Optimized OLAP Systems Integrated Data Indexing Algorithms

Lucian Bornaz, Vladucu Victor
Faculty of Cybernetics, Statistics and Economic Informatics
Academy of Economic Studies, Bucharest
lucianbor@hotmail.com, vladucuvictor@yahoo.com

ABSTRACT: *The need to process and analyze large data volumes, as well as to convey the information contained therein to decision makers naturally led to the development of OLAP systems. Similarly to SGBDs, OLAP systems must ensure optimum access to the storage environment. Thus, OLAP uses indexing algorithms for relational data and n-dimensional summarized data stored in cubes. This paper presents a new n-dimensional cube indexing algorithm which provides better performance in comparison to the already implemented Tree-like index types.*

Keywords: Data warehouse, Indexing algorithm, OLAP, n-Tree

Received: 17 January 2010, Revised 2 March 2010, Accepted 16 March 2010

1. Introduction

Data warehouses represented a natural solution towards increasing the availability of data and information, as well as their accessibility to decision makers. The warehouses store important data coming from different sources for later processing and are an integrant part of analytical processing systems (OLAP).

Unlike OLTP systems, OLAP systems must execute complex interrogations and large data volume analyses. To optimize, analytical processing systems analyze data and store aggregated information in special analytic structures, called cubes.

Similarly to OLTP systems, OLAP systems use indexing algorithms to optimize access to data stored in data warehouses, i.e. cubes.

2. General information about cubes

When stored in an OLAP system, the source data may be indexed to reduce the time necessary for their processing. To index source data, OLAP systems use indexing algorithms similar to OLTP (B-Tree, Bitmap, R-Tree etc.).

Processed data are stored in n-dimensional structures called cubes.

The elements of a cube are the dimensions, members, cells, hierarchies and properties [4] (fig. 1)

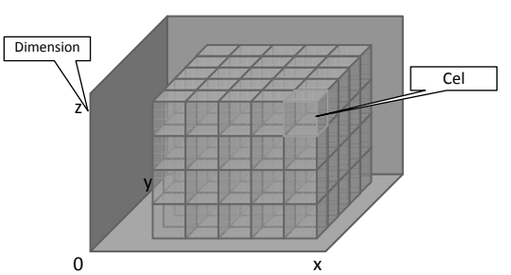

Figure 1. Structure of a tridimensional cube

The dimensions contain descriptive information about the data that is to be summarized. They are essential for data analysis and represent an axis of the cube [5].

The cells of the cube contain summarized data.

In order to optimize performance, OLAP systems implement cube indexing algorithms. The indexes created through this process use the data contained in a cube's dimensions to quickly access the cells containing the data required by the user.

Hence, cubes are indexed using a B-Tree type of algorithm.

3. The B-Tree Index in OLAP Systems

A B-Tree index used in OLAP systems contains sub-trees corresponding to each dimension. The subtrees are connected in such a way that each path to go through the tree from the root node to the final level index blocks (the ones storing the references to the cube's cells) is crossed by a sub-tree corresponding to each dimension.

Thus, a three dimensional cube contains three levels. The first level represents a matrix of planes (bidimensional space), the second level represents a matrix of lines (one dimensional space), and the third level represents a matrix of points in space (0 dimensional space) (fig. 2).

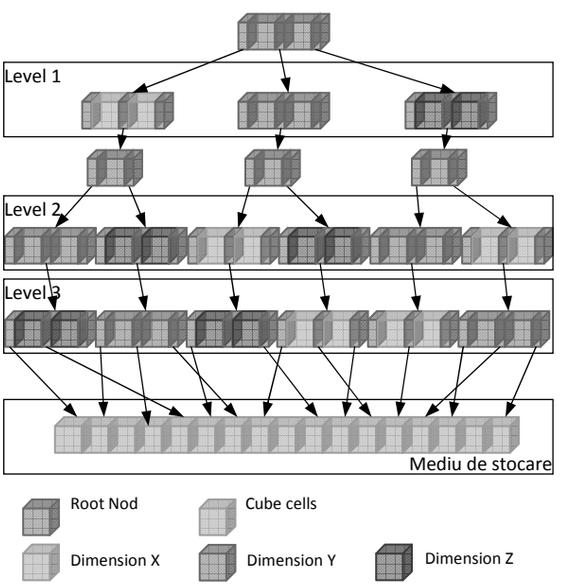

Figure 2. Structure of a B-Tree index belonging to a three-dimensional cube [6]

Looking at the structure of the B-Tree index, it is easily noticeable that the cost of locating one of the cube's cells represents the sum of the costs associated with locating the last level index block of each subtree. The height of such an index is (f.1) [7]:

$$h \le \log_d \frac{r-1}{2} \qquad (f.1)$$

where $h \in N$, d is the number of stored values within an index block, r is the total number of values corresponding to the dimension.

The total cost of a search operation within a subtree is:

$$c_i = h+1 \qquad (f.2)$$

Using (f.2) we can calculate the total cost of a search operation in a B-Tree index in an OLAP system:

$$c_{ti} = \sum_{i=1}^{n} c_{i_i} \qquad (f.3)$$

where c_{i_i} represents the total cost of a search operation in a sub-tree.

4. The n-Tree Indexing Algorithm

Given the characteristics of cubes, as well as the structure of a B-Tree index, it becomes obvious that this indexing algorithm is not optimized for n-dimensional data structures. Thus, the number of subtrees within the index is directly proportional with the number of dimensions. As a consequence, the cube is over-indexed resulting in an overconsumption of processing time and storage space.

The proposed n-dimensional indexing algorithm pays attention to the n-dimensional structure of the data.

Instead of creating sub-trees corresponding to each dimension and subsequently linking them, the equivalent of moving a cursor on each dimension, the n-Tree index creates only one tree which indexes data simultaneously on all dimensions, the equivalent of moving a cursor on the diagonal. As a result, the ndimensional space is gradually divided into ever smaller n-dimensional subdivisions, until the smallest sub-divisions represent the cells of the cube.

Because the diagonal is the shortest distance between 2 points in any multidimensional space, the interrogation cost of an n-Tree indexed cube will be minimum.

The resulting index has the following characteristics:

- no NULL values are indexed;

- the root node contains at least two subordinated index blocks if it does not coincide with the last level index block;

- each index block contains:

- values from each dimension of the cube; the combination of such values represents a reference point in the n-dimensional space;

The index maintains an ordered list containing unique values corresponding to each dimension of the cube. The values in each list represent a subgroup of the values of the respective dimension. Combining values from each list at a time, we can obtain the data needed to identify the reference points in the ndimensional space simultaneously minimizing the space required to store them.

Any value corresponding to a dimension from the subordinated index block is smaller than the value of the respective dimension corresponding to the reference point from the upper level index block. - references to the subordinated index blocks (r_{bs}), corresponding to the reference points (*f.4*):

$$r_{bs} = \prod_{i=1}^{n} a_i \qquad (f.4)$$

where $a_{1..n}$ represents the number of values from dimensions 1..n stored in the index block;

- *n* references to the index blocks that contain larger values in a dimension than the reference point (one for each dimension).

Thus, an index block contains a total of r references (f.5):

$$r = r_{bs} + n \qquad (f.5)$$

where n equals the number of the cube's dimensions.

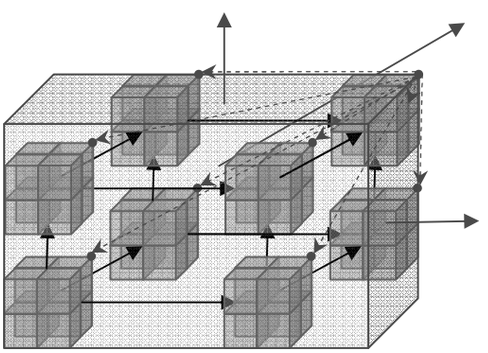

Figure 3. Structure of an n-Tree index corresponding to a three dimensional cube

- the last level index blocks do not contain any references to other index blocks; instead they store the reference to the physical location of the cube's cells;

- the physical size of an index block is approximately one page.

5. Creating an n-Tree Index

To create an n-Tree index, all data in every index is read and n-dimensional points are created. For each of these points, the following operations are carried out:

– an index block corresponding to an n-dimensional sub-space whose reference point has only values larger than that of the processed point is identified; the index block must also have enough free space to store the values of the corresponding dimensions of the processed point plus a reference;

If such an index block is identified, the values are added to the dimensions' corresponding lists and the reference to the physical location of the cube's cell is stored. Otherwise, a new index block is created by dividing one of the neighboring index blocks.

– when a new index block is created, a new cell is added to the parent index block, which stores the values of the reference point, as well as the reference to the new index block;

– if the index block doesn't have enough free space to store the new cell, it splits, similarly to the BTree index blocks, resulting 2x(dimensions number) index blocks.

This process could propagate itself to the root node.

Generally, OLAP systems contain historical data with a low frequency of updating operations but with a large volume of updates.

Updating these data also triggers an updating of the cube, and thus, of its index. Usually, updating an OLAP index is not efficient even is possible. The n- Tree index is not an exception, but this option will not be detailed in this paper.

6. The Performance of the n-Tree Index

The performance of an index depends on its height. A larger height means more physical read operations are needed to identify the cell containing the required data.

The height of the index depends on the size of the index block, the size of the type of indexed data, the number of references to subordinated index blocks stored in each index block, and the number of cells.

By analyzing the structure of an index block (fig. 4), we can compute the number of references it can store.

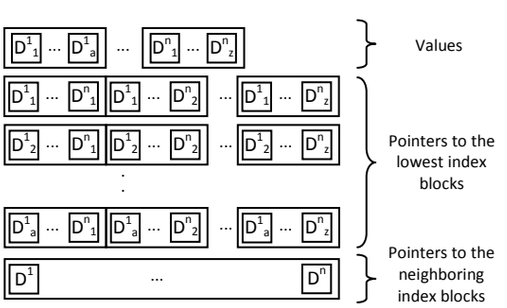

Figure 4. Structure of an index block in an n-Tree index

The volume of the stored data in an index block may be written as (f.6):

$$S_d = \sum_{i=1}^{n} a_i \cdot S_v + \left(\prod_{i=1}^{n} a_i + n \right) \cdot S_{ref} \qquad (f.6)$$

where S_d represents the volume of the stored data, S_v represents the size of the indexed value, n is the number of dimensions and S_{ref} the size of a reference.

Since the data in an OLAP system is rarely modified, the best performance is obtained when the index blocks contain a volume of data equal to their size.

Therefore, we can approximate the value of S_d to be equal to that of a page.

We assume that:

– the size of an index block is of 8kB (this is the most common size in current database systems [2]);

– the size of a reference is 6B (the most common size for a local index [8]);

– the size of the data type is 8B (this is the size of the *datetime* type of data);

– each dimension contains the same number of unique values.

Using the formulae (f.3) and (f.6), we can compare the performance of a n-Tree index to that of a B-Tree index (fig 5-8).

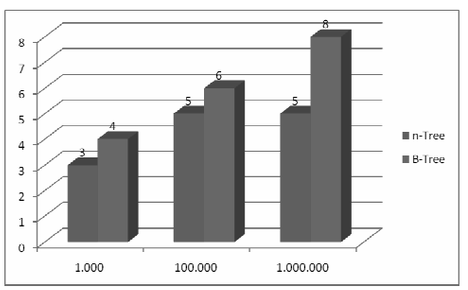

Figure 5. The cost of a search operation in a twodimensional cube

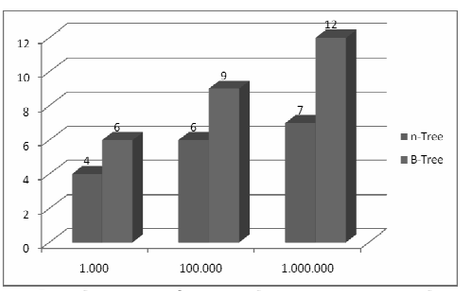

Figure 6. The cost of a search operation in a three dimensional cube

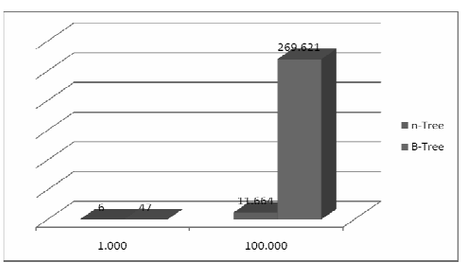

Figure 7. The size [in MB] of an index corresponding to a two dimensional cube

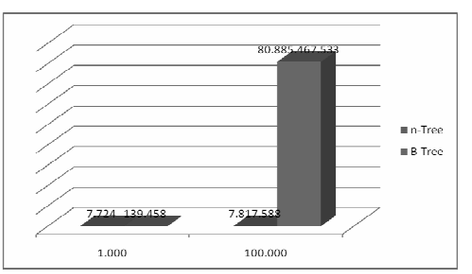

Figure 8. The size [in MB] of an index corresponding to a three dimensional cube

7. Conclusions

The n-Tree index could be considered a more generalized B-tree index. If B-Tree index can index only uni-dimesional data, the n-Tree index is optimized for any n-dimensional data. Moreover, the n-Tree index will be more suitable for indexing spatial data.

As shown in figures 5-9, the n-Tree index outperforms the B-Tree index in locating the cells of the cube. Moreover, the difference in performance increases as the number of the cube's cells rises. In addition, the space occupied by the n-Tree index in much smaller than that needed for a B-Tree index.

Again the superiority of the n-Tree index is all the more evident when the number of the cube's cells increases.

References

[1] Microsoft SQL Server 2005: Changing the Paradigm, Sams.
[2] MCTS 70-431 (2006). Implementing and Maintaining Microsoft SQL Server 2005, Que
[3] Microsoft SQL Server 2008 Administration, Wiley.
[4] Revista Informatica Economic (2002). nr. 4 (24), 2002;[4]
[5] Revista Informatica Economica (2001). nr. 1 (17).
[6] Dehne, Frank., Rau- Andrew (2001). Computing Partial Data Cubes for Parallel Data Warehousing Applications, chaplin, Computational Science – ICCS.
[7] Ubiquitous B-Tree (1979). Douglas Corner, ACM.
[8] Index Internals (2005). Julian Dyke.