B-tree Index Structures for Multimedia Data

Vladimir T. Dimitrov[1]

***Abstract* – There are specific requirements to Data Base Management Systems (DBMS) for multimedia data support. Conventional index structures are designed for storage and retrieval of conventional data. They have demonstrated some limitations in the case of multimedia data. B-tree index structures are more appropriate for storage and retrieval of multidimensional data.**

***Keywords* – Database Systems, Multimedia Data, Tree-Like Index, Multidimensional Data.**

I. INTRODUCTION

Conventional index structures are one dimensional, in sense that they assume a single search key, and they retrieve records that match a given search key value. There are applications that view data as existing in a 2-dimensional space, or even in higher dimensions. This kind of applications are hardly supported by conventional DBMS, instead specialized systems are designed for multidimensional applications. One important way in which these specialized systems distinguish themselves is by using data structures that support certain kind of queries that are not common in SQL applications.

Geographic Information Systems (GIS) are typical example of multidimensional applications. They store its objects (points or shapes) in two-dimensional space. These databases are maps, where the stored objects represents houses, roads, bridges, pipelines, and other physical objects.

The queries used in GIS are not typical of SQL queries, although many of them can be expressed in SQL. These queries can be classified as follow:

- **Partial match queries**. Values are specified for one or more dimensions and DBMS has to search for all points matching those values in those dimensions.
- **Range queries**. Ranges are given for one or more of the directions, and DBMS search for the set of points within those ranges. Shapes can be searched partially or wholly within the range.
- **Nearest-neighbor queries**. DBMS search for the closest point to a given point.
- **Where am I queries?** DBMS search for shapes in which a given point is located.

There are four tree-like structures useful for range queries or nearest neighbor queries on multidimensional data:

1. Multi-key indexes.
2. kd-trees.
3. Quad trees.
4. R-trees.

1-3 are intended for sets of points. The last one is commonly used to represent sets of regions, but is also useful for points.

II. MULTIPLE-KEY INDEXES

In this case, several attributes are representing the data points. A simple tree-like scheme for accessing these points is an index of indexes, or more generally a tree in which the nodes at each level are indexes for one attribute.

In Fig. 1 the idea is illustrated for the case of two attributes. The "root of the tree" is an index for the first of the two attributes. This index could be any type of conventional index, such as a B-tree or a hash table. The index associates with each of its search-key values a pointer to another index. If V is a value of the first attribute, then the index which is returned by following key V and its pointer is an index into the set of points that have V for their value in the first attribute and any value for the second attribute.

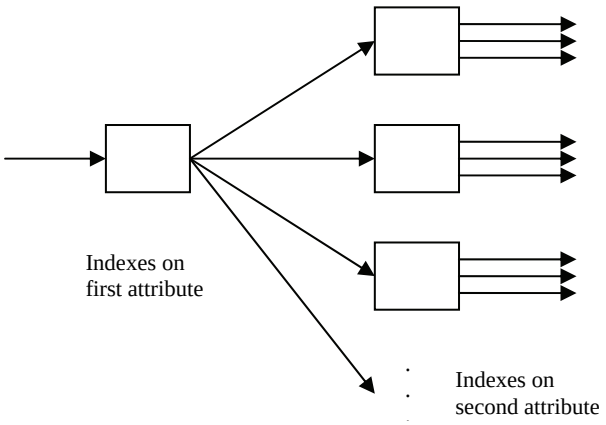

Fig. 1. Nested indexes on different keys

In a multiple-key index, some of the second or higher rank indexes may be very small. Thus, it may be appropriate to implement these indexes as simple tables that are packed several to a block.

Partial-match queries. If the first attribute is specified, then the access is quite efficient, all what have to be done is to find the one subindex that leads to the desired points. If the root is a B-tree index, then two or three disk I/O's have to be done to get the proper subindex and then to use whatever I/O's are needed to access all of that index and the points of

[1]Vladimir T. Dimitrov is associate professor at the Faculty of Technical Sciences, University of Sofia, POB 1829, 1000 Sofia, Bulgaria, e-mail: cht@fmi.uni-sofia.bg

the data file itself. If the first attribute does not have a specified value, then every subindex have to be searched, a potentially time-consuming process.

Range queries. The multiple-key index works quite well for a range query provided the individual indexes themselves support range queries on their attribute. To answer a range query, root index has to be used and the range of the first attribute to find all of the subindexes that might contain answer points. Then each of these subindexes has to be searched using the range specified for the second attribute.

Nearest-neighbor queries. To find the nearest neighbor of point (x_0, y_0), a distance d has to be find first, such that several points are expected to be within distance d of point (x_0, y_0). After that the range query can be applied: $x_0 - d \le x \le x_0 + d$ and $y_0 - d \le y \le y_0 + d$. If there are no points in this range, or if there is a point, but distance from (x_0, y_0) of the closest point is greater than d, then the range has to be increased and search repeated. Search can be ordered so that the closest places are searched first.

III. KD-TREES

kd-tree is generalization of the binary search tree to multidimensional data. It is a main-memory data structure. A kd-tree is a binary tree in which interior nodes have an associated attribute a and a value V that splits that data points into two parts: those with a-value less than V and those with a-value equal or greater than V. The attributes at different levels of the tree are different, with levels rotating among the attributes of all dimensions. In the classical kd-tree, the data points are placed at the nodes, but for the sake of block model of storage two modifications are done:

1. Interior nodes have only an attribute, a dividing value for that attribute, and pointers to left and right children.
2. Leaves are blocks, with space for as many records as a block can hold.

An example of kd-tree is presented in Fig. 2. There are two dimensions (Speed and Age), which are alternatively splitting data points at each level. Leaves contain data points and they can be placed at each level of the kd-tree.

For the lookup values for all dimensions could be given. A lookup of a tuple is performed as in a binary search tree. At every interior node the search is redirected to a subtree which is possible to contain a leaf with the tuple.

Insertion starts as lookup to find the leaf. If the block of the leaf has enough room – the new data point is inserted. If there is no room, the block is divided into two new blocks. Content of the block is distributed into the new two blocks using the attribute corresponding to the leaf level. New interior node is created whose children are the two new blocks. In the new interior node the splitting value is put.

Partial-match queries. If values are given for some of the attributes, then at every level belonging to attribute whose value is known search direction is clear. If there is no value for the attribute at a node, then both its children have to be explored.

Range queries. Sometimes, a range will allow search direction to be directed only to one child of a node, but if the range straddles the splitting value at the node, then both children have to be explored.

Nearest-neighbor queries. They are executed in the same way as in the case of multiple-key indexes.

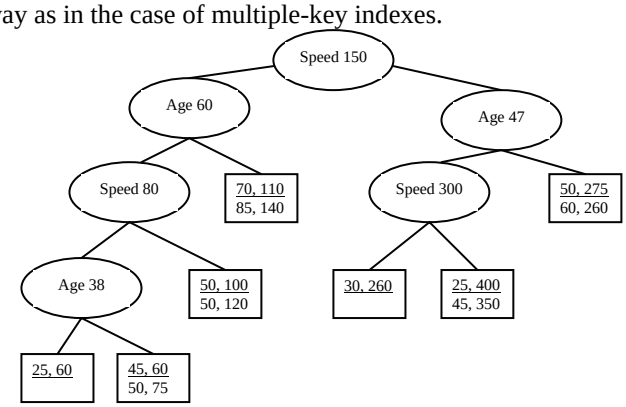

Fig. 2. A kd-tree with two dimensions

Adapting kd-trees to secondary storage. Let kd-tree with n leaves is stored in a file. Then the average length of a path from the root to a leaf will be about log2 n, as for any binary tree. If each node is stored in a block, then to traverse a path one disk I/O per node must be done, which in summary is more than for the typical B-tree. In addition, since interior nodes of kd-tree have relatively little information, most of the block would be wasted space. The twin problems of long paths and unused space cannot be solved completely, but there are two approaches that will make some improvements in performance:

- **Multiway branches at interior nodes**. Interior nodes of a kd-tree could look more like B-tree nodes with many key-pointers pairs. If a node contains n keys, then values of an attribute could be split into n + 1 ranges. If there are n + 1 pointers, then could be followed appropriate one to a subtree that contain only points with attribute in that range. Problems are when reorganization of the nodes has to be done, in order to keep distribution and balance.
- **Group interior nodes into blocks**. In this approach the tree nodes have only two children, but many interior nodes are packed into single block. In order to minimize the number of block that have to be read from disk while traveling down one path, the best is to include in one block a node and all its descendants for some number of levels. That way, once the block with this node is retrieved, it is possible to use some additional nodes on the same block, saving disk I/O's.

IV. QUAD TREES

In quad tree, each interior node corresponds to a square region in two dimensions, or to a k-dimensional cube in k dimensions. If the number of points is no larger than what will fit in a block, then this square is a leaf of the tree, and it is represented by the block that hold its points. If there are too many points to fit in one block, then the square is an interior

node, with children corresponding to its four quadrants. See Fig. 3 for an example of quad tree.

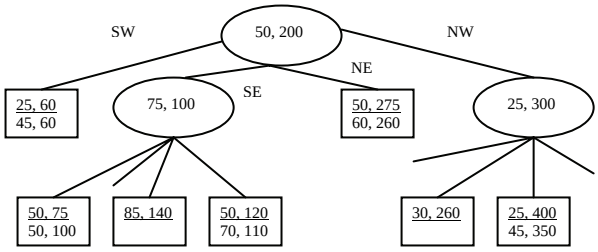

Fig.3. A quad tree

Since interior nodes of a quad tree in k dimensions have 2k children, there is a range of k where nodes fit conveniently into blocks. However, for the 2-dimensional case, the situation is not much better than for the kd-tree; an interior node has four children. In quad tree splitting point for a node has to be the centre of a quad-tree region, which may or may not divide the point in that region evenly. When the number of dimensions is large many null pointers in the interior node could be found. In this case, only non-null pointers can be represented.

Standard operations on quad tree resemble those for kd-tree.

V. R-TREES

An R-tree (region tree) is a data structure that captures some of the spirit of a B-tree for multidimensional data. B-tree node has a set of keys that divide a line into segments. Points along that line belong to only one segment and it is easy to determine a unique child of that node where the point could be found.

An R-tree represents data that consists of 2-dimensional or higher-dimensional regions, which are called data regions. An **interior node** of an R-tree corresponds to some interior region, which not normally a data region. The region can be of any shape, but in practice it is usually rectangle or other simple shape. The R-tree node has, in place of keys, subregions that represent the contents of its children. The subregions are not needed to cover entire region, which satisfactory as long as all the data regions that lie within the region are wholly contained within one of the subregions. The subregions are allowed to overlap, although it is desirable to keep the overlap small. An R-tree for a map is presented in Fig. 4.

R-tree is useful for “where-am-I” queries, which specify a point P and asks for the data regions in which the point lies. Search starts from the root, with which the entire region is associated. The subregions at the root are examined to determine which children of the root correspond to interior regions that contain point P.

If there are zero regions, then P is not any data region. If there is at least one interior region that contains P, then P must be recursively searched at the child corresponding to each such region. When one or more leaves are reached, then actual data regions should be found or a pointer to that record.

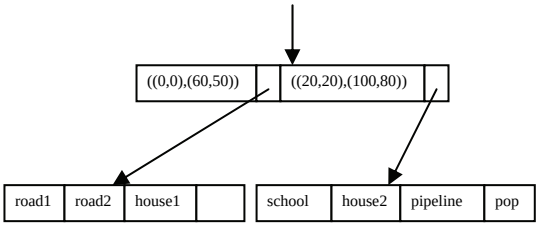

Fig. 4. An R-tree for a map

When new region R into R-tree should be inserted, procedure starts from the root and looks subregion into which R fits. If there is more than one such region, then one of them is picked and process is repeated there. If there is no subregion that contains R, then one of the subregions has to be expanded. Which one to pick may be difficult decision. Idea is to expand regions as little as possible, children’s subregions have to be asked to increase their area as little as possible, change the boundary of that region to include R, and recursively insert R at the corresponding child.

Eventually, a leaf is reached where region R is inserted. If there is no room for R at that leaf, then the leaf must be divided. The new two subregions have to be as small as possible, but they have to cover all the data regions of the original leaf. Having split the leaf, the region is replaced and pointer for the original leaf at the node above is replaced too by a pair of regions and pointers corresponding to the two new leaves. If there is a room in the parent, process is finished. Otherwise, recursively nodes are divided going up the tree.

VI. BITMAP INDEXES

In this index structure file records have permanent numbers and there is some data structure that is used easy to find record by their numbers. Such a structure can be a conventional index on the data file, but in the data file there is no need to support key field (the record number).

A **bitmap index** for a field F is a collection of bit-vectors of length n, one for each possible value that may appear in the field F. The vector for value v has 1 in position i if the record with that number has v in field F, and has 0 there if not.

Bitmap indexes require too much space, when there are many different values for a field, since the total number of bits is the product of the number of records and the number of values. That is why, compression has to be used.

Bitmap indexes have management problems, but they answer partial-match queries very efficiently in many situations. They can also help answer range queries. These kind of queries are supported by using AND/OR operations with bitmap-vectors.

Let bitmap index on field F of a file has n records, and there are m different values for field F that appear in the file. Then the number of bits in all the bit-vectors for this index is mn. This number can be small compared to the size of the file itself, but the larger m is, the more space the bitmap index takes.

But if m is large, then 1’s in a bit-vector will be rare - the probability that any bit is one is 1/m. If 1’s are rare, then bit-

vectors can be encoded to take much fewer than n bits on the average. **Run-length encoding** approach is encoding sequence of i 0's followed by a 1 (**a run**), by some suitable encoding of the integer i. The codes for each run are concatenated together, and that sequence of bits is the encoding of the entire bit-vector.

Representing the integer i as a binary number do not work, because there is no way to divide the concatenated encoded bit-vector on runs. So, the encoding of run length must be more complex. There are many encoding schemas, but they work well only when typical runs are very long. Such a schema uses the number of bits j needed to represent in binary the number i. This number j is approximately $\log_2 i$, and is represented by j - 1 1's and a single 0.

Concatenated sequence of above mentioned codes easy can be divided into j codes and then original vector can be recovered. For that purpose are used 0's as separators – every j-code is a sequence of 1's ending with 0.

Every bit-vector so decoded will end in 1, and any trailing 0's will not be recovered, but the number of records in the file is known, so additional 0's can be added. Since 0 in a bit-vector indicates the corresponding record is not in the described set, then trailing 0's can be ignored.

Let m = n, i.e., each value for the filed on which bitmap index is constructed, has a unique value. The code for a run of a length i have about $2\log_2 i$ bits. If each bit-vector has a single 1, then it has a single run, and the length of that run cannot be longer than n. Thus, $2\log_2 n$ bits are an upper bound on the length of a bit-vector's code in this case. Since there are n bit-vectors in the index (m = n), the total number of bits to represent the index is at most $2\log_2 n$. Without the encoding n^2 bits would be required.

Because only one run can be decoded at a time, operations (AND/OR) on bitmap-vectors can be interleaved with decoding at runtime.

Bit-vectors can be indexed with secondary indexes on values of the vector field, but instead value – a pointer to the bit-vector can be used. Bit-vectors can be packed in blocks of index data file, but if they are very long they can cross block boundaries in a chain of blocks.

There are two aspects to the problem reflecting data-file modifications in a bitmap index:

1. Record numbers must remain fixed once assigned.
2. Changes to the data file require the bit map index to change as well.

When a record is deleted its number is not used again and in the data file a "tombstone" is put. This is a consequence of point 1. The bitmap index must also be changed – for record position in all bit-vectors 0 has to be put.

When a new record is inserted, the next number is assigned to it. This means that next available number has to be supported persistently. In all bit-vectors at the end 1 has to be added where this is appropriate. Technically, 0 must be added too, but this operation can be postponed till next insertion of new record with a value in the corresponding field.

Modification a record will influence bit-vectors – value of 1 has to be changed to 0 and vice versa where is appropriate. Bit-vectors for new values have to be created, as in the case of insertion of new records.

VII. Conclusion

Above presented data structure are goods tools for effective storage and retrieve of persistent multimedia data. Some of them are used in currently available DBMS, but others are used only in specialized systems. Our intension is to extent an open source DBMS like MySQL with some of these indexed data structure for the support effectively and efficiently search and retrieval of multimedia data.

These data structures are at the physical level of an implementation of multimedia DBMS, but more research has to be done at higher levels: query language, query compilation and execution.

Acknowledgement

Presented overview is funded under contract VU-N-202/2006 "Multimodal biometric analyses: Methods and algorithms" with Bulgarian National Fund for Scientific Research.

References

[1] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger, The R*-tree: an efficient and robust access method for points and rectangles. Proc. ACM SIGMOD Intl. Conf. on Management of Data, 1990.

[2] J. L. Bentley, Multidimensional binary search trees used for associative searching, Comm. ACM 18:9, 1975.

[3] J. L. Bentley., Multidimensional binary search trees in database applications, IEEE Trans. On Software Engineering SE-5:4, 1979.

[4] J. L. Bentley, J. H. Friedman, Data structures for range searching, Computing Surveys 13:3, 1979.

[5] W. A. Burkhard, Hashing and tree algorithms for partial match retrieval, ACM Trans. on Database Systems 1:2, 1976.

[6] R.A. Finkel, J. L. Bentley, Quad trees, a data structure for retrieval on composite keys, Acta Informatica 4:1, 1974.

[7] A. Guttman, R-trees: a dynamic index structure for special searching, Proc. ACM SIGMOD Intl. Conf. on Management of Data, 1974.

[8] J. T. Robinson, The K-D-B-tree: a search structure for large multidimensional dynamic indexes, Proc. of ACM SIGMOD Intl. Conf. on Management of Data, 1981.

[9] T. K. Sellis, N. Roussopoulos, C. Faloutsos, The R+-tree: a dynamic index for multidimensional objects, Proc. Intl. Conf on Very Large Databases, 1987.

[10] J. B. Rothnie Jr., T. Lozano, Attribute based file organization in a paged memory environment, Comm. ACM 17:2, 1974.