Construction of Tree-based Indexes for Level-Contiguous Buffering Support

Tomáš Skopal, David Hoksza, Jaroslav Pokorný

Charles University in Prague, FMP, Department of Software Engineering Malostranské nám. 25, 118 00 Prague, Czech Republic {tomas.skopal, david.hoksza, jaroslav.pokorny}@mff.cuni.cz

Abstract. In multimedia databases, the spatial index structures based on trees (like R-tree, M-tree) have been proved to be efficient and scalable for low-dimensional data retrieval. However, if the data dimensionality is too high, the hierarchy of nested regions (represented by the tree nodes) becomes spatially indistinct. Hence, the query processing deteriorates to inefficient index traversal (in terms of random-access I/O costs) and in such case the tree-based indexes are less efficient than the sequential search. This is mainly due to repeated access to many nodes at the top levels of the tree. In this paper we propose a modified storage layout of tree-based indexes, such that nodes belonging to the same tree level are stored together. Such level-ordered storage allows to prefetch several top levels of the tree to the buffer pool by only a few or even a single contiguous I/O operation (i.e. one-seek read). The experimental results show that our approach can speedup the tree-based search significantly.

1 Introduction

The research in database indexing remains still a hot topic – its importance even increases with the emergence of new data types like multimedia data, time series, DNA sequences, etc. For such data, the tree-based indexes are often employed, e.g. the R-tree, X-tree, M-tree, and many others [1, 4]. While there have arisen new criteria for retrieval efficiency, the I/O costs still represent the major efficiency component, especially in multidimensional databases. Simultaneously, the complexity of the new data makes them hardly indexable by tree-based structures, so in many papers the sequential search is referred to perform better (in terms of I/O costs) than any tree-based index [16].

1.1 Problem Analysis

We start with a bit of history on magnetic hard disk drives. Despite the recent boom of new storage media – e.g. flash or hybrid disks – the best (and cheapest) medium for storage and indexing is still the magnetic hard disk drive (HDD) with rotating platters and moving heads. Due to its construction, the I/O efficiency of HDD depends on *access time* and *transfer rate*. The access time is determined by the *seek time* (head moves to a track), *settle time* (precise head positioning) and the *latency* (or rotational delay). The transfer rate is given by megabytes per second of sequentially (contiguously) read/written data from/to a track. While HDD capacity doubles every year and transfer rate increases by 40%, the access time improves only by 8% (because of kinetic limitations of heads). Todays HDD can be of 300GB capacity, 50MB/s transfer rate and 10ms access time. With 8KB disk blocks (or pages) used by file systems, the fetching of a block takes 10.16ms, so the access takes 98.5% of the total time. A contiguous fetch of 800KB data takes only 2.5x the time needed for fetching 8KB data. However, some two decades ago the HDDs exhibited different parameters, the access time about 100ms and the transfer rate at about 150KB/s. Thus, a random access to a disk block is relatively more expensive nowadays than some 20 years ago.

Sequential vs. Tree-based Indexing. The classic access methods have been developed based on a traditional disk model that comes with simplifying assumptions such as an average seek-time and a single data transfer rate. An excellent overview of these problems can be found in [15]. The B-tree or R-tree structures were developed in times of relatively cheap access costs (compared to the transfer rates). The tree node size (mapped to a block) was 2 or 4KB, while sequential reading of large data from HDD was not much more expensive than reading the data by multiple random-access retrievals, e.g. 7s vs. 32s in case of 1MB of data and 4KB blocks. By query processing, a traversal of 1/5 (or less) of the tree index sufficed to be faster than the simple sequential search. Today, the tree-based querying must traverse less than 1/86 to overtake the sequential search. Such a small proportion is achieved by B⁺-tree, or R-tree built on lowdimensional data. However, complex data cannot be retrieved in such an efficient way, because of their high dimensionality. Therefore, in modern applications the sequential search (or sequential-based indexes like VA-file [16]) is reported as more efficient (in terms of I/O costs) than indexing by tree-based structures.

How Large the Tree Nodes Should be? One can ask whether the access times could be reduced simply by enlarging the tree nodes from few (tens) of kilobytes to hundreds or even thousands. Then the number of nodes would be much smaller and so the number of disk accesses would decrease. Here the problem is in the increased number of entries stored in the node (the node capacity). Actually, this is possible for B-tree, but not for structures like R-tree or M-tree. Unlike the B-tree where the node split operation is of linear complexity with the number of entries, in R-tree or M-tree the complexity of node split is super-linear because of (sub)optimal partitioning of entries. A high node capacity also leads to worse approximations (e.g. MBRs in case of R-tree) in the parent node.

Second, although in B-tree the search in a single large node is fast because of use of interval halving, this is not possible in R-tree or M-tree where no universal ordering of entries is guaranteed. This has not to be critical in case of lowdimensional R-tree where the tuples-in-query testing is fast, however, in M-tree the sequential search within a node implies expensive distance computations.

1.2 Paper Contributions

In this paper we use level-separated buffering scheme which leads to more effective buffer utilization. Moreover, we introduce a modified split algorithm which keeps the tree index level-contiguous, that is, nodes belonging to a certain level in the tree are stored together. Such a modified index file layout allows to cheaply prefetch the top levels of the tree and thus further decrease the access costs.

2 Tree-based Indexing

In this section we briefly summarize the structural properties of tree-based indexes and their secondary storage, including buffering into main memory.

First of all, we assume "region-based" redundant trees, where the data objects are stored in the leaves, while each entry in an inner node represents a (spatial) approximation of the appropriate subtree, e.g. an MBR in case of R-tree, or hyper-sphere in case of M-tree. We also assume an inner node with m entries (regions) has m children (see Figure 1a). Such assumptions are satisfied by R-tree, M-tree, but not by the B-tree (which is not region-based).



Fig. 1. (a) Insert into leaf G_0 . (b) The resulting tree, split up to the root.

We subscript each node by a number of its level (*level number*), starting by 0 at the leaf level (see Figure 1). Since indexes grow from bottom to top, a node's level number does not change. Besides the level number, each node obtains an identifier. A node is stored at address (or offset) in index file which is the identifier times node size (we omit the header of the index). The node size is assumed to be the same for leaves and inner nodes and is given in kilobytes.

Inserting & Splitting. By standard insertion, a leaf is found into which a new object is inserted. An overflowed leaf must be split between two leaves, one keeping the old identifier, and a brand new leaf. The two new entries describing two subtrees are inserted into the parent node (one entry is just updated). When the parent node overflows, the splitting is repeated (possibly up to the root level). The insertion with splits is presented in Figure 1. First, the leaf G_0 is chosen for insertion. The insertion raises a sequence of node splits (including the root).

Model Structure vs. Index File Layout. Note that the sequential ordering of nodes in the index file (physical view in Figure 1a) does not preserve the structure (the model view). This is because the new allocated nodes at the end of the index file come from different tree levels after a sequence of splits. In the optimal situation, the physical ordering exactly follows the model ordering given by breadth-first traversal of the tree. With such an organized index file we would be able to prefetch the neighboring nodes by a single contiguous read. Unfortunately, the standard splitting strategy cannot preserve the physical ordering of nodes in accordance with the model, because this would imply O(n) insertion complexity (shifting many nodes), which is impracticable in most cases.

2.1 Standard Buffering & Prefetching

Like other database structures, also indexes use buffering [6] of blocks into memory frames. When a node is requested, its block is fetched from the disk and stored in a frame of the buffer pool. If the same node is requested later, there is a probability that it is still in the buffer, so we avoid an I/O operation. Since the buffer is smaller than the total volume of requested nodes, there must be a *replacement policy* used, which chooses a node from the pool that will be released in order to make space for a requested node. There have been many policies developed [11], like LRU (least recently used), MRU (most recently used), FIFO, Clock, LFU (least frequently used), etc., each suitable for different database conditions.

Because of reasons discussed in Section 1.1, we would like to access a large amount of data in a single contiguous read/write operation. Instead of a single node being requested, we could prefetch several additional nodes from the disk. Such prefetching is actually provided by the HW cache of the HDD. Unfortunately, the nodes are stored in the index file in very different order than is the order of nodes in the tree model. Hence, it would be inappropriate to force the prefetched nodes to be stored in the buffer pool, because such bulk-loading of (possibly not utilizable) nodes would lead to release of many nodes from the buffer which are (maybe) more likely to be requested than the prefetched ones.

3 Related Work

Typically, the tree-based indexes follow linear abstraction of HDD provided by file system. The only factor that has to be minimized is the number of randomaccess I/Os, i.e. number of blocks, each fetched from an arbitrary location on the disk [7]. Most efforts in advanced database indexing have been spent on improving filtering abilities with respect to the model (e.g. R-tree vs. X-tree [1] or M-tree vs. PM-tree [14]). Although the filtering improvements have a substantial impact on the overall efficiency (not only on the I/O costs), at some point, further improving of the model is very hard. At that moment some lowerlevel techniques have to be investigated, related to HW and data storage issues.

3.1 Buffering Techniques

The I/O costs can be substantially reduced by using appropriate buffering strategies. The classic work on index buffering [10] suggests the LRU replacement policy for B⁺-tree as the most effective. Also for multidimensional indexes the LRU policy has been proved as effective, we refer to [5] where an impact of buffering on closest-pair queries in R-tree is discussed.

3.2 Dynamic Layout Rearrangement

A general approach to speedup data retrieval from HDD is the dynamic rearrangement of the storage layout [2, 9]. The main idea follows the assumption that some access patterns are more frequent than other ones, so blocks belonging to the same pattern should be stored together in order to minimize necessary movements of disk heads. The organ-pipe arrangement [12] is an example of such a layout. The rearrangement (also called *shuffling*[12]) resembles file defragmentation for a specific access pattern, where the frequently accessed blocks are moved together during data retrieval with a hope this access pattern will occur again in the future. Although the dynamic rearrangement is a universal method for data management, its usage in database indexing is limited due to the absence of strong access patterns. Even if there exists an access pattern for a particular user, a multi-user access to the index will spoil the efforts spent by rearrangement because of many contradictory access patterns.

In our approach we use a kind of layout rearrangement, however, this one is performed during the construction of the index (i.e. not during query processing).

3.3 Physical Designs

Some recent works leave the linear abstraction of HDD and exploit physical properties of modern disks. In recent years, hard disks are manufactured with zoned recording (or zoning), which groups adjacent disk cylinders into zones. Tracks are longer towards the outer portions of a disk platter as compared to the inner portions. Hence, more data can be recorded in the outer tracks when the maximum linear density, i.e., bits per inch, is applied to all tracks. The results are multiple physical zones in a disk, where seek times and data transfer rates vary significantly across the zones. In [17] the authors optimize dynamic multidimensional access methods (\mathbb{R}^* -tree) given a zoned disk model.

Another adjacent block utilization is presented in [13], however, the authors deal with storage of multidimensional data rather than indexing. The key idea is that the HDD is, in fact, a three-dimensional medium where the adjacent tracks (either within a platter or within a cylinder) can be accessed efficiently.

The drawback of these methods is a requirement on specific system-level software, that provides applications with access to adjacent portions on the disk.

4 Level-Contiguous Indexing

Unlike the proposals in Section 3.3, we use the classic linear abstraction of data storage. Furthermore, we focus on indexes where complex queries are issued, i.e. queries where a substantial volume of nodes at the top levels must be processed. Hence, we do not consider point or interval queries on B^+ -tree, since such queries result in simple one-path traversal. In other words, we assume an access pattern where the inner nodes are accessed much more frequently than the leaves. Based on the assumptions, we propose *level-contiguous* storage layout – an index storage partially preserving the model ordering of nodes for only a little construction overhead. In this layout the nodes are physically ordered by their level numbers.

4.1 Index Traversal Analysis

In the B⁺-tree, the most used query types are the point and interval queries defined for single-key domains, where the traversal is guided along a single path in the tree (an interval query must additionally search the leaf level), see Figure 2a. Assuming that the queries are distributed uniformly, the probability that a node at a level of B⁺-tree will be accessed is proportional to the number of nodes at that level, i.e. a leaf has the smallest probability and the root has 100%. However, some tree-based indexes are used for multidimensional or metric data, e.g. R-tree, X-tree, M-tree, where the nodes represent regions in the indexed space. On such data there is no universal ordering defined, and also the query types are more complex. In particular, the R-tree is used for range query (or window query) and the M-tree is often used for (k-)nearest neighbor query.

Since these structures index data which cannot be ordered (with respect to all queries), the tree traversal goes not along a single path. More likely, to reach all relevant data in the leaves, there must be multiple paths passed through (see Figure 2b). The reason is that leaves relevant to a query are not clustered – they are spread over the entire leaf level.



Fig. 2. (a) Point/interval search in B⁺-tree (b) Range/kNN search in R-tree or M-tree.

Since the nodes represent regions in the indexed space, the top-level nodes' regions have large volume (they must cover all children regions, see Figure 3). Then, during a query processing the nodes are checked against a query region and those children are further processed, which overlap the query. Obviously, the larger regions (nodes at the top levels) have greater probability to be accessed. With high-dimensional data, this means almost all top-level nodes are accessed (due to the *curse of dimensionality* [1,3]). Consequently, many random accesses are performed when querying high-dimensional data, so large portions of top levels are searched in randomized order. This is, in turn, often worse than contiguous sequential search of the entire index file.



Fig. 3. Hierarchical space decomposition by (a) R-tree (b) UB-tree (c) M-tree

4.2 Level-Contiguous Index Storage Layout

In our approach, we focus on "derandomization" of the I/O costs so that infrequent large contiguous I/Os are preferred over many random block I/Os. This can be achieved by a modification of index storage layout, in particular by ensuring that nodes are physically ordered by their level numbers (the order of nodes within a single level does not matter). In such a way, we can read all the nodes at several top levels by a single contiguous fetch, and store them into the buffer. The idea makes use of adjusted node splitting. After an object has been inserted such that some node splits occurred, a special algorithm (called SwapUp, see Listing 1) is executed. The algorithm uses an array mLevelStartIndex, where its *i*-th entry stores the index file position of the first node belonging to *i*-th tree level. In principle, the algorithm propagates the new nodes (produced by splitting at the end of index file) in order to restore the ordering defined by level numbers. This is realized by swapping the new (misplaced) nodes with some old nodes which are located at first positions of a particular level in the index file.

Listing 1 (modified insertion algorithm, SwapUp algorithm)

Notes to the code: The SwapTwoNodesAt swaps the nodes defined by their identifiers (positions in index) together with both parent nodes' links pointing to the swapped nodes. To quickly access the parent node, a *parent identifier* must be additionally stored in each node. However, now also the parent identifiers of the child nodes of the two nodes being swapped must be updated. The GetNodesCountAtLevel returns the number of nodes at a given level before the insertion.

The algorithm running is explained in Figure 4a, which is index file layout related to the tree in Figure 1. Before insertion, the storage layout was levelordered (see the white part in Figure 4a-1). After insertion, the multiple splits caused violation of the ordering (see the grey part). The SwapUp algorithm now propagates the new nodes to correct positions. In Figure 4a-1, the new non-leaf nodes are swapped with the first 3 leaf nodes stored in the index. Then, the two remaining nodes are swapped with the first two level-1 nodes (see Figure 4a-2) and finally, the new root node O_3 is swapped with the old root N_2 (Figure 4a-3). The final index layout (let us denote it as *level-ordered index*) is presented in Figure 4a-4, where the top (bottom, resp.) arrows denote which parents (children) had to be updated with the new node's identifier during the swapping-up. **Time Complexity.** Suppose n is the number of objects in the tree (i.e. $O(\log n)$ is the tree height). There is only $O(\log n)$ seeks performed during the swapping, additional seeks are required for updating the links in parent/child nodes. Thus, the worst-case complexity is $O(\log^2 n)$ when measured in block I/Os as well as in seek operations.



Fig. 4. (a) Swapping-up the new nodes after insertion (which caused multiple splits). (b) Top-level and Bottom-level buffer pools.

4.3 Level-Contiguous Buffering

As we have mentioned in Section 4.1, the nodes at top levels are the most frequently accessed ones. It could appear that the LRU (LFU) replacement policy keeps the top-level nodes in the buffer for a long time, since as the recently (frequently) access nodes are considered the top-level nodes. However, when a query is executed, the greatest amount of nodes read belongs to the leaf level and the "valuable" top-level nodes are replaced by the leaves, because these are temporarily the most recently accessed ones.

Divided Buffer. Due to the obstacles caused by original LRU replacement in a single buffer pool, we propose a buffer logically divided in two parts (see Figure 4b). The first part stores a user-defined number of top-level nodes (the *top-level buffer*), while once a node is loaded into top-level buffer, it will never be replaced. The second part behaves as an ordinary LRU-based buffer for the rest of nodes not buffered by top-level buffer (the *bottom-level buffer*).

Buffering the Top Levels. The top-level buffer can be populated either incrementally (by individual fetch requests of nodes) on an ordinary index, or in a batch by prefetching certain volume of the level-ordered index file.

The prefetching itself can be accomplished in two ways. We can prefetch a large portion of the index at the beginning of index usage (the *bulk prefetching*), so that the entire top-level buffer is populated. Or, we can prefetch smaller (yet sufficiently large) portions at the moment when a requested node is still not loaded in the top-level buffer (the *incremental prefetching*). While the bulk variant minimizes the query time over many queries, the incremental one distributes the prefetch load among several queries.

$\mathbf{5}$ **Experimental Results**

To prove the benefits of level-contiguous storage layout, we have performed extensive experimentation with the R-tree and the M-tree. In the former case, the testing platform was a Pentium4@3GHz, 1GB RAM, Maxtor OneTouch, Ultra ATA 133, 250GB@7200 rpm, 8MB disk cache, avg. seek < 9.3ms, transfer rate 34 MB/sec. In the latter case we used Pentium4@3.6GHz, 1GB RAM, Seagate Barracuda ST3200826A, SATA, 200GB@7200 rpm, avg. seek < 8ms, 8MB disk cache, transfer rate 65MB/s. Both platforms were used with Windows XP system with disabled file-system cache (the HDDs' HW caches were enabled for read), while both HDDs involved in tests were not system disks.

In addition to R-tree and M-tree, we have also performed the tests on sequential file to set up a baseline, where for sequential query processing we have used a buffer of equal size as in case of the competitive R/M-trees. Most of the tests were executed for 100 different query objects and the results were averaged.

5.1**R-tree Testbed**

Query selectivity (objects in result, log. scale)

The first set of tests was aimed at indexing large synthetic multidimensional datasets by the R-tree and its level-contiguous modification (denoted as LC index in figures). There were 3 datasets generated, consisting of 3,000,000, 6,000,000, and 10,000,000 5-dimensional tuples. The tuples were distributed uniformly among 700, 800 and 1000 clusters, respectively. In Table 1 see the R-tree index characteristics.



Query selectivity (objects in result, log.scale) Fig. 5. R-tree: Disk accesses and realtimes for increasing query selectivity.

Query selectivity (objects in result, log. scale)

The number of disk accesses for window queries with increasing query selectivity (number of objects in the answer) is presented in Figure 5a. The label TopBuffer=x% denotes a bulk-prefetch index with size of top-level buffer equal to x% of all inner nodes (i.e. TopBuffer=0% means no top-level buffering, while TopBuffer=100% means all inner nodes can be buffered). The bottom-level buffer is maintained in the remaining buffer memory. As we can see, the LC index with TopBuffer=8% outperforms the classic R-tree ("notLC" indexes) as well as LC indexes with different TopLevel values. Note that we have utilized the top-level buffering also in the notLC indexes, however, here the top-level nodes could not be prefetched, they were accessed one-by-one.

In Figure 5b see the realtimes for the same situation. All the LC indexes show almost 100% speedup when compared with notLC indexes. Surprisingly, the LC indexes outperform the notLC indexes even in case that no top-level buffering and prefetching is employed. In Figure 5c the realtimes show behavior of LC/notLC indexes on the 10,000,000 dataset, and in Figure 6a see the disk accesses on the 3,000,000 dataset.

We have also tested the impact of top-level buffering/prefetching with respect to the order of issued queries. In Figures 6b,c see the average realtime costs for queries with selectivity = 2 (5, respectively), according to the order of the query in a query sequence (or query batch). We can observe the benefits of LC indexes do not decrease in time. In Figure 6b the top-level nodes of LC indexes were prefetched incrementally, by 100 nodes, but as we can notice, there is no significant difference between prefetching incrementally or in a bulk (Figure 6c).



Fig. 6. R-tree: Disk accesses for increasing query selectivity and realtimes for typical response of *i*-th query in a query batch.

5.2 M-tree Testbed

Second, we have implemented level-contiguous M-tree [4] and performed experiments with the Corel [8] feature vectors (65,615 images). The dataset consisted of 262,460 8-dimensional vectors, constructed by merging 4 feature representations (color and layout histograms, textures, color moments). The L_1 distance was used to measure image dissimilarity. See M-tree characteristics in Table 2.

In Figure 7a see the realtimes of kNN queries, with respect to increasing k. Although the classic notLC M-tree gets worse than the sequential file already at k = 15 (or k = 20 in case of M-tree with top-level buffering), the LC indexes remain efficient up to k = 50. The impact of query batch size is presented in Figure 7b, where the LC indexes do not deteriorate when compared with notLC indexes, they get even better. We have also examined the influence of top-level buffer proportion in the total buffer memory, see Figure 7c. We can observe that increasing volume of top-level buffer improves the realtimes quite significantly.

statistics.
ta objects: 262,460 (8D vectors)
e capacity: 19 in inners, 29 in leaves
er of nodes: 1188 inners, 13180 leaves
onstr. time: 2.8min
for seq. file: 0.9MB (10%)
M-TREE Size of TopBuffer •••••••• ••••• •••• •••• •••• ••

Fig. 7. M-tree: Realtimes for kNN queries depending on k, size of query batch, and proportion of TopBuffer.

Finally, in Figure 8 see the structure of accesses to nodes in the tree-based indexes. Besides the root node, which must always be accessed, we can see that the nodes at top levels are accessed indeed frequently, especially in case of M-tree. Thus, the arguments for top-node buffering and level-contiguous index layout seem to be well-founded, and we can expect level-contiguous layout could be beneficial also to other tree-based indexes, like X-tree, UB-tree and others.



Fig. 8. Structure of level accesses for queries in R-tree and M-tree.

In summary, the level-contiguous storage layout supports efficient utilization of access patterns usual for tree-based indexes, so that they can exploit the advantage of contiguous disk reading (like sequential search does it). This property dramatically reduces the random-access I/O overhead spent at top tree levels.

6 Conclusions

In this paper we have introduced level-contiguous storage layout for tree-based indexes. The new layout allows to prefetch the frequently accessed nodes at the top levels of any multidimensional or metric tree based on B^+ -tree. Moreover,

we have proposed divided schema for level buffering, where the prefetched toplevel nodes are stored separately and the replacement policies are not applied to them. The experimental results show that the prefetching together with the top-level buffering significantly improves the performance of query processing (up to 200% speedup) at the costs of a moderate increase of construction costs (about 30%).

References

- C. Böhm, S. Berchtold, and D. Keim. Searching in High-Dimensional Spaces Index Structures for Improving the Performance of Multimedia Databases. ACM Computing Surveys, 33(3):322–373, 2001.
- S. D. Carson. A system for adaptive disk rearrangement. Software Practice and Experience (SPE), 20(3):225-242, 1990.
- E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín. Searching in metric spaces. ACM Computing Surveys, 33(3):273–321, 2001.
- P. Ciaccia, M. Patella, and P. Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In VLDB'97, pages 426–435, 1997.
- A. Corral, M. Vassilakopoulos, and Y. Manolopoulos. The Impact of Buffering on Closest Pairs Queries Using R-Trees. In ADBIS '01: Proceedings of the 5th East European Conference on Advances in Databases and Information Systems, pages 41-54, London, UK, 2001. Springer.
- W. Effelsberg and T. Haerder. Principles of database buffer management. ACM Transactions on Database Systems (TODS), 9(4):560–595, 1984.
- V. Gaede and O. Günther. Multidimensional access methods. ACM Computing Surveys, 30(2):170–231, 1998.
- 8. S. Hettich and S. Bay. The UCI KDD archive [http://kdd.ics.uci.edu], 1999.
- H. Huang, W. Hung, and K. G. Shin. FS2: dynamic data replication in free disk space for improving disk performance and energy consumption. In SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles, pages 263–276, New York, NY, USA, 2005. ACM Press.
- L. F. Mackert and G. M. Lohman. Index scans using a finite LRU buffer: a validated I/O model. ACM Transactions on Database Systems (TODS), 14(3):401–424, 1989.
- R. Ramakrishnan and J. Gehrke. Database Management Systems, 3rd edition. WCB/McGraw-Hill, 2003.
- C. Ruemmler and J. Wilkes. Disk Shuffling, Technical Report HPL-CSP-91-30, Hewlett-Packard Laboratories, 1991.
- S. W. Schlosser, J. Schindler, S. Papadomanolakis, M. Shao, A. Ailamaki, C. Faloutsos, and G. R. Granger. On multidimensional data and modern disks. In 4th USENIX Conference on File and Storage Technologies, pages 225–238, 2005.
- T. Skopal, J. Pokorný, and V. Snášel. Nearest Neighbours Search using the PMtree. In DASFAA '05, Beijing, China, pages 803–815. LNCS 3453, Springer, 2005.
- J. S. Vitter. External memory algorithms and data structures: dealing with massive data. ACM Computing Surveys, 33(2):209–271, 2001.
- 16. R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases, pages 194–205, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.
- B. Yu and S. Kim. An efficient zoning technique for multi-dimensional access methods. In TEAA 2006, LNCS 3888, Springer, pages 129–143, 2006.