THE NATIONAL UNIVERSITY
of SINGAPORE

Founded 1905

School *of* Computing
Lower Kent Ridge Road, Singapore 119260

TRA7/01

Adaptive Pre-computed Partition Top Method for Range Top-k Queries in OLAP Data Cubes

Zheng Xuan LOH, Tok Wang LING, Chuan Heng ANG, Sin Yeung LEE and Hua-Gang LI

July 2001

Technical Report

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

Ivan PNG
Dean of School

Adaptive Pre-computed Partition Top Method for Range Top-k Queries in OLAP Data Cubes

Zheng Xuan Loh Tok Wang Ling Chuan Heng Ang
Sin Yeung Lee Hua-Gang Li
School of Computing
National University of Singapore
S16, 3 Science Drive 2,
Singapore 117543
Email: {lohzheng, lingtw, angch, jlee, lihuagan}@comp.nus.edu.sg

Abstract

A range top-k query finds the top-k maximum values over all selected cells of an On-Line Analytical Processing (OLAP) data cube where the selection is specified by providing ranges of contiguous values for each dimension. The naïve method answers a range query by accessing every individual cell in the data cube. Therefore, the naïve method is very costly and is exponentially proportional to the number of dimensions of the data cube. Here, we propose a new method for handling the range top-k queries efficiently, termed the Adaptive Pre-computed Partition Top method ($\mathcal{APPT}$). This method partitions the given data cube and stores r pre-computed maximum values with the corresponding locations over partitioned sub-blocks. Furthermore, an area termed overflow array stores additional maximum values for sub-block, which requires more than r maximum values for some range queries. By pre-storing the additional maximum values in the overflow array, the response time for the subsequent queries with the similar ranges will be significantly reduced. The experiment results exhibit vast improvement in terms of the response time of our $\mathcal{APPT}$ method as compared to the naïve method. Our method promises an average update cost of $(1 + t)$ total disk accesses, where t is the average number of IO blocks in the overflow array for each sub-block. This method, which is equipped with intelligence and self-learning capability, is able to maintain the value of t below 0.5. Moreover, the $\mathcal{APPT}$ method can be efficiently applied on uniform and non-uniform data distributions in both dense and sparse OLAP data cubes.

1 Introduction

On-Line Analytical Processing (OLAP) uses a multi-dimensional view of aggregate data to provide quick access to strategic information for further analysis. An increasingly popular data model for OLAP applications is the multi-dimensional database (MDDB), also known as data cube. A data cube [GBLP96] is constructed from a subset of attributes in the database. Certain attributes are chosen to be `measure attributes`, i.e., the attributes whose values are of interest. The remaining attributes, are referred to as `dimensions` or `functional attributes`. For instance, consider a data cube maintained by a supermarket, the data can be stored in a cube having three dimensions, YEAR, PRODUCT_TYPE and BRANCH. The value in each cube, the measure attribute, will be the actual SALES.

Using the data cube model, many range OLAP queries [AMS97] can be formulated. In particular, we propose a new pre-computation technique for a class of OLAP queries called `range top-`k `queries`. A range top-k query finds the top-k values in the given range. A range top-k query example from the above supermarket data cube is to find the top-10 best selling products for January 2001 in all western branch. The most direct approach to answer this query is naïve method which is to scan all the involved cells in the data cube, compare all the cells and take the top-10 values as the answer. However, the cost of access is proportional to the number of involved cells and the number of cells in a data cube is exponentially proportional to its dimensions. In an interactive exploration of a data cube, the response time is very crucial. It is imperative to have a system with fast response time. Our focus in this paper is to develop algorithm, which reduce the response time of queries and updates significantly.

Related Work Considerable research has been done in the database community to improve the range query for aggregate function SUM [GAAS99, CI99] and MAX/MIN [AMS97, KLKL98, LLL00a, LLL00b].

In [AMS97], `balanced hierarchical tree structures` are constructed for storing pre-computed maximum values. However, when the dimension of the data cube is larger than one, the number of nodes to be visited increases significantly. Moreover, if the cardinalities of domains are quite different in size, the height of a tree is up to the largest domain.

In [KLKL98], the concept of `maximal cover` and `maximal cover network` are proposed. The number of maximum cover is highly dependent on the content of a data cube. In addition, the storage requirement and also the building time of the maximal cover network is quite high.

The `hierarchical compact cube` [LLL00a] uses a hierarchical structure which stores the maximum value of all the children sub-cubes with the locations. The query algorithm starts from the top most rank of the hierarchical compact cube and trace down the cube if the maximum value is outside the boundary of the cube. If the hierarchical compact cube method were to be used for handling range top-k queries, the probability that tracing down to the original data cube is very high.

The `block-based relative prefix max` approach by [LLL00b] makes use of blocked pre-computed max cube and location pre-computed cube to answer range-max queries. The location pre-computed cube stores pre-computed maximum value over partitions with the location of the maximum value. The blocked pre-computed max cube partitions the given data cube and stores the maximum values, which are in the range of the cell with the lowest dimension index, to the current cell in the partition. This method is high in storage requirement specially for the blocked pre-computed max cube. However, the blocked pre-computed max cube only stores the maximum value for a region, and the next maximum value cannot be derived.

Despite all these work on range-max/min queries, they cannot be applied directly to the range top-k query. In particular, most of the existing methods explore the idea that, given two disjoint regions A and B, if it is known that max(A) > max(B), region B can be ignored directly. However, for the case of top-k query where $k > 1$, even if it is known that max(A) > max(B), the effort of scanning the original data cube cannot be waived since the second maximum value in region A cannot be guaranteed greater than max(B).

Optimizations of top-k queries were studied in [DR99, CG99], however, the concentration are varied from our study for range top-k queries in OLAP data cubes. In [DR99], an approach for answering "top-k" queries is presented by augmenting the query with an additional selection that prunes away the unwanted portion of the answer set. However, this method runs a risk of restarting the execution of the query if the selection returns fewer than the desired number of answers.

In [CG99], the problem of evaluating "top-k" selection queries is being studied. Instead of top-k maximum values studied in our paper, the "top-k" in [CG99] implies the top-k points which are nearest to a point queries along dimension attributes. Hence the problem solved in [CG99] is totally different from what we studied in our paper.

In a range top-k query, it is possible to minimize the chance of scanning the original data cube by keeping more than one value for each region. The intuitive idea behind this is: Given

two disjoint regions A and B, if it is known that a_1, a_2, a_3, b_1, b_2 and b_3 are the top-3 values for region A and B, respectively and the descending order of these values are a_1, b_1, a_2, b_2, a_3 and b_3. To answer a top-3 query, we may take the top-3 values that are known, a_1, b_1 and a_2, as the result without further scanning region A and B.

Paper Organization The remainder of the paper is organized as follows. In Section 2, we give a basic model for the $\mathcal{APPT}$ method. The auxiliary data structures used in the $\mathcal{APPT}$ method together with the query and update algorithm are discussed in Section 3. In addition, an example is given to better illustrate the query algorithm. The experiment results are presented and the performance are analyzed in Section 4. In Section 5, we discuss issues on the $\mathcal{APPT}$ method. Finally, Section 6 concludes the paper and the future work is stated.

2 The Model

Definition 2.1 A `data cube` $\mathcal{DC}$ of d dimension, is a d-dimensional array. Let $D = \{1, 2, \ldots, d\}$ denote the set of `dimensions`. Each dimension has a size n_i, which represents the number of distinct values in the domain at that dimension. Thus, a d-dimensional data cube can be represented by a d-dimensional array of size $n_1 \times n_2 \times \ldots \times n_d$, where $n_i \geq 2$, $i \in D$. The total size of the data cube is $\prod_{i=1}^{d} n_i$. Each entry in the data cube is called a `cell`.

	0	1	2	3	4	5	6	7	8	9	10
0	32	53	97	94	82	40	75	88	98	42	4
1	53	43	96	67	30	93	79	90	97	73	76
2	8	61	96	91	99	36	97	37	1	10	36
3	16	58	62	70	52	97	61	95	80	13	79
4	84	84	37	38	11	71	54	54	72	74	40
5	54	5	48	72	93	34	71	51	83	5	35
6	85	18	95	78	78	17	16	18	43	18	65
7	71	77	39	72	10	87	79	33	48	10	4
8	38	47	79	5	91	96	50	71	66	70	82
9	51	39	73	95	97	47	94	45	92	53	52
10	0	9	98	84	79	77	9	94	65	86	80
11	43	46	18	52	89	93	89	90	85	29	76

Figure 1: A 2-dimensional data cube

Example 2.1 Figure 1 shows a 2-dimensional data cube. The size of the first dimension (rep-

resented as row) is 11 and the second dimension (represented as column) is of size 12. The total size of the data cube is $(11 \times 12) = 132$ while the storage requirement is $(132 \times 4) = 528$ bytes assuming each cell in the data cube is 4 bytes.

Definition 2.2 With respect to a data cube $\mathcal{DC}$ of d dimensions, a `range query` can be specified as $(l_1 : h_1, \ldots, l_d : h_d)$ where l_i and h_i $(1 \leq i \leq d)$ denote the lowest and highest bound of the range query in each dimension of a data cube, respectively.

Example 2.2 In Figure 1, the shaded area represents the range query (3:7, 3:10).

Definition 2.3 Given a data cube $\mathcal{DC}$ of d dimensions and the size of each dimension is n_i $(1 \leq i \leq d)$, with d partition factors $b_1, \ldots, b_d$, the data cube can be partitioned into $\prod_{i=1}^{d}(\lceil n_i/b_i \rceil)$ disjoint sub-regions known as `sub-blocks`. Sub-blocks covered by range queries can be classified into two types, assuming the lowest index of the first cell of a sub-block is $a_1, \ldots, a_d$ and the highest index of the last cell of a sub-block is $e_1, \ldots, e_d$.

1. It is a `full block` when it satisfies one of the following conditions:
   For each i where $1 \leq i \leq d$

   Case (1) $\lfloor a_i/b_i \rfloor = \lfloor (n_i - 1)/b_i \rfloor : e_i = n_i - 1$ and $a_i = \lfloor a_i/b_i \rfloor b_i$, i.e., the sub-block is the last partition for the i^{th} dimension

   Case (2) $\lfloor a_i/b_i \rfloor \neq \lfloor (n_i - 1)/b_i \rfloor : e_i - a_i + 1 = b_i$

2. Otherwise, it is a `partial block`.

Example 2.3 Using a partition factor 4 for both dimensions, the data cube in Figure 1 is partitioned into $\lceil 11/4 \rceil \times \lceil 12/4 \rceil = 9$ sub-blocks. With the range query in Figure 1, the sub-blocks (4:7, 4:7) and (4:7, 8:10) are full block sub-blocks, whereas, the sub-blocks (3:3, 3:3), (3:3, 4:7), (3:3, 8:10) and (4:7, 3:3) are partial block sub-blocks.

3 The $\mathcal{APPT}$ Method

In order to reduce the chance of accessing the original data cube and thus speeding up the range top-k queries, we present a new block-based method called `the Adaptive Pre-computed`

`Partition Top method` ($\mathcal{APPT}$). The $\mathcal{APPT}$ method is proposed to handle both uniform and non-uniform data distribution in OLAP data cubes. In this context, a data cube is said to have `uniform data distribution` if all the sub-blocks in the cube have the same probability containing large values that will be contributed to the answer of the queries. In contrast, a `non-uniform data distribution` data cube contains some sub-blocks with a larger number of maximum values and most of the answers to the queries are inside these sub-blocks. For instance, the products sold in a supermarket include food products, such as milk, bread and cheese and household products, such as pan, kettle and plate. With the supermarket data cube mentioned previously, a range top-k query may request the top-10 best selling products in January 2001 of all western branches. In this case, food products might contribute most of the answers to the query.

3.1 The Data Structures

The $\mathcal{APPT}$ method uses two auxiliary data structures: a `Location Pre-computed Cube`, $\mathcal{LPC}$ and an `Overflow Array`, $\mathcal{OA}$. The $\mathcal{LPC}$ keeps r number of maximum values for each sub-block which is sufficient for uniform data distribution. However, for non-uniform data distribution, more maximum values need to be kept for some of the sub-blocks. These additional maximum values are stored in the $\mathcal{OA}$ and are connected to the $\mathcal{LPC}$ by indices in the $\mathcal{LPC}$.

Definition 3.1 Given a data cube $\mathcal{DC}$ of d dimensions, and d partition factors $b_1, \ldots, b_d$, the location pre-computed cube, $\mathcal{LPC}$ of $\mathcal{DC}$, is a cube such that

1. it has the same dimension d,
2. if the size of the i^{th} dimension in $\mathcal{DC}$ is n_i, i.e., ranges from 0 to $n_i - 1$, then the dimension i in $\mathcal{LPC}$ will range from 0 to $\lceil n_i/b_i \rceil - 1$, and
3. each entry in $\mathcal{LPC}$, $\mathcal{LPC}[x_1, \ldots, x_d]$, is corresponding to a partitioned sub-block in $\mathcal{DC}$ and stores three types of elements:
   (a) the top-r maximum values, where r is the number of maximum values kept,
   (b) the locations of the cells holding the top-r maximum values, and
   (c) an index linked to the record in the $\mathcal{OA}$

We denote the top-r maximum values stored in an entry of $\mathcal{LPC}$, $\mathcal{LPC}[x_1, \ldots, x_d]$, as $\mathcal{LPC}[x_1, \ldots, x_d].Max[i]$, with the corresponding location as $\mathcal{LPC}[x_1, \ldots, x_d].Location[i]$ where

$1 \leq i \leq r$. The index is denoted as $\mathcal{LPC}[x_1, \ldots, x_d].Overflow$, which is set based on the first $\mathcal{OA}$ record for this entry.

An overflow array, $\mathcal{OA}$ is a one dimensional array structure, which is constructed by a set of records. We term each record in the $\mathcal{OA}$ as an `Overflow Record`. Each overflow record has the following structure:

1. f number of maximum values and the locations of cells holding these maximum values where f is called the `Overflow Factor`, and
2. an index specifies the next $\mathcal{OA}$ record if more values are needed, else NULL.

We denote the maximum values of each overflow record as $\mathcal{OA}[j].Max[i]$, the corresponding location as $\mathcal{OA}[j].Location[i]$, and the index of next $\mathcal{OA}$ record as $\mathcal{OA}[j].Next$ where $1 \leq i \leq f$ and j is the index of the records in $\mathcal{OA}$.

Initially, the $\mathcal{OA}$ contains no value and $\mathcal{LPC}[x_1, \ldots, x_d].Overflow$ for of all the entries are NULL. When processing query, if accessing to the original data cube is needed for a sub-block, additional maximum values are added into the $\mathcal{OA}$ for this sub-block. In other words, an overflow record keeps the maximum values which are not pre-stored in $\mathcal{LPC}$ or other overflow records of the sub-block. The construction of the $\mathcal{OA}$ is discussed in details in Section 3.2. Assuming the index of the first overflow record for $\mathcal{LPC}[x_1, \ldots, x_d]$ is j, thus, the $\mathcal{LPC}[x_1, \ldots, x_d].Overflow$ is set to j.

In this paper, we consider each value in the $\mathcal{LPC}$ as a cell, as in the data cube. The total number of sub-blocks in a data cube is $\prod_{i=1}^{d}(\lceil n_i/b_i \rceil)$. Assuming the size of a maximum value is 4 bytes and the location is formed by d indices each of size 2 bytes, a pre-stored maximum value requires $2d + 4$ bytes. We further assume each overflow index in the $\mathcal{LPC}$ is 2 bytes, thus an $\mathcal{LPC}$ requires $\prod_{i=1}^{d}(\lceil n_i/b_i \rceil) \times ([r \times (2d + 4)] + 2)$ bytes of storage.

Example 3.1 Figure 2 presents the $\mathcal{LPC}$ of the data cube $\mathcal{DC}$ shown in Figure 1. The number of maximum values kept for each sub-block, r, is set to 3. Note that $\mathcal{LPC}[2, 1]$ holds the top-3 maximum values and the corresponding locations among the cells in $\mathcal{DC}[i, j]$ where $8 \leq i \leq 11$ and $4 \leq j \leq 7$. The top-3 values are 97, 96 and 94 and the corresponding locations are $(9, 4)$, $(8, 5)$ and $(9, 6)$, respectively. Assuming that after processing some queries, the $\mathcal{OA}$ is constructed as in Figure 2 with f equals to 4. The indices of the entries in $\mathcal{LPC}$ with overflow records, are set. For instance, the overflow record for $\mathcal{LPC}[2, 1]$ with $\mathcal{LPC}[2, 1].Overflow = 1$, is

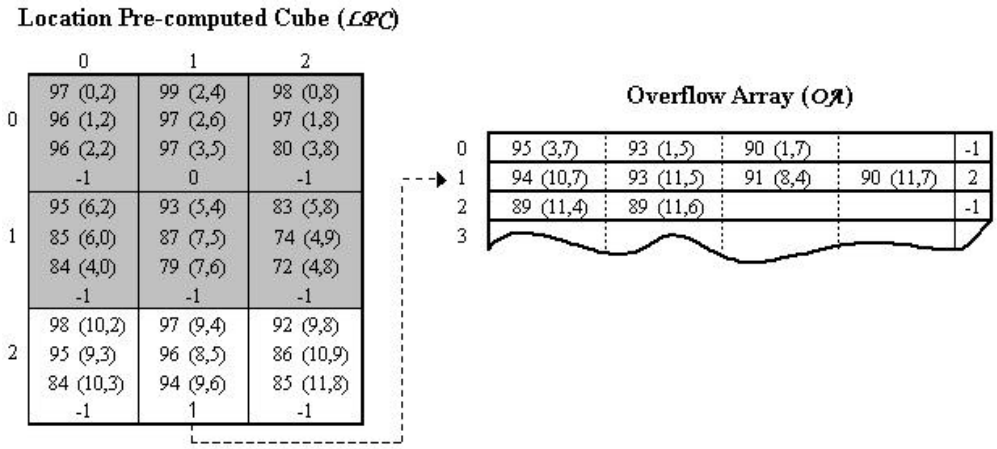


Figure 2: Auxiliary Data Structures of $\mathcal{APPT}$ method

$\mathcal{OA}[1]$. The maximum values kept in $\mathcal{OA}[1]$ are 94, 93, 91 and 90. The $\mathcal{OA}[1].Next$ is 2, which indicates that the next overflow record for $\mathcal{LPC}[2,1]$ is $\mathcal{OA}[2]$. The storage requirement for the $\mathcal{LPC}$ in Figure 2 is $(\lceil 11/4 \rceil \times \lceil 12/4 \rceil) \times ([3 \times (2 \times 2 + 4)] + 2) = 234$ bytes.

3.2 Queries

In this section, the query algorithm for computing the range top-k based on the $\mathcal{LPC}$ and $\mathcal{OA}$ is described. In addition, an example is given to better illustrate the idea. Before we present the algorithm, an overview is given as an introduction to the query algorithm.

3.2.1 Overview

Our query algorithm searches the $\mathcal{LPC}$, the $\mathcal{OA}$ and lastly, the original data cube if necessary. As mentioned earlier, the values in the $\mathcal{LPC}$ represent the maximum values of every sub-block. Thus, by searching the $\mathcal{LPC}$ before the rest, sub-blocks that contain smaller maximum values can be skipped. This also helps to decide whether the $\mathcal{OA}$ can be skipped as well. Our search sequence reduces the disk access significantly.

Three sorted lists, $\mathcal{SL}_1$, $\mathcal{SL}_2$ and $\mathcal{SL}_3$, are used in this algorithm. $\mathcal{SL}_1$ is used for searching the $\mathcal{LPC}$, $\mathcal{SL}_2$ is used for searching the $\mathcal{OA}$ and $\mathcal{SL}_3$ is used for searching the original data cube. All sorted lists sort the maximum values in descending order. $\mathcal{SL}_1$ is used to sort the largest value from each sub-block covered by the range query. The candidate answer to the range top-k query can be found easily since the first node in $\mathcal{SL}_1$ keeps the largest maximum value among all the sub-blocks. If none of the values for a sub-block in the $\mathcal{LPC}$ are in the query range, the

smallest value kept in the $\mathcal{LPC}$ for the sub-block is inserted into $\mathcal{SL}_2$ together with its location and index.

A node in $\mathcal{SL}_2$ represents the upper bound of the values left in a sub-block. If the value of a node in $\mathcal{SL}_2$ is larger than the smallest intermediate top-k result, the corresponding sub-block is further processed using the $\mathcal{OA}$. If there is no overflow record for the sub-block, the same value from $\mathcal{SL}_2$ is inserted into $\mathcal{SL}_3$. In addition, if the smallest value kept in the $\mathcal{OA}$ for a sub-block is larger than the smallest intermediate result, the value is inserted into $\mathcal{SL}_3$ too.

$\mathcal{SL}_3$ is used to indicate that some answers might still be left in a sub-block after processing both $\mathcal{LPC}$ and $\mathcal{OA}$. At this stage, the number of sub-blocks required processing is reduced. When a sub-block is scanned, all the values which are larger or equal to the smallest intermediate result are kept temporary in the main memory. When the final query result is found, the values that are larger or equal to the smallest value in the final result are updated to the $\mathcal{OA}$ on disk, while the result is output to the user.

3.2.2 The Query Algorithm

The sorted lists used in the query algorithm, $\mathcal{SL}_1$, $\mathcal{SL}_2$ and $\mathcal{SL}_3$, store nodes in the form of $< max, position, c >$ where c is the index of an entry in $\mathcal{LPC}$, max is one of the maximum values kept in $\mathcal{LPC}[c]$ or $\mathcal{OA}$ and $position$ is the location of max in $\mathcal{DC}$. Our algorithm assumes the $\mathcal{LPC}$ and $\mathcal{OA}$ are on disk. However, if the size of the main memory is large, the $\mathcal{LPC}$ and the $\mathcal{OA}$ can be loaded into the main memory to enhance the performance.

Throughout the algorithm, we denote the number of top values found as $\#found$ and the result of the top-k query as $Result[]$ where $Result[0]$ depicts the largest maximum value and $Result[k-1]$ is the smallest. The query algorithm is as follows:

Step 1. Initialization

Step 1.1 Entries in $\mathcal{LPC}$ which are covered by range query are loaded into main memory.

Step 1.2 Let $\mathcal{SL}_1$, $\mathcal{SL}_2$ and $\mathcal{SL}_3$ be three empty sorted lists.

Step 1.3 $\#found$ is initialized to 0.

Step 2. Construction of $\mathcal{SL}_1$ and $\mathcal{SL}_2$

For each sub-block covered by the range query,

Step 2.1 the largest value of the corresponding entry in $\mathcal{LPC}$, which is inside the query range is inserted into $\mathcal{SL}_1$ together with its location and index in $\mathcal{LPC}$.

Step 2.2 **if** none of the value of the entry in $\mathcal{LPC}$ is inside the range query, the smallest maximum value from the sub-block in $\mathcal{LPC}$ is inserted into $\mathcal{SL}_2$, together with its location and index in $\mathcal{LPC}$.

Step 3. Processing of $\mathcal{SL}_1$ (Search in $\mathcal{LPC}$)

While ($\mathcal{SL}_1$ is not empty) **and** ($\#found < k$)

Step 3.1 max from the first node in $\mathcal{SL}_1$, $< max, position, c >$, is taken as the $Result[\#found]$ and $\#found$ counter is increased by 1.

Step 3.2(a) **If** the next maximum value in range can be found in $\mathcal{LPC}[c]$ in memory,

While ($\#found < k$) **and** (next maximum value in range can be found from $\mathcal{LPC}[c]$)

Step 3.2.1(a) **If** it is larger than the value of the next node in $\mathcal{SL}_1$, it is taken as $Result[\#found]$. $\#found$ is increased by 1.

Step 3.2.1(b) **Else** this value is inserted into $\mathcal{SL}_2$. Go to Step 3.3

Step 3.2(b) **Else** the smallest value of $\mathcal{LPC}[c]$, together with its location and index, are inserted into $\mathcal{SL}_2$.

Step 3.3 The first node of $\mathcal{SL}_1$ is deleted.

Step 4. Processing of $\mathcal{SL}_2$ (Search in $\mathcal{OA}$)

While $\mathcal{SL}_2$ is not empty

Step 4.1(a) **If** (#found $< k$) **or** (max in the first node of $\mathcal{SL}_2 > Result[k-1]$)

Step 4.1.1(a) **If** $\mathcal{LPC}[c].Overflow$ is NULL, the first node of $\mathcal{SL}_2$ is inserted into $\mathcal{SL}_3$.

Step 4.1.1(b) **Else**

Do

Step 4.1.1.1 Overflow record is loaded from disk.

Step 4.1.1.2 The loaded values, which are in the range query and are larger than $Result[k-1]$, are updated to $Result[]$ and $\#found$ is increased.

While ((the smallest value in overflow record $>$ $Result[k-1]$) **or** ($\#found < k$)) **and** (the index linked to the next overflow record is not NULL)

Step 4.1(b) **Else** go to Step 5.

Step 4.2 The first node of $\mathcal{SL}_2$ is deleted.

Step 5. Processing of $\mathcal{SL}_3$ (Search in data cube)

While $\mathcal{SL}_3$ is not empty

Step 5.1 **If** ($\#found < k$) **or** (max in the first node of $\mathcal{SL}_3 > Result[k-1]$)

Step 5.1.1 The sub-block region which corresponds to c is scanned.

Step 5.1.2 The scanned values, which are in the range query and are larger than $Result[k-1]$, are updated to $Result[]$ and $\#found$ is increased.

Step 5.1.3 The scanned values which are not pre-stored in $\mathcal{LPC}$ and $\mathcal{OA}$ but are larger or equal to the $Result[k-1]$, are kept temporary in the main memory.

Step 5.2 The first node of $\mathcal{SL}_3$ is deleted.

Step 6. Output the result

Step 6.1 $Result[]$ is returned as the query result of the range top-k query.

Step 6.2 **If** ($\#found < k$), a message is given to indicate that the number of cells covered by the range query is less than k.

Step 6.3 The values which are kept temporary in the main memory, and are larger or equal to the final $Result[k-1]$, are updated to the $\mathcal{OA}$ on disk.

3.2.3 An Example

We now present a concrete example using the query algorithm of the $\mathcal{APPT}$ method. With range query (3:7, 3:10) in Figure 1, a top-3 range query, i.e. $k = 3$, is performed on the data cube. Using the $\mathcal{LPC}$ and $\mathcal{OA}$ in Figure 2, the following steps are executed.

Entries of $\mathcal{LPC}$ which are covered by the range query are loaded into the main memory. The largest maximum values in $\mathcal{LPC}[0,1]$, $\mathcal{LPC}[1,1]$, $\mathcal{LPC}[1,2]$ and $\mathcal{LPC}[0,2]$ which are in the query range, are inserted into $\mathcal{SL}_1$. For $\mathcal{LPC}[0,0]$ and $\mathcal{LPC}[1,0]$, none of the pre-stored values

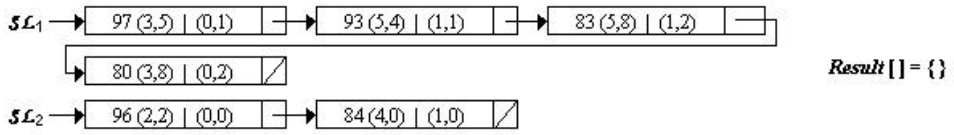


Figure 3: Construction of $\mathcal{SL}_1$ and $\mathcal{SL}_2$

in the $\mathcal{LPC}$ are in the query range, thus, their smallest values in $\mathcal{LPC}$ are inserted into $\mathcal{SL}_2$. The constructed $\mathcal{SL}_1$ and $\mathcal{SL}_2$ are as shown in Figure 3 where the maximum values are sorted in descending order.

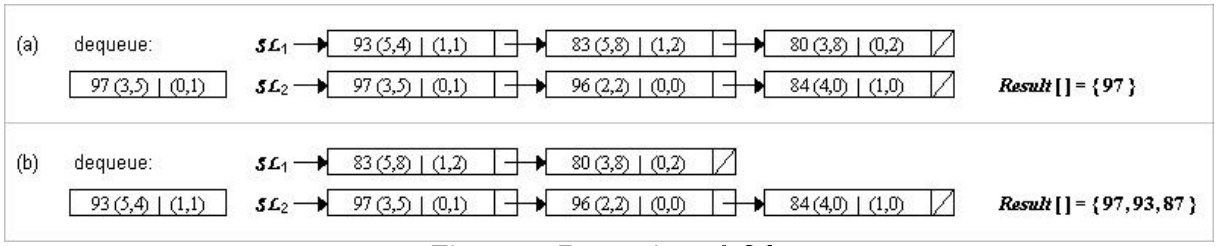


Figure 4: Processing of $\mathcal{SL}_1$

The maximum value in the first node of $\mathcal{SL}_1$, 97 from $\mathcal{LPC}[0,1]$, is set as $Result[0]$. Since 97 is the smallest pre-stored value in $\mathcal{LPC}[0,1]$, thus 97, is inserted into $\mathcal{SL}_2$ and $\#found$ is set to 1. The first node in $\mathcal{SL}_1$ is deleted. Figure 4(a) depicts the current state of $\mathcal{SL}_1$ and $\mathcal{SL}_2$. The current maximum value in the first node of $\mathcal{SL}_1$ is 93 from $\mathcal{LPC}[1,1]$. 93 is set as $Result[1]$ and $\#found$ is increased to 2. The next maximum value from $\mathcal{LPC}[1,1]$ which is in the query range is 87. Since 87 is larger than the maximum value in the next node of $\mathcal{SL}_1$, 83, 87 is set directly as $Result[2]$ and $\#found$ is increased to 3. The first node of $\mathcal{SL}_1$ is deleted and the current state of $\mathcal{SL}_1$ is as shown in Figure 4(b). The processing of $\mathcal{SL}_1$ is stopped as $\#found$ is equal to k, which is 3.

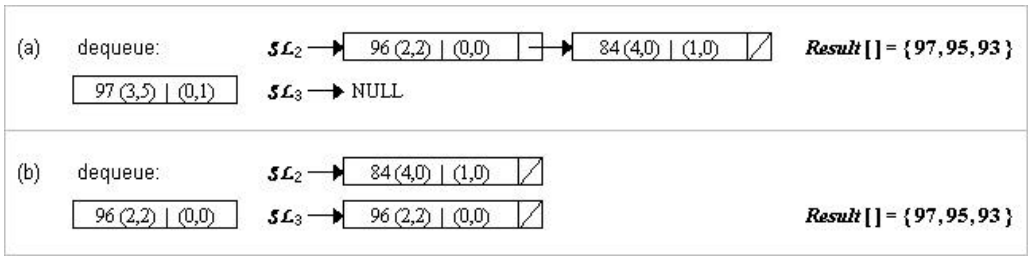


Figure 5: Processing of $\mathcal{SL}_2$

According to Figure 4(b), the first node in $\mathcal{SL}_2$ is 97 from $\mathcal{LPC}[0,1]$, which is larger than $Result[2]$, i.e., 87. The $\mathcal{OA}$ record linked by $\mathcal{LPC}[0,1].Overflow$, $\mathcal{OA}[0]$, is scanned from disk. The only value in $\mathcal{OA}[0]$ which is in the query range and is larger than $Result[2]$, is 95, thus, 95 is updated to $Result[]$ and the first node in $\mathcal{SL}_2$ is deleted as shown in Figure 5(a). The current maximum value in the first node of $\mathcal{SL}_2$ is 96 from $\mathcal{LPC}[0,0]$, which is larger than $Result[2]$.

However, $\mathcal{LPC}[0,0].Overflow$ is equal to NULL. Thus, 96 is inserted into $\mathcal{SL}_3$ and the first node in $\mathcal{SL}_2$ is deleted as shown in Figure 5(b). The next value from $\mathcal{SL}_2$ is 84, which is smaller than $Result[2]$ and the process of $\mathcal{SL}_2$ is terminated.

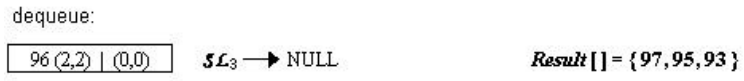


Figure 6: Processing of $\mathcal{SL}_3$

As can be seen from Figure 5(b), the value in the first node of $\mathcal{SL}_3$, 96, is larger than $Result[2]$, hence, the whole block (0:3, 0:3) is scanned. The values in sub-block (0:3, 0:3) which are not pre-stored in $\mathcal{LPC}$, are $\{94, 91, 70, \ldots, 8\}$ in descending order. The $Result[]$ is not updated as 94 is not in the query range. However, 94 is kept temporary in the main memory. The first node in $\mathcal{SL}_3$ is deleted. $\mathcal{SL}_3$ is empty and the algorithm is stopped. $Result[]$={97, 95, 93} is returned as the answer. The value 94 is updated to the $\mathcal{OA}[3]$ and $\mathcal{LPC}[0,0].Overflow$ is set to 3.

3.3 Updates

Updating a cell in the data cube may require updates to the $\mathcal{LPC}$ and $\mathcal{OA}$. Update in the $\mathcal{APPT}$ method can be done in batch.

By using batch update, the input list is scanned once. All the update points are grouped according to the sub-blocks they belong to and are sorted in descending order based on the new values. The batch update algorithm will update the corresponding value in the original data cube, when necessary, update the entries in $\mathcal{LPC}$ and/or $\mathcal{OA}$ which have been loaded into the main memory and finally, update the $\mathcal{LPC}$ and/or $\mathcal{OA}$ reside on disk with the updated values in main memory. By grouping and sorting all the update points, those update-points which are smaller than the smallest value in the entry in $\mathcal{LPC}$ or $\mathcal{OA}$, can be filtered. Hence, batch update helps in reducing the number of updates that have to be done on $\mathcal{LPC}$ and $\mathcal{OA}$. We denote the $\mathcal{LPC}$ and $\mathcal{OA}$ as pre-stored structures.

Definition 4.1 Given an update $< position, new, old >$, where *position* denotes the index of the update point in $\mathcal{DC}$, *new* is the new value and *old* is the old value, it is an `increase-update` if $new > old$, and a `decrease-update` otherwise. An increase-update is `active` if *new* is greater than the smallest value in the pre-stored structures of the sub-block, otherwise, it is a `passive` update. A decrease-update is `active` if *position* is one of the indices in the pre-stored

structures of the sub-block, and `passive` otherwise.

For an active increase update, the update is performed and the values in the pre-stored structures are stored in the appropriate position. For active decrease update, the value is deleted from the pre-stored structures if it is less than the smallest value in the entry of the $\mathcal{LPC}$, otherwise, the entries for the $\mathcal{LPC}$ are stored in the appropriate position. A threshold is set to the number of values left in the entries of $\mathcal{LPC}$. If the number of values left is below the threshold, searching to the original data cube might be needed but can be delayed. Nevertheless, such a search to the original data cube is given a lower priority. It can be withheld until either a search to the data cube is needed for the sub-block when processing queries or when the system is idle. Passive updates (increase or decrease) will not change the values in $\mathcal{LPC}$ and $\mathcal{OA}$ of the sub-block.

Assuming the $\mathcal{LPC}$ and $\mathcal{OA}$ are stored on disk, our update algorithm promises an average update cost of $(1 + t)$ total disk accesses where t is the average number of IO blocks in the overflow array for each sub-block. The values kept in an entry of $\mathcal{LPC}$ is small enough that only one disk access is required, and t disk accesses for searching in $\mathcal{OA}$ of the sub-block. However, the value of t should be less than one as the number of sub-blocks that have an overflow record is less than the total number of sub-blocks in the data cube. Assume that the total number of sub-blocks that have an overflow is 50% of total sub-block in the data cube, the update cost incurs is 1.5 total disk accesses, which is very low. Furthermore, for batch update, only $(1 + t)$ total disk accesses are required for all the update points from the same sub-block if $\mathcal{LPC}$ and $\mathcal{OA}$ are on disk.

3.4 Update Algorithm

In this update algorithm, we use pre-stored structures to denote entry in $\mathcal{LPC}$ and $\mathcal{OA}$ of a sub-block. For each update point $< position, new, old >$ scanned in, the update algorithm processes according to the following rules:

Case (1) **If** $(new > old)$ (an increase update)

Case (1.1) **If** *position* is one of the indices in the pre-stored structures (active update), *old* is updated by *new* and the values in the pre-stored structures are stored in the appropriate position.

Case (1.2) **Else**

Case (1.2.1) **If** $new >$ the smallest value in the pre-stored structures, new is inserted. New record is allocated for if the last $\mathcal{OA}$ record is fully utilized before the insertion or the sub-block does not have an overflow record before update.

Case (1.2.2) **If** $new \leq$ the smallest value in the pre-stored structures (passive update), the update does not change the values in $\mathcal{LPC}$ and $\mathcal{OA}$.

Case (2) **If** $(new < old)$ (a decrease update)

Case (2.1) **If** *position* is one of the indices in the pre-stored structures (either $\mathcal{LPC}$ or $\mathcal{OA}$) (active update),

Case (2.1.1) **If** $new >$ the smallest value in the pre-stored structures, old is replaced by new and the values in the pre-stored structures are re-sorted.

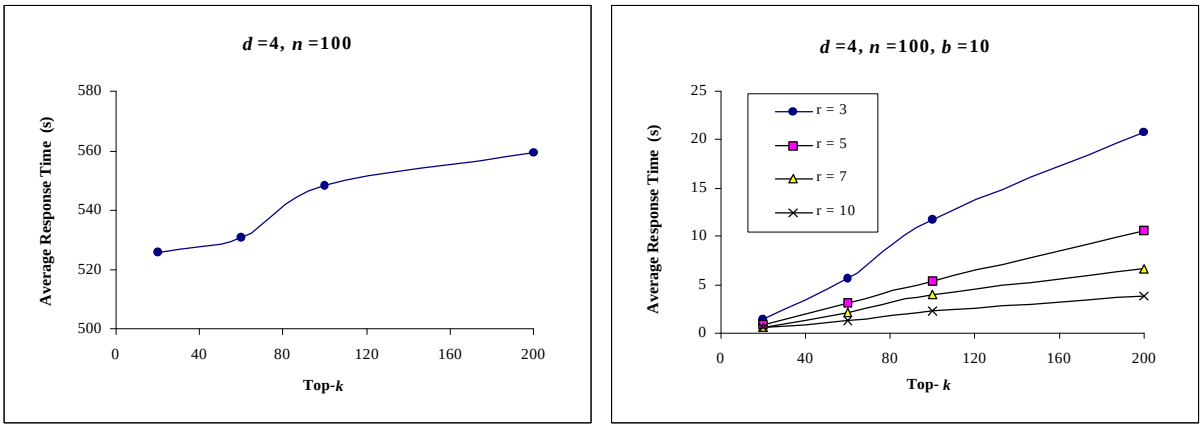
Case (2.1.2) **If** $new <$ the smallest value in the pre-stored structures, old is deleted from the pre-stored structure and the values in the pre-stored structures are re-sorted.

Case (2.2) **Else** no changes is made to $\mathcal{LPC}$ and $\mathcal{OA}$.

4 Experiments and Performance Analysis

The performance of the $\mathcal{APPT}$ method is analyzed using experiment results. A set of non-uniform data cube is randomly generated by intentionally introducing some large numbers in some of the sub-blocks. Without loss of generality, we consider data cubes with equal size for each dimension, construct $\mathcal{LPC}$ and $\mathcal{OA}$ of the data cube, and load them into the main memory. A total of 1000 queries are generated randomly using time as the seed and the average response time required is measured. The experiments are run under Windows 98 on Pentium III with 667MHz CPU and 128MB RAM. Let k denote the number of maximum values being queried, r is the number of maximum values kept in $\mathcal{LPC}$ over partition, f is the size of each record in the $\mathcal{OA}$ while d, n and b is the dimensionality, size of each dimension and partition factor, respectively. Throughout the experiments, we set $d = 4$. Although this is considered relatively small for practical application, it is sufficient to show the efficiency of our method compared with the naïve method.

To answer a range top-k query, the naïve method is to scan all the involved cells in the data cube. However, the cost of access is proportional to the number of involved cells and the number of cells in a data cube is exponentially proportional to the number of dimension. In this section, we firstly compare the average response time for various top-k queries between the naïve method and the $\mathcal{APPT}$ method.



(a) Naïve Method (b) *APPT* Method using only *LPC*

Figure 7: Response Time of Naïve Method (a) and $\mathcal{APPT}$ Method (b)

Due to time limitation, only 40 queries are executed on each top-k for the naïve method. The average number of cells covered by the queries is 3,750,000, ranging from 372 to 13,296,960. The same range queries are performed for the $\mathcal{APPT}$ method using merely the $\mathcal{LPC}$. Figure 7 depicts the processing time required for the $\mathcal{APPT}$ method and the naïve method. Obviously, the $\mathcal{APPT}$ method outperforms the naïve method. For top-200, the naïve method requires an average response time of more than 500 seconds (about 9 minutes), which is intolerable for a decison maker. However, the $\mathcal{APPT}$ method needs only less than 25 seconds using $r = 3$, and less than 5 seconds for $r = 10$, with an extra storage for $\mathcal{LPC}$ 0.36MB and 1.16MB, respectively. This shows an improvement of over 20 times for $r = 3$ and 100 times for $r = 10$ using the $\mathcal{APPT}$ method over the naïve method. Furthermore, the extra storage needed for the $\mathcal{APPT}$ method is considered small if compared with the original data cube, which is of the size 380MB, with n=100 and d=4.

The average response time for the $\mathcal{APPT}$ method using both $\mathcal{LPC}$ and $\mathcal{OA}$ is shown in Figure 8. The average number of cells covered by the range queries is 20,013,281 while the average number of sub-blocks covered by the queries is 3115, out of 10000 number of sub-blocks in the data cube. By comparing the performance of $r = 3$, $r = 5$ and $r = 7$, it is noticed that a larger r outperforms a smaller r. The reason is that by using a larger r, the number of pre-stored

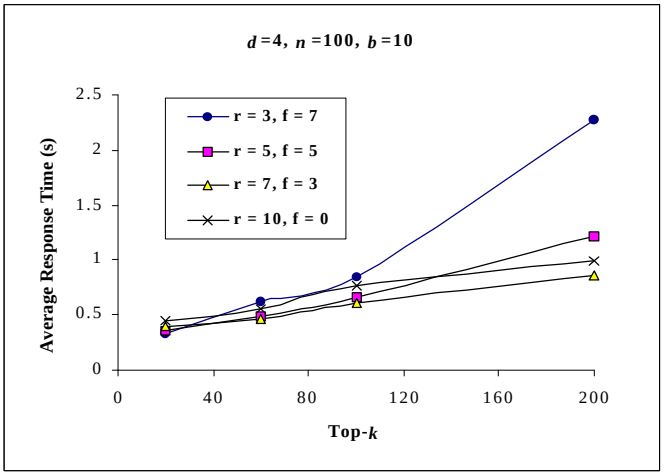


Figure 8: Average Response Time for $\mathcal{APPT}$ Method

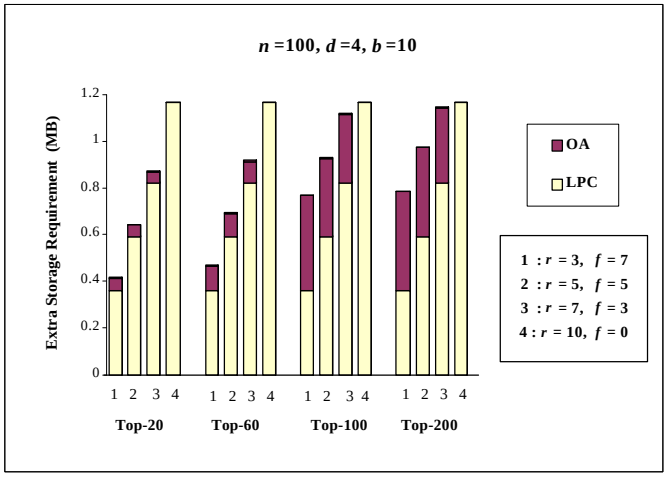


Figure 9: Extra Storage Requirement for $\mathcal{APPT}$ Method

values that can be used to answer a query is more. Hence, the number of accesses to the data cube is decreased. Although larger r gains in response time, the storage requirement is larger than the others as can be noticed from Figure 9. Nevertheless, with a higher storage requirement, the average response time for $r = 10$, $f = 0$ (without $\mathcal{OA}$) is higher than $r = 7$, $f = 3$. Thus, it is shown that the performance for the $\mathcal{APPT}$ method on non-uniformly distributed data cube is efficient with the existance of $\mathcal{OA}$. The principle behind the $\mathcal{APPT}$ method is to allocate more resources to the sub-blocks that require more maximum values. Therefore, the $\mathcal{APPT}$ method can utilize resources adaptively.

Figure 10 depicts that the average response time for two separate experiments running on the same set of range queries. The first experiment performs the queries using $\mathcal{LPC}$ only. The

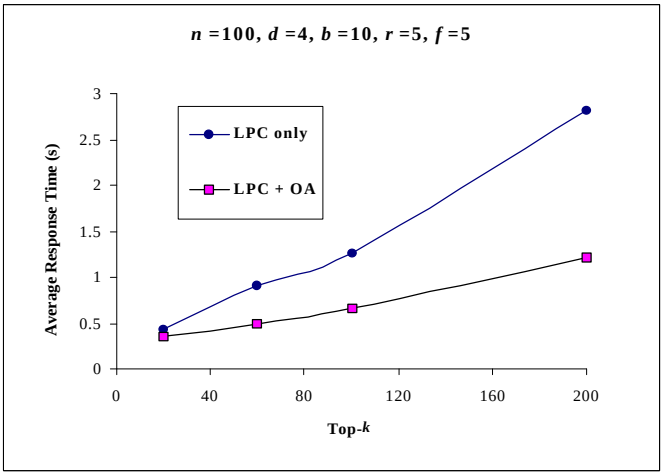


Figure 10: Average Response Time Improvement for Same Range Queries

$\mathcal{OA}$ is constructed from this experiment for the usage of the second experiment. The difference in terms of average response time shows that the performance of the $\mathcal{APPT}$ method is further improved when the range queries are similar over time.

5 Discussion

In real world applications, there exist data cubes which are sparse. This paper describes the $\mathcal{APPT}$ method for dense data cube. However, when the method is applied on a sparse data cube, a marker can be added to the sub-block to indicate that there is no other value in that sub-block, therefore, no searching needs to be done on it.

Our query algorithm takes the assumption that the $\mathcal{LPC}$ and the $\mathcal{OA}$ covered by range query are loaded from disk when answering queries. If the range query is too large that the covered entries in $\mathcal{LPC}$ cannot be fully loaded into the main memory, then the range can be divided into a few portions, and these portions are processed accordingly. However, if the $\mathcal{LPC}$ and/or the $\mathcal{OA}$ are small enough to be loaded into the main memory, the performance of query can be improved.

The query pattern and the data distribution vary for different OLAP data cubes. Therefore, it is difficult to select an optimum value for r and f when a new data cube is created. However, the $\mathcal{APPT}$ method can be made to adapt smartly from time to time. The database system of an organization is being loaded and reloaded after a period of time. Thus, the value of r can be adjusted to suit the application needs. The value of r is increased if overflow records are

needed for majority of the sub-blocks, and is decreased otherwise. The median of the number of maximum values kept in $\mathcal{OA}$ for each sub-block is used to increase r. The reason for choosing median, instead of mean, is to minimize the effect of possible dominant number of values kept in the $\mathcal{OA}$ for minority of the sub-blocks. By increasing the number of values kept in the $\mathcal{LPC}$ and decreasing the number of values kept in the $\mathcal{OA}$, the total IO required is reduced.

It is possible to reduce the total IO needed as well as the average response time when searching the original data cube. The physical storage design of the original data cube plays an important role. Normally, the cells of a data cube after partitioning is arranged in the way that the cells from the same sub-block are distributed into b^{d-1} number of rows where b denotes the partition factor and d is the dimensionality. This also means that when a sub-block is read, at least b^{d-1} number of IO is needed if each IO is of the size n/b. By storing all the cells from one sub-block consecutively, the IO needed for scanning the sub-block is reduced. This physical storage re-organization helps to reduce the response time in our method.

6 Conclusion and Future Work

Efficient calculation of range queries has become more important in recent years. Several pre-computation techniques have been developed to answer range-max queries efficiently, but these methods cannot be applied to the range top-k queries directly. In this paper, we have presented a new algorithm, the `Adaptive Pre-computed Partition Top Method` ($\mathcal{APPT}$) for computing range top-k queries on both uniform and non-uniform data distribution OLAP data cubes. The main idea for speeding up range top-k queries is to pre-compute r maximum values for each sub-block and store additional maximum values for sub-block requires more than r-maximum values when answering some queries. Hence, subsequent queries with the similar ranges will not need to search back to data cube. Using the pre-stored data structures, our method reduces the response time drastically as can be seen from the experiment results presented. Furthermore, the search algorithm processes the range top-k query in the order that reduce the chances of scanning to the original data cube. The performance of the $\mathcal{APPT}$ method can be tuned easily. The value of r and overflow factor, f, can be adjusted to the query pattern to gain the maximum performance out of the storage requirement. The $\mathcal{APPT}$ method incurs very low average update cost of $(1 + t)$ total disk accesses, where t is the average number of records in the $\mathcal{OA}$ for each sub-block and the value of t should be less than 0.5.

The $\mathcal{APPT}$ method performs efficiently when the query pattern of an application is similar

over time. Some extra processing may be needed to handle skew queries and query pattern that changes drastically. The performance of the extension to the $\mathcal{APPT}$ is currently being investigated by the authors.

References

[AMS97] C. Ho, R. Agrawal, N. Mefiddo and R. Srikant. Range queries in OLAP data cubes. In *Proc. of the ACM SIGMOD Conference on Management of Data*, Tucson, Arizona, May 1997.

[CG99] S. Chaudhuri, L. Gravano. Evaluating Top-k Selection Queries. In *Proc. of the 25th Int'l Conference on Very Large Databases*, 1999.

[CI99] C. Y. Chan, Y. E. Ioannidis. Hierarchical Cubes for Range-Sum Queries. In *Proc. of the 25th Int'l Conference on Very Large Databases*, pages 675-686, 1999.

[DR99] D. Donjerkovic, R. Ramakrishnan. Probabilistic optimization of top N queries. In *Proc. of the 25th Int'l Conference on Very Large Databases*, 1999.

[GAAS99] S. Geffner, D. Agrawal, A.E. Abbadi, T. Smith. Relative Prefix Sum: An Efficient Approach for Querying Dynamic OLAP Data Cubes. In *Proc. of the 15th Int'l Conference on Data Engineering*, pages 328-335, 1999.

[GBLP96] J. Gray, A. Bosworth, A. Layman and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, coress-tabs and sub-totals. In *Proc. of the 12th Int'l Conference on Data Engineering*, pages 152-159, 1996.

[KLKL98] D. W. Kim, E.J. Lee, M. H, Kim, Y, J. Lee. An efficient processing of range-MIN/MAX queries over data cube. *Information Sciences*, pages 223-237, 1998.

[LLL00a] S. Y. Lee, T. W. Ling and H. G. Li. Hierarchical compact cube for range-max queries. In *Proc. of the 26th Int'l Conference on Very Large Databases*, 2000.

[LLL00b] H. G. Li, T. W. Ling and S. Y. Lee. Range-max queries in OLAP data cube. In *Proc. of the 11th Int'l Conference on Database and Expert Systems Applications*, pages 467-475, 2000.