Society for Science and Education United Kingdom
TMLAI Transactions on Machine Learning and Artificial Intelligence
Volume 7 No 2
ISSN 2054-7390

A New Indexing Method for Uncertain Databases

Guang-Ho Cha
Department of Computer Engineering, Seoul National University of Science and Technology, Republic of Korea
ghcha@seoultech.ac.kr

ABSTRACT

This paper presents an indexing method called the uncertain data index (UD-index) for uncertain databases. The design objectives of the UD-index are improving the range query performance of the multidimensional indexing methods and providing a compromise between optimal index node clustering. Although more than ten years of database research has resulted in a great variety of multidimensional indexing methods, most efforts have focused on the data-level clustering and there has been no attempt to cluster index nodes themselves in dynamic environments. As a result, most related index nodes are widely scattered on the disk due to dynamic page allocation, and it requires many random disk accesses during the range search. The UD-index avoids that by storing the related nodes contiguously in a segment that contains a sequence of contiguous disk pages. The UD-index improves the range query performance by offering high-performance sequential disk access within a segment. A new cost model is introduced to estimate the range query performance. It takes into consideration the physical adjacency of pages read as well as the number of pages accessed. The analytic performance analysis indicates that the UD-index shows better performance than the traditional indexing methods in most cases.

Keywords: Uncertain database; Multidimensional index; Range query; Index clustering; Sequential disk access.

1 Introduction

More than ten years of database research have resulted in a great variety of multidimensional indexing methods to support range queries, for example, R*-tree [1], X-tree [3], M-tree [5], R-tree [11], LSD-tree [12], TV-tree [15], grid file [18], K-D-B-tree [20], buddy-tree [21], and so on. A recent comprehensive survey on the multidimensional indexing methods can be found in [9]. Several characteristics are common to the existing multidimensional indexing methods:

- They have nodes whose size is a page,
- The size of a page is relatively small, e.g., 4 KB,
- They are dynamic,
- They do not exploit the clustering of index nodes,
- They use disk pages as the clustering unit and do not take into consideration the physical adjacency of individual pages.

The traditional page-based index structures do not satisfy the requirements for the multidimensional long range retrievals. The reasons are as follows: (1) Traditional 4 KB index pages are too small to handle the

DOI: 10.14738/tmlai.72.6234
Publication Date: 17th Mar, 2019
URL: http://dx.doi.org/10.14738/tmlai.72.6234

multidimensional long range retrievals efficiently. Our performance analysis and experimental results will justify this statement. Recent article by Gray and Graefe [10] indicated that the range of 8 KB to 32 KB are preferable to smaller and larger index page sizes for current devices. (2) However, simple larger index pages may consume much disk bandwidth. (3) They have to access many index pages randomly because the index pages are widely scattered on the disk due to dynamic page allocation. (4) To avoid the performance degradation due to many random disk accesses, the related index nodes have to be clustered. However, existing indexing methods do not take into account the index clustering. They take into consideration only the clustering (or partition-ing) of data objects. Moreover, the dynamic index clustering requires the on-line index reorganizations, and the overhead of the global index reorganization is excessive. (5) They measure the search performance by the average number of disk page accesses, and do not take into consideration the physical adjacency of individual pages.

To overcome the drawbacks of the existing multidimensional indexing methods, we propose the segmented page indexing (UD-index) technique. The UD-index is based on the concept of segments. The UD-index considers the disk to be partitioned into a collection of segments. Each segment consists of a set of L contiguous pages on disk. A segment is the unit of clustering in the UD-index. Thus all disk pages in a segment can be read by a single disk sweep, and thus it saves much disk startup and seek time. In the UD-index, all disk pages are addressed by a pair of (segment no, page no). This address-ing scheme means that we can access disk in page unit as well as in segment unit. When random accesses are required or when query ranges are very small, page-based disk ac-cesses can be used instead of segment-based accesses.

2 Related Work

The concept of the segment is similar to the idea of the multi-page block used in the SB-tree [19] and the bounded disorder (BD) access method [16, 17], which are variants of the B-trees, in the sense that they accommodate a set of contiguous pages and support multi-page disk accesses. However, this concept has not been applied to the multidimensional indexing methods because it might consume the disk bandwidth excessively with increas-ing dimensionality. As an instance, let us suppose that a query range overlaps only a half on each dimension of the data region occupied by a segment. Then the wasteness of the disk bandwidth caused by reading a segment instead of reading individual pages is ½ (= 1 - ½) in one-dimensional case, while it is $1 - (½)^d$ in d-dimensional case. In fact, however, the multi-page disk reads such as segment reads are more needed in high dimensions be-cause the probability that the query range overlaps with the regions covered by the index nodes increases with the dimensionality due to the sparsity of the domain space, and thus more disk pages are required to be read in higher dimensions. In addition, unlike the multi-page blocks used in the B-trees in which all index nodes as well as all data objects have total ordering among themselves, the index nodes within segments for multidimensional indexing methods have no linear order among them. This makes the design and mainte-nance, such as partitioning and merging, of the segments in the multidimensional indexing method more difficult than those of the multi-page block in the B-trees.

The concept of segments has also some similarity to supernodes of the X-tree [3]. The supernodes are extended nodes over the usual page size, and thus the read of a large su-pernode at a time can be performed. In contrast to the segment which consists of smaller pages, the supernode is a larger node with variable size designed to avoid splits in the in-ternal nodes. Thus, in the X-tree, larger supernodes are always read regardless of the exact match query or the range query, while, in the UD-index scheme, segments or pages can be read selectively depending on the query type. Additionally, the supernodes are applied only to the internal nodes of the index tree in order to maintain efficient internal index structures.

With respect to the index clustering, the UD-index also has some similarity to the bulk loading of multidimensional indexes [4]. However, in contrast to the UD-index which is a generic dynamic index structure creation method, the bulk loading is applied to the creation of initial index structure. In other words, the bulk loading assumes the initially empty index structure but the UD-index can be used dynamically in any time of the in-dex creation.

3 The UD-index

3.1 The Structure of the UD-index

The UD-index considers the disk to be partitioned as a collection of segments. Segments are separated into index segments and data segments. The index segments accommodate in-ternal nodes of the index tree and the data segments hold the leaf nodes. The reason why we separate the segments into two kinds is two folds: it simplifies the design of the index structure and it encourages the upper part of the index structure to reside in the main memory when we cache the index into the main memory. We call the index segment i-segment and the data segment d-segment.

Each segment consists of a set of L contiguous pages on disk (which can be read or written with a single sweep of the disk arm). In our implementation and experiments, we took L = 16. From the first page encountered by the disk head in reading and writing a segment to the last one in the segment, the pages are numbered 1, 2, …, L. A segment has the following properties:

- A segment consists of a set of L nodes which reside on contiguous pages on disk. The number L is called the fanout of a segment.
- K, $1 \leq K \leq L$, nodes falling in a segment are filled contiguously from the beginning of the segment.Re-rank the search results according to our algorithm.
- The UD-index reads K nodes from a segment at a time rather than all L nodes, and it saves the bandwidth.
- Every node of the UD-index sits on segments.
- Leaf nodes reside in the d-segments and internal nodes are in the i-segments.

3.2 Building the UD-index

The UD-index has a hierarchical node structure. As usual for index structures which support spatial accesses for point data, the UD-index divides the data space into pairwise disjoint data cells. With every data cell a data page is associated, which stores all objects contained in the cell. In this context, we call a data cell a directory region.

When the first entry of an UD-index is inserted, a single page of the d-segment is allocated for the first node of the UD-index. This node is a root node as well as a leaf node. We assume that the range of each dimension is 0 to 100, and a pair of numbers on the directory regions indi-cates (d-segment number, page number). Successive entries are added to this node until an insert forces a split in the root node. This node is then split into two leaf node pages which occupy page 1 and page 2 of the d-segment 1. An i-segment is allocated and the first page of the i-segment is assigned for new root node. The root node now contains a single separator and two pointers. A separator contains the information about the split dimension and the split position in the dimension. The split is performed in dimension 1 at position 60. With subsequent insertions, overflows are occurred in the d-segment 1 and they cause the node split. Whenever a node split occurs, the UD-index looks for an empty page on the segment containing the node receiving the insert. This will be the page number K+1 in the containing segment, where K nodes already exist. We keep the infor-mation in the index header which tells us how many pages are occupied in each segment. If an empty page exists, we place the new node created by the split on that page. If there is no empty page in the segment, then a segment split is necessary. A new segment S is allocated, and the overfull segment R containing the splitting node is read into the memory. Then the L+1 nodes of the segment are distributed into two segments.

3.3 Segment Split Strategy

An important part of the insertion algorithm of the UD-index is the segment split strategy which determines the split dimension and the split position. First, the UD-index finds the internal node u which (directly or indirectly) plays a role of root for the overfull segment R. The separator of the internal node u has the dimension and the position to split the segment R. For example, if a new entry is inserted into the page (1, 1) and it causes the d-segment 1 to overflow, the UD-index finds the internal node which plays a role of root of the d-segment 1. Since the UD-index maintains an array to save the traversal path from the root of the UD-index to the target page where a new entry to be inserted, it is not difficult to find the internal node that plays a role of root for the overfull segment. Starting from the root of the UD-index, we check if the overfull segment can be split into two when we apply the separator (split dimension and split position) of the current internal node to split the segment. If the segment can be split using the separator, the corresponding internal node is selected as the root node of the overfull segment, and the segment is split. The data pages belonging to the right children of u are reallocated to the front positions of a new segment S, and the remaining pages are moved forward so that they fall on the front pages of the segment R. As a result of this segment split strategy, the data pages under the same internal node are collected in the same segment.

4 Cost Model for the UD-Index

One common characteristic of traditional indexing schemes is that the average number of disk pages accessed is used as a performance estimator, and the design goal is to minimize it. These schemes assume that each page access takes one disk I/O, and the total access time is measured by multiplying the number of accessed pages by the average access time per page. Hence, they do not consider the actual performance based on the relative positions of accessed pages, which may result in non-uniform access time for individual pages. In this section, we present a cost model to estimate the performance of the UD-index. The cost model distinguishes whether the pages accessed were stored sequentially or randomly. Table 1 gives the summary of symbols and their definitions used in this section.

4.1 Search Cost

In our cost model, we assume that a segment needs one disk I/O by multi-page read. The search cost ($C_{\boldsymbol{q}}$) for processing the query ***q*** can be estimated by

$$C_{\boldsymbol{q}} = \sum_{j=1}^{m} C_j \tag{1}$$

where *m* is the number of disk accesses and C_j is the cost of the *j*-th disk access. A simple model for the cost C_j of a single disk access is given as follows:

Disk Access Cost = Disk Access Time + Transfer Size / Disk Transfer Rate

Table 1. Average over 6 types of range queries.

Symbols	Definitions
$\boldsymbol{q}$	range query with sides $\boldsymbol{q}$
q_i	length of the query region in the i-th dimension
$E(\boldsymbol{q})$	average number of nodes accessed by range query $\boldsymbol{q}$
N	number of objects in the database
n	number of pages (or segments) in the index tree
n_l	number of leaf nodes in the index tree
n_s	number of segments in the index tree
d	dimensionality of the domain space
x_{ij}	length of the directory region of node j in the i-th dimension
f	fanout of a page
S_p	size of a page
S_s	size of a segment
S_e	size of a data entry
U_p	storage utilization of a page
U_s	storage utilization of a segment
$C_{\boldsymbol{q}}$	search cost ($C_{\boldsymbol{q}}$) for the range query $\boldsymbol{q}$
C_s	access cost for a segment
DA_r	number of disk accesses needed to locate a specific entry on
DA_i	the leaf level

Let ΔA be the average disk access time (seek time + latency time), ΔS_j be the transfer size for the j-th disk access, and ΔR be the disk transfer rate. Then, Equation (1) would be

$$C_{\boldsymbol{q}} = \sum_{j=1}^{m} (\Delta A + \frac{\Delta S_j}{\Delta R}) \tag{2}$$

According to today's device technology [10], reading a 2 KB page from a disk with 10 *ms* average disk access time and 10 MB/*sec* transfer rate incurs 10.2 *ms* of disk access cost. Using these representative values, we set ΔA = 10 *ms* and ΔR = 10 KB/*ms*. Then, the search cost $C_{\boldsymbol{q}}$ in *ms* is given by

$$C_{\boldsymbol{q}} = \sum_{j=1}^{m} (10 + 0.1 \cdot \Delta S_j) \tag{3}$$

where the size unit of ΔS_j is 1KB and m is the number of (random) disk accesses.

The expected number $E(\boldsymbol{q})$ of disk pages accessed by the range query $\boldsymbol{q}$ is usually modeled by means of the so-called Minkowski sum which transforms the range query into an equivalent point query by enlarging the directory regions of the page accordingly [2, 7]. To determine the probability that the directory region of a page intersects the query region, we consider the portion of the data space in which the center point of the query must be located such that query and directory region intersect. Therefore we move the center point of the query to each point of the data space marking the positions where the query rectangle intersects the directory region (see Figure 1). The resulting set of marked positions is called the Minkowski sum which is the original directory region having all sides enlarged by the query side

$$E(\mathbf{q}) = \sum_{j=1}^{n} \prod_{i=1}^{d} (x_{ij} + q_i) \tag{4}$$

length q_i, $1 \le i \le d$, d is the dimensionality of the domain space. Taking into account that the volume of the data space is 1, the Minkowski sum directly corresponds to the intersection probability. Then the expected number $E(\boldsymbol{q})$ of disk pages accessed by the range query $\boldsymbol{q}$ on the d-dimensional unit domain space is given by the following formula

where x_{ij} is the length of the directory region of node j in the i-th dimension, q_i is the length of the query region in the i-th dimension, and n is the number of pages (or segments) in the index tree. Thus, the search cost $C_{\boldsymbol{q}}$ for range query $\boldsymbol{q}$ is given by Equation (1), and m in Equation (1) is determined by the Equation (4).

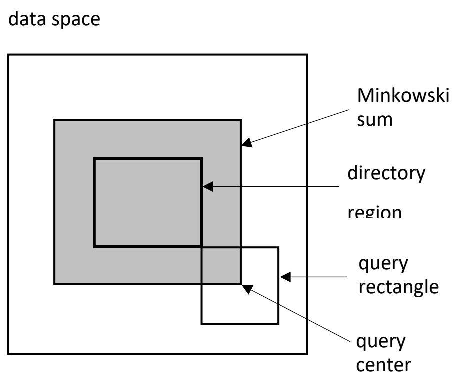

Figure 1. Minkowski sum

4.2 Insertion Cost

Insertion of a new entry into a multidimensional index tree entails, at a minimum, searching from the root for the proper leaf location, placing the new entry in the leaf node, and writing back the modified leaf node. Thus, the number of disk accesses is at least $DA_r + 1$, where DA_r is number of disk accesses needed to locate a specific entry on the leaf level. We can assume that the leaf node may have no room for the new entry, and a split occurs with probability P_{split}. In this case, we must move about half of the entries of the splitting node into a new buffer page, and write the newly created leaf node and the overflowed node with the remained half of the entries out to a disk (two page write). In addition, we must write the parent node out with a new separator (a page write). At this point, the expected number of disk accesses DA_i for an insertion is $DA_r + 1 + P_{split} \cdot 3$. Now when the new separator for the split is inserted in the parent index node, this results in a split in the parent node with probability P_{psplit}. We ignore the event of higher level node splits since the probability P_{psplit} is small in most real situations. Then total number of disk accesses DA_i for an insertion in the conventional multidimensional index tree is

$$DA_i(conv) = DA_r + 1 + P_{split} \cdot 3 \tag{5}$$

Now consider the case of an insert to an UD-index. When a page split occurs, we may find no page on the containing segment to take the new leaf node. Thus, with probability P_{ssplit}, we have to read in the entire segment (L), allocate a new segment, move about a half ($L/2$) of nodes to it and write it out, and then move forward about a half ($L/2$) of nodes remaining in the overfull segment and write back the segment to the disk. We also write the parent node out with a new separator (a page write). We note that the contribution to the cost of insert as a result of a leaf level segment split occurs only with probability $P_{split} \cdot P_{ssplit}$. We also ignore the event of higher level segment split since the probability $P_{split} \cdot P_{psplit} \cdot P_{ssplit}$ is extremely low. Then total number of disk accesses DA_i for an insertion in the UD-index is

$$DA_i(UD) = DA_r + 1 + P_{split}(1 + P_{ssplit} \cdot (L + 2 \cdot L/2)) \tag{6}$$

Taking some typical values of index page fanout f_1 = 300, segment fanout f_2 = 16 (i.e., L = 16), and 70% storage utilization for each page and segment, we have the probability P_{split} is about $1/(300 \cdot 0.7) \approx 0.0048$ and the probability P_{ssplit} is about $1/(16 \cdot 0.7) \approx 0.0893$. Thus, $DA_i(conv)$ is $DA_r + 1.0144$ and $DA_i(UD)$ is $DA_r + 1.0185$. In the estimate of disk accesses for an insert, the number of additional disk accesses due to the UD-index is only 0.0041, an insignificant addition. This result shows that the update cost of the UD-index is comparable to the conventional indexing methods.

4.3 Segment Size

The size of a segment determines its retrieval cost and fanout (number of pages per segment). We choose the segment size which minimizes the search cost C_q in Equation (3). Using Equation (3) with m = 1, single segment access costs for various segment sizes can be computed as shown in Table 2.

Table 2. Single segment access cost on various segment size

Segment Size (*KB*)	4	8	16	32	64	128
Segment Access Cost (*ms*)	10.4	10.8	11.6	13.2	16.4	22.8

Now let's look at the range query costs using various sizes of segments based on our experimental database. Consider a database with following characteristics:

$$d = 2,\ N = 1000000,\ S_e = 12 \text{ bytes},\ S_p = 4 \text{ KB},\ S_s = 64 \text{ KB},\ U_p = 70\%,\ U_s = 70\%.$$

Then a 4 KB index page will contain

$$238 \text{ entries } (= (4 \times 1024 \times 0.7)/12), \text{ i.e., } f = 238.$$

The number of leaf nodes in the index tree is

$$4202\ (= 1000000/238), \text{ i.e., } n_l = 4202.$$

We only consider leaf nodes because nonleaf nodes are less than 1% of the total number of nodes in the index tree in most cases. Then the number of segments in the index tree is

$$376\ (= 4202/(16 \times 0.7)).$$

The area covered by a single segment in a unit domain space is

$$(1/376)^{1/2} = 0.0516$$

Given a range query $\boldsymbol{q}$ with the area of 1% of the whole domain space, the length of each side of the query window is 0.1. Then the expected number $E(\boldsymbol{q})$ of segments accessed by the range query $\boldsymbol{q}$ is

$$376 \cdot (0.0516 + 0.1)^2 = 8.64.$$

Thus, from Equation (3), the expected cost to process a range query with 1% range of whole data space is

$$38.71\ ms\ (= 8.64 \times (10ms + 64 \times 0.7 \times 0.1ms)).$$

Table 3 summarizes the range query processing costs for 4 KB page and various sizes of a segment from 4 KB to 256 KB. The query range varies from 0.01% to 10% of the whole domain space. The 0.01%, 0.1%, 1%, and 10% shown in the first row of each Table denote the sizes of the query range. The shape of a query is assumed to be a square, i.e., the length of the query window in each dimension is same. The q_i in the third column in each Table denotes the length of the query window in the i-th dimension. Since Gray and Graefe [10] indicated that 16 KB is a good size for index pages, we also computed the range query costs for a 16 KB page and various sizes of a segment, and summarized them in Table 4.

Table 3. Range query processing cost for 4KB page size and various segment sizes

S_s (KB)	**n_s**	**q_i**	**C_s**	**$E(q)$ 0.01%**	**$E(q)$ 0.1%**	**$E(q)$ 1%**	**$E(q)$ 10%**	**C_q 0.01%**	**C_q 0.1%**	**C_q 1%**	**C_q 10%**
4	4202	0.0154	10.40	2.71	9.28	55.96	462.05	28.2	96.5	582.0	4805.2
8	3002	0.0183	10.56	2.40	7.48	42.01	335.89	25.3	79.0	443.6	3547.0
16	1501	0.0258	11.12	1.92	4.95	23.75	175.56	21.4	55.0	264.1	1952.2
32	751	0.0365	12.24	1.62	3.48	13.99	93.42	**19.8**	42.6	171.2	1143.5
64	376	0.0516	14.48	1.43	2.60	8.64	50.86	20.7	**37.6**	125.1	736.5
128	188	0.0730	18.96	1.30	2.06	5.63	28.48	24.6	39.1	**106.7**	540.0
256	94	0.1031	27.92	1.20	1.71	3.88	16.53	33.5	47.7	108.3	**461.5**

Table 4. Range query processing cost for 16KB page size and various segment sizes

S_s (KB)	**n_s**	**q_i**	**C_s**	**$E(q)$ 0.01%**	**$E(q)$ 0.1%**	**$E(q)$ 1%**	**$E(q)$ 10%**	**C_q 0.01%**	**C_q 0.1%**	**C_q 1%**	**C_q 10%**
16	1047	0.0309	11.6	1.75	4.09	17.94	126.14	20.3	47.4	208.1	1463.2
32	748	0.0366	12.24	1.62	3.48	13.96	93.10	**19.8**	42.6	170.9	1139.5
64	374	0.0517	14.48	1.42	2.60	8.61	50.62	20.6	**37.6**	124.7	733.0
128	187	0.0731	18.96	1.29	2.05	5.60	28.34	24.5	38.9	**106.2**	537.3
256	94	0.1031	27.92	1.20	1.71	3.88	16.53	33.5	47.7	108.3	**461.5**

Considering only the index page size, the performance of the indexing using 16 KB pages is better than that of the indexing using small 4 KB pages in all example cases. This result is consistent with the statement of the Gray and Graefe. Comparing the segment-based indexing with the page-based indexing, the performance of the segment-based indexing is far better than that of the page-based indexing in all example cases. The indexing with S_s = 4KB in Table 3 and S_s = 16 KB in Table 4 are in fact the page-based indexing with those sizes of pages.

When we consider an optimal segment size, it differs according to the size of the query range. For example, while a 64 KB segment is the best when the query range is 0.1% of the whole data space, a 256 KB segment is the best when the query range is 10% of the whole data space.

In summary, segment sizes in the range of 32 KB to 256 KB are preferable to smaller and larger segment sizes. However, since disks are predicted to have larger transfer rates as the device technology advances, larger segment (> 32 KB) will be preferable.

5 Conclusion

We have introduced the UD-indexing method and the cost model for range queries and insertions in multidimensional data spaces. Both the theoretical performance analysis and the experimental results demonstrate that in most cases the UD-indexing is superior to traditional multidimensional indexing methods for range queries. For random queries such as exact-match queries and k-nearest neighbor queries, there is almost no performance difference between the UD-index and the LSD^h-tree when the page-based access is used. In addition, the performance degradation for updates in the UD-index is negligible. The superiority of the UD-indexing increases greatly as the dimensionality of the domain space in-creases and the query range grows. High dimensionality and long range retrievals are quite common in today's environments such as multimedia databases. The performance ad-vantage of the UD-indexing to traditional indexing methods was revealed up to several times in experiments depending on the data set and the size of range queries. In addition, it has been demonstrated that using larger pages (e.g., 16 KB) is more efficient for range queries than using traditional smaller pages (e.g., 4 KB). The performance advantage of the UD-indexing comes from saving much disk startup time. Moreover, storing a sequence of index pages contiguously within a segment provides a compromise between optimal index node clustering and the excessive full index reorganization overhead. Thus, the UD-indexing methods may be used as an alternative index clustering scheme. The UD-indexing is so generic that it can be applied to any multidimensional indexing methods.

ACKNOWLEDGEMENTS

This research was financially supported by Seoul National University of Science and Technology (2018-0237).

REFERENCES

[1]. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An efficient and robust access method for points and rectangles," *Proc. of ACM SIGMOD International Conference on Management of Data*, pp. 322-331, 1990.

[2]. S. Berchtold, C. Boehm, D.A. Keim, and H.-P. Kriegel, "A Cost Model for Nearest Neighbor Search in High-Dimensional Data Space," *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database System*, pp. 78-86, 1997.

[3]. S. Berchtold, D.A. Keim, H.-P. Kriegel, "The X-tree: An Index Structure for High-Dimensional Data," *Proc. of the 22nd International Conference on Very Large Data Bases*, pp. 28-39, 1996.

[4]. J.v.d. Bercken, B. Seeger, and P. Widmayer, "A Generic Approach to Bulk Loading Multidimensional Index Structures," *Proc. of the International Conference on Very Large Data Bases*, pp. 406-415, 1997.

[5]. P. Ciaccia, M. Patella, and P. Zezula, "M-tree: An Efficient Access Method for Similarity Search in Metric Space," *Proc. of the International Conference on Very Large Data Bases*, pp. 426-435, 1997.

[6]. D. Comer, "The Ubiquitous B-tree," *ACM Computing Surveys*, Vol. 11, No. 2, pp. 121-137, 1979.

[7]. C. Faloutsos and I. Kamel, "Beyond Uniformity and Independence: Analysis of R-trees Using the Concept of Fractal Dimension," *Proc. of SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 4-13, 1994.

[8]. M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, and P. Yanker, "Query by image and video content: the QBIC system," *IEEE Computer*, Vol. 28, pp. 23-32, Sep. 1995.

[9]. V. Gaede and O. Gunther, "Multidimensional Access Methods," *ACM Computing Surveys*, Vol. 30, No. 2, pp. 170-231, June 1998.

[10]. J. Gray and G. Graefe, "The Five-Minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb," *ACM SIGMOD Record*, Vol 26, No. 4, pp. 63-68, Dec. 1997.

[11]. A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. of the ACM SIGMOD International Conference on Management of Data*, pp. 47-57, 1984.

[12]. A. Henrich, "The LSD^h-tree: An Access Structure for Feature Vectors," *Proc. of the 14th International Conference on Data Engineering*, pp. 362-369, 1998.

[13]. D. Knuth, The Art of Computer Programming, vol. 3: Sorting and Searching, Addison Wesley, Reading, MA, 1973.

[14]. J.-H. Lee, D.-H. Kim, and C.-W. Chung, "Multidimensional Selectivity Estimation Using Compressed Histogram Information," *Proc. of the ACM SIGMOD International Conference on Management of Data*, pp. 205-214, 1999.

[15]. K.-I. Lin, H.V. Jagadish, and C. Faloutsos, "The TV-tree: An Index Structure for High-Dimensional Data," *VLDB Journal*, Vol. 3, No. 4, pp. 517-542, 1994.

[16]. W. Litwin and D.B. Lomet, "The Bounded Disorder Access Method," *Proc. of the IEEE International Conference on Data Engineering*, pp. 38-48, 1986.

[17]. D.B. Lomet, "A Simple Bounded Disorder File Organization with Good Performance," *ACM Transactions on Database Systems*, Vol. 13, No. 4, pp. 525-551, Dec. 1988.

[18]. J. Nievergelt, H. Hinterberger, and K.C. Sevcik, "The grid file: an adaptable, symmetric multikey file structure," *ACM Transactions on Database Systems*, Vol. 9, No.1, pp. 38-71, 1984.

[19]. P.E. O'Neil, "The SB-tree: An Index-Sequential Structure for High-Performance Sequential Access," *Acta Informatica*, Vol. 29, pp. 241-265, 1992.

[20]. J.T. Robinson, "The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes," *Proc. of the ACM SIGMOD International Conference on Management of Data*, pp. 10-18, 1981.

[21]. B. Seeger and H.-P. Kriegel, "The Buddy-tree: An Efficient and Robust Access Method for Spatial Data Base Systems," *Proc. of the 16th International Conference on Very Large Data Bases*, pp. 590-601, 1990.