An Adequate Design for Large Data Warehouse Systems: Bitmap index versus B-tree index

Morteza Zaker, Somnuk Phon-Amnuaisuk, Su-Cheng Haw

***Abstract*—Although creating indexes on database is usually regarded as a common issue, it plays a key role in the query performance, particularly in the case of huge databases like a Data Warehouse where the queries are of complicated and ad hoc nature. Should an appropriate index structure be selected, the time required for query response will decrease extensively. To best of our knowledge, to date no comprehensive guideline has been provided for Data Warehouse analysts to opt for suitable indices. Conventionally, most experts go for the Bitmap index as a preferred indexing technique for cases where the indexed attributes are of few distinct values (i.e., low cardinality). Once the index size is huge, the cardinality of indexed columns increases causing the query response time to rise. On the other hand, owing to its indexing and retrieving mechanisms, B-tree index is assumed to be the adequate technique as the column values increase in cardinality. The paper seeks to illustrate how such assumptions mentioned above may not be true under certain circumstances. Empirical evidence is provided to confirm that even though the level of column cardinality may be determined by the index file size, the query processing time is not necessarily set by the level of column cardinality. Surprisingly, the results also indicate how the Bitmap index can be more expeditious than B-tree index on a large dataset with multi-billion records.**

***Index Terms*—Data warehouse, Bitmap index, B-tree index, Query processing**

I. Introduction

A Data Warehouse (DW) is the foundation for Decision Support Systems (DSS) with a large collection of information that can be accessed through an On-line Analytical Processing (OLAP) application. This large database stores current and historical data that come from several external data sources [1]–[3], [5] . The queries built on DW systems are complex and usually include some join operations that incur computational overhead which rises the response time especially when queries are performed on a large dataset. To increase the performance, DW analysts commonly use some solutions such as indexes, summary tables and partition mechanism [4] .

There are various index techniques supported by database vendors such as Bitmap [4], B-tree [3], [6], [7], [9], Projection [8], Join bitmap [10], and Range base bitmap indices [11] among others. A Bitmap index for example is advisable for a system comprising data that are not frequently updated by many concurrent processes [12]–[14] . This is mainly due to the fact that a Bitmap index stores large amounts of row information in each block of the index structure. In addition, since Bitmap index locking is at the block level, any insert, update, or delete activity may result in locking an entire range of values [16]. By contrast, a B-tree index is adequate for a system which is frequently updated because it does not need re-balancing as frequently as other self-balancing search trees. In addition, all leaf blocks of the tree are at the same depth [7]. Thus, choosing the proper type of index structures has a significant impact on the DW environment.

The main problem is that there is no definite guideline for DW analysts to choose appropriate indexing methods. According to common practice, Bitmap index is best suited for columns having low cardinality and should be only considered for low-cardinality data [1], [3], [7] .
Strohm [7]concludes that the advantages of using Bitmap indexes are greatest for low cardinality columns, i.e., columns which have a small number of distinct values compared to the number of rows in the table. If the number of distinct values of a column is less than 1%, then the column is a candidate for a Bitmap index. This assumption may be correct to some extent based on previous algorithms and based on old machine processing used by the database software and hardware respectively, but, as the usage of data is exploding, this assumption may no longer be applicable.

In this paper, we demonstrate that:

(i) Bitmap index on a column with high cardinality is more efficient than a B-tree index.

(ii) The query response time in multi-dimensional queries is not pursued by the time that is needed to one-dimensional queries on both Bitmap index and B-tree index.

(iii) Query utilizing Bitmap index which is executed within a range of predicates is affected by the distribution of data, but does not have any affinity by the cardinality conditions.

The rest of the paper is organized as follows. Sections 2 presents the background studies on Bitmap index, B-tree index and cardinality concepts. Section 3 defines a case study and performance methodology with a set of queries to compare the performances of Bitmap index and B-tree index. Section 4 discusses the experimental results followed by the conclusion in section 5.

TABLE I
BASIC BITMAP INDEX ADOPTED BY[10]

RowId	C	B0	B1	B2	B3
0	2	0	0	1	0
1	1	0	1	0	0
2	3	0	0	0	1
3	0	1	0	0	0
4	3	0	0	0	1
5	1	0	1	0	0
6	0	1	0	0	0
7	0	1	0	0	0
8	2	0	0	1	0

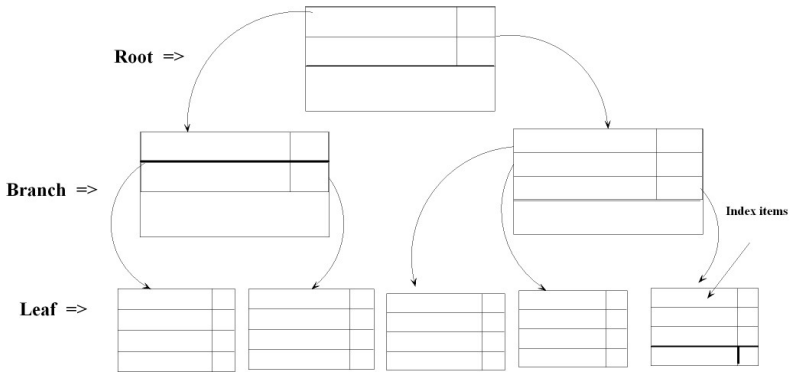


Fig. 1. B-Tree Index structure

II. BACKGROUND AND RELATED WORKS

A. Bitmap index

Bitmap index is built to enhance the performance on various query types including range, aggregation and join queries. It is used to index the values of a single column in a table. Bitmap index is derived from a sequence of the key values which depict the number of distinct values of a column. Each row in Bitmap index is sequentially numbered starting from integer 0. If the bit is set to "1", it indicates that the row with the corresponding RowId contains the key value; otherwise the bit is set to "0".

To illustrate how Bitmap indexes work, we show an example which is based on the example illustrated by E.E-O'Neil and P.P-O'Neil [12] . " *Table I shows a basic Bitmap index on a table containing 9 rows, where Bitmap index is to be created on column* C *with integer values ranging from 0 to 3. We say that the column cardinality of* C *is 4 because it has 4 distinct values. Bitmap index for* C *contains 4 bitmaps, shown as* B0*,* B1*,* B2 *and* B3 *corresponding to the value represented. For instance, in the first row where* RowId =0*, column* C *has the value 2. Hence, in column B2, the bit is set to* "1"*, while the rest of bitmaps bits are set to* "0" *. Similarly, for the second row, bit of* B1 *is* "1" *because the second row of C has the value 1, while the corresponding bits of* B0*,* B2 *and* B3 *are all* "0" *. This process repeats for the rest of the rows [12].*"

B. B-tree index

B-tree [6] stores the index pointers and values to other index nodes by using a recursive tree structure. The data could be easily retrieved by tracing on the pointer. The top-most level of the index is known as root while the lowest level is known as the leaf node. All the other levels in between are called branches (Internal nodes). Both the root and branches contain entries that point to the next level in the index. Leaf nodes consist of the index key and pointers pointing to the physical location in which the corresponding records are stored. For more information we provides general information about the structure of B-tree index and its pages.

A B-tree structure is used by the database server to set up index information. Fig 1 guides that a B-tree index is arranged by the following three types of index nodes:

1) Root Node:
   It includes node pointers to its down branch nodes.

2) Branch Nodes:
   A branch node includes pointers to leaf nodes or the other branch nodes.

3) Leaf Nodes:
   It includes horizontal pointers and index items to the other leaf nodes.

Index Items: The essential piece of an index is called Index Item. It includes a key value that depicts the value of the indexed column for a special row and also contains RowId that the database uses to locate the row in a datapage.

Nodes: It is an index page that group of index items stores in it for the three kind of nodes.

According to some research studies [3], [13], B-tree index has features that make it a well selection criterion on columns with high cardinality values especially in DW's designing.

C. Cardinality

Definition of cardinality in set theory refers to the number of members in the set. On database theory, the cardinality of a table refers to the number of rows contained in a particular table. In terms of OLAP system, cardinality refers to the number of rows in a table. On the other hand, on a data warehousing point of view, cardinality usually refers to the number of distinct values in a column. Generally, there are four levels of cardinalities (as following items); Low, Normal, High and Very high cardinality (also known as Full Cardinality).

Low-cardinality refers to columns which have a very few unique values. Low-cardinality column values are typically Boolean values such as gender or a check-box. For instance, the Product table with a column named Active-Bt is a column with low-cardinality. This column contains only 2 distinct values: 1 or 0, denoting whether the product is available. Because there are just 2 possible values in this column, its

cardinality level would be called as low-cardinality.

Normal-cardinality refers to columns which have sporadic unique values. Examples of such columns with normal-cardinality are addresses or product types. For instance, column named Name-Bit in Order table contains the name of the customers. There may be some customers with the general name, such as John, while others have dissimilar names. While there are many possible values in this column, its cardinality level would be called as normal-cardinality.

High-cardinality is related to columns which has a large number of distinct values containing very unique values. From the DW point of view, since the grouping of characteristics that are not related in one dimension; high cardinality can be called the number of unique combination of values in a dimension which are very high.

Full-cardinality is related to columns which has a very large number of distinct values. Full cardinality values are generally like identification number or e-mail addresses. As an example, in the USER table, an auto-generated number is assigned to each user to uniquely identify them. Recently, full-cardinality is also known as Very High Cardinality in the database community.

D. Related Works

Recently, there are some significant research studies investigating the main limitation of Bitmap index. New indexing strategy applied to bitmap compression schemes requires less space and provides performance gains [12], [14], [17], [20]–[22].

In [19], [20], they have been shown that WAH compression is effective in reducing Bitmap index size. They show that query processing time grows linearly as the index size increases. Besides, they also demonstrate that the query processing time is linear in the number of hits when using a WAH compressed bitmap index. They prove that WAH compressed bitmap indexes/indices are optimal for both low cardinality and high cardinality and that the techniques for compressing bitmap index increase efficiency of in-memory logical operations.

In [14], they investigate some recent developments in bitmap indexing technology under three categories, i.e., encoding, compression, and binning. They discuss how various encoding methods could reduce the index size and improve the query response time. On the other hand, though, several methods of indexing, including B*-tree and B+-tree (extensions of B-tree) are theoretically best suited for single dimensional range queries, but most of them cannot be used to efficiently answer arbitrary multi-dimensional range queries.

In [21], we see the FastBit is a compressed Bitmap index which is implemented with a particular compression schema(is this scheme or schema?). This indexing scheme can answer range queries many times faster than the well-known indexing schemes.

In [22], they claim that FastBit is efficient in both terms of speed and compression amongst data management techniques.

In [12] they show an efficient bitmap index design on modern processors by analyzing the RIDBit and Fast-Bit with the physical design aspects of the two packages. They show that the FastBit indexes are usually larger than RIDBit indexes, but it can answer many queries in less time because it accesses the needed bitmaps in less I/O operations. In fact, the optimizer of database software cannot make use of any indexes to execute some kind of queries. Rather these databases will prefer to do a full table scan. Since there is an abnormal growth of data, table scan will be needed to increase physical disk reads to avoid insufficient memory allocation. Therefore, FastBit can support these queries directly [17], [21], [22] where Oracle 11G does not utilize this method of implementation.

III. Methodology

A. Query Set

In order to compare efficiency of Bitmap index and B-tree index we build a series of queries on some columns for evaluation. In our dataset, there are 3 tables namely Order, Sales and Product. Table II depicts these tables with their column cardinalities indicated. Each table has approximately 1.6 billion records. These records are generated randomly using PL/SQL Block by Oracle11G tools. The Sales table involves low-cardinality columns, while the Order and Product tables involve normal and high cardinality columns respectively. All tables have the *Id-Bit* and *Name-Bit* columns while the *Active-Bit* column only presented in the Product table involving Bitmap index with low cardinality. Likewise, the *Id-Bt* and *Name-Bt* are present in all the tables. However, the *Active-Bt* column in the Product table involves B-tree index with low cardinality. We use a number of queries to study performance of B-tree and Bitmap indexes. In each column, *C1k* has *1000* distinct values appearing randomly on approximately 1,600,000 times each, *C1M* has 1,000,000 distinct values and *C120M* has 120,000,000 distinct values. The columns *Id-Bit* and *Name-Bit* indicate Bitmap index and *Id-Bt* and *Name-Bt* indicate B-tree index in all tables.

TABLE II
VARIOUS COLUMNS WITH THEIR ASSOCIATED DATA TYPES AND COLUMN CARDINALITIES

	Id-Bit Numeric 8 Byte	**Id-Bt** Numeric 8 Byte	**Name-Bit** Varchar 8 Byte	**Name-Bt** Varchar 8 Byte	**Active-Bt** Number 1Byte	**Active-Bit** Number 1Byte
Sales	C1K	C1K	C1K	C1K		
Order	C1M	C1M	C1M	C1M		
Product	C120M	C120M	C120M	C120M	2	2

The Set Query Benchmark has been used for frequent-query application as Star-Schema within data-warehouse design [25], [26] . The Queries of the Set Query Benchmark have been designed on business analysis missions.In order

to evaluate the time required to answer different query types including range, aggregation and join queries; we implemented the six queries adopted by the Set Query Benchmark. Briefly, we describe all of our selected SQL queries used for our performance measurements as indicated in Listing 1 to 6.

Query1A: SELECT count (*) FROM table WHERE ColumnX = 10;
ColumnX is one of Id-Bit and Id-Bt and table is one of Sales with C1k cardinality on its columns, Order with C1M, and Product with C120M cardinality respectively.
Query1B: SELECT count (*) FROM table WHERE ColumnY = 'ABCDEFGH';
ColumnY is one of Name-Bit and Name-Bt. According to E. O'Neil and P. O'Neil [12], since they involve only one column at a time in the WHERE clause, we call Query1 as one-dimensional (1-D) query. There are 12 different instances of Query1B.

Listing 1: Description for Query 1

Query2A: SELECT count (*) FROM tables WHERE ColumnX in (100000, 100000000);
ColumnX is one of Id-Bit and Id-Bt.
Query2B0: SELECT count (*) FROM tables WHERE Id-Bit= 1000 and NOT ID-Bit = 1000000. Query2B1: SELECT count (*) FROM tables WHERE Id-Bt = 1000 and NOT Id-Bt = 1000000;
Query2B0 and Query2B1 are two-dimensional queries where each WHERE clause involves conditions on two columns. There are 12 different instances of Query2B.

Listing 2: Description for Query 2

Query3A: SELECT sum (ColumnM) FROM tables WHERE ColumnN between 100000 and 100000000.
ColumnM, ColumnN is one of Id-Bit and Id-Bt and and M=N= Id-bit or M=N= Id-bt. There is 6 instances of Query3A.
Query3B: SELECT Sum (ColumnM) FROM tables WHERE (ColumnN between 100000 and 1000000 or ColumnN between 1000000 and 10000000 or ColumnN between 10000000 and 30000000 or ColumnN between 30000000 and 60000000 or ColumnN between 60000000 and 100000000);
CoulmnM, ColumnN is one of Id-bit and Id-bt and M=N= Id-bit or M=N= Id-bt . There are 6 instances of Query3B.

Listing 3: Description for Query 3

Query4A: SELECT * FROM tables WHERE columnX is in (1000, 100000, 1000000, 100000000, 1000000000).
ColumnY is one of Id-Bit and Id-Bt. There is 8 instance of Query4A. In the Product table we have 2 other columns, namely Active-Bit and Active-Bt with 2 cardinalities. The Active-Bit is concern with Bitmap index and the Active-Bt is concern with B-tree index in the same table.
Query4B: SELECT * FROM Product WHERE ColumnX is in (1000, 100000, 1000000, 100000000, 1000000000) and Active-bit = 1
ColumnZ is one of Id-Bit and Id-Bt. There are 2 instances of Query4B.

Listing 4: Description for Query 4

Query5A: SELECT Id-Bit, Name-bit, count (*) from tables GROUP BY Id-Bit,Name-bit.
Query5B: SELECT Id-Bt, Name-bt, count (*) from tables GROUP BY Id-Bt, Name-bt;
tables is one of the three existent Tables. There are 8 instances of Query5A and Query4B.
Query5C: SELECT sum (ColumnM) FROM tables WHERE ColumnN $>$ 9000 and ColumnN $<$ 9100
ColumnM, ColumnN is one of Id-Bit and Id-Bt and and M=N= Id-bit or M=N= Id-bt. There is 6 instances of Query5C.

Listing 5: Description for Query 5

Query6: SELECT sum(D.ColumnM) FROM sale E, tables D WHERE E.ColumnN= D. CoulmnP Group by columnM;
Here ColumnM, ColumnN and ColumnP is one of Id-Bit and Id-Bt that M=N=P=Id-Bit or M=N=P= Id-Bt and tables is one of the three existent tables except the Sales table. There are 6 instances of Query6.

Listing 6: Description for Query 6

B. Experimental Setup

We performed our tests on the Microsoft Windows Server 2003 machine with Oracle11G database systems. Table III shows some basic information about the test machines and the disk system. To make sure the full disk access time is accounted for we disabled all unnecessary services in the system and kept the same condition for each query. To avoid

inaccuracy, all queries were run 4 consecutive times to give an average elapsed time.

TABLE III
INFORMATION ABOUT THE TEST SYSTEM

CPU	Pentium 4 (2.6 GHZ)
Disk	7200 RPM, 500 GB
Memory	1 GB
Database	Oracle11G

IV. RESULTS AND DISCUSSIONS

We present the performance measurement experiments in two main parts, namely, (i) the index file size and index construction time and (ii) query retrieval time.

A. Index File Size and Index Construction Time

The time taken to construct B-tree and Bitmap indexes is shown in Table IV. We see that the Bitmap requires slightly more time to build high-cardinality columns (Product table) as compared low-cardinality (Sales table) on the same columns. B-tree, on the other hand, requires considerably more time to build all indexes regardless of the columns' cardinalities. Table IV summarizes the indexes size over various kinds of data cardinality. In Fig 2, we consider only the size of the two columns on Bitmap and B-tree indexes. For high-cardinality cases, Bitmap generates a large number of small bitmap objects and spends much time in allocating memory of these bitmaps. Since the index file size of Bitmap index depends on the cardinality of the column; ultimately, the index size on the columns will be smaller than a B-tree even for full cardinality (100% distinct values) on the same column.

TABLE IV
INDEX FILES SIZE AND INDEX CONSTRUCTION TIME

	Sales		Order		Product	
	Size(MB)	Time(S)	Size(MB)	Time(S)	Size(MB)	Time(S)
ID-Bit	326	1580	1222	2805	3012	3534
Id-Bt	26211	21090	26532	21319	26568	21580
Name-Bit	418	1673	1341	2605	3215	3892
Name-Bt	26911	21638	26821	21430	27190	21802
Active-Bit					288	1544
Active-Bt					0.06	4678

Previous research [18] shows that the index file size of a Bitmap index on column which would be a candidate for primary key will be much larger than a B-tree index on the same column. In contrast, according to our test results, the index file size of a Bitmap index on the above-mentioned column will not be larger than a B-tree index. Similarly, in terms of index construction time, Bitmap index outperforms B-tree significantly.

Table IV and Fig 2 show that to build index on a large column which is involved by B-tree is prohibitively expensive in terms of space and creation time. In other words, the index file size of column which is involved by Bitmap index is significantly smaller than the same column which is involved by B-tree index.

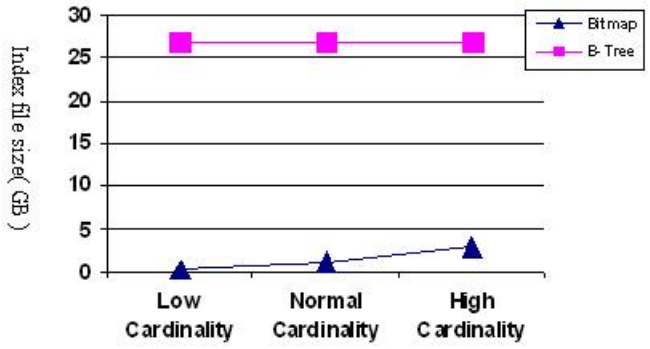


Fig. 2. Index file size of bitmap with various cardinality

B. Query Response Time

In this section, we evaluate the time required to answer the queries. These timing measurements directly reflect the performance of indexing methods. A summary of all the timing measurements on several kinds of queries, as indicated in Listing 1 to 6, are shown in Table V.

Now, we examine the performance on count queries (Query1 and Query2) in detail. In Query1, when the cardinality of the column is high, it takes slightly more time to execute the queries. In all the tables with cardinalities of 1K, 1M and 120M, the average time used by Bitmap index to read in the index blocks is nearly 0.021 s (21 ms). However, in most cases, the average time used by B-tree index is more than 52 ms. Hence, we show that Bitmap index could be best suited for one-dimensional count queries.

In Query2A and Query2B (which are two-dimensional queries and involve two conditions clause of the same structure as Query1), generally, we expect the response time of both indexes to be about twice as long as that of Query1. However, it seems that estimate is not accurate for Bitmap index. Therefore, the time in multi-dimensional queries is not pursued by the time that is needed to one-dimensional queries. On the other hand, B-tree index has a much more growth in the response time (90 ms) as well. We also observed that the time used by Bitmap index is slightly less than the time used by B-tree index.

Next, we focus on Query3. The query response time is different from that of Query1 and Query2. Overall, we see that the time required by both indexes has risen significantly. Since the Bitmap and B-tree indexes use different mechanisms to organize for the table data, the time to resolve the conditions on Query3 will conclude the total query response time. The number of records by these queries that has to be selected is uniformly scattered among rows 100,000 and 100,000,000. Consequently, the elapsed time of both indexes that is needed

TABLE V
QUERY RESPONSE TIME PER SECONDS

	Sales (Low Cardinality)		Order (Normal Cardinality)		Product (High Cardinality)	
	Bitmap	**B-tree**	**Bitmap**	**B-tree**	**Bitmap**	**B-tree**
Query1A	0.018	0.051	0.019	0.053	0.020	0.052
Query1B	0.023	0.056	0.023	0.055	0.024	0.057
Query2A	0.017	0.078	0.017	0.075	0.023	0.076
Query2B	0.021	0.097	0.024	0.101	0.022	0.090
Query3A	21.21	113.52	22.12	112.39	21.20	115.68
Query3B	307.61	1230	308.56	1246.2	308.54	1243.9
Query4A	0.081	0.140	0.081	0.138	0.097	0.151
Query4B					0.044	0.110
Query5C	1.15	5.21	0.92	5.20	0.92	5.23
Query5A ,B	1560.6	**1554.3**	1730.21	**1701.52**	1846.98	**1840.03**
Query6			1108.87	1400.3	1113.39	1440.12

to answer the queries which are executed within a range of predicates is affected by the distribution of data and does not follow the cardinality conditions.

The response time required to retrieve the data for Query4 has a similar trend to that for Query3 with just one difference. The difference stems from the column under the second condition which has extremely low cardinality. Here, with a Bitmap index on the Active-Bit column (Cardinality = 2) in place, we created another Bitmap index on the Id-Bit column containing equal values between 1000 and 1000000000 and then executed Query4A. Subsequently, the Query4B will be re-executed with B-tree indexes on the same conditions. In the previous version of Oracle database software; the Oracle optimizer will choose a full table scan and rather make it use index for B-tree [18] .
Even though the query response time demonstrates that B-tree index takes about twice as much time as Bitmap index, in contrast, we have not observed the mentioned trend during execution tracing in our test system. Thus, we can conclude that with Bitmap indexes, the optimizer of Oracle11G answers to these queries, which are involved with AND, OR and so on is as fast as B-tree index.

Another query that can be a main way to exercise the indexing performance of Bitmap and B-tree is Query5. In Query5A and Query5B, we see that the response time of B-tree is slightly less than that of Bitmap index. On the other hand, the required time to answer these queries is extremely more than that of others. That is because to execute this type of queries, the optimizer will not make use of any indexes. Rather, it will prefer to do a full table scan. Since there is an abnormal growth of data, table scan will be needed to increase physical disk reads to avoid insufficient memory allocation. So this does not scale very well as data volumes increase. Even though there is a certain implementation of Bitmap indexes (FastBit) which can support these queries directly [17], [21], [22], Oracle 11G does not utilize this method of implementation. The required time to answer Query5C that involves Bitmap index is slightly unusual. The time is decreased for a column with high cardinality compared to columns with low cardinality.

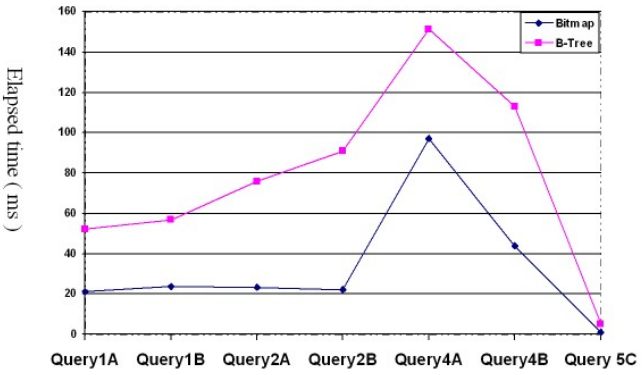


Fig. 3. Query elapse time for Bitmap and B-tree index on high cardinality

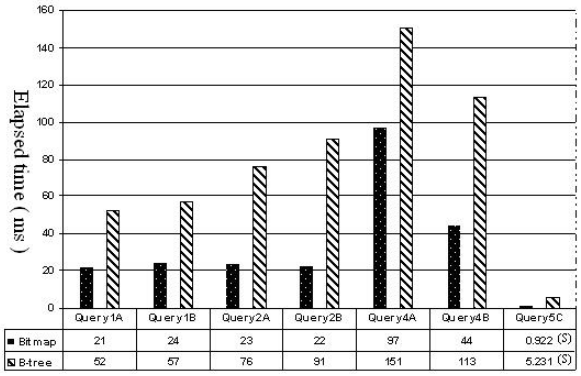


Fig. 4. Query elapse time for Bitmap and B-tree index on high cardinality

Since more general join queries are often submitted interactively, reducing their response time is a critical issue in the DW environment [14], [15] . Thus, the ability to answer Query6 has a strong impact on the query processing performance. Even though, Oracle 11G [7] has implemented the Join Bitmap index to join columns, this is not always possible for ad hoc query, therefore it is strongly necessary to know which indexes are best suited. Nevertheless, we see that the elapsed time of this type of query which is involved in join operations is much faster than that of B-tree index in the case of either high cardinality or low cardinality.

In summary, Fig 3 and Fig 4 shows the query elapse time for the Product table (table with high cardinality). This figure shows that Bitmap index is much faster than B-tree index. Thus, it can be claimed that Bitmap index is adequate for all levels of column cardinality as shown in Fig 5 and Fig 6 where the query elapse time is about constant for each query type.

V. CONCLUSIONS

It is commonly accepted that Bitmap index is more efficient for low cardinality attributes. Our experiment shows that Bitmap index effectively reduces the query response time for

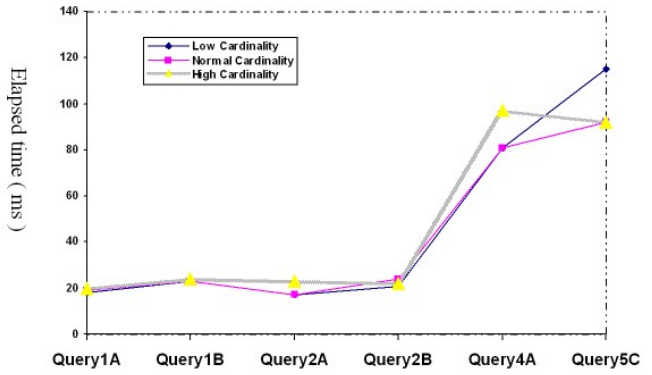
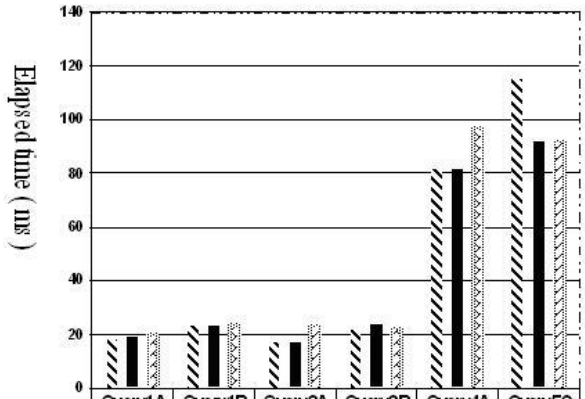


Fig. 5. Query elapse time for Bitmap index on various level of column cardinality



	Query1A	Query1B	Query2A	Query2B	Query4A	Query5C
Low Cardinality	18	23	17	21	81	115
Normal Cardinality	19	23	17	24	81	92
High Cardinality	20	24	23	22	97	92

Fig. 6. Query elapse time for Bitmap index on various level of column cardinality

a column with high cardinality compared B-tree index. We have also shown that Bitmap index file size and index creation time grow gradually as the column cardinality increases as compared to B-tree which grows significantly. In addition, we have demonstrated that although the index file size of bitmap index is affected by column cardinality; the query processing time is constant as the column cardinality increases. Besides, Bitmap index is also efficient for other types of queries, such as joins on keys, multidimensional range queries and computations of aggregates. Thus, we conclude that Bitmap index is the conclusive choice for a DW designing no matter for columns with high or low cardinality.

It is often considered that I/O cost dominates the query response time. Moreover, main memory size may play a role in index performance as small memory size might trigger a lot of paging activities, which then could change the query performance of Both indexing. Thus, our future work includes the evaluation of I/O costs on an upgraded hardware system.

ACKNOWLEDGMENT

We would like to express our deep and sincere gratitude to Professor Ralph Kimball, Professor Bill Inmon, Professor Patrick O'Neil and to all those who have guide us by their inestimable knowledge and logical way of thinking. Their books and papers have been of great value for us.

REFERENCES

[1] S. Chaudhuri, U. Dayal,*An Overview of Data Warehousing and OLAP Technology.*, ACM SIGMOD RECORD. 1997

[2] P. O'Neil, *M*odel 204 Architecture and Performance. In Proceedings of the 2nd international Workshop on High Performance Transaction Systems, Lecture Notes In Computer Science, vol. 359. Springer-Verlag, London, (September 28 - 30, 1987),pp.40-59

[3] R. Kimball, L. Reeves, M. Ross, *T*he Data Warehouse Toolkit. John Wiley Sons, NEW YORK, 2nd edition, 2002

[4] W. Inmon, *Building the Data Warehouse.*, John Wiley Sons, fourth edition, 2005

[5] C. DELLAQUILA and E. LEFONS and F. TANGORRA, *D*esign and Implementation of a National Data Warehouse. Proceedings of the 5th WSEAS Int. Conf. on Artificial Intelligence, Knowledge Engineering and Data Bases, Madrid, Spain, February 15-17, 2006 pp. 342-347

[6] D. Comer,*Ubiquitous b-tree*, ACM Comput. Surv. 11, 2, 1979, pp. 121-13

[7] R. Strohm, *Oracle Database Concepts 11g*,Oracle, Redwood City,CA 94065, 2007

[8] P. O'Neil and D. Quass, *Improved query performance with variant indexes*, In SIGMOD: Proceedings of the 1997 ACM SIGMOD international conference on Management of data.1997

[9] C. Dell aquila and E. Lefons and F. Tangorra, *A*nalytic Use of Bitmap Indices. Proceedings of the 6th WSEAS International Conference on Artificial Intelligence, Knowledge Engineering and Data Bases, Corfu Island, Greece, February 16-19, 2007 pp. 159

[10] P. O'Neil and G. Graefe, *Multi-table joins through bitmapped join indices,* ACM SIGMOD Record 24 number 3, Sep 1995 , pp. 8-11.

[11] K. Wu and P. Yu, *Range-based bitmap indexing for high cardinality attributes with skew*, In COMPSAC 98: Proceedings of the 22nd International Computer Software and Applications Conference. IEEE Computer Society, Washington, DC, USA, 1998, pp. 61-67.

[12] E. E-O'Neil and P. P-O'Neil, *Bitmap index design choices and their performance impli-cations*, Database Engineering and Applications Symposium. IDEAS 2007. 11th International, pp. 72-84.

[13] C. Imho and N. Galemmo and J. Geiger, *Mastering Data Warehouse Design : Relational and Dimensional Techniques*, John Wiley and Sons, NEW YORK.2003

[14] K. Stockinger and K. Wu, *Bitmap indices for data warehouses*, In Data Warehouses and OLAP ,IRM Press,2007, Chapter 7.

[15] A. Mitea, *A* multiple join index for data warehouses. Proceedings of the 5th WSEAS Int. Conf. on DATA NETWORKS, COMMUNICATIONS and COMPUTERS, Bucharest, Romania, October 16-17, 2006 pp. 7

[16] J. Lewis,*Oracle index management secrets*, BMC Software (http://www.dbazine.com), 2006, pp. 37-47.

[17] K. Stockinger and E. Bethel and S. Campbell and E. Dart and K. Wu,*Detecting distributed scans using high-performance query-driven visualization,* In SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, 2006

[18] V. Sharma, *Bitmap index vs. b-tree index: Which and when*, http://www.oracle.com.2006

[19] K. Wu and E. Otoo and A.Shoshani,*O*ptimizing bitmap indices with efficient compression., ACM Trans. Database Syst. 31, 1 (Mar. 2006), pp. 1-38.DOI=http://doi.acm.org/10.1145/1132863.1132864

[20] K. Stockinger and K. Wu and A. Shoshani,*A performance comparison of bitmap indexes*, In CIKM 01: Proceedings of the tenth international conference on Information and knowledge management,2001

[21] K. wu, *An Efficient Compressed Bitmap Index Technology*, Http://sdm.lbl.gov/fastbit/, 2008.

[22] L. Gosink and J. Anderson and W. Bethel and K. Joy, *Variable Interactions in Query Driven Visualization*, The Visualization and Graphics Research Group of the Institute for Data Analysis and Visualization (IDAV), 2007

[23] L. Gosink and J. Anderson and W. Bethel and K. Joy, *Bin-Hash Indexing: A Parallel GPU-Based Method For Fast Query Processing*, The Visualization and Graphics Research Group of the Institute for Data Analysis and Visualization (IDAV), 2007

[24] K. Stockinger and K. Wu and A. Shoshani,*Strategies for processing ad hoc queries on large data warehouses*, In Proceedings of the 5th ACM international Workshop on Data Warehousing and OLAP (McLean, Virginia, USA, November 08 - 08, 2002). DOLAP '02. ACM, New York, NY,2002, pp. 72-79. DOI= http://doi.acm.org/10.1145/583890.583901

[25] P. O'Neil, *The Set Query Benchmark. In The Benchmark Handbook For Database and Transaction Processing Benchmarks*, Jim Gray, Editor, Morgan Kaufmann, 1993.

[26] P. ONeil and E. ONeil, *Database Principles, Programming, and Performance*, 2nd Ed. Morgan Kaufmann Publishers. 2001.

Morteza Zaker is a research student for the faculty of Information Technology, Multimedia University. His research interests are ERP, Advanced Databases and Data Warehouse architecture. He is a system analyst during the last decade.

Somnuk Phon-Amnuaisuk received his B.Eng. from King Mongkut Institute of Technology (Thailand) and Ph.D. in Artificial Intelligence from the University of Edinburgh (Scotland). He is currently an associate Dean for the faculty of Information Technology, Multimedia University, Malaysia where he also leads the Music Informatics Research group. His current research works span over multimedia information retrieval, polyphonic music transcription, algorithmic composition, Bayesian networks, data mining and machine learning.

Dr. Su-Cheng Haw's research interests are in XML Databases and instance storage, Query processing and optimization, Data Modeling and Design, Data Management, Data Semantic, Constraints Dependencies, Data Warehouse, E-Commerce and Web services