPath axes on XML documents. In work [1], the R*-tree has been successfully applied to implementing XPath axes and all of the XPath axes are support. In this work, each node of an XML document is labeled with a five-dimensional tuple. All of the nodes of the XML document are mapped to a point set in a five-dimensional space. Importantly, each of the XPath axes can be implemented by a range query on the above five-dimensional space. Thus, the R*-tree is helpful for improving the query performance of the range queries for XPath axes. This method has been proven efficient in [1]. However, according to our investigations, most of the range queries for the XPath axes are partially-dimensional range queries (i.e., the number of query dimensions in each of the range queries is less than five, although the R*-tree is built in a five-dimensional space). If the existing multidimensional indices (such as the R*-tree, which is used in [1]) are used for

Manuscript received June 10, 2009.

Yaokai Feng is with Graduate School of Information Science and Electrical Engineering, Kyushu University, Japan. Phone/Fax: +81-92-8023574. Email: *fengyk@ait.kyushu-u.ac.jp*

Akifumi Makinouchi is with Department of Information Network Engineering, Kurume Institute of Technology, Japan. Email: *akifumi@cc.kurume-it.ac.jp*

such range queries, then a great deal of information that is irrelevant to the queries also has to be read from disk, which heavily degrades the query performance. Based on this observation, in the present study, a new multidimensional index structure, called Adaptive R*-tree (AR*-tree), is proposed in order to support XPath axes more efficiently. The discussions and experiments with various datasets indicate that the Adaptive R*-tree is better suited to XML documents, especially large documents.

In the remainder of the present paper, Section 2 presents a number of related studies, and observations are presented in Section 3. Section 4 presents the proposed method, a new index structure for XPath axes, including its structure and a search algorithm. The experimental results are presented in Section 5, and Section 6 presents a further discussion. Section 7 concludes the present paper and describes future research.

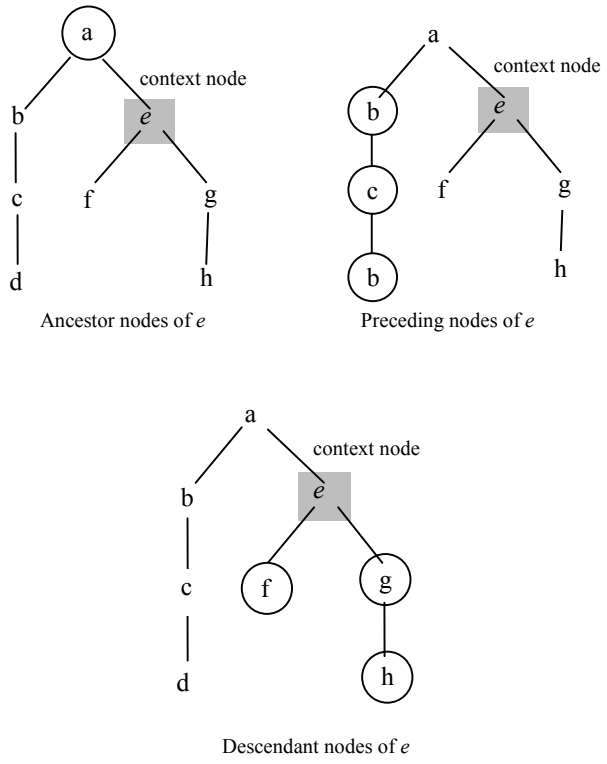

Fig. 1 Example of XPath axes (circled nodes are result elements).

II. Related Studies

The concept of regular path expressions dominates this field of research [4, 5, 6, 7, and 18]. One study [4] presented an index over the prefix-encoding of the paths in an XML document tree (in a prefix-encoding, each leaf *l* of the document tree is prefixed by the sequence of element tags encountered during a path traversal from the document root to *l*). Since tag sequences share common prefixes in such a scheme, a variant of the Patricia-tree is used to support lookups. Clearly, the index structure is tailored to respond to path queries that originate in the document root. Paths that do not have the root as the context node need multiple index lookups or require a post-processing phase (as does a restoration of the document order in the result forest). In [4], refined paths are proposed to remedy this drawback. Refined paths, however, have to be preselected before the index loading time.

Table I All XPath axes.

Axis	Result
child	Direct element child nodes of the context node
descendant	All descendant nodes of the context node
descendant-or-self	Like descendant, plus the context node
parent	Direct parent node of the context node
ancestor	All ancestor nodes of the context node
ancestor-or-self	Like ancestor, plus the context node
following	Nodes following the context node in document order
preceding	Nodes preceding the context node in document order
following-sibling	Like following, same parent as the context node
preceding-sibling	Like preceding, same parent as the context node
attribute	Attribute nodes of the context node
self	Context node itself
namespace	Namespace nodes of the context node

The T-index structure, proposed by Milo and Suciu in [6], maintains (approximate) equivalence classes of document nodes, which are indistinguishable with respect to a given path template. In general, a T-index does not represent the entire document tree, but rather only those document parts relevant to a specific path template. The more permissive and the larger the path template, the larger the resulting index size is. This allows space to be traded for generality. However, a specific T-index supports only those path traversals matching its path template (as reported in [6], an effective applicability test for a T-index is known for a restricted class of queries only).

There is other related work that is not directly targeted at the construction of index structures for XML. In [8], the authors discuss relational support for containment queries. In particular, the multi-predicate merge join (MPMGJN) presented in [8] would provide an almost perfect infrastructure for the XPath accelerator. MPMGJN supports multiple equality and inequality tests. The authors report an order of magnitude speed-up in comparison to standard join algorithms.

Another study [1] (and its extended version [18]) successfully adopted a multidimensional index structure in processing XML queries. In this previous study, an XPath accelerator was proposed that can completely live inside a relational database system, i.e., the structure is a relational storage structure in the sense of [10]. The implementation of the proposal in [1] benefits from advanced index technology, namely, the R-tree, which has by now found its way into mainstream relational database systems. The approach in [1] was developed with a close eye on the XPath semantics and it is able to support all XPath axes.

The main contributions of [1] are that (1) this study proposed a five-dimensional descriptor (labeling schema) for each node

of the XML document, (2) this study clarified that, using this labeling schema, each of the 13 XPath axes can be mapped to a range query in the five-dimensional descriptor-space, and (3) the range queries for XPath axes were implemented using the R*-tree.

In this paper, based on the abovementioned previous study [1], we will (1) present our observations on the range queries of XPath axes, and (2) according to the features of these range queries, present a new index structure (in place of the R*-tree) to further improve the query performance of XPath axes. Since the present study is based on [1], the key concept of [1] is described below.

A. Labeling schema and mapping XPath axes to range queries

Each node v of an XML document is represented by the following five-dimensional descriptor:

$$desc(v)=<pre(v), post(v), par(v), att(v), name(v)>,$$

where $pre(v)$ and $post(v)$ are the preorder and postorder of v, respectively. $par(v)$ is the preorder of the parent node of v. $att(v)$ is a Boolean value indicating whether v is an attribute node. Finally, $name(v)$ is the name of v. In this way, all of the nodes in an XML document can be mapped to a set of points in the five-dimensional *descriptor space* (or *labeling space)*.

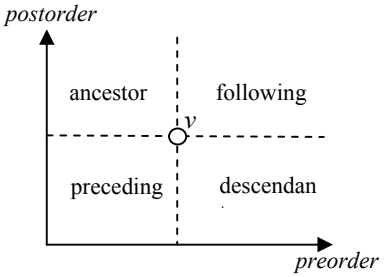

Fig. 2 Four XPath axes in two-dimensional space.

As others have noted [5, 8, 11], $pre(v)$ and $post(v)$ can be used to efficiently characterize the descendants v' of v. We have that

$$v' \text{ is a descendant of } v \Leftrightarrow pre(v')>pre(v) \wedge post(v') < post(v) \quad (1)$$

In the same way, we have that

$$v' \text{ is a ancestor of } v \Leftrightarrow pre(v')<pre(v) \wedge post(v')>post(v) \quad (2)$$

$$v' \text{ is a preceding node of } v \Leftrightarrow pre(v') < pre(v) \wedge post(v') < post(v) \quad (3)$$

$$v' \text{ is a following node of } v \Leftrightarrow pre(v') > pre(v) \wedge post(v') > post(v) \quad (4)$$

According to the above four equations, we can see that the four XPath axes of *descendant*, *ancestor*, *preceding*, and *following* can be mapped to range queries in the two-dimensional space of *preorder*/*postorder*, which is shown in Fig. 2. With the help of the other items in the five-dimensional descriptor, the other XPath axes can also be mapped to range queries. Table Ⅱ presents the ranges of all the XPath axes. As in [1], the two axes of *self* and *namespace* are omitted because they are so simple.

Table Ⅱ XPath axes and their ranges in descriptor space (v is the context node).

XPath Axes	ranges in descriptor-space				
	pre	*post*	*par*	*att*	*name*
child			*pre*(*v*)	F	*
descendant	(*pre*(*v*),∞)	[0, *post*(*v*))		F	*
descendant-or-self	[*pre*(*v*),∞]	[0, *post*(*v*)]		F	*
parent	[*par*(*v*), *par*(*v*)]				*
ancestor	[0, *pre*(*v*))	(*post*(*v*), ∞)			*
ancestor-or-self	[0, *pre*(*v*)]	[*post*(*v*), ∞)			*
following	(*pre*(*v*),∞)	(*post*(*v*), ∞)		F	*
preceding	[0, *pre*(*v*))	[0, *post*(*v*))		F	*
following-sibling	(*pre*(*v*),∞)	(*post*(*v*), ∞)	*par*(*v*)	F	*
preceding-sibling	[0, *pre*(*v*))	[0, *post*(*v*))	*par*(*v*)	F	*
attribute			*pre*(*v*)	T	*

B. Implementation of range queries using R-tree*

Since all of the XPath axes can be mapped to range queries in the five-dimensional descriptor-space, the R*-tree (used in [1]) seems helpful for improving the range query performance. Since we will propose a new structure to further improve the range query performance, the R*-tree is briefly reviewed here.

The R*-tree [12] is a hierarchy of nested multidimensional MBRs. Each non-leaf node of the R*-tree contains an array of entries, each of which consists of a pointer and an MBR. The pointer refers to one child node of this node and the MBR is the minimum bounding rectangle of the child node referred to by the pointer. Each leaf node of the R*-tree contains an array of entries, each of which consists of an object identifier and the object itself (for point-object datasets) or its MBR (for extended object datasets). In the present paper, the object and tuple are used interchangeably. In the R*-tree, the root node corresponds to the entire index space and each of the other nodes represents a sub-space (i.e., the MBR of all of the objects contained in this region) of the space formed by its parent node. Note that, each MBR in the R*-tree nodes is denoted by two points. One is the lowest vertex with the minimum coordinate in each axis and the other is the upper-most vertex with the maximum coordinate in each axis. When the R*-tree is used for a range query, all of the nodes intersecting the query range are accessed and their entries must be checked.

III. Observations

From the abovementioned Table Ⅱ, we can observe that most of the query ranges of XPath axes use only partial items of the five-dimensional descriptor. For example, the *child* axis uses only *par* and *att* and the *parent* axis uses only *pre.* In the present paper, the range queries that use only some (rather than all) of the dimensions of the entire space are called partially-dimensional range queries (denoted as PD range queries). In contrast, the range queries that use all dimensions of the entire space are called all-dimensional range queries (denoted as AD range queries).

Note that all of the existing multidimensional indices are designed to evaluate AD range queries because all of the objects are clustered in the leaf nodes according to their information in all index dimensions and every node contains the information of its entries in all of the index dimensions. Actually, they can also evaluate PD range queries as follows. Using one n-dimensional index in the entire n-dimensional index space, one PD range query using d ($d < n$) query dimensions can be evaluated by simply extending the query range in each of the (n-d) irrelevant index dimensions to the entire data range.

However, a disadvantage of using all-dimensional indices for PD range queries is that each node of the index contains n-dimensional information, but only d-dimensional information is necessary for a PD range query using only d ($d < n$) dimensions. This means that a great deal of unnecessary information, i.e., the information in the irrelevant dimensions, also has to be read from disk, which degrades the query performance. In other words, the irrelevant information in the index nodes decreases the capacity (fanout) of each node. Directing against this disadvantage and considering that most of the query ranges of XPath axes are PD range queries, a new index structure for indexing XML data is proposed in the present paper.

IV. New structure: the AR*-tree

According to the features of the range queries for XPath axes, the Adaptive R*-tree (denoted as AR*-tree) is proposed in order to improve the performance of such range queries.

A. Structure

The key concept of the AR*-tree is to divide each of the n-dimensional R*-tree nodes into n one-dimensional nodes (these n one-dimensional nodes form a *node-group*), each of which holds the information in one dimension, while each node of the R*-tree holds the information in all of the index dimensions. The general structure of the AR*-tree is depicted in Fig. 3.

Whereas each entry in R*-tree nodes includes the MBR information in all of the dimensions, each entry in the nodes of the AR*-tree includes only one-dimensional information. In each node-group, each set of entries having the same index (location) and distributed in different nodes forms an *entry of node-group*, which corresponds to a complete MBR. Each entry of each node in one node-group corresponds to an edge of the MBR, whereas each of the entries in the index nodes of the R*-tree corresponds to a complete MBR.

Fig. 4 shows the structure of the AR*-tree node-group. All of the entries with the same index in the n nodes of this node-group form a complete n-dimensional MBR in the index space. Whereas each entry in the R*-tree nodes includes MBR information in all of the dimensions, each entry in the nodes of the AR*-tree includes only one-dimensional information. The term entry of node-group is used hereinafter, which refers to the set of entries having the same index distributed in all of the different nodes of one node-group. One entry of each index node-group corresponds to a complete MBR in the index space. In Fig. 4, all of the entries in an ellipse form a complete entry of the node-group, which is a complete MBR in the entire index space.

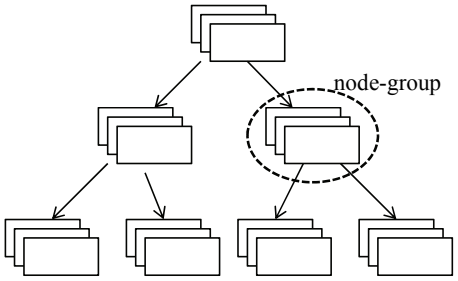

Fig. 3 General structure of AR*-tree.

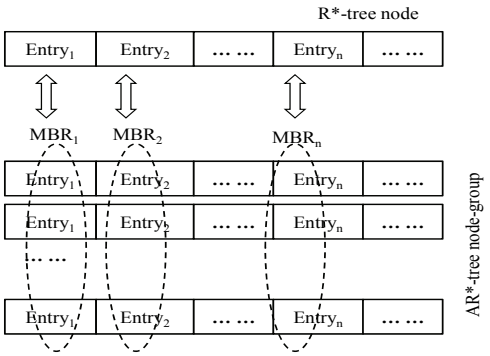

Fig. 4 Structure of AR*-tree node-group.

Fig. 5 is an example of entries in a node-group of the AR*-tree in a two-dimensional index space. In this example, each complete MBR is divided into two parts, which are separately contained in two nodes of one node-group. For example, $Xentry_i$ and $Yentry_i$ in Fig. 5 correspond to the two edges of MBR_i paralleling the X-axis and the Y-axis, respectively. That is, $Xentry_i + Yentry_i = MBR_i$.

The question then arises as to whether the total number of nodes in the AR*-tree becomes n times that in the R*-tree, because each node of the R*-tree has been divided into n nodes. However, this is not the case. The maximum number of entries in each node of the AR*-tree is up to approximately n times that in the R*-tree because the dimensionality of each node in the AR*-tree becomes 1. The structure of the AR*-tree guarantees that it can be applied to PD range queries with any

combinations of the query dimensions and that only the relevant one-dimensional nodes are visited.

The main advantage of the AR*-tree over the R*-tree (all-dimensional index) is that, for PD range queries, only the relevant nodes of the accessed node-groups need be visited, and the other nodes, even though they are in the same node-groups, can be skipped. That is, the information in the irrelevant dimensions (the dimensions that are not used in the present query) need not be read from disk. However, in the R*-tree, the information in all of the index dimensions is contained in each R*-tree node, but only information in the query dimensions are necessary for PD range queries, which means that a great deal of irrelevant information has to be loaded from disk, and this degrades the search performance, especially for large datasets.

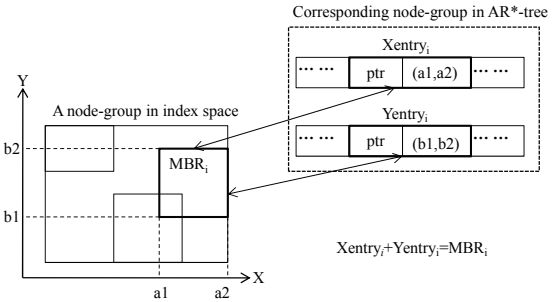

Fig. 5 Example in a two-dimensional space.

B. Algorithms of the AR-tree*

The insert algorithm of the AR*-tree is a naive extensions of the counterparts of the R*-tree. After the new tuple reaches the leaf node-group, it is divided and stored in different nodes of the leaf node-group according to dimension. If a node-group must be split, then all of its nodes must be split at the same time and the split may be up propagated. After a delete operation, if the node-group under-flowed, then all of its nodes should be deleted at the same time and all of its entries are reinserted to the AR*-tree. That is, all of the nodes in each node-group must be born simultaneously and die simultaneously.

A range query algorithm for the AR*-tree, which can be used for AD range queries and PD range queries, is shown in Table Ⅲ.

Table Ⅲ Algorithm for range queries on the AR*-tree.

```
Procedure RangeQuery (rect, node-group)
Input:  rect: query range
  node-group: initial node-group of the query
Output: result: all the tuples in rect
Begin
For each entry e in node-group Do
    If e INTERSECT rect in all query dimensions Then
        If node-group is not at leaf Then
            RangeQuery (rect, e.child);
            //e.child means the child node-group of e
        Else result ←e
EndFor
End
```

Note that

1) An entry in a node-group includes all of the parts with the same index in the different nodes of this node-group, i.e., all of the parts in one ellipse in Fig. 4.

2) When an entry is checked to determine whether it intersects the query range or not, only the nodes in the query dimensions are accessed and the other nodes in the current node-group are skipped. EVEN, not all of the nodes in the query dimensions need to be checked, because the investigation of the current entry can be stopped if the entry is found not to intersect the query range in the current query dimension.

Starting with the root node-group, each entry of the current node-group must be checked to determine whether its MBR intersects the query range or not. If its MBR intersects the query range and the current node-group is not at the leaf level, then this algorithm is invoked recursively with the corresponding child node-group. Note that, when each entry of the current node-group is checked, (1) not all of the nodes in the current node-group have to be accessed (such irrelevant nodes are skipped), and (2) in each of the visited node-groups, not all of the nodes in the relevant dimensions (query dimensions) must be visited. That is, after the current entry is found not to intersect the query range in the present-investigating dimension, further checks are not necessary. An example is shown in Fig. 6.

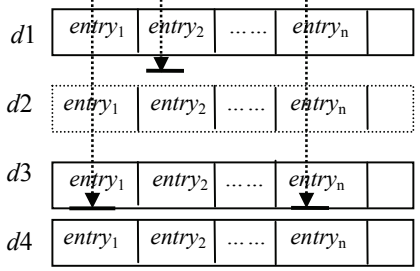

Fig. 6 Example of a four-dimensional AR*-tree node-group.

In the example in Fig. 6, three dimensions, *d*1, *d*3, and *d*4 of the four index dimensions are used in the present query, and the node group shown in Fig. 6 is visited. Each of the entries (an MBR) in this node-group is investigated edge by edge to decide whether it intersects the query range. Since all of the investigations of the entries stopped before the investigation of the node *d*4, the node *d*4 can be skipped, although *d*4 is one of the query dimensions.

The example in Fig. 6 also indicates that, even for AD range queries, the AR*-tree probably has better query performance than the R*-tree. Although, intuitively, it may seem that all of the nodes in the visited node-groups have to be accessed for AD range queries, this is not true. Another example in a three-dimensional index space is shown in Fig. 7.

In Fig. 7, clearly, the current query is an AD range query. Since the MBR of the current node-group intersects the query range, the entries (the dotted cuboids) of this node-group should be investigated. Since all of these entries do not

intersect the query range in the X-Y plane, the Z-axis need not be checked. That is, the node corresponding to the Z-axis in this node-group can be skipped, and the information in the Z-axis in this node-group need not be read from disk. Note that, if the X-Z plane is checked first, the node corresponding to the Y-axis can be skipped. More importantly, for the higher-dimensional spaces, since the MBRs (entries) in each node-group become increasingly sparse, it generally becomes possible to skip more nodes in the visited node-groups. This means that, in the visited node-groups, the information in one or more dimensions may not need to be read from secondary storage, even for AD range queries. In contrast, for the R*-tree, the information corresponding to all of the dimensions in the visited nodes must be read from disk. Thus, even for AD range queries, the AR*-tree probably has better query performance than the R*-tree.

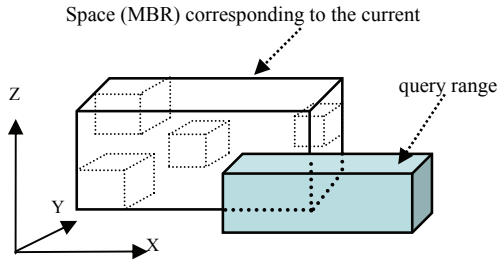

Fig. 7 Example of AD range queries in a three-dimensional space.

C. Discussion on Search Performance

In this section, under the assumption of uniformity, the performance of the AR*-tree for PD range queries is examined mathematically by comparing the performance of the AR*-tree with that of the R*-tree. That is, the tuples (objects) are assumed to be distributed uniformly in the index space. The number of accessed leaf nodes is estimated and compared since it is an important factor with regard to query performance [13], particularly for large datasets. The symbols used in this section are described in Table Ⅳ.

In the case of the R*-tree, the average number of leaf-node accesses (i.e., the number of leaf nodes intersecting the query range), R_l, can be given by

$$R_l = \frac{S_q}{S} \times N_l .$$

If the AR*-tree is used, the average number of leaf node groups intersecting the query range, AR_g, can be given as

$$AR_g = \frac{S_q}{S} \times N_g .$$

Since the node sizes of the R*-tree and the AR*-tree are the same (one node one page), the maximum number of entries in each leaf node of the AR*-tree is roughly n times that in each leaf node of the R*-tree. This is easy to understand considering that the dimensionality of each leaf node in the R*-tree is n times that in the AR*-tree. That is, only one-dimensional information of each entry is contained in each node of the AR*-tree, while n-dimensional information of each entry is required in every node of the R*-tree. In addition, considering that the clustering algorithms (insert algorithms) of the R*-tree and the AR*-tree are the same, we have

$$\frac{N_g}{N_l} \approx \frac{M_r}{M_g} \approx \frac{1}{n} .$$

Table Ⅳ Symbols and their descriptions.

n	Dimensionality of the entire index space
d	Number of query dimensions
S	Volume of the entire index space
Sq	Volume of the extended query range of the PD range query♣
Mr	Maximum number (capacity) of entries in each leaf node of the R*-tree (see [3])
Mg	Maximum number (capacity) of entries in each leaf node-group of the AR*-tree
Nl	Number of leaf nodes in the case of the R*-tree
Ng	Number of leaf node-groups in the AR*-tree

♣ The extended query range of one PD range query refers to the n-dimensional range obtained by extending the d-dimensional given query range in the way that the query ranges in the unused (n-d) dimensions are regarded as their entire data ranges.

In each accessed node-group, at most d nodes are visited for each d-dimensional PD range query. Thus, the number of leaf nodes (not the node-groups) that must be visited, AR_l, can be given by

$$AR_l \le AR_g \times d \approx \frac{S_q}{S} \times N_g \times d$$
$$\approx \frac{S_q}{S} \times \frac{1}{n} \times N_l \times d = \frac{d}{n} \times R_l < R_l \qquad (when \;\; d < n).$$

The last equation indicates that, for PD range queries with $d < n$, the number of accessed leaf nodes in the case of the AR*-tree is less than that in the case of the R*-tree. If $d = n$, then the number of accessed leaf nodes may be approximately the same and it is also possible that $AR_l < R_l$. More importantly, for a fixed n, the lower the number of query dimensions, d, the bigger the advantage of the AR*-tree compared to the R*-tree. This equation can be explained as follows. Since the capacity of each leaf node-group in the AR*-tree is roughly n times that of each leaf node in the R*-tree, the number of accessed leaf node-groups in the AR*-tree is approximately $1/n$ times that of the accessed leaf nodes in the R*-tree. However, in each of the accessed leaf node-groups of the AR*-tree, at most, d nodes must be visited. That is, although this query is relevant to d dimensions, the investigation of the current entry may stop midway if this entry does not intersect the query range in the current dimension.

Note that the above equations are only rough estimations.

V. Experiments

A. Experiment Process

The procedure of the experiments performed in the present study is shown in Fig. 8.

XML documents. The XML documents used in the experiments are generated by XMLgen [17], an easily accessible XML generator, which is developed for the Xmark benchmark project [14]. Using XMLgen, the three documents shown in Table V were generated and used. In this table, XMLgen factor was given as a size factor to control document sizes.

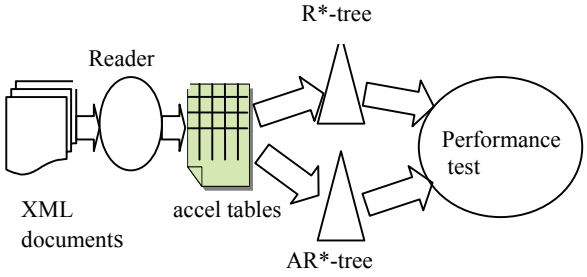

Fig. 8 Experimental procedure.

Table V XML documents used in the experiments.

Document size [MB]	Number of nodes	XMLgen factor
5.5	103,135	0.05
11.1	206,130	0.10
22.4	413,108	0.20
113.8	1,666,316	1.00

Reader. Based on Libxml2 [15] which is an XML C parser and toolkit, we built a loader to obtain one accel table (described in the next paragraph) for each XML document, which is used to build the indices. Since the XML documents have a total of 77 different node names, all of the possible node names are encoded from 0 to 76 in order to be handled by the indices.

Accel table. Each tuple of the accel table is the five-dimensional descriptor of one node of an XML document. As mentioned above, all of the node names are encoded from 0 to 76. The accel table is directly used to build the R*-tree and the AR*-tree.

Building indices. An R*-tree and an AR*-tree are constructed for the accel table of each XML document. The node size is set to 4,096 bytes.

Performance test. We assume that the multidimensional index is disk-resident, which is reasonable for large datasets. Thus, the query performance is tested in terms of the number of node accesses. Except for the three XPath axes of self, attribute, and namespace (they are too simple, and the performance difference between the R*-tree and the AR*-tree could not be shown clearly), the query performance of all of the other 10 XPath axes are tested using the R*-tree and the AR*-tree, respectively. By comparing the query performance of the XPath axes on the R*-tree and the AR*-tree, we will determine whether the R*-tree or the AR*-tree is better suited to XPath axes.

B. Experimental results

The experimental results are shown in Tables VI~VIII. The context nodes for different XML documents are chosen independently. That is, the same XPath axis may be tested with different context nodes for different data documents. For the sake of comparison, all of the tests on the R*-tree and the AR*-tree for the same XML documents used the same context nodes.

From the experimental results, we can obtain the following observation. Except for the XPath axes of *ancestor*s (including *ancestor-or-self*) and *preceding-sibling*, for which the advantage of the AR*-tree is not shown very clearly, the AR*-tree clearly performs better than the R*-tree for the other seven XPath axes. As mentioned in Section 1, the query performance of XPath axes is very important, because the main building block of XQuery is Xpath, and XPath expressions consist of a sequence of XPath axis operations, which are evaluated from left to right. Moreover, each step of an XPath expression (one XPath axis operation) often obtains a great number of intermediate results, which means that the evaluation of one XPath expression may require a great number of XPath axis operations. Thus, any improvement in the query performance of XPath axes will be significant.

Table VI Experiment results.

Documents size(MB)		parent	ancestor	descenden	following
5.5	R*	10.3	13.3	749.0	1420.0
	AR*	4.4	12.9	582.1	1095.2
11.1	R*	11.6	17.8	1507.2	2700.3
	AR*	6.4	14.5	1163.0	2072.2
22.4	R*	13.5	19.3	2995.6	5705.6
	AR*	7.0	15.4	2304.6	4375.1
113.8	R*	25.7	31.7	6236.3	9678.5
	AR*	13.0	23.3	4556.9	7612.6

Table VII Experiment results.

Documents size (MB)		preceding	following-sibling	child
5.5	R*	1186.2	485.1	484.2
	AR*	973.4	338.3	116.0
11.1	R*	1641.3	493.2	1200.4
	AR*	1321.0	353.0	230.5
22.4	R*	2972.0	1030.5	1848.3
	AR*	2323.6	680.4	439.4
113.8	R*	5346.7	3100.2	3996.2
	AR*	4952.1	1932.1	1509.1

Table Ⅷ Experiment results.

Document size (MB)		preceding-sibling	descendent-or-self	ancestor-or-self
5.5	R*	22.6	749.0	13.3
	AR*	21.1	582.1	12.9
11.1	R*	17.8	1507.2	17.8
	AR*	17.7	1163.0	14.5
22.4	R*	21.9	2995.6	13.3
	AR*	18.0	2304.6	12.4
113.8	R*	32.3	5693.3	29.2
	AR*	25.6	4973.1	19.4

VI. How about multi-BTree?

Another naive approach to handling PD range queries is based on the B-tree, herein referred to as *multi-Btree*. In this approach, one B-tree (or a variant thereof) is constructed in each index dimension, using the projections of the objects (tuples for relational data). For PD range queries, the corresponding B-trees are used individually and their results are intersected to obtain the final query result. In total, n B-trees should be constructed in advance for an n-dimensional index space.

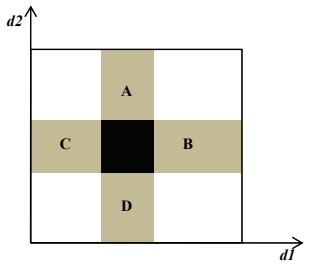

Fig. 9 A PD range query using the multi-Btree.

Fig. 9 is an example of a two-dimensional PD range query evaluated using the multi-Btree, where the two dimensions ($d1$ and $d2$) are used as query dimensions. In this case, the two B-trees constructed on $d1$ and $d2$ are used.

In Fig. 9, the thick shadow region is the given query range. Two range queries are first evaluated on the two corresponding B-trees. All of the objects located in the vertical shadow region and the horizontal shadow region are reported as intermediate results, $R1$ and $R2$, respectively, and then the final result of this PD range query is given by $R1 \cap R2$.

The main advantages of the AR*-tree over the multi-Btree are as follows.

(1) When an entry of a node-group is checked to determine whether it intersects the query range, the AR*-tree can make a decision according to the information in all of the query dimensions, i.e., mutual reference is possible. As a result, regions A, B, C, and D in Fig. 5 can be skipped. However, mutual reference is impossible in queries using the multi-Btree because each B-tree contains only one-dimensional information and these B-trees are used independently. That is, during searching on each B-tree, the algorithm cannot realize the query ranges in the other query dimensions. Thus, a number of unnecessary investigations are thus performed, and a great deal of irrelevant information is read from disk.

(2) In the AR*-tree, only one index is needed, while multiple B-trees are necessary in the multi-Btree. The management and updating of such B-trees incur additional costs.

(3) In the multi-Btree, too many intermediate results may be reported and the intersection operation on the intermediate results may be very time-consuming. Consider a dataset having 1,000,000 data points uniformly distributed in a six-dimensional space. Assume that the given PD range query has four query dimensions and that the query range in each of the four query dimensions is 1/10 of the entire data domain in each respective dimension. In this case, the final result has only $10^6/10^4 = 100$ objects. However, the query result on each B-tree has 10^5 objects and the total number of intermediate results is $4*10^5$.

Thus, the multi-Btree cannot be efficiently used for multidimensional range queries. The performance of multi-dimensional range queries using the multi-Btree has been discussed and investigated in our previous study [19].

VII. Conclusion

The query performance of XPath axes is very important because they are the main building blocks of XQuery. The evaluation of one XPath expression may require a great number of XPath axis operations because, often, each step of an XPath expression (one XPath axis operation) obtains a great number of intermediate results. Thus, any improvement in the query performance of XPath axes will be significant. Multi-dimensional indices have been successfully introduced to the field of querying on XML data. The existing methods apply all-dimensional indices (such as the R*-tree in [1]). In the present paper, a new multidimensional index structure, called AR*-tree, was proposed and discussed. Discussion and experiments using various XML documents showed that the proposed method has a clear performance advantage for XPath axes, as compared with the R*-tree, a well known and popular multi-dimensional index structure. In the future, the performance of the AR*-tree for XPath axes will be examined using various types of XML documents.

Acknowledgments

The authors would like to thank Mr. Satoshi Tani for conducting the experiments.

References

[1] T. Grust: Accelerating XPath Location Steps. Proc. ACM SIGMOD International Conference, pages 109-120, 2002.

[2] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, et .al.: XML Path Language (XPath) 2.0. Technical Report W3C Working Draft, Version 2.0, World Wide Web Consortium, December 2001. http ://www. w3. org/TR/xpath20/.

[3] S. Boag, D. Chamberlin, M. F. Fernandez,et.al.: XQuery 1.0: An XML query language. In W3C Working Draft: http://www.w3.org/TR/xquery/, 2002.

[4] B. F. Cooper, N. Sample, M. J.Franklin, G. R. Hjaltason, and M. Shadmon: A Fast Index for Semistructured Data. Proc. the 27th International Conference on Very Large Data Bases (VLDB), pages 341-360, 2001.

[5] Q. Li and B. Moon: Indexing and Querying XML Data for Regular Path Expressions. Proc. the 27th International Conference on Very Large Data Bases (VLDB), pages 361-370, 2001.
[6] D. Suciu and T. Milo: Index Structures for Path Expressions. Proc. the 7th International Conference on Database Theory (ICDT), LNCS 1540, pages 277-295 Springer Verlag, 1999.
[7] R. Goldman and J. Widom: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. Proc. the 23rd International Conference on Very Large Databases (VLDB), pages 436-445, 1997
[8] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman: On Supporting Containment Queries in Relational Database Management Systems. Proc. ACM SIGMOD International Conference on Management of Data, pages 425-436, 2001.
[9] H. P. Kriegel, M. Potke, and T. Seidl: Managing Intervals efficiently in Object-Relational Databases. Proc. the 26th International Conference on Very Large Databases (VLDB), pages 407-418, 2000.
[10] H. P. Kriegel, M. P. otke, and T. Seidl: Managing Intervals Efficiently in Object-Relational Databases. Proc. the 26th International Conference on Very Large Databases (VLDB), pages 407-418, 2000.
[11] P. F. Dietz and D. D. Sleator: Two Algorithms for Maintaining Order in a List. Proc. the 19th Annual ACM Symposium on Theory of Computing (STOC), pages 365-372, 1987. ACM Press.
[12] N. Beckmann, and H. Kriegel: The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. Proc. ACM SIGMOD International Conference., pages.322-331, 1990.
[13] G.R.l Hjaltason and H. Samet: Distance Browsing in Spatial Database. ACM Transactions on Database Systems, Vol.24, No.2, pages 265-318, 1999.
[14] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse: The XML Benchmark Project. Technical Report INSR0103, CWI, Amsterdam, The Netherlands, April 2001.
[15] XML C parser and toolkit: http://xmlsoft.org/
[16] H. Jiang, H. Lu, W. Wang, B. C. Ooi:XR-Tree: Indexing XML Data for Efficient Structural Joins. Proc. the 19th International Conference on Data Engineering (ICDE), pages 253-263, 2003.
[17] XMLgen: http://monetdb.cwi.nl/xml/downloads.html
[18] T. Grust, M. V. Keulen, and J. Teubner: Accelerating Xpath Evaluation in Any RDBMS. ACM Transactions on Database Systems, Vol. 29, No.1, pages 91-131, 2005.
[19] Y. Feng, A. Makinouchi: Efficient Evaluation of Partially- dimensional Range Queries Using Adaptive R*-tree. Proc. the 17th International Conference on Database and Expert Systems Applications (DEXA), LNCS 4080, pages 687-696, Springer-Verlag, 2006.

Yaokai Feng received his B.S. and M.S. degrees in Computer Science from Tianjin University, China, in 1986 and 1992, respectively. Since he received PhD degree in Information Science from Kyushu University, Japan, in 2004, he has been staying in the same university as an research associate. Now, he is an assistant professor in the same university. He is a member of IPSJ, IEEE, ACM and an editorial board member of IAENG International Journal of Computer Science.

Akifumi Makinouchi received his B.E. degree from Kyoto University, Japan, in 1967, Docteur-ingereur degree from Univrcite de Grenoble, France, in 1979, and D.E. degree from Kyoto University, Japan, in 1988. From 1990 to 2006, he was with the Graduate School of Information Science and Electrical Engineering, Kyushu University, Japan, where he was a professor. From 2006, he is a professor in the Department of Information Network Engineering, Kurume Institute of Technology, Japan. He is a member of IPSJ, ACM, and IEEE and fellow of IEICE.