

KDB_{KD}-Tree: A Compact KDB-Tree Structure for Indexing Multidimensional Data

Byunggu Yu*
University of Wyoming
Department of Computer Science
yu@uwyo.edu

Ratko Orlandic
Illinois Institute of Technology
Department of Computer Science
ratko@cs.iit.edu

Thomas Bailey
University of Wyoming
Department of Computer Science
tbailey@uwyo.edu

Jothi Somavaram
University of Wyoming
Department of Computer Science
jsomavar@uwyo.edu

Abstract

The problem of storing and retrieving high-dimensional data continues to be an important issue. In this paper, we propose an efficient high-dimensional point access method called the KDB_{KD}-tree. The KDB_{KD}-tree eliminates redundant information in KDB-trees by changing the representation of the index entries in the interior pages. Experimental evidence shows that the KDB_{KD}-Tree outperforms other recent variants of KDB-trees, such as KDB_{FD}-trees and KDB_{HD}-trees.

1. Introduction

In many database applications, such as geographic information systems (GISs), traffic and environmental systems, cellular radio systems, and CAD/CAM, each data object is identified by multiple attributes. These applications are known as *multidimensional database applications*. Since the objects of these applications are typically represented as points in a multidimensional space, the retrieval techniques designed for these data are called *point access methods (PAMs)* [3,8]. Because of their conceptual simplicity and relatively good performance, the KDB-tree [6] and its variants [5,7] are still the most popular of all point access methods.

Like most access methods for data on secondary storage, the KDB-tree is just a hierarchy of index pages (nodes). At every level of the structure, the d -dimensional universe is recursively divided into hyper-rectangles. The *root page* represents the entire universe. The *leaf pages*

contain the actual *data entries*, each of which contains the *coordinates* of a d -dimensional point. Every *interior page* contains *index entries*, each of which is represented by a $\langle \text{region_descriptor}, \text{child_pointer} \rangle$ pair. Here, *region_descriptor* represents the hyper-rectangle enclosing all data points stored in the sub-tree rooted at the child page, which is indicated by the *child_pointer*.

Any given query region, which itself is a multi-dimensional rectangle (*window query*), initiates a selection process that starts at the root page and propagates downward, traversing potentially multiple paths in the tree. At each interior page, the entries are tested to select the child pages that satisfy the search predicate. In general, the search predicate tests whether the region of the given child page intersects with the given window query. Whenever a leaf page is accessed, the procedure selects all resident objects that fall within the query.

Many applications, such as image, multimedia, and scientific databases, represent their objects as points in very high-dimensional spaces. Since *coordinates* of a data point represent a vector of length d , as the number d of dimensions increases, the entry size becomes larger. As a result, the *page capacity* (maximum number of entries that can be stored in a page) is reduced. Therefore, the search procedure has to follow longer paths to reach the leaf pages, which decreases retrieval performance. This is one of the important causes that limit the performance of almost all contemporary access methods in high-dimensional situations.

Several techniques, such as the TV-tree [4], the Pyramid Technique [2], and the KDB_{HD}-tree [5], have been proposed to overcome the problems of traditional access methods in high-dimensional spaces. For example, the approach of the TV-tree is based on the observation

*This author's work was funded by NSF EPSCoR grant no. NSFLOC4304-4311COSCI and Informix Research Software Grant.

that in typical high-dimensional data sets only a small number of dimensions carry most of the relevant information. The idea is to store in the interior pages only a small number of features that discriminate well between the point objects and simply ignore the rest of the dimensions. Since fewer features are stored in the interior pages, the interior levels of the index structure are more compact and the spatial searches are more efficient. However, prior to inserting any object into the structure, one must decide which dimensions are important and how many of these dimensions should be used. Since these factors have significant bearing on the performance of the TV-tree, the structure is highly sensitive to bad administrative decisions.

The Pyramid Technique partitions a d -dimensional universe into $2d$ pyramids that meet at the center of the universe. Then, every pyramid is divided into slices parallel to the basis of the pyramid. The d -dimensional vectors, each of which represents a multidimensional data point, are approximated by one-dimensional quantities, called pyramid values, which are indexed by a regular B+-tree. However, the transformation results in a loss of spatial proximity as well as the enlargement of queries falling near the boundaries of the space.

In order to achieve higher performance in high-dimensional spaces, the KDB_{HD} -trees use two heuristic measures. One of the measures relates to the policy of node splitting. The other measure reduces the size of index regions. In higher dimensional space, every index region in the KDB -tree is split along a small subset of dimensions. Since each remaining dimension of the region extends over the entire side of the universe, it contributes nothing to the selectivity of the structure [5]. In the KDB_{HD} -tree, these remaining dimensions are automatically eliminated from the index descriptors.

Even though the KDB_{HD} -tree effectively attacks the problems of KDB -trees in high-dimensional spaces, the *region_descriptors* of the interior pages still contain redundant information. While the redundancies tend to be negligible in low-dimensional spaces, they can become significant in high-dimensional spaces. The retrieval technique proposed in this paper, which we call the KDB_{KD} -tree, eliminates the redundant information in KDB_{HD} -trees by changing the representation of the index entries in the interior pages. Experimental evidence will show that KDB_{KD} -trees outperform two variants of KDB -trees, including the KDB_{HD} -tree, especially in high-dimensional situations.

2. KDB_{KD} -Tree

2.1. Splitting Policy

Perhaps the most interesting aspect of the KDB -tree [6] is its splitting strategy in the interior levels. There are

two kinds of splitting strategies: *forced splitting (FS)* and *first division splitting (FD)* [5]. The FS policy of the original KDB -tree structure [6] extends the hyper-plane that divided the last overflowed leaf page to split the index entries into two groups. Therefore, all index regions which intersect the extended hyper-plane must be split into two subspaces. This process of dividing additional index regions (not just the one that was originally split) results in downward propagation of node splitting. In contrast, the FD-splitting divides an overflowed interior page along the dividing hyper-plane by which the first two child pages are separated. In this case each index entry falls on one or the other side of the first division plane in its entirety, and as a result, none of them need to be split. This solution eliminates the downward propagation of splits that occurs in the FS policy. Even though FD-splitting does not guarantee splitting interior nodes into more-or-less equal halves, this problem is confined to the interior nodes only. In [5], the variant of KDB -trees with first-division splitting is called the KDB_{FD} -tree.

2.2. Basic Idea

The basic idea of the KDB_{KD} -tree is to eliminate the redundant information from the KDB_{FD} -tree structure. Removal of redundant information increases the capacity of the index pages. Although the KDB_{HD} -tree was designed on the basis of the same idea, the KDB_{HD} -tree does not completely eliminate all the redundant information. Figure 1 shows three index regions. In the KDB_{HD} -tree, the index entry for region R1 is $\langle 1, 0, 0.4, cp1 \rangle$; for region R2, $\langle 2, 0.4, 0.5, 1, 1, cp2 \rangle$; and for region R3, $\langle 2, 0.4, 0, 1, 0.5, cp3 \rangle$, where $cp1$, $cp2$, and $cp3$ are child-page pointers. Here a value of 0.4 is shared by all three index entries, and a value of 0.5 is shared by two index entries.

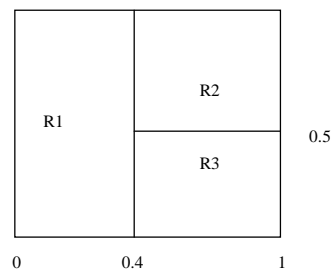


Figure 1. Three index regions

The elimination of the redundant information can be achieved by changing the structure of the interior page from a *list of entries* to a *KD-tree structure* [1]. All entries in an interior page are represented by a single KD -tree. For example, Figure 1 can be represented by a KD -tree as shown in Figure 2. To store a KD -tree in a page, we recursively traverse the KD -tree in the pre-order. For

example, the KD-tree in Figure 2 is represented by $|0.4|cp1|0.5|cp2|$ in the interior page. The KD-tree is a binary tree of nodes [1]. We modify the original KD-tree so that the leaf nodes (*pointer nodes*) point to child pages of the current page and the interior nodes (*division nodes*) represent the split values that divide the index region of the current page into those of the child pages. In this example, the division-node values are 0.4 and 0.5, and the pointer-node values are cp1, cp2, and cp3.

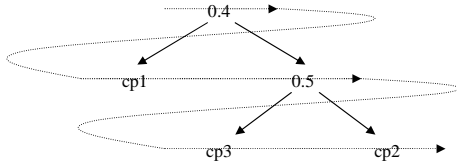


Figure 2. Page structure of the KDB_{KD} -tree

2.3. Design

Figure 3 shows an example of the KDB_{KD} -tree. Instead of a list of index entries, an interior page contains a KD-tree.

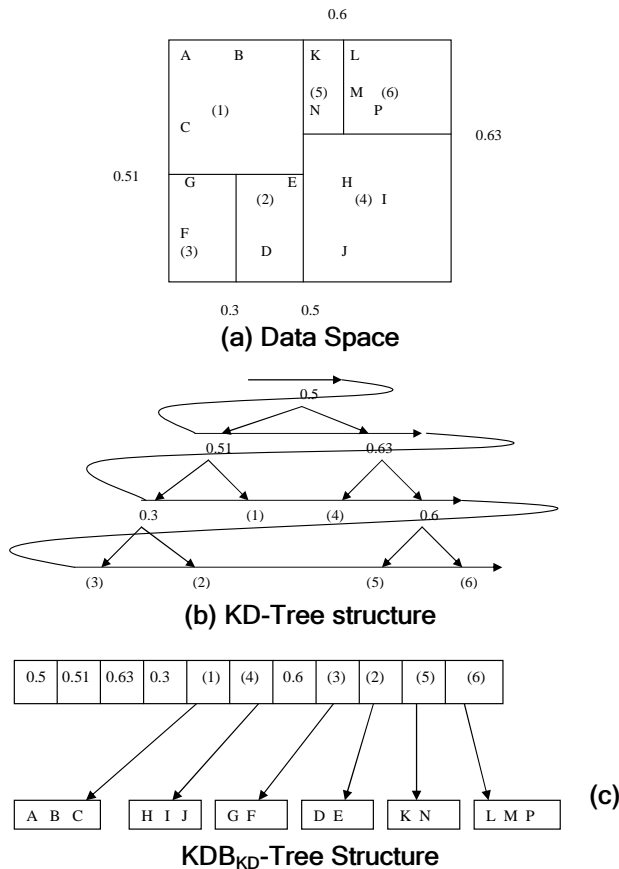


Figure 3. An example of the KDB_{KD} -tree

To insert a new object, the tree has to be traversed from the root to a leaf page. At each interior page, the KD-tree search algorithm is used to locate the child-page pointer. The KD-tree is traversed from the root node to a pointer node. At each division node, the left branch is chosen if the node value is greater than a new point. Otherwise, the right branch is chosen. This KD-tree search reaches the pointer node pointing to the right child page whose region encloses the given point. Once the correct leaf page is reached, the object is inserted into the leaf.

If the leaf page overflows, it is split using a modified KDB_{FD} -tree split algorithm. Since the KDB_{KD} -tree adopts the FD-splitting algorithm, the overflowed interior pages are split by the first division hyper-plane. If d is the dimensionality of the universe, then $(d-1)$ -dimensional hyper-plane that divides the region of the given interior page is perpendicular to one of the dimensions and it can be represented by a coordinate along that dimension. Since the root of the KD-tree represents the first division of the page, this coordinate is used as the root-node value of the KD-tree for the given page. Let O be an interior page containing a KD-tree K that needs to be split. Then, O is split into the left page O and the right page O' . The left page O retains the left sub-tree of K , whereas the right sub-tree of K is assigned to O' . Finally, in the parent page of O , the pointer to O is replaced by a small two-level binary-tree. The root, the left leaf node, and the right leaf node of this two-level binary-tree represent the root of K , the pointer to O , and the pointer to O' , respectively. Figure 4 gives a simple example of this split procedure.

To insert a data point into the index structure, the insert algorithm finds the leaf page whose region encloses the given point. The same procedure is used to process point queries. When the search reaches the leaf page, all data entries in the leaf page are tested, and the data entries whose point coordinates are the same as the coordinates of the given reference point constitute the result set.

The basic algorithm for processing window queries is the same as that of KDB_{FD} -tree. The search starts at the root page and propagates downward. At each interior page, the procedure selects all child pages whose regions intersect with the given query window. When the search reaches the leaf level, the data entries of the selected leaf pages are tested and those that satisfy the given query predicate are selected. This basic processing algorithm for window queries selects child pages at each interior page as follows:

- (1) **if** the high endpoint of the query window is smaller than the value of the current node along dimension i then choose the left-child node,
- (2) **else if** the low endpoint of the query window is greater than or equal to the value of the current node along dimension i then choose the right-child node,
- (3) **else** choose both child nodes.

Note, i is the dimension of the current KD-tree node.

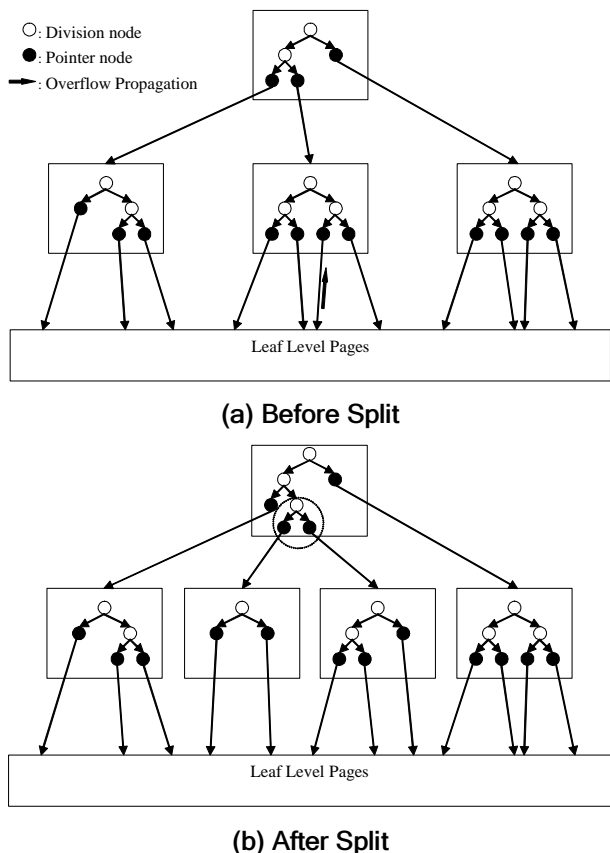


Figure 4. Splitting an interior page (assumption: the capacity of the interior pages is 7 KD-node values)

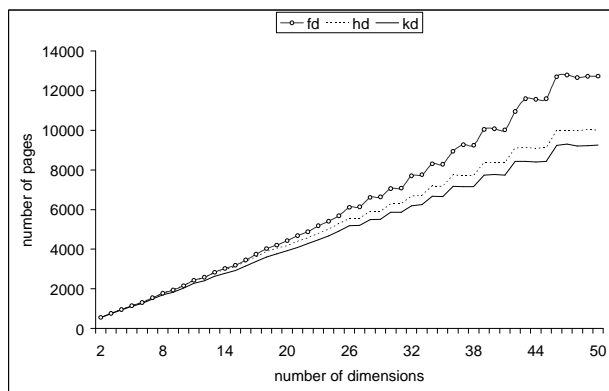
3. Experimental Evidence

An extensive set of experiments was conducted to compare the performance of the KDB_{KD} -tree with that of KDB_{HD} -trees and KDB_{FD} -trees. Since the experimental results in [5] show that KDB_{FS} -trees are inferior to the other variants of KDB -trees, we are concerned only with the performance of KDB_{FD} -trees, KDB_{HD} -trees, and KDB_{KD} -trees.

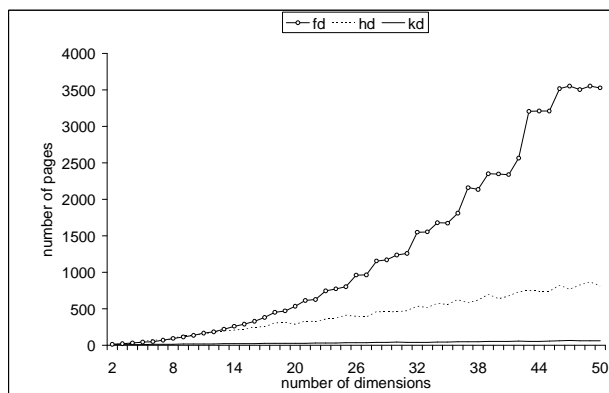
The number of dimensions was varied between 2 and 50. The page size for each structure was fixed at 2K bytes. In each d -dimensional structure, we inserted exactly 65,536 (2^{16}) points. The points were distributed uniformly over the entire universe. Figure 5 shows that, as the number of dimensions grew, the size of the KDB_{FD} -tree and the size of the KDB_{HD} -tree grew at a much faster pace than that of the KDB_{KD} -tree. Since the structure of the leaf-level pages is the same, the difference is significant in the interior levels. Obviously, the interior levels of the KDB_{KD} -tree are much smaller than the corresponding levels of KDB_{FD} -trees, KDB_{HD} -trees.

The performance of window queries was measured as the average number of page accesses over 3,000 randomly

generated queries (i.e., 1,000 point queries, 1,000 *contained_by* window queries, and 1,000 *covered_by* window queries). Note that, while a *contained_by* query looks for all data points that are in the interior of the reference region, a *covered_by* query is to find all data points that are in the interior or on the boundary of the reference region. For each point query, a reference point was randomly generated. For the window queries, each side of a query window was obtained as a pair of randomly generated numbers between 0 and 1. With this query generation, the average extent of the query windows along every dimension was about 1/3. Figure 6a shows the average number of page accesses. Figure 6b shows the percentage improvement for the average number of page accesses. For the “fd” that represents the percentage improvements of the KDB_{KD} -tree over the KDB_{FD} -tree, the percentages were obtained using the formula $100 \times (T_{FD} - T_{KD}) / T_{FD}$, where T_{FD} and T_{KD} are the average number of page accesses generated by all queries performed on a KDB_{FD} -tree and the corresponding KDB_{KD} -tree, respectively. For the “hd”, $100 \times (T_{HD} - T_{KD}) / T_{HD}$, where T_{HD} is the average number of page accesses generated by all queries performed on a KDB_{HD} -tree, was used.

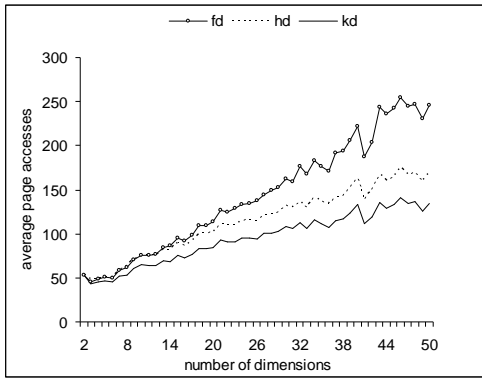


(a)

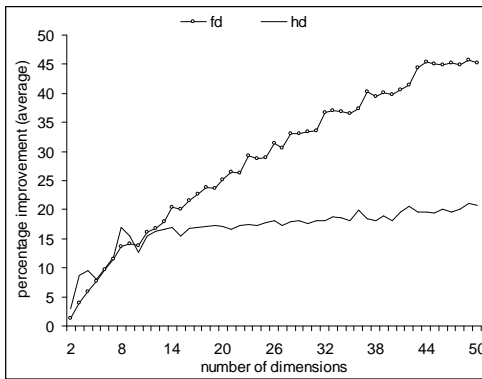


(b)

Figure 5. Size of index structures of (a) entire structure (b) interior structure



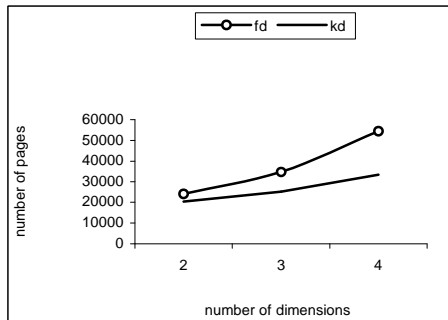
(a)



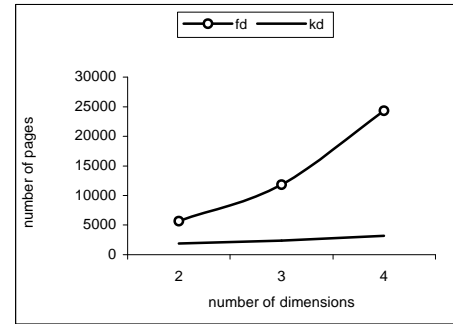
(b)

Figure 6. (a) Performance of the KDB_{FD} -tree, the KDB_{HD} -tree, and the KDB_{KD} -tree; (b) Percentage improvements of the KDB_{KD} -tree

Even for low-dimensional spaces the difference in size is significant (see Figure 7). Here, the number of dimensions was varied between 2 and 4. In this particular experiment, the page size was fixed at 128-byte to magnify the difference, and, in each d -dimensional structure, we inserted exactly 2^{17} points.



(a)



(b)

Figure 7. Size of (a) entire structure (b) interior structure

4. Conclusions

The problem of storing and retrieving data in high-dimensional spaces attracts considerable attention. In this paper we propose a KDB -tree variant that performs well in both low- and high-dimensional spaces. The guiding idea is to attack the limitations of KDB -trees in high-dimensional spaces, while retaining their good performance characteristics in low-dimensional spaces. By changing the representation of the interior pages of KDB -trees in such a way that duplicates are completely eliminated, the average capacity of pages is significantly increased. This results in more compact tree structure and better retrieval performance. Experimental evidence shows that the KDB_{KD} -Tree outperforms some recent variants of KDB -trees. As the number of dimensions increases, the performance improvements grow.

5. References

- [1] J. L. Bentley, "Multidimensional binary search trees used for associative searching", *Communications of ACM* 18(9), 1975, pp. 509-517.
- [2] S. Berchtold, C. Bohm, and H.P. Kriegel, "The Pyramid-Technique: Towards Breaking the Curse of Dimensionality", *Proc. ACM SIGMOD*, 1998, pp. 142-153.
- [3] V. Gaede, O. Gunther, "Multidimensional Access Methods", *ACM Computing Surveys* 30(2), 1998, pp. 170-231.
- [4] K. Lin, H. Jagadish, C. Faloutsos, "The TV-tree: An Index Structure for High-Dimensional Data", *VLDB Journal* 3, 1995, pp. 517-542.
- [5] R. Orlandic and B. Yu, "Implementing KDB -Trees to Support High-Dimensional Data", *Proc. IEEE IDEAS*, 2001, pp. 58-67.
- [6] J.T. Robinson, "The K-D-B - Tree: A Search Structure for Large Multidimensional Dynamic Indexes", *Proc. ACM SIGMOD*, 1981, pp. 10-18.
- [7] B. Seeger, and H. -P. Kriegel, "The buddy-tree: An efficient and robust access method for spatial database systems", *Proc. VLDB*, 1990, pp. 590-601.
- [8] T. Sellis, N. Roussopoulos, and C. Faloutsos, "Multidimensional Access Methods: Trees Have Grown Everywhere", *Proc. VLDB*, 1997, pp. 13-14.