1

Multidimensional Point Data

The representation of multidimensional point data is a central issue in database design, as well as applications in many other fields, including computer graphics, computer vision, computational geometry, image processing, geographic information systems (GIS), pattern recognition, very large scale integration (VLSI) design, and others. These points can represent locations and objects in space, as well as more general records. As an example of a general record, consider an employee record that has attributes corresponding to the employee's name, address, sex, age, height, weight, and Social Security number. Such records arise in database management systems and can be treated as points in, for this example, a seven-dimensional space (i.e., there is one dimension for each attribute or key[1]), albeit the different dimensions have different type units (i.e., name and address are strings of characters; sex is binary; and age, height, weight, and Social Security number are numbers).

Formally speaking, a database is a collection of records, termed a *file*. There is one record per data point, and each record contains several attributes or keys. In order to facilitate retrieval of a record based on some of the values of attributes,[2] we assume the existence of an ordering for the range of values of each of these attributes. In the case of locational or numeric attributes, such an ordering is quite obvious because the values of these attributes are numbers. In the case of alphanumeric attributes, the ordering is usually based on the alphabetic sequence of the characters making up the attribute value. Other data such as color could be ordered by the characters making up the name of the color or, possibly, the color's wavelength. It should be clear that finding an ordering for the range of values of an attribute is generally not an issue; the only issue is what ordering to use!

The representation that is ultimately chosen for this collection depends, in part, on answers to the following questions:

1. What is the type of the data (e.g., is it discrete, is it continuous, is its domain finite)?
2. What operations are to be performed on the data?
3. Should we organize the data or the embedding space from which the data is drawn?

[1] We use the terms *key* and *attribute* (as well as *field*, *dimension*, *coordinate*, and *axis*) interchangeably in this chapter. We choose among them based on the context of the discussion, while attempting to be consistent with their use in the literature.

[2] This is also known as *secondary key retrieval* (e.g., [1046]) to distinguish it from *primary key retrieval*. In secondary key retrieval, the search is based on the value of more than just one attribute, as is the case in traditional searching applications where we are dealing with one-dimensional data.

4. Is the database static or dynamic (i.e., can the number of data points grow and shrink at will)?

5. Can we assume that the volume of data is sufficiently small so that it can all fit in main memory, or should we make provisions for accessing disk-resident data?

The type of the data is an important question. Of course, we must distinguish between data consisting of one and several attributes. Classical attribute values include bits, integers, real numbers, complex numbers, characters, strings, and so on. There is also the issue of whether the domain from which the data is drawn is finite (i.e., bounded) or not. This distinction is used, in part, to determine which of the data organizations in question 3 can be used. An alternative differentiation is on the basis of whether the data is discrete or continuous. In the case of data that is used in conventional scientific computations, discrete data corresponds to integers, while continuous data corresponds to real and complex numbers. In the case of spatial data, we have discrete data (by which we mean distinct samples or instances) such as points (which is the subject of this chapter and, to a lesser extent, Chapter 4), while continuous data consists of objects that take up space such as line segments, areas, surfaces, volumes, and so on (which are the subject of Chapter 2 and, to a lesser extent, Chapter 3). Similarly, temporal data can also be differentiated on the basis of whether it is discrete or continuous. In particular, concepts such as event time and transaction time are indicative of a discrete approach, while concepts such as rates are more indicative of a continuous approach in that they recognize that the data is constantly changing over time. In this book, we do not deal with temporal data (e.g., [951]) or the closely related concept of spatiotemporal data (e.g., [21, 950, 1305]).

The distinction in question 3 is formulated in terms of the notion of an *embedding space*. The embedding space is determined by the span of values of the data. For example, suppose that we have a set of employee records with a state-valued attribute called STATE whose embedding space is the set of names of the 50 U.S. states. In this case, our distinction is between organizing the employee records on the basis of the value of the STATE attribute (e.g., all employee records whose STATE attribute value starts with letters A–M in one group and all employee records whose STATE attribute value starts with letters N–Z in another group) or organizing the employee records on the basis of the value of this attribute for the record of a particular employee (e.g., if employee records are grouped into two groups based on whether the first letter of the value of their STATE attribute is less than or equal to the first letter of the value of the STATE attribute for the record corresponding to an employee named "John Smith").

As another example, in the case of a computer architecture where the representation of each data item is implemented by one word of 32 bits, then, for simple integer data, the embedding space spans the values 0 to $2^{32} - 1$, while it is considerably larger for real data, where it depends on the way in which the representation of simple real data is implemented (i.e., in the case of floating-point representation, the number of bits allocated to the mantissa and to the exponent). In both cases of the example above, the embedding space is said to be finite.

The distinction made in question 3 is often used to differentiate between one-dimensional searching methods (e.g., [1046]) that are tree-based from those that are trie-based. In particular, a *trie* [251, 635] is a branching structure in which the value of each data item or key is treated as a sequence of j characters, where each character has M possible values. A node at depth $k - 1$ ($k \geq 1$) in the trie represents an M-way branch depending on the value of the kth character (i.e., it has M possible values). Some (and at times all) of the data is stored in the leaf nodes, and the shape of the trie is independent of the order in which the data is processed.

For example, address computation methods such as radix searching [1046] (also known as *digital searching* and yielding structures known as *digital trees*) are instances of trie-based methods, since the boundaries of the partition regions of similarly valued

data that result from the partitioning (i.e., branching) process are drawn from among locations that are fixed regardless of the content of the file. Therefore, these trie-based methods are said to organize the data on the basis of the embedding space from which they are drawn. Because the positions of the boundaries of the regions are fixed, the range of the embedding space from which the data is drawn must be known (said to be *bounded*). On the other hand, in one dimension, the binary search tree [1046] is an example of a tree-based method since the boundaries of different regions in the search space are determined by the data being stored. Such methods are said to *organize the data* rather than the embedding space from which the data is drawn. Note that although trie-based methods are primarily designed to organize integer-valued data since the partition values (i.e., the boundaries of the partition regions) are also integer valued, nevertheless they can also be used to organize real-valued data, in which case the real-valued data are compared to predetermined partition values (e.g., 0.5, 0.25, 0.75, in the case of real numbers between 0 and 1), which, again, are the boundaries of the partition regions. Of course, the data must still be bounded (i.e., its values must be drawn from within a given range).

The extension of a trie to multidimensional data is relatively straightforward and proceeds as follows. Assume d-dimensional data where each data item consists of d character sequences corresponding to the d attributes. Each character of the ith sequence has M_i possible values. A node at depth $k-1$ ($k \geq 1$) in the multidimensional trie represents an $\prod_{i=1}^{d} M_i$-way branch, depending on the value of the kth character of each of the d attribute values.[3] For attributes whose value is of numeric type, which is treated as a sequence of binary digits, this process is usually a halving process in one dimension, a quartering process in two dimensions, and so on, and is known as *regular decomposition*. We use this characterization frequently in our discussion of multidimensional data when all of the attributes are locational.

Disk-resident data implies grouping the data (either the underlying space based on the volume—that is, the amount—of the data it contains or the points, hopefully, by the proximity of their values) into sets (termed *buckets*), corresponding to physical storage units (i.e., pages). This leads to questions about their size and how they are to be accessed, including the following:

1. Do we require a constant time to retrieve a record from a file, or is a logarithmic function of the number of records in the file adequate? This is equivalent to asking if the access is via a directory in the form of an array (i.e., direct access) or in the form of a tree (i.e., a branching structure).
2. How large can the directories be allowed to grow before it is better to rebuild them?
3. How should the buckets be laid out on the disk?

In this chapter, we examine several representations that address these questions. Our primary focus is on dynamic data, and we concentrate on the following queries:

1. Point queries that determine if a given point p is in the dataset. If yes, then the result is the address corresponding to the record in which p is stored. This query is more accurately characterized as an *exact match query* in order to distinguish it from the related point query in the context of objects that finds all the objects that contain a given point (see, e.g., Section 3.3.1 of Chapter 3).
2. Range queries (e.g., region search) that yield a set of data points whose specified keys have specific values or values within given ranges. These queries include the partially specified query, also known as the *partial match query* and the *partial range query*, in which case unspecified keys take on the range of the key as their domain.
3. Boolean combinations of 1 and 2 using the Boolean operations **and**, **or**, and **not**.

[3] When the ith character sequence has less than k characters, there is no corresponding branching at depth $k-1$ for this attribute.

When multidimensional data corresponds to locational data, we have the additional property that all of the attributes have the same unit, which is distance in space.[4] In this case, we can combine the attributes and pose queries that involve proximity. Assuming a Euclidean distance metric, for example, we may wish to find all cities within 50 miles of Chicago. This query is a special case of the range query, which would seek all cities within 50 miles of the latitude position of Chicago and within 50 miles of the longitude position of Chicago.[5] A related query seeks to find the closest city to Chicago within the two-dimensional space from which the locations of the cities are drawn—that is, a nearest neighbor query, also sometimes referred to as the *post office problem* (e.g., [1046, p. 563]). This problem arises in many different fields, including computer graphics, where it is known as a *pick query* (e.g., [622]); in coding, where it is known as the *vector quantization problem* (e.g., [747]); and in pattern recognition, as well as machine learning, where it is known as the *fast nearest-neighbor classifier* (e.g., [514]). We do not deal with such queries in this chapter (but see [846, 848, 854] and Sections 4.1 and 4.2 of Chapter 4).

NAME	X	Y
Chicago	35	42
Mobile	52	10
Toronto	62	77
Buffalo	82	65
Denver	5	45
Omaha	27	35
Atlanta	85	15
Miami	90	5

Figure 1.1
Sample list of cities with their x and y coordinate values.

In contrast, proximity queries are not very meaningful when the attributes do not have the same type or units. For example, it is not customary to seek all people with age-weight combination less than 50 year-pounds (year-kilograms) of that of John Jones, or the person with age-weight combination closest to John Jones because we do not have a commonly accepted unit of year-pounds (year-kilograms) or a definition thereof.[6] It should be clear that we are not speaking of queries involving Boolean combinations of the different attributes (e.g., range queries), which are quite common.

The representations that we describe are applicable to data with an arbitrary number of attributes d. The attributes need not be locational or numeric, although all of our examples and explanations assume that all of the attributes are locational or numeric. In order to be able to visualize the representations, we let $d = 2$. All of our examples make use of the simple database of records corresponding to eight cities given in Figure 1.1.[7] Each record has three attributes, NAME, X, and Y, corresponding to their name and location. We assume that our queries retrieve only on the basis of the values of the X and Y attributes. Thus, $d = 2$, and no ordering is assumed on the NAME field. In particular, the NAME field is just used to aid us in referring to the actual locations and is not really part of the record when we describe the various representations in this chapter. Some of these representations can also be adapted to handle records where the attributes are nonlocational, as long as an ordering exists for their range of values. Moreover, Section 4.5 of Chapter 4 contains a discussion of how some of these representations can be adapted to handle records where the only information available is the relative distance between pairs of records.

[4] The requirement that the attribute value be a unit of distance in space is stronger than one that merely requires the unit to be a number. For example, if one attribute is the length of a pair of pants and the other is the width of the waist, then, although the two attribute values are numbers, the two attributes are not locational.

[5] The difference between these two formulations of the query is that the former admits a circular search region, while the latter admits a rectangular search region. In particular, the latter query is applicable to both locational and nonlocational data, while the former is applicable only to locational data.

[6] This query is further complicated by the need to define the distance metric. We have assumed a Euclidean distance metric. However, other distance metrics such as the City Block (also known as Manhattan) and the Chessboard (also known as maximum value) could also be used.

[7] Note that the correspondence between coordinate values and city names is not geographically accurate. We took this liberty so that the same example could be used throughout the chapter to illustrate a variety of concepts.

the x coordinate. A range search for $[B:E]$ is performed by procedure RANGESEARCH. It searches the tree and finds the node with either the largest value $\leq B$ or the smallest value $\geq B$, and then follows the links until reaching a leaf node with a value greater than E. For N points, this process takes $O(\log_2 N + F)$ time and uses $O(N)$ space. F is the number of points found.

Procedure RANGESEARCH assumes that each node has six fields, LEFT, RIGHT, VALUE, PREV, NEXT, and MIDRANGE. LEFT(P) and RIGHT(P) denote the left and right children, respectively, of nonleaf node P (they are null in the case of a leaf node). VALUE is an integer indicating the value stored in the leaf node. PREV(P) and NEXT(P) are meaningful only for leaf nodes, in which case they are used for the doubly linked list of leaf nodes sorted in nondecreasing order. In particular, PREV(P) points to a node with value less than or equal to VALUE(P), while NEXT(P) points to a node with value greater than or equal to VALUE(P). MIDRANGE(P) is a variant of a discriminator between the left and right subtrees—that is, it is greater than or equal to the values stored in the left subtree, and less than or equal to the values stored in the right subtree (see Exercise 2). The MIDRANGE field is meaningful only for nonleaf nodes. Note that by making use of a NODETYPE field to distinguish between leaf and nonleaf nodes, we can use the LEFT, RIGHT, and MIDRANGE fields to indicate the information currently represented by the PREV, NEXT, and VALUE fields, respectively, thereby making them unnecessary.

```
1  procedure RANGESEARCH(B, E, T)
2  /* Perform a range search for the one-dimensional interval [B:E] in the one-
      dimensional range tree rooted at T. */
3  value integer B, E
4  value pointer node T
5  if ISNULL(T) then return
6  endif
7  while not ISLEAF(T) do
8     T ← if B ≤ MIDRANGE(T) then LEFT(T)
9          else RIGHT(T)
10         endif
11 enddo
12 if not ISNULL(T) and VALUE(T) < B then T ← NEXT(T)
13 endif
14 while not ISNULL(T) and VALUE(T) ≤ E do
15    output VALUE(T)
16    T ← NEXT(T)
17 enddo
```

For example, suppose we want to perform a range search for [28:62] on the one-dimensional range tree in Figure 1.7. In this example, we assume that the VALUE field of the leaf nodes contains only the x coordinate values. We first descend the tree to locate, in this case, the node with the largest value $\leq$28 (i.e., (27,35)). Next, following the NEXT links, we report the points (35,42), (52,10), and (62,77). We stop when encountering (82,65).

A two-dimensional range tree is a binary tree of binary trees. It is formed in the following manner. First, sort all of the points along one of the attributes, say x, and store them in the leaf nodes of a balanced binary search tree (i.e., a range tree), say T. With each node of T, say I, associate a one-dimensional range tree, say T_I, of the points in the subtree rooted at I, where now these points are sorted along the other attribute, say y.[12] For example, Figure 1.8 is the two-dimensional range tree for the data of Figure 1.1,

[12] Actually, there is no need for the one-dimensional range tree at the root or its two children (see Exercise 6). Also, there is no need for the one-element, one-dimensional range trees at the leaf nodes, as the algorithms can make a special check for this case and use the data that is already stored in the leaf nodes (but see Exercise 5).

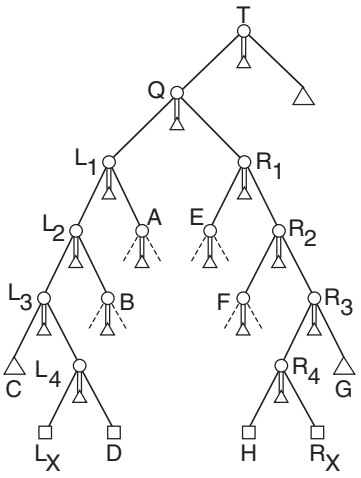

Figure 1.9
Example two-dimensional range tree to illustrate two-dimensional range searching.

For example, the desired closest common ancestor of L_X and R_X in Figure 1.9 is Q. One-dimensional range searches would be performed in the one-dimensional range trees rooted at nodes A, B, D, E, F, and H since 5 $\{L_i\} = \{L_1, L_2, L_3, L_4, L_X\}$ and $\{R_i\} = \{R_1, R_2, R_3, R_4, R_X\}$. For N points, procedure 2DSEARCH takes $O(\log_2^2 N + F)$ time, where F is the number of points found (see Exercise 9).

As a more concrete example, suppose we want to perform a range search for ([25:85],[8:16]) on the two-dimensional range tree in Figure 1.8. We first find the nearest common ancestor of 25 and 85, which is node A. The paths $\{L_i\}$ and $\{R_i\}$ are given by {B,D,I} and {C,G,N}, respectively. Since B and B's left child (i.e., D) are in the path to 25, we search the range tree of B's right child (i.e., E) and report (52,10) as in the range. Similarly, since C and C's right child (i.e., G) are in the path to 85, we search the one-dimensional range tree of C's left child (i.e., F), but do not report any results as neither (62,77) nor (82,65) is in the range. Finally, we check if the boundary nodes (27,35) and (85,15) are in the range, and report (85,15) as in the range.

The range tree also can be adapted easily to handle k-dimensional data. In such a case, for N points, a k-dimensional range search takes $O(\log_2^k N + F)$ time, where F is the number of points found. The k-dimensional range tree uses $O(N \cdot \log_2^{k-1} N)$ space (see Exercise 10) and requires $O(N \cdot \log_2^{k-1} N)$ time to build (see Exercise 11).

Exercises

1. Is there a difference between a balanced binary search tree, where all the data is stored in the leaf nodes, and a one-dimensional range tree?

2. The MIDRANGE field in the one-dimensional range tree was defined as a variant of a discriminator between the left and right subtrees in the sense that it is greater than or equal to the values stored in the left subtree, and less than or equal to the values stored in the right subtree. Why not use a simpler definition, such as one that stipulates that the MIDRANGE value is greater than all values in the left subtree and less than or equal to all values in the right subtree?

3. Why does procedure 2DSEARCH provide the desired result?

4. Prove that no point is reported more than once by the algorithm for executing a range query in a two-dimensional range tree.

5. Procedure 2DSEARCH makes use of a representation of the two-dimensional range tree where each node has a POINT and RANGETREE field. In fact, the POINT field is defined only for leaf nodes, while the RANGETREE field points to a one-dimensional range tree of just one node for leaf nodes. Rewrite 2DSEARCH, 1DSEARCH, and INRANGE so that they do not use a POINT field, and hence interpret the RANGETREE field appropriately.

6. Show that the one-dimensional range trees at the first two levels (i.e., at the root and the two children of the root) of a two-dimensional range tree are never used in procedure 2DSEARCH.

7. Show that $O(N \cdot \log_2 N)$ space suffices for a two-dimensional range tree for N points.

8. Show that a two-dimensional range tree can be built in $O(N \cdot \log_2 N)$ time for N points.

9. Given a two-dimensional range tree containing N points, prove that a two-dimensional range query takes $O(\log_2^2 N + F)$ time, where F is the number of points found.

10. Given a k-dimensional range tree containing N points, prove that a k-dimensional range query takes $O(\log_2^k N + F)$ time, where F is the number of points found. Also, show that $O(N \cdot \log_2^{k-1} N)$ space is sufficient.

11. Show that a k-dimensional range tree can be built in $O(N \cdot \log_2^{k-1} N)$ time for N points.

12. Write a procedure to construct a two-dimensional range tree.

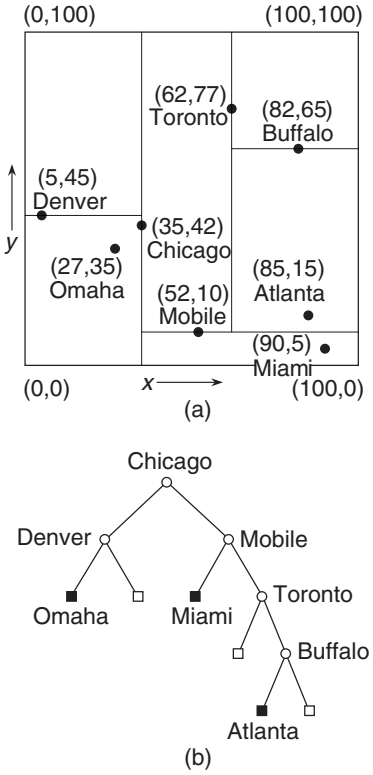

Figure 1.34
A k-d tree ($d = 2$) and the records it represents corresponding to the data of Figure 1.1: (a) the resulting partition of space and (b) the tree representation.

1.5.1 Point K-d Trees

There are many variations of the point k-d tree. Their exact structure depends on the manner in which they deal with the following issues:

1. Is the underlying space partitioned at a position that overlaps a data point, or may the position of the partition be chosen at random? Recall that trie-based methods restrict the positions of the partitions points, thereby rendering moot the question of whether the partition actually takes place at a data point.

2. Is there a choice as to the identity of the partition axis (i.e., the attribute or key being tested)? If we adopt a strict analogy to the quadtree, then we have little flexibility in the choice of the partition axis in the sense that we must cycle through the d different dimensions every d levels in the tree, although the relative order in which the different axes are partitioned may differ from level to level and among subtrees.

The most common variant of the k-d tree (and the one we focus on in this section) partitions the underlying space at the data points and cycles through the different axes in a predefined and constant order. When we can apply the partitions to the underlying space in an arbitrary order rather than having to cycle through the axes, then we preface the name of the data structure with the qualifier *generalized*. For example, a generalized k-d tree also partitions the underlying space at the data points; however, it need not cycle through the axes. In fact, it need not even partition all of the axes (e.g., only partition along the axes that are used in queries [172]). In our discussion, we assume two-dimensional data, and we test x coordinate values at the root and at even depths (given that the root is at depth 0) and y coordinate values at odd depths.

We adopt the convention that when node P is an x-discriminator, then all nodes having an x coordinate value less than that of P are in the left child of P and all those with x coordinate values greater than or equal to that of P are in the right child of P. A similar convention holds for a node that is a y-discriminator. Figure 1.34 illustrates the k-d tree corresponding to the same eight nodes as in Figure 1.1.

In the definition of a discriminator, the problem of equality of particular key values is resolved by stipulating that records that have the same value for a particular key are in the right subtree. As an alternative, Bentley [164] defines a node in terms of a *superkey*. Given a node P, let $K_0(P)$, $K_1(P)$, and so on, refer to its d keys. Assuming that P is a j-discriminator, then for any node Q in the left child of P, $K_j(Q) < K_j(P)$; and likewise, for any node R in the right child of P, $K_j(R) > K_j(P)$. In the case of equality, a superkey, $S_j(P)$, is defined by forming a cyclical concatenation of all keys starting with $K_j(P)$. In other words, $S_j(P) = K_j(P)\ K_{j+1}(P) \ldots K_{d-1}(P)\ K_0(P) \ldots K_{j-1}(P)$. Now, when comparing two nodes, P and Q, we turn to the left when $S_j(Q) < S_j(P)$ and to the right when $S_j(Q) > S_j(P)$. If $S_j(Q) = S_j(P)$, then all d keys are equal, and a special value is returned to so indicate. The algorithms that we present below do not make use of a superkey.

The rest of this section is organized as follows. Section 1.5.1.1 shows how to insert a point into a k-d tree. Section 1.5.1.2 discusses how to delete a point from a k-d tree. Section 1.5.1.3 explains how to do region searching in a k-d tree. Section 1.5.1.4 discusses some variants of the k-d tree that provide more flexibility as to the positioning and choice of the partitioning axes.

1.5.1.1 Insertion

Inserting record r with key values (a, b) into a k-d tree is very simple. The process is essentially the same as that for a binary search tree. First, if the tree is empty, then allocate a new node containing r, and return a tree with r as its only node. Otherwise, search the tree for a node h with a record having key values (a, b). If h exists, then r replaces the record associated with h. Otherwise, we are at a NIL node that is a child of type s (i.e., 'LEFT' or 'RIGHT', as appropriate) of node c. This is where we make the insertion by