Chapter 5

Multidimensional Indexes

All the index structures discussed so far are *one dimensional]* that is, they assume a single search key, and they retrieve records that match a given search-key value. We have imagined that the search key was a single attribute or field. However, an index whose search key is a combination of fields can still be one-dimensional. If we want a one-dimensional index whose search key is the fields $(F_1, F_2, \ldots, F_k)$, then we can take the search-key value to be the concatenation of values, the first from F_1, the second from F_2, and so on. We can separate these values by special marker symbols to make the association between search-key values and lists of values for the fields $F_1, \ldots, F_k$ unambiguous.

Example 5.1: If fields F_1 and F_2 are a string and an integer, respectively, and # is a character that cannot appear in strings, then the combination of values F_1 = `'abcd'` and F_2 = 123 can be represented by the string `'abcd#123'`. □

In Chapter 4, we took advantage of a one-dimensional key space in several ways:

- Indexes on sequential files and B-trees both take advantage of having all keys in a single, sorted order.
- Hash tables require that the search key be completely known for any lookup. If a key consists of several fields, and even one is unknown, we cannot apply the hash function, but must instead search all the buckets.

There are a number of applications that require us to view data as existing in a 2-dimensional space, or sometimes in higher dimensions. Some of these applications can be supported by conventional DBMS's, but there are also some specialized systems designed for multidimensional applications. One important way in which these specialized systems distinguish themselves is by using data structures that support certain kinds of queries that are not common in SQL applications. Section 5.1 introduces us to the typical queries that benefit from an index that is designed to support multidimensional data and multidimensional queries. Then, in Sections 5.2 and 5.3 we discuss the following data structures:

1. *Grid files*, a multidimensional extension of one-dimensional hash-tables.

2. *Partitioned hash functions,* another way that brings hash-table ideas to multidimensional data.

3. *Multiple-key indexes,* in which the index on one attribute A leads to indexes on another attribute B for each possible value of A.

4. *kd-trees*, an approach to generalizing B-trees to sets of points.

5. *Quad trees,* which are multiway trees in which each child of a node represents a quadrant of a larger space.

6. *R-trees,* a B-tree generalization suitable for collections of regions.

Finally, Section 5.4 discusses an index structure called *bitmap indexes.* These indexes are succinct codes for the location of records with a given value in a given field. They are today beginning to appear in the major commercial DBMS's, and they sometimes are an excellent choice for a one-dimensional index. However, they also can be a powerful tool for answering certain kinds of multidimensional queries.

5.1 Applications Needing Multiple Dimensions

We shall consider two general classes of multidimensional applications. One is *geographic* in nature, where the data is elements in a two-dimensional world, or sometimes a three-dimensional world. The second involves more abstract notions of dimensions. Roughly, every attribute of a relation can be thought of as a dimension, and all tuples are points in a space defined by those dimensions.

Also in this section is an analysis of how conventional indexes, such as B-trees, could be used to support multidimensional queries. While in some cases they are adequate, there are also examples where they are clearly dominated by more specialized structures.

5.1.1 Geographic Information Systems

A *geographic information system* stores objects in a (typically) two-dimensional space. The objects may be points or shapes. Often, these databases are maps, where the stored objects could represent houses, roads, bridges, pipelines, and many other physical objects. A suggestion of such a map is in Fig. 5.1.

However, there are many other uses as well. For instance, an integrated-circuit design is a two-dimensional map of regions, often rectangles, composed of specific materials, called "layers." Likewise, we can think of the windows and icons on a screen as a collection of objects in two-dimensional space.

The queries asked of geographic information systems are not typical of SQL queries, although many can be expressed in SQL with some effort. Examples of these types of queries are:

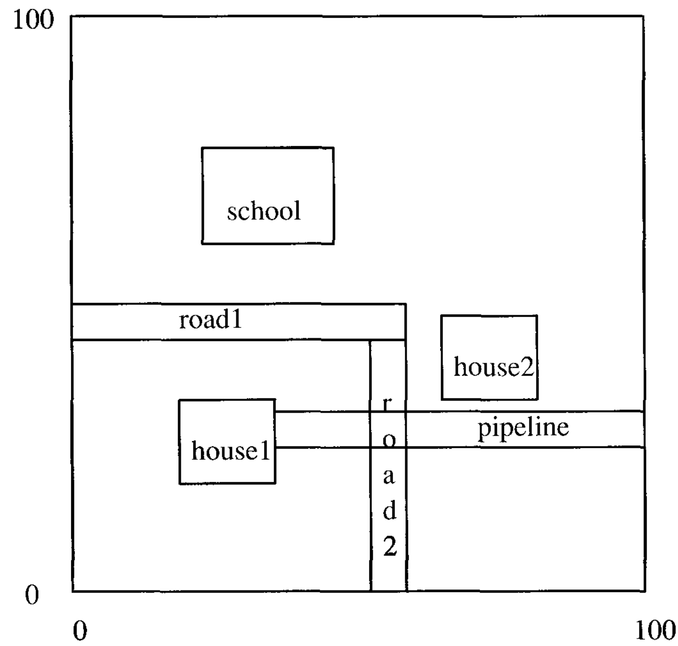

Figure 5.1: Some objects in 2-dimensional space

1. *Partial match queries.* We specify values for one or more dimensions and look for all points matching those values in those dimensions.

2. *Range queries.* We give ranges for one or more of the dimensions, and we ask for the set of points within those ranges, or if shapes are represented, then the set of shapes that are partially or wholly within the range. These queries generalize the one-dimensional range queries that we considered in Section 4.3.4.

3. *Nearest-neighbor queries.* We ask for the closest point to a given point. For instance, if points represent cities, we might want to find the city of over 100,000 population closest to a given small city.

4. *Where-am-Iqueries.* We are given a point and we want to know in which shape, if any, the point is located. A familiar example is what happens when you click your mouse, and the system determines which of the displayed elements you were clicking.

3.1.2 Data Cubes

A recent development is a family of DBMS's, sometimes called *data cube* systems, that see data as existing in a high-dimensional space. These are discussed in more detail in Section 11.4, but the following example suggests the main idea.

Multidimensional data is gathered by many corporations for *decision-support* applications, where they analyze information such as sales to better understand company operations. For example, a chain store may record each sale made, including:

1. The day and time.
2. The store at which the sale was made.
3. The item purchased.
4. The color of the item.
5. The size of the item.

and perhaps other properties of the sale.

It is common to view the data as a relation with an attribute for each property. These attributes can be seen as dimensions of a multidimensional space, the "data cube." Each tuple is *a* point in the space. Analysts then ask queries that typically group the data along some of the dimensions and summarize the groups by an aggregation. A typical example would be "give the sales of pink shirts for each store and each month of 1998."

5.1.3 Multidimensional Queries in SQL

It is possible to set up each of the applications suggested above as a conventional, relational database and to issue the suggested queries in SQL. Here are some examples.

Example 5.2 : Suppose we wish to answer nearest-neighbor queries about a set of points in two-dimensional space. We may represent the points as a relation consisting of a pair of reals:

```
Points(x, y)
```

That is, there are two attributes, x and y, representing the x- and y-coordinates of the point. Other, unseen, attributes of relation Points may represent properties of the point.

Suppose we want the nearest point to the point $(10.0, 20.0)$. The query of Fig. 5.2 finds the nearest point, or points if there is a tie. It asks, for each point p, whether there exists another point q that is closer to $(10.0, 20.0)$. Comparison of distances is carried out by computing the sum of the squares of the differences in the x- and y-coordinates between the point $(10.0, 20.0)$ and the points in question. Notice that we do not have to take the square roots of the sums to get the actual distances; comparing the squares of the distances is the same as comparing the distances themselves. □

```
SELECT *
FROM POINTS p
WHERE NOT EXISTS(
    SELECT *
    FROM POINTS q
    WHERE (q.x-10.0)*(q.x-10.0) + (q.y-20.0)*(q.y-20.0) <
          (p.x-10.0)*(p.x-10.0) + (p.y-20.0)*(p.y-20.0)
);
```

Figure 5.2: Finding the points with no point nearer to $(10.0, 20.0)$

Example 5.3: Rectangles are a common form of shape used in geographic systems. We can represent a rectangle in several ways; a popular one is to give the coordinates of the lower-left and upper-right corners. We then represent a collection of rectangles by a relation Rectangles with attributes for a rectangle-ID, the four coordinates that describe the rectangle, and any other properties of the rectangle that we wished to record. We shall use the relation

```
Rectangles(id, xll, yll, xur, yur)
```

in this example. The attributes are the rectangle's ID, the x-coordinate of its lower-left corner, the y-coordinate of that corner, and the two coordinates of the upper-right corner, respectively.

Figure 5.3 is a query that asks for the rectangle(s) enclosing the point (10.0,20.0). The where-clause condition is straightforward. For the rectangle to enclose $(10.0, 20.0)$, the lower-left corner must have its x-coordinate at or to the left of 10.0, and its y-coordinate at or below 20.0. The upper right corner must also be at or to the right of $x = 10.0$ and at or above $y = 20.0$. □

```
SELECT id
FROM Rectangles
WHERE xll <= 10.0 AND yll <= 20.0 AND
      xur >= 10.0 AND yur >= 20.0;
```

Figure 5.3: Finding the rectangles that contain a given point

Example 5.4: Data suitable for a data-cube system is typically organized into a *fact table*, which gives the basic elements being recorded (e.g., each sale), and *dimension tables,* which give properties of the values along each dimension. For instance, if the store at which a sale was made is a dimension, the dimension table for stores might give the address, phone, and name of the store's manager.

In this example, we shall deal only with the fact table, which we assume has the dimensions suggested in Section 5.1.2. That is, the fact table is the relation

Sales(day, store, item, color, size)

The query "summarize the sales of pink shirts by day and store" is shown in Fig. 5.4. It uses grouping to organize sales by the dimensions day and store, while summarizing the other dimensions through the COUNT aggregation operator. We focus on only those points of the data cube that we care about by using the WHERE-clause to select only the tuples for pink shirts. □

```
SELECT day, store, COUNT(*) AS totalSales
FROM Sales
WHERE item = 'shirt' AND
      color = 'pink'
GROUP BY day, store;
```

Figure 5.4: Summarizing the sales of pink shirts

5.1.4 Executing Range Queries Using Conventional Indexes

Now, let us consider to what extent the indexes described in Chapter 4 would help in answering range queries. Suppose for simplicity that there are two dimensions. We could put a secondary index on each of the dimensions, x and y. Using a B+ tree for each would make it especially easy to get a range of values for each dimension.

Given ranges in both dimensions, we could begin by using the B-tree for x to get pointers to all of the records in the range for x. Next, we use the B-tree for y to get pointers to the records for all points whose y-coordinate is in the range for y. Finally, we intersect these pointers using the idea of Section 4.2.3. If the pointers fit in main memory, then the total number of disk I/O's is the number of leaf nodes of each B-tree that need to be examined, plus a few I/0's for finding our way down the B-trees (see Section 4.3.7). To this amount we must add the disk I/O's needed to retrieve all the matching records, however many they may be.

Example 5.5: Let us consider a hypothetical set of 1,000,000 points distributed randomly in a space in which both the x- and y-coordinates range from 0 to 1000. Suppose that 100 point records fit on a block, and an average B-tree leaf has about 200 key-pointer pairs (recall that not all slots of a B-tree block are necessarily occupied, at any given time). We shall assume there are B-tree indexes on both x and y.

Imagine we are given the range query asking for points in the square of side 100 surrounding the center of the space, that is, $450 < x < 550$ and $450 < y < 550$. Using the B-tree for x, we can find pointers to all the records with x in the range; there should be about 100,000 pointers, and this number of pointers should fit in main memory. Similarly, we use the B-tree for y to get the pointers to all the records with y in the desired range; again there are about 100,000 of them. Approximately 10,000 pointers will be in the intersection of these two sets, and it is the records reached by the 10,000 pointers in the intersection that form our answer.

Now, let us estimate the number of disk I/O's needed to answer the range query. First, as we pointed out in Section 4.3.7, it is generally feasible to keep the root of any B-tree in main memory. Since we are looking for a range of search-key values in each B-tree, and the pointers at the leaves are sorted by this search key, all we have to do to access the 100,000 pointers in either dimension is examine one intermediate-level node and all the leaves that contain the desired pointers. Since we assumed leaves have about 200 key-pointer pairs each, we shall have to look at about 500 leaf blocks in each of the B-trees. When we add in one intermediate node per B-tree, we have a total of 1002 disk I/O's.

Finally, we have to retrieve the blocks containing the 10,000 desired records. If they are stored randomly, we must expect that they will be on almost 10,000 different blocks. Since the entire file of a million records is assumed stored over 10,000 blocks, packed 100 to a block, we essentially have to look at every block of the data file anyway. Thus, in this example at least, conventional indexes have been little if any help in answering the range query. Of course, if the range were smaller, then constructing the intersection of the two pointer sets would allow us to limit the search to a fraction of the blocks in the data file. □

5.1.5 Executing Nearest-Neighbor Queries Using Conventional Indexes

Almost any data structure we use will allow us to answer a nearest-neighbor query by picking a range in each dimension, asking the range query, and selecting the point closest to the target within that range. Unfortunately, there are two things that could go wrong:

1. There is no point within the selected range.

2. The closest point within the range might not be the closest point overall.

Let us consider each of these problems in the context of the nearest-neighbor query of Example 5.2, using the hypothetical indexes on dimensions x and y introduced in Example 5.5. If we had reason to believe that a point within distance d of $(10.0, 20.0)$ existed, we could use the B-tree for x to get pointers to all the records for points whose x-coordinate is between $10 - d$ and $10 + d$. We could then use the B-tree for y to get pointers to all the records whose y-coordinate is between $20 - d$ and $20 + d$.

If we have one or more points in the intersection, and we have recorded with each pointer its x- or y-coordinate (whichever is the search key for the index), then we have the coordinates of all the points in the intersection. We can thus determine which of these points is closest to $(10.0, 20.0)$ and retrieve only its record. Unfortunately, we cannot be certain that there are any points within distance d of the given point, so we may have to repeat the entire process with a higher value of d.

However, even if there is a point in the range we have searched, there are some circumstances where the closest point in the range is further than distance d from the target point, e.g., $(10.0, 20.0)$ in our example. The situation is suggested by Fig. 5.5. If that is the case, then we must expand our range and search again, to make sure that no closer point exists. If the distance from the target to the closest point found so far is d', and $d' > d$, then we must repeat the search with d' in place of d.

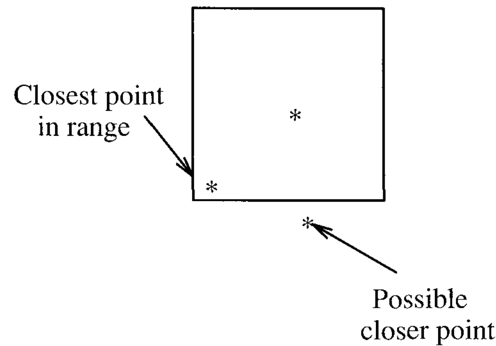

Figure 5.5: The point is in the range, but there could be a closer point outside the range

Example 5.6: Let us consider the same data and indexes as in Example 5.5. If we want the nearest neighbor to target point P = (10.0, 20.0), we might pick $d = 1$. Then, there will be one point per unit of area on the average, and with $d = 1$ we find every point within a square of side 2.0 around the point P, wherein the expected number of points is 4.

If we examine the B-tree for the x-coordinate with the range query $9.0 < x < 11.0$, then we shall find about 2,000 points, so we need to traverse at least 10 leaves, and most likely 11 (since the points with $x = 9.0$ are unlikely to start just at the beginning of a leaf). As in Example 5.5, we can probably keep the roots of the B-trees in main memory, so we only need one disk I/O for an intermediate node and 11 disk I/O's for the leaves. Another 12 disk I/O's search the B-tree index on the y-coordinate for the points whose y-coordinate is between 19.0 and 21.0.

If we intersect the approximately 4000 pointers in main memory, we shall find about four records that are candidates for the nearest neighbor of point

(10.0, 20.0). Assuming there is at least one. we can determine from the associated x- and y-coordinates of the pointers which is the nearest neighbor. One more disk I/O to retrieve the desired record, for a total of 23 disk I/O's, completes the query. However, if there is no point in the square with $d = 1$, or the closest point is more than distance 1 from the target, then we have to repeat the search with a larger value of d. □

The conclusion to draw from Example 5.6 is that conventional indexes might not be terrible for a nearest-neighbor query, but they use significantly more disk I/0's than would be used, say, to find a record given its key and a B-tree index on that key (which would probably take only two or three disk I/0's). The methods suggested in this chapter will generally provide better performance and are used in specialized DBMS's that support multidimensional data.

5.1.6 Other Limitations of Conventional Indexes

The previously mentioned structures fare no better for range queries than for nearest-neighbor queries. In fact, our approach to solving a nearest-neighbor query in Example 5.6 was really to convert it to a range-query with a small range in each dimension and hope the range was sufficient to include at least one point. Thus, if we were to tackle a range query with larger ranges, and the data structure were indexes in each dimension, then the number of disk I/0's necessary to retrieve the pointers to candidate records in each dimension would be even greater than what we found in Example 5.6.

The multidimensional aggregation of the query in Fig. 5.4 is likewise not well supported. If we have indexes on item and color, we can find all the records representing sales of pink shirts and intersect them, as we did in Example 5.6. However, queries in which other attributes besides item and color were specified would require indexes on those attributes instead.

Worse, while we can keep the data file sorted on one of the five attributes, we cannot keep it sorted on two attributes, let alone five. Thus, most queries of the form suggested by Fig. 5.4 would require that records from all or almost all of the blocks of the data file be retrieved. These queries of this type would be extremely expensive to execute if data was in secondary memory.

5.1.7 Overview of Multidimensional Index Structures

Most data structures for supporting queries on multidimensional data fall into one of two categories:

1. Hash-table-like approaches.

2. Tree-like approaches.

For each of these structures, we give up something that we have in the one-dimensional structures of Chapter 4.

- With the hash-bashed schemes — grid files and partitioned hash functions in Section 5.2 — we no longer have the advantage that the answer to our query is in exactly one bucket. However, each of these schemes limit our search to a subset of the buckets.

- With the tree-based schemes, we give up at least one of these important properties of B-trees:

  1. The balance of the tree, where all leaves are at the same level.
  2. The correspondence between tree nodes and disk blocks.
  3. The speed with which modifications to the data may be performed.

  As we shall see in Section 5.3, trees will often be deeper in some parts than in others; often the deep parts correspond to regions that have many points. We shall also see that it is common that the information corresponding to a tree node is considerably smaller than what fits in one block. It is thus necessary to group nodes into blocks in some useful way.

5.1.8 Exercises for Section 5.1

Exercise 5.1.1: Write SQL queries using the relation

```
Rectangles(id, xll, yll, xur, yur)
```

from Example 5.3 to answer the following questions:

* a) Find the set of rectangles that intersect the rectangle whose lower-left corner is at $(10.0, 20.0)$ and whose upper-right corner is at $(40.0, 30.0)$.

b) Find the pairs of rectangles that intersect.

c) Find the rectangles that completely contain the rectangle mentioned in (a).

d) Find the rectangles that are completely contained within the rectangle mentioned in (a).

! e) Find the "rectangles" in the relation Rectangles that are not really rectangles; i.e., they cannot exist physically.

For each of these queries, tell what indexes, if any, would help retrieve the desired tuples.

Exercise 5.1.2: Using the relation

```
Sales(day, store, item, color, size)
```

from Example 5.4, write the following queries in SQL:

* a) List all colors of shirts and their total sales, provided there are more than 1000 sales for that color.

b) List sales of shirts by store and color.

c) List sales of all items by store and color.

! d) List for each item and color the store with the largest sales and the amount of those sales.

For each of these queries, tell what indexes, if any, would help retrieve the desired tuples.

Exercise 5.1.3 : Redo Example 5.5 under the assumption that the range query asks for a square in the middle that is $n \times n$ for some n between 1 and 1000. How many disk I/O's are needed? For which values of n do indexes help?

* **Exercise 5.1.4:** Repeat Exercise 5.1.3 if the file of records is sorted on x.

!! **Exercise 5.1.5 :** Suppose that we have points distributed randomly in a square, as in Example 5.6, and we want to perform a nearest neighbor query. We choose a distance d and find all points in the square of side $2d$ with the center at the target point. Our search is successful if we find within this square at least one point whose distance from the target point is d or less.

* a) If there is on average one point per unit of area, give as a function of d the probability that we will be successful.

b) If we are unsuccessful, we must repeat the search with a larger d. Assume for simplicity that each time we are unsuccessful, we double d and pay twice as much as we did for the previous search. Again assuming that there is one point per unit area, what initial value of d gives us the minimum expected search cost?

5.2 Hash-Like Structures for Multidimensional Data

In this section we shall consider two data structures that generalize hash tables built using a single key. In each case, the bucket for a point is a function of all the attributes or dimensions. One scheme, called the "grid file," usually doesn't "hash" values along the dimensions, but rather partitions the dimensions by sorting the values along that dimension. The other, called "partitioned hashing," does "hash" the various dimensions, with each dimension contributing to the bucket number.

5.2.1 Grid Files

One of the simplest data structures that often outperforms single-dimension indexes for queries involving multidimensional data is the *grid file*. Think of the space of points partitioned in a grid. In each dimension, *grid lines* partition the space into *stripes*. Points that fall on a grid line will be considered to belong to the stripe for which that grid line is the lower boundary. The number of grid lines in different dimensions may vary, and there may be different spacings between adjacent grid lines, even between lines in the same dimension.

Example 5.7: Let us introduce a running example for this chapter: the question "who buys gold jewelry?" We shall imagine a database of customers for gold jewelry that tells us many things about each customer — their name, address, and so on. However, to make things simpler, we assume that the only relevant attributes are the customer's age and salary. Our example database has twelve customers, which we can represent by the following age-salary pairs:

(25,60) (45,60) (50,75) (50,100)
(50,120) (70,110) (85,140) (30,260)
(25,400) (45,350) (50,275) (60,260)

In Fig. 5.6 we see these twelve points located in a 2-dimensional space. We have also selected some grid lines in each dimension. For this simple example, we have chosen two lines in each dimension, dividing the space into nine rectangular regions, but there is no reason why the same number of lines must be used in each dimension. We have also allowed the spacing between the lines to vary. For instance, in the age dimension, the three regions into which the two vertical lines divide the space have width 40, 15, and 45.

In this example, no points are exactly on a grid line. But in general, a rectangle includes points on its lower and left boundaries, but not on its upper and right boundaries. For instance, the central rectangle in Fig. 5.6 represents points with $40 < age < 55$ and $90 < salary < 225$. □

5.2.2 Lookup in a Grid File

Each of the regions into which a space is partitioned can be thought of as a bucket of a hash table, and each of the points in that region has its record placed in a block belonging to that bucket. If needed, overflow blocks can be used to increase the size of a bucket.

Instead of a one-dimensional array of buckets, as is found in conventional hash tables, the grid file uses an array whose number of dimensions is the same as for the data file. To locate the proper bucket for a point, we need to know, for each dimension, the list of values at which the grid lines occur. Hashing a point is thus somewhat different from applying a hash function to the values of its components. Rather, we look at each component of the point and determine the position of the point in the grid for that dimension. The positions of the point in each of the dimensions together determine the bucket.

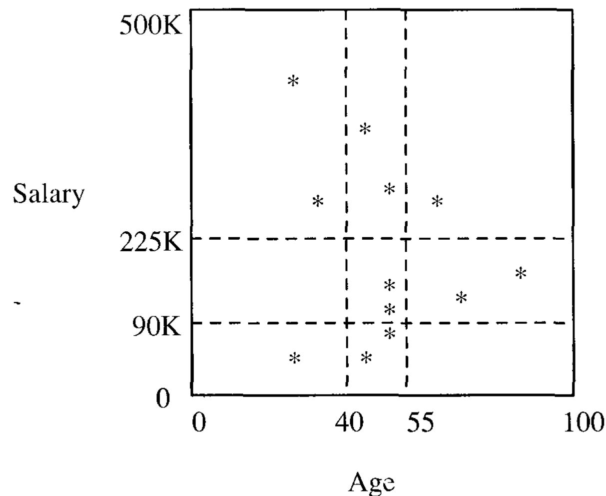

Figure 5.6: A grid file

Example 5.8: Figure 5.7 shows the data of Fig. 5.6 placed in buckets. Since the grids in both dimensions divide the space into three regions, the bucket array is a 3 x 3 matrix. Two of the buckets:

1. Salary between $90K and $225K and age between 0 and 40, and

2. Salary below $90K and age above 55

are empty, and we do not show a block for that bucket. The other backets are shown, with the artificially low maximum of two data points per block. In this simple example, no bucket has more than two members, so no overflow blocks are needed. □

5.2.3 Insertion Into Grid Files

When we insert a record into a grid file, we follow the procedure for lookup of the record, and we place the new record in that bucket. If there is room in the block for the bucket then there is nothing more to do. The problem occurs when there is no room in the bucket. There are two general approaches:

1. Add overflow blocks to the buckets, as needed. This approach works well as long as the chains of blocks for a bucket do not get too large. If they do, then the number of disk I/O's needed for lookup, insertion, or deletion eventually grows unacceptably large.

2. Reorganize the structure by adding or moving the grid lines. This approach is similar to the dynamic hashing techniques discussed in Section 4.4, but there are additional problems because the contents of buckets are linked across a dimension. That is, adding a grid line splits all the

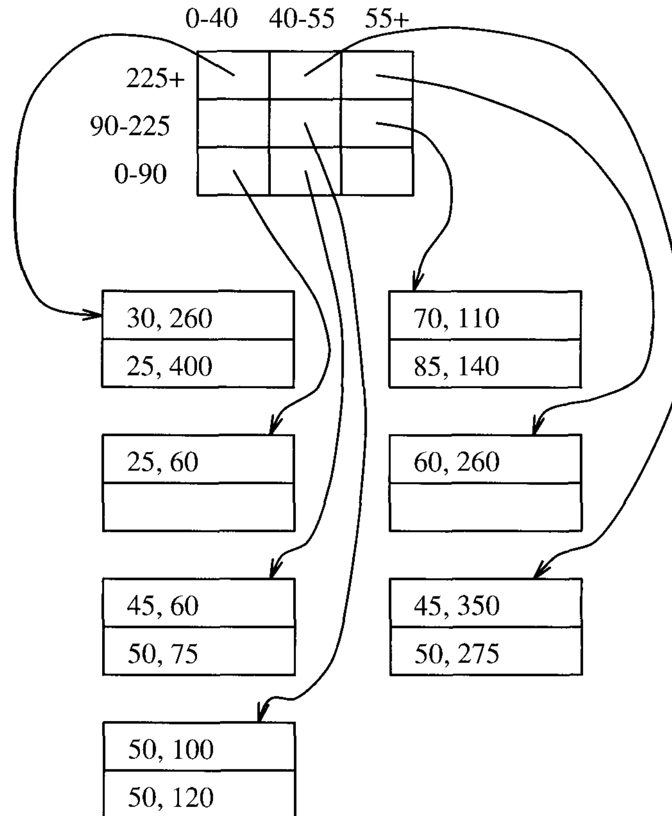

Figure 5.7: A grid file representing the points of Fig. 5.6

buckets along that line. As a result, it may not be possible to select a new grid line that does the best for all buckets. For instance, if one bucket is too big, we might not be able to choose either the dimension of the split or the point of the split without making many empty buckets or leaving several very full ones.

Example 5.9: Suppose someone 52 years old with an income of $200K buys gold jewelry. This customer belongs in the central rectangle of Fig. 5.6. However, there are now three records in that bucket. We could simply add an overflow block. If we want to split the bucket, then we need to choose either the age or salary dimension, and we need to choose a new grid line to create the division. There are only three ways to introduce a grid line that will split the central bucket so two points are on one side and one on the other, which is the most even possible split in this case.

1. A vertical line, such as age = 51, that separates the two 50's from the 52. This line does nothing to split the buckets above or below, since both

Accessing Buckets of a Grid File

While finding the proper coordinates for a point in a three-by-three grid like Fig. 5.7 is easy, we should remember that the grid file may have a very large number of stripes in each dimension. If so, then we must create an index for each dimension. The search key for an index is the set of partition values in that dimension.

Given a value v in some coordinate, we search for the greatest key value w less than or equal to v. Associated with w in that index will be the row or column of the matrix into which v falls. Given values in each dimension, we can find where in the matrix the pointer to the bucket falls. We may then retrieve the block with that pointer directly.

In extreme cases, the matrix is so big, that most of the buckets are empty and we cannot afford to store all the empty buckets. Then, we must treat the matrix as a relation whose attributes are the corners of the nonempty buckets and a final attribute representing the pointer to the bucket. Lookup in this relation is itself a multidimensional search, but its size is smaller than the size of the data file itself.

points of each of the other buckets for age 40-55 will be to the left of the line age = 51.

2. A horizontal line that separates the point with salary = 200 from the other two points in the central bucket. We may as well choose a number like 130, which will also split the bucket to the right (that for age 55-100 and salary 90-225).

3. A horizontal line that separates the point with salary = 100 from the other two points. Again, we would be advised to pick a number like 115 that also splits the bucket to the right.

Choice (1) is probably not advised, since it doesn't split any other bucket; we are left with more empty buckets and have not reduced the size of any occupied buckets. Choices (2) and (3) are equally good, although we might pick (2) because it puts the horizontal grid line at salary = 130, which is closer to midway between the upper and lower limits of 90 and 225 than we get with choice (3). The resulting partition into buckets is shown in Fig. 5.8. □

5.2.4 Performance of Grid Files

Let us consider how many disk I/O's a grid file requires on various types of queries. We have been focusing on the two-dimensional version of grid files, although they can be used for any number of dimensions. One major problem

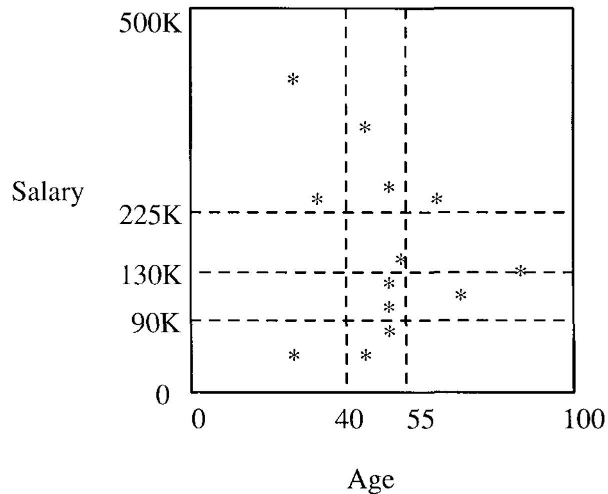

Figure 5.8: Insertion of the point (52, 200) followed by splitting of buckets

in the high-dimensional case is that the number of buckets grows exponentially with the dimension. If large portions of a space are empty, then there will be many empty buckets. We can envision the problem even in two dimensions. Suppose that there were a high correlation between age and salary, so all points in Fig. 5.6 lay along the diagonal. then no matter where we placed the grid lines, the buckets off the diagonal would have to be empty.

However, if the data is well distributed, and the data file itself is not too large, then we can choose grid lines so that:

1. There are sufficiently few buckets that we can keep the bucket matrix in main memory, thus not incurring disk I/O to consult it, or to add rows or columns to the matrix when we introduce a new grid line.

2. We can also keep in memory indexes on the values of the grid lines in each dimension (see the box on "accessing buckets of a grid file"), or we can avoid the indexes altogether and use main-memory binary search of the values defining the grid lines in each dimension.

3. The typical bucket does not have more than a few overflow blocks, so we do not incur too many disk I/O's when we search through a bucket.

Under those assumptions, here is how the grid file behaves on some important classes of queries.

Lookup of Specific Points

We are directed to the proper bucket, so the only disk I/O is what is necessary to read the bucket. If we are inserting or deleting, then an additional disk

write is needed. Inserts that require the creation of an overflow block cause an additional write.

Partial-Match Queries

Examples of this query would include "find all customers aged 50," or "find all customers with a salary of $200K." Now, we need to look at all the buckets in a row or column of the bucket matrix. The number of disk I/O's can be quite high if there are many buckets in these rows or columns.

Range Queries

A range query defines a rectangular region of the grid, and all points found in the buckets that cover that region will be answers to the query, with the exception of some of the points in buckets on the border of the search region. For example, if we want to find all customers aged 35-45 with a salary of 50-100, then we need to look in the four buckets in the lower left of Fig. 5.6. In this case, all buckets are on the border, so we may look at a good number of points that are not answers to the query. However, if the search region involves a large number of buckets, then most of them must be interior, and all their points are answers. For range queries, the number of disk I/0's may be large, as we may be required to examine many buckets. However, since range queries tend to produce large answer sets, we typically will examine not too many more blocks than the minimum number of blocks on which the answer could be placed by any organization whatsoever.

Nearest-Neighbor Queries

Given a point P, we start by searching the bucket in which that point belongs. If we find at least one point there, we have a candidate Q for the nearest neighbor. However, it is possible that there are points in adjacent buckets that are closer to P than Q is; the situation is like that suggested in Fig. 5.5. We have to consider whether the distance between P and a border of its bucket is less than the distance from P to Q. If there are such borders, then the adjacent buckets on the other side of each such border must be searched also. In fact, if buckets are severely rectangular — much longer in one dimension than the other — then it may be necessary to search even buckets that are not adjacent to the one containing point P.

Example 5.10: Suppose we are looking in Fig. 5.6 for the point nearest $P =$ (45,200). We find that $(50, 120)$ is the closest point in the bucket, at a distance of 80.2. No point in the lower three buckets can be this close to $(45, 200)$, because their salary component is at most 90, so we can omit searching them. However, the other five buckets must be searched, and we find that there are actually two equally close points: (30,260) and $(60, 260)$, at a distance of 61.8 from P. Generally, the search for a nearest neighbor can be limited to a few

buckets, and thus a few disk I/O's. However, since the buckets nearest the point *P* may be empty, we cannot easily put an upper bound on how costly the search is. □

5.2.5 Partitioned Hash Functions

Hash functions can take a list of attribute values as an argument, although typically they hash values from only one attribute. For instance, if *a* is an integer-valued attribute and 6 is a character-string-valued attribute, then we could add the value of *a* to the value of the ASCII code for each character of *b,* divide by the number of buckets, and take the remainder. The result could be used as the bucket number of a hash table suitable as an index on the pair of attributes *(a, b).*

However, such a hash table could only be used in queries that specified values for both *a* and *b.* A preferable option is to design the hash function so it produces some number of bits, say *k.* These *k* bits are divided among *n* attributes, so that we produce k_i bits of the hash value from the ith attribute, for $i = 1,2,\ldots,n$, where $\sum_{i=1}^{n} k_i = k$. More precisely, the hash function *h* is actually a list of hash functions $(h_1, h_2, \ldots, h_n)$, such that h_i applies to a value for the ith attribute and produces a sequence of k_i bits. The bucket in which to place a tuple with values $(v_1, v_2, \ldots, v_n)$in the *n* attributes that are involved in the hashing is computed by concatenating the bit sequences $h_1(v_1)h_2(v_2)\cdots h_n(v_n)$.

Example 5.11: If wo have a hash table with 10-bit bucket numbeis (1024 buckets), we could devote four bits to attribute *a* and the remaining six bits to attribute 6. Suppose we have a tuple with a-value A and b-value B, perhaps with other attributes that are not involved in the hash. We hash *A* using a hash function h_a associated with attribute *a* to get four bits, say 0101. We then hash *B,* using a hash function h_b, perhaps receiving the six bits 111000. The bucket number for this tuple is thus 0101111000, the concatenation of the two bit sequences.

By partitioning the hash function this way, we get some advantage from knowing values for any one or more of the attributes that contribute to the hash function. For instance, if we aie given a value *A* for attribute *a,* and we find that $h_a(A) = 0101$, then we know that the only tuples with a-value *A* are in the 64 buckets whose numbers arc of the form 0101 • • • , where the • • • represents any six bits. Similarly, if we are given the b-value B of a tuple, we can isolate the possible buckets of the tuple to the 16 buckets whose number ends in the six bits $h_b(B)$. □

Example 5.12: Suppose we have the "gold jewelry" data of Example 5.7, which we want to store in a partitioned hash table with eight buckets (i.e., three bits for bucket numbers). We assume as before that two records are all that can fit in one block. We shall devote one bit to the age attribute and the remaining two bits to the salary attribute.

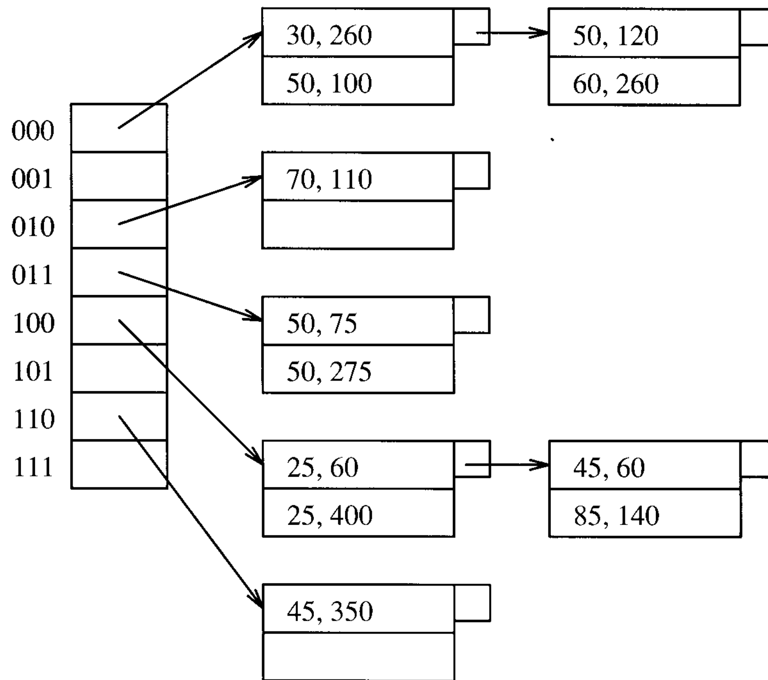

Figure 5.9: A partitioned hash table

For the hash function on age, we shall take the age modulo 2; that is, a record with an even age will hash into a bucket whose number is of the form $0xy$ for some bits x and y. A record with an odd age hashes to one of the buckets with a number of the form $1xy$. The hash function for salary will be the salary (in thousands) modulo 4. For example, a salary that leaves a remainder of 1 when divided by 4, such as 57K, will be in a bucket whose number is $z01$ for some bit z.

In Fig. 5.9 we see the data from Example 5.7 placed in this hash table. Notice that, because we have used mostly ages and salaries divisible by 10, the hash function does not distribute the points too well. Two of the eight buckets have four records each and need overflow blocks, while three other buckets are empty. □

5.2.6 Comparison of Grid Files and Partitioned Hashing

The performance of the two data structures discussed in this section are quite different. Here are the major points of comparison.

- Partitioned hash tables are actually quite useless for nearest-neighbor queries or range queries. The problem is that physical distance between points is not reflected by the closeness of bucket numbers. Of course we

could design the hash function on some attribute a so the smallest values were assigned the first bit string (all 0's), the next values were assigned the next bit string $(00 \cdots 01)$, and so on. If we do so, then we have reinvented the grid file.

- A well chosen hash function will randomize the buckets into which points fall, and thus buckets will tend to be equally occupied. However, grid files, especially when the number of dimensions is large, will tend to leave many buckets empty or nearly so. The intuitive reason is that when there are many attributes, there is likely to be some correlation among at least some of them, so large regions of the space are left empty. For instance, we mentioned in Section 5.2.4 that a correlation between age and salary would cause most points of Fig. 5.6 to lie near the diagonal, with most of the rectangle empty. As a consequence, we can use fewer buckets, and/or have fewer overflow blocks in a partitioned hash table than in a grid file.

Thus, if we are only required to support partial match queries, where we specify some attributes' values and leave the other attributes completely unspecified, then the partitioned hash function is likely to outperform the grid file. Conversely, if we need to do nearest-neighbor queries or range queries frequently, then we would prefer to use a grid file.

5.2.7 Exercises for Section 5.2

model	*speed*	*ram*	*hard-disk*
A	300	32	6.0
B	333	64	4.0
C	400	64	12.7
D	350	32	10.8
E	450	96	14.0
F	400	128	12.7
G	450	128	18.1
H	233	32	4.0
I	266	64	6.0
J	300	64	6.0
K	350	64	12.0
L	400	128	6.0

Figure 5.10: Some PC's and their characteristics

Exercise 5.2.1: In Fig. 5.10 are specifications for twelve PC's. Suppose we wish to design an index on speed and hard-disk size only.

* a) Choose five grid lines (total for the two dimensions), so that there are no more than two points in any bucket.

Handling Tiny Buckets

We generally think of buckets as containing about one block's worth of data. However, there are reasons why we might need to create so many buckets that the average bucket has only a small fraction of the number of records that will fit in a block. For example, high-dimensional data will require many buckets if we are to partition significantly along each dimension. Thus, in the structures of this section and also for the tree-based schemes of Section 5.3, we might choose to pack several buckets (or nodes of trees) into one block. If we do so, there are some important points to remember:

- The block must keep in its header information about where each record is, and to which bucket it belongs.
- If we insert a record into a bucket, wo may not have room in the block containing that bucket. If so, we need to split the block in some way. We must decide which buckets go with each block, find the records of each bucket and put them in the proper block, and adjust the bucket table to point to the proper block.

! b) Can you separate the points with at most two per bucket if you use only four grid lines? Either show how or argue that it is not possible.

! c) Suggest a partitioned hash function that will partition these points into four buckets with at most four points per bucket.

! Exercise **5.2.2**: Suppose we wish to place the data of Fig. 5.10 in a three-dimensional grid file, based on the speed, ram, and hard-disk attributes. Suggest a partition in each dimension that will divide the data well.

Exercise 5.2.3: Choose a partitioned hash function with one bit for each of the three attributes speed, ram, and hard-disk that divides the data of Fig. 5.10 well.

Exercise 5.2.4: Suppose we place the data of Fig. 5.10 in a grid file with dimensions for speed and ram only. The partitions are at speeds of 310, 375, and 425, and at ram of 40 and 75. Suppose also that only two points can fit in one bucket. Suggest good splits if we insert points at:

* a) Speed = 250 and ram = 48.

b) Speed = 333 and ram = 48.

Exercise 5.2.5: Suppose we store a relation $R(x, y)$ in a grid file. Both attributes have a range of values from 0 to 1000. The partitions of this grid file happen to be uniformly spaced; for *x* there are partitions every 20 units, at 20, 40, 60, and so on, while for *y* the partitions are every 50 units, at 50, 100, 150, and so on.

a) How many buckets do we have to examine to answer the range query

```
SELECT *
FROM R
WHERE 310 < x AND x < 400 AND 520 < y AND y < 730;
```

*! b) We wish to perform a nearest-neighbor query for the point $(110, 205)$. We begin by searching the bucket with lower-left corner at (100, 200) and upper-right corner at $(120, 250)$, and we find that the closest point in this bucket is $(115, 220)$. What other buckets must be searched to verify that this point is the closest?

! **Exercise 5.2.6**: Suppose we have a grid file with three lines (i.e., four stripes) in each dimension. However, the points *(x, y)* happen to have a special property. Tell the largest possible number of nonempty buckets if:

* a) The points are on a line; i.e., there is are constants a and *b* such that $y = ax + b$ for every point *(x, y)*.

b) The points are related quadratically; i.e., there are constants *a, b,* and c such that $y = ax^2 + bx + c$ for every point *(x, y)*.

Exercise 5.2.7: Suppose we store a relation *R(x,y,z) in* a partitioned hash table with 1024 buckets (i.e., 10-bit bucket addresses). Queries about *R* each specify exactly one of the attributes, and each of the three attributes is equally likely to be specified. If the hash function produces 5 bits based only on *x,* 3 bits based only on y, and 2 bits based only on *z,* what is the average number of buckets that need to be searched to answer a query?

!! **Exercise 5.2.8:** Suppose we have a hash table whose buckets are numbered 0 to 2™ — 1; i.e., bucket addresses are n bits long. We wish to store in the table a relation with two attributes *x* and y. A query will either specify a value for *x* or *y,* but never both. With probability *p,* it is *x* whose value is specified.

a) Suppose we partition the hash function so that m bits are devoted to *x* and the remaining n - *m* bits to *y.* As a function of m, n, and *p,* what is the expected number of buckets that must be examined to answer a random query?

b) For what value of *m* (as a function of n and *p)* is the expected number of buckets minimized? Do not worry that this *m* is unlikely to be an integer.

*! **Exercise 5.2.9:** Suppose wo have a relation $R(x, y)$ with 1,000,000 points randomly distributed. The range of both x and y is 0 to 1000. We can fit 100 tuples of R in a block. We decide to use a grid file with uniformly spaced grid lines in each dimension, with m as the width of the stripes. we wish to select m in order to minimize the number of disk I/O's needed to read all the necessary buckets to ask a range query that is a square 50 units on each side. You may assume that the sides of this square *never* align with the grid lines. If we pick m too large, we shall have a lot of overflow blocks in each bucket, and many of the points in a bucket will be outside the range of the query. If we pick m too small, then there will be too many buckets, and blocks will tend not to be full of data. What is the best value of *ml*

5.3 Tree-Like Structures for Multidimensional Data

We shall now consider four more structures that are useful for range queries or nearest-neighbor queries on multidimensional data. In order, wcshall consider:

1. Multiple-key indexes.
2. *kd*-trees.
3. Quad trees.
4. R-trees.

The first three are intended for sets of points. The R-tree is commonly used to represent sets of regions; it is also useful for points.

5.3.1 Multiple-Key Indexes

Suppose we have several attributes representing dimensions of our data points, and we want to support range queries or nearest-neighbor queries on these points. A simple tree-like scheme for accessing these points is an index of indexes, or more generally a tree in which the nodes at each level are indexes for one attribute.

The idea is suggested in Fig. 5.11 for the case of two attributes. The "root of the tree" is an index for the first of the two attributes. This index could be any type of conventional index, such as a B-tree or a hash table. The index associates with each of its search-key values — i.e., values for the first attribute — a pointer to another index. If *V* is a value of the first attribute, then the index we reach by following kcy *V* and its pointer is an index into the set of points that have *V* for their value in the first attribute and any value for the second attribute.

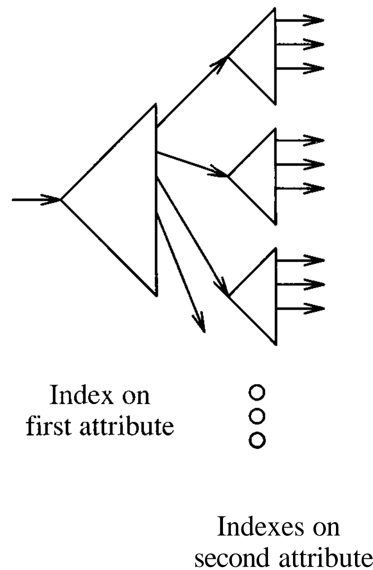

Figure 5.11: Using nested indexes on different keys

Example 5.13 : Figure 5.12 shows a multiple-key index for our running "gold jewelry" example, where the first attribute is age, and the second attribute is salary. The root index, on age, is suggested at the left of Fig. 5.12. We have not indicated how the index works. For example, the key-pointer pairs forming the seven rows of that index might be spread among the leaves of a B-tree. However, what is important is that the only keys present are the ages for which there is one or more data point, and the index makes it easy to find the pointer associated with a given key value.

At the right of Fig. 5.12 are seven indexes that provide access to the points themselves. For example, if we follow the pointer associated with age 50 in the root index, we get to a smaller index where salary is the key, and the four key values in the index are the four salaries associated with points that have age 50. Again, we have not indicated in the figure how the index is implemented, just the key-pointer associations it makes. When we follow the pointers associated with each of these values (75, 100, 120, and 275), we get to the record for the individual represented. For instance, following the pointer associated with 100, we find the person whose age is 50 and whose salary is $100K. □

In a multiple-key index, some of the second or higher rank indexes may be very small. For example, Fig 5.12 has four second-rank indexes with but a single pair. Thus, it may be appropriate to implement these indexes as simple tables that are packed several to a block, in the manner suggested by the box "Handling Tiny Buckets" in Section 5.2.5.

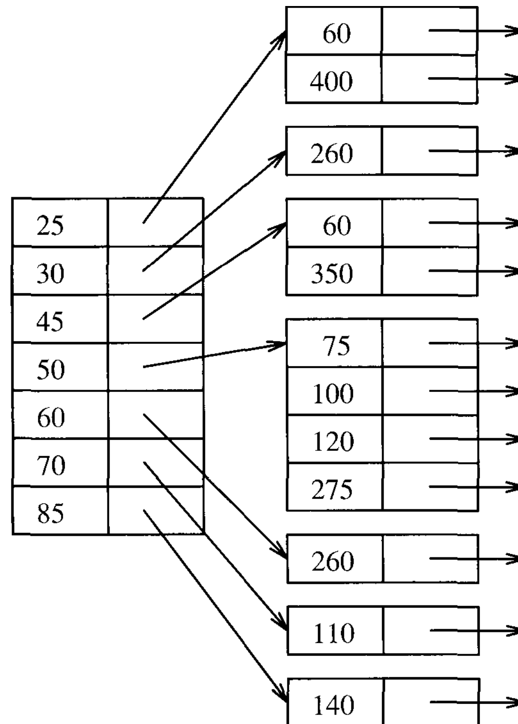

Figure 5.12: Multilevel indexes for age/salary data

5.3.2 Performance of Multiple-Key Indexes

Let us consider how a multiple key index performs on various kinds of multidimensional queries. We shall concentrate on the case of two attributes, although the generalization to more than two attributes is unsurprising.

Partial-Match Queries

If the first attribute is specified, then the access is quite efficient. e use the root index to find the one subindex that leads to the points we want. For example, if the root is a B-tree index, then we shall do two or three disk I/O's to get to the proper subindex, and then use whatever I/O's arc needed to access all of that index and the points of the data file itself. On the other hand, if the first attribute does not have a specified value, then we must search every subindex, a potentially time-consuming process.

Range Queries

The multiple-key index works quite well for a range query, provided the individual indexes themselves support range queries on their attribute (e.g.. if they are

B-tree indexes). To answer a range query, we use the root index and the range of the first attribute to find all of the subindexes that might contain answer points. We then search each of these subindexes, using the range specified for the second attribute.

Example 5.14: Suppose we have the multiple-key index of Fig. 5.12 and we are asked the range query 35 < age < 55 and 100 < salary < 200. When we examine the root index, we find that the keys 45 and 50 are in the range for age. We follow the associated pointers to two subindexes on salary. The index for age 45 has no salary in the range 100 to 200, while the index for age 50 has two such salaries: 100 and 120. Thus, the only two points in the range are $(50, 100)$ and $(50, 120)$. □

Nearest-Neighbor Queries

The answering of a nearest-neighbor query with a multiple-key index uses the same strategy as for almost all the data structures of this chapter. To find the nearest neighbor of point (x_0, y_0), we find a distance d such that we can expect to find several points within distance d of (x_0, y_0). We then ask the range query $x_0 - d < x < x_0 + d$ and $y_0 - d \leq y \leq y_0 + d$. If there turn out to be no points in this range, or if there is a point, but distance from (x_0, y_0) of the closest point is greater than d (and therefore there could be a closer point outside the range, as was discussed in Section 5.1.5), then we must increase the range and search again. However, we can order the search so the closest places are searched first.

5.3.3 *kd*-Trees

A kd-tree (k-dimensional search tree) is a main-memory data structure generalizing the binary search tree to multidimensional data. We shall present the idea and then discuss how the idea has been adapted to the block model of storage. A kd-tree is a binary tree in which interior nodes have an associated attribute a and a value V that splits the data points into two parts: those with a-value less than V and those with a-value equal to or greater than V. The attributes at different levels of the tree are different, with levels rotating among the attributes of all dimensions.

In the classical kd-tree, the data points are placed at the nodes, just as in a binary search tree. However, we shall make two modifications in our initial presentation of the idea to take some limited advantage of the block model of storage.

1. Interior nodes will have only an attribute, a dividing value for that attribute, and pointers to left and right children.

2. Leaves will be blocks, with space for as many records as a block can hold.

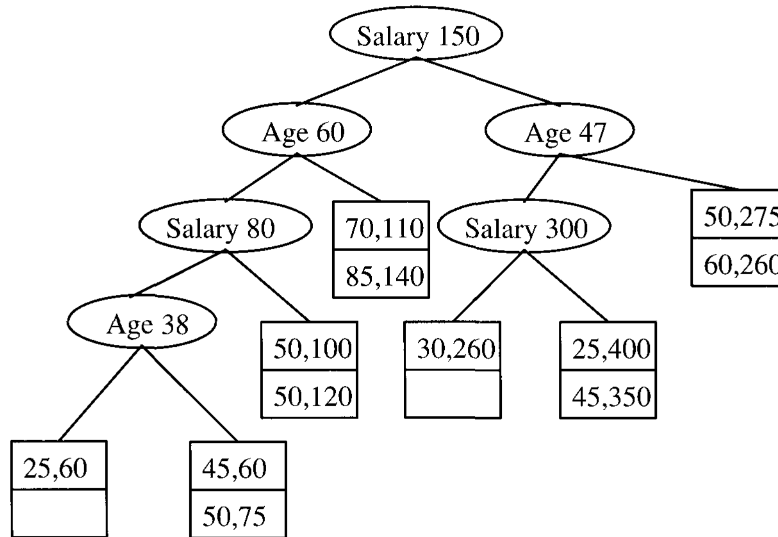

Figure 5.13: A kd-tree

Example 5.15: In Fig. 5.13 is a kd-tree for the twelve points of our running gold-jewelry example. We use blocks that hold only two records for simplicity; these blocks and their contents are shown as square leaves. The interior nodes are ovals with an attribute — either age or salary — and a value. For instance, the root splits by salary, with all records in the left subtree having a salary less than \$150K, and all records in the right subtree having a salary at least \$150K.

At the second level, the split is by age. The left child of the root splits at age 60, so everything in its left subtree will have age less than 60 and salary less than \$150K. Its right subtree will have age at least 60 and salary less than \$150K. Figure 5.14 suggests how the various interior nodes split the space of points into leaf blocks. For example, the horizontal line at salary = 150 represents the split at the root. The space below that line is split vertically at age 60, while the space above is split at age 47, corresponding to the decision at the right child of the root. □

5.3.4 Operations on kd-Trees

A lookup of a tuple given values for all dimensions proceeds as in a binary search tree. We make a decision which way to go at each interior node and are directed to a single leaf, whose block we search.

To perform an insertion, we proceed as for a lookup. We are eventually directed to a leaf, and if its block has room we put the new data point there. If there is no room, we split the block into two, and we divide its contents according to whatever attribute is appropriate at the level of the leaf being

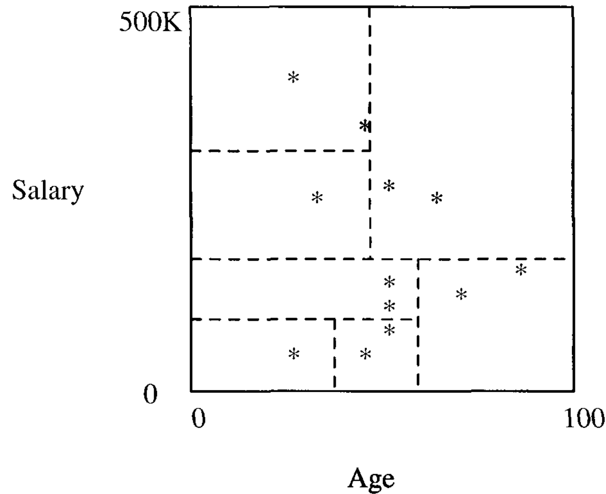

Figure 5.14: The partitions implied by the tree of Fig. 5.13

split. We create a new interior node whose children are the two new blocks, and we install at that interior node a splitting value that is appropriate for the split we have just made.[1]

Example 5.16: Suppose someone 35 years old with a salary of $500K buys gold jewelry. Starting at the root, we know the salary is at least $150K, so we go to the right. There, we compare the age 35 with the age 47 at the node, which directs us to the left. At the third level, we compare salaries again, and our salary is greater than the splitting value, $300K. We are thus directed to a leaf containing the points (25,400) and (45,350), along with the new point (35,500).

There isn't room for three records in this block, so we must split it. The fourth level splits on age, so we have to pick some age that divides the records as evenly as possible. The median value, 35, is a good choice, so we replace the leaf by an interior node that splits on age = 35. To the left of this interior node is a leaf block with only the record (25,400), while to the right is a leaf block with the other two records, as shown in Fig. 5.15. □

The more complex queries discussed in this chapter are also supported by a *kd*-tree. Here are the key ideas and synopses of the algorithms:

Partial-Match Queries

If we are given values for some of the attributes, then we can go one way when we are at a level belonging to an attribute whose value we know. When we don't

[1] One problem that might arise is a situation where there are so many points with the same value in a given dimension that the bucket has only one value in that dimension and cannot be split. We can try splitting along another dimension, or we can use an overflow block.

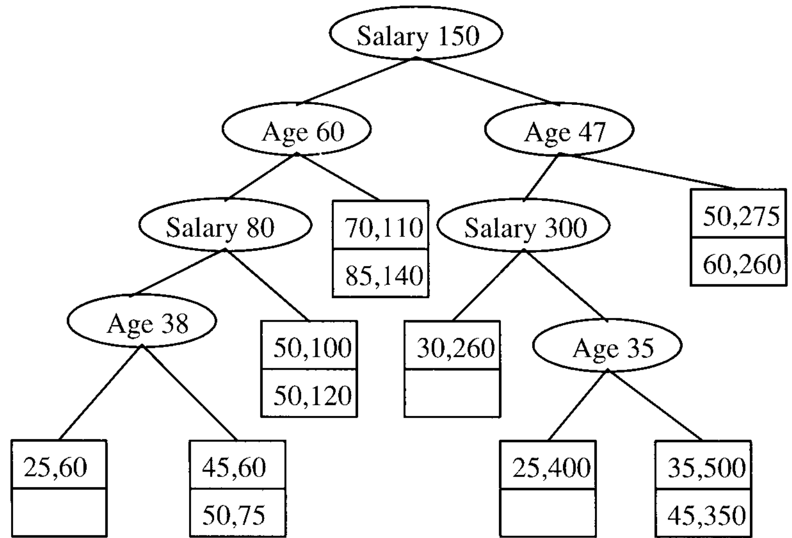

Figure 5.15: Tree after insertion of $(35, 500)$

know the value of the attribute at a node, we must explore both of its children. For example, if we ask for all points with age = 50 in the tree of Fig. 5.13, we must look at both children of the root, since the root splits on salary. However, at the left child of the root, we need go only to the left, and at the right child of the root we need only explore its right subtree. Suppose, for instance, that the tree were perfectly balanced, had a large number of levels, and had two dimensions, of which one was specified in the search. Then we would have to explore both ways at every other level, ultimately reaching about the square root of the total number of leaves.

Range Queries

Sometimes, a range will allow us to move to only one child of a node, but if the range straddles the splitting value at the node then we must explore both children. For example, given the range of ages 35 to 55 and the range of salaries from \$100K to \$200K, we would explore the tree of Fig. 5.13 as follows. The salary range straddles the \$150K at the root, so we must explore both children. At the left child, the range is entirely to the left, so we move to the node with salary \$80K. Now, the range is entirely to the right, so we reach the leaf with records $(50, 100)$ and $(50, 120)$, both of which meet the range query. Returning to the right child of the root, the splitting value age = 47 tells us to look at both subtrees. At the node with salary \$300K, we can go only to the left, finding the point $(30, 260)$, which is actually outside the range. At the right child of the node for age = 47, we find two other points, both of which are outside the range.

Nothing Lasts Forever

Each of the data structures discussed in this chapter allow insertions and deletions that make local decisions about how to reorganize the structure. After many database updates, the effects of these local decisions may make the structure unbalanced in some way. For instance, a grid file may have too many empty buckets, or a kd-tree may be greatly unbalanced.

It is quite usual for any database to be restructured after a while. By reloading the database, we have the opportunity to create index structures that, at least for the moment, as as balanced and efficient as is possible for that type of index. The cost of such restructuring can be amortized over the large number of updates that led to the imbalance, so the cost per update is small. However, we do need to be able to "take the database down"; i.e., make it unavailable for the time it is being reloaded. That situation may or may not be a problem, depending on the application. For instance, many databases are taken down overnight, when no one is accessing them.

Nearest-Neighbor Queries

Use the same approach as was discussed in Section 5.3.2. Treat the problem as a range query with the appropriate range and repeat with a larger range if necessary.

5.3.5 Adapting kd-Trees to Secondary Storage

Suppose we store a file in a kd-tree with n leaves. Then the average length of a path from the root to a leaf will be about $\log_2 n$, as for any binary tree. If we store each node in a block, then as we traverse a path we must do one disk I/O per node. For example, if $n = 1000$, then we shall need about 10 disk I/O's, much more than the 2 or 3 disk I/O's that would be typical for a B-tree, even on a much larger file. In addition, since interior nodes of a kd-tree have relatively little information, most of the block would be wasted space.

We cannot solve the twin problems of long paths and unused space completely. However, here are two approaches that will make some improvement in performance.

Multiway Branches at Interior Nodes

Interior nodes of a kd-tree could look more like B-tree nodes, with many key-pointer pairs. If we had n keys at a node, we could split values of an attribute a into $n + 1$ ranges. If there were $n + 1$ pointers, we could follow the appropriate one to a subtree that contained only points with attribute a in that range

Problems enter when we try to reorganize nodes, in order to keep distribution and balance as we do for a B-tree. For example, suppose we have a node that splits on age, and we need to merge two of its children, each of which splits on salary. We cannot simply make one node with all the salary ranges of the two children, because these ranges will typically overlap. Notice how much easier it would be if (as in a B-tree) the two children both further refined the range of ages.

Group Interior Nodes Into Blocks

We may, instead, retain the idea that tree nodes have only two children. We could pack many interior nodes into a single block. In order to minimize the number of blocks that we must read from disk while traveling down one path, we are best off including in one block a node and all its descendants for some number of levels. That way, once we retrieve the block with this node, we are sure to use some additional nodes on the same block, saving disk I/O's. For instance, suppose we can pack three interior nodes into one block. Then in the tree of Fig. 5.13, we would pack the root and its two children into one block. We could then pack the node for salary = 80 and its left child into another block, and we are left with the node salary = 300, which belongs on a separate block; perhaps it could share a block with the latter two nodes, although sharing requires us to do considerable work when the tree grows or shrinks. Thus, if we wanted to look up the record $(25,60)$, we would need to traverse only two blocks, even though we travel through four interior nodes.

5.3.6 Quad Trees

In a *quad tree,* each interior node corresponds to a square region in two dimensions, or to a k-dimensional cube in k dimensions. As with the other data structures in this chapter, we shall consider primarily the two-dimensional case. If the number of points in a square is no larger than what will fit in a block, then we can think of this square as a leaf of the tree, and it is represented by the block that holds its points. If there are too many points to fit in one block, then we treat the square as an interior node, with children corresponding to its four quadrants.

Example **5.17:** Figure 5.16 shows the gold-jewelry data points organized into regions that correspond to nodes of a quad tree. For case of calculation, we have restricted the usual space so salary ranges between 0 and \$400K, rather than up to \$500K as in other examples of this chapter. We continue to make the assumption that only two records can fit in a block.

Figure 5.17 shows the tree explicitly. We use the compass designations for the quadrants and for the children of a node (e.g., SW stands for the southwest quadrant — the points to the left and below the center). The order of children is always as indicated at the root. Each interior node indicates the coordinates of the center of its region.

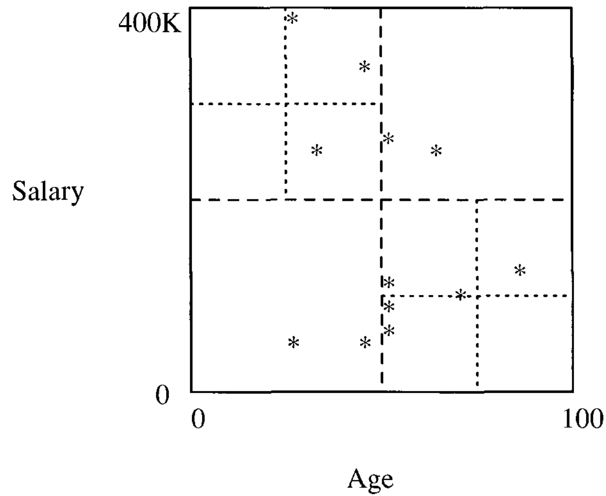

Figure 5.16: Data organized in a quad tree

Since the entire space has 12 points, and only two will fit in one block, we must split the space into quadrants, which we show by the dashed line in Fig. 5.16. Two of the resulting quadrants — the southwest and northeast — have only two points. They can be represented *by* leaves and need not be split further.

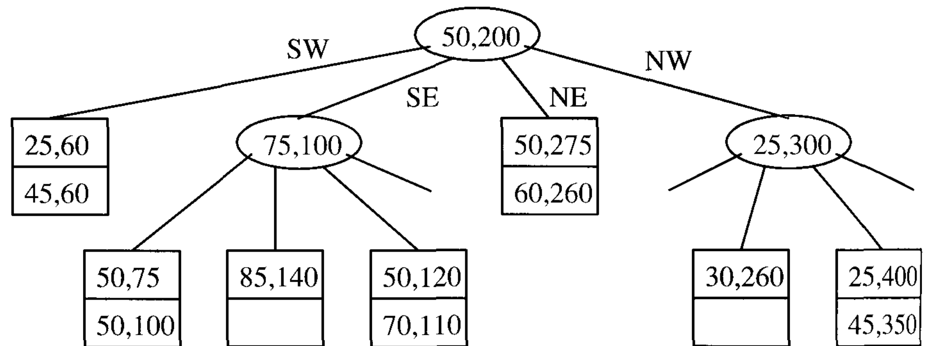

Figure 5.17: A quad tree

The remaining two quadrants each have more than two points. Both are split into subquadrants, as suggested by the dotted lines in Fig. 5.16. Each of the resulting quadrants has two or fewer points, so no more splitting is necessary. □

Since interior nodes of a quad tree in k dimensions have 2^k children, there is a range of k where nodes fit conveniently into blocks. For instance, if 128, or

2^7, pointers can fit in a block, then $k = 7$ is a convenient number of dimensions. However, for the 2-dimensional case, the situation is not much better than for kd-trees; an interior node has four children. Moreover, while we can choose the splitting point for a kd-tree node, we are constrained to pick the center of a quad-tree region, which may or may not divide the points in that region evenly. Especially when the number of dimensions is large, we expect to find many null pointers (corresponding to empty quadrants) in interior nodes. Of course we can be somewhat clever about how high-dimension nodes are represented, and keep only the non-null pointers and a designation of which quadrant the pointer represents, thus saving considerable space.

We shall not go into detail regarding the standard operations that we discussed in Section 5.3.4 for kd-trees. The algorithms for quad trees resemble those for kd-trees.

5.3.7 R-Trees

An *R-tree* (region tree) is a data structure that captures some of the spirit of a B-tree for multidimensional data. Recall that a B-tree node has a set of keys that divide a line into segments. Points along that line belong to only one segment, as suggested by Fig. 5.18. The B-tree thus makes it easy for us to find points; if we think the point is somewhere along the line represented by a B-tree node, we can determine a unique child of that node where the point could be found.

Figure 5.18: A B-tree node divides keys along a line into disjoint segments

An R-tree, on the other hand, represents data that consists of 2-dimensional, or higher-dimensional regions, which we call *data regions.* An interior node of an R-tree corresponds to some *interior region,* or just "region," which is not normally a data region. In principle, the region can be of any shape, although in practice it is usually a rectangle or other simple shape. The R-tree node has, in place of keys, subregions that represent the contents of its children. Figure 5.19 suggests a node of an R-tree that is associated with the large solid rectangle. The dotted rectangles represent the subregions associated with four of its children. Notice that the subregions do not cover the entire region, which is satisfactory as long as all the data regions that lie within the large region are wholly contained within one of the small regions. Further, the subregions are allowed to overlap, although it is desirable to keep the overlap small.

5.3.8 Operations on R-trees

A typical query for which an R-tree is useful is a "where-am-I" query, which specifies a point P and asks for the data region or regions in which the point lies.

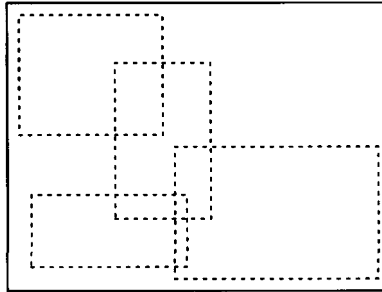

Figure 5 19: The region of an R-tree node and subregions of its children

We start at the root, with which the entire region is associated. We examine the subregions at the root and determine which children of the root correspond to interior regions that contain point *P*. Note that there may be zero, one, or several such regions.

If there are zero regions, then we are done; *P* is not in any data region. If there is at least one interior region that contains *P*, then we must recursively search for *P* at the child corresponding to *each* such region. When we reach one or more leaves, we shall find the actual data regions, along with either the complete record for each data region or a pointer to that record.

When we insert a new region *R* into an R-tree, we start at the root and try to find a subregion into which *R* fits. If there is more than one such region, then we pick one, go to its corresponding child, and repeat the process there. If there is no subregion that contains *R*, then we have to expand one of the subregions. Which one to pick may be a difficult decision. Intuitively, we want to expand regions as little as possible, so we might ask which of the children's subregions would have their area increased as little as possible, change the boundary of that region to include *R*, and recursively insert *R* at the corresponding child.

Eventually, we reach a leaf, where we insert the region *R*. However, if there is no room for *R* at that leaf, then we must split the leaf. How we split the leaf is subject to some choice. We generally want the two subregions to be as small as possible, yet they must, between them, cover all the data regions of the original leaf. Having split the leaf, we replace the region and pointer for the original leaf at the node above by a pair of regions and pointers corresponding to the two new leaves. If there is room at the parent, we are done. Otherwise, as in a B-tree, we recursively split nodes going up the tree.

Example 5.18: Let us consider the addition of a new region to the map of Fig. 5 1 Suppose that leaves have room for six regions. Further suppose that the six regions of Fig. 5.1 are together on one leaf, whose region is represented by the outer (solid) rectangle in Fig. 5.20.

Now, suppose the local cellular phone company adds a POP (point of presence, or antenna) at the position shown in Fig. 5.20. Since the seven data regions do not fit on one leaf, we shall split the leaf, with four in one leaf and

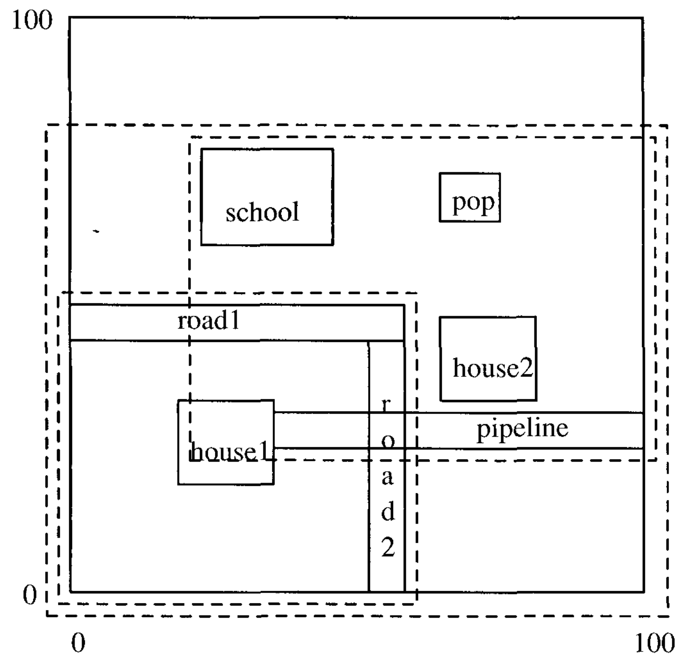

Figure 5.20: Splitting the set of objects

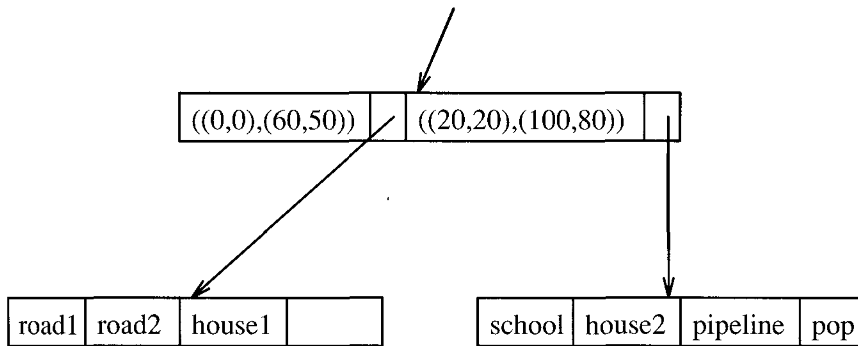

Figure 5.21: An R-tree

three in the other. Our options are many; we have picked in Fig. 5.20 the division (indicated by the inner, dashed rectangles) that minimizes the overlap, while splitting the leaves as evenly as possible.

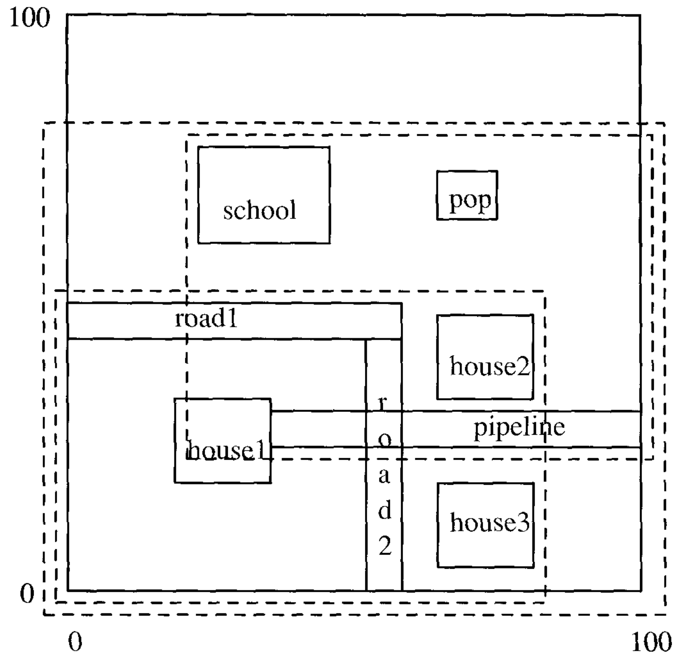

Figure 5.22: Extending a region to accommodate new data

We show in Fig. 5.21 how the two new leaves fit into the R-tree. The parent of these nodes has pointers to both leaves, and associated with the pointers are the lower-left and upper-right corners of the rectangular regions covered by each leaf. □

Example 5.19 : Suppose we inserted another house below house2, with lower-left coordinates $(70,5)$ and upper-right coordinates $(80,15)$. Since this house is not wholly contained within either of the leaves' regions, we must choose which region to expand. If we expand the lower subregion, corresponding to the first leaf in Fig. 5.21, then we add 1000 square units to the region, since we extend it 20 units to the right. If we extend the other subregion by lowering its bottom by 15 units, then we add 1200 square units. We prefer the first, and the new regions are changed in Fig. 5.22. We also must change the description of the region in the top node of Fig. 5.21 from $((0,0),\ (60,50))$ to $((0,0),\ (80,50))$. a

5.3.9 Exercises for Section 5.3

Exercise 5.3.1: Show a multilevel index for the data of Fig. 5.10 if the indexes

are on:

a) Speed, then ram.

b) Ram then hard-disk.

c) Speed, then ram, then hard-disk.

Exercise 5.3.2 : Place the data of Fig. 5.10 in a *kd*-tree. Assume two records can fit in one block. At each level, pick a separating value that divides the data as evenly as possible. For an order of the splitting attributes choose:

a) Speed, then ram, alternating.

b) Speed, then ram, then hard-disk, alternating.

c) Whatever attribute produces the most even split at each node.

Exercise 5.3.3: Suppose we have a relation $R(x, y, z)$. where the pair of attributes x and y together form the key. Attribute x ranges from 1 to 100, and y ranges from 1 to 1000. For each x there are records with 100 different values of y, and for each y there are records with 10 different values of x. Note that there are thus 10,000 records in *R*. We wish to use a multiple-key index that will help us to answer queries of the form

```
SELECT z
FROM R
WHERE x = C AND y = D;
```

where *C* and *D* are constants. Assume that blocks can hold ten key-pointer pairs, and we wish to create dense indexes at each level, perhaps with sparse higher-level indexes above them, so that each index starts from a single block. Also assume that initially all index and data blocks are on disk

* a) How many disk I/O's are necessary to answer a query of the above form if the first index is on x?

b) How many disk I/O's are necessary to answer a query of the above form if the first index is on y?

! c) Suppose you were allowed to buffer 11 blocks in memory at all times. Which blocks would you choose, and would you make x or y the first index, if you wanted to minimize the number of additional disk I/O's needed?

Exercise 5.3.4: For the structure of Exercise 5.3.3(a), how many disk I/O's are required to answer the range query in which $20 < x < 35$ and $200 < y < 350$. Assume data is distributed uniformly; i.e., the expected number of points will be found within any given range.

Exercise 5.3.5: In the tree of Fig. 5.13, what new points would be directed to:

* a) The block with point (30, 260)?

b) The block with points (50, 100) and (50, 120)?

Exercise 5.3.6 : Show a possible evolution of the tree of Fig. 5.15 if we insert the points (20, 110) and then (40, 400).

! **Exercise 5.3.7:** We mentioned that if a *kd*-tree were perfectly balanced, and we execute a partial-match query in which one of two attributes has a value specified, then we wind up looking at about $\sqrt{n}$ out of the n leaves.

a) Explain why.

b) If the tree split alternately in d dimensions, and we specified values for m of those dimensions, what fraction of the leaves would we expect to have to search?

c) How does the performance of (b) compare with a partitioned hash table?

Exercise 5.3.8: Place the data of Fig. 5.10 in a quad tree with dimensions speed and ram. Assume the range for speed is 100 to 500, and for ram it is 0 to 256.

Exercise 5.3.9 : Repeat Exercise 5.3.8 with the addition of a third dimension, hard-disk, that ranges from 0 to 32.

*! **Exercise 5.3.10** : If we are allowed to put the central point in a quadrant of a quad tree wherever we want, can we always divide a quadrant into subquadrants with an equal number of points (or as equal as possible, if the number of points in the quadrant is not divisible by 4)? Justify your answer.

! **Exercise 5.3.11:** Suppose we have a database of 1,000,000 regions, which may overlap. Nodes (blocks) of an R-tree can hold 100 regions and pointers. The region represented by any node has 100 subregions, and the overlap among these regions is such that the total area of the 100 subregions is 150% of the area of the region. If we perform a "where-am-I" query for a given point, how many blocks do we expect to retrieve?

! **Exercise 5.3.12:** In the R-tree represented by Fig. 5.22, a new region might go into the subregion containing the school or the subregion containing house3 Describe the rectangular regions for which we would prefer to place the new region in the subregion with the school (i.e., that choice minimizes the increase in the subregion size).

5.4 Bitmap Indexes

Let us now turn to a type of index that is rather different from the kinds seen so far. We begin by imagining that records of a file have permanent numbers, $1, 2, \ldots, n$. Moreover, there is some data structure for the file that lets us find the ith record easily for any i.

A *bitmap index* for a field F is a collection of bit-vectors of length n, one for each possible value that may appear in the field F. The vector for value v has 1 in position i if the ith record has v in field F, and it has 0 there if not.

Example 5.20 : Suppose a file consists of records with two fields, F and G, of type integer and string, respectively. The current file has six records, numbered 1 through 6, with the following values in order: (30, foo), (30, bar), (40, baz), (50, foo), (40, bar), (30, baz).

A bitmap index for the first field, F, would have three bit-vectors, each of length 6. The first, for value 30, is 110001, because the first, second, and sixth records have $F = 30$. The other two, for 40 and 50, respectively, are 001010 and 000100.

A bitmap index for G would also have three bit-vectors, because there are three different strings appearing there. The three bit-vectors are:

Value	Vector
foo	100100
bar	010010
baz	001001

In each case, the 1's indicate in which records the corresponding string appears.
D

5.4.1 Motivation for Bitmap Indexes

It might at first appear that bitmap indexes require much too much space, especially when there are many different values for a field, since the total number of bits is the product of the number of records and the number of values. For example, if the field is a key, and there are n records, then n^2 bits are used among all the bit-vectors for that field. However, compression can be used to make the number of bits closer to n, independent of the number of different values, as we shall see in Section 5.4.2.

You might also suspect that there are problems managing the bitmap indexes. For example, they depend on the number of a record remaining the same throughout time. How do we find the ith record as the file adds and deletes records? Similarly, values for a field may appear or disappear. How do we find the bitmap for a value efficiently? These and related questions are discussed in Section 5.4.4.

The compensating advantage of bitmap indexes is that they allow us to answer partial-match queries very efficiently in many situations. In a sense they

offer the advantages of buckets that we discussed in Example 4.16, where we found the Movie tuples with specified values in several attributes without first retrieving all the records that matched in each of the attributes. An example will illustrate the point.

Example 5.21: Recall Example 4.16, where we queried the relation

```
Movie(title, year, length, studioName)
```

with the query

```
SELECT title
FROM Movie
WHERE studioName = 'Disney' AND
      year = 1995;
```

Suppose there are bitmap indexes on both attributes studioName and year. Then we can intersect the vectors for year = 1995 and studioName = 'Disney'; that is, we take the bitwise AND of these vectors, which will give us a vector with a 1 in position i if and only if the ith Movie tuple is for a movie made by Disney in 1995.

If we can retrieve tuples of Movie given their numbers, then we need to read only those blocks containing one or more of these tuples, just as we did in Example 4.16. To intersect the bit vectors, we must read them into memory, which requires a disk I/O for each block occupied by one of the two vectors. As mentioned, we shall later address both matters: accessing records given their numbers in Section 5.4.4 and making sure the bit-vectors do not occupy too much space in Section 5.4.2. □

Bitmap indexes can also help answer range queries. We shall consider an example next that both illustrates their use for range queries and shows in detail with short bit-vectors how the bitwise AND and OR of bit-vectors can be used to discover the answer to a query without looking at any records but the ones we want.

Example 5.22: Consider the gold jewelry data first introduced in Example 5.7. Suppose that the twelve points of that example are records numbered from 1 to 12 as follows:

1:	(25,60)	2:	(45,60)	3:	(50,75)	4:	(50,100)
5:	(50,120)	6:	(70,110)	7:	(85,140)	8:	(30,260)
9:	(25,400)	10:	(45,350)	11:	(50,275)	12:	(60,260)

For the first component, age, there are seven different values, so the bitmap index for age consists of the following seven vectors:

25:	100000001000	30:	000000010000	45:	010000000100
50:	001110000010	60:	000000000001	70:	000001000000
85:	000000100000				

For the salary component, there are ten different values, so the salary bitmap index has the following ten bit-vectors:

```
 60: 110000000000   75: 001000000000  100: 000100000000
110: 000001000000  120: 000010000000  140: 000000100000
260: 000000010001  275: 000000000010  350: 000000000100
400: 000000001000
```

Suppose we want to find the jewelry buyers with an age in the range 45-55 and a salary in the range 100-200. We first find the bit-vectors for the age values in this range; in this example there are only two: 010000000100 and 001110000010, for 45 and 50, respectively. If we take their bitwise OR, we have a new bit-vector with 1 in position i if and only if the ith record has an age in the desired range. This bit-vector is 011110000110.

Next, we find the bit-vectors for the salaries between 100 and 200 thousand. There are four, corresponding to salaries 100, 110, 120, and 140; their bitwise OR is 000111100000.

The last step is to take the bitwise AND of the two bit-vectors we calculated by OR. That is:

$$011110000110 \text{ AND } 000111100000 = 000110000000$$

We thus find that only the fourth and fifth records, which are $(50, 100)$ and $(50, 120)$, are in the desired range. □

5.4.2 Compressed Bitmaps

Suppose we have a bitmap index on field F of a file with n records, and there are m different values for field F that appear in the file. Then the number of bits in all the bit-vectors for this index is mn. If, say, blocks are 4096 bytes long, then we can fit 32,768 bits in one block, so the number of blocks needed is $mn/32768$. That number can be small compared to the number of blocks needed to hold the file itself, but the larger m is, the more space the bitmap index takes.

But if m is large, then 1's in a bit-vector will be very rare; precisely, the probability that any bit is 1 is $1/m$. If 1's are rare, then we have an opportunity to encode bit-vectors so that they take much fewer than n bits on the average. A common approach is called *run-length encoding*, where we represent a *run*, that is, a sequence of i 0's followed by a 1, by some suitable binary encoding of the integer i. We concatenate the codes for each run together, and that sequence of bits is the encoding of the entire bit-vector.

We might imagine that we could just represent integer i by expressing i as a binary number. However, that simple a scheme will not do, because it is not possible to break a sequence of codes apart to determine uniquely the lengths of the runs involved (see the box on "Binary Numbers Won't Serve as a Run-Length Encoding"). Thus, the encoding of integers i that represent a run length must be more complex than a simple binary representation.

Binary Numbers Won't Serve as a Run-Length Encoding

Suppose we represented a run of i 0's followed by a 1 with the integer i in binary. Then the bit-vector 000101 consists of two runs, of lengths 3 and 1, respectively. The binary representations of these integers are 11 and 1, so the run-length encoding of 000101 is 111. However, a similar calculation shows that the bit-vector 010001 is also encoded by 111; bit-vector 010101 is a third vector encoded by 111. Thus, 111 cannot be decoded uniquely into one bit-vector.

We shall use one of many possible schemes for encoding. There are some better, more complex schemes that can improve on the amount of compression achieved here, by almost a factor of 2, but only when typical runs are very long. In our scheme, we first need to determine how many bits the binary representation of i has. This number j, which is approximately $\log_2 i$, is represented in "unary," by $j - 1$ 1's and a single 0. Then, we can follow with i in binary.[2]

Example 5.23: If $i = 13$, then $j = 4$; that is, we need 4 bits in the binary representation of i. Thus, the encoding for i begins with 1110. We follow with i in binary, or 1101. Thus, the encoding for 13 is 11101101.

The encoding for $i = 1$ is 01, and the encoding for $i = 0$ is 00. In each case, $j = 1$, so we begin with a single 0 and follow that 0 with the one bit that represents i. □

If we concatenate a sequence of integer codes, we can always recover the sequence of run lengths and therefore recover the original bit-vector. Suppose we have scanned some of the encoded bits, and we are now at the beginning of the sequence of bits that encodes some integer i. We scan forward to the first 0, to determine the value of j. That is, j equals the number of bits we must scan until we get to the first 0 (including that 0 in the count of bits). Once we know j, we look at the next j bits; i is the integer represented there in binary. Moreover, once we have scanned the bits representing i, we know where the next code for an integer begins, so we can repeat the process.

Example 5.24: Let us decode the sequence 11101101001011. Starting at the beginning, we find the first 0 at the 4th bit, so $j = 4$. The next 4 bits are 1101, so we determine that the first integer is 13. We are now left with 001011 to decode.

[2]Actually, except for the case that $j = 1$ (i.e., $i = 0$ or $i = 1$), we can be sure that the binary representation of i begins with 1. Thus, we can save about one bit per number if we omit this 1 and use only the remaining $j - 1$ bits.

Since the first bit is 0, we know the next bit represents the next integer by itself; this integer is 0. Thus, we have decoded the sequence 13, 0, and we must decode the remaining sequence 1011.

We find the first 0 in the second position, whereupon we conclude that the final two bits represent the last integer, 3. Our entire sequence of run-lengths is thus 13, 0, 3. From these numbers, we can reconstruct the actual bit-vector, 0000000000000110001. □

Technically, every bit-vector so decoded will end in a 1, and any trailing O's will not be recovered. Since we presumably know the number of records in the file, the additional O's can be added. However, since 0 in a bit-vector indicates the corresponding record is not in the described set, we don't even have to know the total number of records, and can ignore the trailing O's.

Example 5.25: Let us convert some of the bit-vectors from Example 5.23 to our run-length code. The vectors for the first three ages, 25, 30, and 45, are 100000001000, 000000010000, and 010000000100, respectively. The first of these has the run-length sequence (0, 7). The code for 0 is 00, and the code for 7 is 110111. Thus, the bit-vector for age 25 becomes 00110111.

Similarly, the bit-vector for age 30 has only one run, with seven O's. Thus, its code is 110111. The bit-vector for age 45 has two runs, $(1,7)$. Since 1 has the code 01, and we determined that 7 has the code 110111, the code for the third bit-vector is 01110111. □

The compression in Example 5.25 is not great. However, we cannot see the true benefits when n, the number of records, is small. To appreciate the value of the encoding, suppose that $m = n$, i.e., each value for the field on which the bitmap index is constructed, has a unique value. Notice that the code for a run of length i has about $2\log_2 i$ bits. If each bit-vector has a single 1, then it has a single run, and the length of that run cannot be longer than n. Thus, $2\log_2 n$ bits is an upper bound on the length of a bit-vector's code in this case.

Since there are n bit-vectors in the index (because $m = n$), the total number of bits to represent the index is at most $2n\log_2 n$. Notice that without the encoding, n^2 bits would be required. As long as $n > 4$, we have $2n\log_2 n < n^2$, and as n grows, $2n\log_2 n$ becomes arbitrarily smaller than n^2.

5.4.3 Operating on Run-Length-Encoded Bit-Vectors

When we need to perform bitwise AND or OR on encoded bit-vectors, we have little choice but to decode them and operate on the original bit-vectors. However, we do not have to do the decoding all at once. The compression scheme we have described lets us decode one run at a time, and we can thus determine where the next 1 is in each operand bit-vector. If we are taking the OR, we can produce a 1 at that position of the output, and if we are taking the AND we produce a 1 if and only if both operands have their next 1 at the same position. The algorithms involved are complex, but an example may make the idea adequately clear.

Example 5.26: Consider the encoded bit-vectors we obtained in Example 5.25 for ages 25 and 30: 00110111 and 110111, respectively. We can decode their first runs easily; we find they are 0 and 7, respectively. That is, the first 1 of the bit-vector for 25 occurs in position 1, while the first 1 in the bit-vector for 30 occurs at position 8. We therefore generate 1 in position 1.

Next, we must decode the next run for age 25, since that bit-vector may produce another 1 before age 30's bit-vector produces a 1 at position 8. However, the next run for age 25 is 7, which says that this bit-vector next produces a 1 at position 9. We therefore generate six O's and the 1 at position 8 that comes from the bit-vector for age 30. Now, that bit-vector contributes no more 1's to the output. The 1 at position 9 from age 25's bit-vector is produced, and that bit-vector too produces no subsequent 1's.

We conclude that the OR of these bit-vectors is 100000011. Referring to the original bit-vectors of length 12, we see that is almost right; there are three trailing O's omitted. If we know that the number of records in the file is 12, we can append those O's. However, it doesn't matter whether or not we append the O's, since only a 1 can cause a record to be retrieved. In this example, we shall not retrieve any of records 10 through 12 anyway. □

5.4.4 Managing Bitmap Indexes

We have described operations on bitmap indexes without addressing three important issues:

1. When we want to find the bit-vector for a given value, or the bit-vectors corresponding to values in a given range, how do we find these efficiently?

2. When we have selected a set of records that answer our query, how do we retrieve those records efficiently?

3. When the data file changes by insertion or deletion of records, how do we adjust the bitmap index on a given field?

Finding Bit-Vectors

The first question can be answered based on techniques we have already learned. Think of each bit-vector as a record whose key is the value corresponding to this bit-vector (although the value itself does not appear in this "record"). Then any secondary index technique will take us efficiently from values to their bit-vectors. For example, we could use a B-tree, whose leaves contain key-pointer pairs; the pointer leads to the bit-vector for the key value. The B-tree is often a good choice, because it supports range queries easily, but hash tables or indexed-sequential files are other options.

We also need to store the bit-vectors somewhere. It is best to think of them as variable-length records, since they will generally grow as more records are added to the data file. If the bit-vectors, perhaps in compressed form, are

typically shorter than blocks. then we can consider packing several to a block and moving them around as needed. If bit-vectors arc typically longer than a block, we should consider using a chain of blocks to hold each one. The techniques of Section 3.4 are useful.

Finding Records

Now let us consider the second question: once we have determined that we need record k of the data file, how do we find it. Again, techniques we have already seen may be adapted. Think of the kth record as having search-key value k (although this key does not actually appear in the record). We may then create a secondary index on the data file, whose search key is the number of the record.

If there is no reason to organize the file any other way, we can even use the record number as the search key for a primary index, as discussed in Section 4.1. Then, the file organization is particularly simple, since record numbers never change (even as records are deleted), and we only have to add new records to the end of the data file. It is thus possible to pack blocks of the data file completely full, instead of leaving extra space for insertions into the middle of the file as we found necessary for the general case of an indexed-sequential file in Section 4.1.6.

Handling Modifications to the Data File

There are two aspects to the problem of reflecting data-file modifications in a bitmap index.

1. Record numbers must remain fixed once assigned.

2. Changes to the data file require the bitmap index to change as well.

The consequence of point (1) is that when we delete record i, it is easiest to "retire" its number. Its space is replaced by a "tombstone" in the data file. The bitmap index must also be changed, since the bit-vector that had a 1 in position % must have that 1 changed to 0. Note that we can find the appropriate bit-vector, since we know what value record i had before deletion.

Next consider insertion of a new record. We keep track of the next available record number and assign it to the new record. Then, for each bitmap index, we must determine the value the new record has in the corresponding field and modify the bit-vector for that value by appending a 1 at the end. Technically, all the other bit-vectors in this index get a new 0 at the end, but if we are using a compression technique such as that of Section 5.4.2, then no change to the compressed values is needed.

As a special case, the new record may have a value for the indexed field that has not been seen before. In that case, we need a new bit-vector for this value, and this bit-vector and its corresponding value need to be inserted

into the secondary-index structure that is used to find a bit-vector given its corresponding value.

Last, let us consider a modification to a record i of the data file that changes the value of a field that has a bitmap index, say from value v to value w. We must find the bit-vector for v and change the 1 in position i to 0. If there is a bit-vector for value w, then we change its 0 in position i to 1. If there is not yet a bit-vector for w, then we create it as discussed in the paragraph above for the case when an insertion introduces a new value.

5.4.5 Exercises for Section 5.4

Exercise 5.4.1: For the data of Fig. 5.10 show the bitmap indexes for the attributes:

* a) Speed,

b) Ram, and

c) Hard-disk,

both in *(i)* uncompressed form, and (M) compressed form using the scheme of Section 5.4.2.

Exercise 5.4.2: Using the bitmaps of Example 5.22, find the jewelry buyers with an age in the range 20-40 and a salary in the range 0-100.

Exercise 5.4.3: Consider a file of 1,000,000 records, with a field F that has m different values.

a) As a function of m, how many bytes does the bitmap index for F have?

! b) Suppose that the records numbered from 1 to 1,000,000 are given values for the field F in a round-robin fashion, so each value appears every m records. How many bytes would be consumed by a compressed index?

!! **Exercise 5.4.4:** We suggested in Section 5.4.2 that it was possible to reduce the number of bits taken to encode number i from the $2\log_2 i$ that we used in that section until it is close to $\log_2 i$. Show how to approach that limit as closely as you like, *as* long as i is large. *Hint:* We used a unary encoding of the length of the binary encoding that we used for i. Can you encode the length of the code in binary?

Exercise 5.4.5: Encode, using the scheme of Section 5.4.2, the following bitmaps:

* a) 0110000000100000100.

b) 10000010000001001101.

r) 000100000000000010000010000.

*! **Exercise 5.4.6:** We pointed out that compressed bitmap indexes consume about $2n \log_2 n$ bits for a file of n records. How does this number of bits compare with the number of bits consumed by a B-tree index? Remember that the B-tree index's size depends on the size of keys and pointers, as well as (to a small extent) on the size of blocks. However, make some reasonable estimates of these parameters in your calculations. Why might we prefer a B-tree, even if it takes more space than compressed bitmaps?

5.5 Summary of Chapter 5

- *Multidimensional Data:* Many applications, such as geographic databases or sales and inventory data, can be thought of as points in a space of two or more dimensions.

- *Queries Needing Multidimensional Indexes:* The sorts of queries that need to be supported on multidimensional data include partial-match (all points with specified values in a subset of the dimensions), range queries (all points within a range in each dimension), nearest-neighbor (closest point to a given point), and where-am-i (region or regions containing a given point).

- *Executing Nearest-Neighbor Queries*: Many data structures allow nearest-neighbor queries to be executed by performing a range query around the target point, and expanding the range if there is no point in that range. We must be careful, because finding a point within a rectangular range may not rule out the possibility of a closer point outside that rectangle.

- *Grid Files:* The grid file slices the space of points in each of the dimensions. The grid lines can be spaced differently, and there can be different numbers of lines for each dimension. Grid files support range queries, partial-match queries, and nearest-neighbor queries well. as long as data is fairly uniform in distribution.

- *Partitioned Hash Tables:* A partitioned hash function constructs some bits of the bucket number from each dimension. They support partial-match queries well, and are not dependent on the data being uniformly distributed.

- *Multiple-Key Indexes:* A simple multidimensional structure has a root that is an index on one attribute, leading to a collection of indexes on a second attribute, which can lead to indexes on a third attribute, and so on. They are useful for range and nearest-neighbor queries.

- *kd-Trees*: These trees are like binary search trees, but they branch on different attributes at different levels. They support partial-match, range,

and nearest-neighbor queries well. Some careful packing of tree nodes into blocks must be done to make the structure suitable for secondary-storage operations.

4 *Quad Trees:* The quad tree divides a multidimensional cube into quadrants, and recursively divides the quadrants the same way if they have too many points. They support partial-match, range, and nearest-neighbor queries.

4 *R-Trees*: This form of tree normally represents a collection of regions by grouping them into a hierarchy of larger regions. It helps with where-am-i queries and, if the atomic regions are actually points, will support the other types of queries studied in this chapter, as well.

4 *Bitmap Indexes:* Multidimensional queries are supported by a form of index that orders the points or records and represents the positions of the records with a given value in an attribute by a bit vector. These indexes support range, nearest-neighbor, and partial-match queries.

4 *Compressed Bitmaps:* In order to save space, the bitmap indexes, which tend to consist of vectors with very few 1's, are compressed by using a run-length encoding.

5.6 References for Chapter 5

Most of the data structures discussed in this section were the product of research in the 1970's or early 1980's. The *kd*-tree is from [2]. Modifications suitable for secondary storage appeared in [3] and [13]. Partitioned hashing and its use in partial-match retieval is from [12] and [5]. However, the design idea from Exercise 5.2.8 is from [14].

Grid files first appeared in [9]. The quad tree is in [6]. The R-tree is from [8], and two extensions [15] and [1] are well known.

The bitmap index has an interesting history. There was a company called Nucleus, founded by Ted Glaser, that patented the idea and developed a DBMS in which the bitmap index was both the index structure and the data representation. The company failed in the late 1980's, but the idea has recently been incorporated into several major commercial database systems. The first published work on the subject was [10]. [11] is a recent expansion of the idea.

There are a number of surveys of multidimensional storage structures. One of the earliest is [4]. More recent surveys are found in [16] and [7]. The former also includes surveys of several other important database topics.

1. N. Beekmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: an efficient and robust access method for points and rectangles," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1990), pp. 322-331.

2. J. L. Bentley. "Multidimensional binary search trees used for associative searching," *Comm. ACM* 18:9 (1975), pp. 509-517.

3. J. L. Bentley, "Multidimensional binary search trees in database applications," *IEEE Trans. on Software Engineering* SE-5:4 (1979), pp. 333-340.

4. J. L. Bentley and J. H. Friedrnan. "Data structures for range searching," *Computing Surveys* 13:3 (1979), pp. 397-409.

5. W. A. Burkhard, "Hashing and trie algorithms for partial match retrieval," *ACM Trans. on Database Systems* 1:2 (1976), pp. 175-187.

6. R. A. Finkel and J. L. Bentley, "Quad trees, a data structure for retrieval on composite keys," *Acta Informatica* 4:1 (1974), pp. 1-9.

7. V. Gaede and 0. Gunthcr, "Multidimensional access methods," *Computing Surveys* 30:2 (1998), pp. 170-231.

8. A. Guttman, "R-trees: a dynamic index structure for spatial searching," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1984), pp. 47-57.

9. J. Nievergelt, H. Hinterberger, and K. Sevcik, "The grid file: an adaptable, symmetric, multikey file structure," *ACM Trans. on Database Systems* 9:1 (1984), pp. 38 71.

10. P. O'Neil, "Model 204 architecture and performance," *Proc. Second Intl. Workshop on High Performance Transaction Systems,* Springer-Verlag, Berlin, 1987.

11. P. O'Neil and D. Quass, "Improved query performance with variant indexes," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1997), pp. 38-49.

12. R. L. Rivest, "Partial match retrieval algorithms," *SIAM J. Computing* 5:1 (1976), pp. 19-50.

13. J. T. Robinson, "The K-D-B-tree: a search structure for large multidimensional dynamic indexes," *Proc. ACM SIGMOD Intl. Conf. on Mamagement of Data* (1981), pp. 10-18.

14. J. B. Rothnie Jr. and T. Lozano, "Attribute based file organization in a paged memory environment, *Comm. ACM* 17:2 (1974), pp. 63-69.

15. T. K. Sellis, N. Roussopoulos, and C. Faloutsos, "The R+-tree: a dynamic index for multidimensional objects," *Proc. Intl. Conf. on Very Large Databases* (1987), pp. 507-518.

16. C. Zaniolo, S. Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmanian, and R. Zicari, *Advanced Database Systems,* Morgan-Kaufmann, San Francisco, 1997.