WALAILAK JOURNAL
http://wjst.wu.ac.th Information Technology

Optimizing Range Query Processing for Dual Bitmap Index[†]

Naphat KEAWPIBAL, Ladda PREECHAVEERAKUL and Sirirut VANICHAYOBON[*]

Department of Computer Science, Faculty of Science, Prince of Songkla University, Songkhla 90110, Thailand

([*]Corresponding author's e-mail: sirirut.v@psu.ac.th)

Received: 16 June 2017, Revised: 10 December 2017, Accepted: 19 December 2017

Abstract

Bitmap-based indexes are known to be the most effective indexing method for retrieving and answering selective queries in a read-only environment. Various types of encoding bitmap indexes significantly improve query time efficiency by utilizing fast Boolean operations directly on the index before retrieving the raw data. In particular, the dual bitmap index improves the performance of equality queries in terms of the space vs. time trade-off. However, the performance of range queries is unsatisfactory. In this paper, an optimizing algorithm is proposed to improve the range query processing for the dual bitmap index. The results of the experiment conducted show that the proposed algorithm, called Dual-simRQ, reduces the number of bitmap vectors scanned and the Boolean operations performed, which impacts the overall performance for range query processing.

Keywords: Encoding bitmap indexes, dual bitmap index, range query processing, Dual-simRQ

Introduction

Advances in technology have caused a data explosion and a massive amount of data is generated by various devices [1-3]. The problem of data management, including storage requirements and query processing, has become more difficult. The issue of query performance has been extensively studied, especially in regard to data warehouses and scientific data management applications [3-6]. Generally, a data warehouse is deployed as a read-only environment and is used to support forecasting and decision-making applications [3-5,7,8]. Therefore, a lot of complex queries (i.e., equality, range, and membership queries) are submitted to data warehouses in order to retrieve data and operate further processes. Due to the massive amount of data and the complexity of some queries, a long time might be taken to process a query before a result can be returned [4,5]. As a result, query performance and reducing query processing time have become an important issue in data warehousing.

Indexing is the best-known method of speeding up search operations and the retrieval process, without requiring additional hardware [4,9]. Indexing methods can be categorized as hash-based indexes, tree-based indexes, and bitmap-based indexes [10-13]. In particular, it is easy to represent data in a binary format in bitmap-based indexes and query processing can be executed extremely fast by applying the Boolean operators, AND, OR, NOT, and XOR, directly on the index before accessing raw data [4,5]. A simple bitmap index [14] was first introduced and recommended for space- and time-efficiency for the attribute with low cardinality, for example, the attribute, sex which has a cardinality of 2. Unfortunately, when the cardinality is high, the size of a bitmap index is significantly increased which leads to a problem

[†]Presented at the 14th International Joint Conference on Computer Science and Software Engineering: July 12th -14th, 2017

with the bitmap index space requirement. To deal with this problem, several strategies have been employed to improve the simple bitmap index which can be divided into 4 categories [6,9,10,15-17]: binning, compressing, encoding, and multi-level and multi-component. Among these 4, encoding strategies relate to encoding schemes for representing data by using a small number of bitmap vectors while maintaining the query processing time.

A number of sophisticated encoding bitmap indexes have been proposed in the last 10 years, including the range bitmap index [15,18], interval bitmap index [15], encoded bitmap index [19], scatter bitmap index [20,21], and dual bitmap index [22]. Among these 5 encoding bitmap indexes, the range bitmap index and interval bitmap index produce a large number of bitmap vectors when dealing with high cardinality attributes, which is likely to have an impact on the space requirement, while the encoded bitmap index generates the least number of bitmap vectors but its query processing time is unsatisfactory [23-26]. On the other hand, the dual bitmap index requires a smaller number of bitmap vectors than the range, interval, and scatter bitmap indexes. Specifically, to answer an equality query in the form "$A = v$", the dual bitmap index uses 2 bitmap vectors and one bitwise-AND operation. As a result, the query processing time of the dual bitmap index is extremely fast. Further, to answer a range query in the form "$v_1 \leq A \leq v_2$", each of 2 bitmap vectors is accessed and performed by a bitwise-AND operation to obtain each query value, and then, bitwise-OR operations are used to get the final result of the query. The dual bitmap index has therefore been demonstrated to improve query performance for an equality query from the point of view of a trade-off between space required and time consumed [22]. In contrast, the query performance for a range query is unsatisfactory.

In this paper, an algorithm is presented to improve the query performance of range query processing for the dual bitmap index, namely the Dual-simRQ. The objective of the algorithm is to simplify the retrieval function by eliminating unnecessary scanning of bitmap vectors and bitwise operations. The algorithm provides less complexity of the retrieval function which is an impact on the execution time required for accessing bitmap vectors as well as the time used for bitwise operations.

RID	A	S^0	S^1	S^2	S^3	S^4	S^5	S^6	S^7	S^8	S^9	S^{10}	S^{11}	S^{12}	S^{13}	S^{14}	D^0	D^1	D^2	D^3	D^4	D^5
1	3	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1
2	9	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	0	0
3	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0
4	8	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	1	0
5	10	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	0
6	3	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1
7	4	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1
8	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
9	12	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1	0	0	0
10	5	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
11	13	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0
12	6	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0
13	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
100,000	2	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1
(a)		(b)															(c)					

Figure 1 An example of Table *T* (a) a simple bitmap index (b) and a dual bitmap index on attribute *A* with *C* = 15 (c).

Literature review

Simple bitmap index [14]

As mentioned above, a simple bitmap index is a well-known method of indexing which improves the speed of query processing with stored data, especially in a data warehouse. The simple bitmap index

consists of *C* bitmap vectors, where *C* is the number of distinct values in the indexed attribute (cardinality), with each bitmap vector corresponding to one precise value of the indexed attribute. To represent value $'v'$, the i[th] bit in the bitmap vector representing value $'v'$ is set to 1 if the i[th] row of the indexed attribute contains value $'v'$, otherwise, the bit is set to 0.

Assume that a domain of attribute *A* given by table *T* is {0, 1, 2, ..., 14}, as shown in **Figure 1(a)**. The simple bitmap index consists of 15 bitmap vectors. As shown in **Figure 1(b)**, for example, in bitmap vector S^3 used for representing value $'3'$, the 1[st] and 6[th] bit in S^3 are set to 1. When answering equality queries, one bitmap vector associated with the query value is scanned without performing any bitwise operations. For example, in order to evaluate the equality query "$A = 3$", the bitmap vector S^3 is scanned. The row that contains bit 1 is returned as the result of this query, i.e., the 1[st] and 6[th] rows. However, when range queries are submitted, there are more than one bitmap vectors read, associated with the query values, and a bitwise-OR operation is performed on them before answering the query. For example, in order to evaluate the range query "$3 \leq A \leq 13$", the bitmap vectors between S^3 to S^{13} are scanned. Then, bitwise-OR operations are performed among those bitmap vectors as $S^3 \vee S^4 \vee S^5 \vee S^6 \vee S^7 \vee S^8 \vee S^9 \vee S^{10} \vee S^{11} \vee S^{12} \vee S^{13}$ to obtain the final result. Table *T* in **Figure 1(a)** is used as the example for the remainder of this paper.

Dual bitmap index [22]

The key design of a dual bitmap index is the use of 2 bitmap vectors to represent indexed attribute values by computing the values of *r* and *s* using Eqs. (1) and (2), respectively. The dual bitmap index consists of $\left\lceil \sqrt{2C + 0.25} + 0.5 \right\rceil$ bitmap vectors.

$$r = \left\lceil \sqrt{2(HiC - v) + 0.25} + 0.5 \right\rceil \quad (1)$$

$$s = \left\lceil r - 1 - \left[\left(\frac{(n-r)(n-r-1)}{2} \right) \bmod r \right] \right\rceil \quad (2)$$

where *v* is the attribute value of any row, *n* is the total number of bitmap vectors created, and $HiC = \binom{n}{2}$

By using Eq. (3) to represent attribute values, each value is identified by 2 bitmap vectors, i.e., D^r and D^s. If the value at i[th] row corresponds to the values of *r* and *s*, the bit at i[th] row of D^r and D^s is set to 1. Otherwise, it is set to 0.

$$D^j = \begin{cases} 1, & j = r \text{ and } j = s, \\ 0, & \text{otherwise.} \end{cases} \quad (3)$$

For example, using table *T*, the dual bitmap index consists of 6 bitmap vectors, $\{D^0, D^1, D^2, D^3, D^4, D^5\}$. To represent value $'3'$ at the first row of attribute *A*, the 1[st] bit of D^5 and D^1 is set to 1 since the values of *r* and *s* are 5 and 1, respectively, while every 1[st] bit remaining in the other bitmap vectors is set to 0. **Figure 1(c)** shows a complete dual bitmap index created on attribute *A* with 15 cardinalities. The evaluation of an equality query in the dual bitmap index performs the retrieval function as in Eq. (4).

$$"A = v" = D^r \wedge D^s \quad (4)$$

To answer an equality query, for example, in order to evaluate the equality query "$A = 3$", the retrieval function for this query is created as $D^5 \wedge D^1$ since the values of *r* and *s* is 5 and 1, respectively. The result of this equality query is the 1[st] and 6[th] row. To answer range queries, 2 bitmap vectors identifying each query value can be dynamically created using Eq. (4). Then, bitwise-OR operations are used to obtain the final result. For example, to evaluate the range query "$3 \leq A \leq 13$", the retrieval

function for this query is created as $(D^5 \wedge D^1) \vee (D^5 \wedge D^0) \vee (D^4 \wedge D^3) \vee (D^4 \wedge D^2) \vee (D^4 \wedge D^1) \vee (D^4 \wedge D^0) \vee (D^3 \wedge D^2) \vee (D^3 \wedge D^1) \vee (D^3 \wedge D^0) \vee (D^2 \wedge D^1) \vee (D^2 \wedge D^0)$.

As in the above example, a large number of bitmap vectors and logical operations are performed in order to answer a range query when the range of the query values is wide. Therefore, the performance of traditional range query processing by the dual bitmap index is degraded. In this paper, an algorithm is proposed, called Dual-simRQ, to optimize a range query processing for the dual bitmap index by minimizing the number of bitmap vectors scanned and the number of bitwise operations performed.

Materials and methods

This section presents the Dual-simRQ algorithm, which is able to simplify the retrieval function of range query processing with the dual bitmap index. This algorithm consists of 4 phases, 1) building the RsJoin table, 2) building the FreqSorting table, 3) simplifying the retrieval function, and 4) generating a new retrieval function, as described in the following.

RsValue Table

value	*r*	*s*
3	5	1
4	5	0
5	4	3
6	4	2
7	4	1
8	4	0
9	3	2
10	3	1
11	3	0
12	2	1
13	2	0

(a)

RsJoin Table

unique_r	*join_s*
5	$\{1, 0\}$
4	$\{3, 2, 1, 0\}$
3	$\{2, 1, 0\}$
2	$\{1, 0\}$

(b)

Figure 2 The result of the temporary RsValue table (a) and the RsJoin table for the range query "$3 \leq A \leq 13$" (b).

1) Building the RsJoin table

Generally, each indexed attribute value is identified by both the values of *r* and *s* in the dual bitmap index. When a range query is submitted, an RsValue table is temporarily created by storing a pair of values of *r* and s corresponding to all the query values. Obviously, each distinct value of *r* relates to different values of *s*. Therefore, a set of values for *s* can be created which relate to each distinct value of *r*, where the value of *r* and associated set of values of *s* are stored in an RsJoin table. For example, in order to evaluate a range query in the form, "$3 \leq A \leq 13$", the RsValue table is firstly created by calculating the values of *r* and *s* for the query values between 3 and 13, as shown in **Figure 2(a)**. Subsequently, the RsJoin table is built by finding the distinct values of *r* and their associated values of *s*, as shown in **Figure 2(b)**. In the first row, the value of *r* is 5 which relates to the values 1, and 0 of *s*, which yields set {1, 0} for *r* = 5.

2) Building the FreqSorting table

The objective of this phase is to find the unique sets of *join_s* (called *unique_s*) given by the RsJoin table and their frequency, resulting in a frequency table. Basically, the frequency of *unique_s* provides an easy combination of the relevant values of *r* which correspond to *unique_s*. Furthermore, the number of items in the *unique_s* has an impact on the reduction of the number of bitwise operations performed

among the relevant bitmap vectors. Therefore, the FreqSorting table is created by arranging the frequency of *unique_s* from high to low because the values of *unique_s* with high frequency relate to the number of relevant values of *r*. In addition, the smaller sets of *unique_s* may possibly be subsets of a larger set. Therefore, in order to eliminate the redundant items, the number of items in each of *unique_s* is arranged from small to large if the frequency is the same. **Figure 3(a)** illustrates the frequency belonging to each of *unique_s* without sorting, which yields sets of {1,0}, {3,2,1,0}, and {2,1,0} with the frequency of 2, 1, and 1, respectively. The frequency of set {3,2,1,0} and {2,1,0} is equal to 1 but the number of items in the latter set is less, i.e., there are 3 items in set {2,1,0} while there are 4 items in set {3,2,1,0}. Therefore, the FreqSorting table consists of the sets of {1, 0}, {2, 1, 0}, and {3, 2, 1, 0}, as shown in **Figure 3(b)**.

Frequency Table

freq	*unique_s*
2	$\{1,0\}$
1	$\{3,2,1,0\}$
1	$\{2,1,0\}$

(a)

FreqSorting Table

freq	*sort_s*
2	$\{1,0\}$
1	$\{2,1,0\}$
1	$\{3,2,1,0\}$

(b)

Figure 3 The result of the Frequency table (a) and the FreqSorting table for the range query "$3 \leq A \leq 13$" (b).

Algorithm 1 Dual-simRQ algorithm

Input: *RsJoin* and *FreqSorting* tables
Output: The array of *minS* and *minR*

```
1: ind ← 0
2: for each row fs in FreqSorting table do                ▷ sort_s > 1 items
3:     S ← fs.sort_s
4:     R ← ∅
5:     if len(S) > 1 then
6:         for each row rs in RsJoin table do
7:             if S ⊆ rs.join_s then
8:                 Add rs.unique_r to a new item of R
9:                 Remove items in rs.join_s from item in S
10:            end if
11:        end for
12:        Update FreqSorting table
13:        minS[ind] ← S
14:        minR[ind] ← R
15:        ind ← ind + 1
16:    end if
17: end for
18: for each row fs in FreqSorting table do          ▷ sort_s has only 1 item
19:    S ← fs.sort_s
20:    R ← ∅
21:    if S ≠ ∅ then
22:        for each row rs in RSJoin table do
23:            if S = rs.join_s then
24:                Add rs.unique_r to a new item of R
25:            end if
26:        end for
27:        minS[ind] ← S
28:        minR[ind] ← R
29:        ind ← ind + 1
30:    end if
31: end for
32: return minS, minR
```

Figure 4 The Dual-simRQ algorithm.

3) Simplifying the retrieval function

This phase generates a simplification of the retrieval function by utilizing the RsJoin and the FreqSorting tables as an input and produces an array structure of *minS* and *minR* corresponding to the values of *r* and *s*, respectively as an output. **Figure 4** shows the detail of the Dual-simRQ algorithm. There are 32 steps in the algorithm. In the 1st step, the value of *ind* is initially set to 0. In the 2nd to 17th steps, the relevant values of *r* are combined into a set corresponding to each *sort_s* if the number of items in the *sort_s* is more than one. In the 3rd and 4th steps, each *sort_s* retrieved from the FreqSorting table is assigned to a set *S* and an empty set is assigned to a set *R* in order to hold the *sort_s*, and its relevant values of *r*, respectively. In the 5th to 11th steps, the values of *unique_r* in the RsJoin table are gathered into a set if the set *S* is a subset of each *join_s* in the RsJoin table, which yields a set *R*. Then, the *join_s* is updated by removing the redundant items from *S* at the 9th step. In the 12th step, the FreqSorting table is updated by removing the redundant items from *S* if the set *S* is a subset of each *sort_s*. In the 13th and 14th steps, the results of sets *S* and *R* are assigned to *minS* and *minR*, respectively, at the index of *ind*. The value of *ind* is increased by 1 at the 15th step. In the 18th to 31st steps, the sets *sort_s* containing only one item are executed by repeating the same process to find the relevant values of *r* in the RsJoin table corresponding to the each of set *sort_s*. Consequently, each index of the array *minS* and *minR* consists of sets of *S* and sets of *R*, respectively, as shown in **Figure 5**.

MinTerm

ind	$minS[ind]$	$minR[ind]$
0	$\{1, 0\}$	$\{5, 4, 3, 2\}$
1	$\{3, 2\}$	$\{4\}$
2	$\{2\}$	$\{3\}$

Figure 5 The result of array *minS* and *minR* for the range query "$3 \leq A \leq 13$".

4) Generating a new retrieval function phase

In this phase, a new retrieval function is generated by retrieving all the items within the array *minS* and *minR* at every index, as shown in **Figure 5**. Each item of *minS* refers to values of *s* while each item of *minR* refers to values of *r*. These items within *minS* and *minR* relate to an identifier of a bitmap vector for the dual bitmap index. Therefore, all the related bitmap vectors specified by items in *minS* are performed by a bitwise-OR operation as well as all items in *minR*. Then, each result of *minS* and *minR* is performed in a bitwise-AND operation to obtain a sub-result of at any index value. For example, at the *ind* = 0, *minS*[0] consists of 2 values (i.e., 1, 0) while *minR*[0] consists of 4 values (i.e., 5, 4, 3, and 2). The bitwise-OR operations are performed between bitmap vectors D^0 and D^1, to give the result for *minS*[0]. Subsequently, the bitwise-OR operations are performed among bitmap vectors D^5, D^5, D^4, and D^2, to give the result for *minR*[0]. After that, the results of *minS*[0] and *minR*[0] are performed in a bitwise-AND to obtain the sub-result for *ind* = 0. Finally, the end result is obtained by performing bitwise-OR operations among all the sub-results. Therefore, the retrieval function for the range query "$3 \leq A \leq 13$" is $\big((D^1 \vee D^0) \wedge (D^5 \vee D^4 \vee D^3 \vee D^2)\big) \vee \big((D^3 \vee D^2) \wedge D^4\big) \vee (D^2 \wedge D^3)$.

Results and discussion

This section presents a theoretical analysis and experimental results in which the range query processing time for 2 algorithms, the traditional range query processing (Dual-tradRQ) and the Dual-simRQ, was compared.

Table 1 A comparative study of Dual-tradRQ and Dual-simRQ for range query processing

Bitmap index algorithm	Space requirement	Range query processing time	
		Number of bitmap vectors scanned	Number of Boolean operation used
Dual-tradRQ	$\lceil\sqrt{2C+0.25}+0.5\rceil$	$2k$	$2k$-1
Dual-simRQ		[3, $2k$]	[2, $2k$-1]

C is the cardinality of the attribute and *k* is the range of query values

Table 1 shows the theoretical analysis of the 2 algorithms for range query processing based on a dual bitmap index. Both the algorithms require the same number of bitmap vectors, i.e., $\lceil\sqrt{2C+0.25}+0.5\rceil$. For the performance of range query processing, the Dual-tradRQ scans 2*k* bitmap vectors and uses 2*k*-1 Boolean operators. The number of bitmap vectors scanned for the Dual-simRQ varies from 3 to 2*k* bitmap vectors and the number of Boolean operations used also varies between 2 and 2*k*-1. For example, for Dual-simRQ, if it is assumed that there are 2 query values submitted and the former retrieval function can be reduced, this implies that those query values have the same value of *r*. Therefore, the minimum number of bitmap vectors scanned is 3 and the minimum number of Boolean operations used is 2 (1 AND and 1 OR). On the other hand, if the retrieval function cannot be reduced, this means that the query values have different values of *r*, in which case, the number of bitmap vectors scanned is 4 and the number of Boolean operations used is 3 (i.e., 2 ANDs and 1 OR).

The experiments were conducted on a 3.20 GHz Intel® Core™ i5-4570HQ with a 4.00 GB main memory, running on 64-bit Windows 10 as an operating system. The range query processing time includes the CPU and I/O time required to answer range queries. In particular, the CPU time for the Dual-simRQ includes the processes involved in simplifying the retrieval function which occupies a small part of execution time.

The experiment used the TPC-H benchmark data set retrieved from [27]. This benchmark consists of 8 tables. The experiment was conducted on the tables PART and ORDERS which contain 20,000,000 and 150,000,000 records, respectively. Three representative attributes were used from table PART and one representative attribute was used from table ORDERS, as follows (the values in parentheses indicate the cardinality of the corresponding attribute). The attributes selected were: BRAND (25), SIZE (50), and TYPE (150) from the table PART, and attribute CLERK (1,000) from the table ORDERS. This provided different cardinalities ranging from low to high. In addition, 5 different range value ratios ranging from 10 to 80 % of the cardinality, representing the number of continuous query values in a range query, were selected.

Figure 6 highlights the abovementioned theoretical analysis by comparing the Dual-simRQ algorithm with the Dual-tradRQ for evaluating range queries on the selected attributes with respect to five range ratios with 4 different cardinalities. The Dual-tradRQ does not perform any simplification of the retrieval function. **Figures 6(a) - 6(d)** show the range query processing time for cardinalities of 25, 50, 150, and 1,000, respectively. When the range ratio increased, the overall query processing time of Dual-tradRQ was significantly increased while that of Dual-simRQ increased only slightly. When the cardinality and the range ratio are low and the retrieval function can be slightly minimized, then, the CPU time used by Dual-simRQ is slightly more than that used by Dual-tradRQ. However, when both the cardinality and range ratio are high, the CPU time used by Dual-simRQ is less than that of Dual-tradRQ. Furthermore, the Dual-simRQ uses less I/O time than Dual-tradRQ for all cardinality and range ratios, which implies that the number of bitmap vectors scanned by Dual-simRQ is less than the number of bitmap vectors scanned by Dual-tradRQ. Overall, the experimental results show that the proposed algorithm, Dual-simRQ, is much faster than Dual-tradRQ in every range ratio for all selected attributes, indicating that the Dual-simRQ is able to optimize the range query processing of the dual bitmap index for every range ratios and different ranges of cardinality.

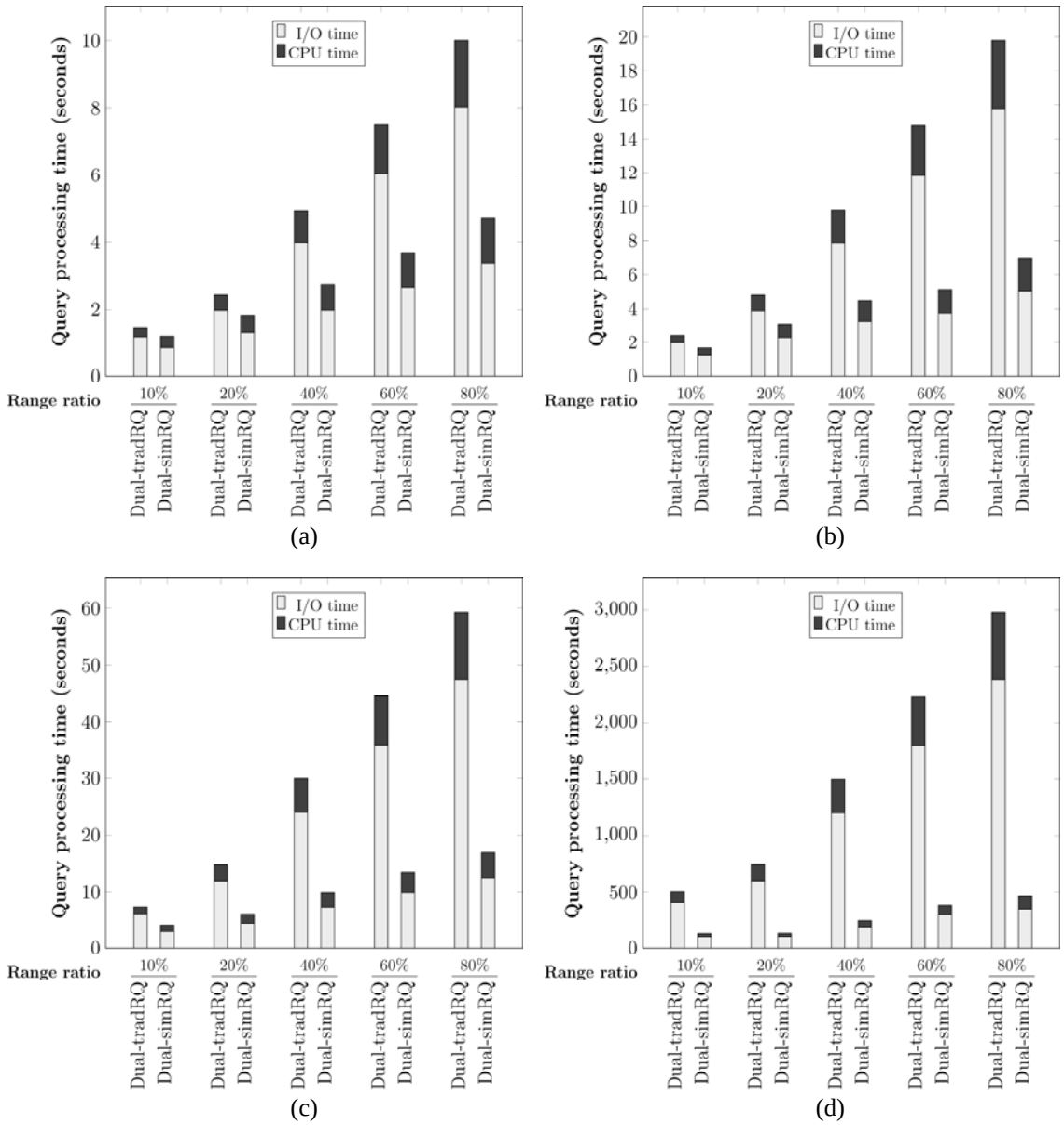

Figure 6 The comparison of range query processing time with 5 range value ratios on the attribute BRAND with C = 25 (a), attribute SIZE with C = 50 (b), attribute TYPE with C = 150 (c), and attribute CLERK with C = 1,000 (d).

Conclusions

With the availability of significantly increasing amounts of data emerges both the challenge and the opportunity to discover large amounts of valuable knowledge. However, the requirement for data storage and the time required to process queries on such large amounts of data has become a major problem when answering complex queries. Various bitmap-based indexes exist which aim to reduce the storage requirement without sacrificing query processing time by performing fast Boolean operations at the index

level. The dual bitmap index optimizes the space vs. time trade-off when answering equality queries. However, the dual bitmap index takes a long time when answering range queries. In this paper, the Dual-simRQ algorithm was introduced, which is capable of reducing the query processing time for range queries based on a dual bitmap index. The Dual-simRQ algorithm eliminates the unnecessary scanning of bitmap vectors and performing of Boolean operations. The results of the experiment described show that the Dual-simRQ is much faster than the Dual-tradRQ algorithm. In addition, future research will investigate a new encoding bitmap index algorithm to optimize the performance of all kinds of selective queries in terms of the space vs. time trade-off.

Acknowledgements

This research was financially supported by Faculty of Science Research Fund: 158002, Prince of Songkla University, Hat Yai, Songkhla.

References

[1] S Sagiroglu and D Sinanc. Big data: A review. *In*: Proceedings of 2013 International Conference on Collaboration Technologies and Systems, San Diego, USA, 2013, p. 42-7.
[2] S Chaudhuri. What next? A half-dozen data management research goals for big data and the cloud. *In*: Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Scottsdale, Arizona, USA, 2012, p. 1-4.
[3] A Katal, M Wazid and R H Goudar. Big data: Issues, challenges, tools and good practices. *In*: Proceedings of 2013 6th International Conference on Contemporary Computing, Noida, India, 2013, p. 404-9.
[4] S Chaudhuri and U Dayal. An overview of data warehousing and OLAP technology. *ACM SIGMOD Rec.* 1997; **26**, 65-74.
[5] CY Chan and YE Ioannidis. Bitmap index design and evaluation. *ACM SIGMOD Rec*. 1998; **27**, 355-66.
[6] K Wu, A Shoshani and K Stockinger. Analyses of multi-level and multi-component compressed bitmap indexes. *ACM Trans. Database Syst*. 2010; 35, 1-52.
[7] K Kambatla, G Kollias, V Kumar and A Grama. Trends in big data analytics. *J. Parallel Distr. Comput.* 2014; **74**, 2561-73.
[8] X Wu, X Zhu, GQ Wu and W Ding. Data mining with big data. *IEEE Trans. Knowl. Data Eng*. 2014; **26**, 97-107.
[9] K Stockinger and K Wu. Bitmap indices for data warehouses. *Data Warehouses OLAP Concepts Architect. Solut.* 2007; **1**, 157-78.
[10] ZQ Abdulhadi, Z Zuping and HI Housien. Bitmap index as effective indexing for low cardinality column in data warehouse. *Int. J. Comput. Appl*. 2013; **68**, 38-42.
[11] LJ Gosink, K Wu, EW Bethel, JD Owens and KI Joy. 2008, Bin- Hash Indexing: A Parallel Method for Fast Query Processing, Technical Report. Laurence Berkeley National Laboratories, USA.
[12] Y Mei, K Ji and F Wang. A survey on bitmap index technologies for large-scale data retrieval. *In*: Proceedings of 2013 6th International Conference on Intelligent Networks and Intelligent Systems, Shenyang, China, 2013, p. 316-9.
[13] Z Chen, Y Wen, J Cao, W Zheng, J Chang, Y Wu, G Ma, M Hakmaoui and G Peng. A survey of bitmap index compression algorithms for big data. *Tsinghua Sci. Tech.* 2015; 20, 100-15.
[14] PO Neil and D Quass. Improved query performance with variant indexes. *ACM SIGMOD Rec*. 1997; **26**, 38-49.
[15] CY Chan and YE Ioannidis. An efficient bitmap encoding scheme for selection queries. *ACM SIGMOD Rec*. 1999; **28**, 215-26.
[16] M Stabno and R Wrembel. RLH: Bitmap compression technique based on run-length and huffman encoding. *Inform. Syst*. 2009; **34**, 400-14.

[17] S Kim, J Lee, SR Satti and B Moon. SBH: Super byte-aligned hybrid bitmap compression. *Inform. Syst*. 2016; **62**, 155-68.
[18] KL Wu and S Yu. Range-based bitmap indexing for high cardinality attributes with skew. *In*: Proceedings of the 22nd Annual International Computer Software and Applications Conference, Vienna, Austria, 1998, p. 61-6.
[19] MC Wu and AP Buchmann. Encoded bitmap indexing for data warehouses. *In*: Proceedings of 14th International Conference on Data Engineering, Orlando, USA, 1998, p. 220-30.
[20] S Vanichayobon, J Manfuekphan and L Gruenwald. Scatter bitmap: Space-time efficient bitmap indexing for equality and membership queries. *In*: Proceedings of 2006 IEEE Conference on Cybernetics and Intelligent Systems, Bangkok, Thailand, 2006, p. 1-6.
[21] W Weahama, S Vanichayobon and J Manfuekphan. Using data clustering to optimize scatter bitmap index for membership queries. *In*: Proceedings of 2009 International Conference on Computer and Automation Engineering, Bangkok, Thailand, 2009, p. 174-8.
[22] N Wattanakitrungroj and S Vanichayobon. Dual bitmap index: Space-time efficient bitmap index for equality and membership queries. *In*: Proceedings of 2006 International Symposium on Communications and Information Technologies, Bangkok, Thailand, 2006, p. 568-73.
[23] GR Alam, MY Arafat, M Kamal and U Iftekhar. A new approach of dynamic encoded bitmap indexing technique based on query history. *In*: Proceedings of 5th International Conference on Electrical and Computer Engineering, Dhaka, Bangladesh, 2008, p. 974-9.
[24] J Sainui, S Vanichayobon and N Wattanakitrungroj. Optimizing encoded bitmap index using frequent itemsets mining. *In*: Proceedings of 2008 International Conference on Computer and Electrical Engineering, Phuket, Thailand, 2008, p. 511-5.
[25] A Keawpibal, N Wattanakitrungroj and S Vanichayobon. Enhanced encoded bitmap index for equality query. *In*: Proceedings of the 8th International Conference on Computing Technology and Information Management, Seoul, South Korea, 2012, p. 293-8.
[26] N Keawpibal, J Duangsuwan, W Wettayaprasit, L Preechaveerakul and S Vanichayobon. DistEQ: Distributed equality query processing on encoded bitmap index. *In*: Proceedings of the 12th International Joint Conference on Computer Science and Software Engineering, Songkhla, Thailand, 2015, p. 309-14.
[27] TPC-H: A Decision Support Benchmark. Available at: http://www.tpc.org/tpch, accessed November 2017.