

B-trees: Bearing Fruits of All Kinds

Beng Chin Ooi

Kian-Lee Tan

Department of Computer Science
National University of Singapore
3 Science Drive 2, Singapore 117543
Email: {ooibc, tankl}@comp.nus.edu.au

Abstract

Index structures are often used to support search operations in large databases. Many advanced database application domains such as spatial databases, multimedia databases, temporal databases, and object-oriented databases, call for index structures that are specially designed and tailored for the domains. Interestingly, in each of these domains, we find methods that are based on one distinct structure – the B-tree. Invented some thirty years ago, the B-tree has been challenged repeatedly, but has retained its competitiveness. In this paper, we first give a quick review of B-trees. We then present its adaptations to various domains. For each domain, we present representative B-tree-based structures and their search operations. We conclude that the B-tree is truly an ubiquitous structure that has stood the test of times with wide acceptance in many domains.

Keywords: B-tree, spatial databases, multimedia databases, high-dimensional databases, main memory databases.

1 Introduction

The B-tree index structure [4] is one of the most widely used and studied data structure. It is a balanced, multiway, external file organization in which every path from the root of the tree to a leaf of the tree is of the same length. Each non-leaf node (except the root node) in the tree has between $\lceil n/2 \rceil$ and n children, where n (called the *order*) is fixed for particular tree. A B-tree maintains its efficiency despite insertion and deletion of data, and is highly scalable (i.e., if you increase the size of a database by a factor of 100-400, you add just one level to the B-tree index!).

Invented by Rudolf Bayer and Ed McCreight thirty years ago, the B-tree has been challenged continually. Right from the beginning in 1969, despite its simplicity and efficiency, its value was not immediately recognized: simply splitting an overflowing page into two half empty pages and potentially wasting 50% of storage capacity was a real sacrilege (in the old days, the magnetic disk technology was immature and disk capacity was only 7.5 MB) [3]. Ironically (and interestingly), similar argument is upheld against the B-tree today by main memory database proponents, i.e., wastage of memory space.

In this paper, we first give a brief review of B-tree. We then present its adaptations to various domains, including temporal databases, spatial databases, high-dimensional databases, and main

memory databases. For each domain, we present representative B-tree-based structures and their search operations.

The rest of this paper is organized as follows. In the next Section, we give a quick overview of the family of B-trees. In Section 3, we discuss the basic paradigm that is used to exploit B-trees in most application domains. Sections 4-8 present some example domains in greater depth. Finally, we conclude in Section 9.

2 The Family of B-trees

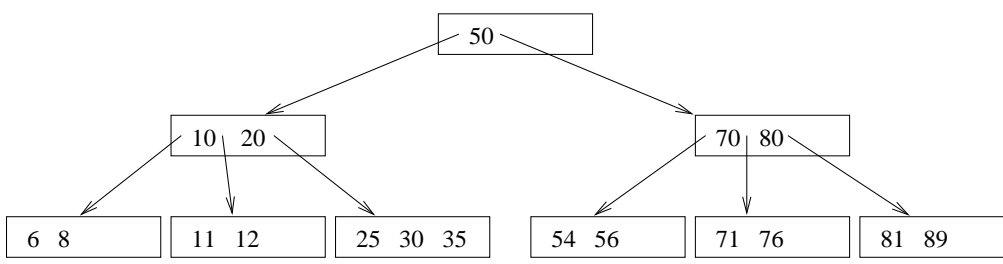
A B-tree of order n is a multi-way search tree with the following properties [4]:

- The root has at least two subtrees unless it is a leaf.
- Each non-root and each non-leaf node holds $k-1$ keys and k pointers to subtrees where $\lceil n/2 \rceil \leq k \leq n$.
- Each leaf node holds $k-1$ keys where $\lceil n/2 \rceil \leq k \leq n$.
- All leaf nodes are at the same level.

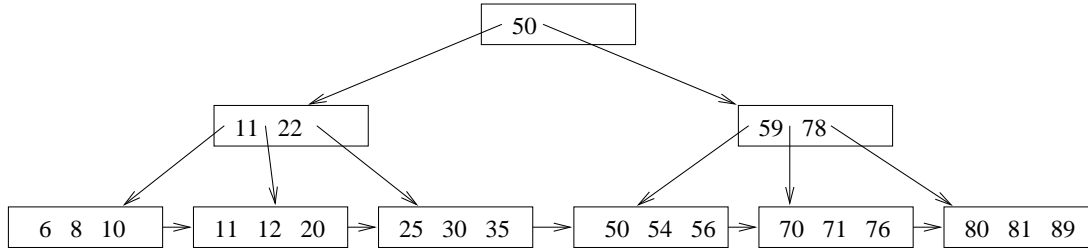
Thus, a B-tree is always at least half full, has few levels, and is perfectly balanced. Typically, each node corresponds to a disk page. Figure 1(a) shows a sample B-tree structure. Here, we have a set of 18 key values ranging from 6 to 89.

Since each node of a B-tree represents a page, accessing one node means one disk I/O. Therefore, the fewer nodes that are created the better it will be. To better utilize the storage capacity of the B-tree nodes, Donald Knuth introduced a variant called the B*-tree. In a B*-tree, all nodes except the root are required to be at least two-thirds full, not just half full, as in a B-tree.

While the B-tree structure provides fast exact match search (number of disk page accesses is equal to the height of the tree + 1 data page), range search requires a depth-first-traversal of the tree which may require examining some internal nodes multiple times. To reduce the computation overhead, the B⁺-tree is proposed. B⁺-tree differs from the B-tree in the following ways. First, all key values appear in the leaf nodes. Values in the internal nodes need not be keys - they serve only as *separators* to direct the search process. As shown in Figure 1(b), all the 18 keys are stored at the leaf nodes, and values 22, 59 and 78 are separators that do not appear in the database. Second, all the leaf nodes are linked. In this way, a range query $[a, b]$ requires searching for a , and examining the leaf nodes (by following the next-leaf pointer) until a value larger than b is found. Among the different variants, the B⁺-tree is one that has been implemented and used.



(a) B-tree



(b) B+-tree

Figure 1: The B-tree and B⁺-tree.

3 The Power of Transformation

Databases are increasingly being employed for advanced applications such as object-oriented databases, geographic and spatial databases, temporal databases, data warehousing, high-dimensional databases, and even XML databases. Many specialized indexing structures were and continue to be proposed to deal with these applications [7]. Among these, there are always B-tree-based schemes being pursued. Two reasons account for this. First, as B-tree is widely deployed in commercial systems, there is a need to be able to reuse it to allow these systems to cope with different applications' demands. Second, solutions based on B-trees are typically easy to implement and understand.

Almost all B-tree based schemes (even for different applications) are based on the paradigm of *transformation*: the key attribute is mapped into a single dimensional space so that B-tree can be used to index the transformed data. This is typically done either using linear-ordering techniques (such as z-order, morton order, peano-hilbert, row order, etc.), or using some transformation functions.

Clearly, applying transformation technique implies lost of information. In other words, it is possible for multiple key values to be mapped to the same value in the transformed space. However, because we are dealing with single dimensional data in the transformed space, the size of the attributes are typically small (compared to the original data). This gives rise to a small index search key, and hence small index size. As a result, the search process is very much faster.

Because of the transformation, query processing follows a two-phase strategy:

- In phase one, the query in the original data space is transformed to the single dimensional space in order for it to be evaluated on the transformed space. We note that typically a query in the original space may be mapped into multiple sub-queries in the transformed space.
- Phase two examines the returned answers and prune away false drops that may arise. This is necessary because it is typically the case that a

superset of the answers will be accessed due to the lost in information.

While it may appear that it is straightforward to apply transformation on any dataset to reuse B-trees, to guarantee good performance is a non-trivial task. In other words, a careless choice of transformation scheme can lead to very poor performance. Existing B-tree-based schemes, as we shall see in the next few sections, come with very clever observations made by their inventors. It is a combination of such insights with B-tree that make its performance comparable (and in many cases better) than specialized techniques.

Table 1 gives a list of some of the B-tree techniques that have been proposed in the literature, some of which will be presented in greater depth in subsequent sections. The list is certainly not exhaustive.

4 Indexing Low Dimensional Spatial

Spatial data are data associated with spatial coordinates and extents such as points, lines, polygons, and volumetric objects. Spatial data are multi-dimensional data and hence include high-dimensional data. However, when the field was initially investigated, the number of dimensions tend to low, and that affected the design of indexes for spatial applications. These spatial applications include computer-aided design (CAD), geographic information systems (GIS), computational geometry and computer vision. In these applications, retrieval of data is primarily based on spatial proximity relationships amongst objects rather than on attribute values of objects. For example, in GIS, a query may be "retrieve all restaurants that are *within* a certain distance from a particular road". *Spatial operators* like *intersection*, *adjacency*, and *containment* are much more expensive to compute than conventional relational *join* and *select* operators. This is due to the irregularity in the shapes of the spatial objects. For example, consider the intersection of two polyhedra. Besides the need to test all points of one polyhedron against the other, the result of the operation is not always a polyhedron but it may sometimes consist of a set of polyhedra.

Object oriented databases	SC-tree [14], CH-tree [14], H-tree [16]
Temporal databases	Monotonic B+-tree [9], Bitemporal B+-tree [13], Transaction time [19], Valid time [20]
String databases	Prefix B-tree [5], String B-tree [12]
Spatial databases	location-key based [1], UB-tree [24] Z-ordering [22]
High-dimensional databases	iMinMax [21], iDistance [26], UB-tree [24], Pyramid [6]

Table 1: B-tree based schemes.

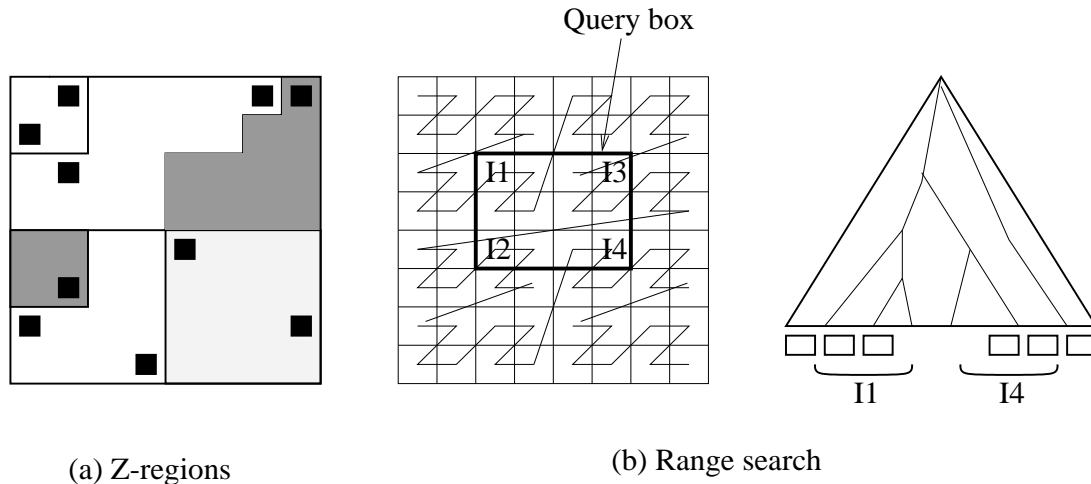


Figure 2: Z-regions of UB-tree, and its range search.

Object representation and indexing structures are treated separately in spatial database applications. Objects of complex shape are approximated by simpler containers for retrieval purposes. The aim of such approximation is to filter out as much false drops as possible before performing more expensive testing on the actual complex spatial objects. These objects are then indexed directly in their natural space or transformed and indexed in a parametric space.

For the latter approach, the data space is partitioned into grid cells of the same size, which are then numbered in certain fixed pattern. The idea is to assign a number to each representative grid in a space and these numbers are then used to obtain a representative number for the spatial objects. Techniques on ordering multi-dimensional objects using single-dimensional values have been proposed. These include the *Peano curve* [18], *locational keys* [1], *Z-ordering* [23], *Hilbert curve* [11], and *gray ordering* [10]. The mapping functions used in mapping must preserve the proximity between data well enough in order to yield reasonably good spatial search. A spatial object with extent is represented by a set of numbers or one-dimensional objects. These one-dimensional points are indexed using B⁺-trees.

One such prominent scheme is the universal B-tree (UB-tree) [2, 24]. In the UB-tree, the Z-ordering scheme is used to map a multidimensional space to single dimensional space. The *Z-value* of a point, computed by interleaving the bits of the bitstring representations of the values of the dimensions of the point, is the ordinal number of the point on the Z-curve. Based on the Z-value of the point, it can therefore be stored into a B⁺-tree.

The novelty of the UB-tree lies in the way it organizes the data space. Basically, it splits the space into disjoint *Z-Regions*, each of which is a space covered by an interval on the Z-curve. Moreover, each Z-region corresponds exactly to one page on the secondary storage, i.e., one leaf page of the B⁺-tree. Fig-

ure 2(a) shows a set of 10 points, and 5 Z-regions assuming each page can hold only 2 points (For visibility, we have draw each point smaller).

To process a multidimensional range query, the UB-tree computes the Z-regions that intersect the query. Each Z-region (which is an interval) corresponds to a range query on the B⁺-tree storing the Z-values. Figure 2(b) shows an example. While it is possible to precompute all Z-value intervals for a query, this can incur significant overhead as many of them may fall into the same Z-region. UB-tree avoids the overhead by constructing the intervals dynamically during runtime by processing the query page-by-page.

5 High-dimensional Indexing

Many emerging database applications such as image, time series and scientific databases, manipulate high-dimensional data. In these applications, two queries are common: the typical window queries and similarity queries. Similarity search or nearest neighbor search finds objects in the high-dimensional database that are similar to a given query object, and nearest neighbor search is a central requirement in such applications.

Indexes originally designed for low-dimensional databases deteriorate rapidly in performance with the increase in the dimensionality of data. This phenomenon is widely known as the dimensionality curse, and is caused by a few factors. One, the increase in dimensions reduces the number of entries that can be stored in an internal node of spatial indexes that capture spatial information. The smaller fan-out contributes not only to increased overlap between node entries but also the height of the corresponding spatial index. Two, as the number of dimensions increases, the area covered by the query increases tremendously. Three, the sparsity of data in high-dimensional databases and low contrast in distance

make high-dimensional indexing a very hard problem to solve.

Instead of designing a completely new indexing structure, we proposed two indexes, known as iMinMax(θ) [21] and iDistance [26], respectively for window and similarity queries.

In [21], we observed that firstly, data points in high dimensional space can be ordered based on the maximum value of all dimensions; secondly, if an index key does not fit in any query window, the data point will not be in the answer set. The first observation enables us to represent high-dimension data in single dimensional space, and reuse the existing B⁺-tree, while the second observation provides a mechanism to prune the search space. This leads to the design of the iMinMax(θ). The iMinMax(θ) adopts a simple transformation function to map high dimension points to a single dimension space. Let x_{min} and x_{max} be respectively the smallest and largest values among all the d dimensions of the data point (x_1, x_2, \dots, x_d) $0 \leq x_j \leq 1, 1 \leq j \leq d$. Let the corresponding dimension for x_{min} and x_{max} be d_{min} and d_{max} respectively. The data point is mapped to y over a single dimensional space as follows:

$$y = \begin{cases} d_{min} + x_{min} & \text{if } x_{min} + \theta < 1 - x_{max} \\ d_{max} + x_{max} & \text{otherwise} \end{cases}$$

where θ is a tuning knob that determines the edge (maximum or minimum) that a point will be mapped to, and hence can be used to optimize the transformation for different data distributions. We note that the transformation actually partitions the data space into different partitions based on the dimension which has the largest value or smallest value, and provides an ordering within each partition. With such a transformation, a B⁺-tree is used to index the transformed values.

Range queries on the original d -dimensional space have to be transformed to the single dimensional space for evaluation. In iMinMax(θ), the original query on the d -dimensional space is mapped into d subqueries — one for each dimension — for each subquery, a typical range search is performed. Let us denote the subqueries as q_1, q_2, \dots, q_d , where $q_i = [l_i, h_i]$ $1 \leq i \leq d$. For the j th query subrange in q , $[x_{j1}, x_{j2}]$, we have:

$$q_j = \begin{cases} [j + \max_{i=1}^d x_{i1}, j + x_{j2}] & \text{if } \min_{i=1}^d x_{i1} + \theta \geq 1 - \max_{i=1}^d x_{i1} \\ [j + x_{j1}, j + \min_{i=1}^d x_{i2}] & \text{if } \min_{i=1}^d x_{i2} + \theta < 1 - \max_{i=1}^d x_{i2} \\ [j + x_{j1}, j + x_{j2}] & \text{otherwise} \end{cases}$$

The union of the answers from all subqueries provides the candidate answer set from which the query answers can be obtained, i.e., $ans(q) \subseteq \cup_{i=1}^d ans(q_i)$. We note that the leaf nodes of the B⁺-tree contains the high-dimensional point, i.e., even though the index key on the B⁺-tree is only single dimension, the leaf node entries contain the triple (x_{key}, x, ptr) where x_{key} is the single dimensional index key of point x and ptr is the pointer to the data page containing other information that may be related to the high-dimensional point. Therefore, the false drop affects only the vectors used as index keys, rather than the data itself.

As the iMinMax(θ) does not capture information about metric distance or spatial relationship, it is not able to support exact K nearest neighbor search. However, the index is efficient for approximate KNN search when a small error can be tolerated.

Following the similar concept of partitioning data space, we proposed a technique called iDistance [26]

for KNN search that can be adapted based on the data distribution. For uniform distributions, it behaves like iMinMax(θ). For highly clustered distributions, it behaves as if a hierarchical clustering tree had been created especially for this problem.

In iDistance, the high-dimensional space is split into partitions either based on some space partitioning or data partitioning strategies. Each partition is associated with an *anchor* point (called reference point) whereby other points in the same partitions can be made reference to. These reference points are numbered, O_i ($i = 0, 1, \dots, m - 1$) where m is the number of partitions. A data point in a unit d -dimensional space, $p(x_1, x_2, \dots, x_d)$ $0 \leq x_j \leq 1, 1 \leq j \leq d$, is indexed based on the distance from the nearest reference point O_i as follows:

$$y = i * c + dist(p, O_i)$$

where $dist(p, O_i)$ is a distance function that returns the distance between p and O_i , and c is some constant to stretch the data ranges. Since distance is a scalar value, we used the B⁺-tree as the indexing structure. We note that an auxiliary structure must also be used to store the reference points.

Since the minimum radius required for retrieving the complete KNNs cannot be predetermined, an iterative approach that examines increasingly larger sphere in each iteration has to be employed. The algorithm works as follows. Given a query point q , finding K nearest neighbors (NN) begins with a query sphere defined by a *relatively small* radius r around q . r is increased gradually in each iteration until all the k nearest neighbors are obtained.

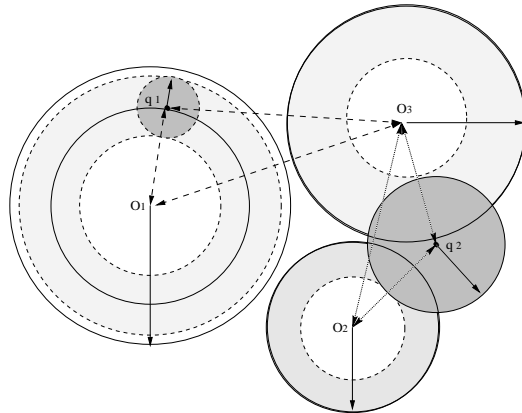


Figure 3: Searching for NN queries q_1 and q_2 .

Figure 3 shows an example with clusters in the same axis system. Here, for query point q_1 , only cluster S_1 is necessarily to be searched; for query point q_2 , both S_2 and S_3 have to be searched.

Searching in iDistance begins by scanning the auxiliary structure to identify the centroids whose data space (sphere area of cluster or data partition) overlaps with the query sphere defined by q_i and r . The search starts with a small radius, and step by step, the radius is increased to form a bigger query sphere. For each enlargement, there are three main cases to consider.

1. The data space S_i contains the query point, q_i . In this case, we want to traverse the data space sufficiently to determine the K nearest neighbors. This is done by first locating the leaf node where q_i may be stored. Since this node does not necessarily contain points whose distance are closest to q_i compared to its sibling nodes, we need to search left and right (inward and outward of data space) from the reference point accordingly.

2. The data space intersects the query sphere. In this case, we only need to search leftward (inward) since the query point is outside the data space.
3. The data space does not intersect the query sphere. Here, we do not need to examine the data space.

The search stops when the K nearest neighbors have been identified from the data subspaces that intersect with the current query sphere and when further enlargement of query sphere does not change the K nearest neighbors list. The stop criterion is when the distance of the K^{th} NN object to q is less than search radius r .

6 Indexing Temporal Databases

Typically, when data is updated in a database system, its old copy is discarded and the most recent version is retained. Conventional databases that have been designed to capture only the most recent data are known as *snapshot databases*. However, historical data is increasingly becoming an integral part of corporate databases. In such databases, versions of records are kept and the database grows as the time progresses. Data is retrieved based on the time for which it is valid or recorded. Databases that support the storage and manipulation of time varying data are known as *temporal databases*.

In a temporal database, the temporal data is modeled as collections of line segments. These line segments have a *begin time*, an *end time*, a *time-invariant attribute*, and a *time-varying attribute*. Temporal data can either be *valid time* or *transaction time* data. Valid time represents the time interval when the database fact is true in the modeled world, whereas transaction time is when a transaction is committed. A less commonly used time is the user-defined time, and more than one user-defined time is allowed.

One of the challenges for temporal databases is to support efficient query retrieval based on time and key. To support temporal queries efficiently, a temporal index that indexes and manipulates data based on temporal relationships is required. Valid time intervals of a time-invariant object can overlap, but each interval is usually closed. On the other hand, transaction time intervals of a time-invariant object do not overlap, and its last interval is usually not closed. Both properties present unique problems to the design of time indexes.

In [13], we show that temporal data can also be linearized so that the B^+ -tree structure can be employed without any modification. The proposed approach involves three steps: mapping temporal data into a two-dimensional space, linearizing the points, and building a B^+ -tree on the ordered points.

In the first step, the temporal data is mapped into points in a triangular two-dimensional space: a time interval $[T_s, T_e]$ is transformed to a point $[T_s, T_e - T_s]$. Figure 4 illustrates the transformation of the time interval to the spatial representation for the *tourist* relation. The x -axis denotes the discrete time points in the interval $[0, now]$, and the y -axis represents the time duration of a tuple. The points on the line named *time frontier* represent tuples with ending time of *now*. The time frontier will move dynamically along with the progress of time.

In the second step, points in the two-dimensional space are mapped to a one-dimensional space by defining a linear order on them. Given two points, $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$, the paper proposes three linear orders:

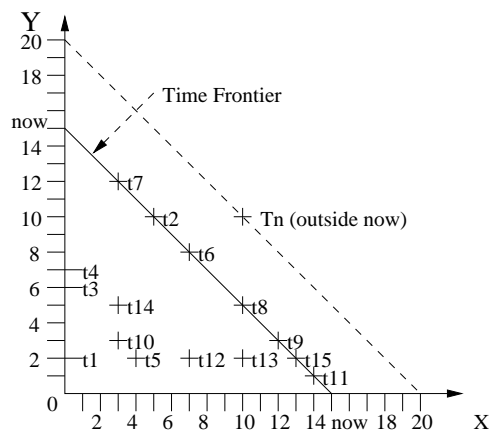


Figure 4: Spatial representation of a relation

- *D(iagonal)-order* ($<_D$). $P_1 <_D P_2$ iff (a) $(x_1 + y_1) < (x_2 + y_2)$; or (b) $(x_1 + y_1) = (x_2 + y_2)$ and $x_1 < x_2$.
- *V(ertical)-order* ($<_V$). $P_1 <_V P_2$ iff (a) $x_2 + y_2 = now$ and $x_1 < x_2$; or (b) $x_1 + y_1 \neq now$ and $x_2 + y_2 \neq now$ and $x_1 < x_2$; or (c) $x_1 + y_1 \neq now$ and $x_2 + y_2 \neq now$ and $x_1 = x_2$, and $y_1 < y_2$.
- *H(orizontal)-order* ($<_H$). $P_1 <_H P_2$ iff (a) $x_2 + y_2 = now$ and $y_1 < y_2$; or (b) $x_1 + y_1 \neq now$ and $x_2 + y_2 \neq now$ and $y_1 < y_2$; or (c) $x_1 + y_1 \neq now$ and $x_2 + y_2 \neq now$ and $y_1 = y_2$, and $x_1 < x_2$.

Figure 5 provides a graphic representation of the three linear orders defined above. By transforming the points through linearizing based on any of the above orders, we can index the temporal database using a B^+ -tree. A temporal query can be mapped to a spatial search on the two-dimensional space, which in turn can be translated to a range search operation on the linear space defined by the ordering relation. For example, consider the query “Find all persons who left the United States on or after day 5.” This query can be efficiently handled by traversing the D-order B^+ -tree and retrieving all points in the interval $[(0, 5), (14, 0)]$. However, not all temporal queries can be efficiently handled using the D-order. For example, consider the query “List all persons who entered the United States on or before day 5”. The D-order performs poorly for this query, while the V-order is superior, hence different indexes (constructed using different ordering relations) have to be used to support the various types of queries. However, the index is more suitable for valid times, which are mostly closed intervals. For data with open intervals, expensive reorganization is necessary. This highlights the common problem of transformation that one transformation may not be generic or efficient enough for all query types. Nevertheless, the main advantage of this method is the ease with which this indexing scheme can be implemented using existing DBMSs.

7 B^+ -trees in Main Memory

As main memory size increases, it is now possible to store the entirety of an index (and even a database!) in the main memory. It is now not uncommon to find affordable computer systems with main memory in the order of magnitude of gigabytes. For example, a SUN E450 machine with 2 CPUs and 4 GB RAM costs no more than US\$30K, and this price is expected to drop. For some commercial database systems, it is now possible to pin the whole index in the

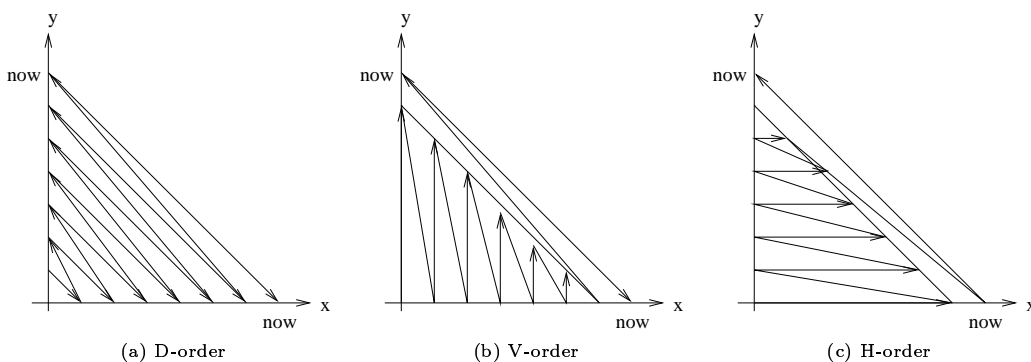


Figure 5: The three orderings for points in the two-dimensional space.

main memory to reduce I/O cost. However, conventional disk-based indexes are not cache conscious as they do not optimize on the utilization of L2 cache. Although the performance gain is achieved by reducing I/O reads and writes, it has not been maximized. To further improve performance, it is important to make effective use of the L2 caches. Consider the SUN Fire 4800 we used in our experiments for example, a L1 cache hit incurs about 2 processor clocks, while a L2 miss incurs more than 10 processor clocks when data is required to be fetched from slower but larger capacity RAM. Indeed, optimizing the utilization of L2 cache is somewhat similar to optimizing the utilization of RAM to the disk-based DBMS. Indexes that have been designed to exploit smaller but much faster cache memory are said to be cache conscious. Like the disk-based indexes which page the structure into 4K or 8K blocks due to the way data blocks are read from disk, cache conscious indexes page the structure based on cache lines. This indeed has generated renewed interests in designing memory-based auxiliary structures that optimize CPU cycles and memory space.

In main memory indexing, disk I/O cost is no longer an optimization parameter; instead, memory access, cache misses, translation lookahead buffer (TLB) misses and computation become the main costs. Traditional disk-based techniques (e.g., B⁺-trees), while applicable, are largely designed to minimize I/O cost, which are no longer an optimization issue in main memory systems. As such, they failed to exhibit the benefits that they are designed for. However, as CPU cycle and memory cache read improve drastically, the B⁺-tree proved to be cache conscious and yield good performance. In fact, it has been shown the the performance of the B⁺-tree is no longer inferior than the T-tree [15], which evolved from AVL Trees and B-trees and was designed specifically for the main memory indexing.

The CSB⁺-tree [25], called Cache Sensitive B⁺-tree, is a variant of B⁺-trees that stores all the child nodes of any given node contiguously, and keeps only the address of the first child in each node (see Figure 6). The rest of the children can be found by adding an offset to that address. Since only one child pointer is stored explicitly, the utilization of a cache line is high. CSB⁺-trees support incremental updates in a way similar to B⁺-trees.

Hash-based methods are very efficient for exact match query, but may perform poorly for range queries. In [8], we proposed a B⁺-tree called the HB-tree that incorporates the concept of hashing in its leaf node to support fast exact match and range queries (see Figure 7). We exploit the cache line size as the bucket size and squeeze in as many $\langle key, RID \rangle$ as possible to reduce cache misses. The performance study shows that the HB provides very

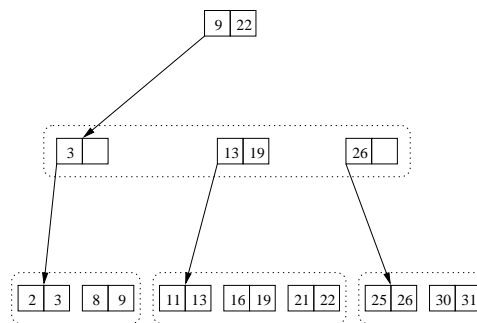


Figure 6: The CSB⁺-tree.

fast search for both exact match and range queries. It is a compromise between the fast random access provided by hashing and the range search provided by the B⁺-tree.

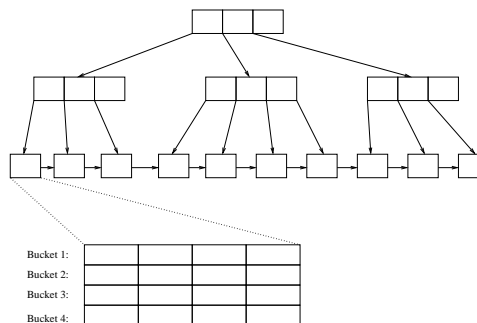


Figure 7: The HB⁺-tree.

The B-tree structure is suitable for exploiting cache line in main memory indexing in the same way it was proposed for earlier machines with small memory. This again illustrates the durability of the B-tree.

8 String Databases

String databases are becoming important due to the growth of string data in many real world applications (e.g., web documents, e-commerce data, genome databases). It is not uncommon to search these databases for substrings that match a query string. As such, the need for effective string indexes has been urgent. Here, we shall present a promising B-tree-based scheme, the String B-tree [12] whose worst-case performance is the same as that of the B-trees.

In the String B-tree, each string is stored in a contiguous sequence of disk pages, and the keys of the strings are *logical pointers* to the external memory

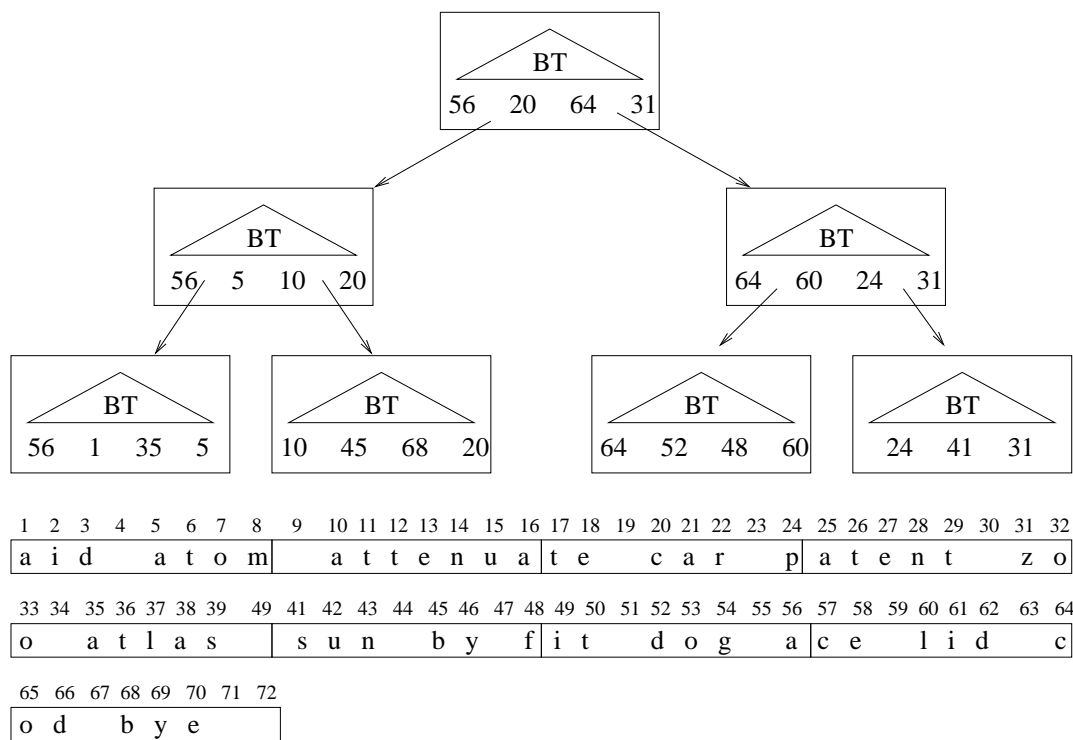


Figure 8: The String B-tree.

addresses of their first character. Consider the example shown in Figure 8 (see the bottom half). Here, we assume that the disk page size is 8 characters, and there are 15 strings {aid, atom, ..., bye}. The value 48 is the logical pointer to string “fit” and 14 is the logical pointer to suffix “nuate”. There is a space between two words to denote special endmarkers of a string.

The String B-tree comprises two logical levels. At the first level, it is a B⁺-tree where the logical pointers of the strings are stored at the leaf nodes of the tree (see upper portion of Figure 8). The pointers, however, are not ordered by their values, but rather by the lexicographical order of the strings. Thus, in the figure, we find 56 (representing string “ace”) appearing before key 1 (representing string “aid”). However, unlike conventional B⁺-tree, each entry in an internal node is the triplet (first key, last key, pointer), where pointer points to the subtree that contains the strings whose lexicographical orders are bounded by the order given by first key and last key. With this level, one can perform a prefix search as follows: Given a query string *S*, as the B⁺-tree is retrieved, the data pages that contain the strings corresponding to the logical pointers are retrieved. By comparing *S* with these strings, it can be easily determined which child node should be traversed next.

To improve on the performance, the second logical level is introduced in String B-tree. This level represents the internal organization of each node, i.e., each B⁺-tree node is organized as a Patricia trie [17]. By traversing the Patricia trie, fewer disk pages need to be retrieved.

The same scheme can be easily extended for substring search by storing all suffixes of all strings in the database, and perform a prefix search on the structure [12].

9 Conclusion

B-tree has come a long way. As we have seen in this paper, it has enjoyed 30 years of success not only in the traditional domain in which it was first designed

for, but also in many other domains. Its ability to adapt to “hostile climate” has made it one of the most fruitful tree. We conclude that the B-tree is truly an ubiquitous structure that has stood the test of times with wide acceptance in many domains.

References

- [1] D. Abel and J. Smith. A data structure and algorithm based on a linear key for a rectangular retrieval problem. *International Journal of Computer Vision, Graphics and Image Processing*, 24(1):1–13, 1983.
- [2] R. Bayer. The universal b-tree for multidimensional indexing: General concepts. In *World-Wide Computing and Its Applications*, pages 198–209, 1997.
- [3] R. Bayer. Acceptance speech for sigmod innovations award 2001. 2001.
- [4] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1(3):173–189, 1972.
- [5] R. Bayer and K. Unterauer. Prefix b-tree. *TODS*, 2(1):11–26, 1977.
- [6] S. Berchtold, C. Böhm, and H-P. Kriegel. The pyramid-technique: Towards breaking the curse of dimensionality. In *Proc. 1998 ACM SIGMOD International Conference on Management of Data*, pages 142–153. 1998.
- [7] E. Bertino, B.C. Ooi, R. Sacks-Davis, K.L. Tan, J. Zobel, B. Shilovsky, and B. Catania. *Indexing Techniques for Advanced Database Systems*. Kluwer Academic Publishers, August 1997.
- [8] B. Cui, B.C. Ooi, and K.L. Tan. Hash-based b+-trees for main memory access and join. In *submitted for publication*, 2001.

- [9] R. Elmasri, G.T.J. Wu, and V. Kouramajian. The time index and the monotonic b+-tree. In *Temporal Databases*, pages 433–456, 1993.
- [10] C. Faloutsos. Gray-codes for partial match and range queries. *IEEE Transactions on Software Engineering*, 14(10):1381–1393, 1988.
- [11] C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. In *Proc. 1989 ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 247–252. 1989.
- [12] P. Ferragina and R. Grossi. The string b-tree: A new data structure for string search in external memory and its applications. *J. ACM*, 46(2):236–280, 1999.
- [13] C.H. Goh, H. Lu, B.C. Ooi, and K.L. Tan. Indexing temporal data using existing b+-trees. *Data and Knowledge Engineering*, 18:147–165, 1996.
- [14] W. Kim, K.C. Kim, and A. Dale. Indexing techniques for object-oriented databases. In *Object-Oriented Concepts, Databases, and Applications*, pages 371–394. Addison-Wesley, 1989.
- [15] T. Lehman and M. Carey. A study of index structure for main memory management systems. In *Proc. 12th International Conference on Very Large Data Bases*, pages 294–303. 1986.
- [16] C.C. Low, B.C. Ooi, and H. Lu. H-trees: A dynamic associative search index for oodb. In *SIGMOD Conference 1992*, pages 134–143, 1992.
- [17] D.R. Morrison. PATRICIA — Practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.
- [18] G.M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. In *IBM Ltd.* 1966.
- [19] M. Nascimento. A two-stage b+-tree based approach to index transaction time. In *IADT*, pages 513–520, 1998.
- [20] M. Nascimento and M. Dunham. Indexing valid time databases via b+-tree. *IEEE TKDE*, 11(6):929–947, 1999.
- [21] B. C. Ooi, K. L. Tan, C. Yu, and S. Bressan. Indexing the edge: a simple and yet efficient approach to high-dimensional indexing. In *Proc. 18th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 166–174. 2000.
- [22] J. Orenstein. Spatial query processing in an object-oriented database system. In *SIGMOD Conference 1986*, pages 326–336, 1986.
- [23] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *Proc. 1984 ACM-SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 181–190. 1984.
- [24] F. Ramsaka, V. Markl, R. Fenk, M. Zirkel, K. Elhardt, and R. Bayer. Integrating the ub-tree into a database system kernel. In *VLDB 2000*, pages 263–272, August 2000.
- [25] J. Rao and K. Ross. Making b+-tree cache conscious in main memory. In *Proc. 2000 ACM SIGMOD International Conference on Management of Data*, pages 475–486. 2000.
- [26] C. Yu, B.C. Ooi, K.L. Tan, and H. V. Jagadish. Indexing the distance: An efficient method to knn processing. In *VLDB 2001*, pages 421–430, Roma, Italy, September 2001.