The RD-Tree: a structure for processing
Partial-MAX/MIN queries in OLAP

Woo Suk Yang, Yon Dohn Chung, Myoung Ho Kim*

Division of Computer Science, Department of Electrical Engineering and Computer Science,
Korea Advanced Institute of Science and Technology (KAIST), 373-1, KuSung-Dong, YuSung-Gu,
Taejon, 305-701, Republic of Korea

Received 10 February 2000; received in revised form 10 March 2001; accepted 8 May 2001

Abstract

Online analytical processing (OLAP) systems have been introduced to facilitate decision support applications. While most previous studies deal with the situation where the aggregate functions are applied to all cells in a given range, this paper considers a class of queries, called the Partial-MAX/MIN query, that are applied only to specified cells in a given range. We propose the Rank Index and Rank Decision Tree (RD-Tree) for efficient processing of the partial-max/min queries. Through experiments, we show our approach has an efficient and robust processing capability for partial-max/min queries.

© 2002 Elsevier Science Inc. All rights reserved.

Keywords: Partial-max/min query; Query processing; OLAP; Databases

1. Introduction

Online analytical processing (OLAP) [1] allows organizations to analyze the data built from their data warehouses, and supports their decision making of business strategies. A popular data model for OLAP applications is the...
multi-dimensional database [2], commonly called a data cube [3,4]. In construction of a data cube, certain attributes are selected from the data warehouse. Some of these attributes are chosen as metrics of interest and are referred to as the measure attributes, and the remaining $d$ attributes are referred to as the dimensions or the functional attributes. The measure attributes of all records with the same combination of functional attributes are combined into an aggregate value. Thus, a data cube can be viewed as a $d$-dimensional array that is indexed by $d$ functional attributes, and the cells of the array contain the values of measure attributes for the corresponding combination of the functional attributes.

Some previous approaches have been proposed for processing aggregates (especially for a given range) in OLAP data cubes. [3] proposed a solution for range-sum queries by using the prefix-sum array method. This method precomputes the sums of several sub-regions and uses them for processing user queries. In [5], a method for range-max aggregate queries was proposed. They defined the maximal cover concept and used it through a hierarchical tree structure.

In this paper, we consider a class of queries, called partial-max (or min) query, which chooses the maximum (or minimum) value among selected cells of a data cube, where selection is specified by providing a subset of the functional attributes. (In [3], the query that finds the sum of selected cells of a data cube is called a partial-sum query.) This type of partial aggregate queries, such as partial-max (min), partial-sum, partial-count and so on, are frequently used with respect to categorical attributes. For instance, consider an insurance data cube with the functional attributes of ‘state’, ‘time period’ and ‘insurance type’, and the measure attribute of ‘revenue’. A partial max query may obtain the maximum revenue from the states of California, Florida and Texas, for the first quarters of 1997, 1998 and 1999, and for life and health insurances.

In this paper, we focus on the efficient processing of partial-max/min queries in OLAP data cubes.

**Partial-MAX/MIN problem.** The one-dimensional partial-max/min problem is formally stated as follows.\(^1\) Let $A$ be a data array of size $m$, indexed from 0 to $m-1$. Let $M = \{0, 1, \ldots, m-1\}$ be the set of indices of $A$. Given a query $I$, which is represented by a subset of $A$’s index domain i.e., $I \subseteq M$, the problem of finding the partial-max/min of query $I$, denoted by $P_{\text{max}}(A, I)$ and $P_{\text{min}}(A, I)$, is defined as follows:

$$P_{\text{max}}(A, I) = \text{MAX}_{i \in I} A[i],$$

\(^1\) It is also applied to the $d$-dimensional environment, since the $d$-dimensional space is actually mapped to the one-dimensional disk space.
\[ P_{\text{min}}(A, I) = \text{MIN}_{i \in I} A[i]. \]

**Example 1.** Suppose that there are a data array \( A = \langle 4, 7, 3, 6, 9 \rangle \) and a query \( I = \{0, 1, 3\} \). (In the paper, we use ‘(‘ and ‘)’ for array notation.) Then, the partial-max and partial-min of the query \( I \) are as follows:

\[ P_{\text{max}}(A, I) = 7, \quad P_{\text{min}}(A, I) = 4. \]

2. **Rank index**

**Definition 1.** Let there be a data array \( A \) of size \( m \). The rank index \( R \) for \( A \) is an array of size \( m \) such that the value of \( R[i] \) is an index for the array \( A \) which satisfies the following inequality:

\[ A[R[i]] \geq A[R[j]], \quad \text{if} \ i < j. \]

Fig. 1 shows an example of the rank index structure, where the data array \( A \) contains 12 values. In the figure, the maximum value of \( A \) is \( A[R[0]] \) and the minimum is \( A[R[11]] \). The \( i \)th largest value in \( A \) is indicated by \( R[i−1] \) i.e., \( A[R[i−1]] \).

**Observation 1.** Let \( A \) be a data array of size \( m \) and \( R \) be the rank index for \( A \). For a partial-max/min query \( I \), the following properties hold.

- \( P_{\text{max}}(A, I) = A[R[k]], \) if \( R[k] \in I \) and there is no \( j \) such that \( R[j] \in I \) and \( j < k \).
- \( P_{\text{min}}(A, I) = A[R[k]], \) if \( R[k] \in I \) and there is no \( j \) such that \( R[j] \in I \) and \( j > k \).

According to the above observation, the maximum or minimum value over the specified cells can be easily determined through the sequential search of rank index \( R \) of data array \( A \).

In the rest of the paper, a partial-max/min query \( I \) is denoted by a bit vector \( V_I = (b_0 b_1 \ldots b_{m-1}) \), where \( b_k = 1 \) if \( k \in I \) and \( b_k = 0 \) otherwise. The weight of a bit vector \( V \) is the number of 1’s in \( V \) and is denoted by \( |V| \). The length of \( V \) is the total number of bits in \( V \).

**Fig. 1.** An example of rank index.
3. RD-Tree: rank decision tree

For finding the maximum value in the rank index, we have to search the rank index sequentially and check if the cell is specified by the query. That is, in the search, the first cell $R[i]$ in rank index $R$ that is specified by the query becomes the index for the maximum value i.e., $A[R[i]]$. Thus, for the partial-max query where all the specified cells have relatively small values, we may have to search the entire array of $R$. For instance, when $I = \{0, 9\}$ (i.e., cells $A[0]$ and $A[9]$ are specified) in Fig. 1, we have to search $R$ sequentially until $R[10]$.

**Definition 2.** Let $A$ be the data array representing a data cube, $R$ be the rank index for $A$ and $I$ be a partial query. The rank-max ($I$) is the smallest $k$ such that $A[R[k]]$ is equal to $\max_{i \in I} A[i]$. Similarly, the rank-min ($I$) is the largest $k$ such that $A[R[k]]$ is equal to $\min_{i \in I} A[i]$.

Thus, $P_{\text{max}}(A, I) = A[R[\text{rank-max}(I)]]$ and $P_{\text{min}}(A, I) = A[R[\text{rank-min}(I)]]$. We propose an index structure, named the Rank Decision Tree (RD-Tree), by which we can find the value of rank-max and rank-min of the given query efficiently.

Since the problem of finding the minimum value is an exact dual of that of finding the maximum value, we will mainly discuss on finding the maximum value. The discussion on the partial-min query will be briefly described in Section 3.4.

3.1. Structure of the RD-Tree

Before presenting the RD-Tree, we first define the concept of a Rank Bisection Signature (RBS). An RBS for a data array of size $m$ is an $m$-bit vector, where each bit indicates whether the value of the corresponding cell is ranked above the half or not.

**Definition 3.** Let there be a data array $A$ and its rank index $R$. The rank bisection signature (RBS) of $A$ is a bit vector $VS = (b_0 b_1 \ldots b_{m-1})$ such that

$$b_{R[i]} = \begin{cases} 1 & \text{if } 0 \leq i < \lfloor m/2 \rfloor, \\ 0 & \text{otherwise}. \end{cases}$$

**Definition 4.** Let $A$ be a data array of size $m$, and $R$ be a rank index for $A$. We define $A_{i,u}$ to be an array of size $u - l + 1$ such that $\{A_{i,u}[i] \mid i = 0, \ldots, u - l\} = \{A[R[k]] \mid k = l, \ldots, u\}$ and the relative orders among the elements in $A_{i,u}$ are equal to those in $A$. $\text{RBS}_{i,u}$ is defined as a rank bisection signature applied to $A_{i,u}$.
Note that $A_{lu}$ consists of $u - l + 1$ elements of $A$ whose ranks in $A$ are from $(l + 1)$th to $(u + 1)$th. Here, the interval ‘$l$: $u$’ is called a rank interval.

**Example 2.** Suppose that a data array $A$ is $\langle 2, 3, 4, 1 \rangle$. Then some possible $A_{lu}$’s and $RBS_{lu}$’s are as follows:

- $A_{0:3} = \langle 2, 3, 4, 1 \rangle$, $RBS_{0:3} = 0110$,
- $A_{0:1} = \langle 3, 4 \rangle$, $RBS_{0:1} = 01$,
- $A_{2:3} = \langle 2, 1 \rangle$, $RBS_{2:3} = 10$.

**Definition 5.** Let $A$ be a data array of size $m$ ($m > 1$). Then the RD-Tree for $A$ is a binary tree such that:

- The root node of the tree is $RBS_{0:m-1}$.
- A leaf node of the tree is $RBS_{lu}$, where $u - l = 1$.
- A left child of $RBS_{lu}$ is $RBS_{l+w-1}$, where $w$ is the weight of $RBS_{lu}$. (Note that the weight of $RBS_{lu}$ is $\lfloor (u - l + 1)/2 \rfloor$.)
- A right child of $RBS_{lu}$ is $RBS_{l+w}$, if the weight of $RBS_{lu}$ is greater than 2. Otherwise, the right child does not exist.

The RD-Tree is basically a binary tree whose node is an RBS with a rank interval. The rank interval of a node is bisected into two subintervals and each of them is used for the node’s two children. Fig. 2 shows an example of the RD-Tree. The root node represents the $RBS_{0:7}$, that is, it indicates the data which are ranked above the half among all data values. The data ‘5’, ‘8’, ‘7’ and ‘6’ are ranked in the top half, so they are selected for the left child in Level 2. In this node (i.e., $RBS_{0:3}$), the data ‘8’ and ‘7’ are ranked above the half among ‘5’, ‘8’, ‘7’ and ‘6’. Finally, the left-most node in Level 3 (i.e., $RBS_{0:1}$) shows that the data ‘8’ is ranked higher than ‘7’, thus we can find that the data ‘8’ is the maximum among all data values. In the figure, the data ranked above the half in each rank interval are underlined for readers’ convenience.
3.2. Search process in the RD-Tree

To perform a search in the RD-Tree, we introduce the concepts of the upper query vector and lower query vector.

**Definition 6.** Given an RBS $V_S$ and a query vector $V_I$, the upper query vector denoted by $V_I^{\text{upper}}$ is defined as $V_I \land V_S$. The positions of 1’s in $V_I^{\text{upper}}$ represent the data that are specified by the query and ranked above (or equal to) the half. Similarly, the lower query vector $V_I^{\text{lower}}$ defined as $V_I \land \neg V_S$ identifies the set of data that are specified by the query and ranked below the half. (Here, ‘$\land$’ and ‘$\neg$’ denote bitwise ‘and’ and ‘negation’ operators respectively.)

Fig. 3 gives an illustration of RBS, $V_I^{\text{upper}}$ and $V_I^{\text{lower}}$ for data array $A$ and a partial-max query vector $V_I$. If the weight of $V_I^{\text{upper}}$ (i.e., the number of 1’s in $V_I^{\text{upper}}$) is greater than zero, then we have only to search the cells specified by $V_I^{\text{upper}}$. It is because the maximum of the cells specified by the query is included in the top half with respect to ranking.

The search process at node $V_S$ in the RD-Tree follows the left child or right child, depending on whether the weight of $V_I^{\text{upper}}$ is greater than zero or not. Now, since the length of a child node is only half of the parent node, we need to modify the query vector in each level appropriately to work for a child node. We define $\text{Half-Project}(V_1, V_2)$ as the operator for construction of a bit vector whose length is the weight of $V_2$.

**Definition 7.** Let $V_1$ and $V_2$ be the bit vectors of size $m$. $\text{Half-Project}(V_1, V_2)$ projects the bits of $V_1$ whose corresponding bits in $V_2$ are specified by ‘1’, while keeping the relative orders among the bits in the result vector the same as those in $V_1$. (See Fig. 4.)

Thus, the branching rule in the node of the RD-Tree is as follows.

- If $|V_I^{\text{upper}}| > 0$, the result of $\text{Half-Project}(V_I^{\text{upper}}, V_S)$ is the query vector that is applied to the left child of a parent node $V_S$.
Otherwise, the result of $\text{Half-Project} \ (V^\text{lower}_I, \neg V^*_S)$ is a query vector to the right child.

**Algorithm 1** (Algorithm for rank-max($I$)).

GetRankMax($I$)  
$I$: a query  
begin  
$V_I$ = ConvertBitVector($I$);  
return (Search_RD_Tree($V_I$, Root RBS of RD_Tree, 0))  
end  

Search_RD_Tree($V_I$, $V_S$, L)  
$V_I$: a query vector  
$V_S$: an RBS  
L: the highest rank in the RBS  
begin  
$V^\text{upper}_I$ = $V_I \land V^*_S$;  
$V^\text{lower}_I$ = $V_I \land \neg V^*_S$;  
if $|V^*_S|$ = 1 and $|V^\text{upper}_I|$ = 1 then return L; end if  
if $|\neg V^*_S|$ = 1 and $|V^\text{upper}_I|$ = 0 then return L + $|V^*_S|$; end if  
if $|V^\text{upper}_I|$ > 0 then  
val = Search_RD_Tree(Half-Project($V^\text{upper}_I$, $V^*_S$), LeftChild RBS, L)  
else  
val = Search_RD_Tree(Half-Project($V^\text{lower}_I$, $\neg V^*_S$), RightChild RBS, L + $|V^*_S|$)  
end  
return val;  
end

Algorithm 1 is the procedure for finding rank-max($I$) of query $I$ on the RD-Tree, and Fig. 5 is an example for explaining the algorithm. The example is based on the RD-Tree in Fig. 2. First, we obtain the query vector ‘11001101’ for the given query and do the ‘and’ operation with the root node ($RBS_{0.7}$) of the tree. Since the weight of $V^\text{upper}_I$ (i.e., $|V^*_I \land RBS_{0.7}|$) is greater than zero, we branch to the left child node ($RBS_{0.3}$). This means that the maximum of the cells specified by the query is located in the top half. When branching to the left child, we recompute the query vector using the Half-Project operator, so the...
new query vector in this level becomes ‘1001’. Second, when we get the result of ‘1001 ∧ 0110’ (0110 is the left child in Level 2), the weight of the result ‘0000’ is zero. Thus, we branch to the right child i.e., $RBS_{2,3}$. Note that, when we branch to the right child, the result of $Half$-$Project(V_l^{lower}, V_S)$ is the query vector. In the leaf level, we apply a new query vector ‘11’, which is from $Half$-$Project(1001, 1001)$, and hence the bigger one of the two cells in $RBS_{2,3}$ becomes the maximum of the query. In consequence, the maximum of the query is $A[R[2]] = 6$.

**Observation 2.** In a *RD-Tree*, we observe the following.

- Let $n_{\text{child}}$ be the length of a child node and $n_{\text{parent}}$ be the length of its parent node. Then, $n_{\text{child}} \leq \lceil n_{\text{parent}} / 2 \rceil$ and the sum of the lengths of two children is $n_{\text{parent}}$.
- The depth of the *RD-Tree* for data array $A$ of size $m$ is $\lceil \log_2 m \rceil$.

According to the above observation, we can see that the RD-Tree requires $m \lceil \log_2 m \rceil$ bits of memory space for representing a data array of size $m$ in the worst case, and we read about $2m$ bits for finding the maximum.

### 3.3. RD-Tree on the disk

When we store the RD-Tree on the disk, we have to access one disk block for a single node even if the node is a very small one, and this becomes a potential performance problem. To prevent the problem, we propose a modified RD-Tree structure in which each leaf node contains information of one block size. For the modified RD-Tree, the second condition of Definition 5 is rewritten as follows.

- When the number of bits of a disk block is $h$, a leaf node of the RD-Tree is $RBS_{l,u}$, where $u - l < h$. 

Fig. 5. Finding rank-max(I) through the RD-Tree.
Also, the algorithm must be modified such that the lines marked by (A) and (B) in Algorithm 1 are replaced by the following lines.

\[
\begin{align*}
\text{if } |V_S| \leq h \text{ and } |V_{\text{upper}}^>| > 0 \text{ then return } L; & \quad \text{endif} \quad \rightarrow (A') \\
\text{if } |V_S| \leq h \text{ and } |V_{\text{upper}}^>| = 0 \text{ then return } L + |V_S|; & \quad \text{endif} \quad \rightarrow (B')
\end{align*}
\]

After accessing the leaf node block, we obtain a rank interval in which the maximum (among the cells specified by the query) is located. Thus, we can find the maximum through a sequential search of the rank index in this interval, which can be processed in the memory without additional disk accesses. Observation 4 and Lemma 1 below address performance issues of the modified RD-Tree.

**Observation 3.** Let \( h \) be the number of bits in a disk block. The depth of the modified RD-Tree is \( \lceil \log_2 m/h \rceil + 1 \), and the amount of disk space for the modified RD-Tree is \( \left( \left( \lceil \log_2 m/h \rceil + 1 \right) \cdot m/h \right) \) blocks in the worst case.

**Lemma 1.** Suppose that the number of bits in a block is \( h \), the size of data array \( A \) is \( m \), and the size of one index slot in the rank index \( R \) is \( i \) bytes. Then, the number of block accesses \( x \) for finding the maximum in the balanced RD-Tree is bounded by: \( 2n \leq x \leq 2n - 1 + 4i \), where \( n \) is the number of blocks for the root node of the RD-Tree i.e., \( n = \lceil m/h \rceil \).

**Proof.** In all cases of finding the partial maximum in the balanced RD-Tree, we have to traverse a path (from the root to a leaf node) of the tree. The number of block accesses \( x \) for the path traversal is \( 2n - 1 \), which is from

\[
\sum_{i=1}^{i=1+\log_2 n} \frac{n}{2^{n-1}}
\]

Here, \( 1 + \log_2 n' \) is the height of the RD-Tree i.e., the path length. And, in the best case, we can find the maximum in one additional block access of the Rank Index. Thus, the lower bound of \( x \) is \( 2n' (= 2n - 1 + 1) \). In the worst case, we have to search all index slots of the Rank Index that the half of a leaf node (of the RD-Tree) indicates. So, \( (8 \times i \times h/2)/h \) block accesses are necessary for the Rank Index search. Therefore, the upper bound of \( x \) is \( 2n - 1 + 4i \). \( \square \)

### 3.4. Processing partial-min queries

Until now, we have developed the rank index, RBS and RD-Tree for partial-max queries. These structures can be easily applied for partial-min queries. In rank index \( R \), we can find the minimum by searching \( R \) sequentially from \( m - 1 \) to 0. In the RD-Tree, we check if \( |V_{\text{lower}}^<| \) is greater than zero. That is, if \( |V_{\text{lower}}^<| > 0 \), we branch to the right child. Otherwise, we branch to the left child.
Algorithm 2 is the procedure for processing partial-min queries in the RD-Tree. Note that the partial-max and partial-min queries are processed in a single RD-Tree, not two separate RD-Trees.

Algorithm 2 (Algorithm for rank-min(I)).
\[ \text{GetRankMin}(I) \]
\[ I: \text{a query} \]
begin
\[ V_I = \text{ConvertBitVector}(I); \]
return (Search_RD_Tree_for_Min(\( V_I \), Root RBS of RD_Tree, m-1))
end

Search_RD_Tree_for_Min(\( V_I, V_S, u \))
\[ V_I: \text{a query vector} \]
\[ V_S: \text{a RBS} \]
\[ u: \text{the lowest rank in the RBS} \]
begin
\[ V_{upper} = V_I \land V_S; \]
\[ V_{lower} = V_I \land \neg V_S; \]
\[ \text{if } |\neg V_S| = 1 \text{ and } |V_{lower}| = 1 \text{ then return } u; \text{ end if } \]
\[ \text{if } |V_S| = 1 \text{ and } |V_{lower}| = 0 \text{ then return } u - |\neg V_S|; \text{ end if } \]
\[ \text{if } |V_{lower}| > 0 \text{ then } \]
\[ \text{val} = \text{Search_RD_Tree_for_Min}=(\text{Half-Project}(V_{lower}, \neg V_S), \text{RightChild RBS, } u) \]
\[ \text{else } \]
\[ \text{val} = \text{Search_RD_Tree_for_Min}=(\text{Half-Project}(V_{upper}, V_S), \text{LeftChild RBS, } u, |\neg V_S|) \]
end if
return val;
end

4. Performance evaluation

In this section, we evaluate the performance of the proposed method through experiments. We use the uniform probability function for generating \( 2^{10} \) experimental data. A block size of 4096 bytes and a data cell size of 4 bytes are used for the experiments. Since as far as we know, there is no previous work that is specific to partial-max/min queries, we compare the following three approaches: (i) sequential searching with the Rank Index only, (ii) finding 2. We have experimented \( 2^{10}, 2^{20} \) and \( 2^{30} \) data. And, we have observed almost the same performance patterns. In the paper, we present the result for \( 2^{10} \) data.
the rank interval through the RD-Tree and searching the Rank Index (RD-Tree + Rank Index), and (iii) searching through the Projection Index [6]. The Projection Index on attribute C of table T consists of a stored sequence of attribute values from C, in order by the row number in T from which the values are extracted.

Fig. 6 shows the number of disk accesses with various cell selectivities of queries. In this experiment, we use a uniform probability for generating partial-max/min queries i.e., a set of specified cells among which we find the maximum/minimum are uniformly distributed over the entire cells. As the figure shows, the ‘Rank Index only’ method and the ‘RD-Tree + Rank Index’ method provide significantly better performance than the ‘Projection Index’ method. The ‘Rank Index only’ method shows a little better performance than the ‘RD-Tree + Rank Index’ method. This is because the queries are generated in a uniform fashion, and hence the maximum (or minimum) is probably located in the high (or low) rank area so that ‘the direct search of the rank index’ is more efficient than ‘the search of the rank index after finding a rank interval of the query through the RD-Tree’.

Fig. 6 is based on the uniform distribution of specified cells with respect to ranking in each partial query. However, the set of specified cells in a partial query is not uniformly distributed with respect to ranking in practice. The following queries give some typical examples: “Find the highest temperature in winter months” and “Find the maximum sales record among the stores that sold less than the average in the past year”. In Fig. 7, as an experimental parameter, we use the rank ratio of a partial-max query, which is defined as
how the query's result is ranked high. For example, if the result of a partial query is within the top 30% among all data values, then the rank ratio of the query is 30%. The ‘RD-Tree + Rank Index’ method shows an almost constant time irrespective of the rank ratio, while the performance of ‘Rank Index only’ method becomes worse with the increase of the rank ratio. This is because increasing the rank ratio causes the increase of sequential search space.

In Fig. 8, we show the storage overhead of the ‘RD-Tree + Rank Index’ method. Y-axis represents the percentage of additional disk space compared with the projection index method. The figure shows that the space overhead of
our approach is not much, and hence the approach is affordable to support a huge data cube.

5. Conclusion

In this paper, we have considered the efficient processing of the partial-max/min queries in OLAP applications. The partial-max/min query is a class of queries that are applied to categorical attributes, and are highly applicable in various decision support processes. For processing partial-max/min aggregates, we have first proposed the concept of Rank Index. Then, we have proposed the RD-Tree for efficiently finding the rank of the given query. The RD-Tree is a binary tree where each node is an RBS. The RBS is a kind of bitmap index that classifies the data into two groups: the highly ranked group and the lowly ranked group. In Algorithms 1 and 2, we have shown that both the partial-max and partial-min queries can be processed in a single RD-Tree.

In the performance experiments, we have found that the proposed method efficiently finds the maximum and minimum of a partial query. With not much space overhead, our approach provides a robust performance regardless of the various types of queries. Basically, the complexity for the construction of the RD-Tree is $O(n \log n)$, which is the sorting complexity. Thus, in most OLAP applications, we can use the RD-Tree with periodic reconstruction for the updated data. However, for some highly dynamic environments, the reconstruction approach may not be feasible. As future work, we are investigating the maintenance algorithm which adaptively reconstructs a part of the RD-Tree for updated and newly inserted data values.

References