Efficient evaluation of partially-dimensional range queries in large OLAP datasets

Feng, Yaokai
Graduate School of Information Science and Electrical Engineering, Kyushu University

Kaneko, Kunihiko
Graduate School of Information Science and Electrical Engineering, Kyushu University

Makinouchi, Akifumi
Department of Information Network Engineering, Kurume Institute of Technology

出版情報：International Journal of Data Mining, Modelling and Management. 3 (2), pp.150-171, 2011-07. Inderscience
バージョン：

Efficient Evaluation of Partially-dimensional Range Queries in Large OLAP Datasets

Yaokai Feng†*, Kunihiko Kaneko†, Akifumi Makinouchi††

†Graduate School of Information Science and Electrical Engineering,
Kyushu University
744 Motooka, Nishi-ku, Fukuoka, Japan
Email: fengyk@ait.kyushu-u.ac.jp, Kaneko@ait.kyushu-u.ac.jp

††Department of Information Network Engineering,
Kurume Institute of Technology, Kurume, Japan
Email: akifumi@cc.kurume-it.ac.jp
*Corresponding author

Abstract In light of the increasing requirement for processing multidimensional queries on OLAP (relational) data, the database community has focused on the queries (especially range queries) on the large OLAP datasets from the view of multidimensional data. It is well known that multidimensional indices are helpful to improve the performance of such queries. However, we found that much information irrelevant to queries also has to be read from disk if the existing multidimensional indices are used with OLAP data, which greatly degrade the search performance. This problem comes from particularity on the actual queries exerted on OLAP data. That is, in many OLAP applications, the query conditions probably are only with partial dimensions (not all) of the whole index space. Such range queries are called PD (Partially-Dimensional) range queries in this study. Based on R*-tree, we propose a new index structure, called AR*-tree, to counter the actual queries on OLAP data. The results of both mathematical analysis and many experiments with different datasets indicate that the AR*-tree can clearly improve the performance of PD range queries, esp. for large OLAP datasets.

Key words: OLAP, multidimensional index, multidimensional range query, R*-tree, relational data, B^+-tree.

1 Introduction

There is increasing requirement for processing multidimensional range queries on OLAP (relational) data. Typical examples like "**Select … From** *LINEITEM* **Where** QUANTITY ≤ 25 and 0.1 ≤ DISCOUNT ≤ 0.3 and 2001-01-01 ≤ SHIPDATE ≤ 2001-12-31", where *LINEITEM* is a table of TPC-H benchmark, with 16 attributes including QUANTITY, DISCOUNT, SHIPDATE, and so on. Thus, researches and applications in the OLAP field are developing quickly in recent years and many models/platforms have been presented. For example, OLAM (On Line Analytical Mining) is a powerful technology for supporting knowledge discovery from large databases and data warehouses that mixes together OLAP functionalities and data mining algorithms for extracting regularities from data (Cuzzocrea, 2007).

However, the issue that how to efficiently find necessary information from OLAP data is still at the basic position. This paper is about how to efficiently evaluate the actual queries on OLAP data. It has been made clear that range queries on large OLAP datasets can be efficiently implemented from the viewpoint of multidimensional data. Thus, multidimensional indices are helpful to improve query performance of such multidimensional range queries (Markl, 1999a, 1999b), in which the tuples of OLAP data are clustered among the leaf nodes to restrict the node accesses for queries.

Many multidimensional index structures have been proposed. However, in many cases that existing indexing technologies are used in OLAP environments, a great deal of information irrelevant to queries also has to be read from disk, which greatly degrades the query performance. This problem arises from the fact that, in many applications involving OLAP data, the query conditions are likely of only partial

dimensions (not all) of the entire index space. Such range queries are referred to herein as partially-dimensional (*PD*) *range queries*. That is, although the index is built in an *n*-dimensional space, each of the actual range queries may only use *d* (*d* < *n*) dimensions. Moreover, the dimensions used in every query and their quantities are not fixed and are not known in advance. The attributes (dimensions) used in the query condition of each PD range queries are called *query attributes* (*dimensions*). On the contrary, those having query ranges (i.e., a query conditions) given in every dimension of the entire index space are called all-dimensional (AD) range queries.

Figure 1 shows examples of PD and AD range queries. The shaded regions are the query range. The query conditions in (a) and (b) are of the form "WHERE $a_1 < X < a_2$", where only the index attribute X is used in the query condition, even though the index spaces have two and three dimensions, respectively. The query condition in (c) is of the form "WHERE $a_1 < X < a_2$ AND $c_1 < Z < c_2$", where two dimensions of the entire three-dimensional index space are used in the query condition. The queries in (a), (b) and (c) are PD range queries because only partial dimensions of the entire index space are used in the query condition. However, the query in (d) is an AD range query because all of the index dimensions are used in the query condition.

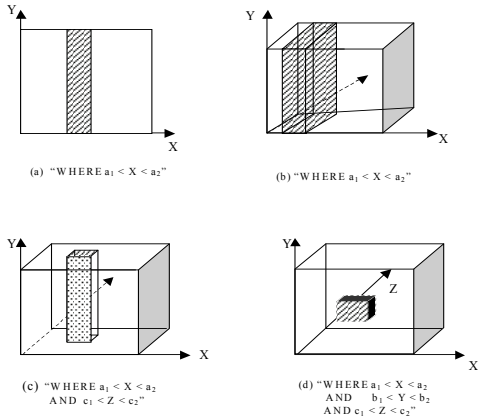


Figure 1 PD and AD range queries.

Based on this observation, an index structure for efficiently evaluating PD range queries in OLAP applications is proposed in this paper. It can be used for PD range queries having any combinations of the query dimensions and only the necessary information is read from the disk. We discuss our proposal based on the R*-tree (Beckmann, 1990). This is because that the R*-tree is a well-known variant of the R-tree and has been used in many researches and some commercial database products (Informix, 2004). The results of both mathematical analysis and numerous experiments with various datasets indicate that our proposal is more suitable for OLAP data than the existing indexing methods.

In the remainder of this paper, following a presentation of related works in Section 2, Section 3 describes and discusses three naïve methods for PD range queries. Then, the proposed method handling PD range queries, AR*-tree, is presented and explained in Section 4. Section 5 presents a mathematical investigation on the search performance of the AR*-tree. The experimental results are presented in Section 6. Finally, the conclusion and our future work are drawn in Section 7.

2 Related Work

Many index structures have been proposed in the recent decades. Examples include the R*-tree (Beckmann, 1990), X-tree (Berchtold, 1996), SR-tree (Katayama, 1997) and A-tree (Sakurai, 2000). V. Gaede and O. Gunther (1998) conducted a survey on multidimensional indices and a more recent survey has been presented by Y. Cui (2003). Some of these index structures (e.g., R*-tree) have been popularly used in researches and in several commercial database products (Informix, 2004, and Adler, 2001). Multidimensional indices such as Octrees (Samet, 1989) and Target Tree (Morse et al., 2005) are used in

computer-assisted surgery (CAS), in which two-dimensional images are used to guide surgical procedures. UB-tree (Bayer 1997 and Ramsak 2005) uses a space-filling curve to map a multidimensional universe to one-dimensional space with the goal of sorting multidimensional data. The multidimensional indices have been successfully introduced to the field of OLAP data. However, all existing multidimensional indices are designed for the AD range queries and there has been a lack of focus on PD range queries, which is popular in OLAP applications.

In the work (Berchtold, 2000), a higher-dimensional index is divided into several lower dimensional indices. The lower-dimensional indices are used instead of the higher-dimensional index and their search results are intersected to form the final result. It is still for AD queries and its purpose is to alleviate the well-known "dimensionality curse" problem. Even with respect to simply alleviating the dimensionality curse, their approach is confronted with the big challenge that the number of intermediate results may be very large and the intersection operation of the intermediate results may be very expensive.

This study is based on the R*-tree, 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. When the R*-tree is used for a range query, all of the nodes intersecting the query range are accessed and their entries have to be checked. Figure 2 shows the structure of the R*-tree, along with that of its nodes, and the expression of each MBR.

When the R*-tree is used for OLAP applications, original data-records may be placed in leaf-nodes, or elsewhere. In the latter case, pointers to records or tuples (record/tuple identifiers) are placed in the leaf-nodes. In the present study, both the R*-tree and the proposed Adaptive R*-tree, employ the latter case, in which values of non-index attributes have to be accessed via tuple/record pointers in the leaf nodes (refer to Figure 2).

Taking the R*-tree as an example, we briefly review how multidimensional indices are used to index OLAP data stored in a relational table. Let T be a relational table with m attributes, denoted by $T(A_1, A_2, \ldots, A_m)$. Attribute A_i ($1 \le i \le m$) has domain $D(A_i)$, a set of possible values for A_i. Each tuple t in T is denoted by $<a_1,a_2, \ldots,a_m>$, where a_i ($1 \le i \le m$) is an element of $D(A_i)$. When the R*-tree is used to index T, the attributes that can possibly be used in the possible query conditions are usually chosen as *index attributes*, on which the R*-tree is constructed. An order-preserving transformation (Jagadish, 2000) is necessary for transforming each of non-numeric index attributes to numeric values. After the necessary transformations, the index attributes form a multidimensional space, called *index space*, in which each tuple of T is mapped to one point. Every dimension of the index space is called *index dimension*, which corresponds to one of the index attributes. The terms *index attribute* and *index dimension* are used interchangeably hereinafter.

The term of *partial-record accesses* was used in the paper (Shao, 2004), in which a storage management architecture, called *Clotho*, is proposed to decouple the memory page layout from the non-volatile storage data organization. However, Clotho implements the partial-record accesses by a data placement strategy among the different levels of the memory hierarchy. That paper is on how to physically organize the original relational data. It allows for decoupled data layouts and different representations of the same table at the memory and storage levels. A multi-dimensional clustering method is also proposed (Bhattacharjee, 2003). However, it is also on the physical organization of the original relational data. In addition, there has been significant recent interest in column-oriented databases (or "column-store"), in which each attribute is stored separately, such that successive values of that attribute are stored consecutively on disk. This is in contrast to most common database systems that store relations in rows ("row stores") where values of different attributes from the same record are stored consecutively. This study is different from all of the methods mentioned here in that the present study is on multi-dimensional indexing and does not touch upon the physical organization of the original relational data. For example, dislike column-stores, we do not obtain the record-data through different "mini-pages". The present study is on pure indexing and its purpose is to improve the performance of searching the identifiers of the results.

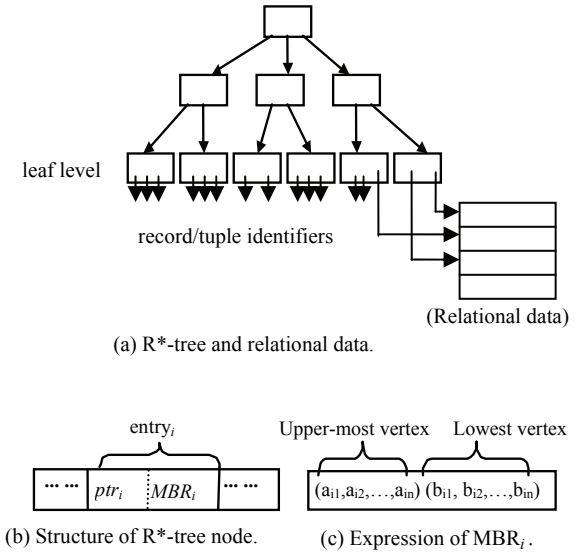


Figure 2 Structures of R*-tree, R*-tree node and MBR.

3 Naïve Methods for PD Range Queries

Three naïve methods for PD range queries are introduced in this section.

3.1 All-dimensional Index

In the existing multidimensional indices, every node contains the information of its entries at all of the index dimensions. Using one n-dimensional index in the entire n-dimensional index space, a PD range query using d ($d < n$) query dimensions can be evaluated by simply extending the query range in each of the other (n-d) irrelevant index dimensions to the entire data range.

The advantage of this approach is that only one index is necessary. However, each node contains n-dimensional information of all its entries, but only d-dimensional information is necessary for a PD range query using d ($d < n$) query dimensions. This means that a great deal of unnecessary information, i.e., the information in the irrelevant (n-d) dimensions, also has to be read from disk, which obviously degrades the query performance.

3.2 Multi-Btree

In this naïve approach, using the projections of all the tuples to each of the index dimensions, one B-tree (or a variant thereof) is constructed. For a PD range query, 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.

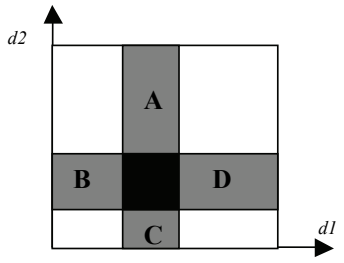


Figure 3 An PD range query using multi-Btree.

Figure 3 is an example of a PD range query evaluated using the multi-Btree, where the two dimensions (d_1 and d_2) are used as query dimensions. In this example, two B-trees constructed on the dimensions of d_1 and d_2 are used. In Figure 2, the thick shadow region is the given query range. At first, two range queries are evaluated on the two B-trees, respectively. All of the objects located in the vertical shadow region and the horizontal shadow region are reported as intermediate results, R_1 and R_2. The final result of this PD range query is given by $R_1 \cap R_2$.

The disadvantage of multi-Btree is that queries on different B-trees are evaluated independently without any mutual reference. That is, during searching on each B-tree, the algorithm cannot realize the query ranges in the other index dimensions. Thus, unnecessary checks such as those in regions A, B, C, and D in Figure 2 cannot be pruned. Many unnecessary investigations are thus performed and a great deal of irrelevant information is read from disk and, many unnecessary intermediate results are reported by each B-tree. In addition to the unnecessary I/O operations, the intersection phase is also very time-consuming. Considering 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 the query range in each of the four query dimensions is 1/6 of the entire data range. In this case, the final result has only $1{,}000{,}000/6^4 = 771$ tuples. However, the query result on each of the four B-trees has 1,000,100/6=166,666 tuples! The total number of intermediate results is 666,666! Even if the query range in each of the query dimensions is decreased to 1/10 of the data range, the above three numbers are 100, 100,000 and 400,000, respectively. That is, although only 100 tuples are in the final result, up to 400,000 objects are reported as intermediate results. Another disadvantage of the multi-Btree approach is that managing/updating many disk-resident B-trees requires additional costs.

3.3 Multiple Multidimensional Indices

Why do not we prepare one index for each possible combination of query attributes? Since there are probably many actual combinations of attributes used in the possible PD range queries, it is not always feasible in applications with large datasets to build one index for each of such possible combinations of index attributes, because (1) numerous indices have to be constructed and managed, (2) many attributes are repeatedly included in different indices, which is too space-consuming for large datasets and results in a large update cost, and (3) the combinations of index attributes that are possibly used in the PD queries are often unpredictable. Note that, there are a total of (2^n-1) different combinations for n index attributes. Thus, this naïve method is not considered in this study.

4 Adaptive R*-tree

Our proposal for PD range queries, called Adaptive R*-tree (AR*-tree), is presented in this section.

4.1 General Structure

The key concept of the AR*-tree is that, instead of n-dimensional R*-tree nodes, the AR*-tree consists of *node-groups*, each of which have n one-dimensional nodes. Each node in every node-group holds information of its entries in only one dimension and each node-group having n one-dimensional nodes holds the information in all the n dimensions. Note that each R*-tree node holds information of its entries in all of the dimensions. The general structure of the AR*-tree is depicted in Figure 4.

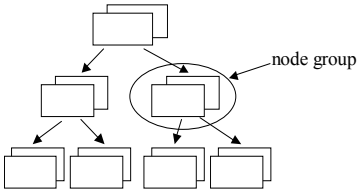


Figure 4 General structure of AR*-tree.

Here we note that the AR*-tree is not obtained by simply dividing each of the R*-tree nodes into a node-group. This is because that 1) the size of each node (not node-group) in the AR*-tree and the size of each R*-tree node are the same as one disk page, and different nodes in (even the same) node-groups are stored in a separate disk pages from each other, and 2) *n*-dimensional information of each entry is contained in every R*-tree node. However, in the AR*-tree, only one-dimensional information of each entry is contained in every node. Thus the number of entries in every node of the AR*-tree is much larger than that in each of the R*-tree nodes. In this way, the fact is that several R*-tree nodes correspond to one node-group in the AR*-tree.

Figure 5 shows one 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 every entry in R*-tree nodes includes MBR information in all of the dimensions, every entry in the nodes of the AR*-tree includes only one-dimensional information, which means every entry only corresponds to an MBR edge. The term *entry of node-group* is used hereinafter, which refers to the set of entries having the same index distributed in all the different nodes of one node-group. One entry of each index node-group corresponds to a complete MBR in the index space, such as $Xentry_i$+$Yentry_i$ in Figure 6. In Figure 5, all of the entries in an ellipse form an entry of this node-group. Figure 6 shows an example of entries in the 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.

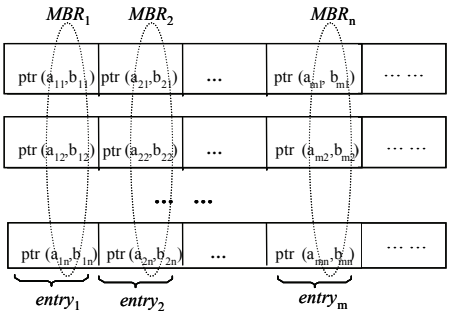


Figure 5 General structure of node-groups in an *n*-dimensional AR*-tree.

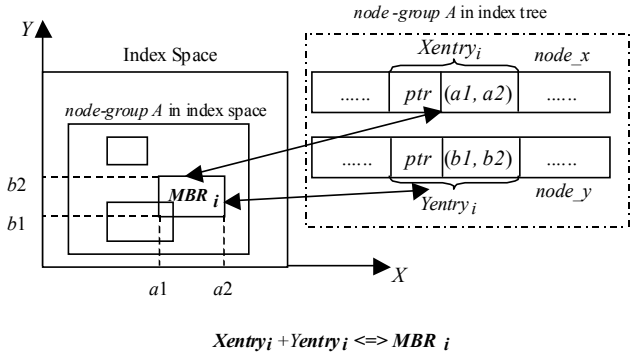


Figure 6 Example of entries of node-groups in a two-dimensional space.

The question may arises that whether or not the number of nodes in the AR*-tree increase greatly. The answer is *No*! This is because the maximum number of entries in every node of the AR*-tree increases

greatly since only one-dimensional information is necessary. According to our experiments, the number of nodes slightly increases by less than 10% (see Table 3 in Section 6).

4.2 Algorithms of AR-tree*

Table 1 Algorithm **RangeQuery.**

```
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 the query dimensions**) Then
        If (node-group is not at leaf) Then
            RangeQuery (rect, e.child);
              //e.child: the child node-group corresponding to e
        Else result ← result+e
EndFor
End
```

Note that

*) An entry of a node-group includes all of the entries with the same index in the different nodes of this node-group. That is, it corresponds to a complete MBR.

**) When an entry of one node-group (one MBR) is checked to determine whether it intersects the query range, only the nodes corresponding to the query dimensions are accessed and the other nodes even in the same node-group are skipped. Moreover, in many cases, even not all of the nodes in the query dimensions need to be checked. This is because this investigation is made dimension by dimension, so it can stop if the current entry is found not to intersect the query range in the current query dimension.

The insert algorithm of the AR*-tree is a naïve 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 dimensions. If some node-group must be split, then all of its nodes have to 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 inserted to the AR*-tree again. That is, all of the nodes in each node-group must be born and die simultaneously. A range query algorithm for the AR*-tree is shown in Table 1.

Starting with the root node-group, each entry of the current node-group needs to be checked as to whether its MBR intersects the query range. 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 child node-groups. Otherwise, the current node-group is skipped.

The structure of the AR*-tree and the above search algorithm guarantee that it can be applied for PD range queries with any combinations of the index dimensions and that only the information in the relevant dimensions needs to be visited.

Now let us see the dynamic performance of the AR*-tree. Because more nodes need to be accessed, the AR*-tree costs longer time than the R*-tree for data insertions and deletions. According to our investigations, for a six-dimensional dataset having 200,000 data, the time cost for one insertion operation on the AR*-tree is about 4.0~8.3 times of that on the R*-tree if no overflow happened (no node-splitting). If node-splitting happened, it becomes about 7.5~11.7 times. And, for one deletion operation, the AR*-tree costs about 4.5~7.8 times longer than the R*-tree. Thus, the dynamic performance of the AR*-tree is much worse than that of the R*-tree, which means that the AR*-tree is not oriented to applications with frequently changed data. Anyway, OLAP data do not change often. Thus, the query performance is the most important performance factor.

5 Analytical Comparison

Table 2 Symbols and their descriptions.

n	Dimensionality of the entire index space
d	Number of the query dimensions
S	Volume of the entire index space
S_q	Volume of the extended query range of PD range query[♣]
M_r	Maximum number (capacity) of entries in each leaf node of R*-tree
M_g	Maximum number (capacity) of entries in each leaf node-group of AR*-tree
N_l	Number of leaf nodes in the case of R*-tree
N_g	Number of leaf node-groups in AR*-tree

In this section, under the assumption of uniform distribution, the performance of the AR*-tree for PD range queries is mathematically analyzed by comparing with the naïve methods. The number of accessed leaf nodes is estimated and compared since it is an important factor with regard to query performance (Hjaltason, 1999), especially for large datasets. The symbols used in this section are described in Table 2.

5.1 AR-tree vs. R*-tree*

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

$$R_l = \frac{S_q}{S} \times N_l \,. \tag{1}$$

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

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

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 only one-dimensional information of each entry is contained in each of AR*-tree nodes. In addition, considering that the clustering algorithms (insert algorithms) of the R*-tree and the AR*-tree are the same, we have

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

In each accessed node-group, at most[1] 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

$$\begin{aligned} AR_l &\le AR_g \times d \approx \frac{S_q}{S} \times N_g \times d \\ &\approx \frac{S_q}{S} \times \frac{1}{n} \times N_l \times d = \frac{d}{n} \times R_l < R_l \qquad (when\ \ d < n). \end{aligned} \tag{2}$$

Equation (2) 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. Even if $d = n$ (AD queries), then the number of accessed leaf nodes may be approximately the same and it is also possible that $AR_l < R_l$. Moreover, for a fixed n, the lower the number of query dimensions is, the bigger the advantage of the AR*-tree compared to the R*-tree is.

♣ The extended query range of one PD range query refers to the n-dimensional range obtained by extending the unused (n-d) dimensions to their entire data ranges.

[1] Although this query is relevant to d dimensions, investigation of each node-group-entry may stop midway (see Sections 4.2 and 6.3.1).

Equation (2) can be explained as follows. Because 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.

5.2 AR-tree vs. multi-Btree*

For simplicity, in addition to the above-mentioned assumptions, we also assume that the entire index space and the given query ranges are hyper cubes. In addition, the following notations are also used herein.

p' : side length of the given query range.

p : side length of the entire index space.

l : number of leaf nodes in each B-tree of Multi-Btree. We assume that every B-tree has the same number of leaf nodes. This assumption is reasonable considering the above assumptions.

Let us first investigate the Multi-Btree. The average number of leaf node accesses in each relevant B-tree, N_b, can be given by $N_b = l \times p'/p$. Since a total of d B-trees must be used independently, the total number of leaf node accesses, TN_b, can be given by

$$TN_b = d \times \frac{p'}{p} \times l \,. \qquad (3)$$

Then, let us consider the case of the AR*-tree. The size of the d-dimensional query range is $(p')^d$, and the size of the extended query range (see Table 2), S_q, can be given by

$$S_q = (p')^d \times p^{n-d} \,.$$

Considering that the size of the entire index space is p^n, the average number of accessed leaf node-groups, AR_g, can be given by

$$AR_g = \frac{S_q}{p^n} \times N_g = \frac{(p')^d \times p^{n-d}}{p^n} \times N_g \,.$$

As mentioned above, at most d nodes in each node-group are visited. Thus, the average number of accessed leaf nodes, AR_l, can be given by

$$AR_l \le AR_g \times d = \frac{(p')^d \times p^{n-d}}{p^n} \times N_g \times d = d \times \left(\frac{p'}{p}\right)^d \times N_g. \qquad (4)$$

In order to compare Equations (3) and (4), we need to compare l and N_g. Since each node-group of the AR*-tree contains one one-dimensional node for each index dimension, we can estimate that $N_g \approx l$.

In addition, since $p' < p$, we have

$$\left(\frac{p'}{p}\right)^d < \frac{p'}{p} \qquad (when \;\; d > 1) \,.$$

Comparing Equations (3) and (4), we can see that

$$AR_l < TN_b \quad (when \;\; d > 1) \,. \qquad (5)$$

Another observation is that, the advantage of the AR*-tree over the multi-Btree approach becomes better as the query range becomes smaller and the number of query dimensions becomes larger.

Note that, the above mathematic comparison is only a rough analysis. Anyway, we believe that it can help the readers understand our proposal.

6 Experiments

The experimental environment and the experimental results are presented in this section.

6.1 Datasets and Experimental Environment

A synthetic dataset with Zipf distribution (like the studies (Hong, 2001 and Zhang, 2001) and TPC-H benchmark dataset are used to examine the behavior of the AR*-tree for range queries.

***6D-Zipf200000*:** A table having 200,000 tuples with six floating index attributes, each of which follows a Zipf distribution with a constant of 1.5 like that in (Zhang, 2001).

***TPC-H data*:** The table of LINEITEM in the TPC-H benchmark is used because it has the most (16) attributes with all data types (floating, integer, date, string, Boolean) among the eight tables of the TPC-H benchmark. The table is generated with 200,000 tuples. Six out of the 16 attributes are chosen as index attributes, including SHIPDATE (date), QUANTITY (floating), DISCOUNT (floating), SHIPMODE (character string), SHIPINSTRUCT (character string), and RETURNFLAG (character string), because these attributes are often used as query attributes.

Although the price of main memory chips continues to drop and the size of main memory in many systems has increased greatly in recent years, indices (especially their leaf nodes) for large relational datasets still tend to be stored in secondary storage. That is, the I/O cost is still the performance bottleneck for many indexing systems with large datasets. Thus, many works on the disk-resident indexing systems use the number of node accesses, especially the number of leaf node accesses, as an important factor of search performance.

The page size of our system is 4096 bytes. Query performance is measured in term of the number of leaf node accesses. PD range queries are tested with different numbers of query dimensions, ranging from 1 to 6. B^+-tree is used in the multi-Btree.

6.2 Performance Comparison using 6D-Zipf200000

Without loss of generality, the query ranges in all the query dimensions are set to be equal. These ranges are varied from 10% to 100% in increments of 10%. The range query for the range of the same size is repeated 100 times with random locations, and the averages are presented.

Table 3 Parameters of indices.

	Adaptive R*-tree	R*-tree	Multi-Btree
Capacity of leaf nodes	340	77	340
Height	3	4	3
Total number of leaf nodes	4848 (808 groups)	4439	6777

The parameters of the indices in our experiments are shown in Table 3. The total number of the lead nodes of the multi-Btree (6777) is the total number of leaf nodes in all of B-trees. We also can find that the height of the AR*-tree is less than that of the R*-tree. Why? If we compare the R*-tree nodes and the AR*-tree node-groups, we will find each AR*-tree node-group can hold more entries than each R*-tree node. This is because each node in AR*-tree contains only one-dimensional information. In other words, the number of nodes (not node-groups) in each level of the AR*-tree is larger than that in each level of the R*-tree.

The experimental results are depicted in Figure 7 through Figure 12, where the X-axis represents the *side length of the query range* in each dimension and the Y-axis represents *the number of leaf node accesses.* In these figures, the number of leaf node accesses in the case of multi-Btree indicates the total number of the leaf node accesses for the queries on all of the relevant B^+-trees.

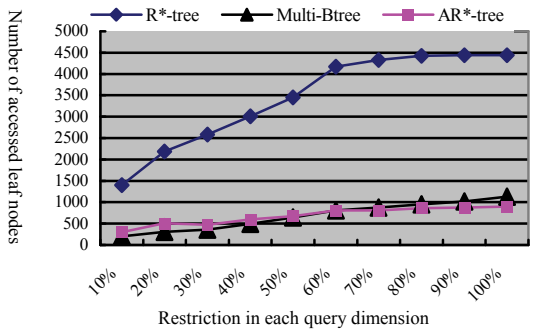


Figure 7 Number of query dimensions is one.

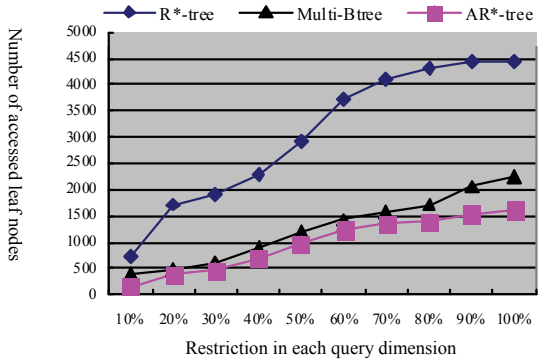


Figure 8 Number of query dimensions is two.

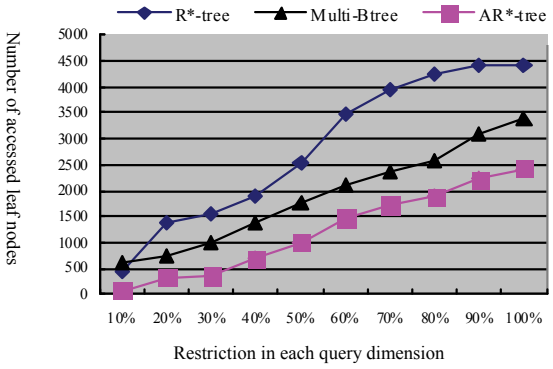


Figure 9 Number of query dimensions is three.

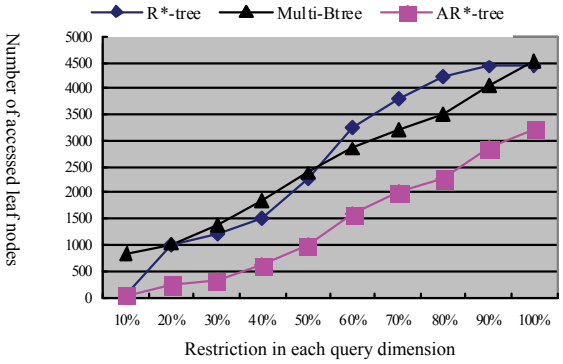


Figure 10 Number of query dimensions is four.

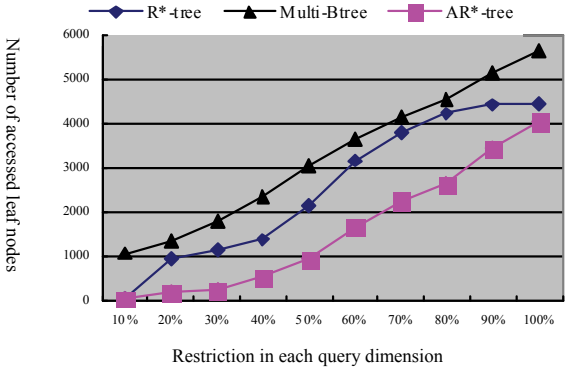


Figure 11 Number of query dimensions is five.

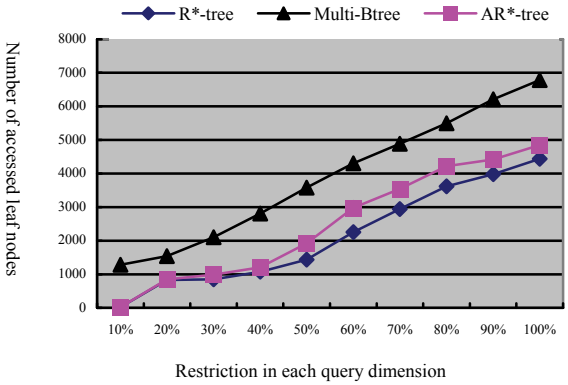


Figure 12 Number of query dimensions is six.

The result shows that

(1) For $d = 1$ (d is the number of query dimensions), the performance of PD range queries on the AR*-tree is slightly worse than that on multi-Btree, but it is far better than that on the R*-tree.

(2) From d=2, the performance of PD range queries on the AR*-tree begins to outperform that on multi-Btree and remains better than that on the R*-tree. In addition, as d increases, the difference in performance between the AR*-tree and the multi-Btree becomes larger. However, the performance advantage of the AR*-tree compared to the R*-tree becomes weaker as d increases.

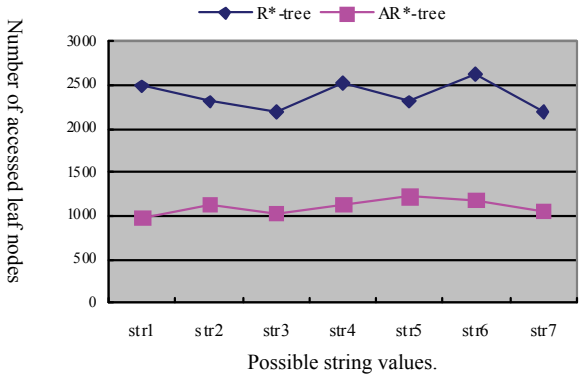
(3) The performance of PD range queries on the R*-tree becomes better as the number of query dimensions increases. This is because the search region can be limited in more dimensions as the number of query dimensions increases.

6.3 Performance Comparison using TPC-H Data

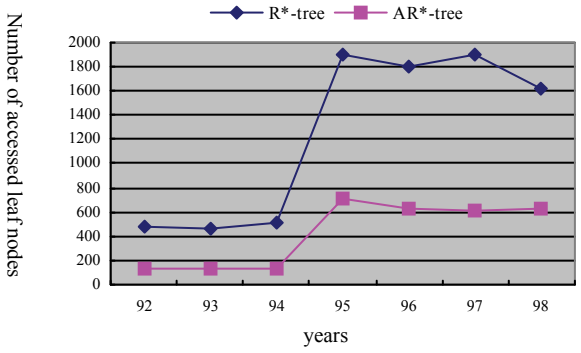
Without loss of generality, the query ranges in all floating query dimensions are set to be equal. These ranges are varied from 10% to 100% in increments of 10%. The query range in the date dimension is set to be one year (unit: day), and that in each category dimension is set to be one of the possible values, chosen randomly. The range query for the range of the same size is repeated 100 times with random locations, and the averages are presented.

6.3.1 AR-tree vs. R*-tree*

The search performance varies greatly with different query attributes and different combinations of query attributes because the attribute values have very different distributions in different dimensions. As examples in the case that only one attribute is used in the query condition, the experimental results with SHIPDATE (date type) and SHIPMODE as query attribute are shown in Figure 13(a) and (b), respectively. In Figure 13(a), the X-axis lists the possible values, and in Figure . 13(b), the X-axis indicates seven different years.



(a) Range query using SHIPMODE.



(b) Range query using SHIPDATE.

Figure 13 Performance comparison using one attribute.

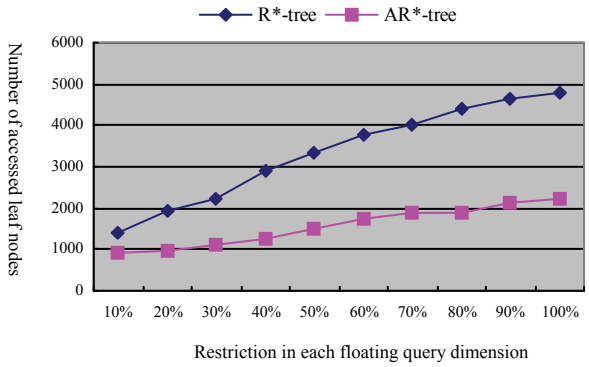


Figure 14 Search performance comparison having two query attributes.

For the queries with more than one query attribute, the number of possible combinations of the query attributes is too great to enumerate herein. All of the experiments using different combination of query dimensions indicate that the AR*-tree has better performance than the R*-tree. Figures 14 and 15 are examples. Figure 14 has DISCOUNT (floating) and QUANTITY (floating) as query attributes, where the X-axis indicates the restriction in each query dimension. Figure 15 has six query dimensions, where the X-axis indicates the restriction in each query dimension having a floating data type.

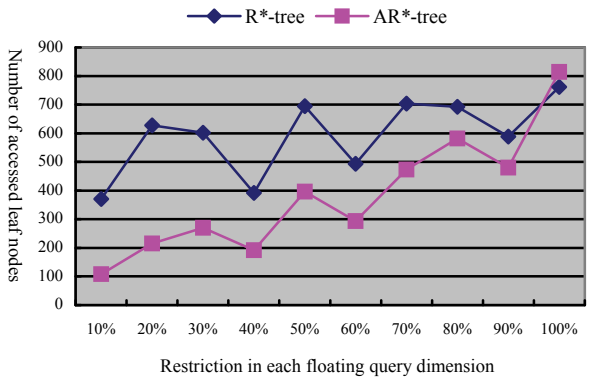


Figure 15 Search performance comparison having six query attributes.

Note that the number of accessed leaf nodes in Figure 15 is not equal to the total number of leaf nodes when the X-axis extends to 100%. This is because the query ranges in the date dimension and the category dimensions are not 100%.

In our experiments, we found that the AR*-tree still has better search performance in some cases for AD queries. Why? Intuitively, all of the nodes in the visited node-groups have to be accessed for AD range queries. Actually, however, this is not true. This is because, as mentioned in Section 4.2, if the AR*-tree is used for a d-dimensional query ($d = n$ in this case) then, the fact that this query is relevant to all the n dimensions does not mean that all n nodes in each accessed node-group have to be visited. Investigation may stop midway, and the remaining nodes in the current node-group can be skipped. Let us examine this case in detail through Figure 15, an example in a three-dimensional index space.

The query in Figure .16 is an AD (all-dimensional) range query because the current query has three query dimensions. Since the MBR of the current node-group intersects the query range, the entries (the dotted cubes) in 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, although the current query is an AD range query. Note that, in

this example, if the X-Z plane is checked first then, the node corresponding to the Y-axis also can be skipped. More importantly, for higher-dimensional spaces, because the MBRs (entries) in each node-group become sparser, it generally becomes possible to skip more nodes. This discussion indicates an important feature of the AR*-tree that, not all of the nodes (of the accessed node-groups) in the query dimensions have to be accessed. Even for the AD range queries, not all of the nodes in the accessed node-groups have to be accessed.

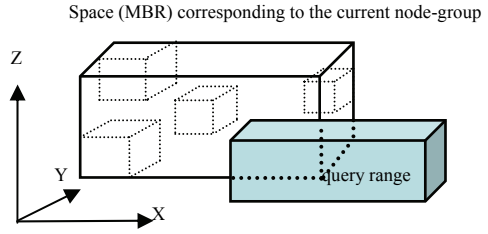


Figure 16 Example in 3-dimensional index space.

6.3.2 AR-tree vs. multi-Btree*

This comparison is very complex. B^+-trees built on the floating dimensions or on the date dimension have many more nodes than those built on category dimensions because category dimensions have only few possible values, in which B^+-trees can be built with buckets. Thus, for range queries with only category attributes, the multi-Btree has far fewer leaf node accesses than the AR*-tree. However, for queries with the floating dimensions or the date dimension and for queries in which the number of query dimensions is more than one, the AR*-tree has fewer leaf node accesses, where the number of leaf node accesses of the AR*-tree is approximately 21%~73% of that of multi-Btree. Still, as the number of query dimensions increases, the advantage of the AR*-tree become clearer. No Figure ures of the experimental results are presented herein, because of the large number of possible combinations of query dimensions and the space limitation.

Moreover, when the multi-Btree is used for range queries, as mentioned in Section 4.2, a large number of intermediate results are generated, especially for queries with the category attributes. These intermediate results have to be intersected to obtain the final results, which is very expensive. For example, using SHIPMODE, SHIPINSTRUCT and QUANTITY as query dimensions, where the query values in SHIPMODE and SHIPINSTRUCT are chosen randomly from the possible values and the query range in QUANTITY is set to 20%. The test is repeated 10 times, we found that the final result has 672 tuples on average. However, the intermediate result contains 22,169 tuples from the B^+-tree on SHIPMODE, 63,433 tuples from the B^+-tree on SHIPINSTRUCT, and 34,728 tuples from the B^+-tree on QUANTITY. We then implemented an intersection operation with the help of AVL-tree. If the time cost of the intersection operations is considered, we found that, except in cases having one query dimension, in which no intersections are needed, the time cost of the AR*-tree is 3.6 - 14.0 times smaller than that of the multi-Btree. The results of numerous experiments indicate that the multi-Btree is not well suited to multidimensional queries. Besides the intersection operation is very expensive, the tuples of the result set must be fetched randomly from the disk.

7 Conclusion and Future Work

In this paper, we showed that the existing multidimensional indices are not appropriate for OLAP environment, in which partially-dimensional (PD) range queries are popular and should be supported effiently. And then, based on the R*-tree (as an example), we proposed an index structure (called AR*-tree) consisting of groups of *n* one-dimensional nodes. In each node of the AR*-tree only the information in one dimension is contained. If the AR*-tree is used for the PD range queries in OLAP environments, only the nodes corresponding to the query dimensions need to be visited. Thus, the AR*-tree can efficiently deal with PD range queries with any combination of the query dimensions. A simple

mathematical analysis and many experiments indicate that the proposed method has better search performance for PD range queries than the naïve methods.

Although the R*-tree was used in the present paper, many other hierarchical multidimensional indices (e.g., the members of R-tree family) can also be employed. The basic requirements for applying the proposed method to an index are that (1) the index is based on MBR (Minimum Bounding Rectangle) and that (2) in the index the region covered by every node must be completely contained within the region of its parent node.

We also discussed the dynamic performance of the AR*-tree. We found that the AR*-tree is not oriented to dynamic data, although it also can handle the updating operations (insertions and deletions). Because the data in most OLAP applications do not changed often, the query performance is the most important.

Our future works in this study include 1) examining the behaviors of the AR*-tree using real datasets and 2) implementing the "group-by" operation efficiently on the AR*-tree by using the feature of the AR*-tree that each node has only one-dimensional information.

Acknowledgement

The authors would like to thank Mr. Zhibin Wang, who conducted some of the experiments.

References

Adler, D.W. (2001) "DB2 Spatial Extender-Spatial data within the RDBMS". Proc. VLDB Conf., pp 687-690.

Bayer, R. (1997) "The universal B-tree for multidimensional Indexing: General Concepts" , Proc. WWCA Intl. Conference.

Beckmann, N. and Kriegel, H. (1990) "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles", Proc. ACM SIGMOD Intl. Conf., pp.322-331.

Berchtold, S. Bohm, C. (2000) "Optimal Multidimensional Query Processing Using Tree Striping", Proc. 2nd intl. Conf. on Data Warehousing and Knowledge Discovery (DaWak), pp. 244-257.

Berchtold, S. Keim, D. and Kriegel, H. (1996) "The X-tree: An Index Structure for High dimensional data", Proc. The 22nd VLDB Intl. Conf., pp.28-39.

Bhattacharjee, B. et al. (2003) "Efficient Query Processing for Multi-Dimensionally Clustered Tables in DB2", Proc. VLDB Intl. Conference.

Cui, Y. (2003) "High-dimensional Indexing", Lecture Notes in Computer Science (LNCS), Vol. 2341 (Monograph).

Cuzzocrea, A. (2007) "An OLAM-Based Framework for Complex Knowledge Pattern Discovery in Distributed-and-Heterogeneous-Data-Sources and Cooperative Information Systems", Proc. DaWaK, LNCS 4654, pp. 181-198.

Feng, Y. and Makinouchi, A. (2004) "Improving Query Performance on OLAP-Data Using Enhanced Multidimensional Indices", Proc. ICEIS Intl. Conf., pp.282-289.

Feng, Y. and Makinouchi, A. (2006) "Dfficient Evaluation of Partially-Dimensional Range Queries Using Adaptive R*-tree", Proc. DEXA, LNCS 4080, pp. 687-696.

Gaede, V. and Gunther, O. (1998) "Multidimensional Access Methods", ACM Computing Surveys, Vol.30, No.2, pp.170-231.

Guttman, A. et al. (1984) " R-Trees: A Dynamic Index Structure for Spatial Searching". Proc. ACM SIGMOD Conference, pp. 47–57, Boston, MA.

Hjaltason, G. R. and Samet, H. (1999) "Distance Browsing in Spatial Database", ACM Transactions on Database Systems, Vol.24, No.2, pp. 265~318.

Hon, S. Song, B. and Lee, S. (2001) "Efficient Execution of Range-Aggregate Queries in Data Warehouse Environments", Proc. the 20th Intl. Conf. on CONCEPTUAL MODELING.

Hong, S. Song, B. and Lee, S. (2001) "Efficient Execution of Range-Aggregate Queries in Data Warehouse Environments", Proc. 20th international Conference on CONCEPTUAL MODELING (ER 2001), pp.299-310.

Informix (2004) Informix Spatial DataBlade Module, IBM (ww306.ibm.com/software/ data/ Informix /blades/spatial/rtree.html).

Jagadish,H. V. Koudas, N. and Srivastava, D. (2000) "On Effective Multi-Dimensional Indexing for Strings", Proc. ACM SIGMOD Conf., pp.403-414.

Katayama, N. and Satoh, S. (1997) "The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries", Proc. ACM SIGMOD Intl. Conf., pp.369-380.

Kothuri, R.K.V., Ravada, S. and Abugov, D. (2002) "Quadtree and R-tree Indexes in Oracle Spatial: A Comparison Using GIS Data", Proc. SIGMOD Conf. pp 546-557.

Kotidis, Y. and Roussopoulos, N. (1998) "An Alternative Storage Organization for ROLAP Aggregate Views Based on Cubetrees", Proc. ACM SIGMOD Conference.

Leutenegger, S. T. et al. (1997) “STR: A Simple and Efficient Algorithm for R-tree Packing”, Proc. ICDE Intl. Conf., pp. 497-506.

Markl, V. Ramsak, F. and Bayer, R. (1999) “Improving OLAP Performance by Multidimensional Hierarchical Clustering”, Proc. IDEAS Intl. Symposium, pp165-177.

Markl, V. Zirkel, M. and Bayer, R. (1999) “Processing Operations with Restrictions in Relational Database Management Systems without External Sorting”, Proc. ICDE Intl. Conf., pp 562-571.

Morse, M.D. Patel, J.M. and Grosky, W. I. (2005) “Efficient Evaluation of Radial Queries using the Target Tree”, Proc. BMDE workshop in conjunction with ICDE pp. 130-137.

Ramsak, F. and Markl, V. (2000) “Integrating the UB-tree into a Database System Kernel”, Proc. VLDB Intl. Conf., pp. 263-272.

Roussopoulos, N. and Roussopoulos, M. (1997) “Cubetree: Organization of and Bulk Incremental Updates on the Data Cube”, Proc. ACM SIGMOD Intl. Conference.

Samet, H. (1984) “The Quadtree and Related Hierarchical Data Structures”, ACM Computer Survey, 16(2), pp. 187-260.

Sakurai, Y. et al. (2000) “ The A-tree: An Index Structure for High-Dimensional Space Using Relative Approximation”, Proc. The 26th VLDB Intl. Conf. , PP. 516-526.

Samet,H. (1989) “ Implementing Ray Tracing with Octtrees and neighbor Finding”, Computers & Graphics, PP 445-460.

Shao, M. et al. (2004) “Clotho: Decoupling Memory Page Layout from Storage Organization”, Proc. VLDB Intl. Conf., pp. 696-707.

White, D. A. and Jain, R. (1996) “Similarity Indexing with the SS-tree”, Proc. ICDE Intl. Conf. pp. 516-523.

Zhang, C. et.al. (2001) “On Supporting Containment Queries in Relational Database Management Systems”, Proc. SIGMOD Intl. Conf., pp. 425-436