# Indexing Multidimensional Data

Wentao Wu

May 9, 2011

**Abstract**

Many real-world applications, such as Geographic Information Systems (GIS), Computer-Aided Design (CAD), and multimedia databases, involve multidimensional data. Typical queries on these data include region queries and nearest neighbor queries, which cannot be well supported by standard indexing methods used in relational database such as B-tree and hashing.

The problem of indexing multidimensional data has been extensively studied for more than three decades. In this survey, we give a brief overview on existing indexing methods, including space-filling curves, grid files, kd-trees, quad-trees, and R-tree and its variants. We focus on the high-level ideas of the methods, and discuss the pros and cons they have.

## 1 Introduction

Many real-world applications involve multidimensional data. For instance, Geographic Information Systems (GIS) store and manipulate spacial data including points (e.g., locations of small objects), lines (e.g., rivers and roads), and two- or three-dimensional regions (e.g., buildings and lakes). Computer-Aided Design (CAD) is another area where spatial data are intensively entailed, such as surfaces of designed objects. In fact, even a traditional $k$-arity relation could be treated as a set of $k$-dimensional points.

In addition to straightforward queries on multidimensional data, such as "Find the students in CS department with age between 22 and 27 and GPA between 3.0 and 4.0", two more kinds of queries arise quite naturally. One is *region query*, for example, "Find all the gas stations within 1 mile of the CS building". The other is *nearest neighbor query*, for example, "Find the nearest UW Credit Union branch from the CS building". Multidimensional data management systems hence should have the ability of efficiently answering these queries as well.

Indexing techniques are important strategies for efficiently accessing the data stored within a database. In traditional relational database systems, B-tree is the standard scheme used to index data. B-trees can be built on either a single column or multiple columns of the table, and thus can also be used to index multidimensional data. However, B-trees can support neither region queries nor nearest neighbor queries well. The inherent difficulty is that, to build a B-tree index, a *total ordering* should be defined on the indexed dimensions. However, there is no such ordering that can perfectly preserve the spacial proximity that is naturally implied by these two new queries.

The problem of indexing multidimensional data has been studied extensively in the past three decades, and various indexing structures have been proposed in the literature. It's impossible to enumerate all of these techniques without a work of several books. Therefore in this short survey, we can only focus on a couple of most influential ideas in the history.

The rest of this paper is organized as follows. We will start with space-filling curves in Section 2, which naturally follows the basic idea of using B-tree to index multidimensional data, by defining a total ordering that better preserves spacial proximity. We then introduce several partition-based indexing methods in Section 3, including kd-trees, quad-trees, and grid files. Finally, in Section 4, we present $R$-tree and two of its important variants $R^+$-tree and $R^*$-tree, which may be the most successful indexing scheme on multidimensional data so far. We conclude the paper by giving some remarks in Section 5.

## 2 Space-Filling Curves

As briefly discussed in Section 1, B-tree based indexing scheme cannot well support region queries and nearest neighbor queries on multidimensional data, due to the lack of a good total ordering that can retain the spacial proximity of data points. Space-filling curves address this problem by introducing some total ordering that can preserve spacial proximity *better* than the *dictionary ordering* commonly used in B-tree. Two important space-filling curves were proposed in the literature. One is *Z-ordering curve* [1], and the other is *Hilbert curve* [2].

Z-ordering curve uses binary representation for the coordinates of the points, and assign an address to a point by interleaving the bits of its X and Y coordinates[1]. A linear ordering then can be defined on the points based on the addresses they receive. Z-ordering curve can achieve good spatial clustering of points. However, it suffers some problems like *long diagonal jump*, where two points with close X coordinates but distant Y coordinates will be defined as adjacent in the total ordering, which are by no means proximate in space. Hilbert curve improves Z-ordering curve by overcoming this long diagonal jump issue.

Nonetheless, the idea of space-filling curves can only provide some approximation on the notion of *spacial proximity.* No matter how the total ordering is defined, there are always some cases where points intuitively close to each other will be defined relatively far in the ordering. The inherent problem is that distance in a higher dimensional space cannot be completely captured in a space with lower dimension.

## 3 Partition-Based Indexing Methods

Instead of defining some one dimensional structure to approximate the proximity of points located in multidimensional spaces, partition-based indexing methods directly utilize all the dimensions in the space. We briefly examine three partition-based indexing methods here: *kd-trees* [3], *quad-trees* [4], and *grid files* [5].

### 3.1 Kd-Trees

Kd-tree is short for *k-dimensional tree*, which is a multidimensional extension of the well-known *binary search tree* data structure. Each node in the tree is a $k$-dimensional data point. Every non-leaf node could be imagined as generating a splitting *hyperplane* which divides the whole space into two subspaces, along the dimension chosen. Points with smaller coordinate values than the node on the selected dimension (i.e., points to the left of the hyperplane) will go to the left subtree of the node, while points with larger coordinate values (i.e., points to the right of the hyperplane) will go to the right subtree of the node.

[1]In this survey, we will use the case of 2-dimensional space as example when we need to illustrate the idea of a method, which is easier to imagine and present.

The whole procedure of constructing the tree hence can be thought as recursively bisecting the (sub)spaces until all the data points are processed.

kd-tree works well for region queries and nearest neighbor queries, according to the empirical study in [3]. However, it can only handle point data, and does not take paging of secondary memory into account. Also, since its structure may be distorted by skewed data, care should be taken to balance the kd-tree.

### 3.2 Quad-Trees

Different from kd-tree, which partitions the space based on the given data points, quad-tree uses a deterministic partitioning strategy. Each node in the tree corresponds to a square-shaped region of the data space. In particular, the root of the tree corresponds to the entire data space. Each internal node has *exactly* four children, with respect to the four *quadrants* within the space. The nodes (i.e., blocks containing points within the corresponding space) are encoded and addressed with Z-ordering and can be further indexed and accessed with B-trees.

Quad-trees can be quite efficiently used in answering region queries and nearest neighbor queries on very large geographical data [6]. However, it has some limitations. First, like kd-tree, it does not address the paging problem as well, and random accesses to the blocks seem quite likely since it uses Z-ordering based addressing mechanism. Second, it is possible but difficult to apply similar strategies into space with dimensionality higher than 2. For a $k$-dimensional space, the space represented by each internal node should be partitioned into $2^k$ subspaces. The performance and scalability is hence questionable in the case even when $k$ is not very large. For example, each internal node will have 1,024 children in a 10-dimensional space.

### 3.3 Grid Files

Grid file uses a partitioning scheme that can be thought of as extending the Extendible Hashing [7] idea into multidimensional spaces. A *grid directory* is used to guide the search, whose entries can be split and merged upon data insertion and deletion. Given a point, its X and Y coordinates will be searched individually to locate the directory entry that points to the bucket that contains it. The bucket is then fetched and the point is obtained. Hence at most two disk accesses are required to answer a point query, if the grid directory is also stored on the disk.

It is easy to use grid files to support region queries and nearest neighbor queries. Basically what we need to do is only to locate the buckets that are relevant to the queries. For region queries, they are the buckets that intersect the given window. For nearest neighbor queries, they are the bucket that contains and those buckets that are close to the given point. However, it also has some problems. First, it cannot well support region data. Second, it has scalability problems since the size of the directory increases exponentially with the growth of dimensionality.

## 4 R-Tree and Its Variants

R-tree, originally proposed by Guttman [8], is a dynamic tree-based index structure that can support both point and region data. The structure of R-tree is very similar to B-tree. Leaf nodes of R-tree contain the actual data, while internal nodes contain entries that are only used for search purpose. Each entry contained in an internal node is the *minimum* *bounding box* that can completely enclose all the objects in the child node pointed to by it. Since R-tree adopts a similar structure as B-tree, it can be optimized for paging on secondary storage. Meanwhile, R-tree can also support region queries and nearest neighbor queries quite naturally.

### 4.1 $R^+$-Tree

One important drawback of the original R-tree is that, the bounding boxes indicated in search entries can overlap with each other. Since R-tree will search all subtrees rooted at the entries that intersect the given query window, unnecessary checks are likely to happen, which degrade the performance. Motivated by this, $R^+$-tree [9] modifies R-tree by requiring that the search entries inside an internal node should be disjoint with each other. However, this brings new problems. First, to keep the original semantics of R-tree (i.e., each search entry is a bounding box of the underlying objects), objects may need to be split and stored within multiple nodes. This decreases the storage utilization and a $R^+$-tree will usually be larger than a corresponding R-tree. Second, introducing duplicates into the structure makes construction and maintenance more difficult.

### 4.2 $R^*$-Tree

$R^*$-tree [10] is another important variant of R-tree, which tries to minimize both coverage and overlap. The original R-tree only focuses on minimizing coverage, while the $R^+$-tree minimizes the overlap in an extreme way (i.e., disallowing any overlap). $R^*$-tree uses two strategies to reduce both. One is a revised node split algorithm. Node split occurs in R-tree when a node overflows, and Guttman proposes a quadratic and a linear split algorithm in his original paper, based on a set of heuristics. The split algorithm used in $R^*$-tree actually considers a different set of heuristics. The other strategy used in $R^*$-tree to improve performance is the concept of *forced reinsertion* at node overflow. It avoids split on certain node overflows by reinserting the entries of the node into the tree, which is the same as what is done in R-tree when a node is underfill. This strategy is based on the observation that reinserting entries can refine the spatial structure of the tree.

Basically speaking, the contribution of $R^*$ is just an improved implementation based on a set of new heuristics. However, these heuristics are strictly tested and demonstrated to work well. The empirical study in the paper is unbelievably detailed, and the resulted version of R-tree outperforms all other variants on performance and is robust against ugly data distributions. Meanwhile, implementation of $R^*$-tree is only slightly complex than R-tree. These aspects make the $R^*$-tree structure very attractive in practice.

### 4.3 Other Variants

There are also other variants of R-tree, such as Hilbert R-tree [11] and Priority R-tree [12]. We will not go down into more details here. There is a recent book [13] that gives a detailed exploration on R-trees.

## 5 Concluding Remarks

In this paper, we investigate the most influential structures used in indexing multidimensional data. Among all the structures covered, R-trees seem to be the most successful one, which have been widely implemented and found their way into commercial DBMS's. The original R-tree paper is far from to be perfect. However, it proposes a relatively simple but complete solution to a very important problem, which has comparable performance with more complex indexing structures. Moreover, by using a B-tree like structure, it naturally addresses the problem of paging on secondary storage, which is a desirable property for implementing it inside database systems. These are the key reasons why the *R*-tree paper has such a big impact and receives so many citations.

## References

[1] J. A. Orenstein, "Spatial query processing in an object-oriented database system," in *International Conference on Management of Data*, vol. 15, 1986, pp. 326–336.

[2] C. Faloutsos and S. Roseman, "Fractals for secondary key retrieval," in *PODS*, 1989, pp. 247–252.

[3] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of The ACM*, vol. 18, pp. 509–517.

[4] H. Samet, "The quadtree and related hierarchical data structures," *ACM Comput. Surv.*, vol. 16, no. 2, pp. 187–260, 1984.

[5] J. Nievergelt, H. Hinterberger, and K. C. Sevcik, "The grid file: An adaptable, symmetric multikey file structure," *ACM Transactions on Database Systems*, vol. 9, pp. 38–71, 1984.

[6] H. Samet, J. Sankaranarayanan, and H. Alborzi, "Scalable network distance browsing in spatial databases," in *SIGMOD Conference*, 2008, pp. 43–54.

[7] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible hashing - a fast access method for dynamic files," *ACM Trans. Database Syst.*, vol. 4, no. 3, pp. 315–344, 1979.

[8] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *International Conference on Management of Data.* ACM, 1984, pp. 47–57.

[9] T. K. Sellis, N. Roussopoulos, and C. Faloutsos, "The r+-tree: A dynamic index for multi-dimensional objects," in *VLDB*, 1987, pp. 507–518.

[10] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: An efficient and robust access method for points and rectangles," in *SIGMOD Conference*, 1990, pp. 322–331.

[11] I. Kamel and C. Faloutsos, "Hilbert r-tree: An improved r-tree using fractals," in *VLDB*, 1994, pp. 500–509.

[12] L. Arge, M. de Berg, H. J. Haverkort, and K. Yi, "The priority r-tree: A practically efficient and worst-case-optimal r-tree," in *Cache-Oblivious and Cache-Aware Algorithms*, 2004.

[13] Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, and Y. Theodoridis, *R-Trees: Theory and Applications (Advanced Information and Knowledge Processing).* Springer, 2005.