ISSN No. 0976-5697

Volume 3, No. 3, May-June 2012

International Journal of Advanced Research in Computer Science

REVIEW ARTICLE

Available Online at www.ijarcs.info

Using Index Table to Query Optimization: B-Tree Index, Bitmap Index and Function-Based Index

Chanchai Supaartagorn
Department of Mathematics Statistics and Computer
Faculty of Science, Ubon Ratchathani University
Ubonratchathani, Thailand
scchansu@mail2.ubu.ac.th

Abstract: Query optimization is the component of a database management system that attempts to determine the most efficient way to execute a query. One of the techniques to optimize the query is to use an indexing table. This article proposes the concept of a B-Tree index, a Bitmap index and a Function-based index. Our comparative study of 3 techniques show the results are as follows: the B-Tree index is efficient for high-cardinality attributes. This index is not suitable for complex queries. The Bitmap index is only efficient for low-cardinality attributes. It is useful in processing complex queries. This index improves complex query performance by applying a Boolean operator. The Function-based index allows the creation of index on expression and internal function.

Keywords: Query optimization; Indexing table; B-Tree index; Bitmap index; Function-based index

I. INTRODUCTION

A database is a set of tables containing data where there are connections between the data stored in one table and data stored in other tables. Nowadays, a database plays an important role for every organization. It is a tool for decision making and planning to achieve its objectives or goals.

With the rapid expansion of the internet, data is growing at an exponential rate. Businesses are building larger databases to cope with this enormous data growth rate [1]. In addition, over the past decade, two clear trends have occurred: a) the database systems have been deployed in new areas, such as electronic commerce, with a new set of database requirements, and b) the databases have become increasingly complex with support for very large numbers of concurrent users [2]. The large amount of data, more complexity and answers are expected quickly are the critical problems when accessing a database. Modern day database workloads include: On-line Transaction Processing Systems, Enterprise Resource Planning, Customer Relationship Management, On-line Analytical Processing and Data Analysis over Data-warehouses. The queries generated by these workloads are increasingly complex and the databases are larger than ever [3]. There are many techniques to speed up query processing or to improve the query optimization, for example, index table, SQL tuning, database server tuning, etc. This article explores the index table technique; reviews the B-tree index, Bitmap index and Function-based index.

II. INDEX TABLE CONCEPTS

Index table is a data structure that improves the speed of data retrieval operations on a database table. The index is stored in a file that contains the key and the address to access the record directly. Comparable to the book, the index is the index page to collect a glossary of the book by alphabetic order. Readers can search data easily and quickly. However, there are both in time and in space to store the index table.

A. B-Tree Index:

Indexes are stored on disk in the form of a data structure known as B-tree. A B-tree order m is a tree with the following structural properties [4]:

a. The root is either a leaf or has between 2 and m children.
b. All nonleaf nodes (except the root) have between m/2 and m children.
c. All leaves are at the same depth.

B-tree follows the same structure as a binary search tree, in that each key in a node has all key values less than the key as its left children, and all key values more than the key as its right children [5]. In addition, leaf nodes store the ROWID values that represent the address of data. The B-tree in Figure 1 is an example of B-tree index.

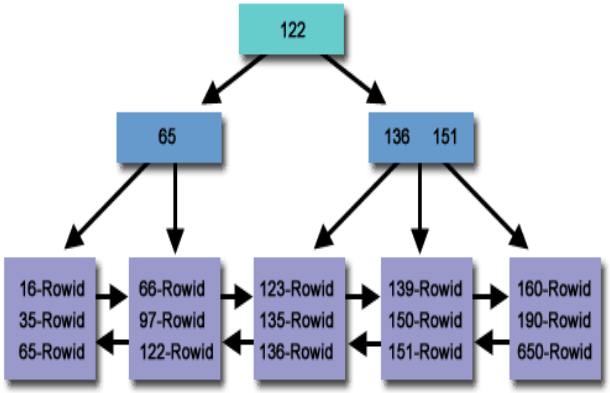

Figure 1. B-tree index [6].

If we need to search the value 66, the process starts from the root node. The value 66 is less than 122, then down to the tree on the left. The value 66 is greater than 65, then down to the tree on the right that it is the leaf node. This leaf node stores the ROWID of the value 66, then it retrieves the record of value 66.

B. Bitmap Index:

A bitmap index on an indexed attribute consists of one vector of bits per attribute value, where the size of each

bitmap is equal to the cardinality of the indexed relation. The bitmaps are encoded such that, the i^{th} record has a value of v in the indexed attribute if and only if the i^{th} bit in the bitmap associated with the attribute value v is set to 1, and the i^{th} bit in each of the other bitmaps is set to 0. This is called a Value-List index [7]. An example of a Value-List index for table T is shown in Figure 2, where each column in Figure 2(b) represents a bitmap S^v associated with an attribute value v.

RID	...	X	...
1		B	
2		C	
3		A	
4		H	
5		A	
6		G	
7		D	
8		D	
9		F	
.		.	
.		.	
.		.	
10000		E	

(a) Table T

S^A	S^B	S^C	S^D	S^E	S^F	S^G	S^H
0	1	0	0	0	0	0	0
0	0	1	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	0	0	1	0	0
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
0	0	0	0	1	0	0	0

(b) Value-List Index

Figure 2. Example of a Value-List Index.

C. *Function-based Index:*

Function-based index allows the creation of index on expression and internal function. Traditionally, performing a function on an indexed column in the Where clause of a query would not be used. It performs the full-table scan. There are many DBMS that can use function-based index, for example, Oracle 8i or higher can use function-based index to counter this problem. It is assumed that the following index is built in a customer table:

Create index customer_idxl on customer(cust_name);

and the following SQL statement is executed:

*Select * From customer Where cust_name = 'John';*

Since the execution plan of this statement is established as the range scan of the index customer_idxl, its result set is searched via the index. However, if another SQL statement is executed as follows:

*Select * From customer Where trim(cust_name) = 'John';*

This statement cannot take advantage of the index customer_idxl. The inefficient execution plan of the second SQL statement is optimized by the following function-based index:

Create index customer_fbil on customer(trim(cust_name));

The result set of this SQL statement is searched via the index customer_fbil [8]. By using function-based indexes, a database designer can create a matching index that exactly matches the predicate within the SQL Where clause. This ensures that the query is retrieved with a minimal amount of disk I/O and the fastest possible speed [6].

III. PERFORMANCE EXPERIMENTS

The experiments are run on a PC AMD Athlon(tm) II x2 270 Processor 3.40 GHZ and 2 GB RAM, running Microsoft Windows 7 Service Pack 1. Oracle Database 11g as the DBMS. We create an employee table for this experiment. The schema of an employee table is defined as follows:

Create table employee (
empno number(10),
ename varchar2(20),
sal number(10),
gender varchar2(1),
primary key(empno));

A. *Compare between without index and B-tree index:*

Our synthetic dataset used on the implementation is 100,000 tuples. The SQL code is as follows:

*Select * From employee Where empno=100000;*

The empno column (Cardinality=100,000) was created as B-tree index. Figure 3 and 4 show the execution plan of without index and B-tree index respectively.

```
SQL Plus
SQL> select * from employee where empno=100000;

Elapsed: 00:00:01.40

Execution Plan
----------------------------------------------------------
Plan hash value: 2119105728

-------------------------------------------------------------------------------
| Id  | Operation          | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |          |   625 | 25000 |  3017   (1)| 00:00:37 |
|* 1  |  TABLE ACCESS FULL| EMPLOYEE |   625 | 25000 |  3017   (1)| 00:00:37 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("EMPNO"=100000)

Note
-----
   - dynamic sampling used for this statement


Statistics
----------------------------------------------------------
        351  recursive calls
          0  db block gets
      11257  consistent gets
      13605  physical reads
          0  redo size
        606  bytes sent via SQL*Net to client
        416  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          5  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

Figure 3. Execution plan of SQL code (without index)

```
SQL Plus
SQL> select * from employee where empno=100000;

Elapsed: 00:00:00.30

Execution Plan
----------------------------------------------------------
Plan hash value: 3821236522

-------------------------------------------------------------------------------
| Id  | Operation                    | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |           |     1 |    40 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEE  |     1 |    40 |     2   (0)| 00:00:01 |
|* 2  |   INDEX RANGE SCAN           | EMPNO_IDX |     1 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("EMPNO"=100000)

Note
-----
   - dynamic sampling used for this statement


Statistics
----------------------------------------------------------
        422  recursive calls
          0  db block gets
        117  consistent gets
        506  physical reads
          0  redo size
        610  bytes sent via SQL*Net to client
        416  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          6  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

Figure 4. Execution plan of SQL code (B-tree index)

From Figure 3 and 4, using B-tree index in empno column shows better performance for querying than without index are as follows:

a. Without index Disk access time = 1.40 ms, Consistent gets = 11,257 and Physical reads = 13,605
b. B-tree index Disk access time = 0.30 ms, Consistent gets = 117 and Physical reads = 506

Moreover, when implementing with 200,000, 1,000,000 and 2,000,000 tuples, disk access time is shown in Figure 5.

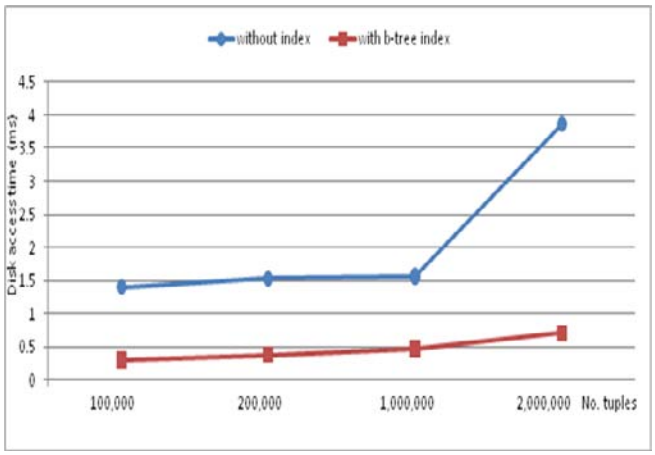

Figure 5. Performance comparison of without index and B-tree index

The results of this experiment are according to the research of Hyunja Lee and Junho Shim [9]. They propose vertical schema as a primary table structure for the triple information in RDBMS and a pivot table index created from the basic vertical table. The number of tuples of the vertical directly affects performance. The results, when implementing with 100,000 tuples, index outperforms by more than 3 times, while with 2,000,000 tuples, it outperforms by more than 40 times.

B. Compare size of index table between B-tree index and Bitmap index:

Our synthetic dataset used on the implementation is 100,000 tuples. The gender column (Cardinality=2) was created as B-tree index and Bitmap index. Figure 6 and 7 show the size of B-tree index (GENDER_IDX) and Bitmap index (GENDER_BMX) respectively.

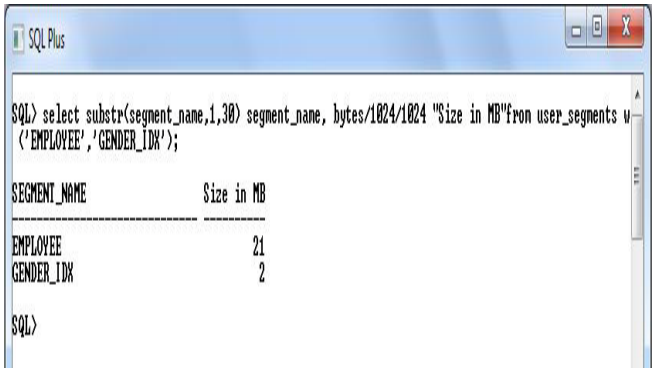

Figure 6. Size of B-tree index table (GENDER_IDX)

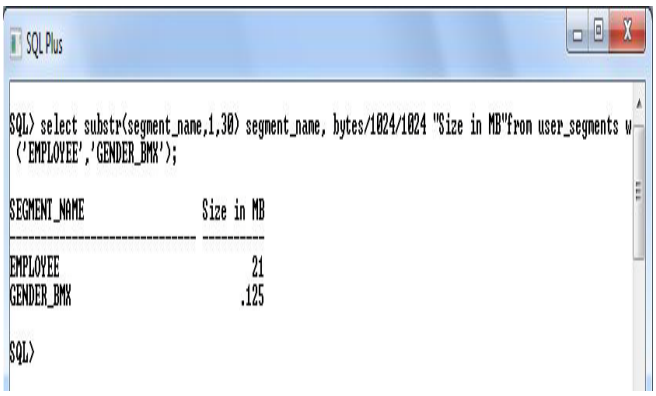

Figure 7. Size of Bitmap index table (GENDER_BMX)

From Figure 6 and 7, using Bitmap index in gender column shows better performance for querying than B-tree index in gender column. Moreover, when implementing with 200,000, 1,000,000 and 2,000,000 tuples, size of index table is shown in Figure 8.

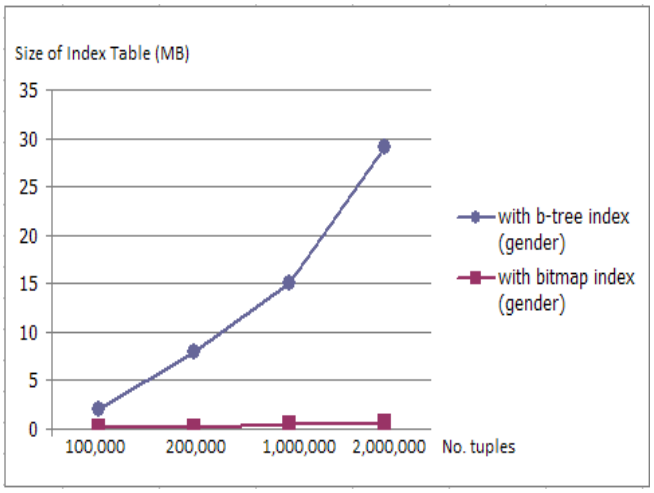

Figure 8. Size of index table of B-tree index and Bitmap index

From Figure 8, Bitmap index is efficient for low-cardinality attributes. On the other hand, B-tree index is efficient for high-cardinality attributes. The results of this experiment are according to the research of Ali Hamadou and Kehua Yang [10]. They present an efficient bitmap indexing technique based on Word-Aligned Hybrid for data warehouses. They can observe that when the cardinality of indexed attribute grows, the size of the bitmap index grows exponentially.

C. Experiment of Function-based index:

We experimented with the SQL code is as follows:

*Select * From employee Where UPPER(ename)='JOHN';*

If we create a regular index on the ename column, we see that the index is not used. The result is shown in Figure 9.

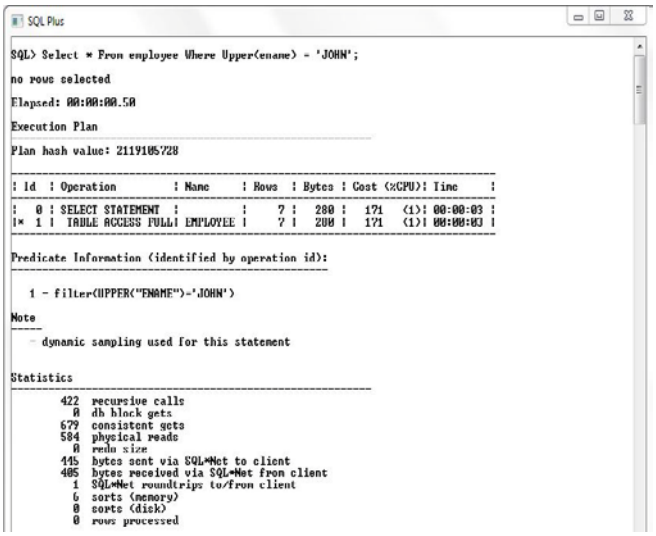

Figure 9. Execution plan of SQL code (regular index)

Then we replace the regular index with a function-based index on the ename column is as follows:

Create Index ename_idx On employee(UPPER(ename));

We see that the index is used after run the SQL code. The result is shown in Figure 10.

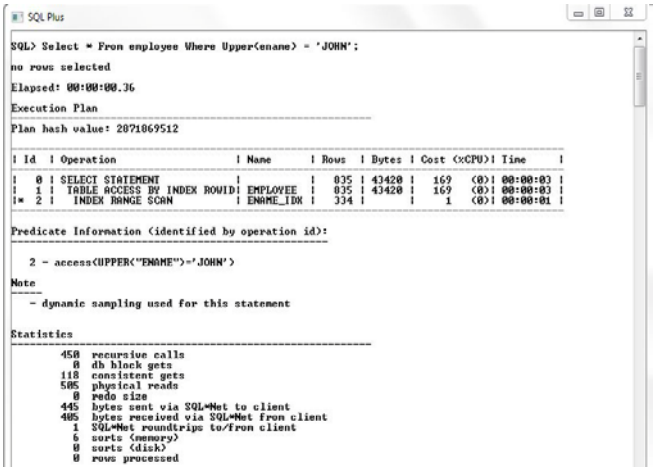

Figure 10. Execution plan of SQL code (function-based index)

From Figure 9 and 10, using Function-based index shows better performance for querying than regular index are as follows:

a. Regular index Disk access time = 0.50 ms, Consistent gets = 679 and Physical reads = 584
b. Function-based index Disk access time = 0.36 ms, Consistent gets = 118 and Physical reads = 505

IV. CONCLUSION

Index table is the technique used to speed up query processing. However, there are both in time and in space to store the index table. In this article, we propose and advise the use of B-Tree index, Bitmap index and Function-based index.

B-tree index is stored on disk in the form of a data structure known as B-tree.

a. B-tree index is efficient for high-cardinality attributes, for example, employee code attribute, student code attribute, etc.
b. B-tree index is not suitable for complex queries, for example, query with a Boolean operators, because it must create composite key.
c. B-tree index is useful in On-line Transaction Processing Systems. The system is characterized by a large number of short on-line transactions (Insert, Delete, Update).

Bitmap index consists of C (the number of distinct values of the indexed attribute) bitmap vectors each of which is created to represent each distinct value of indexed attribute [11].

a. Bitmap index is efficient for low-cardinality attributes, for example, gender attribute, department code attribute, etc.
b. Bitmap index improves complex query performance by applying low-cost Boolean operation such as AND, OR and NOT in the selection predicate on multiple indices, thereby reducing the search space before going to the primary source data [11].
c. Bitmap index is useful for various database applications such as data warehousing. The requests for information from data warehouse are usually complex and ad-hoc queries.

Function-based index can define an expression or an SQL function based on the columns of a target table as its column. Accordingly, it encourages various kinds of search conditions to be supported by their relevant indexes [8]. A database designer can create a matching index that exactly matches the predicate within the SQL Where clause.

V. REFERENCES

[1] B. Dageville, K.Dias, “Oracle’s Self-Tuning Architecture and Solutions”, IEEE Data Engineering Bulletin, vol. 29, no. 3, pp. 24-31, March 2006.

[2] Oracle Corporation, “The Self-Managing Database: Guide Application & SQL Tuning”, Oracle White Paper, 2003 [updated 2003 Nov; cited 2011 Jan], Available from: http://www.oracle.com/technetwork/database/focus-areas/manageability/twp-manage-automatic-sql-tuning-132307.pdf

[3] Surajit Chaudhuri, “Query Optimizers: Time to Rethink the Contact?”, In 35th SIGMOD International Conference on Management of Data Providence, 29 June – 2 July 2009, pp.961-968.

[4] Mark Allen Weiss, “Data Structures and Algorithm Analysis in C++”, Addison-Wesley, CA, 2007, pp.165.

[5] Ovais.tariq, “Understading B+tree Indexes and how they Impact Performance”, [Internet], 2011, [updated 18 July 2011 ; cited 2012 March], Available from: http://www.ovaistariq.net/733/understanding-btree-indexes-and-how-they-impact-performance/

[6] Burleson Consulting, “Turbocharge SQL with advanced Oracle indexing” [Internet], 2010, [updated 2010 Apr 8; cited 2012 Feb 15], Available from: http://www.dba-oracle.com/art_9i_indexing.htm

[7] C.Y. Chan and Y.E. loannidis, “Bitmap Index Design and Evaluation”, Proceeding of the 1998 ACM SIGMOD international conference on Management of data, pp. 355-366.

[8] Hyunho Lee and Wonsuk Lee, “Query Optimization for Web BBS by Analytic Function and Function-based Index in Oracle DBMS*”, In Sixth International Conference on Advanced Language Processing and Web Information Technology, 22-24 August 2007, Luoyang, Henan, China, pp.606-611.

[9] Hyunja Lee and Junho Shim, “Pivoted Table Index for Querying Product-Property-Value Information” In 3rd International Conference on Ubiquitous Information Management and Communication, 15-16 January 2009, Suwon: S.Korea, pp. 58-62.

[10] Ali Hamadou and Kehua Yang, “An Efficient Bitmap Indexing Strategy based on Word-Aligned Hybrid for Data Warehouses”, In International Conference on Computer Science and Software Engineering, 12-14 December 2008, Wuhan, Hubei, China, pp. 486-490.

[11] Janya Sainui, Sirirut Vanichayobon and Niwan Wattanakitrungroj, “Optimizing Encoded Bitmap Index using Frequent Itemsets Mining”, In International Conference on Computer and Electrical Engineering, 21-22 December 2008, Phuket, Thailand, pp. 511-515.

