10 Access Methods

Apostolos N. Papadopoulos
Aristotle University of Thessaloniki

Kostas Tsichlas
Aristotle University of Thessaloniki

Anastasios Gounaris
Aristotle University of Thessaloniki

Yannis Manolopoulos
Aristotle University of Thessaloniki

10.1 Introduction

We are witnessing a tremendous growth in the size of the data gathered, stored, and processed by various kinds of information systems. Therefore, no matter how big memories become, there is always the need to store data in secondary or even tertiary storage to facilitate access. Even if the data set can fit in main memory, there is still a need to organize data to enable efficient processing. In this chapter, we discuss the most important issues related to the design of efficient access methods (i.e., indexing schemes), which are the fundamental tools in database systems for efficient query processing. For the rest of the discussion, we are going to use the terms access method and index interchangeably.

Take, for example, a large data set containing information about millions of astronomical objects (e.g., stars, planets, comets). An astronomer may require some information out of this data set. Therefore, the most natural way to proceed is to store the data set in a database management system (DBMS) in order to enjoy SQL-like query formulation. For example, a possible query in natural language is: “show me all stars which are at most 1000 light-years away from the sun.” To answer such a query efficiently, one should avoid the exhaustive examination of the whole data set. Otherwise, the execution of each query will occupy the system for a long period of time, which is not practical and leads to performance degradation.

For the rest of the discussion, we are mainly interested in **disk-based access methods**, where the data set as well as the auxiliary data structures to facilitate access reside on magnetic disks. The challenge in this case is to perform as few **disk accesses** as possible, because each random access to the disk (i.e., reading or writing a block) costs about 5–8 ms, which is significantly slower than processing in main memory. Moreover, we assume that our data are represented by records of the form $<a_1; a_2,\ldots, a_m>$, where each a_i denotes an attribute value. Attribute values may be simple, an integer for example, or may correspond to more complex objects such as points in 3D space or other geometric shapes. When needed, we are going to make clear the kind of data supported by each access method. For example, some access methods are good in organizing unidimensional objects (e.g., price, salary, population), whereas others have been specifically designed to handle points or rectangles in 2D or 3D space, text, DNA sequences, time-series, to name a few.

10.2 Underlying Principles

In contrast to memory-resident data structures, handling large data collections requires the corresponding access method (or at least a large part of it) to reside on secondary storage. Although flash memories are currently widely used, the magnetic disk continues to be the predominant secondary storage medium, used extensively by large information systems. The fundamental disk limitation is that accessing data on a disk is hundreds of times slower than accessing it in main memory. In fact, this limitation was the driving force underlying the development of efficient access methods trying to reduce the impact of this limitation as much as possible. In this section, we discuss briefly some key issues in access methods.

10.2.1 Blocks and Records

The fundamental characteristic of an access method is that the data are accessed in chunks called **pages** or **blocks**. In particular, whenever a data item *x* is requested, instead of fetching only *x*, the system reads a whole set of data items that are located near item *x*. In our context, near means within the same block. Each block can accommodate a number of data items. Usually, all blocks are of the same size *B*. Typical block sizes are 4 Kb, 8 Kb, 16 Kb, or larger. Obviously, the larger the block size, the more data items can fit in every block. Moreover, the number of items that fit in each block depends also on the size of each data item. One of the primary concerns in the design of efficient access methods is storing data items that are likely to be requested together in the same block (or nearby blocks). Thus, the target is to fetch into memory more useful data by issuing only one block access. Since this is not feasible in all cases, a more practical goal is to reduce the number of accesses (reads or writes) as much as possible.

There are two fundamental types of block accesses that are usually supported by access methods: a **random access** involves fetching a randomly selected block from the disk, whereas a **sequential access** just fetches the next block. Usually, a random access is more costly because of the way magnetic disks operate. To facilitate a random access, the **disk heads** must be positioned right on top of the track that contains the requested block, thus requiring a significant amount of time, called **seek time**. In fact, seek time is the predominant cost of an I/O disk operation. On the other hand, a sequential access just reads blocks one by one in a get-next fashion, thus requiring less seek time. However, to facilitate sequential access blocks must be located in nearby positions on the disk to minimize the required seek operations. In general, sequential access is more restrictive and easier to obtain than random access. In addition, random access is more useful because of the flexibility offered to access any block any time.

Next, we describe briefly how records are organized inside each block. We limit our discussion for the case where records are of **fixed size** denoted as *R*. Therefore, the maximum number of records that can fit in a block of size *B* is simply $\lfloor B/R \rfloor$. There are two basic alternatives we may follow to organize these records inside the block. The first approach is to force that all free space will be placed at the end of the block. This means that whenever we delete a record, its place will be taken by the last record in the block. The second alternative is to use a small index in the block header recording information about which record slot is occupied and which one is free. The second alternative avoids moving records inside the block, but reduces the capacity of the block because of the index used. In case **variable size** records are allowed, we expect that less storage will be required but processing time may increase due to some extra bookkeeping required to locate each record.

10.2.2 Fundamental Operations

Although access methods have different capabilities depending on their design and the problem they call to support, they all are primarily built to support a set of fundamental operations. The most significant one is **searching**. In the simplest case, a search operation takes as input a value and returns some information back to the caller. For example: “What is the perimeter of the Earth?,” “Display the ids of the customers located in Greece,” “What is the salary of John Smith?” or “Is Jack Sparrow one of our customers?” All these **queries** can be formulated as simple search operations. However, to support these queries as efficiently as possible, the corresponding access method should provide access to the appropriate record attributes. For example, to find the perimeter of the Earth, the access method must be searchable by the name of an astronomical object. Otherwise, the only way to spot the answer is to resort to sequential scanning of the whole data set. Similarly, to display the customers residing in Greece, our access method must be able to search by the name of the country.

There are other search-oriented queries that are clearly much more complex. For example, “find the names of the cities with a population at least 1 million and at most 5 millions.” Clearly, this query involves searching in an interval of populations rather than focusing on a single population value. To support such a query efficiently, the access method must be equipped with the necessary tools. Note also, that depending on the application, searching may take other forms as well. For example, if the access method organizes points in the 2D space, then we may search by a region asking for all points falling in the region of interest. In any case, to facilitate efficient search, the access method must be organized in such a way that queries can be easily handled, avoiding scanning the whole database.

Two operations that change the contents of an access method are the insertion of new objects and the deletion of existing ones. If the access method does not support these operations, it is characterized as **static**; if both are supported, it is called **dynamic**; whereas if only insertions are supported, then it is called **semi-dynamic**. In the static case, the access method will be built once and there is no need to support insertions/deletions. The dynamic case is the most interesting and challenging one, since most of the real-life applications operate over data sets that change continuously, and potentially quite rapidly. Thus, insertions and deletions must be executed as fast as possible to allow for efficient maintenance of the access method.

In some cases, there is a need to build an access method when the corresponding data set is known in advance. The simplest solution is to just perform many invocations of the insertion operation. However, we can do much better because the data set is known, and therefore with appropriate preprocessing the index may be built much faster than by using the conservative one-by-one insertion approach. The operation of building the index taking into consideration the whole data set is called **bulk loading**.

10.3 Best Practices

In this section, we study some important indexing schemes that are widely used both in academia and industry. First, we discuss about the ***B*-tree** and **hashing** which are the predominant access methods for 1D indexing. Then, we center our focus to spatial access methods and discuss the ***R*-tree** and briefly some of its variations.

10.3.1 Fundamental Access Methods

The two dominant categories of fundamental external memory indexing methods are tree-based methods and hash-based methods. For tree-based methods, the dominant example is the B-tree [5], while for hash-based methods linear [38] and extendible [18] hashing are the most common ones. In the following, we briefly present both methods and their variants/extensions.

10.3.1.1 *B*-Tree

The *B*-tree [5] is a ubiquitous data structure when it comes to external memory indexing, and it is a generalization of balanced binary search trees. The intuition behind this generalization is that reading a block should provide the maximum information to guide the search. The use of binary trees may result in all nodes on a search path to reside in distinct blocks, which incurs an $O(\log_2 n)$ overhead while our goal is to impose that the number of blocks to read in order to find an element is $O(\log_B n)$ (i.e., logarithmic with respect to the data set cardinality). The *B*-tree with parameters k (corresponding to the internal node degree) and c (corresponding to leaf capacity) is defined as follows:

1. All internal nodes v have degree $d(v)$ such that $\ell \cdot k \leq d(v) \leq u \cdot k$, where $u > \ell > 0$. The only exception is the root, whose degree is lower-bounded by 2.
2. All leaves lie at the same level; that is, the depth of all leaves is equal.
3. All leaves l have size $|l|$ such that $\ell' \cdot c \leq |l| \leq c$, where $0 < \ell' < 1$.

The maximum height of the tree is $\lfloor \log_{\ell \cdot k}(n/\ell' c) + 1 \rfloor$ while its minimum height is $\lceil \log_{u \cdot k}(n/c) \rceil$. Depending on the satellite information of each element, the size of the pointers, as well as any other needed information within each block, we may set accordingly constants k, c, ℓ, ℓ', and u. Additionally, these parameters are also affected by the desired properties of the tree. Henceforth, for simplicity and without loss of generality, we assume that $k = c = B$, $u = 1$, and $\ell = \ell' = 1/2$.

In the classic *B*-tree, internal nodes may store elements (records) apart from pointers to children. This results in the decrease of parameter k and thus the height of the tree is increased. In practice, and to avoid this drawback, the B^+-trees [15] are extensively used. These trees store elements only at leaves while internal nodes store routing information related to the navigation during search within the tree. In this way, the parameter k is increased considerably and the height of the tree is reduced.

To perform a search for an element x, the search starts from the root and moves through children pointers to other internal nodes toward the leaves of the tree. When a leaf l is found, it is brought into main memory and a sequential or binary search is performed to find and return the element x or to report failure. One can also return the predecessor or the successor of the element x (which is x if it exists in the tree), but one more I/O may be needed. To support range search queries, the B^+-tree is usually changed so that all leaves constitute a linked list. As a result, for the range query $[x_1, x_2]$, first the leaf is located that contains the successor of x_1, and then a linear scan of all leaves whose value is within the range $[x_1, x_2]$ is performed with the help of the linked list of leaves. Before moving to update operations, we first briefly discuss the rebalancing operations. The B^+-tree (in fact all such trees) can be restored after an update operation by means of splits, fusions, and shares.

A node v is split when there is no available space within the node. In this case, half the information contained in v (pointers and routing information in internal nodes or elements at leaves) is transferred to a new node v' and thus both nodes have enough free space for future insertions. However, the father of v has its children increased by one, which means that there may be a cascading split possibly reaching even the root. A node v requires fusion with a sibling node v' when the used space within v is less than $\frac{1}{2}B$ due to a deletion. In this case, if the combined size of v and v' is $>B$, then some information is carried over from v' to v. In this case we have a share operation which is a terminal rebalancing operation in the sense that the number of children of the father of v and v' remains the same. Otherwise, all information of v is transferred to v' and v is deleted. The father of v' has its children reduced by one and so there may be cascading fusions toward the root.

An insertion operation invokes a search for the proper leaf l in which the new element must be inserted to respect the sorted order of elements in the B^+-trees. If l has available space, then the new element is inserted and the insertion terminates. If it does not have available space, then the leaf is split into two leaves l and l' and the new element is inserted to either l or l', depending on its value. Then, based on whether the father of l and l' needs split, the process continuous until an internal node with free space is reached or until the root is split. A deletion operation is similar with the exception that it invokes a fusion operation for rebalancing. In Figure 10.1 an example of a B^+-tree is depicted.

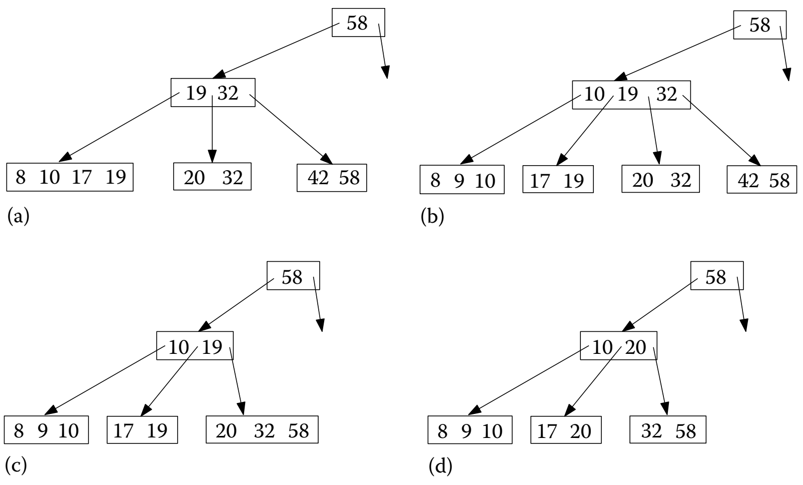


FIGURE 10.1 An example of update operations for the B^+-tree is depicted. Internal nodes contain only routing information. (a) An example of a B^+-tree ($B = 4$). (b) Adding 9 and performing a split. (c) Removing 42 and performing a fusion. (d) Removing 19 and performing a share.

A variant of the B^+-tree is the B^*-tree [15], which balances adjacent internal nodes to keep them more densely packed. This variant requires that $l = l' = \frac{2}{3}$ instead of $\frac{1}{2}$. To maintain this, instead of immediately splitting up a node when it gets full, its elements are shared with an adjacent node. As soon as the adjacent node gets full, then the two nodes are split into three nodes.

In total, search and update operations can be carried out in $O(\log_B n)$ I/Os while a range search query can be carried out in $O(\log_B n + (t/B))$ I/Os, where t is the number of reported elements.

10.3.1.2 *B*-Tree Variations and Extensions

There are numerous variants and extensions of B-trees, and we only report some of them. Most of these, if not all, are quite complicated to implement and can be used in practice only in particular scenarios. The Lazy B-tree [33] support updates with $O(1)$ worst-case rebalancings (not counting the search cost) by carefully scheduling these operations over the tree. The ISB-tree [33] uses interpolation search in order to achieve a $O(\log_B \log n)$ expected I/Os for searching and updating, provided that the distribution of the elements belongs to a large family of distributions with particular properties. It is simple to extend basic B^+-trees to maintain a pointer to the parent of each node as well as to maintain that all nodes at each level are connected in a doubly linked list. Applying these changes, the B^+-tree can support efficiently finger searches [11] such that the number of I/Os for searching becomes $O(log_B d)$, where d is the number of leaves between a leaf l designated as the finger and the leaf l' we are searching for. When searching for nearby leaves, this is a significant improvement over searching for l' from the root. One can also combine B-trees with hashing [43] to speed-up operations.

There are numerous extensions of B-trees that provide additional functionalities and properties. The weight-balanced B-tree [4] has the weight property that normal B-trees lack. For an internal node v let $w(v)$ be its weight, which is the number of elements stored in the subtree of v. The weight property states that an internal node v is rebalanced only after $\Theta(w(v))$ updates have been performed at its subtree since the last update. This property is very important to reduce complexities when the B-tree has secondary structures attached to internal nodes [4]. A partial persistent B-tree [7] is a B-tree that maintains its history attaining the same complexities with normal B-trees. Efficient fully persistent B-trees have very recently been designed [12] and allow updates in the past instances of the B-tree giving rise to different history paths. String B-trees [19] have been designed to support efficiently search and update operations on a set of strings (and its suffixes) in external memory. Cache-oblivious B-trees [9] have been designed that do not need to know basic parameters of the memory hierarchy (like B) in order to attain the same complexities as normal B-trees, which make heavy use of the knowledge of these parameters. Finally, buffer trees [3] are B-trees that allow for efficient execution of batch updates (or queries). This is accomplished by lazily flushing toward the leaves all updates (or queries) in the batch. In this way, each update can be supported in $O((1/B)\log_{M/B}(n/B))$ I/Os.

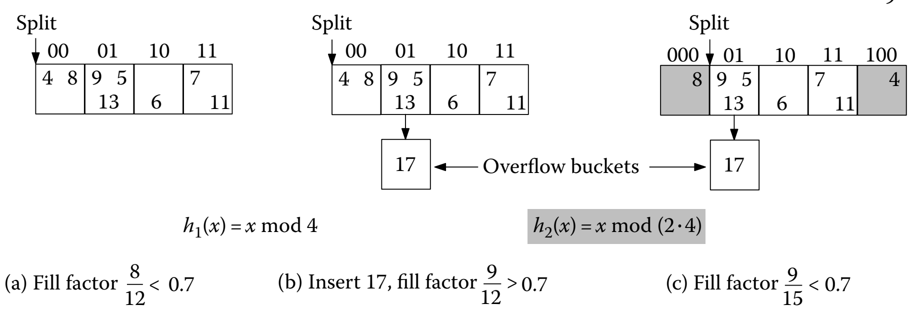
10.3.1.3 Hashing

Another much used technique for indexing is hashing. Hashing is faster than B-trees by sacrificing in functionality. The functionality of hashing is limited when compared to B-trees because elements are stored unsorted and thus there is no way to support range queries or find the successor/predecessor of an element. In hashing, the basic idea is to map each object to a number corresponding to the location inside an array by means of a hash function. It is not our intention to describe hash functions and thus we refer the interested reader to [16] and the references therein for more information on practical hash functions.

The largest problem with hashing is collision resolution, which happens when two different elements hash to the same location. The hash function hashes elements within buckets, which in this case is a block. When the bucket becomes full, then either a new overflow bucket is introduced for the same hashed value or a rehashing is performed. All buckets without the overflowing buckets constitute the primary area. Two well-known techniques are **linear hashing** [38] and **extendible hashing** [18]. The first method extends the hash-table by one when the fill factor of the hash-table (the number of elements divided by the size of the primary area) goes over a critical value. The second one extends the primary area as soon as an overflowing bucket is about to be constructed.

In linear hashing, data are placed in a bucket according to the last k bits or the last $k + 1$ bits of the hash function value of the element. A pointer *split* keeps track of this boundary. The insertion of a new element may cause the fill factor to go over the critical value, in which case the bucket on the boundary corresponding to k bits is split into two buckets corresponding to $k + 1$ bits. The number of buckets with k bits is decreased by one. When all buckets correspond to $k + 1$ bits, another expansion is initiated constructing new buckets with $k + 2$ bits. Note that there is no relationship between the bucket in which the insertion is performed and the bucket that is being split. In addition, linear hashing uses overflow buckets although it is expected that these buckets will be just a few. The time complexity for a search and update is $O(1)$ expected. In Figure 10.2 an example of an insertion is depicted.

On the other hand, extendible hashing does not make use of overflow buckets. It uses a directory that may index up to 2^d buckets, where d, the number of bits, is chosen so that at most B elements exist in each bucket. A bucket may be pointed by many such pointers from the directory since they may be indexed



Split
00 01 10 11
4 8 9 5 13 6 7 11
Split
00 01 10 11
4 8 9 5 13 6 7 11
17
Overflow buckets
Split
000 01 10 11 100
8 9 5 13 6 7 11 4
17
$h_1(x) = x \bmod 4$
$h_2(x) = x \bmod (2\cdot 4)$
(a) Fill factor $\frac{8}{12} < 0.7$
(b) Insert 17, fill factor $\frac{9}{12} > 0.7$
(c) Fill factor $\frac{9}{15} < 0.7$

FIGURE 10.2 Assuming that the critical fill factor is 0.7, then when 17 is inserted in (a), an overflow bucket is introduced (b) since the bucket corresponding to bits 01 has no space. After the insertion, the fill factor is >0.7 and as such we construct a new hash function $h_2(x)$ and introduce a new bucket (c). $h_2(x)$ is applied to gray buckets (3 bits) while $h_1(x)$ is applied to white buckets (2 bits) only.

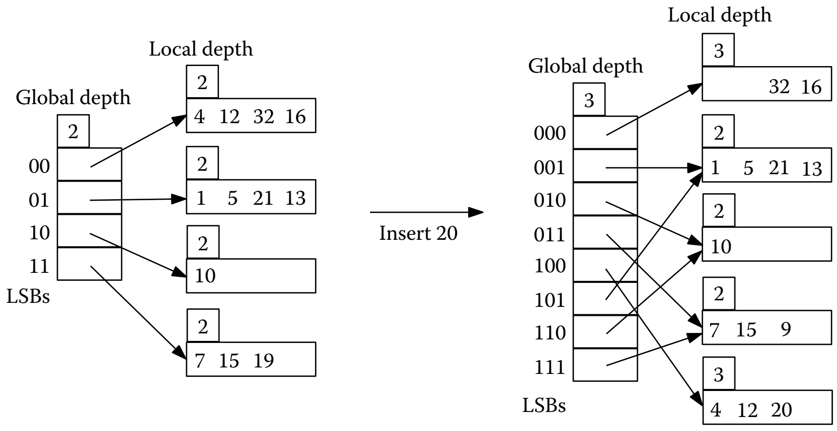


FIGURE 10.3 An example of re-organization for extendible hashing. After the insertion of 20 the first bucket overflows and we construct a new table of global depth 3; that is, with 2^3 entries. Only the overflowing bucket is split and has local depth 3, thus needing 3 bits to navigate. The buckets that did not overflow need only 2 bits and thus 2 pointers point to them.

by using less than d bits. An insertion either causes the directory to double or some of entries of the directory are changed. For example, a bucket pointed by eight entries will be split and both new buckets are pointed by four entries. In case a bucket is pointed only by one entry and needs to be split, then the directory needs to be doubled. The advantage of extendible hashing is that it never has more than two disk accesses for any record since there are no overflow buckets to traverse. The main problems with this variation on hashing are total space utilization and the need for massive reorganization (of the table). In Figure 10.3, an example of such a reorganization for an insertion is depicted.

There are many other hashing schemes with their own advantages and disadvantages. Cuckoo hashing [45] is one such scheme which is very promising because of its simplicity. The basic idea is to use two hash functions instead of only one and thus provide two possible locations in the hash-table for each element. When a new element is inserted, it is stored in one of the two possible locations provided by the hash functions. If both of these locations are not empty, then one of them is kicked out. This new displaced element is put into its alternative position and the process continues until a vacant position has been found or until many such repetitions have been performed. In the last case, the table is rebuilt with new hash functions. Searching for an element requires inspection of just two locations in the hash-table, which takes constant time in the worst case.

10.3.2 Spatial Access Methods

The indexing schemes discussed previously support only one dimension. However, in applications such as Geographic Information Systems (GIS), objects are associated with spatial information (e.g., latitude/longitude coordinates). In such a case, it is important to organize the data taking into consideration the spatial information. Although there are numerous proposals to handle spatial objects, in this chapter we will focus on the R-tree index [24], which is one of the most successful and influential spatial access methods, invented to organize large collections of rectangles for VLSI design.

10.3.2.1 *R*-Tree

R-trees are hierarchical access methods based on B^+-trees. They are used for the dynamic organization of a set of d-dimensional geometric objects representing them by the minimum bounding d-dimensional rectangles (for simplicity, MBRs in the sequel). Each node of the R-tree corresponds to the MBR that bounds its children. The leaves of the tree contain pointers to the database objects instead of pointers to children nodes. The nodes are implemented as disk blocks. It must be noted that the MBRs that surround different nodes may overlap each other. Besides, an MBR can be included (in the geometrical sense) in many nodes, but it can be associated to only one of them. This means that a spatial search may visit many nodes before confirming the existence of a given MBR. An R-tree of order (m, M) has the following characteristics:

- Each leaf node (unless it is the root) can host up to M entries, whereas the minimum allowed number of entries is $m \leq M/2$. Each entry is of the form (*mbr, oid*), such that *mbr* is the MBR that spatially contains the object and *oid* is the object's identifier.
- The number of entries that each internal node can store is again between $m \leq M/2$ and M. Each entry is of the form (*mbr, p*), where p is a pointer to a child of the node and *mbr* is the MBR that spatially contains the MBRs contained in this child.
- The minimum allowed number of entries in the root node is 2.
- All leaves of the R-tree are located at the same level.

An R-tree example is shown in Figure 10.4 for the set of objects shown on the left. It is evident that MBRs R_1 and R_2 are disjoint, whereas R_3 has an overlap with both R_1 and R_2. In this example we have assumed that each node can accommodate at most three entries. In a real implementation, the capacity of each node is determined by the block size and the size of each entry which is directly related to the number of dimensions.

The R-tree has been designed for dynamic data sets and, therefore, it supports insertions and deletions. To insert a new object's MBR, the tree is traversed top-down, and at each node a decision is made to select a branch to follow next. This is repeated until we reach the leaf level. The decision we make at each node is based on the criterion of area enlargement. This means that the new MBR is assigned to the entry which requires the least area enlargement to accommodate it. Other variations of the R-tree, such as the R^*-tree [8], use different criteria to select the most convenient path from the root to the leaf level. Upon reaching a leaf L, the new MBR is inserted, if L can accommodate it (there is at least one available slot). Otherwise, there is a node overflow and a **node split** occurs, meaning that a new node L' is reserved. Then, the old entries of L (including the new entry) are distributed to two nodes L and L'. Note that a split at a leaf may cause consecutive splits in the upper levels. If there is a split at the root, then the height of the tree increases by one.

The split operation must be executed carefully to maintain the good properties of the tree. The primary concern while splitting is to keep the overlap between the two nodes as low as possible. This is because the higher the overlap, the larger the number of nodes that will be accessed during a search operation. In the original R-tree proposal, three split policies have been studied, namely, exponential, quadratic, and linear.

Exponential split: All possible groupings are exhaustively tested and the best one, with respect to the minimization of the MBR enlargement, is chosen.

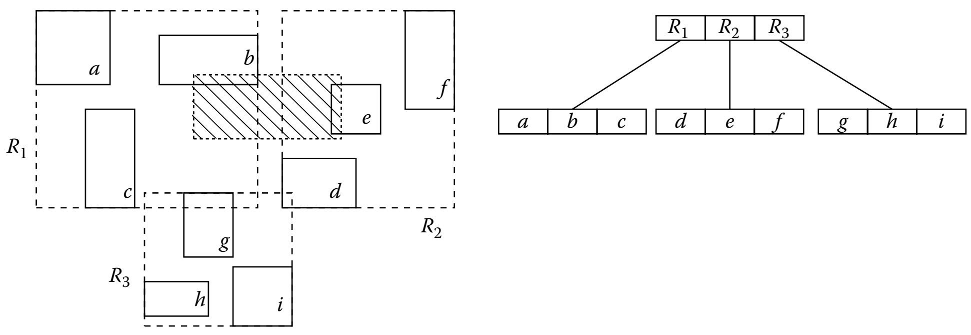


FIGURE 10.4 R-tree example.

```
Algorithm RangeSearch (Node N, Region Q)
Input: root mode N, query region Q
Output: answer set A
1.  if (N is not a leaf node)
2.      examine each entry E of N to find those E.mbr that intersect Q
3.      foreach such entry E call RangeSearch(E.ptr,Q)
4.  else // N is a leaf node
5.      examine all entries E and find those for which E.mbr intersects Q
6.      add these entries to the answer set A
7.  Endif
```

FIGURE 10.5 The *R*-tree range search algorithm.

Quadratic split: Choose two objects as seeds for the two nodes, where these objects if put together create as much empty space as possible. Then, until there are no remaining objects, insert the object for which the difference of dead space if assigned to each of the two nodes is maximized in the node that requires less enlargement of its respective MBR.

Linear split: Choose two objects as seeds for the two nodes, where these objects are as far apart as possible. Then, consider each remaining object in a random order and assign it to the node requiring the smallest enlargement of its respective MBR.

The quadratic split policy is the best choice that balances efficiency and effectiveness and, therefore, it is widely used in *R*-tree implementations.

In a deletion, an entry is removed from the leaf level of the tree, which may cause a node underflow. In this case, **re-insertion** of entries is applied to reduce the space requirements of the tree. In general, a deletion has the opposite effect than that of an insertion. In both cases, the path from the root to the leaf level that was affected by the operation must be adjusted properly to reflect the changes performed in the tree. Thus, the deletion of an entry may cause a series of deletions that propagate up to the root of the *R*-tree. If the root has only one child, then it is removed and the height of the tree decreases by one.

In the sequel, we examine briefly how search is performed in an *R*-tree. We will center our focus to range queries, where the user defines a query region *Q* and the answer to the query contains the spatial objects that intersect *Q*. For example, in Figure 10.4, the query region is shown filled, whereas the answer to the query is composed of the objects *b* and *e*. The outline of the algorithm that processes range queries in an *R*-tree is given in Figure 10.5. For a node entry *E, E.mbr* denotes the corresponding MBR and *E.p* the corresponding pointer to the next level. If the node is a leaf, then *E.p* denotes the corresponding object identifier (*oid*). We note that the rectangles that are found by range searching constitute the candidates of the filtering step. The actual geometric objects intersected by the query rectangle have to be found in a refinement step by retrieving the objects of the candidate rectangles and testing their intersection.

10.3.2.2 R-Tree Bulk Loading

Recall that bulk loading is the process of building an index by taking into consideration the data set which is known in advance. Thus, usually this operation is applied for static data sets or when insertions and deletions are rare. In most of the cases, when bulk loading is applied, the leaf level of the tree is created first. Then, the upper tree levels can be built one by one until we reach the root.

The first bulk-loading algorithm for *R*-trees proposed in [47] first sorts the data objects by using the *x* coordinate of their center. If objects are points rather than rectangles, then the *x* coordinate of the point is used. By using the sorted order, the leaves may be formed by placing the first *M* entries in the first leaf, until no more data are available. This way, all leaves will be 100% full, except maybe of the last leaf which may contain less.

Another contribution to this problem is reported in [30]. The algorithm is similar to that of [47] in that again a sorting is performed in order to build the leaf level of the tree. Sorting is performed by using the Hilbert value of the data objects' centroids. According to the performance evaluation given in [30], this approach shows the best overall performance with respect to the cost of performing queries.

STR (Sort-Tile-Recursive) is a bulk-loading algorithm for *R*-trees proposed by Leutenegger et al. [32]. Let *n* be a number of rectangles in 2D space. The basic idea of the method is to tile the address space by using *V* vertical slices, so that each slice contains enough rectangles to create approximately $\sqrt{n/M}$ nodes, where *M* is the *R*-tree node capacity. Initially, the number of leaf nodes is determined, which is $L = \lceil n/M \rceil$. Let $V = \sqrt{L}$. The rectangles are sorted with respect to the *x*-coordinate of the centroids, and *V* slices are created. Each slice contains *V.M* rectangles, which are consecutive in the sorted list. In each slice, the objects are sorted by the *y*-coordinate of the centroids and are packed into nodes (placing *M* objects in a node). Experimental evaluation performed in [32] has demonstrated that the STR method is generally better than previously proposed bulk-loading methods. However, in some cases the Hilbert packing approach performs marginally better.

10.3.2.3 R-Tree Variations and Extensions

Several *R*-tree variations have been proposed in the literature to improve the performance of queries. Here we discuss briefly three successful variations that show better performance than the original proposal by Guttman. These variations differ in several aspects like the way insertions and deletions are performed, the optimization criteria being used, the split policy applied, and the storage utilization.

The R^+*-tree*: R^+-trees were proposed as an alternative that avoids visiting multiple paths during point location queries, aiming at the improvement of query performance [52]. Moreover, MBR overlapping of internal modes is avoided. This is achieved by using the clipping technique. In other words, R^+-trees do not allow overlapping of MBRs at the same tree level. In turn, to achieve this, inserted objects have to be divided in two or more MBRs, which means that a specific object's entries may be duplicated and redundantly stored in several nodes. Therefore, a potential limitation of R^+-trees is the increased space requirements due to redundancy.

The R^**-tree*: R^*-trees [8] were proposed in 1990 but are still very well received and widely accepted in the literature as a prevailing performance-wise structure that is often used as a basis for performance comparisons. As already discussed, the *R*-tree is based solely on the area minimization of each MBR. On the other hand, the R^*-tree goes beyond this criterion and examines the following: (1) minimization of the area covered by each MBR, (2) minimization of the overlap between MBRs, (3) minimization of MBR margins (perimeters), and (4) maximization of storage utilization. The R^*-tree follows an engineering approach to and the best possible combinations of the aforementioned criteria. This approach is necessary, because the criteria can become contradictory. For instance, to keep both the area and the overlap low, the lower allowed number of entries within a node can be reduced. Therefore, storage utilization may be impacted. Also, by minimizing the margins so as to have more quadratic shapes, the node overlapping may be increased.

The Hilbert R-tree: The Hilbert *R*-tree [31] is a hybrid structure based on the *R*-tree and the B^+-tree. Actually, it is a B^+-tree with geometrical objects being characterized by the Hilbert value of their centroid. The structure is based on the Hilbert space-filling curve. It has been shown in [42] that the Hilbert space-filling curve preserves well the proximity of spatial objects. Entries of internal tree nodes are augmented by the largest Hilbert value of their descendants. Therefore, an entry *e* of an internal node is a triple of the form < *mbr, H, p* > where *mbr* is the MBR that encloses all the objects in the corresponding subtree, *H* is the maximum Hilbert value of the subtree, and *p* is the pointer to the next level. Entries in leaf nodes are exactly the same as in *R*-trees, R^+-trees, and R^*-trees and are of the form < *mbr, oid* >, where *mbr* is the MBR of the object and *oid* the corresponding object identifier.

10.3.3 Managing Time-Evolving Data

Time information plays a significant role in many applications. There are cases where in addition to the data items per se, the access method must maintain information regarding the time instance that a particular event occurred. For example, if one would like to extract statistical information regarding the sales of a particular product during the past 5 years, the database must maintain historical information. As another example, consider an application that tracks the motion patterns of a specific species. To facilitate this, each location must be associated with a timestamp. Thus, by inspecting the location in consecutive timestamps, one may reveal the motion pattern of the species. For the rest of the discussion, we assume that time is discrete and each timestamp corresponds to a different time instance.

One of the first access methods that was extended to support time information is the *B*-tree and its variations. The most important extensions are the following:

The time-split B-tree [39]: This structure is based on the write-once *B*-tree access method proposed in [17], and it is used for storing **multi-version data** on both optical and magnetic disks. The only operations allowed are insertions and searches, whereas deletions are not supported. The lack of deletions in conjunction with the use of two types of node splits (normal and time-based) is the main reason for the structure's space and time efficiency. However, only exact-match queries are supported efficiently.

The fully persistent B^+-tree [36]: In contrast to the time-split *B*-tree, the fully persistent B^+-tree supports deletions in addition to insertions and searches. Each record is augmented by two fields t_{start} and t_{end}, where t_{start} is the timestamp of the insertion and t_{end} is the timestamp when the record has been deleted, updated, or copied to another node. This way, the whole history can be recorded and queries may involve the past or the present status of the structure.

The multi-version B-tree [6]: This access method is asymptotically optimal and allows insertions and deletions only at the last (current) timestamp, whereas exact-match and range queries may be issued for the past as well. The methodology proposed in [6] may be used for other access methods, when there is a need to transform a simple access method to a multi-version one.

There are also significant research contributions in providing time-aware spatial access methods. An index that supported space and time is known as **spatiotemporal** access method. Spatiotemporal data are characterized by changes in location or shape with respect to time. Supporting time increases the number of query types that can be posed by users. A user may focus on a specific time instance or may be interested in a time interval. Spatiotemporal queries that focus on a single time instance are termed **time-slice queries**, whereas if they focus on a time interval, they are termed **time interval queries**. If we combine these choices with spatial predicates and the ability to query the past, the present, or the future, spatiotemporal queries can be very complex, and significant effort is required to process them.

A large number of the proposed spatiotemporal access methods are based on the well-known *R*-tree structure. In the following, we discuss briefly some of them.

The 3D R-tree [54]: In this index, time is considered as just another dimension. Therefore, a rectangle in 2D becomes a box in 3D. The 3D *R*-tree approach assumes that both ends of the interval $[t_{start}, t_{end})$ of each rectangle are known and fixed. If the end time t_{end} is not known, this approach does not work well, due to the use of large MBRs leading to performance degradation with respect to queries. In addition, conceptually, time has special characteristic, i.e., it increases monotonically. This suggests the use of more specialized access methods.

The partially persistent R-tree [34]: This index is based on the concept of partial persistency [35]. It is assumed that in spatiotemporal applications, updates arrive in time order. Moreover, updates can be performed only on the last recorded instance of the database, in contrast to general bitemporal data. The partially persistent *R*-tree is a directed acyclic graph of nodes with a number of root nodes, where each root is responsible for recording a subsequent part of the ephemeral *R*-tree evolution. Object records are stored in the leaf nodes of the PPR-tree and maintain the evolution of the ephemeral *R*-tree data objects. As in the case of the fully persistent B^+-tree, each data record is augmented to include the two lifetime fields, t_{start} and t_{end}. The same applies to internal tree nodes, which maintain the evolution of the corresponding directory entries.

The multi-version 3D R-tree [53]: This structure has been designed to overcome the shortcomings of previously proposed techniques. It consists of two parts: a multi-version *R*-tree and an auxiliary 3D *R*-tree built on the leaves of the former. The multi-version *R*-tree is an extension of the multi-version *B*-tree proposed by Becker et al. [6]. The intuition behind the proposed access method is that time-slice queries can be directed to the multi-version *R*-tree, whereas time-interval queries can be handled by the 3D *R*-tree.

10.4 Advanced Topics

In this section, we discuss some advanced topics related to indexing. In particular, we focus on three important and challenging issues that attract research interest mainly due to their direct impact on performance and because they radically change the way that typical indexing schemes work. Namely, these topics are (1) cache-oblivious indexing, (2) on-line indexing, and (3) adaptive indexing.

10.4.1 Cache-Oblivious Access Methods and Algorithms

In 1999, Frigo et al. [20] introduced a new model for designing and analyzing algorithms and data structures taking into account memory hierarchies. This is the **cache-oblivious model** which, as its name implies, is oblivious to the parameters as well to some architectural characteristics of the memory hierarchy. Data structures designed in this model do not know anything about the memory hierarchy, but have performance which is comparable to data structures that have been designed with knowledge of the particular memory hierarchy. In a nutshell, this is accomplished by laying out the data structure over an array which is cache-oblivious by default. The main problem is how to design this layout and how to maintain it when it is subjected to update operations. The main advantage of these indexing schemes (an example is the cache oblivious *B*-tree [9]) is not only their simplicity and portability among different platforms but also their innate ability to work optimally in all levels of the memory hierarchy.

10.4.2 On-Line and Adaptive Indexing

Traditional physical indexing building follows an off-line approach to analyzing workload and creating the data structures to enable efficient query processing. More specifically, typically, a sample workload is analyzed with a view to selecting the indices to be built with the help of auto-tuning tools (e.g., [2,14,56]). This is an expensive process, especially for very large databases. On-line and adaptive indexing have introduced a paradigm shift, where indices are created on the y during query processing. In this way, database systems avoid a complex and time-consuming process, during which there is no index support, and can adapt to dynamic workloads. We draw a distinction between on-line and adaptive indexing following the spirit of [28]. According to that distinction, on-line indexing monitors the workload and creates (or drops) indices on-line during query execution (e.g., [13,40]). Adaptive indexing takes one step further: in adaptive indexing the process of index building is blended with query execution through extensions to the logic of the operators in the query execution plan (e.g., [23,25]). In the sequel, we discuss briefly these two approaches which differ significantly from the typical bulk-loading process for index building.

On-line Indexing: The paradigm of on-line indexing solutions departs from traditional schemes in that the index tuning mechanism is always running. A notable example of such category is described in [13], where the query engine is extended with capabilities to capture and analyze evidence information about the potential usefulness of indices that have not been created thus far. Another interesting approach is soft-indices [40], which build on top of an index-tuning mechanism that continuously collects statistical information and periodically solves the NP-hard problem of index selection. Similarly to adaptive indexing, the new decisions are enforced during query processing but without affecting the operator implementation.

Adaptive Indexing: The most representative form of adaptive indexing is **database cracking** [25], which is mostly tailored to (in-memory) column-stores [1]. Database cracking revolves around the concept of continuous physical re-organization taking place at runtime. Such physical re-organization is automated, in the sense that does not involve human involvement, does not contain any off-line preparatory phase, and may not build full indices. The latter means that the technique is not only adaptive but also relies on partial indexing that is refined during workload execution in an incremental manner. To give an example, suppose that a user submits a range query on the attribute $R.A$ for the first time and the range predicate is $value1 < R.\ A < value2$. According to database cracking, during execution, a copy of $R.\ A$ will be created, called the *cracker column* of $R.\ A$. The data in that cracker column is physically re-organized and is split in three parts: (1)$value1 \leq R.\ A$; (2) $value1 < R.\ A < value2$; and (3) $R.\ A \leq$ value2. Moreover, an AVL tree, called *cracker index*, is created to maintain the partitioning information. This process of physical re-organization continues with every query. Overall, each query may benefit from the cracking imposed by previous queries and the new splits introduced may be of benefit for subsequent queries. In other words, the access methods are created on the y and are query-driven, so that they eventually match the workload. Database cracking is characterized by low initialization overhead, is continuously refined, and may well outperform techniques that build full indices, because the latter need a very large number of queries in order to amortize the cost of full index building to be outweighed. Cracking can be extended to support updates [26] and complex queries [27].

Nevertheless, database cracking may converge slowly and is sensitive to query pattern. Such limitations are mitigated by combining traditional indexing techniques (B-trees) with database cracking, resulting in the so-called adaptive merging [23]. The key distinction between the two approaches is that adaptive merging relies on merging rather than on partitioning, and that adaptive merging is applicable to disk-based data as well.

10.5 Research Issues

The field of access methods continues to be one of the most important topics in database management systems, mainly because of its direct impact on query performance. Although the field has been active for several decades, modern applications pose novel challenges that must be addressed carefully toward efficient processing. Here, we discuss briefly some of these challenges.

High-dimensional data: One of the most important challenges emerges when the number of attributes (dimensions) increases significantly. R-trees and related indexing schemes perform quite well for dimensionalities up to 15 or 20. Above this level, the dimensionality curse renders indexing difficult mainly due to concentration of measure effect. Unfortunately, the cases where high dimensionalities appear are not few. For example, in multimedia data management, images are often represented by feature vectors, each containing hundreds of attributes. There are several proposals in the literature to improve performance in these cases, trying to reduce the effects of dimensionality curse. For example, the X-tree access method proposed in [10] introduces the concept of **supernode** and avoids node splits if these cannot lead to a good partitioning. As another example, Vector Approximation File [55] is an approximation method that introduces error in query processing. For static data sets, dimensionality reduction may also be applied to first decrease the number of dimensions in order to be easier for access methods to organize the data.

Modern hardware: Hardware is evolving, and although the magnetic disk is still the prevailing secondary storage medium, new types of media have been introduced, such as **solid-state drives**.

Also, processors enjoy a dramatic change by the ability to include multiple cores in the same chip. These changes in hardware bring the necessity to change the way we access and organize the data. For example, solid-state drives do not suffer from the seek time problem and, therefore, the impact of I/O time on the performance becomes less significant. This means that operations that are I/O-bounded with magnetic disks may become CPU-bounded with modern hardware, due to the reduction in I/O time. The indexing problem becomes even more challenging as new processing and storage components like GPUs and FPGAs are being used more often. The new access methods must take into account the specific features of new technology.

Parallelism and distribution: The simultaneous use of multiple instances of resources such as memory, CPU, and disks brings a certain level of flexibility to query processing, but it also creates significant challenges in terms of overall efficiency. By enabling concurrent execution of tasks, we are facing the problem of coordinating these tasks and also the problem of determining where data reside. Therefore, parallel and distributed access methods are needed that are able to scale well with the number of processors. In addition to scalability, these techniques must avoid bottlenecks. For example, if we simply take an access method and place each block to a different processor, the processor that hosts the root will become a hot spot. Although there are many significant contributions for parallel/distributed indexing (e.g., distributed hash-tables), modern programming environments such as MapReduce on clusters or multi-core CPUs and GPUs call for a reconsideration of some concepts to enable the maximum possible performance gains and scalability.

On-line and adaptive indexing: On-line and adaptive indexing are technology in evolution; thus multiple aspects require further investigation. Among the most important ones, we highlight the need of investigation of concurrency control in database cracking techniques, the most fruitful combination of off-line, on-line, and adaptive techniques, and the application of database cracking to row stores. Finally, as new indexing methodologies are being proposed, we need a comprehensive benchmark in order to assess the benefits and weaknesses of each approach; a promising first step toward this direction is the work in [29].

10.6 Summary

Access methods are necessary toward efficient query processing. The success of an access method is characterized by its ability to organize data in such a way that locality of references is enhanced. This means that data that are located in the same block are likely to be requested together. In this chapter, we discussed some fundamental access methods that enjoy a wide-spread use by the community due to their simplicity and their excellent performance. Initially, we discussed some important features of access methods and then we described *B*-trees and hashing which are the prevailing indexing schemes for 1D data and *R*-trees, which is the most successful family for indexing spatial and other types of multidimensional data. For *R*-trees we also described briefly bulk-loading techniques, which is an important operation for index creation when the data set is available. Finally, we touched the issues of on-line and adaptive indexing, which enjoy a growing interest due to the ability to adapt dynamically based on query workloads. Access methods will continue to be a fundamental research topic in data management. In the era of big data, there is a consistent need for fast data management and retrieval, and thus indexing schemes are the most important tools in this direction.

Glossary

***B*-tree:** A tree-based index that allows queries, insertions, and deletions in logarithmic time.
Bulk loading: The insertion of a known before-hand sequence of objects into a set of objects.
Cache-oblivious model: A model that supports the design of algorithms and data structures for memory hierarchies without knowing its defining parameters.

Cuckoo hashing: Hashing that uses two hash functions to guarantee worst-case $O(1)$ accesses per search.
Database cracking: A form of continuous physical re-organization taking place at runtime.
Disk access: Reading (writing) a block from (to) the disk.
Disk-based access method: An access method that is designed specifically to support access for disk-resident data.
Disk head: The electronic component of the disk responsible for reading and writing data from and to the magnetic surface.
Dynamic access method: An access method that allows insertions, deletions, and updates in the stored data set.
Extendible hashing: A hashing method that doubles the index size to avoid overflowing.
Fixed-size record: A record with a predetermined length which never changes.
Hashing: An index method that uses address arithmetic to locate elements.
Linear hashing: A hashing method that extends the main index by one each time the fill factor is surpassed.
Multi-version data: Data augmented by information about the time of their insertion, deletion, or update.
Node split: The operation that takes place when a node overflows, i.e., there is no room to accommodate a new entry.
On-line indexing: An indexing scheme where the index tuning mechanism is always running.
***R*-tree:** A hierarchical access method that manages multidimensional objects.
Random access: The operation of fetching into main memory a randomly selected block from disk or other media.
Re-insertion: The operation of re-inserting some elements in a block. Elements are first removed from the block and then inserted again in the usual manner.
Semi-dynamic access method: An access method that supports queries and insertions of new items; it does not support deletions.
Seek time: Time required for the disk heads to move to the appropriate track.
Solid-state drive: A data storage device that uses integrated circuits solely to store data persistently.
Spatiotemporal data: Data containing both spatial and temporal information.
Static access method: An access method that does not support insertions, deletions, and updates.
Supernode: A tree node with a capacity which is a multiple of the capacity of a regular node.
Time-slice query: A query posed on a specific timestamp.
Time-interval query: A query that refers to a time interval defined by two timestamps.
Variable-size record: A record whose length may change due to to changes in the size of individual attributes.

Further Information

There is a plethora of resources that the interested reader may consult to grasp a better understanding of the topic. Since access methods are related to physical database design, the book of Lightstone et al. [37] may be of interest to the reader. For a thorough discussion of *B*-trees and related issues, we recommend the article of Graefe in [22]. Also, almost every database-oriented textbook contains one or more chapters devoted to indexing. For example, Chapters 8–11 of [46] cover indexing issues in detail, in particular *B*-trees and hashing, in a very nice way.

For a detailed discussion of spatial access methods, the reader is referred to two books from Samet, [50] and [51]. In [50] the reader will find an in-depth examination of indexing schemes that are based on space portioning (e.g., linear quadtrees). Also, in [51], in addition to the very detailed study of spatial access methods, the author presents metric access methods, where objects reside in a metric space rather than in a vector space. For a thorough discussion of the *R*-tree family, the reader is referred to the book of Manolopoulos et al. [41]. We also mention a useful survey paper of Gaede and Günther [21], which covers indexing schemes that organize multidimensional data.

In many applications, time information is considered very important and thus it should be recorded. For example, in a fleet management application, we must know the location of each vehicle and the associated timestamp. There are many approaches to incorporate time information into an index. Some of them were briefly discussed in Section 10.3.3. The reader who wants to cover this issue more thoroughly may consult other more specialized resources such as the survey of Salzberg and Tsotras [49] as well as the article of Nguyen-Dinh et al. [44]. In addition to the indexing schemes designed to support historical queries, some access methods support future queries. Obviously, to answer a query involving the near future, some information about the velocities of moving objects is required. The paper by Šaltenis et al. [48] describes such an index.

The previous resources cover more or less established techniques. For modern research issues in the area, the best sources are the proceedings of leading database conferences and journals. In particular, the proceedings of the major database conferences *ACM SIGMOD/PODS*, *VLDB*, *IEEE ICDE*, *EDBT/ICDT*, *ACM CIKM*, and *SSDBM* contain many papers related to access methods and indexing. Also, we recommend the journals *ACM Transactions on Database Systems*, *IEEE Transactions on Knowledge and Data Engineering*, *The VLDB Journal*, and *Information Systems* where, in most cases, the published articles are enhanced versions of the corresponding conference papers.

References

1. DJ. Abadi, PA. Boncz, and S. Harizopoulos. Column oriented database systems. *PVLDB*, 2(2):1664–1665, 2009.
2. S. Agrawal, S. Chaudhuri, L. Kollár, AP. Marathe, VR. Narasayya, and M. Syamala. Database tuning advisor for Microsoft SQL server 2005. In *Proceedings of VLDB*, Toronto, ON, Canada, pp. 1110–1121, 2004.
3. L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.
4. L. Arge and JS. Vitter. Optimal external memory interval management. *SIAM Journal on Computing*, 32(6):1488–1508, 2003.
5. R. Bayer and EM. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.
6. B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. On optimal multiversion access structures. In *Proceedings of the 3rd International Symposium on Advances in Spatial Databases*, Singapore, pp. 123–141, 1993.
7. B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion b-tree. *The VLDB Journal*, 5(4):264–275, 1996.
8. N. Beckmann, H-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of ACM SIGMOD*, Atlantic City, NJ, pp. 322–331, 1990.
9. MA. Bender, ED. Demaine, and M. Farach-Colton. Cache-oblivious b-trees. *SIAM Journal on Computing*, 35(2):341–358, 2005.
10. S. Berchtold, DA. Keim, and H-P. Kriegel. The x-tree: An index structure for high-dimensional data. In *Proceedings of VLDB*, Bombay, India, pp. 28–39, 1996.
11. GS. Brodal, G. Lagogiannis, C. Makris, A. Tsakalidis, and K. Tsichlas. Optimal finger search trees in the pointer machine. *Journal of Computer and System Sciences*, 67(2):381–418, 2003.
12. GS. Brodal, K. Tsakalidis, S. Sioutas, and K. Tsichlas. Fully persistent b-trees. In *SODA*, pp. 602–614, 2012. Available at http://siam.omnibooksonline.com/2012SODA/data/papers/498.pdf (accessed on April 15, 2013).
13. N. Bruno and S. Chaudhuri. An online approach to physical design tuning. In *ICDE*, Orlando, FL, pp. 826–835, 2007.

14. S. Chaudhuri and VR. Narasayya. Self-tuning database systems: A decade of progress. In *Proceedings of VLDB*, Vienna, Austria, pp. 3–14, 2007.
15. D. Comer. Ubiquitous b-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.
16. M. Dietzfelbinger. Universal hashing and k-wise independent random variables via integer arithmetic without primes. In *Proceedings of the 13th Annual Symposium on Theoretical Aspects of Computer Science, STACS'96*, Grenoble, France, pp. 569–580, 1996.
17. MC. Easton. Key-sequence data sets on indelible storage. *IBM Journal of Research and Development*, 30(3):230–241, 1986.
18. R. Fagin, J. Nievergelt, N. Pippenger, and HR. Strong. Extendible hashing: A fast access method for dynamic files. *ACM Transactions on Database System*, 4(3):315–344, 1979.
19. P. Ferragina and R. Grossi. The string b-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, 1999.
20. M. Frigo, CE. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *IEEE FOCS*, New York, pp. 285–298, 1999.
21. V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.
22. G. Graefe. Modern b-tree techniques. *Foundations and Trends in Databases*, 3(4):203–402, 2011.
23. G. Graefe and HA. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT*, Lausanne, Switzerland, pp. 371–381, 2010.
24. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of ACM SIGMOD*, Boston, MA, pp. 47–57, 1984.
25. S. Idreos, ML. Kersten, and S. Manegold. Database cracking. In *CIDR*, Asilomar, CA, pp. 68–78, 2007.
26. S. Idreos, ML. Kersten, and S. Manegold. Updating a cracked database. In *Proceedings of ACM SIGMOD*, Beijing, China, pp. 413–424, 2007.
27. S. Idreos, ML. Kersten, and S. Manegold. Self-organizing tuple reconstruction in column-stores. In *Proceedings of ACM SIGMOD*, Providence, RI, pp. 297–308, 2009.
28. S. Idreos, S. Manegold, and G. Graefe. Adaptive indexing in modern database kernels. In *EDBT*, Berlin, Germany, pp. 566–569, 2012.
29. I. Jimenez, J. LeFevre, N. Polyzotis, H. Sanchez, and K. Schnaitter. Benchmarking online index-tuning algorithms. *IEEE Data Engineering Bulletin*, 34(4):28–35, 2011.
30. I. Kamel and C. Faloutsos. On packing r-trees. In *CIKM*, Washington, DC, pp. 490–499, 1993.
31. I. Kamel and C. Faloutsos. Hilbert r-tree: An improved r-tree using fractals. In *Proceedings of VLDB*, Santiago, Chile, pp. 500–509, 1994.
32. I. Kamel and C. Faloutsos. Str: A simple and efficient algorithm for r-tree packing. In *ICDE*, Sydney, New South Wales, Australia, pp. 497–506, 1999.
33. A. Kaporis, C. Makris, G. Mavritsakis, S. Sioutas, A. Tsakalidis, K. Tsichlas, and C. Zaroliagis. Isb-tree: A new indexing scheme with efficient expected behaviour. *Journal of Discrete Algorithms*, 8(4):373–387, 2010.
34. G. Kollios, VJ. Tsotras, D. Gunopoulos, A. Delis, and M. Hadjieleftheriou. Indexing animated objects using spatiotemporal access methods. *IEEE Transactions on Knowledge and Data Engineering*, 13(5):441–448, 2001.
35. A. Kumar, VJ. Tsotras, and C. Faloutsos. Designing access methods for bitemporal databases. *IEEE Transactions on Knowledge and Data Engineering*, 10(1):1–20, 1998.
36. S. Lanka and E. Mays. Fully persistent b+-trees. In *Proceedings of ACM SIGMOD*, Denver, CO, pp. 426–435, 1991.
37. SS. Lightstone, TJ. Teorey, and T. Nadeau. *Physical Database Design: The Database Professional's Guide to Exploiting Indexes, Views, Storage, and More*. Morgan Kaufmann, San Fransisco, CA, 2007.
38. W. Litwin. Linear hashing: A new tool for file and table addressing. In *Proceedings of VLDB*, *VLDB'80*, Montreal, Quebec, Canada, pp. 212–223, 1980.

39. D. Lomet and B. Salzberg. Access methods for multiversion data. In *Proceedings of ACM SIGMOD*, Portland, OR, pp. 315–324, 1989.
40. M. Lühring, K-U. Sattler, K. Schmidt, and E. Schallehn. Autonomous management of soft indexes. In *ICDE Workshops*, Istanbul, Turkey, pp. 450–458, 2007.
41. Y. Manolopoulos, A. Nanopoulos, AN. Papadopoulos, and Y. Theodoridis. *R-Trees: Theory and Applications*. Springer-Verlag, New York, 2005.
42. B. Moon, HV. Jagadish, C. Faloutsos, and JH. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):124–141, 2001.
43. MK. Nguyen, C. Basca, and A. Bernstein. B+hash tree: Optimizing query execution times for on-disk semantic web data structures. In *Proceedings of the 6th International Workshop on Scalable Semantic Web Knowledge Base Systems*, *SSWS'10*, Shanghai, China, pp. 96–111, 2010.
44. LV. Nguyen-Dinh, WG. Aref, and MF. Mokbel. Spatiotemporal access methods: Part2 (2003–2010). *IEEE Data Engineering Bulletin*, 33(2):46–55, 2010.
45. R. Pagh and FF. Rodler. Cuckoo hashing. In *Proceedings of the 9th Annual European Symposium on Algorithms*, *ESA'01*, Berlin, Germany, pp. 121–133, 2001.
46. R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, Boston, MA, 2002.
47. N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed r-trees. In *Proceedings of ACM SIGMOD*, Austin, TX, pp. 17–31, 1985.
48. S. Šaltenis, CS. Jensen, ST. Leutenegger, and MA. Lopez. Indexing the positions of continuously moving objects. In *Proceedings of ACM SIGMOD*, Dallas, TX, pp. 331–342, 2000.
49. B. Salzberg and VJ. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys*, 31(2):158–221, 1999.
50. H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.
51. H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, San Francisco, CA, 2005.
52. T. Sellis, N. Roussopoulos, and C. Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of VLDB*, Brighton, U.K., pp. 507–518, 1987.
53. Y. Tao and D. Papadias. Mv3r-tree: A spatio-temporal access method for timestamp and interval queries. In *Proceedings of VLDB*, Rome, Italy, pp. 431–440, 2001.
54. Y. Theodoridis, M. Vazirgiannis, and T. Sellis. Spatiotemporal indexing for large multimedia applications. In *Proceedings of 3rd IEEE International Conference on Multimedia Computing and Systems*, Hiroshima, Japan, pp. 441–448, 1996.
55. R. Weber, H-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of VLDB*, New York, pp. 194–205, 1998.
56. DC. Zilio, J. Rao, S. Lightstone, GM. Lohman, AJ. Storm, C. Garcia-Aranello, and S. Fadden. Db2 design advisor: Integrated automatic physical database design. In *Proceedings of VLDB*, Toronto, Ontario, Canada, pp. 1087–1097, 2004.