2004－DBS－134　(I)　(5)
2004／7／13

Towards Range Queries with Partial Dimensions in OLAP Applications

Yaokai Feng, Zhibin Wang, Akifumi Makinouchi, and Ryu Hiroshi
Graduate School of Information Science and Electrical Engineering, Department of Intelligent Systems,
Kyushu University.
{fengyk,akifumi}@is.kyushu-u.ac.jp

Abstract

Multidimensional indices are very helpful to improve query performance on multidimensional data including relational data in ROLAP systems. The existing multidimensional indices are directed to "queries with all dimensions" (called QAD in this study). That is, the dimensions used in each query are all the dimensions in the whole space. However, in many applications, especially in OLAP-related ones, the queries may be only with some (partial) dimensions (not all) of the whole space, which is called QPD (Queries with Partial Dimensions) in this study. This study focuses on range queries with partial dimensions (RQPD), which is popular in OLAP applications. If the existing multidimensional indices are used in RQPD, the dimensions unused in the query are thought as spanning the whole data ranges, which often lead to not-good search performance. In these cases, certainly, we also can construct many indices with all the necessary combination of dimensions. However, this is very space/time-consuming since many indices have to be constructed and some dimensions may be used many times in different indices, which is not always feasible. In this study, we propose a novel solution to RQPD problem. With our solution, only one index is necessary to such applications. The performance of our solution is discussed in detail and is examined by experiments.

1. Introduction

There is increasing requirement for processing multidimensional range queries on business data usually stored in relational tables. For example, Relational On-Line Analytical Processing (ROLAP) in data warehouse is required to answer complex and various types of range queries on large amount of such data. Typical examples include "**Select** sum (EXTENDEDPRICE* DISCOUNT) **From** *LINEITEM* **Where** QUANTITY ≤ 25 and 0.1 ≤ DISCOUNT ≤ 0.3 and 2001-01-01 ≤ SHIPDATE ≤ 2001-12-31", where *LINEITEM* is a table having sixteen attributes used in TPC-H benchmark [1]. In this query, three attributes QUANTITY, DISCOUNT, and SHIPDATE form the range condition. In order to improve good performance for such multidimensional range queries, multidimensional indices are helpful [2,3], in which the tuples are clustered among the leaf nodes to restrict the nodes to be accessed for queries.

Many index structures have been proposed in the last two decades. Examples include R*-tree [4], X-tree [5], SR-tree [6], and so on.. Some of them (e.g., R*-tree) have been successfully used in many researches on multidimensional data (GIS data, multimedia data, etc.) and OLAP data [7]. In this study, our proposal is based on the R*-tree, since the R*-tree have been used in many researches and is regarded as one of successful hierarchical index structures. Here, we want to note that many other hierarchical indices also can be used in this study.

The existing multidimensional indices are directed to "queries with all dimensions" (called QAD in this study). That is, the dimensions used in each

query are all the dimensions in the whole space. However, in many applications, especially in OLAP-related ones, the queries may be only with some (partial) dimensions (not all) of the whole index space, which is called QPD (Queries with Partial Dimensions) in this study. This study focuses on range queries with partial dimensions (RQPD), which is popular in OLAP applications. If the existing multidimensional indices are used in RQPDs, the dimensions of the index space that are not used in the query are thought as spanning the whole data ranges, which often lead to not-good search performance. In these cases, certainly, we also can construct many indices with all the necessary dimension combination for each kind of QPDs. However, this is very space/time-consuming since many indices have to be constructed and managed, and some dimensions may be used many times in different indices. Actually, this is not always feasible. In Section 3, QAD and QPD will be discussed in detail. In this study, we propose a novel solution to RQPD issue. With our solution, only one index is necessary to such applications. Our proposal and related algorithms will be presented in Section 4. Our solution is discussed in Section 5 and is examined by experiments in Section 6. Section 7 is current conclusion and future work of this study.

2. Indexing OLAP Data Using R*-tree

Now, we briefly recall how the traditional R*-tree to index business data stored in a relational table and give some terms. Let T be a relational table with n attributes, denoted by T(A_1, A_2, ⋯, A_n). Attribute Ai ($1 \le i \le n$) 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, ⋯,a_n>, where a_i ($1 \le i \le n$) is an element of D(A_i). When the R*-tree is used in T, some of the attributes are usually chosen as *index attributes*, which are used to build the R*-tree. For simplification of description, it is supposed without loss of generality that the first k ($1 \le i \le n$) attributes of T, <A_1,A_2, ⋯ ,A_k>, are chosen as index attributes. Since the R*-tree can only deal with numeric data, an order-preserving transformation is necessary for each non-numeric index attributes. After necessary transformations, the k index attributes form an k-dimensional space, called *index space*, where each tuple of T corresponds to one point. It is not difficult to find such a mapping function for Boolean attributes and date attributes. For Boolean data, “True” and “False” can be mapped onto 1 and 0, respectively, if “True” > “False” is assumed forcedly. This ordering has no practical problems, because the predicate of “equality” such as “A = True” or “A = False” is the only predicate pattern for the Boolean attribute. Although implementation of “date” depends on DBMS, typical example of “date” in TPC-H benchmark consists of three integers representing year, month, and day. A simple function to get a numeric value for a “date” is to use the number of days from some reference date to this ``date". In this paper, the day of Jan. 1, 1900 is used as the reference day, that is, the number of days from Jan. 1, 1900 to Apr. 5, 1998 is used to represent the date of Apr. 5, 1998. Anyway, it is not easy to map an arbitrary character string to a unique numeric data. The work [8] proposes an efficient approach that maps character strings to real numeric values within [0,1], where the mapping preserves the lexicographic order. This approach is also used in this study to deal with attributes of character string.

3. QPD and RQPD

As mentioned above, QPD means such queries that used only partial dimensions in the whole index space and this study focus on the range QPD (or say RQPD), which is very popular in OLAP applications. In contrast to QPD, the queries used all the dimensions in the index space are called QADs. Fig.1 are examples of QAD and QPD.

In Fig.1, the shaded regions are query range. (a) and (b) are range queries like “WHERE a_1 < X < a_2”, where only the attribute X is used in the queries and the dimensionalities of the index spaces are two and three, respectively. (c) is range query like

"WHERE $a_1 < X < a_2$ AND $c_1 < Z < c_2$", where two dimensions of the whole 3-dimensional index space are used in the query. The cases of (a), (b) and (c) are QPDs since only partial dimensions of the whole index space are used in the queries. Certainly, the case of (c) is a QAD since all the dimensions of the whole index space are used in the query like "WHERE $a_1 < X < a_2$ AND $b_1 < Y < b_2$ AND $c_1 < Z < c_2$".

Let us see an instance using Table 1, where QPDs are necessary.

Table 1. A relational table with 8 attributes.

A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8

Table 1 has 8 attributes A_1~A_8. And, the practical attribute combinations possibly used in queries are {{A_1},{A_2},{A_3},{A_4},{A_5},{A_6}, {A_1,A_2},{A_2,A_4},{A_1,A_3,A_5},{A_2,A_4,A_6}}. Thus, the index attributes include A_1 ~ A_6.

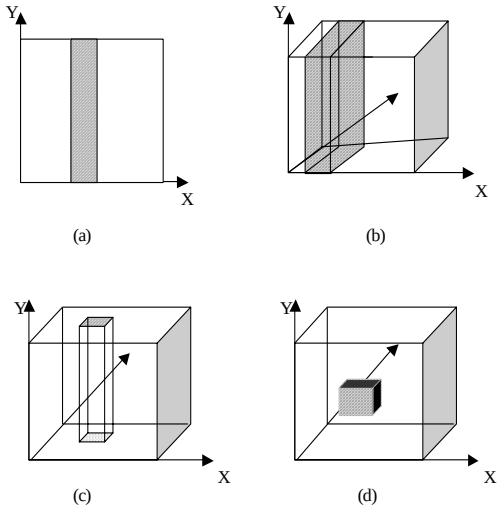


Fig. 1. QPD and QAD

For this example, it is certainly not feasible for large databases that one index is built for each possible combination of query attributes since so many indices need to be constructed and managed, and, in these indices, there are many attributes used repeatedly. One naïve idea is to build one multidimensional index using the six possible index attributes of A_1 ~ A_6. For each practical QPD, e.g., the queries using the index attributes of A_2 and A_4 only, the query ranges in the other four index attributes (i.e., A_1, A_3, A_5, A_6) are thought as the whole data ranges in such attributes.

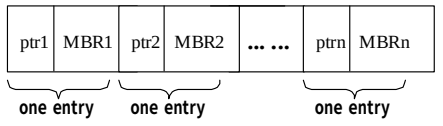
The above examples including Fig.1. and Table 1 are on query range, on which this study focuses since the they are popular in OLAP applications. The range QPD is denoted by RQPD in this study.

4. Our proposal: Array-node R*-tree and its Algorithms

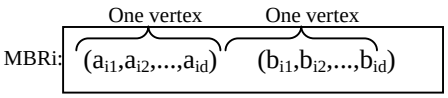
4.1 Structure of Our Proposal

In order to make our novel proposal: array-node R*-tree easier to understand, let us briefly recall the construction of the original R*-tree.

R*-tree is a hierarchy of nested multidimensional MBRs (Minimum Bounding Rectangles). 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 non-leaf node and the MBR is the minimum bounding rectangle of the child node refereed 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 its corresponding point (for point-object databases) or its MBR (for extended object databases). In R*-tree, the root node corresponds the whole index space and each other node corresponds to one sub-space (one rectangle region, i.e., the MBR of all the objects in this region) of the space corresponded to by its parent node. Note that, each MBR in the R*-tree nodes is denoted by two points. One is the vertex with the minimum coordinate in each axis and the other is the vertex with the maximum coordinate in each axis. Hereafter in this paper, no distinction is made between R*-tree nodes and their corresponding MBRs in the multidimensional index space. The Fig.2 is the structures of each non-leaf R*-tree node, (a) non-leaf node and (b) MBR. The structure of each leaf node is omitted.



(a) Structure of each non-leaf node



(b) Structure of each MBR in the R*-tree nodes.

d: dimensionality

Fig.2. Structure of each R*-tree non-leaf node

When the R*-tree is used for range query, all the nodes intersecting with the query range have to be accessed and all the entries in such nodes have to be checked.

The main idea of our proposal is to divide each *d*-dimensional R*-tree into *d* one-dimensional nodes, each of which only holds information in one dimension for each MBR, while each of the original R*-tree node holds information in all dimensions for each MBR. That is, the previous one node will become one node group. Let us see an example using the node in Fig.2. The structure of that node (in Fig.2) in our proposal is shown in Fig.3.

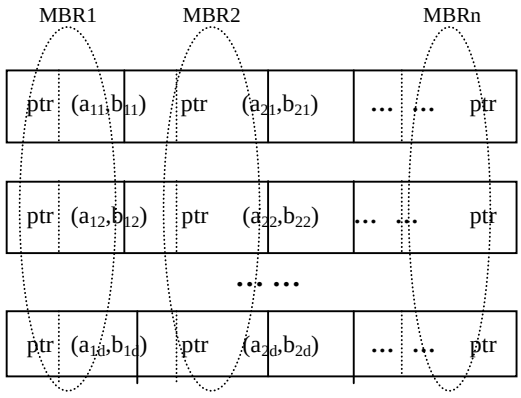


Fig. 3. Structure of node group in our proposal

Astute readers may pay attention to that the total number of nodes in our proposal will become *d* times of the original R*-tree. We want to say that Fig.3 is only a schematic drawing. In fact, the maximum number of entries in each node of our proposal increases greatly since the dimensionality of each node becomes 1 from *d* and our proposal also follow the principle of "one node one page".

Now let us make a comparison between the structure of the original R*-tree and that of our proposal. See Fig. 4.

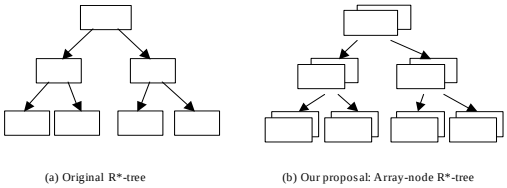


Fig.4. Our proposal vs. the original R*-tree

Since the nodes in each group organized like an array, our proposal is called *Array-node R*-tree*.

4.2 Algorithms of Array-node R*-tree

In this section, the insert algorithm, delete algorithm, range query algorithm will be discussed. Because the insert and delete algorithms are similar as those of the R*-tree, their details are not included in this paper.

4.2.1 Insert and delete algorithms

If the node groups are regarded as supernodes, the ChooseInsertGroup algorithm of node group is the same with the ChooseInsertNode algorithm of the R*-tree.

The split algorithm is also similar to that of the R*-tree. The only different point is that the nodes in that node group must be split in the same time. In the delete algorithm, all the nodes in the under-flowed node group must be deleted in the same time.

4.2.2 Range query algorithm

The procedure of range query algorithm on Array-node R*-tree can be described simply as follows.

Algorithm: range query on Array-node R*-tree

1) Start from root node group
2) Check each entry in this node group to determine if its

MBR intersects with the query range.

a. For each entry to check, only the nodes in this node group that correspond to the query dimensions are necessary to visit.

b. If some entry does not intersect with the query range in some query dimension, then the check of this entry stops. That is, the child node group corresponding to this entry does not need to visit.

c. If the current node group is not at the leaf level then, for the entries that intersect with the query range in all the query dimensions, this algorithm is called in recursively.

If the current node group is at the leaf level, report the entries that intersect with the query range in all the query dimensions.

5. Discussion about Array-node R*-tree

In this section, the Array-node R*-tree is discussed in detail by comparing with R*-tree and with multi-B-trees.

5.1 Array-node R*-tree vs. R*-tree

What advantages does the Array-node R*-tree actually have over the R*-tree? To answer this question clearly, the following estimation is made under the assumption of the multidimensional data (tuples in this study) being distributed uniformly in the index space.

The symbols with their descriptions are showed in Table 2.

Table 2. Symbols and description

Symbols	Description
S	Size of the whole index space
S_q	Size of the extended query range♣
M_r	Maximum number of entries in each leaf node of R*-tree
M_a	Maximum number of entries in each leaf node of Array-node R*-tree
C	Total number of indexed data
N_l	Number of leaf nodes in the case of R*-tree being used
N_g	Number of leaf node groups in the case of Array-node R*-tree being used
d	Number of dimensions used in query
n	Number of dimensions in the whole index space i.e., number of index attributes in this study

In the case of the R*-tree being used to index this database, 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 Array-node R*-tree is used to this case, the number of leaf node groups intersecting with the query range, AR_g, can be given by

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

In the case of the Array-node R*-tree, because the inserting algorithm of the Array-node R*-tree is the same with that of the R*-tree and the number of objects in every leaf group is the same with that of objects in every leaf node, we can say

$$\frac{N_g}{N_l} \approx \frac{M_r}{M_a} .$$

Since the number of dimensions of each R*-tree leaf node is n, while that of each Array-node R*-tree node is only 1, and the node size is the same between the R*-tree and the Array-node R*-tree (one page), we can say

$$\frac{M_r}{M_a} \approx \frac{1}{n} .$$

In the same time, we know that the number of dimensions used in the query is d. Thus, in each node

♣ The extended query range means the range after extended by spanning the ranges of the unused dimensions to the whole data ranges.

group that we have to access, at most d nodes are necessary to visit. Thus, in the case of the Array-node R*-tree, the maximum number of the leaf nodes that we are necessary to visit, AR_l, can be given by

$$AR_l = AR_g \times d .$$

Considering the above equations , we can say

$$AR_l \approx \frac{S_q}{S} \times \frac{1}{n} \times N_l \times d = \frac{d}{n} \times R_l \le R_l .$$

The meaning of Equation (6) is that, for the range QPDs, the number of accessed leaf nodes of the Array-node R*-tree is less than that of the R*-tree. More important, the less the number of dimensions used in queries is, the bigger the advantage of the Array-node R*-tree, which is also verified by the experimental results presented in Section 7.

5.2 Array-node R*-tree vs. multi-B-trees

The readers may ask "how about several B-trees (or say multi-B-trees) are used in the case of QPD instead of the Array-node R*-tree?". Certainly, several B-trees can also be used in QPDs instead of one Array-node R*-tree. That is, we can construct one B-tree (or its variant) using the projection of the objects (tuples) on each index dimension. Totally, n B-trees are necessary for n-dimensional index space. When one QPD are executed, the corresponding B-trees are used individually, and the final result can be given by merging the respective query results. Let us see the details.

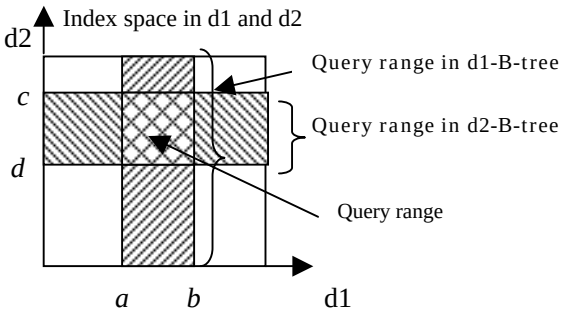


Fig. 5. The case of multi-B-tree

Taking the following case as an example, we can understand the advantages of our proposal (i.e., Array-node R*-tree). Assume that two dimensions, d1 and d2, are used in some query. See Fig. 5. For this case, two B-trees are necessary, called d1-B-tree and d2-B-tree, individually.

At first, the query on d1-B-tree is executed with the query range "a<d1<b" and the result is the set of *result1*. In this query, all the nodes intersecting with *Query range in d1-B-tree* (see Fig. 5) have to be accessed and all the objects located in *Query range in d1-B-tree* are reported. In the same way, the query on d2-B-tree is executed with the query range "d<d2<c" and all the nodes intersecting with *Query range in d2-B-tree* have to be visited. The query result is *result2*. After that, the final result of this QPD, *result*, with d1 and d2 is given by

$$result = result1 \bigcap result2 .$$

The disadvantage of the above method is that the two B-trees are queried independently. That is, the two queries are respectively executed on the two B-trees, where no mutual reference is possible. Many unnecessary investigations are executed and many unnecessary objects are reported. For example, the final result of the QPD with d1 and d2 is only 10 objects, but several hundreds of objects may be reported by each B-tree. This is obviously a problem. Some other demerits of the multi-B-tree include that the management of many B-trees and the final merging both need extra cost. If the Array-node R* -tree is used in the cases of QPDs, the above problems all disappear.

The main advantage of the Array-node R*-tree over multi B-trees is that only one index is needed and many unnecessary investigation can be avoided. The secret is just "mutual reference". Let us see the details.

In a n-dimensional Array-node R*-tree, every node group consists of n one-dimensional nodes corresponding to n dimensions of the index space. Most importantly, every node group corresponds to

one group of objects distributed in some subspace (one MBR) of the whole index space. Thus, the node groups also can be called "*supernodes*". The main features of the Array- node R*-tree include the following two points. One is that, in each node group, only the nodes of the query dimensions are necessary to visit. The other is "mutual reference". When visiting one node group, each entry is checked in all the query dimensions. Only the children whose MBRs actually intersect with the query range in all the query dimensions are followed. On the contrary, in the case of multi-B-tree, only information in one dimension is used to determine the children to follow and many indices are necessary.

6. Experiments

Using the following two datasets, *6D-Uniform200000* and *6D-Zipf200000*, the behaviors of Array-node R*-tree are examined and comparison with the original R*-tree is made.

6D-Uniform200000 200,000 uniformly distributed 6-dimensional floating data. From the view of OLAP data, the dataset is a table of 200,000 tuples with 6 index attributes and the attribute values are uniformly distributed in every attribute.

6D-Zipf200000 200,000 6-dimensional floating data with zipf distribution. From the view of OLAP data, the dataset is a table of 200,000 tuples with 6 index attributes and the attribute values in every attribute are zipf distributed.

The number of query dimensions, or say, the number of the dimensions used in queries is from 1 to 6, i.e., from the minimum number to the maximum number. The pagesize of our system is 4096 bytes and all the tests are repeated 100 with the query ranges of different locations. The state of the Array-node R*-tree and the number of node accesses of each range query are examined and is reported in this section. And, they are compared with the original R*-tree.

Due to the limitation of pages, only the experimental result using *6D-Zip200000* dataset is included in this paper. The result using *6D-Uniform200000* dataset is similar.

The appearances of the R*-tree and the Array-node R*-tree built with *6D-Uniform200000* dataset are shown in Table 2.

Table 2. Appearances of indices

Items	R*-tree	Array-node R*-tree
M	39	203
m	17	91
Height	4	3
Total number of nodes	7331	8424
Memory usage	70.7%	70.32%

Note that, the numbers of M and m in Table 2 means the upper bound and the lower bound, respectively, on the number of the entries in each node for R*-tree. And, for the Array-node R*-tree, they mean the upper bound and the lower bound, respectively, on the number of the entries in each node group.

The queries with different range size and with different query dimensions are tested and given in Figs. 6~11. In these figures, the x-axis: query range side length means the side length of the query range in each dimension. OR*-tree refers to the original R*-tree and AR*-tree means the Array-node R*-tree.

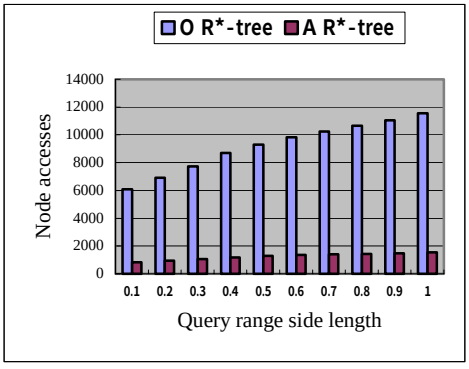


Fig.6. The number of query dimensions=1

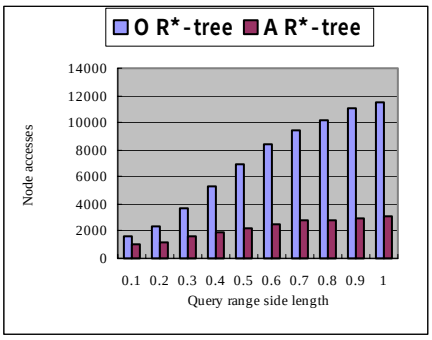


Fig.7. The number of query dimensions=2

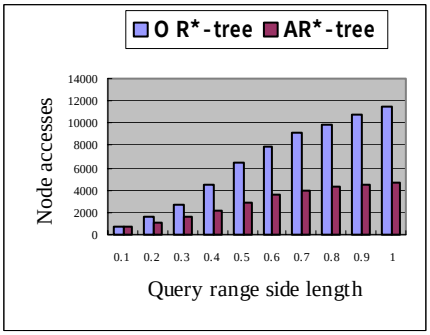


Fig.8. The number of query dimensions=3

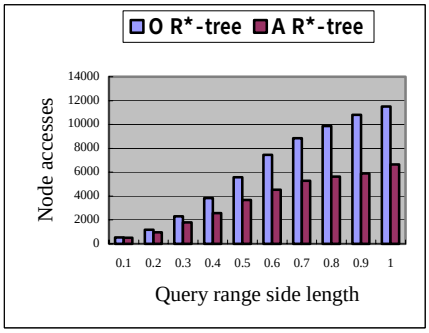


Fig.9. The number of query dimensions=4

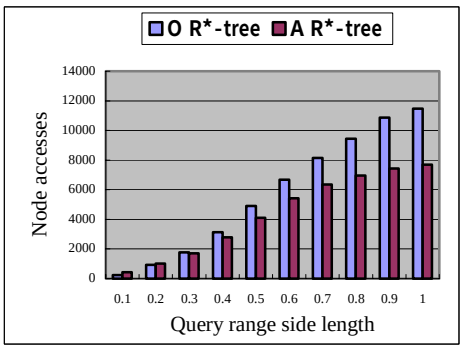


Fig.10. The number of query dimensions=5

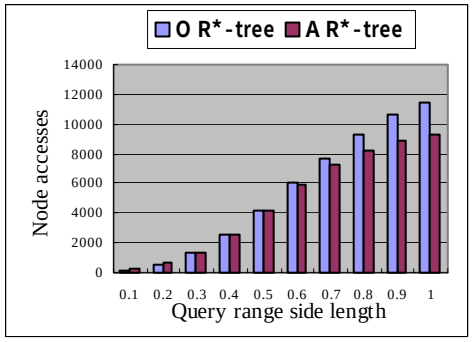


Fig.11. The number of query dimensions=6

7. Conclusion

This study focuses on the queries with partial dimensions (QPD in this study), which is very popular in many applications, especially in OLAP ones. If the traditional methods are used in the case of QPD, there are some problems and performance is not good. This study proposed a novel solution to the issue of QPD, called Array-node R*-tree, which is discussed in detail and is examined by experiments.

8. Reference

[1] TPC benchmark H standard specification. http://www.tpc.org/tpch/

[2] N. Katayama, S. Satoh, “The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries”, Proc. ACM SIGMOD Intl. Conf. 1997.

[3] H. V. Jagadish, N. Koudas, and D. Srivastava, “On Effective Multi-Dimensional Indexing for Strings”, Proc. SIGMOD Conference, 2000.

[4] C. Chung, S. Chun, J. Lee, “Dynamic Update Cube for Range-Sum Queries”, Proc. VLDB Conference, 2001.

[5] N. Beckmann, and H. Kriegel, “The R*-tree: An Efficient and Robust Access Method for Points and Rectangles”, Proc. ACM SIGMOD Intl. Conf.,1990.

[6] S.Berchtold, D.Keim, H.P.Kriegel, “The X-tree: An Index Structure for High-dimensional data”, Proc. VLDB Intl. Conf., 1996.

[7] V. Markl, “Processing Operations with Restrictions in Relational Database Management Systems without External Sorting”, Proc. ICDE Intl. Conf. 1999.