Data Indexing Algorithms

Lucian Bornaz
Academy of Economic Studies, Bucharest, România

ABSTRACT

Although many ways of improving the performances of database systems exist, the most efficient one consists in implementing an effective data indexing mechanism. This must ensure the balance between the processor, memory and storage resources of the database server accordingly to the type, structure, the physical organization and the cardinality of data, the type of queries and the number of competing transactions.
This paper presents an analysis of the most used indexing algorithms (B+-Tree and Bitmap) in the nowadays DBRMS and new ways to increase their performance. Also, it is presented a new kind of index, generalized index, which implement all the identified optimizing methods.
Keywords: indexes, indexing algorithms, database optimization, b-tree, bitmap.

1.INTRODUCTION

Unavoidably, the informational society is based on large data processing information systems. The need to store, manipulate and interrogate these systems requires the implementing of high-performance database systems, capable of reacting as fast as possible to the user-submitted requests.
Although many ways of improving the performances of database systems exist, the most efficient one consists in implementing an effective data indexing mechanism. Thus, the efficient data indexing based on the type, structure, the physical organization and the cardinality of data, the type of queries and the number of competing transactions may increase the performance of the database administration systems and, implicitly, the performance of the entire information system. To be able to deliver this performance increase, the indexing algorithms must ensure the balance between the various resources of the system (storage device, memory, processor etc).

2.B+-TREE AND BITMAP INDEXES

The most used indexes in nowadays DBMRS are B+-Tree and Bitmap. These indexes have a tree-like structure that is structurally and functionally identical. Every node of the tree consists in one index block which contains several items. Every item store a value and a reference to the next lower lever block index, accordingly to the value[1]. This structure is necessary to efficiently identify the leaf blocks which contain the pointers to the records.
Trees are read from top to bottom sequentially. Each node involves a physical read from the storage device. Because the storage devices are the slowest devices of the computers, it is desired to reduce the physical reads as much as possible.
It is easy to see that number of physical reads increases with the height of the tree[2]. Reducing tree height could be done using a larger block size, a smaller data type, compression algorithms or through optimizing structure and functioning of leaf blocks.
Consequently, the analysis of the structure and functionality of the leaf blocks of the two types of indexes could show the index specific elements and performance.

Bitmap indexes

The items of the Bitmap leaf blocks consist of one value (*col0*), 2 pointers (*col1*, *col 2*) towards the first and last pages where the records are stored (*Start ROWID*, *End ROWID*) and a bitmap structure (*col3*) needed to compute the record pointers[3], consisting in a list of 1 and 0 values, one for every record stored in the pages

[1] Kavin Loney, **Oracle Database 10g: The Complete Reference**, Oracle Press, 2004;
[2] Douglas Comer, **Ubiquitous B-Tree**, ACM, 1979;
[3] Julian Dyke, **Bitmap Index Internals**, 2005;

between *Start ROWID* and *End ROWID* (figure 1).

row#0[8013] flag: -----, lock: 0
col 0; len 2; (2): c1 03
col 1; len 6; (6): 01 00 01 82 00 00
col 2; len 6; (6): 01 00 01 83 00 07
col 3; len 3; (3): 00 c2 44

Figure 1 – The structure of the leaf block item of the bitmap index[4]

The record pointers are computed using the *Value*, *Start ROWID*, *End ROWID*, *Bitmap* fields and a conversion table which store the pointers to the physical location of records (figure 2).

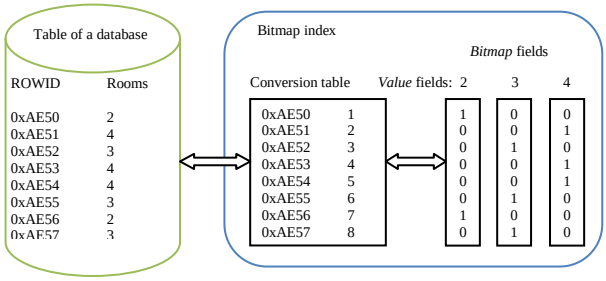

Figure 2 – Record pointers computation using a bitmap index

This structure of the leaf block enables a substantial reduction of the index height by storing several pointers in the same item but when data with high cardinality has to be handled, because of the larger size of the item, the size of the index grows making it inefficient (figure 3).

Figure 3 – Leaf blocks needed for a 100.000 rows table, an 8 kB page, relatively to the number of distinct keys[5]

Queries are very efficient due to the *Bitmap* field. Complex queries using several indexes entail very efficient binary computations. However, because the record pointers are obtained based on computations, more processor resources are needed. If we are dealing with tables that have to be frequently updated, the performance of the index is dramatically reduced[6].

Storing several pointers in one item may, especially when many transactions run simultaneously, increase the probability of delays or transaction locks. Thus, as the data cardinality decrease, the index becomes more efficient in carrying out queries but simultaneously the probability of transaction locks increases when the data is updated.

As the data cardinality increases, the probability of a transaction locks decreases as well as the need for processor resources. However, the data manipulation queries remain inefficient, mostly because of the high probability of a transaction lock and because of the significant processor resources need to update the *Bitmap* fields and the conversion table.

It is obvious that Bitmap indexes may not be efficiently used in OLTP systems where data is frequently updated. However, their height reducing mechanisms may be implemented for B+-Tree indexes keeping in mind the latter's specificity.

B+-Tree indexes

B+-Tree indexes have been designed to index high cardinality data that is frequently updated. The leaf block items store one value (*col0*) and one physical record pointer (*col1*). This solution, although increases the index size, reduces the necessary processor resources and the probability of transaction locks (figure 4).

row#0[8024] flag: -----, lock: 0
col 0; len 2; (2): c1 03
col 1; len 6; (6): 01 00 01 82 00 00

[4] Julian Dyke, **Bitmap Index Internals**, 2005;

[5] Julian Dyke, **Bitmap Index Internals**, 2005;

[6] Jonathan Lewis**, Understanding Bitmap Indexes**, www.dbazine.com;

Figure 4 – The structure of the leaf block item of the B+-tree index[7]

Because the pointers are stored within the index, there is no need of additional computations. However, complex queries involve large data types (*ROWID*), which is inefficient compared to binary operations.

Queries are very efficient, especially for high cardinality data, but become inefficient as the data cardinality decreases.

Sequential data queries are efficient due to the maintaining of pointers between neighboring leaf blocks.

The probability of a transaction lock is reduced because strictly the needed records are locked but needed resources increase with the number of locks. Therefore, reducing the number of locks could improve the database server performance.

Most modern DBRMS implement a mechanism of raising the lock level[8]. If too many resources are to be locked, the DBRMS will lock the next upper resource which contains all necessary resources.

Depending of the DBMRS type, the lock levels are row, page, extend, table, file, database. Other lock levels could be metadata, application etc.

So, for example, if too many rows are to be locked, the lock level raising mechanism could lock the entire page which contains them, or even an entire extend and so on. Thereby, the DBMRS has to manage only one lock and reduces drastically the needed memory and processor resources.

3.THE GENERALIZED INDEX

Analyzing the main data indexing algorithms (B+-Tree and Bitmap) implemented in today's DBRMS, we see that although they have a wide applicability, they do not always ensure the balance between the database server resources. Furthermore, they cannot adapt either to the evolution of the physical organization and cardinality of the data or to the changing number of simultaneously executed transactions.

Comparing the structure, functioning and performances of B+-Tree and Bitmap indexes, we can create a new indexing algorithm that implements only the performance enhancing elements of the two and that is flexible enough to allow it to adapt simultaneously with the data evolution and table usage.

Thus, the new generalized index is derived from the B+-Tree index, it has the same properties and functioning mechanisms but it implements a new type of leaf block item which is able to store several pointers towards records of the same value in an uncompressed form, to reduce the need for processor resources, as follows:

- It doesn't index the *NULL* values;
- The root node has at least 2 children nodes if it isn't leaf node in the same time;
- Every index block contains at least *m-1* items and a pointer (the index is of *m* rank);
- If an index block has *d-1* pointers, it has *d* pointers;
- The leaf blocks have the following properties:
  - They don't have pointers towards index blocks;
  - They have *n* items, the item pointers referring data blocks, for cluster indexes, of records, for non-cluster indexes;
  - Each item stores at least one pointer towards records.
- The index block size is approx. one page.

Storing more than one pointer into items could increase the probability of the transaction locks. So, in order to avoid those locks, the generalized index implements a mechanism to control the maximum number of pointers that are allowed to be stored in any one leaf block item and in any one leaf block allowing for an optimization of data indexation relatively on the number of simultaneously executed transactions. In order to create this mechanism, the index contains the following fields:

- *WReferences* controls the maximum number of pointers stored into every item and can have values between 0 and 100 with following meanings:
  - 0 allows the storage of maximum number of pointers;
  - 100 allows the storage of only one pointer for every item;
  - Any other value allows the storage of N_{ref} pointers:

[7] Julian Dyke, **Index Internals**, 2005;

[8] **Scaling Out SQL Server 2005**, Realtimepublishers, 2005;

IBM Informix Dynamic Server Performance Guide;

$$N_{ref\_\max} = \left| \frac{db\_size}{el\_size} \cdot \frac{100}{W\,\mathrm{Re}\,ferences} \right| \qquad (1)$$

WReferences>0, $N_{ref\_max} \in N$,
db_size is the index block size;
el_size is the item size of the leaf block if *col 2* stores only one pointer.

- *ElementFillRatio* controls the maximum number of pointers stored into every leaf block and can have values between 0 and 100 with following meanings:
  - 0 allows the storage of only one pointer;
  - 100 allows the storage of maximum number of pointers;
  - Any other value allows the storage of n_{ref} pointers:

$$n_{ref\_\max} = \left| \frac{ElementFillRatio}{100} \cdot N_{ref\_\max} \right| \qquad (2)$$

$n_{ref\_max} \in N$
ElementFillRatio>0

The control of the maximum number of pointers stored into the leaf block and into leaf block items allows the database administrator to balance the index according to the number of simultaneously executed transactions. As more records are stored into the leaf block items as more transaction locks could take place. However, if the number of simultaneously executed transaction is low, storing more than one record pointer into the leaf block items could increase the index performance by reducing the index height (figure 5).

Figure 5 – The size of leaf block index relatively to the number of records which hold the same value, for different kind of indexes[9]

The control of the maximum number of pointers stored into the leaf block and into leaf block items allows the control at different lock level (row and page) depending of DBRMS type.
Relatively to the B+-Tree index, the generalized index provides database systems with increased performance irrespective of the data evolution, interrogation types and the number of simultaneously executed transactions. It can adapt to the system parameter changes either though manual configuration or through the implementation of a self-regulating mechanism. The index size is reduced and the cost of data interrogation and handling is diminished.
Relatively to Bitmap indexes, the generalized index needs lower processor resources and provides the necessary control to avoid transaction locks.
Furthermore, the generalized index is a strong platform for compression algorithms implementation or other optimization methods.

4.REFERENCES

[1] Douglas Comer, **Ubiquitous B-Tree**, ACM, 1979;
[2] Julian Dyke, **Bitmap Index Internals**, 2005;
[3] Julian Dyke, **Index Internals**, 2005;
[4] Jonathan Lewis**, Understanding Bitmap Indexes**, www.dbazine.com;
[5] Kavin Loney, **Oracle Database 10g: The Complete Reference**, Oracle Press, 2004;
[6] **IBM Informix Dynamic Server Performance Guide**;
[7] **Scaling Out SQL Server 2005**, Realtimepublishers, 2005.

[9] Optimized B+-Tree index doesn't store the values for subsequently items which store pointers to the record with the same values (similarly to the IBM Informix).