# A Greedy Approach to Query Processing for Different Query Distributions in Data Warehouses

Ye-In Chang, Hue-Ling Chen, and Chien-Show Lin
*Dept. of Computer Science and Engineering*
*National Sun Yat-Sen University*
*Kaohsiung, Taiwan, R.O.C*
*E-mail: changyi@cse.nsysu.edu.tw*

## Abstract

*The Range-Based Index (RBI) reduces the response time for on-line decision support in the data warehouses, especially for attributes with high cardinality in data records. However, partitioning the entire ranges into the number of bitmap vectors is a very critical issue for the RBI, since the task of fetching data from the disk for the attribute checking is very time-consuming. Moreover, from the history of users' queries in the data warehouses, queries are frequently performed on data records with the same value or ranges of the attribute. It takes long disk I/O time when these data records are partitioned into different bitmap vectors. In this paper, we consider the history of users' queries on the design of the partitioning strategy. Based on the greedy approach, we propose the GreedyExt and GreedyRange strategies for answering exact queries and range queries, respectively. These two strategies decide the set of queries to construct the bitmap vectors such that data records with the high frequency of the query can be quickly and directly accessed. Then, the response time can be reduced in most of situations.*

**Keywords**: bitmap vector, data warehouse, disk access, range-based bitmap index, response time.

## 1. Introduction

In the *range-based index (RBI)*, a bitmap vector is used to represent a range, instead of a distinct value. The operations based on RBI for processing complex adhoc queries are performed quickly for data records with high cardinality in data warehouses, like those of decision support systems in many scientific and business domains [2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14]. For example, to answer a range query such as "$4 \leq A \leq 6$", we only need one bitmap vector $RB$ to access the attributes that may include 4.6, 5.9, 5.96, and so on. However, for the range query such as "$3 \leq A \leq 6$", the range of the bitmap vector $RB$ does not fully cover the range query. It is very time-consuming to fetch all data from the related bitmap vectors in the disk for checking the attribute value against the query condition.

Since data records in the data warehouse are often non-uniformly distributed, it is a very critical issue to partition the entire ranges into the number of bitmap vectors for the range-based bitmap index [1, 5, 6, 7, 14]. As compared with the other previous strategies, *the Dynamic Bucket Expansion and Contraction strategy* (DBEC) [15] could reduce disk I/O time to answer range queries, when the distribution of data is not uniform. The key point of the DBEC strategy is to partition the ranges with non-equal width but almost the same number of data records. For example, considering the data 14, 9, 10, 19, 21, 15, 17, 16, 25, 12, 15, 10, the partitions of the DBEC strategy are {9, 10}, {10, 14, 12}, {15, 16, 15}, {19, 21, 17} and {25}, and the number of data records in each partition are 2, 3, 3, 3, 1, respectively. However, the DBEC strategy could not guarantee to get the partition result with the given number of bitmap vectors. For the above example, it constructs 5 bitmap vectors instead of only 4 bitmap vectors due to the limit of storage space. Moreover, when the users' queries for different data records with the same value, these data records may be partitioned into different bitmap vector based on the DBEC strategy, which takes long disk I/O time. For the above example, data records with the same value 10, are partitioned into 2 bitmap vectors. If we want to retrieve value 10, we need to check two bitmap vectors.

In this paper, based on the greedy approach, we propose the GreedyExt and GreedyRange strategies for answering exact queries and range queries, respectively. The two strategies consider the history of the users' queries and construct the given number of the bitmap vectors for the data records which have high frequency of the query and the attribute value against the query condition. The data records with the same value or ranges of the attribute will be assigned into

Table 1: A stock trading example

| Record ID | Ticker Symbol | Trading Volume | Closing Price | Exchange |
|---|---|---|---|---|
| 1 | AAPL | 4,575,000 | 36.625 | NASDAQ |
| 2 | ABF | 64,200 | 24.500 | NYSE |
| 3 | AET | 369,000 | 72.625 | NYSE |
| 4 | CPQ | 8,968,800 | 51.375 | NYSE |
| 5 | DEC | 4,461,100 | 49.750 | NYSE |
| 6 | DELL | 2,714,400 | 89.750 | NASDAQ |
| 7 | HWP | 3,009,300 | 90.250 | NYSE |
| 8 | IBM | 7,657,700 | 92.500 | NYSE |
| 9 | IFMX | 3,493,600 | 33.000 | NYSE |
| 10 | INTC | 17,694,400 | 65.500 | NASDAQ |
| 11 | LGNT | 2,600 | 47.250 | NASDAQ |
| 12 | MSFT | 18,288,600 | 91.125 | NASDAQ |

Table 2: Two range-based bitmap vectors for the attribute 'Trading Volume'

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $BM_1$: | ( 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 ) |
| $BM_2$: | ( 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | $\underline{1}$ | 0 | $\underline{1}$ ) |

the same partition. Then, the average response time of answering queries could be reduced. From our performance analysis, we prove that our GreedyExt and GreedyRange strategies could reduce the number of disk accesses a lot.

The rest of the paper is organized as follows. In Section 2, we briefly describe the range-based bitmap index for data warehouses. In Section 3, we present our GreedyExt and GreedyRange strategies which take the history of users' queries into consideration. In Section 4, we make the analysis on our GreedyExt and GreedyRange strategies. In Section 5, we give the conclusion.

## 2. Range-Based Bitmap Index

The main idea of range-based index is the reduction of storage overhead by means of partitioning [15], *i.e.*, attribute values are split into small number of ranges that are represented by bitmap vectors. What is more, a bit is set to 1 if a record falls into specified range; otherwise, this bit is set to 0. The bitmap vector is used to represent a range rather than a distinct value for the attribute. Take Table 1 as an example. We assume that a maximum trading volume of 20,000,000 shares per day, which is quite a reasonable assumption for a stock exchange. We divide the values of attribute 'Trading Volume' into two equally sized ranges: $[0; 10,000,000)$ and $[10,000,000; 20,000,000)$. Two bitmap vectors, $BM_1$ and $BM_2$, are built for these two ranges, respectively, as shown in Table 2. Since the 10th and the 12th bits of

Table 3: The process of the AND operation on two bitmap vectors

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $BM_1$: | ( 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 ) |
| $BM_3$: | ( 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 ) |
| Result: | ( 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 ) |

the bitmap vector $BM_2$ are set to 1 (which are underlined), the trading volume of the 10th and 12th stocks, respectively, are both greater than 10,000,000 stocks per day.

Although the small number of bitmap vectors are needed to be stored based on RBI, the resulting query process might be longer For eample, there is the query like this: "select all stocks at NYSE that have a trading volume of ranges between 4,000,000 and 10,000,000 shares." In order to retrieve the data to answer the query, the AND operation is performed on two bitmap vectors, $BM_1$ and $BM_3$, which are built for the range $[0; 10,000,000)$ and the attribute 'Exchange=NYSE', respectively. Table 3 show the process of the AND operation. The result of 7 candidates, which are represented by the bit 1, need to be checked against the value between 4,000,000 and 10,000,000. With RBI, one of the great difficulties is to find an optimal partitioning of the range in order to decrease the number of candidates checking which may be time-consuming for query processing. Moreover, users may frequently query for data records with the same value or range of the attribute, *e.g.*, "select the stock which has the closing price at 91.125 or ranges between 90 to 100". These queries is frequently happened in the data warehouse. The data records with the value 91.125 or ranges between 90 and 100 have the high frequency of the query. The response time can be reduced based on an optimal partitioning of the range on the attribute 'Closing Price'.

*The Dynamic Bucket Expansion and Contraction strategy* (DBEC) [15] constructs bitmap vectors for range-based bitmap index for high cardinality attributes with skew. It consists of two phases. In the first phase, data records are sequentially scanned and multiple equally-spaced vectors are used to count the number of records falling into each vectors. Whenever a vector has accumulated more than a certain number of records, it is dynamically expanded into more smaller-range vectors. Upon completion of counting, multiple small-range vectors are then contracted into the final vectors such that each vector contains about the same number of records. In the second phase, data records are then sequentially scanned again to actually build the bitmap indexes, with a bitmap vector rep-

resenting the range of each final vector. However, the data records with the same value or ranges of the attribute may be partitioned into different bitmap vectors by the DBEC strategy such that each vector contains about the same number of records. Furthermore, the DBEC could use more bitmap vectors than the given one which is the limit of the storage. Consequently, it costs the large number of disk accesses to retrieve these similar data records in different bitmap vectors.

## 3. Strategies for the History of Queries

From the history of users' queries, we may find that most of users' queries frequently focus on the same value or range of data records. Therefore, we present two greedy strategies, GreedyExt and GreedyRange, to construct the bitmap vectors by considering the history of users' queries. The variables used in the GreedyExt and GreedyRange strategies are shown in Table 4.

### 3.1. The GreedyExt Strategy for Exact Queries

The basic idea of the GreedyExt strategy is to decide the ranges according to the frequencies of users' exact queries on some distinct values of the attribute in the history. Take Table 1 as an example. The frequency ($= freq_i$) of users' queries on the attribute 'Closing Price' ($= Val_i$) are shown in Table 5. The frequency, $freq_i$, is calculated by $cq_i/tol$, and is recorded in $freq\_table$, where $cq_i$ is the count of the exact query for data record with the value $Val_i$ of the attribute and $tol = \sum_{i=1}^{n} cq_i$, $1 \leq i \leq n$. If the storage space of the bitmap index is limited to be $M$ bits, we can use the $PN$ bitmap vectors to construct the bitmap index, where $PN = \lfloor M/n \rfloor$. In the following example, we assumed that the frequencies of the values of the attribute have already been calculated. We also assume that there are 1000 data records and 5000 bits as the offered storage space. Therefore, we construct the bitmap index by using $5 = (\lfloor 5000/1000 \rfloor)$ bitmap vectors.

The GreedyExt procedure is shown in Figure 1. Take Table 5 as an example. First, we sort $freq\_table$ according to the frequency $freq_i$ of each query on $Val_i$, in a descending order, and record the result in $f\_table$, as shown in Table 6. Since $PN = 5$, we have $NumHFQ = \lfloor (5-1)/2 \rfloor = 2$. That is, we will find the two values, 91.125 and 65.500, which have the highest two frequencies of the query in $f\_table$, respectively. Then, we sort the values, 91.125 and 65.500, in an ascending order, and record them in $HFQ$. Table 7 illustrates the GreedyExt strategy step by step. In Step 1, we obtain $HFQ[1] = 65.500$ and $HFQ[2] = 91.125$.

Next, in Step 2 of Table 7, we have $R_2 = HFQ[1] = 65.500$ and $R_4 = HFQ[2] = 91.125$. Finally, in Step

Table 4: Variables in the GreedyExt and GreedyRange strategies

| Variable | Description |
|---|---|
| $n$ | The number of records |
| $PN$ | The number of the bitmap vectors |
| $freq\_table$ | The table contains $Val_i$ and the corresponding $freq_i$ |
| $Val_i$ | The $i$th value of $freq\_table$ |
| $freq_i$ | The frequency of the query for $Val_i$ |
| $f\_table$ | The table contains $V_i$ and the corresponding $f_i$ |
| $V_i$ | The $i$th value of $f\_table$ |
| $f_i$ | The frequency of the query for $V_i$ |
| $HFQ$ | The array recording queries |
| $NumHFQ$ | The length of the array $HFQ$ |
| $r_i$ | The range $i$ of the query |
| $R_i$ | The partitioned range of the $i$th bitmap vector |
| $BM_i$ | The bitmap vector of the partitioned range $R_i$ |
| $fixR$ | The length of the initial range |

Table 5: Table $freq\_table$: an example of the GreedyExt strategy

| $Val_i$ | $freq_i$ |
|---|---|
| 36.625 | 0.06 |
| 24.500 | 0.01 |
| 72.625 | 0.02 |
| 51.375 | 0.14 |
| 49.750 | 0.05 |
| 89.750 | 0.02 |
| 90.250 | 0.03 |
| 92.500 | 0.10 |
| 33.000 | 0.04 |
| 65.500 | 0.20 |
| 47.250 | 0.01 |
| 91.125 | 0.32 |

**Procedure** *GreedyExt*($n$, $PN$, $freq\_table$);
**begin**
/* Step 1: construct array $HFQ[i]$ */
  sort table $freq\_table$ according to $freq_i$
    in a descending order and store the result in $f\_table$;
  $NumHFQ := \lfloor (PN-1)/2 \rfloor$;
  sort $NumHFQ$ values $f\_table.V_i$
    in an ascending order, $1 \leq i \leq NumHFQ$,
    and record the result in array $HFQ$;
/* Step 2: according to $HFQ[i]$, each range $R_{2*i}$ can be defined as follows. */
  **for** $i := 1$ to $NumHFQ$ **do**
    $R_{2*i} = HFQ[i]$;
/* Step 3: each range $R_{2*i-1}$ can be defined as follows, $1 \leq i \leq (NumHFQ+1)$. */
  let $R_1$ be a range such that $R_1 < HFQ[1]$;
  **for** $i := 2$ to $NumHFQ$ **do**
    let $R_{2*i-1}$ be a range
      such that $HFQ[i-1] < R_{2*i-1} < HFQ[i]$;
  let $R_{2*(NumHFQ)+1}$ be a range
    such that $HFQ[NumHFQ] < R_{2*(NumHFQ)+1}$;
**end**;

Figure 1: The *GreedyExt* procedure

Table 6: Table $f\_table$ for the GreedyExt strategy after sorting

| $Val_i$ | $freq_i$ |
|---|---|
| 91.125 $(HFQ)$ | 0.32 |
| 65.500 $(HFQ)$ | 0.20 |
| 51.375 | 0.14 |
| 92.500 | 0.10 |
| 36.625 | 0.06 |
| 49.750 | 0.05 |
| 33.000 | 0.04 |
| 90.250 | 0.03 |
| 72.625 | 0.02 |
| 89.750 | 0.02 |
| 24.500 | 0.01 |
| 47.250 | 0.01 |

Table 7: The process of the GreedyExt strategy

| $Val_i$ | $freq_i$ | Step 1 | Step 2 | Step 3 |
|---|---|---|---|---|
| 24.500 | 0.01 | | | |
| 33.000 | 0.04 | | | |
| 36.625 | 0.06 | | | |
| 47.250 | 0.01 | | | $R_1$ $(BM_1)$ |
| 49.750 | 0.05 | | | |
| 51.375 | 0.14 | | | |
| 65.500 | 0.20 | $HFQ[1]$ | $R_2$ | $R_2$ $(BM_2)$ |
| 72.625 | 0.02 | | | |
| 89.750 | 0.02 | | | $R_3$ $(BM_3)$ |
| 90.250 | 0.03 | | | |
| 91.125 | 0.32 | $HFQ[2]$ | $R_4$ | $R_4$ $(BM_4)$ |
| 92.500 | 0.10 | | | $R_5$ $(BM_5)$ |

3 of Table 7, the final results are shown as follows: $R_1 < 65.500$ $(= HFQ[1])$, $65.500$ $(= HFQ[1]) < R_3 < 91.125$ $(= HFQ[2])$ and $91.125$ $(= HFQ[2]) < R_5$. There are 5 $(= PN)$ bitmap vectors, $BM_1$ to $BM_5$, for five ranges, $R_1$ to $R_5$, respectively, which are built for the frequencies of users' exact queries in Table 5. Therefore, a large number of users' exact queries on data records with the same value of the attribute can be answered quickly and directly from the disk, instead of checking against the query condition. Then, the average response time is reduced.

### 3.2. The GreedyRange Strategy for Range Queries

The basic idea of the GreedyRange strategy is to decide the ranges according to the frequencies of users' queries on some ranges of values of the attribute in the history. Take Table 1 as an example. Table 8 shows the history of the users' queries on the attribute 'Closing Price' which are ranged between $Val_i$ and $Val_{i+1}$, where $Val_{i+1} = Val_i + fixR$. For example, the range $r_1$ ranges between 0 and 10, where $fixR$ is equal to 10. The frequency, $freq_i$, is calculated by $cq_i/tol$, where $cq_i$ is the count of the range query for data records form $Val_i$ to $Val_{i+1}$, and is $tol = \sum_{i=1}^{n} cq_i$, $1 \leq i \leq n$ and recorded in $freq\_table$. In the following example, the frequency, $freq_i$, for each range $r_i$ of the attribute for the query have been already calculated and are shown in Table 8. If we limit the storage of the bitmap index to be $M$ bits, we can use the $PN$ bitmap vectors to construct the bitmap index, where $PN = \lfloor M/n \rfloor$. In our example, we assume that there are 1000 records and 5000 bits as the offered storage space. Therefore, we construct the bitmap index by using $5 = (\lfloor 5000/1000 \rfloor)$ bitmap vectors or even smaller than 5 bitmap vectors.

Table 8: Table $freq\_table$: an example of the GreedyRange strategy

| $r_i$ | $Val_i$ | $freq_i$ |
|---|---|---|
| $r_1$ | 0 | 0 |
| $r_2$ | 10 | 0 |
| $r_3$ | 20 | 0.01 |
| $r_4$ | 30 | 0.10 |
| $r_5$ | 40 | 0.06 |
| $r_6$ | 50 | 0.14 |
| $r_7$ | 60 | 0.20 |
| $r_8$ | 70 | 0.02 |
| $r_9$ | 80 | 0.02 |
| $r_{10}$ | 90 | 0.35 |

Table 9: Table $f\_table$ for the GreedyRange strategy after sorting

| $r_i$ | $Val_i$ | $freq_i$ |
|---|---|---|
| $r_{10}$ | 90 $(HFQ)$ | 0.35 |
| $r_7$ | 60 $(HFQ)$ | 0.20 |
| $r_6$ | 50 | 0.14 |
| $r_4$ | 30 | 0.10 |
| $r_5$ | 40 | 0.06 |
| $r_8$ | 70 | 0.02 |
| $r_9$ | 80 | 0.02 |
| $r_3$ | 20 | 0.01 |
| $r_1$ | 0 | 0 |
| $r_2$ | 10 | 0 |

The GreedyRange procedure is shown in Figure 2. Take Table 8 as an example. First, we construct the table $freq\_table$ and sort table $freq\_table$, according to the frequency of each query, $freq_i$, in a descending order, and record the result in $f\_table$, as shown in Table 9. Since $PN$ $(= 5)$, we have $NumHFQ = \lfloor (5 - 1)/2 \rfloor = 2$. That is, we find two values, 100 and 70, which have the highest two frequencies of the query in $f\_table$, respectively. Then, we sort values, 90 and 60, in an ascending order, and record them in $HFQ$. Table 10 illustrates the GreedyExt strategy step by step. In Step 1, we obtain $HFQ[1] = 60$ and $HFQ[2] = 90$.

Next, in Step 2 of Figure 10, we obtain $60(= HFQ[1]) < R_2 \leq 70$ $(= HFQ[1] + fixR)$ and $90(=$

**Procedure** *GreedyRange*($n$, $PN$, $freq\_table$);
**begin**
/* Step 1: construct array $HFQ[i]$ */
sort table $freq\_table$ according to $freq_i$
in a descending order and store the result in $f\_table$;
$NumHFQ := \lfloor (PN-1)/2 \rfloor$;
sort $NumHFQ$ values $f\_table.V_i$
in an ascending order, $1 \leq i \leq NumHFQ$,
and record the result in array $HFQ$;
/* Step 2: according to $HFQ[i]$, each range $R_{2*i}$ can be defined as follows. */
**for** $i := 1$ to $NumHFQ$ **do**
let $R_{2*i}$ be a range
such that $HFQ[i] < R_{2*i} \leq HFQ[i] + fixR$;
/* Step 3: each range $R_{2*i-1}$ can be defined as follows, $1 \leq i \leq (NumHFQ + 1)$. */
let $R_1$ be a range such that $R_1 \leq HFQ[1]$;
**for** $i := 2$ to $NumHFQ$ **do**
let $R_{2*i-1}$ be a range
such that $HFQ[i-1] + fixR < R_{2*i-1} \leq HFQ[i]$;
let $R_{2*(NumHFQ)+1}$ be a range
such that $HFQ[NumHFQ] + fixR < R_{2*(NumHFQ)+1}$;
**end**;

Figure 2: The *GreedyRange* procedure

Table 10: The process of the GreedyRange strategy

| $r_i$ | $Val_i$ | $freq_i$ | Step 1 | Step 2 | Step 3 |
|---|---|---|---|---|---|
| $r_1$ | 0 | 0 | | | |
| $r_2$ | 10 | 0 | | | |
| $r_3$ | 20 | 0.01 | | | $R_1$ ($BM_1$) |
| $r_4$ | 30 | 0.10 | | | |
| $r_5$ | 40 | 0.06 | | | |
| $r_6$ | 50 | 0.14 | | | |
| $r_7$ | 60 | 0.20 | $HFQ[1]$ | $R_2$ | $R_2$ ($BM_2$) |
| $r_8$ | 70 | 0.02 | | | $R_3$ ($BM_3$) |
| $r_9$ | 80 | 0.02 | | | |
| $r_{10}$ | 90 | 0.35 | $HFQ[2]$ | $R_4$ | $R_4$ ($BM_4$) |

$HFQ[2]) < R_4 \leq 100$ ($= HFQ[2] + fixR$). Finally, in Step 3 of Figure 10, the final results are as follows: $R_1 \leq 60$ ($= HFQ[1]$) and $70$ ($= HFQ[1] + fixR$) $\leq R_3 < 90$ ($= HFQ[2]$). Therefore, only 4 bitmap vectors, $BM_1$ to $BM_4$, for ranges $R_1$ to $R_4$, respectively, are built for the history of users' range queries in Table 8. A large number of users' range queries on data records with the same range of the attribute can be answered quickly and directly from the disk, instead of checking against the query condition. Then, the average response time could be reduced in most of situations.

## 4. Performance Study

Since the task of fetching data from the disk has a direct effect on the response time, we take the average *number of disk accesses* as the performance measure. Let those $n$ sorted data records, $V_{ij}$, be divided into $PN$ bitmap vector, where $V_{ij}$ is the $j$th data record in the $i$th bitmap vector, and each bitmap vector, $BM_i$ contains $k_i$ data records. Suppose the probability of each data record queried is identical in the data warehouse, the average number of disk accesses is calculated as follows. Take the bitmap vector $BM_1$ as an example. If we want to search the data record $V_{11}$, we must access $k_1$ data records form the disk. Similarly, we access $k_1$ data records for $V_{12}, V_{13}, \cdots, V_{1k_1}$, and the number of disk accesses for the data records in $BM_1$ is $k_1 * k_1$. The total number of disk accesses for all data records is $k_1*k_1+k_2*k_2+\cdots+k_{PN}*k_{PN} = {k_1}^2+{k_2}^2+\cdots+{k_{PN}}^2$ and the average number of disk accesses is

$({k_1}^2 + {k_2}^2 + \cdots + {k_{PN}}^2)/n = \sum_{i=1}^{PN} {k_i}^2/n$.

When we use the GreedyExt or GreedyRange strategies by considering the history of queries, the number of disk accesses could be decreased. We construct the bitmap vectors, $BM_i$, according to the frequent queries, which are recorded in $HFQ$. If we have the query which asks the values that are part of $HFQ$, we can directly access the data records form the disk according to the bitmap vector. Therefore, the average number of disk accesses, which is $\sum_{i=1}^{PN} {k_i}^2/n$ originally can be reduced to

$((1 - \sum_{i=1}^{NumHFQ} f_i) * \sum_{v_j \notin HFQ, v_j \in R_i} k_i + \sum_{i=1}^{NumHFQ} f_i * 1)/n$

based on the GreedyExt and GreedyRange strategies, where 1 is the summation of the frequency of the queries and the other variables are shown in Table 4. If the queries are recorded in $HFQ$, we only use the bitmap vector and access data from disk once. On the other hand, if the queries do not be recorded in $HFQ$, we compute that the number of disk accesses is equal to $((1-\sum_{i=1}^{NumHFQ} f_i)*\sum_{v_j \notin HFQ, v_j \in R_i} k_i)$ by considering the queries which are not recorded in $HFQ$.

Take Table 7 as an example. Two distinct values 65.500 and 91.125 are recorded in $HFQ$ and $NumHFQ = 2$. For those queries on values recorded in $HFQ$, we can use only the bitmap vectors, $BM_2$ and $B_4$, which takes one disk access, respectively. When we consider the queries on the other values except the values recorded in $HFQ$, the number of disk accesses is equal to $(1 - 0.20 - 0.32) * (6^2 + 3^2 + 1^2) = 22.08$. Therefore, the average number of disk accesses is

$((1 - \sum_{i=1}^{NumHFQ} f_i) * \sum_{v_j \notin HFQ, v_j \in R_i} k_i + \sum_{i=1}^{NumHFQ} f_i * 1)/n$
$= ((1-0.20-0.32)*(6^2+3^2+1^2)+(0.20+0.32)*1)/12$
$= 1.83$

as compared to

$\sum_{i=1}^{PN=5} {k_i}^2/n = (6^2 + 1^2 + 3^2 + 1^2 + 1^2)/12 = 4$.

Take Table 10 as another example. Two ranges $r_7$

and $r_{10}$ are recorded in $HFQ$ and $NumHFQ = 2$. For those queries on ranges recorded in $HFQ$, we can use only the bitmap vectors $BM_2$ and $BM_4$, which takes one disk access, respectively. When we consider the queries on the other ranges except the ranges recorded in $HFQ$, the number of disk accesses is equal to $(1 - 0.20 - 0.35) * (6^2 + 2^2) = 18$. Therefore, the average number of disk accesses is

$((1 - \sum_{i=1}^{NumHFQ} f_i) * \sum_{v_j \notin HFQ, v_j \in R_i} k_i + \sum_{i=1}^{NumHFQ} f_i * 1)/n$
$= ((1 - 0.20 - 0.35) * (6^2 + 2^2) + (0.20 + 0.35) * 1)/10$
$= 1.855$

as compared to
$\sum_{i=1}^{PN=4} {k_i}^2/n = (6^2 + 1^2 + 2^2 + 1^2)/10 = 4.2.$

## 5. Conclusion

To reduce the response time on fetching the data records with the high frequency of the query from the disk, we consider the history of users' queries on the design of the partitioning strategy for the range-based bitmap index. We have presented the GreedyExt and GreedyRange strategies for answering exact queries and range queries, respectively. Based on these two strategies, the data records with the high frequency of the query can be quickly and directly accessed. We have made the analysis on the performance of our GreedyExt and GreedyRange strategies. We have proved that the two strategies can reduce the average number of disk accesses a lot in order to reduce the response time.

## Acknowledgment

This research was supported in part by the National Science Council of Republic of China under Grant No. NSC-95-2221-E-110-101 and National Sun Yat-Sen University. The authors like to thank "Aim for Top University Plan" project of NSYSU and Ministry of Education, Taiwan, for partially supporting the research.

## References

[1] K. Aouiche, J. Darmont, O. Boussaid and F. Bentayeb, "Automatic Selection of Bitmap Join Indexes in Data Warehouses," *Proc. of the 7th Int. Conf. on Data Warehousing and Knowledge Discovery,* pp. 64–73, 2005.

[2] S. Azefack, K. Aouiche and J. Darmont, "Dynamic Index Selection in Data Warehouses," *Proc. of the 4th Int. Conf. on Innovations in Information Technology,* pp. 1–5, 2007.

[3] L. Bellatreche, K. Boukhalfa and M. Mohania, "Pruning Search Space of Physical Database Design," *Proc. of the 18th Int. Conf. on Database and Expert Systems Applications,* pp. 479–488, 2007.

[4] Z. Z. Gong, K. F. Hu and Q. Li. Da, "A Grouping Aggregation Algorithm Based on the Dimension Hierarchical Encoding in Data Warehouse," *Proc. of the 6th Int. Conf. on Computer Information Systems and Industrial Management Applications,* pp. 135–142, 2007.

[5] N. Goyal, S. K. Zaveri and Y. Sharma, "Improved Bitmap Indexing Strategy for Data Warehouses," *Proc. of the 9th Int. Conf. on Information Technology,* pp. 213–216, 2006.

[6] A. Gupta, K. C. Davis and J. G. Litton, "Performance Comparison of Property Map and Bitmap Indexing," *Proc. of the 5th ACM Int. Workshop on Data Warehousing and OLAP,* pp. 65–71, 2002.

[7] M. Jurgens and H. J. Lenz, "Tree Based Indexes vs. Bitmap Indexes: A Performance Study," *Int. Journal of Cooperative Information Systems,* Vol. 10, No. 1 pp. 355–376, March 2001.

[8] Y. Lim and M. Kim, "A Bitmap Index for Multidimensional Data Cubes," *Proc. of the 15th Int. Conf. on Database and Expert Systems Applications,* pp. 349–358, 2004.

[9] E. O'Neil, P. O'Neil and K. Wu, "Bitmap Index Design Choices and Their Performance Implications," *Proc. of the 11th Int. Symp. on Database Eng. and Applications,* pp. 72–84, 2007.

[10] R. R. Sinha, M. Winslett, Wu. Kesheng, K. Stockinger and A. Shoshani, "Adaptive Bitmap Indexes for Space-Constrained Systems," *Proc. of the 24th Int. Conf. on Data Eng.*, pp. 1418–1420, 2008.

[11] K. Stockinger, "Bitmap Indices for Speeding Up High-Dimensional Data Analysis," *Proc. of the 13th Int. Conf. on Database and Expert Systems Applications,* pp. 881–890, 2002.

[12] K. Stockinger, K. Wu and A. Shoshani, "Evaluation Strategies for Bitmap Indices with Binning," *Proc. of the 15th Int. Conf. on Database and Expert Systems Applications,* pp. 120–129, 2004.

[13] G. Velinov, D. Gligoroski and M. K. Popovska, "Hybrid Greedy and Genetic Algorithms for Optimization of Relational Data Warehouses," *Proc. of the 25th Int. Conf. on Artificial Intelligence and Applications,* pp. 470–475, 2007.

[14] R. Wrembel and C. Koncilia, *Data Warehouses and OLAP: Concepts, Architectures, and Solutions,* Idea Group Inc, Hershey, Pennsylvania, USA, 2007.

[15] K. L. Wu and P. S. Yu, "Range-Based Bitmap Indexing for High-Cardinality Attributes with Skew," *Proc. of Int. Conf. on Computer Software and Applications,* pp. 61–67, 1998.