Revised

Tree-Based Indexes for Image Data

Leonard Brown
Le Gruenwald
The University of Oklahoma
School of Computer Science
Norman, OK, 73019
lbrown@cs.ou.edu
gruenwal@cs.ou.edu

Abstract

As in conventional DataBase Management Systems (DBMSs), to allow users to efficiently access and retrieve data objects, a MultiMedia DataBase Management System (MMDBMS) must employ an effective access method such as indexing and hashing. This paper provides a survey of tree-based multidimensional indexing techniques for MMDBMSs that maintain image data represented as feature vectors. These techniques support such data while maintaining desirable characteristics of a B-tree, an index structure most commonly used in traditional DBMSs.

In this survey, we provide descriptions of each tree as well as give examples of the different data organization schemes. We also describe the advantages and disadvantages of using each technique. In addition, we provide classifications of the trees using several different properties. These classifications should assist researchers in identifying the strengths and weaknesses of any new indexing technique they develop as well as help users determine the most appropriate data structure for their applications.

1. Introduction

An index is designed to facilitate searching for data records. In an index for a traditional DataBase Management System (DBMS), the values of a search key are sorted together. Generally, addresses are attached to the values that point to the locations of their associated data records. Using the search key values, then, a data record can be located quickly. In addition, this type of data structure supports locating records with their search keys in between a range of values. This is in contrast to other data access methods like hashing [KS91].

In a MultiMedia DataBase Management System (MMDBMS), users should be able to retrieve the data as efficiently as they can in a traditional DBMS. So, the images stored in an MMDBMS

should be indexed. Unlike traditional data, however, images are complex. Attempts to reflect this complexity usually result in images being represented as a set of values or attributes. For example, an image may be represented by the color of its primary object, the chain code representing the shape of that object, and a Boolean value indicating whether or not the image is a landscape. Such a set of attributes is referred to as a *feature vector*. When represented in this manner, each image becomes a point in a k-dimensional space, where k is the number of features in each vector [Fal96]. Figure 1.1 illustrates this using a three-dimensional feature vector. The vector contains the values 3.5 in the ***X***-dimension, 0 in the ***Y***-dimension, and 8 in the ***Z***-dimension.

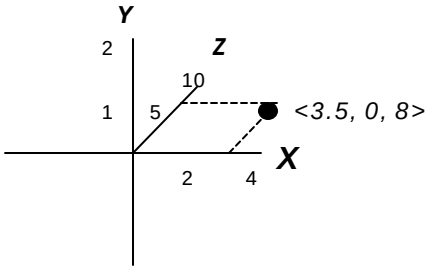

Figure 1.1 - Point in Three-Dimensional Space

Such a feature vector can be created for any image by applying some feature extraction algorithms to it. When the same group of general algorithms are applied to the set of images in an MMDBMS, the images would then be represented as feature vectors with the same dimensions. This means that they can be treated as different points on the same k-dimensional space.

When developing indexing methods for image data, then, researchers often operate under the assumption that the images are represented as feature vectors in the same multidimensional space. To access data of this type, any of the dimensions should be used. So, for an indexing technique for an MMDBMS to perform efficiently, it must be designed to search all dimensions of the data.

The indexing technique used in an MMDBMS should be able to efficiently satisfy several different types of queries. In [Fal96], the types of queries are classified into three categories. The first type is *range queries* that ask the system to return all data elements that are contained inside or

intersect with a specified region of the multidimensional space. This classification also refers to those queries that request images containing a particular set of attributes, called *subpattern matching* [LJF94]. Note that the specified region could be as small as a single point. In this case, the query is an *exact match* or *point query*.

As an example, consider the three-dimensional space displayed in Figure 1.1. A range query for this domain could be to return the points with values in the ***X***-dimension, between 0 and 2, values in the ***Y***-dimension between 1 and 2, and values in the ***Z***-dimension between 0 and 5. Figure 1.2 illustrates the region of space specified by this query as shaded. Any data elements that are contained in this region would be returned in response to this example query.

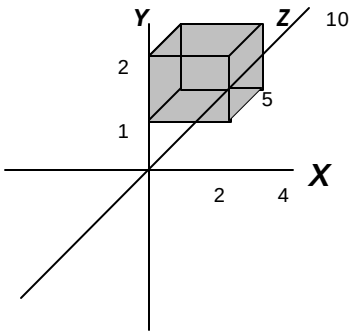

Figure 1.2 - Three-Dimensional Range Query

The second type of query listed in [Fal96] is called a *nearest neighbor query*. This query describes those queries in which the users request the n data elements that are the most similar to a specified region of the multidimensional space. This, of course, requires the MMDBMS to have a definition of "most similar".

An example of this query is illustrated in Figure 1.3. Using the Euclidean Distance as a measure for similarity, a user wants the top two images most similar to the point ⟨3, 0.5, 4⟩. This triple is the query point and is represented by the black point in the figure. The other points represent the feature vectors of the images in the database. The points that are shaded gray indicate the vectors that satisfy the query.

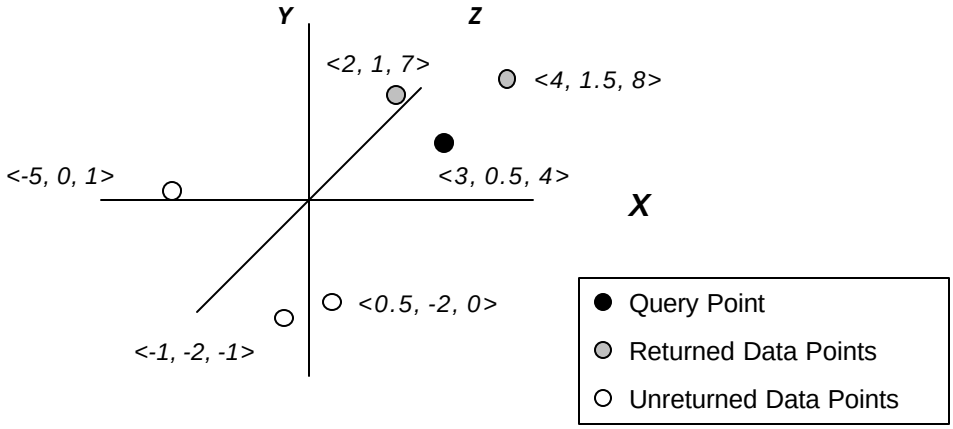

Figure 1.3 - Two Nearest Neighbors Query

The third type of query according to [Fal96] is a *spatial join*. These are queries that return pairs of data elements that are similar. An example of this is to find the redundant images in a database. For these queries, as with the nearest neighbor queries, a definition of similarity is needed. Satisfying these queries in an MMDBMS involves some function that takes two images as input and returns a scalar value as output. This value, called the distance, is then used as a measurement of the similarity between the two images. The goal of this type of query, then, is to find pairs of images whose distance is less than some specified value.

Figure 1.4 illustrates an example of a spatial join. In it, similar points are defined as those whose Euclidean Distance is smaller than 2. The only two points that have this property are shaded gray in the illustration. So, in response to this spatial join query, the MMDBMS would return ⟨-1, -2, -1⟩ and ⟨0.5, -2, 0⟩.

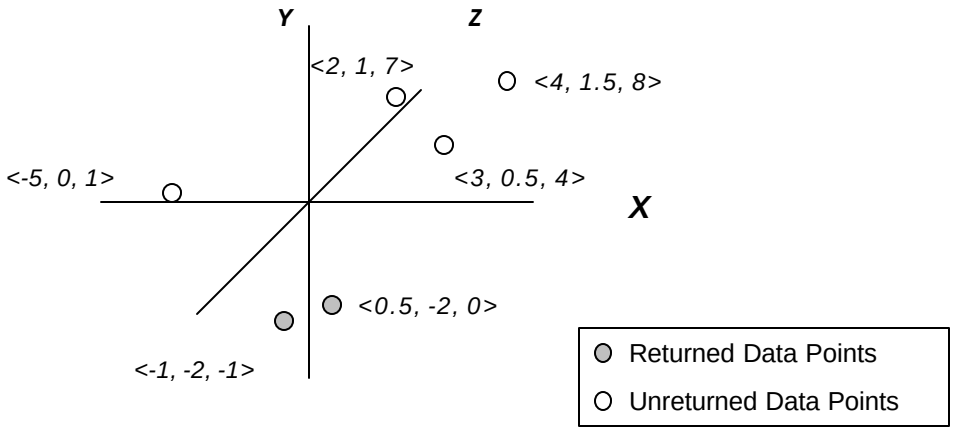

Figure 1.4 - Spatial Join

Appropriate multidimensional indexes are needed to efficiently solve each of above queries. First, for range queries, the index should provide an ordering of the data. Starting from the minimum value of any dimension, the index should facilitate successively moving to the next highest value until the maximum is reached. For nearest neighbor queries, the index should maintain information about the distances between data elements. Ideally, the MMDBMS should be able to move to the next closest neighbor of a data point using the index. Finally, the index should help satisfy spatial joins by grouping together points that are near one another. Thus, appropriate multidimensional indexes provide an ordering of the data, and group similar elements together.

This ability to order data is an advantage of indexing when compared to other data access methods such as hashing. This is because it is difficult to design an effective hashing function that preserves order [KS91]. One of the advantages of hashing, however, is that it allows the MMDBMS to directly compute the address of a data point, while indexes force the system to search for it using some data structure. This means that indexes are generally slower.

To reduce the time an index takes to locate data items, we need to minimize the number of data accesses it uses during the search. This can be accomplished using a tree-based index that stores the data elements in its leaves. Beginning at the root, we can traverse a single path down to one of its leaves to locate a particular data point. Thus, the number of steps a tree uses for searching is

proportional to its height. Because of this, much of the research into multidimensional access methods has been focused on developing tree-based indexes.

In addition to storing the k-dimensional points described earlier, many of the proposed trees are designed to store geometric shapes. These trees store the shapes using their Minimum Bounding Region (MBR). This refers to the smallest region, usually rectangular, that encloses the entire shape. Since these trees are storing regions, they have to resolve different issues than structures that do not use MBRs, such as maintaining overlapping data. This difference means that the multidimensional tree-based indexes that use MBRs should be categorized separately from the ones that do not.

This paper provides a survey of both types of the existing multidimensional tree-based indexes. The remainder of it is organized in the following manner: In Section 2, we will list the basic characteristics of tree-based multidimensional indexing techniques. In Section 3, we will describe indexing trees that do not use MBRs, and in Section 4, we will describe those that do. In Section 5, we classify the indexing structures based on the properties they use to organize data. Finally, in Section 6, we will summarize and indicate directions for future research.

2. Requirements for Tree-Based Techniques

A B-tree is one of the most popular methods in databases for indexing traditional data. The data structure allows efficient insertions and deletions while remaining balanced [Com79]. These properties should be present in a tree-based indexing structure for multidimensional data as well.

Unfortunately, as it is defined, the B-tree is inappropriate for multidimensional data. The data structure uses a single key to index the data records. To use the B-tree with multidimensional data, the data must be converted to a single dimension. One method of accomplishing this is to concatenate all of the attributes together into one long single-dimension string for each data object. The main drawback to this approach is that the dimension that is ordered first in the long string will have the priority in determining the distance between two data objects. Another drawback is that it is difficult to satisfy range queries using this technique [Kum94].

Researchers would like an indexing structure that is as effective for image data as a B-tree is for traditional data. Consequently, most of the research in this area has been directed to modifying the B-tree so it can effectively index multidimensional data. This means that most of the proposed trees in this paper will be variants of the B-tree. So, to describe these trees, it will be necessary to first describe a B-tree.

The B-tree indexes relations using their primary keys. In this tree, each node contains a list of key values and pointers. The tree limits the number of key values each node may contain. The number of values must be less than some maximum, M, and greater than some minimum, m, where M = 2m. The pointers refer to subtrees that only contain data elements grouped in the ranges created by the key values. For example, say a node had a set of k values x_1, x_2, ... , x_k. One pointer would point to the set values less than x_1. The next would point to the set of values between x_1 and x_2. The next refers to the set of values between x_2 and x_3. This pattern continues, and the next to last pointer in the node refers to the values between x_{k-1} and x_k. Finally, the last pointer refers to the values larger than x_k. [Com79].

An illustration of this tree is given in Figure 2.1. In this example, the value of M equals 2, so each node contains either 1 or 2 values. The root contains values 16 and 30. Its left child points to values that are less than 16, and its right child points to values greater than 30. Finally, its middle child points to values between 16 and 30.

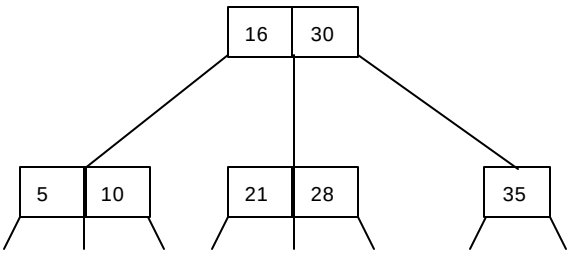

Figure 2.1 - A B-Tree

Since the multidimensional index trees are based on the same data structure, they share similar characteristics. Each node of the index trees corresponds to a specific section of a multidimensional data space, and each of its subtrees corresponds to specific subsections. In addition, each node stores

a set of values and pointers, like their counterparts in a B-tree. The set of values identifies the region of space corresponding to the node. For example, if the node represented a rectangular region of data space, the values would represent the upper and lower bounds along each dimension. The set of pointers refers to the children of the node. If the node is a leaf, these pointers refer to the actual data elements.

Also like the B-tree, each node has a limit on the number of regions it can contain, bounded above by some M, and below by some $m \leq M/2$. These limits are important when considering inserting and deleting data points. If a node is full, meaning that it already contains M data space partitions, inserting a region into it requires that a different region be removed and stored elsewhere. Generally, this requires a reorganization of an entire section of the tree. In Figure 2.1, an example of a full node is the one that contains the values 21 and 28.

One method of reorganization involves deleting some of elements of the full node, and reinserting them. A more common method, however, is to create a new sibling of the full node and store some of the data regions inside it. This operation used in the method is called a *split*, and is illustrated in Figure 2.2. To insert the value 25, the node containing 21 and 28 must be split. So, a new sibling to that node is created by adding a right child to the root. The value 28 is stored in the sibling. This allows the new value, 25, to be stored in the root.

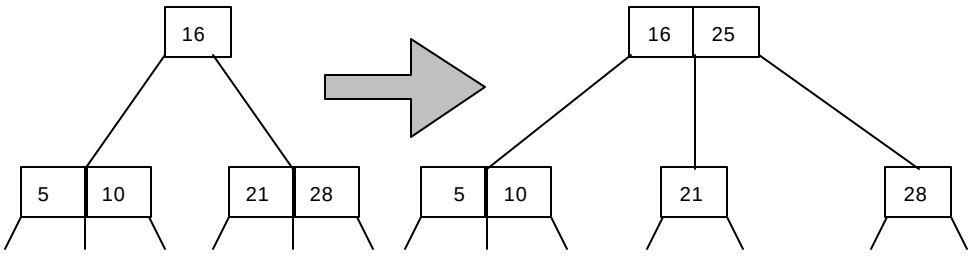

Figure 2.2 - The Split Operation

When a sibling is created for a node, a new pointer must be added to its parent. The parent could be a full node, which means that it would also have to be split. This means that the splitting

operation can propagate upwards. For some of the trees in this survey, however, the splitting operation can also propagate downwards, meaning that the children of the full node may be split as well.

Similarly, a delete operation on a node containing m data regions will result in a node containing too few elements. In these cases, the contents of the node will have to be combined with the contents from other nodes. This operation is called a *merge* which, like the splitting operation, can dramatically reorganize the structure of a tree.

In the following sections, we will describe the multidimensional tree-based indexes with these characteristics. We will list their unique approaches to partitioning the data space as well as describe their advantages and disadvantages.

3. Indexing Techniques Without MBRs:

The indexes described in this section are designed to store multidimensional points. In them, each node corresponds to a specific region of the data space, say R. The children of the nodes correspond to subregions of R. The nodes at the leaves contain pointers to the data elements that are contained in their respective regions.

The index trees in this section differ from one another based on their algorithms for splitting and creating the subregions for an internal node. Each method of partitioning a region has its own advantages and disadvantages. These properties are described in the following paragraphs:

K-D-B Tree:

The K-D-B tree was presented in [Rob81]. As described in Section 2, each of the internal nodes stores values to identify a section of the multidimensional data space, and a set of pointers referencing its children. The section identified by a node does not overlap with any of the regions of its siblings, and are created from splitting the larger regions corresponding to its parent. This split is performed along a single dimension. The dimension used for splitting alternates according to a round-

robin algorithm for all of the dimensions in the feature vector. This means that an ordering must exist for the dimensions.

The splits in the K-D-B tree are made in order to divide the data points in the resulting partitions as evenly as possible, and to minimize the number of splits. So, each region is split independently of the other partitions in the data space. This means that other partitions do not have to be considered when splitting a data region. [Rob81].

Figure 3.1 illustrates the data space partitioning for the K-D-B tree using the alternating dimensions. The data points are labeled with letters, and the regions are labeled with numbers. The lines in the data space indicate the divisions. For example, the first division partitioned the data into the two regions containing ***1234*** and ***5678***. The figure also illustrates that the next division in the left region occurred along the other dimension, and partitioned the data into regions ***12*** and ***34***. These are the regions that would be stored in a B-tree, as displayed in Figure 3.2.

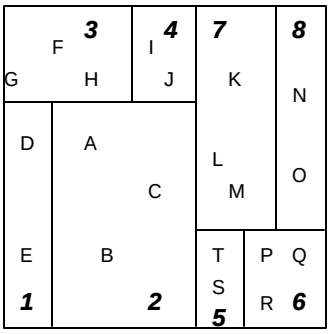

Figure 3.1 - Data Space Partition for the K-D-B Tree

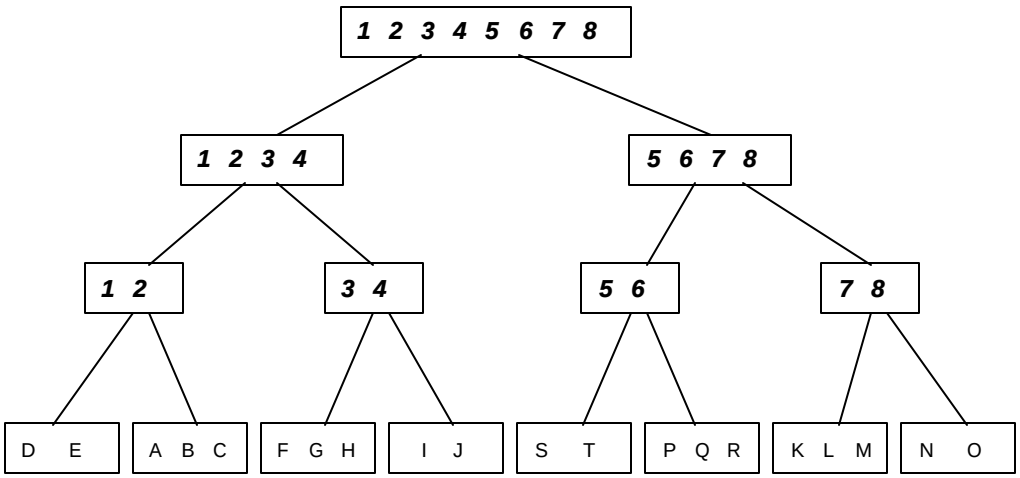

Figure 3.2 - K-D-B Tree Corresponding to Figure 3.1

G-Tree:

In the G-tree of [Kum94], the properties of a B-tree are combined with the properties of a grid file. The grid file is a data access method that divides the address space along each dimension. The file is so named because the divisions occur in a grid-like fashion. Each feature vector in this data structure, then, is mapped to the closest multidimensional grid point in the address space [NHS84].

The G-tree is a balanced index structure that, like the K-D-B tree, divides the data space into a set of nonoverlapping, rectangular regions. When a split occurs to a full node, its corresponding region is split in half with respect to the area of the region. This is in contrast to the K-D-B tree that splits in order to obtain an even number of data points in each region.

The split is performed along a single dimension, and the dimension used alternates using the round-robin scheme. Again, this implies that the dimensions must be ordered. Since the dimension alternates in an ordered fashion, a unique string can be used to identify each partition. Because each split creates two new partitions, the identifier can be a binary string.

So, the dimension used for splitting alternates on each level of the tree. In addition, since a region is formed by dividing another in half, the value used for splitting also does not have to be stored. This means that the position of each node in the G-tree directly identifies its corresponding region. So,

although its splitting procedure is more restrictive than the K-D-B tree, the G-tree has the advantage of requiring less storage. [Kum94].

Figure 3.3 illustrates the data space partitioning for the G-tree. As in Figures 3.1 and 3.2, the data points are labeled with letters, and the nonempty regions would be stored in a B-tree. These regions, however, are labeled by the binary strings corresponding to its location. For example, the data regions on the left of the first division are labeled beginning with 0, while the ones on the right begin with 1. Similarly, in the left half of Figure 3.3, the data regions at the top are identified starting with 01, while those at the bottom are identified starting with 00.

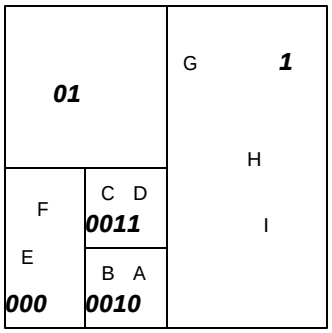

Figure 3.3 - Data Space Partition for the G-Tree

MB$^+$-Tree:

The MB$^+$-tree, presented in [YVD95], is so named because it is a modification to another common variation of the B-tree called the B$^+$-tree. In the B$^+$-tree, each of the data values is stored in the leaves, copied in the internal nodes in the tree whenever necessary. The leaves are linked together so that they may be searched quickly [Com79].

Like the G-tree, the MB$^+$-tree partitions the data space into several disjoint, rectangular regions. Also like the G-tree, the regions are ordered, and the tree is balanced.

The MB$^+$-tree partitions the data space by dividing it with several slices along the first dimension. As each partition becomes full, it must be split. The splits occur along the first dimension

until the resulting strips become too thin, meaning that the width is smaller than some predetermined value. When this occurs, the data region is divided independently of the other regions along the second dimension. The region will continue to be split in this manner until the strips become too thin with respect to the second dimension. At this point, the next dimension is split. This pattern continues as new elements are added to the tree. [YVD95].

So, one of the advantages of the MB^+-tree is that unlike the G-tree, the slices along a dimension may occur anywhere. So, if there are a large number of data points in a region clustered together, the MB^+-tree may still split the number of data elements evenly. This decreases the number of levels in the tree, and thus reduces the time spent for searching data elements. The disadvantage is that the value used for splitting must now be stored at each level.

Figure 3.4 illustrates a two-dimensional data space partitioning for the MB^+-tree. Again, the data points are labeled with letters. The data regions, however, would be stored in a B^+-tree. These regions are labeled using the order of the dimensions. For example, the data regions on the left are labeled beginning with 0, the labels in the middle begin with 1, and the labels on the right start with 2. The second number indicates the position of the region along the next dimension.

These labels can be used to order all of the data regions. With this single ordering, the data regions are arranged in a B^+-tree. This allows the search algorithm to directly move from one region to the next by using the linked list of leaves. [YVD95].

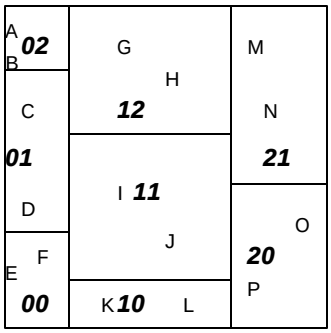

Figure 3.4 - Data Space Partition for the MB^+-Tree

BV-Tree:

The BV-tree was presented in [Fre95]. It was designed to overcome one of the disadvantages of the K-D-B tree that can occur when an insertion of an element into a data space partition causes its corresponding node to split. Specifically, the split may propagate downward, meaning that the full node's children may have to be split along the same division used to split the node itself. This is a disadvantage because the division that separates the node's data elements evenly may not divide its children evenly. So, it is possible that many nodes in the tree may become sparse [Fre95, LS90].

To avoid this problem, the BV-tree uses the concept of promoting data space partitions during splits. A partition is promoted by moving it from its current node in the BV-tree to its parent. This is done whenever one of the data space partitions developed from a split is completely contained inside another partition below it. The lower one must be promoted to the level of the higher partition and is referred to as its *guard*. So, instead of the splitting operation propagating to the lower partition, it is simply promoted. [Fre95].

An example of a promotion is displayed in Figure 3.5. Assume Data Space Partition 1 is a subset of Partition 5, the shaded region. Partition 5, then, must be promoted along with its children so that it is on the same level as Partition 1. After this promotion, Partition 5 is the guard of Partition 1.

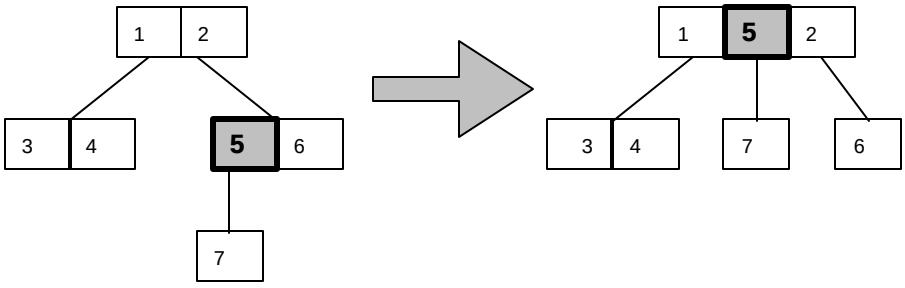

Figure 3.5 - Promotion of a Data Space Partition in the BV-tree.

One of the drawbacks that arises from promoting guards is that the searching algorithm defined for the BV-tree in [Fre95] is more complex. Although promoted, the guards must be searched as if they are still at their original position in the tree. This requires storing them as the search proceeds down

the tree to the leaves. At each level in the tree, the appropriate guard must be compared with the current node to determine the next child to be accessed in the searching algorithm. [Fre95].

hB-Tree:

Like the BV-tree, the hB-tree or holey-brick B-tree of [LS90] was designed to overcome the disadvantages arising from splitting full nodes in K-D-B trees. To prevent the splits from propagating downward into the children, the hB-tree uses the concept of "holey-bricks". This refers to the regions that form when the data space is partitioned.

To divide the data space, the divisions of the region may occur along more than one dimension. This permits creating rectangular holes in the region by partitioning along pieces of several dimensions. This gives more flexibility in how the data is partitioned. This flexibility provides a solution for the situations where cutting through a node's region using one entire dimension would mean splitting the subregions of the children.

Not only does this prevent splits from propagating downwards, but it also distributes data elements evenly in situations where a single-dimensional cut cannot. To illustrate, consider the example presented in [LS90]. It is based on a hypothetical set of data points arranged in a plus sign in two-dimensional space. One single-dimensional division of the data will not result in an even number of points in the new partitions. One of them will contain at least 10 of the 13 data points. Figure 3.6 illustrates this uneven split of the data points by dividing the data points into sets S1 and S2.

To divide the data more evenly, a two-dimensional split of the rectangular region must be used. Such a division allows the hB-tree to create two partitions where one contains six points, and the other contains seven. Figure 3.7 [LS90] illustrates such a two-dimensional split of the data points into sets S1' and S2'.

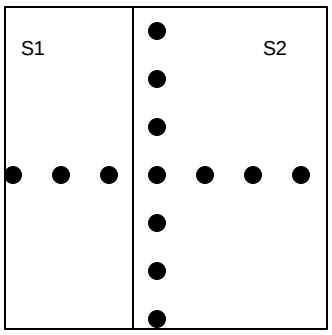

Figure 3.6 - Single-Dimensional Split of Data Points [LS90]

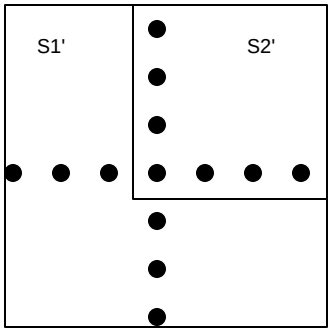

Figure 3.7 - Two-Dimensional Split of Data Points [LS90]

A disadvantage of this indexing scheme is that a more complex representation is required for storing the nodes. This is because a node now potentially corresponds to a rectangular region of the data space with several holes removed. Both the rectangular region and its holes must be represented in the node. In addition, as defined in [LS90], many of the nodes of the hB-tree may have more than one parent which means that this data structure is actually not a tree. Despite this, a data element can be located by traversing only one path from the root to a leaf. [LS90].

VP, MVP-Trees:

Although many trees partition the data based on their positions along a dimension, the Vantage Point Tree, or VP-tree, introduced in [Yia93] uses a different technique. At each node in the tree, it

selects one of the data points to function as a vantage point, and the division of the remaining data points is based on their distances from it. This is accomplished by sorting the data points based on their distances from the vantage point, then dividing them into n groups for a n-ary VP-tree. The division is performed so that the number of data points in each partition is as even as possible. Because it is based on the distance from a point, the partitions of a VP-tree are spherical instead of rectangular. [Yia93, Chi94].

One of the advantages of the VP-tree is that it is naturally suited to solve nearest neighbor queries since the divisions of the data points are based solely on their relative distances from one another [Yia93, Chi94]. A disadvantage, however, is that when the data space has a large number of dimensions, the partitions become very thin. This results in an increase in the number of branches of the tree that may be searched [BO97].

Because of the disadvantage, a modification to the VP-tree was proposed in [BO97]. It is called the Multi-Vantage Point Tree (MVP-tree). Instead of using one vantage point at each node, the MVP-tree uses two. The first vantage point is used to create m partitions, where m is the order of the tree. In each of these partitions, the second vantage point creates m more divisions for a total of m^2 partitions. The advantage this gives us is that we can create more divisions per internal node of the index tree. This increases the fanout, which reduces the search time. [BO97].

Figures 3.8 and 3.9 illustrate the VP-tree and MVP-tree spherical data space partitioning, respectively. Again, the data points are labeled with letters. In Figure 3.8, the data points are partitioned into three groups based on vantage point **V**. The first group consists of the data elements that are the closest to **V**, which are A, B, C, and D. The second group contains the next closest data elements, specifically E, F, G, and H. The last group contains the remaining data elements I, J, and K.

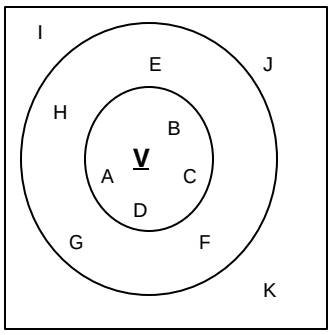

Figure 3.8 - Data Space Partition for the VP-Tree

In Figure 3.9, four data space partitions are created from two vantage points, **V1** and **V2**. The first vantage point divides the data into two groups, one consisting of A, B, C, and D, and the other consisting of E, F, G, and H. Vantage point V2 divides each of these partitions into two more groups. Based on their distances, A and B would belong in one partition, and C and D in another. Similarly, points E and F would be in a division separate from points G and H.

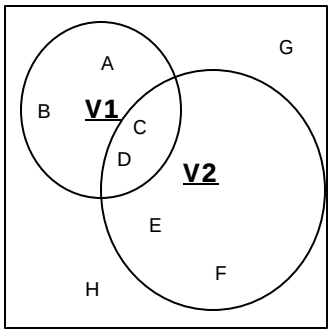

Figure 3.9 - Data Space Partition for the MVP-Tree

LSD-Tree:

The LSD-tree, or Local Split Decision Tree, was presented in [HSW89]. This tree is so named because the criteria used for splitting is performed independently for each rectangular partition. The split is not restricted to any specific dimension or whether or not it must divide the data space in half. This

means that we may split using any direction and any value we choose. A possible partitioning is illustrated in Figure 3.10.

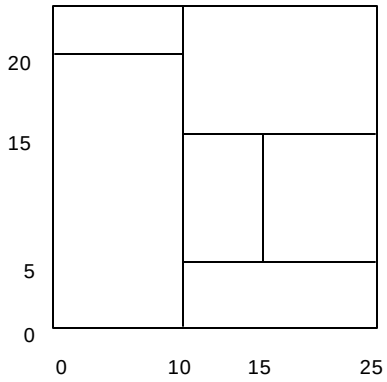

Figure 3.10 - Data Space Partition for the LSD-Tree

This flexibility has a drawback, however. Since any dimension can be used for partitioning at any time, the splitting information must be stored for each node. Thus, a directory is needed to store the partitions. Specifically, the directory must store the dimensions and data values used for splitting. [HSW89].

The directory for Figure 3.10 is displayed in Figure 3.11. Each node stores the division as a double ⟨D, V⟩, where D stands for the Dimension used, and V stands for the Value. A node's left child represents the data space for those values smaller than its division, while the right child represents the larger ones.

For example, in Figure 3.10, the first division occurred using the value 10 along the horizontal dimension, say 1. This means that the double <1, 10> is stored in the root of the directory tree. All data points with a horizontal value less than of 10 will stored to the left of the root, and all points with a horizontal value greater than 10 will be stored to the right.

The next partition divides the left side along the vertical or second dimension using a value of 20. Another divides the right side along the same dimension using a value of 15. So, the doubles <2, 20> and <2, 15> are the left and right child of the root.

The next division divides the points with a value in the first dimension greater than 10 and a value in the second dimension less than 15. This division occurs along the second dimension at the value 5. Thus, the left child of <2, 15> will be the division <2, 5>.

Finally, the top half of this last partition is divided into two groups based on whether their value in the first dimension is less or greater than 15. So, the right child of <2, 5> will be the double <1, 15>. This directory is illustrated in Figure 3.11.

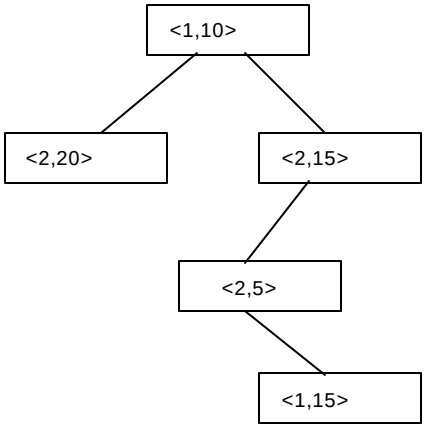

Figure 3.11 - LSD-Tree Directory for Figure 3.10

4. Indexing Techniques Using MBRs:

As stated earlier, the data structures in Section 3 are designed to store data elements that are points. For these trees to store data elements that are geometric shapes, they must first be converted to points using some transformation technique. For example, by using the upper and lower bounds in each dimension, a k-dimensional rectangle can be identified by a point in a 2k-dimensional space [Jag90].

By contrast, the data structures in this section can store geometrical shapes without such a transformation. These trees group data elements together just as the structures in Section 3, except that they store their Minimum Bounding Region (MBR). As stated in Section 1, this is the smallest region, usually rectangular, that covers all of the geometric shapes in its group. Figure 4.1 illustrates an example of a rectangular MBR.

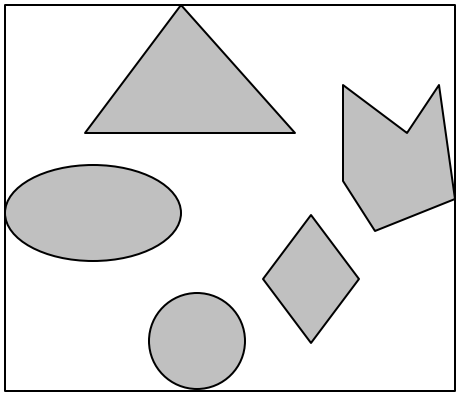

Figure 4.1 - MBR for a Set of Shapes

Each node of the index trees described below corresponds to the MBR of its group of shapes. Like the trees in Section 3, when an insertion is applied to a full node, the elements are split into two groups. Each of these groups is stored in a new node containing its respective MBR.

The trees in this section differ based on the properties of the MBRs. These properties include the shape of the MBRs, the algorithms used for splitting, and the amount the bounding regions are permitted to overlap.

R-Tree, R^+-Tree, R*-Tree:

The R-tree, or Rectangle Tree, of [Gut84] is one of the more popular data structures for indexing spatial data. In this tree, each node contains tuples of the form ⟨I, ptr⟩, where ptr is the address of a child of the node, and I indicates its rectangular MBR by identifying the upper and lower bounds for each dimension. Similar to the values in a B-tree, the number of tuples in each node of an R-tree cannot be greater than some M and less than some $m \leq M/2$. Also, like the B-tree, the R-tree is balanced. [Gut84].

As the B-tree has popular variants called the B^+ and B*-tree, the R-tree has the R^+ and R*-tree. The R^+-tree, presented in [SRF87], differs from the R-tree in that the MBRs are not permitted to overlap. This is accomplished by allowing spatial data elements to be split among different nodes in the

tree. When MBRs are permitted to overlap, the search algorithm must traverse multiple paths of the tree. If they do not overlap, however, a shape can be located using only one path of the tree. The drawback to this method of avoiding overlap is that the algorithm for deleting a node is more complex in an R^+-tree than in an R-tree. [SRF87].

The R*-tree of [BKSS90] is more like the R-tree in that it allows overlapping minimum bounding rectangles. The R*-tree differs from the R-tree in that it determines each MBR based on its area, margin of space, and overlap with other MBRs, where the R-tree determines an MBR based simply on its area. In addition, the R*-tree uses a concept called Forced Reinsert, which tries to prevent splits by deleting, then reinserting elements of a full node. [BKSS90].

Figure 4.2 illustrates a set of MBRs for an R-tree. The data regions are labeled with letters, and the MBRs are labeled with numbers. The rectangle surrounding the entire figure is labeled ***1***. This rectangle covers three smaller MBRs, ***2***, ***3***, and ***4***. Finally, these smaller rectangles are each an MBR for a set of data regions.

The corresponding R-tree is described in Figure 4.3. As indicated earlier, in the node labeled ***1***, there would exist values indicating the upper and lower bounds of the MBR along each dimension. In addition, there would be pointers to each of its children, namely the nodes labeled ***2***, ***3***, and ***4***. They each would store the values of the boundaries of their respective MBRs as well as pointers to their data regions.

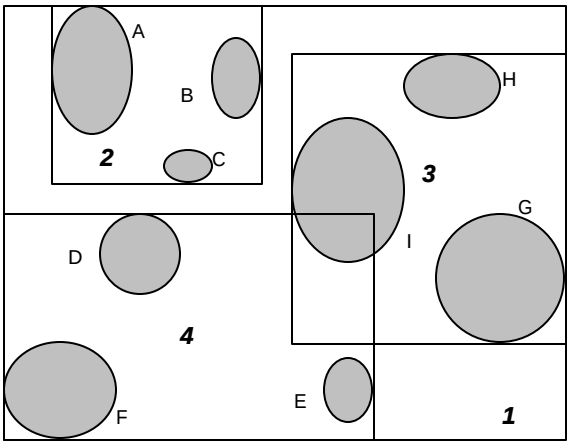

Figure 4.2 - Minimum Bounding Rectangles for the R-Tree

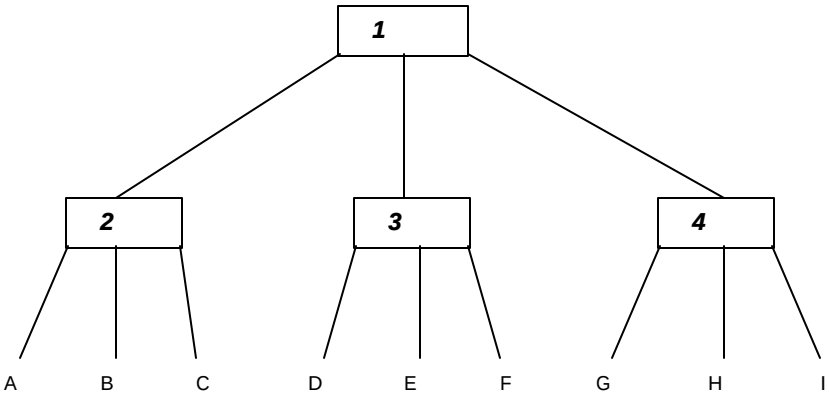

Figure 4.3 - R-Tree Corresponding to Figure 4.2

Buddy Tree:

Like the R^+-tree, the buddy tree, presented in [SK90], prevents its MBRs from overlapping. Unlike the R^+-tree, however, its spatial data does not have to be split among multiple paths. So, searching can be performed using only one path in the tree. This tree is able to accomplish this by restricting the minimum bounding regions to regions called buddy rectangles.

Buddy rectangles are formed as a result of repeatedly dividing the data space in half. Splitting a full node, and subsequently merging an underfilled node can only be done with these rectangles. The use of these partitions prevents splits from propagating downward as they do in R^+-trees. [SK90].

P-Tree:

The Polyhedral or P-trees of [Jag90] use a different approach for their MBRs. Instead of using a rectangular bounding region, the P-tree uses a more general shape. Specifically, the sides of the bounding region do not have to be perpendicular to the direction of any of the dimensions of the data

space. In many cases, this will reduce the volume of the MBRs [Jag90]. Since the volume is reduced, the overlap is reduced, and the probability of searching multiple branches of the tree is decreased.

The entries in a P-tree node differ from ones in an R-tree in that their MBRs must be specified differently. The R-tree bounding regions can be identified by the upper and lower values of each dimension of the data space, since they are only oriented in those directions. In contrast, the polyhedral regions may be composed of hyperplanes running in several different orientations. These directions define an orientation space that may have more dimensions than the data space. The entries in a P-tree, then, must define the upper and lower bounding values for each direction of the orientation space to describe the MBR. [Jag90].

Figure 4.4 illustrates a set of MBRs for a P-tree. As in Figure 4.2, the data regions are labeled with letters, and the MBRs are labeled with numbers. The corresponding P-tree is the same as the one displayed in Figure 4.3.

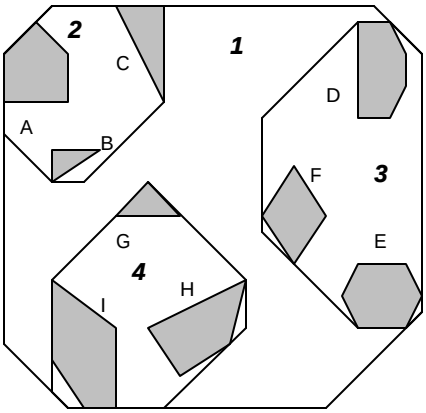

Figure 4.4 - Polyhedral Minimum Bounding Rectangles for the P-Tree

X-Tree:

The X-tree presented in [BKK96] is also focused on minimizing the overlap occurring between MBRs in order to reduce the amount of paths in the tree that must be searched. The tree does not, however, split a data region among multiple MBRs as in the R^+-tree. Instead, this tree analyzes the

amount of overlap that will occur for each split, then choosing the minimum overlap or creating a supernode to avoid it altogether. This involves a more complex algorithm used for splitting the original R-tree.

The term X-tree is short for eXtended node tree. As indicated earlier, this data structure is so named because when the splitting algorithm cannot a split without overlap, it creates a new type of node called a supernode. This term refers to an internal node of the X-tree whose capacity is larger than the normal nodes. This means that they will contain more data values and pointers than the maximum limit of the other nodes. This also helps reduce the number of levels in the tree, which improves the search time to locate all of the nodes. [BKK96].

Figure 4.5 [BKK96] illustrates this concept. The darkened circles in the nodes represent pointers to its children. The shaded node, containing more pointers than the other nodes, is a supernode. Its creation allows the tree to have only two levels of internal nodes.

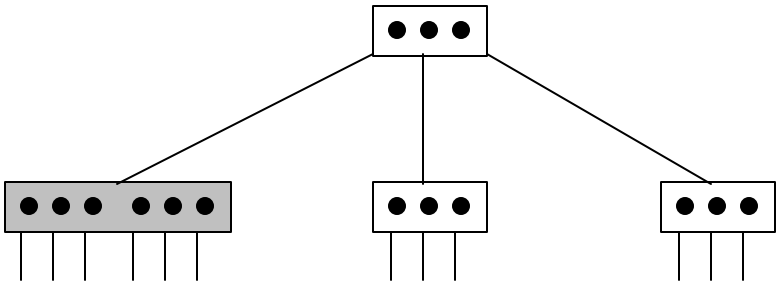

Figure 4.5 - An X-Tree with a Supernode [BKK96]

SR-Tree, SS-Tree:

The SR-tree was presented in [KS97]. It stands for the Sphere-Rectangle Tree, and is a modification of another data structure presented in [WJ96] called the SS or Sphere-Sphere Tree. In the SS-tree, the nodes correspond to the minimum bounding sphere of the data objects. The main advantage of using spheres instead of rectangles is that the former can be represented by only its center

and radius. This requires less space than rectangles that require storing the upper and lower bounding values of the data structure for each dimension. [KS97, WJ96].

A drawback of the SS-tree is that the minimum bounding spheres have more volume than the minimum bounding rectangles. This increases the amount of overlap. This problem is addressed by the Sphere-Rectangle Tree, where each node corresponds to the region of data space identified by the intersection of the minimum bounding sphere and the minimum bounding rectangle of its data elements. While this reduces the amount of overlap, the use of the intersection of the two geometrical shapes means that a more complex representation is needed for each node. Specifically, this region is identified by the center of its minimum bounding sphere and the minimum of the longest distance to the minimum bounding spheres and rectangles of its children. These values are the center and radius of each node, respectively. This complex representation also means that more computations are needed to recompute these regions, which increases the time taken to perform insertions and deletions. [KS97].

Figures 4.6 and 4.7 illustrate a set of MBRs for an SS-tree and SR-tree, respectively, with the MBRs for the SR-tree shaded in Figure 4.7. Once again, the data regions are labeled with letters, and the MBRs are labeled with numbers. The corresponding trees are the same as the one displayed in Figure 4.3.

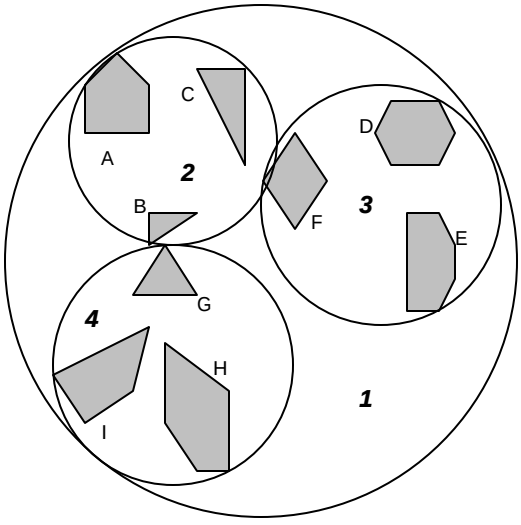

Figure 4.6 - Spherical Minimum Bounding Rectangles for the SS-Tree

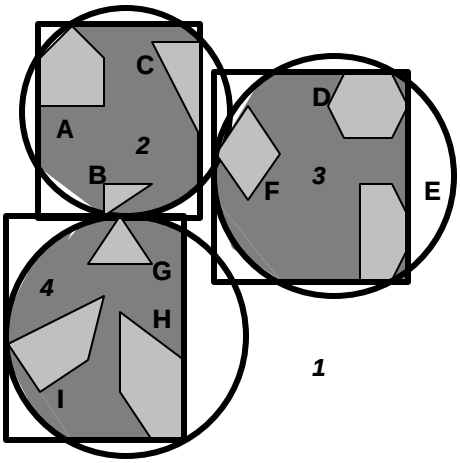

Figure 4.7 - Intersection (Shaded) of Spheres and Rectangles for the SR-Tree

TV-Tree:

One of the problems with many of the trees in this section is that they are inefficient when dealing with data that has a large number of dimensions. Because of this, the Telescopic Vector Tree (TV-tree) was developed and presented in [LJF94]. Its purpose is to reduce the number of dimensions used by the index.

This reduction is accomplished by using only the dimensions that are necessary for distinguishing the data points. Such dimensions are called active. This is opposed to inactive dimensions that describe the dimensions at the beginning of the feature vectors that have all of the same values [LJF94]. Determining which dimensions are needed to distinguish the data points implies that some features are more important than others [KS97]. So, although this results in reducing the number of dimensions, it does not treat each dimension of the data space equally.

As the data contained in the tree's MBRs changes, the dimensions that are active will change. The number of active dimensions, however, remains constant. So, a TV-α tree is one that contains α active dimensions. For example, let an MBR of a TV-2 tree contain the points $\langle 3, 0.5, 5, 3\rangle$, $\langle 3, 0.5,$

7, 0⟩, and ⟨3, 0.5, 10, 8⟩. Since the values in the first two dimensions of the points are the same, dimensions 1 and 2 are inactive. The active dimensions, then, are dimensions 3 and 4. This means that in addition to the points specifying the bounding region, each node must identify the number of dimensions it stores, with the last α considered active.

5. Tree Index Classifications

The indexing trees surveyed in this paper use different techniques to partition the data points of a multidimensional space. As a result of these differences, some trees may be better suited than others for specific applications. If the expected types of queries and distribution of the data are known, then the properties of these techniques should be considered when determining the index that is the most appropriate.

Different classifications of the indexing trees may be developed based on the properties of their partitioning techniques. In this section, we categorize the trees surveyed in this paper using such classifications. We will also describe the advantages and disadvantages of each classification which should help developers in determining the trees that are the most appropriate for their applications. We begin by listing the various trees in the far left column of Table 1. In the next column, we provide the reference of the paper that introduced the tree. The remaining columns list the properties used to classify the trees.

As indicated by the sections in this paper, one method of classifying indexing techniques is based on whether or not they use Minimum Bounding Regions (MBRs). This classification is important because it indicates the computations that must be performed and the splitting information that must be stored for each data partition. If an indexing tree uses MBRs, then a minimum cover must be computed for the data elements stored at each of its internal nodes. In addition, an MBR must be stored at each internal node which means storing the upper and lower bounds along each dimension if it is rectangular. Alternatively, an advantage is that MBRs cover only the data elements and not the entire

multidimensional space. This reduces the amount of time spent searching by quickly eliminating large amounts of the data space.

In addition to this advantage, indexes that use MBRs are designed to index multidimensional data shapes, while the ones that do not are designed for only points. While it is true that the indexes without MBRs could maintain multidimensional shapes by transforming them into a set of data points indicating each of its upper and lower values along every dimension, this transformation is undesirable. The reason is that it increases the number of dimensions used to represent each shape. So, whether or not a tree uses MBRs is our first classification, and is listed in Table 1 under the column marked "MBR".

In [SRF87], researchers identify three different properties of an indexing method. The first property, called *position*, is based on whether the division of the data space is predetermined or varies based on the distribution of the data elements. Techniques using the first procedure are called *fixed*, and techniques using the second are called *adaptable*. Fixed-position trees have the advantage that since the split is predetermined, the values and dimensions used do not have to be stored explicitly. Another advantage is that irrespective of the order that data elements were added into or deleted from the tree, the resulting partitions will be similar. Thus, it is not required to know the distribution of the data beforehand to optimize the performance of the tree [SK90]. An advantage of adaptable-position trees, however, is that the number of data elements may be evenly split independently of their positions in the data space. As stated earlier, this reduces the amount of time used to search the tree for a data element.

The second property, *dimensionality*, is based on the number of dimensions split by each division. Either one dimension is used for dividing data regions, or all of the dimensions are used. Using multidimensional divisions improves the ability to create partitions that contain an even number of data points from any data distribution. Again, this reduces the searching time of the tree. The disadvantage of multidimensional partitions is that for every dimension, the values used for splitting must be stored at each node. This increases the amount of storage space used by the index.

The final property, *locality*, describes the number of regions that are split on each division. Grid methods divide all regions along the same dimension with the same splits, while brickwall methods

divide each region individually. Dividing each region individually requires a tree-based index to store each partition. Consequently, all of the tree-based indexes surveyed are brickwall methods.

These properties are included in Table 1, except for locality because of the redundancy. Position is indicated as either fixed or adaptable using the definitions described earlier. The dimensionality of the partitioning is identified in the column marked "Dim." as multi for multidimensional divisions and single for single-dimensional ones.

Three additional properties are defined in [SK90]. The first is based on whether the regions *overlap*. Ideally, the data partitions should not overlap. This ensures that a query point will not be contained in more than one node at each level of the index tree. Thus, only one path of the tree must be accessed, which reduces the time used for searching. Still, it is difficult to split some sets of multidimensional shapes without using overlapping partitions. Thus, a disadvantage of the trees that do not use overlapping regions is that they require a more complex splitting algorithm when indexing shapes.

The second property is based on whether the partitions are *rectangular*. Rectangular partitions are simple to implement and are stored easily using upper and lower bounds along each dimension. Regions based on other shapes, however, may provide smaller covers for the data which eliminates more empty data space from the tree. In addition, since only a center and radius are needed, spherical data shapes use less storage space at each node in the tree.

The final property, *completeness,* is based on whether or not the entire data space is partitioned. The last property refers to those indexing methods that do not store empty data regions such as the area labeled ***01*** in Figure 3.3. Trees that avoid partitioning such regions may eliminate large amounts of the data space. This reduces the size of the tree. Nevertheless, there is a tradeoff in that trees that completely partition the data space use simpler splitting algorithms.

As with the earlier set, these properties are included in Table 1. The trees that allow overlapping data partitions are indicated under the column heading "Over.". Similarly, the trees use rectangular data partitions are indicated in the column marked "Rect.", and trees that completely partition the entire data space are indicated in the column marked "Comp.".

We classify tree-based indexes according to other properties as well. To retrieve a multidimensional data point from the tree-based indexing techniques, its region must first be located. This is also true for inserting and deleting data elements. Locating a region corresponds to searching the tree. If a data point is contained in only one region, then only one path of the tree must be searched. Otherwise, several subtrees must be searched. The ability to *search one path* is the next classification included in Table 1. This classification is related closely to the "overlap" property in that only one path must be searched in indexes whose data space partitions are disjoint. However, there are indexes, such as the BV-tree, where the partitions do overlap, yet only one path is traversed from the root to a leaf during a search.

Whether or not a split can *propagate* to its children is our next classification. As discussed in section 3, a split of a full node that propagates downward may unevenly divide the data elements of its children. This means that some nodes, and therefore pages, may have an extremely low utilization, so it would be impossible to determine a minimum utilization for the entire tree [Fre95]. So, in the column in Table 1 marked "Split Propagate", we indicate whether or not a split of a full node can propagate into its children. Splits, however, are operations that occur after insertions. Some of the tree-based indexes are not considered to be dynamic by their developers, so the entire tree should be reconstructed after an insertion or deletion. Therefore, splits should not occur in these trees. We marked those such trees in this survey with the word "static" in this column.

The next property is whether or not the tree is *balanced*, meaning that all of the leaves are at the same level [KS91]. This property is listed under the column heading "Bal." in Table 1. A tree that is not balanced may have extremely long paths to a data element. This increases the amount of time needed to search and access it. Keeping a tree balanced, however, requires using split and merge operations to perform insertions and deletions. This increases the time it takes to perform those operations.

Finally, we categorize the data based on whether or not the tree requires an *ordering* on the attributes of the feature vector to perform the partitioning, listed under the column heading "Order".

Generally, if an ordering is needed on the dimensions, then they are not treated equally. The dimension listed first will have priority in determining the data partitions.

The indexing methods listed in Sections 3 and 4 are categorized all of these classifications in Table 1. They are sorted in alphabetical order using the names of the trees.

Tree	Ref.	MBR	Position	Dim.	Overlap	Rect.	Comp.	Search One Path	Split Propagate	Bal.	Order
Buddy	SK90	Yes	Fixed	Multi	No	Yes	No	Yes	No	No	No
BV	Fre95	No	Adaptable	Multi	Yes	No	Yes	Yes	No	No	No
G	Kum94	No	Fixed	Single	No	Yes	No	Yes	No	Yes	Yes
HB	LS90	No	Adaptable	Multi	No	No	Yes	Yes	No	Yes	No
K-D-B	Rob81	No	Adaptable	Single	No	Yes	Yes	Yes	Yes	Yes	Yes
LSD	HSW89	No	Adaptable	Single	No	Yes	Yes	Yes	No	No	No
MB+	YVD95	No	Adaptable	Single	No	Yes	Yes	Yes	No	Yes	Yes
MVP	BO97	No	Adaptable	Multi	No	No	Yes	Yes	Static	Yes	No
P	Jag90	Yes	Adaptable	Multi	Yes	No	No	No	No	Yes	No
R	Gut84	Yes	Adaptable	Multi	Yes	Yes	No	No	No	Yes	No
R*	BKSS90	Yes	Adaptable	Multi	Yes	Yes	No	No	No	Yes	No
R+	SRF87	Yes	Adaptable	Multi	No	Yes	No	Yes	Yes	Yes	No
SR	KS97	Yes	Adaptable	Multi	Yes	No	No	No	No	Yes	No
SS	WJ96	Yes	Adaptable	Multi	Yes	No	No	No	No	Yes	No
TV	LJF94	Yes	Adaptable	Multi	Yes	No	No	No	No	Yes	Yes
VP	Yia93	No	Adaptable	Multi	No	No	Yes	Yes	Static	Yes	No
X	BKK96	Yes	Adaptable	Multi	Yes	Yes	No	No	No	Yes	No

Table 1 - Tree-Based Indexing Classifications

6. Summary and Future Work

This paper provided a survey of the tree-based techniques used to index multidimensional data. These techniques treat the data as points on a multidimensional grid partitioned into several regions. Points that are contained in the same region are stored together in the tree, where each node points to a region that is further subdivided by its children.

In addition, this paper presented properties for categorizing the indexing methods. Not only do these categories provide a way to classify different indexing structures, but they can be used to determine the most appropriate tree to use given the requirements of an application and its expected data distribution.

For future work in classifying indexing trees, it would be useful to perform a more detailed study on their performances using the same sets of data. Each of the references compares the performance of its tree against some of the other trees in this survey, but none uses all of them. These comparisons should be performed using all of the types of queries listed in this paper as well as insertions and deletions. They should also be tested using both even and uneven data distribution patterns. In addition, it should analyze the overall node utilization and amount of space required by each index. These evaluations should lead to determining the general types of applications best suited for each index. The results of the evaluations can be combined with Table 1 in Section 5 to select the best index for an actual or hypothetical application.

Another area for future work is the development of new indexes by modifying the trees listed in this paper based on the classifications presented in Section 5. It may be desirable, for example, for some application to have an index that does not completely partition the data, uses only one path for searching, prevents a split of a full node from propagating downward into its children, is balanced, and does not require an ordering on the dimensions of the feature vectors of the data elements. Since none of the trees has all of these properties together, it would be useful to perform research on the ability to modify one of the trees such as the R+ or BV-tree. Thus, the information in Table 1 illustrates several open areas of research for modifying or creating new indexing trees.

It is worth noting that the trees presented in this paper are useful for indexing multimedia objects represented as multidimensional points or regions. Not all multimedia database management systems represent their data using such feature vectors, however. For example, in a view-based system some of the data elements are represented as a set of operations modifying other stored objects [GS96]. For this and other types of multimedia databases, alternate indexing strategies may have to be considered.

References:

[BKK96] Berchtold, Stefan, Daniel A. Keim, and Hans-Peter Kriegel, "*The X-Tree: An Index Structure for High-Dimensional Data*", Proceedings of the 22nd International Conference on Very Large Databases, 1996, pp. 28 - 39.

[BKSS90] Beckmann, Norbert, et. al., "*The R*-tree: An Efficient and Robust Access Method for Points and Rectangles*", Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, May 1990, pp. 322-331.

[BO97] Bozkaya, Tolga and Meral Ozsoyoglu, "*Distance-Based Indexing for High-Dimensional Metric Spaces*", Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data, May 1997, pp. 357-368.

[Chi94] Chiueh, Tzi-cker, "*Content-Based Image Indexing*", Proceedings of the 22nd International Conference on Very Large Databases, 1994, pp. 582 - 593.

[Com79] Comer, Douglas, "*The Ubiquitous B-Tree*", Computing Surveys, Vol. 11, No. 2, June 1979, pp. 121-137.

[Fal96] Faloutsus, Christos, Searching Multimedia Databases by Content, Kluwer Academic Publishers, Boston, 1996.

[Fre95] Freeston, Michael, "*A General Solution of the n-dimensional B-tree Problem*", Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, 1995, pp. 80-91.

[GS96] Gruenwald, Le and Greg Speegle, "*Research Issues in View-Based Multimedia Database Systems*", Proceedings of the 2nd World Conference on Integrated Design & Process Technology, December, 1996.

[Gut84] Guttman, Antonin, "*R-trees: A Dynamic Index Structure for Spatial Searching*", Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, 1984, pp. 47-57.

[HSW89] Henrich, Andreas, Hans-Werner Six, and Peter Widmayer, "*The LSD Tree: Spatial Access to Multidimensional Point and Non Point Objects*", Proceedings of the 15th International Conference on Very Large Databases, 1989, pp. 45 - 53.

[Jag90] Jagadish, H. V., "*Spatial Search with Polyhedra*", Proceedings of the 6th International Conference on Data Engineering, 1990, pp. 311 - 319.

[Jag96] Jagadish, H. V., "*Indexing for Retrieval by Similarity*", Multimedia Database Systems: Issues and Research Directions, Subrahmanian, V. S. and S. Jajodia (Eds.), Spriner-Verlag, NewYork, 1996, pp. 165-184.

[KS91] Korth, Henry F. and Abraham Silberschatz, Database System Concepts, McGraw-Hill, Inc., New York, 1991.

[KS97] Katayama, Norio, and Shin'ichi Satoh, "*The SR-Tree: An Index Structure for High-Dimensional Nearest Neighbor Queries*", Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data, May 1997, pp. 369-380.

[Kum94] Kumar, Akhil, "*G-Tree: A New Data Structure for Organizing Multidimensional Data*", IEEE Transactions on Knowledge and Data Engineering, Vol. 6, No. 2, April 1996, pp. 341 - 347.

[LJF94] Lin, King-Ip, H. V. Jagadish, and Christos Faloutsos, "*The TV-Tree: An Index Structure for High-Dimensional Data*", VLDB Journal, Vol. 3, 1994, pp 517-542.

[LS90] Lomet, David B. and Betty Salzberg, "*The hB-Tree: A Multiattribute Indexing Method with Good Guaranteed Performance*", ACM Transactions on Database Systems, Vol. 15, No. 4, December, 1990, pp. 625-658.

[NHS84] Nievergelt, J., H Hinterberger, and K. C. Sevcik, "*The Grid File: An Adaptable, Symmetric, Multikey File Structure*", ACM Transactions on Database Systems, Vol. 9, No. 1, March 1984, pp. 38-71.

[Rob81] Robinson, John T., *"The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes*", Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data, April 1981, pp. 10-18.

[SK90] Seeger, Bernhard, and Hans-Peter Kriegel, *"The Buddy-Tree: An Efficient and Robust Method for Spatial Data Base Systems*", Proceedings of the 16th International Conference on Very Large Databases, 1990, pp. 590-601.

[SRF87] Sellis, Timos, Nick Roussopoulos, and Christos Faloutsos, *"The R+-Tree: A Dynamic Index for Multidimensional Objects*", Proceedings of the 13th International Conference on Very Large Databases, 1987, pp. 507-518.

[WJ96] White, David A. and Ramesh Jain, "*Similarity Indexing with the SS-tree*", Proceedings of the 12th International Conference on Data Engineering, 1996, pp. 516-523.

[Yia93] Yianilos, Peter, N. "*Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces*", Proceedings of the 3rd Anuual ACM-SIAM Symposium on Discrete Algorithms, 1993, pp. 311-321.

[YVD95] Yang, Qi, Asha Vellaikal, and Son Dao, "*MB+-Tree: A New Index Structure for Multimedia Databases*", Proceedings of the International Workshop on Multimedia Database Management Systems, August, 1995, pp. 151-158.