International Journal of Database Theory and Application
Vol. 4, No. 2, June, 2011

# $Ag^+$-tree: an Index Structure for Range-aggregation Queries in Data Warehouse Environments

Yaokai Feng[a], Akifumi Makinouchi[b]

[a]*Faculty of Information Science and Electrical Engineering, Kyushu University, Japan*
*fengyk@ait.kyushu-u.ac.jp*

[b]*Faculty of Information Network Engineering, Kurume Institute of Technology, Japan*
*akifumi@cc.kurume-it.ac.jp*

***Abstract***

*Range-aggregate queries are popular in many applications in data warehouse environments with large business relational databases. To evaluate these efficiently, several studies on data cubes (such as the aggregate cubetree) have been carried out. In the well-known aggregate cubetree, each entry in every node stores the aggregate values of its corresponding subtree. Therefore, range-aggregate queries can be processed without visiting the child subtree whose nodes are all fully included in the query range. However, the aggregate cubetree does not consider range queries using partial dimensions and range queries without aggregation operations. Concretely, 1) a great deal of information that is irrelevant to the queries also has to be read from the disk for partial-dimensional range queries, and 2) while it improves the performance of range queries with aggregate operations, it degrades the performance of range queries without aggregate operations. As part of our research on this problem, previously we proposed an index structure, called the Aggregate-tree (denoted as Ag-tree), which does away with the above-mentioned weaknesses of the aggregate cubetree. Additionally in this paper, we make the Ag-tree more complete by sorting the entries in each of the nodes. The final index structure proposed in this study is called an $Ag^+$-tree.*

***Keywords:*** *Range-aggregate query; Multi-dimensional index; Range query; Aggregation operation*

## 1. Introduction

Range-aggregate queries are very popular in applications with large relational datasets. An example of a range-aggregate query over a relation $F(D_1, D2, \ldots, Dn, M)$ is "Select AggregateFunction($M$) From $F$ Where $l_1 \le D_1 \le h_1$ and $l_2 \le D_2 \le h_2$ and ... and $l_k \le D_k \le h_k$ ", where $F$ is the fact table on which the range-aggregate query is executed. $D_1$, $D_2$, ..., and $D_n$ are dimension attributes, $M$ is a measurement attribute, and the $k$ dimension attributes ($D_1$, $D_2$, ... , $D_k$) ($k \le n$) are used to determine the query range. The attributes used for the query condition in the where-clause are called *query attributes* (*dimensions*). Generally, range-aggregate queries include range-COUNT/SUM/AVG

/MIN/MAX queries. If no aggregate operation is included, the query is aimed at the tuples in the query range.

In many applications, the query condition (range) is often formed with some (but not all) of the dimensions in the fact table. In our previous work [1], we called such queries *Partially- Dimensional range queries* (referred to herein as *PD range queries*). Conversely, queries in which the conditions are decided by all dimensions are called *All-Dimensional range queries* (AD range queries for short). Here, we explain PD range queries using an example. For a relational table $T$ with eight attributes, $A_1$ - $A_8$, assume that the actual attribute combinations possibly used in query conditions are {{$A_1$, $A_3$}, {$A_2$, $A_5$},{$A_3$, $A_4$}, {$A_5$, $A_6$}, {$A_1$, $A_2$}, {$A_2$, $A_4$}, {$A_1$, $A_3$, $A_5$}, and {$A_2$, $A_4$, $A_6$}}. All of these queries are PD range queries. In fact, there can be many actual combinations of query attributes used in all possible PD range queries. Thus, it is not always feasible in applications with large datasets to build a individual index for each possible combination of query attributes, because (1) numerous indices have to be constructed and managed, (2) many attributes are repeatedly included in different indices (e.g., $A_1$ in three indices in the above example). This is too space-consuming for large datasets and resulting in a large maintenance cost, and (3) the combinations of index attributes that are possibly used in the user-provided PD queries are often unpredictable. Note that, there are a total of ($2^{n-1}$) different combinations for $n$ possible query attributes.

A data warehouse [2] is a database system used for analysis, which extracts, integrates, and transforms data from an Online Transaction Processing (OLTP) database and stores them in efficient structures for analysis. Relational databases use star schemas [3] to represent analysis data. The fact table is the center of the star schema, and consists of dimension attributes related to the dimension tables and measurement attributes that are numeric values that may be aggregated. In data warehouse environments, generally, the size of the fact table is so large that the query processing can be time consuming. Various methods have been proposed to solve this problem, such as the scheme maintaining materialized views [4,10,11] for the fact table through the data cube [5,12,13], which is an operator that computes aggregate functions over all possible groups in the fact table. By keeping the results of the data cube operation as a materialized view, fast query execution is possible through the summary information of the fact table. This materialized view is stored as a relation, which may be large. This means that the overhead is high for creating an additional index in the materialized view to improve query performance.

To improve the performance of cube queries, cubetree [8,9] was proposed and used as a technique materializing a data cube through an R-tree-like structure. But the queries have to traverse all internal and leaf nodes in the query range to compute range-aggregate queries [7]. Countering this problem, the study in [16] proposed an enhanced cubetree, called aggregate cubetree, in which each entry in internal nodes stores the aggregate value(s) of its entire subtree. Therefore, by using these aggregate values, range-aggregate queries can be processed without visiting the child nodes whose parent nodes are fully included in the query range. The aggregate cubetree is better than the original cubetree because it can evaluate queries with a smaller number of node accesses.

Obviously, however, the aggregate cubetree suffers from the following two important drawbacks. 1) It does not consider PD range queries, which, as mentioned above, are very popular in data warehouse environments. A great deal of information irrelevant to the queries also has to be read from the disk if it is used for PD range queries. 2) It does

not consider queries without aggregation operations. In actual applications, not all queries have aggregate operations; that is, some queries are aimed at the tuples in the given range. Certainly, the structure of the aggregate cubetree can also be used for range queries without aggregate operations. However, while this improves the performance of range queries with aggregate operations, it degrades the performance of range queries without aggregate operations. This is because many aggregation values (especially when several aggregate functions are applied simultaneously) in the accessed nodes also have to be read from the disk, even though they may not be used in the queries.

As the first part of this study, to counter the weaknesses of the aggregate cubetree, we proposed an index structure [1], called an *Aggregate-tree* (denoted as Ag-tree), which has the following two outstanding properties. 1) For PD range queries with or without aggregate operations, only the information that is relevant to the queries is read from the disk. 2) For range queries without aggregate operations, irrespective of whether they are AD or PD range queries, aggregate values are not read from the disk. In this way, the above-mentioned drawbacks of the aggregate cubetree are overcome.

If we investigate the structure of the Ag-tree more carefully, we find that each of its nodes contains information in only one dimension. This leads naturally to the question: "Why don't we sort them to improve the search performance?". In this paper, we introduce our final proposal in this study, the $Ag^+$-tree, which can be informally defined as a sorted Ag-tree.

The rest of this paper is organized as follows. Section 2 introduces various related works. Our proposal, the $Ag^+$-tree, is presented in Section 3, together with certain necessary algorithms and a discussion. The experimental results are presented in Section 4, while Section 5 concludes this study.

## 2. Related Works

To improve the performance of range queries with aggregate operations, much research has been carried out, especially on the data cube [16, 20, 21, 22, 23]. Ref. [20] introduced a technique called the quotient cube, which is a summary structure for a data cube that preserves its semantics, and has applications in online exploration and visualization. The study in [21] focused on how to reduce the size of data cubes, while that in [22] investigated how to perform aggregations on multiple dimensions simultaneously. Ref. [23] studied high-dimensional OLAP (e.g., 100 dimensions). We found that the aggregate cubetree [16] is related to our study. Before we discuss the aggregate cubetree, let us briefly review its predecessor, the cubetree.

The cubetree is a tree structure for storing a data cube. If the data cube is built on a fact table with $n$ dimension attributes, then it is composed of hyper-planes from $n$-dimensions to zero-dimensions. The basic idea of the cubetree [8,9] is to map the $n$- to zero-dimensional hyper-planes into an $n$-dimensional space, and store them in an R-tree. The data cube is organized as $n$-1 to zero-dimensional data, except for the original $n$-dimensional data, which is called a dataless cubetree, while the cubetree made up of several $n$-1 dimensional R-trees from the dataless cubetree is called a reduced cubetree. If the data cube is created as a reduced cubetree, the size of the data cube can be reduced and the clustering effects can be improved. Other techniques such as sorting and bulk loading can also be applied to improve the clustering effects. However, the cubetree needs to access all the leaf nodes within the query range to process a range-aggregate query, which is the main reason that the query performance decreases as the

size of the query range increases. In data warehouse environments, aggregate queries are frequently requested for large ranges, where a large number of leaf nodes have to be accessed. Thus, the original cubetree possibly shows worse performance for such queries than scanning the entire table [16].

The basic structure of the aggregate cubetree [25] is similar to that of the original cubetree based on an R-tree. In the aggregate cubetree, each node is composed of a number of entries. Each leaf node entry is made up of dimension attributes and the aggregate value of the measurement attributes, while each internal node entry is composed of an MBR (Minimum Bounding Rectangle), child node pointer, and an aggregate value. The aggregate value for each internal node entry is the aggregate value of all the tuples in the corresponding subtree and can be calculated from the aggregate values of all its child node entries.

The range queries on R-tree-like structures are performed by recursively searching nodes overlapping the query range. In the cubetree, all the nodes overlapping the query range must be accessed for range-aggregate queries. In the aggregate cubetree, however, by using the aggregate values in the entries of the internal nodes, range-aggregate queries can be performed without visiting child nodes whose parent nodes are fully included in the query range. Therefore, the number of node accesses can be reduced compared with the cubetree.

Although the aggregate cubetree was originally directed to data cubes, its structure can certainly be used for range queries on relational datasets. In fact, it can be used for both AD and PD range queries, with or without aggregate operations. Each entry in the leaf nodes corresponds to one tuple. Besides the index attributes, the measurement attributes are also contained in the leaf nodes. Figure 1 shows an example of the aggregate cubetree [25].

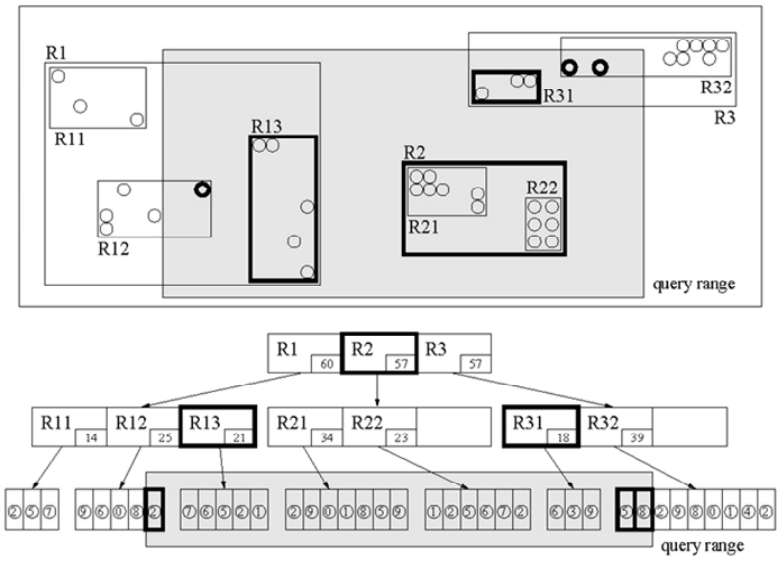


**Fig. 1**. Example of aggregate cubetree.

By storing all necessary aggregate values in the intermediate nodes, the aggregate cubetree can improve the performance of range-aggregate queries. However, according to our investigation, it suffers from the following two weaknesses.

**1. It does not consider PD range queries.** Like many other existing multidimensional indices, in the aggregate cubetree, all the objects (tuples) 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 the index dimensions (i.e., each node entry includes the MBR ranges in all index dimensions). That is, the aggregate cubetree is also suited to evaluating AD range queries. Using an $n$-dimensional aggregate cubetree, a PD range query with $k$ ($k < n$) query dimensions can be evaluated by simply extending the query range in each of the remaining ($n$-$k$) irrelevant index dimensions to the entire data range. The weakness is that each node in the aggregate cubetree contains $n$-dimensional information, but only $k$-dimensional information is necessary for a $k$-dimensional PD range query, which means that a great deal of unnecessary information (i.e., the information in the irrelevant dimensions) also has to be read from disk. This certainly degrades the query performance significantly.

**2. It does not consider queries without aggregation operations.** The aggregate cubetree is aimed at queries with aggregate operations. However, there are many other queries that do not need aggregate operations (e.g., those aimed at the tuples in the query range). If the aggregate cubetree is used to evaluate such queries, then many aggregation values in the accessed nodes also have to be read from the disk, although these are not used in the queries. In other words, the aggregate values in the aggregate cubetree nodes reduce the capacity of each node, particularly in cases where several aggregate values for COUNT, SUM, AVG, MIN, and MAX are required.

Using the method proposed in this study, both the above problems can be solved. That is, for PD range queries, the information in the irrelevant dimensions need not be read from disk, while no aggregate values are read from disk for queries without aggregate operations.

## 3. Our Proposal

### 3.1 General Structure

To improve common PD range queries executed on the R*-tree (not the aggregate cubetree), we proposed the Adaptive R*-tree [17]. In the Adaptive R*-tree, each of the nodes in the original $n$-dimensional R*-tree is divided into $n$ nodes, each of which contains information of its entries in one dimension. That is, the information contained in each entry in an R*-tree node is divided and stored in different nodes according to its dimensions. In this way, for PD range queries, only those nodes corresponding to the query dimensions are read from disk. Mathematical analysis and several experiments have verified that the Adaptive R*-tree has much better query performance for PD range queries than the original R*-tree and, in most cases, it also has better query performance for AD range queries [17]. The Adaptive R*-tree is extended in this study for aggregate operations, which is called the $Ag^+$-tree.

The key ideas of the $Ag^+$-tree include the following. 1) Each node in an $n$-dimensional aggregate cubetree is divided into $n$ one-dimensional nodes, each of which only holds information in one dimension. Thus, for PD range queries with or without aggregate operations, only nodes corresponding to the query dimensions are possibly accessed. The nodes in the irrelevant dimensions can be skipped. 2) Unlike the aggregate cubetree, all the aggregate values in the $Ag^+$-tree are contained in a specific node, which is independent of the index parts of the $Ag^+$-tree. In this way, nodes for aggregate values need not be accessed for queries without aggregate operations. The general structure of the $Ag^+$-tree is depicted in Fig. 2, and is similar to the structure in our previous paper [1]. The difference lies in the content of

each node, as explained in Section 3.2. In Fig. 2, the white nodes form the index part, while the black nodes, called *aggregate nodes*, store aggregate values.

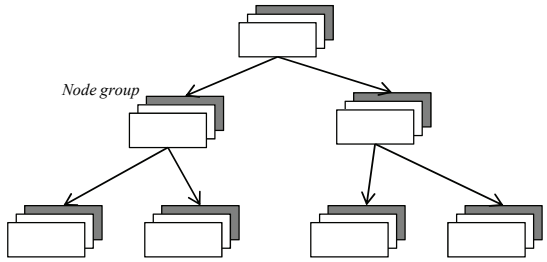


**Fig. 2.** $Ag^+$-tree Structure.

In an aggregate cubetree, each node entry corresponds to one complete MBR (one subspace in the index space). In the $Ag^+$-tree, however, each node entry only corresponds to one edge of the corresponding MBR, i.e., a complete *n*-dimensional MBR is divided into *n* edges, which are stored in different nodes of a single node-group.

Here, the following two questions arise.

**How to determine the capacity of each node-group?** Obviously, all the nodes in each node-group must have the same number of entries. Based on the principle of "one node per page", we can determine the capacity of each node in the index part (say $f_x$) and that of an aggregate node (say $f_a$). Then, the final fanout/capacity of each node-group is the smaller of $f_x$ and $f_a$, i.e., $\min\{f_x, f_a\}$. Generally, $f_a > f_x$. Thus, commonly speaking, the final fanout/capacity of each node-group is determined by the index-part. In other words, the aggregate nodes generally do not influence the fanout of the index part.

**Whether the ``node-dividing'' of the $Ag^+$-tree results in a significant increase in the total number of nodes?** The answer is "No". This is because each node in the $Ag^+$-tree contains only information in one dimension and no aggregate values, which means a node in the $Ag^+$-tree can contain a larger number of entries than a node in the aggregate cubetree.

### 3.2 Sorting Entries in Each Node

From Section 3.1, we know that all the information in each of the $Ag^+$-tree nodes is in one dimension. It is very natural to sort the entries in each of the nodes to speed up the queries executed on the $Ag^+$-tree. In order to sort the entries, we have modified the construction algorithm for the Ag-tree [1]. That is, sorting of the entries is done while the $Ag^+$-tree is being built. And, a number is attached to each entry to keep the corresponding relationship of the entries in different nodes in each node-group.

### 3.3 Algorithms

**Insert and delete algorithms**. The insert algorithm for the $Ag^+$-tree is a naive extension of its counterpart for the Ag-tree [1]. When a new tuple reaches a leaf node-group, it is divided and stored in different nodes according to its dimensions. If a split is necessary for an overflowing node-group, all its nodes have to be split at the same time and the split may be propagated upwards. After a delete operation, if the node-group has under-flowed, all its

nodes should be deleted at the same time and the entries are inserted again. That is, all the nodes in each node-group must be born simultaneously and die simultaneously. For aggregate nodes, as the new tuples are inserted, the corresponding aggregate values in the ancestor node-groups should be updated. The insert and delete algorithms are omitted here. Note that, the sorting operation for the entries in each node is carried out while the index structure is being built.

**Algorithm for range queries without aggregate operations.** This algorithm, shown in Table 1, can be used for both AD and PD range queries without aggregate operations.

**Table 1**
Algorithm for range queries without aggregate operations.

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

*) An entry includes all parts with the same index in the different nodes of this node-group.

**) When an entry is investigated to determine whether it intersects the query range, only the nodes corresponding to the query dimensions need be accessed. Even in the visited node-groups, not all nodes corresponding to the query dimensions need be checked since investigation of the current entry can be terminated if it is found not intersecting the query range in the current query dimension.

**Algorithm for range queries with aggregate operations**. An algorithm for Range-SUM queries is given in Table 2. The algorithms for other aggregate functions can be accomplished in a similar way. Like the algorithm in Table 1, when an entry is checked to determine its relationship (*IsFullyContainedBy*, *NotSeparatedFrom*) with the query range, only the nodes corresponding to the query dimensions need be accessed. Not even all the nodes in the query dimensions need be checked (see Table 1). Once the current entry *e* is found not to be fully contained in the query range in some query dimension (*condition 1* in this algorithm) and the *node-group* is not at leaf level, the relationship of *NotSeparatedFrom* will be checked (*condition* 2). In *condition* 2, once the current entry *e* is found not intersecting the query range in some query dimension, the investigation of *e* is terminated.

When the search reaches the leaf nodes, *condition* 1 is used to judge whether an entry is contained in the query range. The measured attribute values of the entries located in the query range are added to the result.

**Table 2**
Algorithm for range queries with aggregate operations.

```
Procedure  RangeSUM (rect, nodegroup)
Input:  rect;  //query range
        nodegroup; // initial node-group of the query
Output:  result; //sum of aggregate values in the query range
Begin
  SUM_type result ← 0;
  For each entry e in nodegroup Do
    If  e IsFullyContainedBy rect in all the query dimensions (condition 1) Then
      result ← result+ e.aggregateSUM
    Else If  (nodegroup is not at a leaf) AND
      (e NotSeparatedFrom  rect in all the query dimensions) (condition 2)  Then
      result ← result+RangeSUM (rect, e.child);
  EndFor
  Return result
End
```

### 3.4 Performance Discussion: $Ag^+$-tree vs. Aggregate Cubetree

The unique features of the $Ag^+$-tree are that information in different dimensions is divided and stored in different nodes, and the aggregate values are stored in independent nodes. Thus, the $Ag^+$-tree appears to be efficient for PD range queries, irrespective of whether they include aggregate operations.

We could certainly also build two aggregate cubetrees for range queries with and without aggregate operations. This would, however, (a) be most wasteful of space, especially for large datasets, (b) require extra cost to maintain two indices, and (3) not solve the above problem of PD range queries even if two indices are built.

The structure of the $Ag^+$-tree guarantees that it can be applied to PD range queries with any combinations of query dimensions. Information in dimensions that are irrelevant to the current query is not read from disk for PD range queries and the aggregate values are read from disk only when they are necessary. Moreover, as mentioned in the discussion of Table 1, not even all nodes (of the visited node-groups) corresponding to the query dimensions need to be checked.

We can also interpret the difference between the $Ag^+$-tree and the aggregate cubetree in the following way. 1) Only the information in one dimension (one edge of the MBR) is contained in each node in the $Ag^+$-tree, whereas each node in the aggregate cubetree contains $n$-dimensional information ($n$ edges of a complete MBR). 2) Every entry in a

node in the aggregate cubetree must contain the aggregate values, thereby decreasing the capacity (maximum number of entries) of each node, especially in cases where several aggregate functions are needed. In the index part of the $Ag^+$-tree, however, no aggregate values are needed. Thus, the capacity of each leaf node-group in the $Ag^+$-tree is roughly $n$ times that of a leaf node in the $n$-dimensional aggregate cubetree. For uniformly distributed data, the number of accessed leaf node-groups in the $Ag^+$-tree is roughly $1/n$ of that of accessed leaf nodes in the aggregate cubetree. However, in each accessed leaf node-group in the $Ag^+$-tree, at most $d$ (the number of query dimensions) nodes need to be visited. Then, the number of actually accessed leaf nodes in the $Ag^+$-tree is not greater than $d/n$ ($d \le n$) of that in the aggregate cubetree.

### 3.5 Performance Discussion: $Ag^+$-tree vs. R*-tree

In this section, under the assumption of uniform distribution, the performance of the $Ag^+$-tree is discussed mathematically. The number of accessed leaf nodes is estimated and compared with the R*-tree since this is an important factor with regard to query performance, especially for large datasets.

The symbols used in this section are as follows: $n$: dimensionality; $d$: number of query dimensions; $S$: volume of the entire index space; $S_q$: volume of the extended query range of a PD range query; $N_l$: number of leaf nodes in the case of the R*-tree; $M_r$: maximum number of entries in each leaf node; $M_g$: maximum number of entries in each leaf node-group in an $Ag^+$-tree; and $N_g$: number of leaf node-groups in an $Ag^+$-tree.

In the case of an R*-tree, the average number of leaf-node accesses, $R_l$, is given by $R_l = \frac{S_q}{S} \times N_l$. If an $Ag^+$-tree is used, then the average number of leaf node groups intersecting the query range, $AR_g$, is given by $AR_g = \frac{S_q}{S} \times N_g$.

Since the node sizes of the R*-tree and $Ag^+$-tree are the same (one node per page), the maximum number of entries in each leaf node of an $Ag^+$-tree is roughly $n$ times that in each leaf node of an R*-tree. This is easily understood by considering that only one-dimensional information for each entry is contained in each of the $Ag^+$-tree nodes. In addition, considering that the clustering algorithms (insert algorithms) of the R*-tree and $Ag^+$-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, $AR_l$, is given by

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

This equation indicates that, for PD range queries with $d < n$, the number of accessed leaf nodes in the case of an $Ag^+$-tree is less than that in the case of an R*-tree. Even if $d = n$ (AD queries), then the number of accessed leaf nodes is approximately the same and it is possible that $AR_l < R_l$. Moreover, for a fixed $n$, the lower the number of query dimensions, $d$, the bigger the advantage of the $Ag^+$-tree compared to the R*-tree.

This can be explained as follows. Because the capacity of each leaf node-group in an $Ag^+$-tree is roughly $n$ times that of each leaf node in an R*-tree, the number of accessed leaf

[1] Although this query is relevant to $d$ dimensions, investigation of each node-group-entry may stop midway.

node-groups in an $Ag^+$-tree is approximately $1/n$ times that of the accessed leaf nodes in an R*-tree. However, in each of the accessed leaf node-groups of the $Ag^+$-tree, at most $d$ nodes must be visited.

## 4. Experiments

Datasets with a Zipf distribution with a constant of 1.5 were used to examine the behavior of the $Ag^+$-tree for range queries with or without aggregate operations. The datasets have 300,000 tuples having six dimensions (attributes).

The pagesize of our system is 4096 bytes. Query performance is measured in term of the number of leaf node accesses, for the following reasons. (1) I/O cost is still the performance bottleneck for many systems and thus, the number of accessed nodes is tested and compared in many studies on multidimensional indexing. (2) Leaf nodes constitute the overwhelming majority of the total nodes and they tend to be stored in secondary storage [19]. (3) The $Ag^+$-tree is wider than the aggregate cubetree and it is perhaps also lower than the aggregate cubetree, since each node-group holds more tuples. Thus, comparing the number of accessed leaf nodes is fairer.

Range queries with varying numbers of query dimensions are tested with different range sizes. The ratio of the side length of the query range to the domain size of the corresponding query dimension varies from 10% to 80% in increments of 10%. A range query of the same size is repeated 100 times with random locations and the average performance is presented. From the experimental results, we can see that the $Ag^+$-tree can be used more efficiently for range queries, irrespective of whether they include aggregate operations.

### 4.1 Range Queries without Aggregate Operations

**Table 3**
Leaf node accesses for range queries without aggregate operations.

| Range Size (%) | | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 |
|---|---|---|---|---|---|---|---|---|---|
| d=1 | $Ag^+$-tree | 300 | 475 | 507 | 601 | 674 | 804 | 807 | 858 |
| | Cube tree | 1450 | 2260 | 2714 | 3150 | 3626 | 4290 | 4546 | 4607 |
| d=2 | $Ag^+$-tree | 158 | 380 | 459 | 704 | 979 | 1242 | 1342 | 1396 |
| | Cube tree | 758 | 1709 | 1964 | 2387 | 2993 | 3801 | 4140 | 4498 |
| d=3 | $Ag^+$-tree | 72 | 326 | 349 | 678 | 1008 | 1465 | 1723 | 1883 |
| | Cube tree | 439 | 1390 | 1707 | 2052 | 2638 | 3591 | 4009 | 4352 |
| d=4 | $Ag^+$-tree | 35 | 247 | 304 | 616 | 994 | 1583 | 2019 | 2288 |
| | Cube tree | 73 | 996 | 1251 | 1580 | 2305 | 3316 | 3811 | 4308 |
| d=5 | $Ag^+$-tree | 23 | 175 | 230 | 504 | 912 | 1645 | 2250 | 2618 |
| | Cube tree | 35 | 934 | 1160 | 1493 | 2192 | 3203 | 3702 | 4289 |

The experimental results depicted in Table 3 show the following. (1) As the number of query dimensions increases, the performance advantage of the $Ag^+$-tree compared to the aggregate cubetree decreases. This can be understood easily from Section 3.4 and Section 3.5. (2) The performance of range queries executed on the aggregate cubetree improves as the number of query dimensions increases. We think this is because that the search region can be limited in more dimensions. (3) As the number of query dimensions increases, the query performance of the $Ag^+$-tree varies in an interesting way. On the one hand, the number of accessed node-groups decreases for the same reason as (2). On the other hand, the number of accessed nodes in each accessed node-group may increase. Under these two conflicting influences, as the number of query dimensions increases, the query performance of the $Ag^+$-tree does not change much where the query size is not greater than 40%. However, if the query size exceeds 50%, the query performance of the $Ag^+$-tree clearly deteriorates as the number of query dimensions grows. This appears to be because more nodes generally have to be accessed in each visited node-group.

### 4.2 Range Queries with Aggregate Operations

The Range-SUM query is used as an example. The experimental results are given in Table 4, which shows that both the query performance of the aggregate cubetree and that of the $Ag^+$-tree deteriorate slowly as the query size grows. This appears to be because the following two conflicting tendencies occur as the query size increases.

A) More nodes intersect the query range.

B) More nodes are fully contained in the query range and their child/descendant nodes do not need to be checked.

**Table 4**
Leaf node accesses for range queries with aggregate operations.

| Range Size (%) | | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 |
|---|---|---|---|---|---|---|---|---|---|
| d=1 | $Ag^+$-tree | 176 | 243 | 418 | 454 | 463 | 603 | 624 | 656 |
| | Cube tree | 866 | 1660 | 1814 | 2254 | 2613 | 2908 | 3214 | 3500 |
| d=2 | $Ag^+$-Tree | 158 | 230 | 400 | 470 | 502 | 612 | 670 | 692 |
| | Cube tree | 758 | 1220 | 1601 | 1940 | 2280 | 2706 | 2960 | 3212 |
| d=3 | $Ag^+$-tree | 96 | 212 | 423 | 491 | 561 | 690 | 723 | 802 |
| | Cube tree | 234 | 1041 | 1207 | 1847 | 2012 | 2541 | 2704 | 2945 |
| d=4 | $Ag^+$-Tree | 35 | 280 | 460 | 561 | 702 | 770 | 850 | 1060 |
| | Cube tree | 73 | 802 | 1160 | 1510 | 1802 | 2150 | 2540 | 2706 |
| d=5 | $Ag^+$-Tree | 23 | 228 | 520 | 702 | 862 | 984 | 1106 | 1260 |
| | Cube tree | 35 | 670 | 1008 | 1406 | 1752 | 2004 | 2206 | 2450 |

In fact, as the query dimension *d* increases, the query range becomes smaller because it is limited in more dimensions. In this case, there are also two conflicting factors affecting the performance. Thus, the number of accessed leaf nodes increases slowly as the query dimension *d* grows.

A) The number of accessed nodes tends to decrease.

B) The number of nodes contained fully in the query range tends to decrease.

## 5. Conclusion

In this study, we proposed a novel structure (referred to as the $Ag^+$-tree) for range queries in data warehouse environments. In the new structure, the information of each node entry is divided and stored in individual nodes according to the index dimensions. In addition, the necessary aggregate values are stored in separate nodes that are independent of the index nodes. In this way, for partially-dimensional range queries (PD range queries) with or without aggregate operations, only the information in the query dimensions needs to be read from disk. And, for range queries without aggregate operations, the aggregate values are not read from disk. The discussions and various experiments indicate that the proposed method can be used efficiently for range queries, irrespective of whether they include aggregate operations. Although the proposed method is based on an R*-tree, many other structures can also be employed. The basic requirements for applying the proposed method to an index structure are that (1) the index is MBR-based, and (2) in the index, the region that is covered by every node must be completely contained within the region of its parent node. Note that, the idea of the proposed method cannot be used with hyper-sphere-based structures (such as an SS-tree), which are directed to nearest neighbor queries (e.g., similarity queries) and cannot be efficiently used for range queries.

## Acknowledgments

This research was supported in part by the Japan Society for the Promotion of Science through a Grant-in-Aid for Scientific Research (22500093). In addition, the authors would like to thank Mr. Zhibin Wang, who carried out some of the experiments.

## References

1. Y. Feng, A. Makinouchi, Ag-tree: a novel structure for range queries in data warehouse environments, Lecture notes in Computer Science (LNCS) 3882 (2006) 498-512.
2. S. Chaudhuri, U. Dayal, An overview of data warehousing and OLAP technology, ACM SIGMOD Record 26(1) (1997) 65-74.
3. R. Kimball, The data warehouse toolkit, John Wiley & Sons, 1996.
4. N. Roussopoulos, Materialized views and data warehouses, ACM SIGMOD Record 27(1) (1998) 21-26.
5. J. Gray, A. Bosworth, A. Layman, H. Piramish, Data Cube: a relational aggregation operator generalizing group-by, crosstab, and sub-totals, In: Proc. Int. Conf. on Data Engineering (ICDE), 1996, pp. 152-159.
6. A. Guttman: R-trees: a dynamic index structure for spatial searching, In: Proc. ACM SIGMOD Int. Conf. on Management of Data, 1984, pp. 47-57.
7. C. Ho, R. Agrawal, N. Megiddo, R. Srikant, Range queries in OLAP Data Cubes, In: Proc. ACM SIGMOD Int. Conf. on Management of Data, 1997, pp. 73-88.
8. N. Roussopoulos, Y. Kotdis, M. Roussopoulos, Cubetree: organization of and bulk incremental update on the Data Cube, In: Proc. ACM SIGMOD Int. Conf. on Management of Data, 1997, pp. 89-99.

9. Y. Kotdis, N. Roussopoulos, An alternative storage organization for ROLAP aggregate views based on Cubetrees, In: Proc. ACM SIGMOD Int. Conf. on Management of Data, 1998, pp. 249-258.
10. H. Gupta, Selections of views to materialize in a data warehouse, In: Proc. Int. Conf. on Database Theory (ICDT), Delphi, 1997, pp. 98-112.
11. S. Mumick, D. Quass, B.S. Mumick, Maintenance of Data Cubes and summary tables in a warehouse, In: Proc. ACM SIGMOD Int. Conf. on Management of Data, Tucson, Arizona, 1997, pp. 100-111.
12. V. Harinarayan, A. Rajaraman, J.D. Ullman, Implementing data cubes efficiently, In: Proc. ACM SIGMOD Int. Conf. on Management of Data, 1996, pp. 205-216.
13. S. Sarawagi, R. Agrawal, A. Gupta, On computing the data cube, Research Report, IBM Almaden Research Center, San Jose, Ca, 1996.
14. N. Beckmann, H.P. Kriegel, R. Schneider, B. Seeger, The R*-tree: an efficient and robust access method for points and rectangles, In: Proc. ACM SIGMOD Int. Conf. on Management of Data, Atlantic City, NJ, 1990, pp. 322-331.
15. S. Agrawal, R. Agrawal, P. Deshpande, et al., On the computation of multidimensional aggregates, In: Proc. Int. Conf. on Very Large Databases (VLDB), 1996, pp. 506-521.
16. S. Hong, B. Song, S. Lee, Efficient execution of range aggregate queries in data warehouse environments, In: Proc. Int. Conf. on the Entity Relationship Approach (ER), 2001, pp. 299-310.
17. Y. Feng, A. Makinouchi, Efficient evaluation of partially-dimensional range queries in large OLAP Datasets, International Journal of Data Mining, Modelling and Management, 3(2) (2011), to appear.
18. C. Zhang, J. Naughton, et al., On supporting containment queries in relational database management systems, In: Proc. SIGMOD Int. Conf. on Management of Data, 2001, pp. 425-436.
19. G.R. Hjaltason, H. Samet, Distance browsing in spatial database, ACM Trans. on Database Systems 24(2) (1999) 265-318.
20. L.V.S. Lakshmanan, J. Pei, Y. Zhao, Qctrees: an efficient summary structure for semantic OLAP, In: Proc. ACM SIGMOD Int. Conf. on Management of Data, 2003.
21. W. Wang, H. LU, J. Feng, J.X. Yu, Condensed cube: an effective approach to reducing Data Cube size, Proc. Int. Conf. on Data Engineering (ICDE), 2002.
22. D. Xin, J. Han, X. Li, B.W. Wah, Star-cubing: computing Iceberg Cubes by top-down and bottom-up integration, In: Proc. Int. Conf. on Very Large Databases (VLDB), 2003.
23. Y. Feng, A. Makinouchi, Batch-incremental nearest neighbor search algorithm and its performance evaluation, IEICE Trans. on Information and Systems E86-D(9) (2003) 1856-1867.
24. X. Li, J. Han, H. Gonzalez, High-dimensional OLAP: a minimal cubing approach, In: Proc. Int. Conf. on Very Large Databases (VLDB), 2004, pp. 528-539.
25. S. Hong, B. Song, S. Lee, Efficient execution of range-aggregate queries in data warehouse environments, In: Proc. 20th Int. Conf. on Conceptual Modeling (ER), 2001, pp. 122-129.

## Authors

**Yaokai Feng** received his B.S. and M.S. degrees in Computer Science from Tianjin University, China in 1986 and 1992, respectively. After receiving his PhD degree in Information Science from Kyushu University, Japan, in 2004, he remained at the same university as a Research Associate. Currently, he is an Assistant Professor at 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 Univercite de Grenoble, France in 1979, and D.E. degree from Kyoto University, Japan, in 1988. Between 1990 and 2006, he was with the Graduate School of Information Science and Electrical Engineering, Kyushu University, Japan, where he was a Professor. Since 2006, he has been a Professor in the Department of Information Network Engineering, Kurume Institute of Technology, Japan. He is a member of IPSJ, ACM, and IEEE and a fellow of IEICE.