Spatial, textual and multimedia databases

Erik Zeitler
erik.zeitler@it.uu.se

Abstract

This paper presents an overview of indexes for spatial and multimedia databases, whose indexes are often of the same kind as spatial ones. Records in spatial databases are characterized by their locations in an n-dimensional space where n is 1 or greater. A spatial database query involves finding all objects at a certain location, or finding all locations or areas occupied by a certain set of objects. The retrieval will be time-consuming if the DBMS must traverse the entire database in the search for the matching records, but will be speeded up if an index is used. For spatial database applications it is important to study efficient retrieval, insertion and deletion. Implementations of spatial indexes that facilitate these operations have been a very active field of study in database research.

1 Introduction

Two fundamental kinds of spatial database queries can be identified [9]:

- Feature query: Given an object, determine its location or constituent cells.
- Location query: Given a location in space, determine which objects occupy that location, and (optionally) the remaining constituent locations of those objects. The location query could be either a point query, or a window query (in one or more dimensions).

A location query could be

- a point query, e.g. “find point objects that are located exactly in a given point”, or “find objects that overlap with a given point”;
- a region (in one or more dimensions) “find objects that overlap (with a certain point or region)”;
- an approximation (this is often a case of an interval or nearest-neighbor query);
- a nearest-neighbor query, e.g. “find the n points that are closest to a certain other point”.

Furthermore, two fundamental representations can be identified:

- Implicit (image based) representation: A representation consisting of unit-size cells. For each cell in the space, a list of objects occupying that cell is stored.
- Explicit (object-based) representation: For each object, a list of its constituent cells is stored.

When implicit representation is used, the location query is easily answered, whereas a feature query has to examine all cells in space to be answered. On the other hand, an explicit representation is efficient for the feature query, whereas the execution of a location query must examine all objects. The goal is to find a spatial index that can efficiently answer both types of queries. In this paper, a few classes of indexes will be discussed:

- Hash tables. These are useful for point queries in one dimension. Query complexity is typically O(1).
- Space filling curves. These provide a mapping from the *d*-dimensional space to a one-dimensional space of index values (integers). They support point and range queries in spaces of any dimensionality.
- Containment hierarchies (trees) are an important class of spatial indexes. These hierarchies split the search space hierarchically, thereby reducing the search complexity typically from linear to logarithmic. For implicit representations, space is aggregated into successively larger blocks; while for explicit representations, objects are aggregated into successively larger groups.

This paper is organized as follows: In Section 2, the concept of inverted files is introduced. Methods for searching one-dimensional data are described, like the classical B-tree and its variants. Section 3 discusses space filling curves, which allow transformations of multi dimensional attributes to a one-dimensional attributes. Section 4 presents object-based and image-based multi-dimensional spatial indexes.

2 One dimensional spatial indexes

An inverted file is the simplest and most popular access structure in database systems. Inverted files are important for efficient execution of the location query when an object based representation is used. For each distinct value of an attribute, the inverted file stores a list of pointers to records that have the same attribute value, and (optionally) the length of that list. Finding certain values or ranges is then a matter of searching the list of attribute values. This *postings list* of attribute values can in turn be indexed using a hash table or a B-tree. On the other hand, image based representations are not frequently used in one-dimensional applications.

2.1 B-trees

A B-tree is a one-dimensional index structure. It has the structure of a search tree with the following additional properties:

- It is *balanced*. On insertions and deletions, B-trees are automatically re-balanced if needed.
- The space wasted on deletions never becomes excessive. A B-tree is always 50…100% full. This ratio (0.5…1.0) is called the *fill factor*.

Formally, a B-tree of order p is defined as follows [11];

1. Each node in a B-tree is of the form
   $<P_1, K_1, Pr_1, P_2, K_2, Pr_2, \ldots, P_{q-1}, K_{q-1}, Pr_{q-1}, P_q>$,
   where $q \leq p$. Each P is a pointer to a subtree, and each Pr_i is a data pointer, i.e., a pointer to the location in the postings list for the records that have attribute value K_i.
2. Within each node, $K_1 < K_2 < \ldots < K_{q-1}$.
3. For all attribute values X in a subtree pointed at by P_i we have
   $K_{i-1} < X < K_i$ for $1 < i < q$, $X < K_1$, and $K_{q-1} < X$.
4. Each internal node has at least $\lceil p/2 \rceil$ tree pointers. The root node has at least two pointers unless it is the only node.

5. All leaf nodes are at the same level. The subtree pointers of the leaf nodes are null.

2.1.1 B-tree operations

A search starts at the root and descends the tree. For each node, the searched attribute value v is compared the K_i values in the tree. Depending on whether v is less or greater than K_i, the search will continue down the left or right subtree respectively. If $v=K_i$ the records pointed at by Pr_i is returned as a result.

When a value is inserted, a search is first performed to find the location of insertion. Nodes that overflow are split. The first $\lceil (p+1)/2 \rceil$ entries are kept in the original node, and the remaining entries are moved to a new node, which will be pointed to by a new tree pointer in the parent node of the original node. If the parent node overflows, the process will be repeated for that node. Thus, node overflows and splits might propagate all the way up to the root, thereby creating a new tree level.

If a value or a subtree is deleted from a node so that it is filled less than $0.5p$, then the node is merged with its neighbor. This process might also propagate so that one deletion results in the removal of an entire tree level.

2.1.2 Tree fanout and B-tree variants

A tree with many entries per node (high fanout) will have fewer levels and hence fewer comparisons will be needed until the value is found. On the other hand, a tree node with many subtrees will not fit into a single disk block. To fit a node into a single disk block we employ the following model. Each node consists of p subtree pointers of size P, $p-1$ record pointers of size P_r, and $p-1$ search attribute values of size V. Assuming block size B, we have

$$pP + (p-1)(P_r+V) \leq B \Leftrightarrow p \leq (B+P_r+V) / (P+P_r+V)$$

This gives a maximum value for the B-tree fanout, given the disk block and pointer sizes.

A variant of the B-tree, the *B⁺-tree*, is often used in applications. The B⁺-trees store record pointers only at leaf level. This leads to fewer levels and higher-capacity indexes. Furthermore, all B⁺-tree leaf nodes are linked together, which allows sequential access. Formally, a B⁺-tree is defined in the same way as a B-tree, with the following exceptions:

1. Each internal node in a B⁺-tree is of the form
   $\langle P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q \rangle$,
   and a leaf node is of the form
   $\langle\langle K_1, Pr_1\rangle, \langle K_2, Pr_2\rangle, \ldots, \langle K_{q-1}, P_{rq-1}\rangle, P_{next}\rangle$
   where P_{next} is a pointer to the next leaf node in the tree.

3. For all attribute values X in a subtree pointed at by P_i we have
   $K_{i-1} < X \leq K_i$ for $1 < i < q$, $X \leq K_1$, and $K_{q-1} < X$.

5. All leaf nodes are at the same level. Leaf nodes have no subtree pointers.

The maximum fanout is computed using a model similar to the B-tree:

$$Pp^+ + ((p^+-1)V) \leq B \Leftrightarrow p^+ \leq (B+V)/(P+V)$$

Comparing the formulas for p and p^+, we can show that

$$(B+P_r+V) / (P+P_r+V) \leq (B+V)/(P+V) \Leftrightarrow$$

$$BP + PP_r + PV + VB + VP_r + V^2 \leq BP + BP_r + BV + VP + V^2 \Leftrightarrow$$

$$P \leq B,$$

which is always true – QED. So the B+-tree index is of higher capacity because its maximum fanout is higher.

One more variant on B-trees is to change the fill factor constraint. For instance, the B*-tree requires each node to be at least 2/3 full. Some DBMSs allows the user to change the fill factor requirement for the leaf level and for the node level separately.

3 Space filling curves reduce dimensionality

Space filling curves is a class of access structures for multi-dimensional image based representations of unit size cells. They are a mapping between a multi-dimensional representation and the integers, since they impose a linear ordering of the elements (pixels) of an *n*-dimensional space. Thus, space filling curves reduce a multi dimensional location query to a one dimensional query, which in turn can be speeded up using one-dimensional indexes like hash tables or B-trees.

Samet [9] lists six desirable properties of space filling curves:

- Uniqueness: The curve should pass through each location in space once and only once
- Simple mapping: The mapping between the *d*-dimensional space and the integers should be simple.
- Stability: The ordering should be stable, i.e. the relative ordering of the individual locations is preserved when the resolution is doubled.
- Adjacency preservation: Two locations that are adjacent in space are neighbors along the curve and vice versa. This is impossible to satisfy for all locations at all space sizes.
- Easiness of retrieval: The process of retrieving the neighbors of a location in space should be simple.
- Admissible: At each position in the ordering at least one adjacent neighbor in each of the directions must have already been encountered. This is useful in several algorithms.

The simplest space filling curve is the row order, which is also known as the multi dimensional array access structure. In many computer graphics applications, this ordering is very natural since a bitmapped image is organized in memory in the same way as a bitmapped image.

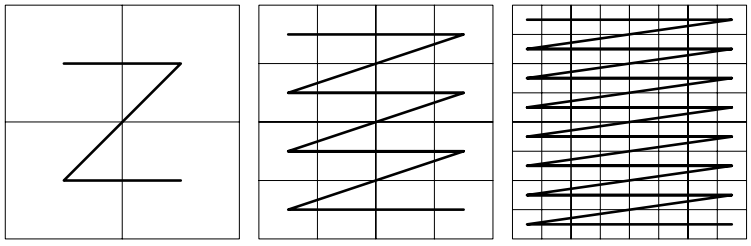

Figure 1. The row ordering in two dimensions is shown for 2x2, 4x4, and 8x8 collections of cells (resolution 1, 2, and 3). The ordering enumerates the cells starting from the upper left corner and following the curve.

Another example of a space filling curve is the Morton order or z-ordering. The z-ordering for two dimensions is shown in

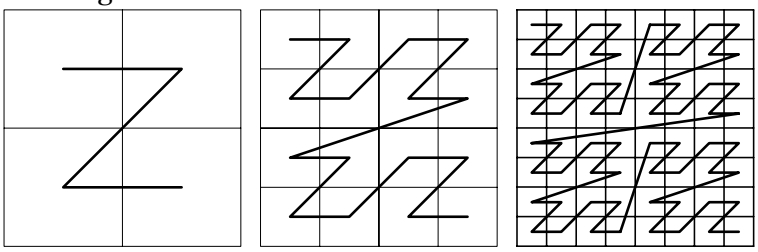

Figure 2. This ordering is a more complicated mapping than the row order, but it fulfills other properties (e.g. stability and admissibility) better than the row order. It is also simply calculated by bitwise interlacing the keys. Many other orderings have been proposed, like the Peano-Hilbert order, Gray order etc. However, there is no space-filling curve that can fulfill all desirable properties at the same time.

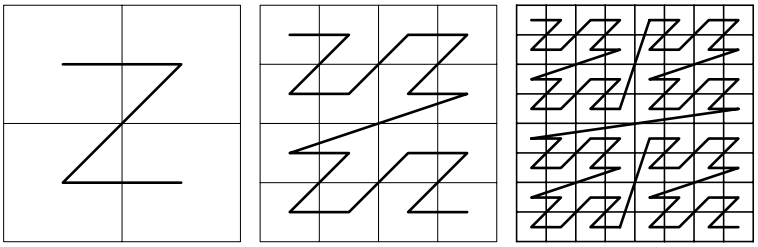

Figure 2. The Morton order in two dimensions is shown for 2x2, 4x4, and 8x8 collections of cells. The ordering enumerates the cells starting from the upper left corner and following the curve.

Figure 3 shows how 2-D region query is transformed into a set of 1-D interval queries using Morton order. The shaded region query shown in the upper part of Figure 3 is transformed into this set of 1-D interval queries shown in the lower part of the figure.

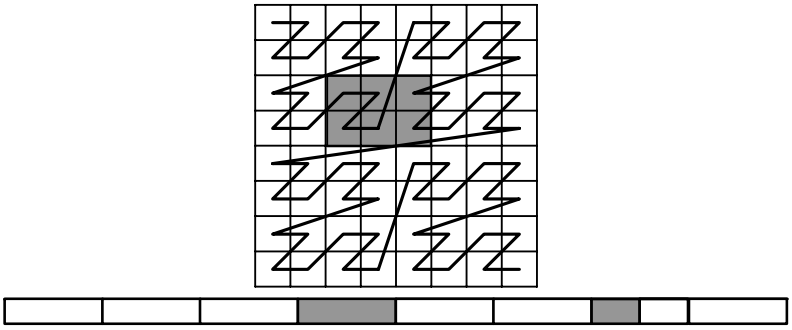

Figure 3. Upper part: A spatial region query shown as a shaded region in a 2-D collection of cells. Lower part: The region of the query is transformed into a set of 1-D intervals.

3.1 UB-trees

The UB-tree (Universal B-tree) [2] enables B-trees to be used for multi dimensional data. The records of the B+-tree are stored according to the Morton order. Insertion, deletion, and point queries are performed in the B+-tree. Range searches in multidimensional point data will be transformed using the order as shown in Figure 3.

4 Multi-dimensional spatial indexes

In this section, some object-based and image-based indexes are discussed. First, some image-based hierarchies are presented, including the grid file and k-D tree. Next, the classical R-tree is discussed. Finally, some R-tree variants and their basic properties are outlined.

4.1 Block hierarchies

Block hierarchies decomposes the space by successively partitioning it into blocks. In the classical block hierarchies, the number of splits on each level is either 2, where each node splits the space in two, or 2^d, where each node performs one split per dimension.

Regular decompositions (bintrees and region quad trees) place the first split at the origin. Successive splits are placed in the most crowded quadrant. Regular decompositions might well result in unbalanced trees if the distribution is not uniform. Figure 4 A shows the result of three successive splits of a region quadtree in two dimensions where the data distribution is towards the lower right.

Irregular decompositions have more freedom to place splits. The classical k-D tree [4] splits one dimension per level. The level below is splitting the next dimension, and so on in a round robin fashion. Figure 4 B shows a space partitioned by a k-D tree.

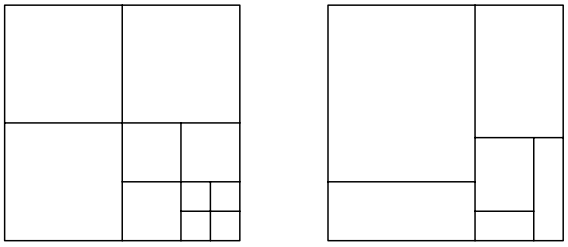

Figure 4 A. A region quad tree partitioning in two dimensions. B. The result of a generalized k-D tree split.

Figure 5 shows a classification of block hierarchies. Note that the quad tree is often called “octtree” for 3-dimensional spaces since each node divides the space in eight parts.

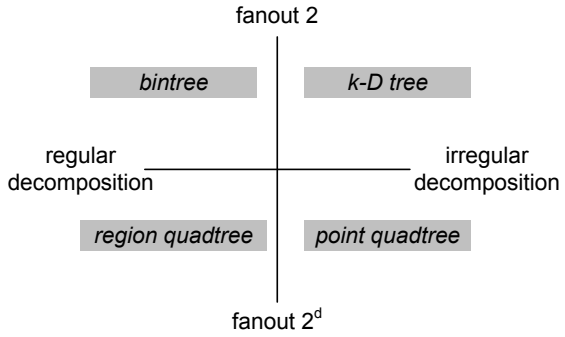

Figure 5. A classification of block hierarchies.

4.2 R-trees

The R-tree is an object-based index structure for objects which have spatial extent. It can be said to capture the spirit of B^+-trees in n dimensions, to enable answering location and range queries without the need of a space filling curve as in the UB-tree. The R-tree aggregates every M objects, that are close to each other, to larger blocks. This process is repeated recursively until there is only one node left. Searching R-trees for objects that contain a location a involves descending all sub trees that contain a. Sub trees that do not contain a are pruned from the search.

The original R-tree article by Guttman [7] presents the R-tree data structure, the basic algorithms, and some performance tests on some different R-tree reorganization algorithms. If M is the maximum number of entries that fit into a node, and $m \leq M/2$ is a parameter specifying the minimum number of entries in a node, Guttman formally defines an R-tree for an n-dimensional space as follows:

- Every leaf node contains between m and M index records unless it is the root.
- For each index record `<I, P>` in a leaf node, $I=(I_o, I_1, ..., I_n)$ is the smallest n-dimensional rectangle (bounding box) that spatially contains the object. I_k is a closed bounded interval describing the extent of the object along dimensin k. It might have one or both endpoints equal to infinity, indicating that the object extends indefinitely.
- Every non-leaf node has between m and M children unless it is the root.
- For each entry `<I, Pr>` in a non-leaf node, I is the smallest n-dimensional rectangle that spatially contains the rectangles in the child node.
- The root node has at least two children unless it is a leaf.
- All leaves appear on the same level.

Searching index records in an R-tree is similar to B-tree search. A search starts at the root and descends the tree. Given a root node T, find all index records that overlap a search rectangle S, by inspecting the rectangle part EI of an index entry E:

If T is not a leaf, check each entry E to determine whether EI overlaps S. For all overlapping entries, search the tree whose root node is pointed to by E_p. If T is a leaf, check all entries E to determine whether EI overlaps S. If so, E is a qualifying record.

The insertion into R-trees is similar to that of B-trees. New index records are added to the appropriate leaf, and overflowing nodes are split. The node split must be done so that subsequent searches will only examine nodes that contain qualifying records. To find a good node split is an optimization problem, to which Samet [9] outlines three possible objectives: (i) minimize the overlap between sibling nodes (Figure 6(a)); (ii) minimize the total area spanned by the bounding boxes of the sibling nodes (Figure 6(b)); and (iii) minimize the dead area (Figure 6(b)).

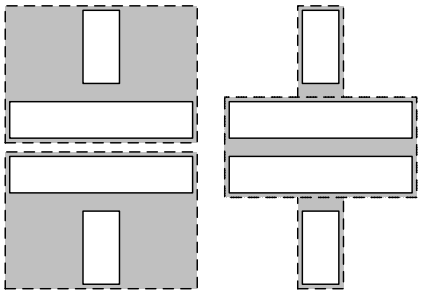

Figure 6. (a) Aggregation of two bounding boxes minimizing the overlapping area. (b) Aggregation of the same boxes minimizing the total area. The dead area for the two aggregations is shaded.

In the original R-tree article, Guttman attempts to minimize the overlapping area, thus achieving the partitioning shown in Figure 6(a). Guttman implements three different optimization methods;

- exhaustive search with complexity 2^{M-1} – which is infeasible even for moderately sized nodes,
- a quadratic cost algorithm which is quadratic in M and linear in the number of dimensions,
- a linear cost algorithm which is linear both in M and in the number of dimensions.

(a)

The quadratic and linear algorithms do not guarantee to find the minimum. However, Guttman's experiments show that the linear cost algorithms produced node splits of quality comparable to higher-cost algorithms.

One important improvement of the R-tree is the R*-tree, proposed by Beckmann et al [3]. It differs in three important ways from the R-tree in the insertion procedure. These differences lead to shorter search times and more efficient storage utilization. First, the insertion decision works differently for leaf nodes and internal nodes. When an object is inserted, the overlap is minimized for each internal node it traverses, whereas the increase in area is minimized at the leaf node. Second, when a node overflows, that node is not split immediately as in an R-tree. Instead, the R*-tree attempts to move some of the objects of that node to another node. This is achieved by re-inserting a fraction of the objects in the overflowing node into the tree. This procedure is called *deferred split*. Third, the R*-tree has a two stage procedure for splitting nodes. The first stage selects the dimension for the split and the second selects the split position in that dimension.

5 Text and multimedia databases

Text and multimedia information retrieval, data mining, pattern recognition, machine learning, computer vision, biomedical databases, data compression and statistical data analysis all deal with complex or multi-dimensional data. To enable classification and retrieval in these applications, objects are characterized using features that are extracted from the objects. These features are represented using a set of parameters – a set of coordinates in an *n*-dimensional space. Therefore, the representation in these databases and index structures is similar to those of spatial databases. Consequently, all indexing methods outlined in this paper are important also for these kinds of databases. Similarity

queries are a common class of queries in text and multimedia databases. Similarity between multimedia objects is reflected by proximity in feature space. Typically, such queries are of the kind `withindist(`$\underline{x}_0$`,d)` or `k_NN(`$\underline{x}_0$`)`. A class of spatial indexes, *metric trees,* has been developed for these kinds of queries. Some of these include Similarity Search trees (SS-trees) [12], M-trees [5], and distance indexes (D-indexes) [6]. All these exploit the triangle inequality of a metric space.

5.1 M-tree and D-index

The M-tree is similar to an R*-tree in that only leaf nodes contain pointers to records. Each leaf node contains the feature value, a pointer to the record, and the distance from the feature value to the feature value of its parent node. Each internal node has a feature value, a covering radius, and a distance from its parent. Since each internal node has a covering radius, the containment hierarchy of an M-tree can be envisioned as a set of (possibly overlapping) (hyper-) spheres in feature space. These spheres – whose centers are called *pivot points* – decrease the number of distance calculations for proximity searches. When compared to and R*-tree, the M-tree showed to outperform the R*-tree for distance and k-NN queries.

One shortcoming of the M-tree is the fact that regions may overlap, so that several branches need to be visited during a search. On the contrary, the hierarchy of the D-index [6] is non-overlapping. Like the M-tree, each node has a pivot point and a radius. However, each partitioning circle have a region of width 2ρ along the boundary that excludes is neither inside nor outside the circle. Figure 7 shows a partitioning circle with excluded middle partitioning.

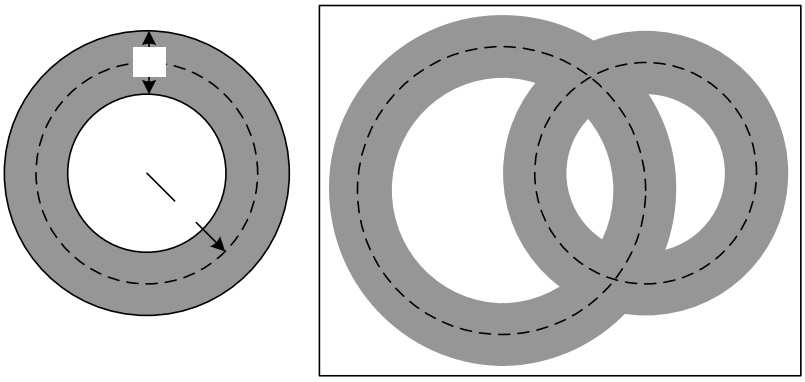

Figure 7. A. Excluded middle partitioning.
B. A combination of two excluded middle partitionings form separable sets (1…4) and an exclusion set (shaded).

Because of the excluded middle partitioning, each level *h* in the D-index will contain a number of buckets and an exclusion set, which is the union of all exclusion regions of that level. Each bucket contains its center point and radius, along with pointers to the objects that lie inside their region. The exclusion set of level *h* is further partitioned by the rest of the tree, as shown in Figure 8. A query of the `withindist(`$\underline{x}_0$`,d)` will descend all the buckets in the D-hierarchy that intersect with the query ball region.

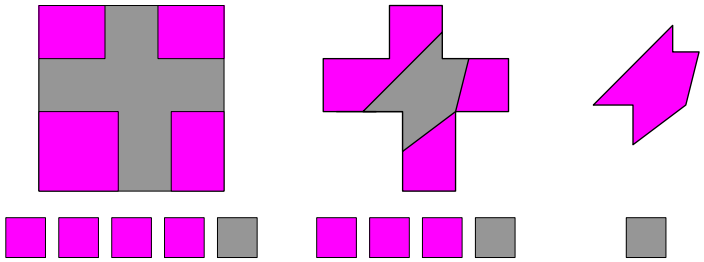

Figure 8. Three levels of hierarchy in a D-index. Four separable buckets are present at the first level. The exclusion bucket of the first level is partitioned at the second level into three separable buckets. Since there are no more levels of the tree, the figure shown to the right is the exclusion bucket of the whole structure.

References

[1] Bayer: Binary B-Trees for Virtual Memory, ACM-SIGFIDET Workshop 1971.

[2] F. Ramsak, V. Mark, R. Fenk, M. Zirkel, K. Elhardt, R. Bayer, "Integrating the UB-Tree into a Database System Kernel", in Proc. VLDB 2000, pp 263–272.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger, "The R*-tree: An efficient and robust access method for points and rectangles", in Proc. SIGMOD, (Atlantic City, NJ). 322–331.

[4] J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching", Communications of the ACM 1975, Vol. 18, issue 9, pp 509–517.

[5] P. Ciaccia, M. Patella, P. Zezula, "M-tree: An Efficient Access Method for Similarity Search in Metric Spaces", in Proc. VLDB 1997.

[6] V. Dohnal, C. Gennaro, P. Savino, P. Zezula, "D-Index: Distance Searching Index for Metric Data Sets", Multimedia Tools and Applications, 21, 9–33, Kluwer 2003.

[7] A. Guttman, "R-trees. A Dynamic Index Structure for Spatial Searching", pp 47–57 1984.

[8] Frank Ramsak, Volker Markl, Robert Fenk, Martin Zirkel, Klaus Elhardt, Rudolf Bayer, " Integrating the UB-Tree into a Database System Kernel", in Proc. VLDB 2000.

[9] H. Samet, "Object-Based and Image-Based Object Representations", ACM Computing Surveys, Vol. 36, No. 2, June 2004, pp. 159–217.

[10] G. Salton, A. Wong, C. S. Yang, "A Vector Space Model for Automatic Indexing", Communications of the ACM, Vol. 18, No. 11, November 1975, pp. 613–620.

[11]R. Elmasri, S. Navathe, "Fundamentals of Database Systems", fourth edition, Addison-Wesley 2004.

[12] D. A. White, R. Jain, "Similarity Indexing with the SS-tree", Proc. ICDE 1996, pp. 516–523, Washington 1996.