An efficient, robust method for processing of partial top-k/bottom-k queries using the RD-Tree in OLAP

Yon Dohn Chung \textsuperscript{a,}*  
Woo Suk Yang \textsuperscript{b}  
Myoung Ho Kim \textsuperscript{c}

\textsuperscript{a} Department of Computer Science and Engineering, Korea University, Anam-Dong, Seongbuk-Gu, Seoul, 136-713, South Korea  
\textsuperscript{b} R&D Lab., ARA Network Technologies Co., Ltd., Gajeong-Dong, Yuseong-Gu, Daejon, 305-350, South Korea  
\textsuperscript{c} Department of Computer Science, KAIST, Guseong-Dong, Yuseong-Gu, Daejeon, 305-701, South Korea

Received 12 May 2006; received in revised form 11 September 2006; accepted 4 October 2006  
Available online 30 November 2006

Abstract

Online analytical processing (OLAP) is a widely used technology for facilitating decision support applications. In the paper, we consider partial aggregation queries, especially for \textit{partial top-k/bottom-k}, which retrieve the top/bottom-k records among the specified cells of the given query. For the efficient processing of partial ranking queries, this paper proposes a set of algorithms using the \textit{RD-Tree}, which is a data structure previously proposed for \textit{partial max/min} queries. Through experiments with real data, we show the efficiency, robustness, and low storage overhead of the proposed method.

© 2006 Published by Elsevier B.V.

Keywords: Aggregation; Partial ranking queries; RD-Tree; Top-k/bottom-k; OLAP; Decision support applications

1. Introduction and problem statement

Aggregation, which is a popular operation in OLAP, is to compute an aggregate value of a \textit{measure} attribute for a collection of data records specified with a combination of dimension attributes. Many approaches have been proposed in the past for aggregation processing \cite{1-12}, such as range-sum, range-max/min, partial-max/min, and partial-sum/count queries. This paper focuses on \textit{partial top-k/bottom-k query} processing in OLAP. A partial top-k/bottom-k query retrieves the top or bottom-k values among the cells specified by the query. For example, consider an insurance data cube with 3 dimension attributes (‘state’, ‘time period’, and ‘insurance type’) and one measure attribute (‘revenue’). A partial top-10 query may ask for the largest 10 revenue values from the states of ‘California’, ‘Texas’, and ‘Florida’, for the 3rd quarters of ‘2001’, ‘2003’, and ‘2005’, and for the ‘Life’ and ‘Health’ insurance types.

\textbf{Definition 1.} \textit{[Partial top-k/bottom-k problem]}

Let $A$ be a data array of size $m$,\textsuperscript{1} indexed from 0 to $m - 1$, and $M = \{0, 1, \ldots, m-1\}$ be the set of indices of $A$. Given a partial query vector, $I$, which consists of elements of $M$, the problem of finding the top or bottom-k values

\* Corresponding author.  
E-mail addresses: ydchung@korea.ac.kr (Y.D. Chung),  
wsyang@aratech.co.kr (W.S. Yang), mhkim@dbserver.kaist.ac.kr (M.H. Kim).

0167-9236/$-$ see front matter © 2006 Published by Elsevier B.V.  
doi:10.1016/j.dss.2006.10.001

\textsuperscript{1} We assume that the data cube is a one dimensional data array. Since a multi-dimensional data cube can be mapped into a one-dimensional array, one dimensional data array notation can be used without loss of generality.
from the partial query vector $I$, denoted by partial-top-$k$ $(A, I)$ and partial-bottom-$k(A, I)$, is:

- \text{partial - top} - k(A, I) = \text{top} - k\{A[i] | i \in I\};
- \text{partial - bottom} - k(A, I) = \text{bottom} - k\{A[i] | i \in I\}.

**Example 1.** Suppose a data array $A = \{4, 7, 3, 6, 9, 1, 20, 1, 17, 19, 32, 5\}$, where $M = \{0, 1, \ldots, 11\}$, and a partial query vector $I^2 = \{0, 1, 3, 5\}$ are given. Then, partial-top-$2(A, I) = \{7, 6\}$, and partial-bottom-$2(A, I) = \{2, 4\}.

2. Background and motivation

In our previous work [13], we proposed several methods for the efficient processing of partial max/min queries. This section briefly describes these methods and their limitations in processing partial top-$k$/bottom-$k$ queries.

2.1. Rank Index and RD-Tree (rank decision tree)

**Definition 2.** [Rank Index]

The Rank Index $R$ for a data array $A$ of size $m$ is an array of size $m$ such that the value of $R[i]$ is an index for a data array $A$ that satisfies the inequality: $A[R[i]] \geq A[R[i+1]]$, $i=0, \ldots, m-1$.

Using this Rank Index, the top-$k$ values in data array $A$ are represented by $A[R[0]], A[R[1]], \ldots,$ and $A[R[k-1]]$, in this order. However, in order to process partial max/min queries i.e., find the max/min values among a selected subset of the data array, the entire Rank Index may need to be scanned. In order to avoid this inefficient scan, we proposed a binary tree structure called the RD-Tree. Each node in the RD-tree has a bit-vector, called a Rank Bisection Signature (RBS), where each bit of the RBS indicates whether the value of the corresponding cell is above or equal to the midpoint or not. Note that by definition in each RBS, half of the bits have values of 1 and the other half of the bits have values of 0.

Let there be a data array $A$ of size $m$ and its Rank Index $R$. The RBS of $A$ is a bit vector $V_S = (b_0, b_1, \ldots, b_{m-1})$, where $b_{R[i]} = 1$ if $0 \leq i < \lfloor m/2 \rfloor$; 0 otherwise. $A_{\text{u-l}}$, which is a subset of $A$, is defined as an array of size $u-l+1$ such that $A_{\text{u-l}}[i] = A[R[l+i]]$, $i=0, \ldots, u-1$. Note that $A_{\text{u-l}}$ denotes the $u-l+1$ elements of $A$ whose ranks in $A$ are from $(l+1)$ to $(u+1)$. $A_{\text{u-l}}$ is defined as an RBS applied to $A_{\text{u-l}}$.

Here, we call ‘$l$–$u$’ a rank interval. For example, suppose that a data array $A$ is $\{2, 3, 4, 1\}$. Then $A_{[0:3]} = \{2, 3, 4\}$, RBS$_{[0:3]} = \{010\}$ and $A_{[0:1]} = \{3, 4\}$, RBS$_{[0:1]} = \{01\}$.

**Definition 3.** [RD-Tree]

The RD-Tree of a data array of size $m > 1$ is a binary tree such that:

- The root node is $RBS_{[0:m-1]}$, and a leaf node is $RBS_{[u:l]}$ where $u-l = 1^3$.
- The left child of $RBS_{[u:l]}$ is $RBS_{[u:l+w-1]}$, where $w$ is $(u-1+l)/2$.
- The right child of $RBS_{[u:l]}$ is $RBS_{[u:w-1]}$, if $u-1+l > 2$. Otherwise, the right child does not exist.

Fig. 1 shows an example of an RD-Tree, where the values ranked above the midpoint in each rank interval are underlined. Let us define a few more notations in order to explain the search procedure that uses the RD-Tree. Suppose $V_I$ is the query vector, and $V_S$ is the current RBS in the RD-Tree. The upper query vector $V_I^{\text{upper}}$ is defined by $V_I \land V_S$, and the lower query vector $V_I^{\text{lower}}$ is defined by $V_I \land \neg V_S$ where $\land$, ‘$\lor$‘ and ‘$\neg$’ denote bitwise ‘AND’ and ‘NEGATION’. The positions of the 1’s in $V_I^{\text{upper}}$ (or $V_I^{\text{lower}}$) represent the data specified by $V_I$ and ranked above (or below) or at the halfway mark. To find the maximum (or minimum), we follow the left child (or the right child) depending on the weight of $V_I \land V_S$, which is denoted by $|V_I \land V_S|$. (The weight of a bit vector means the number of 1’s in the vector.) Since the size of the child RBS is one half of that of the parent, we need to reduce the size of the query vector accordingly.

**Definition 4.** [Half-Project]

Let $V_I$ be the current query vector and $V_2$ be the current RBS. $\text{Half-Project}(V_I, V_2)$ projects the bits of $V_I$ whose corresponding bits in $V_2$ have ‘1’ values, while keeping the relative orders among the bits in the result vector the same as those in $V_I$.

Thus, when branching to child nodes, we recompute the query vector $V_I$ as ‘Half-Project($V_I^{\text{upper}}, V_S$)’ for the left child, and ‘Half-Project($V_I^{\text{lower}}, \neg V_S$)’ for the right child.

Fig. 2 shows an example of processing partial max queries i.e., finding the maximum among the cells.

---

2 We represent the query specification as a vector for convenience, because it is used as a kind of bitmap index in the paper, although it is defined as a subset of $M$.

3 When we use the RD-Tree on disks, this condition is changed as follows: ‘$u-l<h$’, where $h$ is the number of bits of one disk block.
specified by a query. (For partial-min queries, refer to [13].) First, using the query vector \( V_1 '11001101,' \) we perform the bitwise ‘AND’ operation with the root node. The weight of the result of the ‘AND’ operation is greater than 0, which means that the maximum of the cells specified by the query is located in the top half. Thus, we branch to the left child (RBS0:3). Now, the query vector is half-reduced to ‘1001’, which comes from Half-Project(‘01000001’, ‘01100011’). Since the weight of (‘1001’ AND ‘0110’) is zero, we branch to the right child i.e., RBS2:3. The new query vector is ‘11’, which is obtained from Half-Project(\( V'_1 \) lower, \( \neg V'_3 \)) (=Half-Project (1001, 1001)). By comparing the query vector ‘11’ and ‘01’ (RBS2:3), we find that the maximum of the given query specification is \( A[R[2]]=6 \).

2.2. Motivation

Previous approaches, such as the Projection Index [10], Rank Index [13], and RD-Tree [13], were not targeted specifically for partial top-k/bottom-k queries. The Projection Index was for general aggregation, the Rank Index for general ranking queries, and the RD-Tree for partial max/min queries. Though they may not be very efficient, all of them can be applied to partial top-k/bottom-k query processing. Below is the estimated cost analysis of these methods for partial top-k queries based on a data array of size \( m \). (The costs of bottom-k queries are alike.)

2.2.1. Projection index

In this method, we have to obtain the values of the specified cells (from the disk), and sort them (in memory) into top-k. Thus, the cost will be the sum of the disk access cost required for retrieving the specified cells via the Projection Index, and the in-memory sorting cost for the selected cells.

2.2.2. Rank index

In order to process the partial top-k queries via the Rank Index, we have to search the index sequentially, and check whether each cell is specified by the query or not. Since the Rank Index is itself a sorted result, this operation does not incur any additional cost for sorting. In the case where the \( k \)-th value is the minimum, the entire index has to be scanned.

2.2.3. RD-Tree + rank index

The RD-Tree supports the efficient finding of partial max/min values. Hence, we may process the partial top-k queries by first finding the maximum via the RD-Tree, and then sequentially searching the remaining \( k-1 \) values via the Rank Index. Therefore, the cost of this combined approach will be the sum of the cost required for finding the maximal cell via the RD-Tree and the cost required for the sequential search and comparison on the Rank Index. It is observed that the cost incurred by the sequential search of the Rank

Fig. 1. An example of an RD-Tree.

Fig. 2. An example of finding the partial maximum in the RD-Tree.
3. The proposed method for partial top-k/bottom-k queries

In this section, we propose a method for the efficient processing of partial top-k/bottom-k queries, in which we basically use the RD-Tree structure for storing the rank information of the base data. Note that, since the partial max/min query is subsumed by a partial top-k/bottom-k query, the proposed method can also efficiently process partial max/min queries.

3.1. The algorithm

In searching through the RD-Tree, we determine whether the max/min value for the query is located above or at the halfway point of the given interval or not, based on the weight of \((V_I \Lambda V_S)\), where \(V_I\) is a partial query bit vector (that is half-reduced when going down the tree) and \(V_S\) is an RBS node of the RD-Tree: If the weight is greater than or equal to ‘1’, the maximum value is located in the top half. Otherwise, it is in the bottom half. Thus, to find the partial max/min value, it is sufficient to branch to only one of two children. However, in the case of top-k/bottom-k queries, we have to consider those cases where \(i\) results are located in the left subtree and the remaining \(k-i\) results in the right subtree.

In order to process partial top-k/bottom-k queries using the RD-Tree, branching to both children should be handled as follows. (In this paper, we mainly focus on the processing of top-k queries. The bottom-k query processing is a simple modification of top-k query processing.)

**Observation 1.** Suppose that \(V_S\) is an RBS node of the RD-Tree, and \(V_I\) is a partial query bit vector for a partial top-k query. When comparing \(V_S\) and \(V_I\) during the search of the RD-Tree:

- If \(|V_I^{\text{upper}}|=k\), \(V_I^{\text{upper}}\) represents the result of the top-k query.
- If \(|V_I^{\text{upper}}|>k\), the result of the top-k query is obtained by searching through only the left subtree of the current node with the half-reduced query vector \(V_I^{\text{upper}}\).
- If \(|V_I^{\text{upper}}|<k\), the result of the top-k query is obtained by calculating the sum of the \(|V_I^{\text{upper}}|\) results from the search of the left subtree and the \(k-|V_I^{\text{upper}}|\) results from the right subtree. The upper query vector \(V_I^{\text{upper}}\) is used for searching the left subtree, and the lower query vector \(V_I^{\text{lower}}\) is used for the right subtree.

Here, when summing up the partial results of the nested calls, we need to restore the bit position information that was Half-Projected.

**Definition 5.** [Reverse-Half-Project: RHP]

Suppose there are two bit vectors \(V_1\) (with a size of \(m\) bits) and \(V_2\) (with a size of \(m/2\) bits). Then, Reverse-Half-Project is to find an \(m\)-bit bit vector \(V\) such that \(V_2=\text{Half-Project}(V_1, V_2)\) and \(|V \Lambda \neg V_1|=0\).

For example, the result of Reverse-Half-Project ('1001', '01100011') is '01000001', since '001'=Half-Project('01000001', '01100011') and '01000001' \(\Lambda \neg \) '01100011'=0.
Fig. 3 shows the proposed algorithm used for the processing of the partial top-k query using the RD-Tree. (In the Appendix, we describe the algorithm for bottom-k queries (Fig. A-1). The bottom-k queries are also processed on the same RD-Tree.) Lines 9 and 10 handle the 3rd case in Observation 1. Let us show an example of the use of this algorithm. Fig. 4 illustrates a part of the search process for selecting the top 4 values among 8 cells specified by $V_I$ (i.e., $|V_I|=8$) on a data array of 16 cells. In the root node (a), we find that the top 2 values are located in the top half (i.e., the left subtree in Level 2), while the remaining 2 values are in the bottom half (Line 8 of the algorithm).

Fig. 4. An example of RD-Tree traversal for finding top-4 values.

Fig. 5. Result generation for Fig. 4.
Fig. 4 shows the part of the search corresponding to the right subtree. In the right child node in Level 2, the partial query vector is half-reduced such that Half-Project($V_{lower}$, ¬(Root node RBS)) = '1110110', and $k$ becomes two (b). Since the weight of the new query vector is 3, we branch to the left child in Level 3 with the same value of $k$ (Lines 5 and 6 of the algorithm). In Level 3, we reach a terminal condition ((c) in the figure and Line 3 of the algorithm). Then, the result bit vectors are returned to the caller functions. Fig. 5 illustrates the result generation process – (a), (b), (c) in this order – of Fig. 4. The result bit vector returned from Line 3 is Reverse-Half-Projected in Line 7 (and also in Line 10), and is finally bitwise OR-ed with $V_{upper}$ in Line 10.

**Property 1.** For an RD-Tree of $m$ data values, the number of bits which need to be accessed to process a partial top-$k$ query is at most $2m$.

---

4. Implementation issues

4.1. Storage structure

Since the RD-Tree is basically a balanced binary tree and the size of a node is fixed (according to the level), the children of an RD-Tree node can be accessed by computing offsets without the need for links. Also, in storing RD-Tree nodes on disks, the order of node placement could affect the overall performance. While searching the RD-Tree, we traverse the RD-Tree nodes in the direction of the root to the leaves. Therefore, it would be better to place the RD-Tree nodes clustered based on the order of traversal i.e., depth-first order. Then, the address for a child node is calculated as follows:

**Definition 6.** The size $\text{subtree-size}(m)$ of a subtree rooted by an $m$-bit RD-Tree node is $m(d+1)-2d$, where $d=\lceil \log_2 m \rceil$. Then, the addresses of the child nodes ($\text{ADDR}_{\text{left-child}}(p, n)$ and $\text{ADDR}_{\text{right-child}}(p, n)$) of a node, whose address and size are $p$ and $n$, respectively, are:

\[
\text{ADDR}_{\text{left-child}}(p, n) = p + n; \text{ADDR}_{\text{right-child}}(p, n) = p + n + \text{subtree-size}(n/2)
\]

4.2. Optimizing bit operations

The proposed algorithms use bit operations, such as ‘Half-Project’, ‘Reverse-Half-Project’, and ‘Counting 1’s. However, some of these operations are not directly supported by simple CPU instructions, and thus their processing cost could be high. To speed up processing, we propose a pre-computation approach, where a 2-dimensional array stores the results of $|V_{upper}|$, $|V_{lower}|$, $|V_3|$, $|V_{left}|$, $|V_{right}|$, and $|V_{root}|$. Fig. 6. Average time for processing a top-100 query.

Fig. 7. Number of disk accesses.

Fig. 8. Query processing time according to the value of $k$ (in top-$k$).
and ‘Half-Project (\(|V_I|_{\text{upper}}, V_S\))’ for all combinations of \(V_I\) and \(V_S\).

\[
\text{struct} \{ \text{/* for all pairs of } (V_I, V_S) \text{,} \\
\text{uint8_t WEIGHT_OF_VI_UPPER;} \\
\text{uint8_t WEIGHT_OF_VI_LOWER;} \\
\text{uint8_t WEIGHT_OF_VS;} \\
\text{uint8_t HALF_PROJ} \} \text{ PreComputedArray}[256][256]; / * for all pairs of } (V_I, V_S); 256=2^8 */
\]

In the case where \(V_I\) and \(V_S\) are long (more than 8 bits), they are partitioned into 8 bit fragments, and each pair of fragments is applied to the array. By merging the results of the fragments, we can obtain the full result. The Reverse-Half-Project operation is also pre-computed similarly. This approach significantly reduces the CPU cost with reasonable memory overhead (256 KB = 256 × 256 × 4 Bytes), as shown in the next section.

5. Performance evaluation

We implemented the proposed method and experimentally compared it with the other methods introduced in Section 2 — the (1) Projection Index, (2) Rank Index, (3) RD-Tree + RI (the RD-Tree with the Rank Index), and (4) RD-Tree + RHP (the RD-Tree with the Reverse-Half-Project operation — the proposed method). We implemented the proposed method in two versions (with and without the optimization of the bit operations).

We used a Linux (2.6.11, 32bit mode) server system with one AMD Opteron 242 1.5 GHz CPU, 2GB main memory, and an Ultra 320 SCSI HDD (10,000 RPM). We used a real data set for the experiment, which is the network traffic log acquired from a network traffic collector of an ISP company. The collector stores the real-time traffic records, each of which consists of <source/destination IP, source/destination Port, application type, transfer bytes, transfer packets>. We used 200 million log records accumulated for 15 days to construct a data cube consisting of (day, IP, application type, transfer bytes). The first 3 fields are dimension attributes and the last field is the measure attribute. A sample query used in the experiment is “Select the top 100 records with respect to the number of bytes transferred, where the source is in Country-A, and its application type is a worm virus.”

We made up a query

---

4 We have changed the name of the country due to security reasons.
vector based on the IP addresses of Country-A and the application tags for the worm virus.

Fig. 6 shows the performance of the five methods with different numbers of data records. It can be seen that the RD-Tree-based methods are much faster than the Projection Index method. The optimization strategy for bit operations improves the performance by 3 times.

Fig. 7 shows the number of disk accesses required for processing the given query, where the size of one disk block is 4096 bytes. The RD-Tree-based methods access less disk blocks than the Projection Index or Rank Index methods. In addition, they are highly scalable i.e., they perform well for a huge data cube.

Fig. 8 shows the results of experiments with a variety of k values. We used randomly generated query vectors with a selectivity of 50%, and measured the average query processing time. Unlike the other methods, the proposed method is robust to high k values. Fig. 9 shows the result of experiments in which we varied the rank ratio of the selected values between 10% and 90%. Here, a 10% ratio means the selected values are ranked in the top 10% of the entire data set. As mentioned in Section 2, the Rank Index method performed poorly when the selected values were ranked low (or high in the case of bottom-k queries). On the other hand, the proposed method is almost independent of the rank ratio of the query results.

Fig. 10 shows the effect of the query selectivity on the overall performance. We measured the average query processing time and the number of disk accesses for the Projection Index and RDT+RHP methods. The proposed method provides much better performance, regardless of the selectivity values. Also, as illustrated in Fig. 11, the proposed method requires the least disk space among the methods tested herein.

6. Conclusion

OLAP applications make heavy use of aggregation functions for decision support analysis. Among the various aggregation functions, ranking-related operations including ‘max/min’ and ‘top-k/bottom-k’ are the most popularly used. This paper specifically deals with partial top-k/bottom-k queries that retrieve the top or bottom-k values among the cells specified by the query. For this purpose, we used the RD-Tree structure which was previously developed for partial max/min queries. We proposed a new operation called ‘Reverse-Half-Project’ and also a set of algorithms for finding top and bottom-k values. Through experiments with real data, we demonstrated that the proposed method performs more efficiently and robustly than other previously proposed methods.

Acknowledgement

This work was supported by the Korea Research Foundation Grant funded by the Korean Government (MOEHRD) (KRF-2006-D00463).

Appendix A. The partial bottom-k query processing algorithm

The structures of the RD-Tree and the rank bisection signature (RBS) are symmetric. Also, the upper part and the lower part of an RBS can be exchanged through the operation of 1’s complement. Therefore, we can easily process the partial bottom-k queries using the same RD-Tree (which was used for processing partial top-k queries) like follows:

```
GetBottomK(V_r, V_s, k)
V_r: a query bit vector
V_s: an RBS (initially set as the root RBS of the RD-Tree)
k: the number of values to be selected
Begin
1) V_upper = V_r ∧ V_s;
2) V_lower = V_r ∧ ¬V_s;
3) if |V_lower| < k then return V_lower, end if
4) if V_lower ∧ k and |V_upper| < 0 then return V_upper, end if
5) if |V_lower| > k then
6) V_r = GetBottomK(Half-Project(V_r, V_s), RightChildRBS, k)
7) return Reverse-Half-Project(V_r, V_s)
8) else
9) V_r = GetBottomK(Half-Project(V_r, V_s), LeftChildRBS, k-|V_lower|)
10) return V_r ∧ Reverse-Half-Project(V_r, V_s)
End
```

Fig. A-1. The algorithm for processing partial bottom-k queries.
References


Yon Dohn Chung is an assistant professor in the department of Computer Science and Engineering, Korea University, Seoul, Korea. He received his BS degree in Computer Science from the Korea University in 1994, and his MS and PhD degrees in Computer Science from Korea Advanced Institute of Science and Technology (KAIST), Daejon, Korea, in 1996 and 2000, respectively. Before joining the Korea University, he worked as a post-doc. and a research professor in KAIST during 2000–2002, and he was a faculty member of the department of Computer Engineering, Dongguk University, Seoul, Korea. His research interests include data warehouses, mobile databases, and distributed systems.

Woo Suk Yang is the technical director of ARA technologies’ R&D Lab, Daejon, Korea. He received his BS, MS, and PhD degrees in Computer Science from KAIST, in 1993, 1995, and 2006, respectively. His research interests include data warehouses, database systems, networks, and web-based systems.

Myoung Ho Kim received his BS and MS degrees in Computer Engineering from the Seoul National University, Seoul Korea, in 1982 and 1984, respectively, and his PhD degree in Computer Science from Michigan State University, East Lansing, MI, in 1989. In 1989, he joined the faculty of the Department of Computer Science at KAIST, Daejon, Korea, where currently he is a professor. His research interests include database systems, data stream processing, sensor networks, workflow, XML and distributed processing. He is a member of the ACM and IEEE Computer Society.