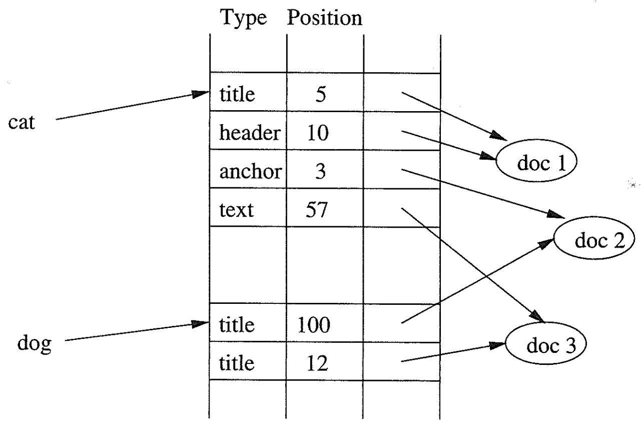

Figure 14.10: Storing more information in the inverted index

**Insertion and Deletion From Buckets**

We show buckets in figures such as Fig. 14.9 as compacted arrays of appropriate size. In practice, they are records with a single field (the pointer) and are stored in blocks like any other collection of records. Thus, when we insert or delete pointers, we may use any of the techniques seen so far, such as leaving extra space in blocks for expansion of the file, overflow blocks, and possibly moving records within or among blocks. In the latter case, we must be careful to change the pointer from the inverted index to the bucket file, as we move the records it points to.

b) Mention cats in an anchor — presumably a link to a document about cats.

We can answer this query by intersecting pointers. That is, we follow the pointer associated with "cat" to find the occurrences of this word. We select from the bucket file the pointers to documents associated with occurrences of "cat" where the type is "anchor." We then find the bucket entries for "dog" and select from them the document pointers associated with the type "title." If we intersect these two sets of pointers, we have the documents that meet the conditions: they mention "dog" in the title and "cat" in an anchor. □

### 14.1.9 Exercises for Section 14.1

**Exercise 14.1.1:** Suppose blocks hold either five records, or 20 key-pointer pairs. As a function of $n$, the number of records, how many blocks do we need to hold a data file and: (a) A dense index (b) A sparse index?

**More About Information Retrieval**

There are a number of techniques for improving the effectiveness of retrieval of documents given keywords. While a complete treatment is beyond the scope of this book, here are two useful techniques:

1. *Stemming.* We remove suffixes to find the "stem" of each word, before entering its occurrence into the index. For example, plural nouns can be treated as their singular versions. Thus, in Example 14.8, the inverted index evidently uses stemming, since the search for word "dog" got us not only documents with "dog," but also a document with the word "dogs."

2. *Stop words.* The most common words, such as "the" or "and," are called *stop words* and often are excluded from the inverted index. The reason is that the several hundred most common words appear in too many documents to make them useful as a way to find documents about specific subjects. Eliminating stop words also reduces the size of the inverted index significantly.

**Exercise 14.1.2:** Repeat Exercise 14.1.1 if blocks can hold up to 50 records or 500 key-pointer pairs, but neither data- nor index-blocks are allowed to be more than 80% full.

! **Exercise 14.1.3:** Repeat Exercise 14.1.1 if we use as many levels of index as is appropriate, until the final level of index has only one block.

! **Exercise 14.1.4:** Consider a clustered file organization like Fig. 14.6, and suppose that 5 records, either studio or movie records, will fit on one block. Also assume that the number of movies per studio is uniformly distributed between 1 and $m$. As a function of $m$, what is the average number of disk I/O's needed to retrieve a studio and all its movies? What would the number be if movies were randomly distributed over a large number of blocks?

**Exercise 14.1.5:** Suppose that blocks can hold either five records, twenty key-pointer pairs, or 100 pointers. Using the indirect-buckets scheme of Fig. 14.7:

a) If the average search-key value appears in 10 records, how many blocks do we need to hold 5000 records and its secondary index structure? How many blocks would be needed if we did *not* use buckets?

! b) If there are no constraints on the number of records that can have a given search-key value, what are the minimum and maximum number of blocks needed?

! **Exercise 14.1**
age number of
search-key valu
is in memory to
incurring addit
memory.

**Exercise 14.1**
we can perform
could be used t

a) Documen
each othe

b) Documen
tion.

c) Documen

**Exercise 14.1**
wish to build
word-pointer p
a document. T
Distribution" i
frequent word i

a) Suppose
that have
need to h

b) Repeat (a
in the inc

c) Suppose
How man

d) Repeat (c

e) What is t

## 14.2 B-T

While one or t
there is a more
This family of
is most often u

! **Exercise 14.1.6:** On the assumptions of Exercise 14.1.5(a), what is the average number of disk I/O's to find and retrieve the twelve records with a given search-key value, both with and without the bucket structure? Assume nothing is in memory to begin, but it is possible to locate index or bucket blocks without incurring additional I/O's beyond what is needed to retrieve these blocks into memory.

**Exercise 14.1.7:** If we use an augmented inverted index, such as in Fig. 14.10, we can perform a number of other kinds of searches. Suggest how this index could be used to find:

a) Documents in which "cat" and "dog" appeared within five positions of each other in the same type of element (e.g., title, text, or anchor).

b) Documents in which "dog" followed "cat" separated by exactly one position.

c) Documents in which "dog" and "cat" both appear in the title.

**Exercise 14.1.8:** Suppose we have a repository of 2000 documents, and we wish to build an inverted index with 10,000 words. A block can hold ten word-pointer pairs or 50 pointers to either a document or a position within a document. The distribution of words is Zipfian (see the box on "The Zipfian Distribution" in Section 16.4.3); the number of occurrences of the $i$th most frequent word is $100000/\sqrt{i}$, for $i = 1, 2, \ldots, 10000$.

a) Suppose our inverted index only records for each word all the documents that have that word. What is the maximum number of blocks we could need to hold the inverted index?

b) Repeat (a) if the 400 most common words ("stop" words) are *not* included in the index.

c) Suppose our inverted index holds pointers to each occurrence of each word. How many blocks do we need to hold the inverted index?

d) Repeat (c) if the 400 most common words are not included in the index.

e) What is the averge number of words per document?

## 14.2 B-Trees

While one or two levels of index are often very helpful in speeding up queries, there is a more general structure that is commonly used in commercial systems. This family of data structures is called *B-trees*, and the particular variant that is most often used is known as a *B+ tree*. In essence:

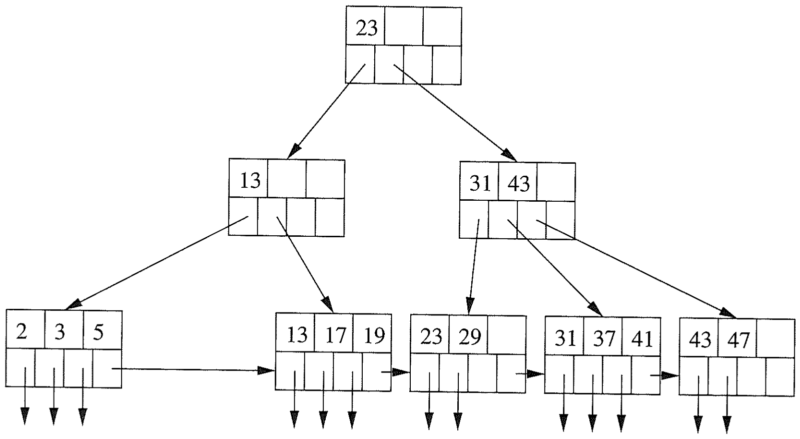

Figure 14.19: Completing the deletion of key 11

that the average block has an occupancy midway between the minimum and maximum, i.e., a typical block has 255 pointers. With a root, 255 children, and $255^2 = 65025$ leaves, we shall have among those leaves $255^3$, or about 16.6 million pointers to records. That is, files with up to 16.6 million records can be accommodated by a 3-level B-tree. □

However, we can use even fewer than three disk I/O's per search through the B-tree. The root block of a B-tree is an excellent choice to keep permanently buffered in main memory. If so, then every search through a 3-level B-tree requires only two disk reads. In fact, under some circumstances it may make sense to keep second-level nodes of the B-tree buffered in main memory as well, reducing the B-tree search to a single disk I/O, plus whatever is necessary to manipulate the blocks of the data file itself.

### 14.2.8 Exercises for Section 14.2

**Exercise 14.2.1:** Suppose that blocks can hold either ten records or 99 keys and 100 pointers. Also assume that the average B-tree node is 70% full; i.e., it will have 69 keys and 70 pointers. We can use B-trees as part of several different structures. For each structure described below, determine (*i*) the total number of blocks needed for a 100,000-record file, and (*ii*) the average number of disk I/O's to retrieve a record given its search key. You may assume nothing is in memory initially, and the search key is the primary key for the records.

a) The data file is a sequential file, sorted on the search key, with 20 records per block. The B-tree is a dense index.

b) The same as (a), but the data file consists of records in no particular order, packed 20 to a block.

S

There are B-tr
leaf has too fe
rationale is tha
occasional dele
probably soon
pairs once agai

Further, if
we need to rep
delete its point
it can be guara
the B-tree, we
record at a lea

c) The same a

d) The data fil
primary blo
primary blo
are in no pa

! e) Instead of
leaves hold
on average,
block.

**Exercise 14.2.2**
query that is ma

**Exercise 14.2.3**
long. How many

**Exercise 14.2.4**
B-tree (*i*) interio

a) $n = 11$; i.e.

b) $n = 12$; i.e

**Exercise 14.2.5**
the changes for

a) Lookup th

b) Lookup th

c) Lookup all records with keys less than 30.

d) Lookup all records with keys greater than 30.

e) Lookup all records in the range 20 to 30.

f) Insert a record with key 1.

g) Delete the record with key 23.

h) Insert records with keys 14 through 16.

i) Delete all the records with keys 23 and higher.

! **Exercise 14.2.6:** In Example 14.17 we suggested that it would be possible to borrow keys from a nonsibling to the right (or left) if we used a more complicated algorithm for maintaining keys at interior nodes. Describe a suitable algorithm that rebalances by borrowing from adjacent nodes at a level, regardless of whether they are siblings of the node that has too many or too few key-pointer pairs.

! **Exercise 14.2.7:** If we use the 3-key, 4-pointer nodes of our examples in this section, how many different B-trees are there when the data file has the following numbers of records: (a) 6 (b) 10 !! (c) 15.

! **Exercise 14.2.8:** Suppose we have B-tree nodes with room for three keys and four pointers, as in the examples of this section. Suppose also that when we split a leaf, we divide the pointers 2 and 2, while when we split an interior node, the first 3 pointers go with the first (left) node, and the last 2 pointers go with the second (right) node. We start with a leaf containing pointers to records with keys 1, 2, and 3. We then add in order, records with keys 4, 5, 6, and so on. At the insertion of what key will the B-tree first reach four levels?

**Exercise 14.2.9:** When duplicate keys are allowed in a B-tree, there are some necessary modifications to the algorithms for lookup, insertion, and deletion that we described in this section. Give the changes for: (a) lookup (b) insertion (c) deletion.

## 14.3 Hash Tables

There are a number of data structures involving a hash table that are useful as indexes. We assume the reader has seen the hash table used as a main-memory data structure. In such a structure there is a *hash function* $h$ that takes a search key (the *hash key*) as an argument and computes from it an integer in the range 0 to $B-1$, where $B$ is the number of *buckets*. A *bucket array*, which is an array indexed from 0 to $B-1$, holds the headers of $B$ linked lists, one for each bucket of the array. If a record has search key $K$, then we store the record by linking it to the bucket list for the bucket numbered $h(K)$.

## Should We Delete From B-Trees?

There are B-tree implementations that don't fix up deletions at all. If a leaf has too few keys and pointers, it is allowed to remain as it is. The rationale is that most files grow on balance, and while there might be an occasional deletion that makes a leaf become subminimum, the leaf will probably soon grow again and attain the minimum number of key-pointer pairs once again.

Further, if records have pointers from outside the B-tree index, then we need to replace the record by a "tombstone," and we don't want to delete its pointer from the B-tree anyway. In certain circumstances, when it can be guaranteed that all accesses to the deleted record will go through the B-tree, we can even leave the tombstone in place of the pointer to the record at a leaf of the B-tree. Then, space for the record can be reused.

c) The same as (a), but the B-tree is a sparse index.

d) The data file is a sequential file, and the B-tree is a sparse index, but each primary block of the data file has one overflow block. On average, the primary block is full, and the overflow block is half full. However, records are in no particular order within a primary block and its overflow block.

! e) Instead of the B-tree leaves having pointers to data records, the B-tree leaves hold the records themselves. A block can hold ten records, but on average, a leaf block is 70% full; i.e., there are seven records per leaf block.

**Exercise 14.2.2:** Repeat Exercise 14.2.1 in the case that the query is a range query that is matched by 200 records.

**Exercise 14.2.3:** Suppose pointers are 4 bytes long, and keys are 20 bytes long. How many keys and pointers will a block of 16,384 bytes have?

**Exercise 14.2.4:** What are the minimum numbers of keys and pointers in B-tree (*i*) interior nodes and (*ii*) leaves, when:

a) $n = 11$; i.e., a block holds 11 keys and 12 pointers.

b) $n = 12$; i.e., a block holds 12 keys and 13 pointers.

**Exercise 14.2.5:** Execute the following operations on Fig. 14.13. Describe the changes for operations that modify the tree.

a) Lookup the record with key 40.

b) Lookup the record with key 41.

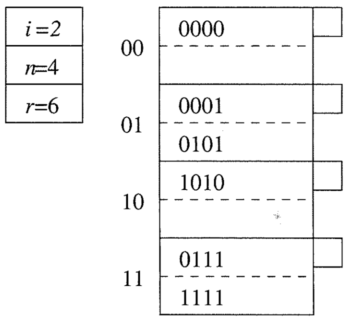

Figure 14.29: Adding a fourth bucket

the 1.7 ratio of records to buckets. Then, we shall raise $n$ to 5 and $i$ becomes 3. □

Lookup in a linear hash table follows the procedure we described for selecting the bucket in which an inserted record belongs. If the record we wish to look up is not in that bucket, it cannot be anywhere.

### 14.3.9 Exercises for Section 14.3

**Exercise 14.3.1:** We did not discuss how deletions can be carried out in a linear or extensible hash table. The mechanics of locating the record(s) to be deleted should be obvious. What method would you suggest for executing the deletion? In particular, what are the advantages and disadvantages of restructuring the table if its smaller size after deletion allows for compression of certain blocks?

! **Exercise 14.3.2:** The material of this section assumes that search keys are unique. However, only small modifications are needed to allow the techniques to work for search keys with duplicates. Describe the necessary changes to insertion, deletion, and lookup algorithms, and suggest the major problems that arise when there are duplicates in each of the following kinds of hash tables: (a) simple (b) linear (c) extensible.

**Exercise 14.3.3:** Show what happens to the buckets in Fig. 14.20 if the following insertions and deletions occur:

*i.* Records $g$ through $j$ are inserted into buckets 0 through 3, respectively.

*ii.* Records $a$ and $b$ are deleted.

*iii.* Records $k$ through $n$ are inserted into buckets 0 through 3, respectively.

*iv.* Records $c$ and $d$ are deleted.

**Exercise 14.3.4:** In an extensible hash table with $n$ records per block, what is the probability that an overflowing block will have to be handled recursively; i.e., all members of the block will go into the same one of the two blocks created in the split?

**Exercise 14.3.5:** Suppose keys are hashed to four-bit sequences, as in our examples of extensible and linear hashing in this section. However, also suppose that blocks can hold three records, rather than the two-record blocks of our examples. If we start with a hash table with two empty blocks (corresponding to 0 and 1), show the organization after we insert records with hashed keys:

a) $1111, 1110, \ldots, 0000$, and the method of hashing is extensible hashing.

b) $1111, 1110, \ldots, 0000$, and the method of hashing is linear hashing with a capacity threshold of 75%.

c) $0000, 0001, \ldots, 1111$, and the method of hashing is extensible hashing.

d) $0000, 0001, \ldots, 1111$, and the method of hashing is linear hashing with a capacity threshold of 100%.

**Exercise 14.3.6:** Suppose we use a linear or extensible hashing scheme, but there are pointers to records from outside. These pointers prevent us from moving records between blocks, as is sometimes required by these hashing methods. Suggest several ways that we could modify the structure to allow pointers from outside.

! **Exercise 14.3.7:** Some hash functions do not work as well as theoretically possible. Suppose that we use the hash function on integer keys $i$ defined by $h(i) = i^2 \bmod B$, where $B$ is the number of buckets.

a) What is wrong with this hash function if $B = 10$?

b) How good is this hash function if $B = 16$?

c) Are there values of $B$ for which this hash function is useful?

!! **Exercise 14.3.8:** A linear-hashing scheme with blocks that hold $k$ records uses a threshold constant $c$, such that the current number of buckets $n$ and the current number of records $r$ are related by $r = ckn$. For instance, in Example 14.24 we used $k = 2$ and $c = 0.85$, so there were 1.7 records per bucket; i.e., $r = 1.7n$.

a) Suppose for convenience that each key occurs exactly its expected number of times.[7] As a function of $c$, $k$, and $n$, how many blocks, including overflow blocks, are needed for the structure?

[7]This assumption does not mean all buckets have the same number of records, because some buckets represent twice as many keys as others.

b) Keys w
ords wi
is, if $\lambda$
the act
Under t
a funct

! **Exercise 14**
hash into a ta
to keep block
What are the
store this has

## 14.4 M

All the inde
assume a sing
key value. A
dimensional
must be prov
far in this ch
several ways:

- Indexes
single l

- Hash t
lookup
cannot

In the balanc
for multidim
fields that fo
result.

### 14.4.1 A

There are a r
2-dimensiona
cations can b
cialized syste
these speciali
support certa

One impo
data. A *ge*
dimensional

b) Keys will not generally distribute equally, but rather the number of records with a given key (or suffix of a key) will be *Poisson distributed.* That is, if $\lambda$ is the expected number of records with a given key suffix, then the actual number of such records will be $i$ with probability $e^{-\lambda}\lambda^i/i!$. Under this assumption, calculate the expected number of blocks used, as a function of $c$, $k$, and $n$.

! **Exercise 14.3.9:** Suppose we have a file of 1,000,000 records that we want to hash into a table with 2000 buckets. 100 records will fit in a block, and we wish to keep blocks as full as possible, but not allow two buckets to share a block. What are the minimum and maximum number of blocks that we could need to store this hash table?

## 14.4 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. Although the search key may involve several attributes, the one-dimensional nature of indexes such as B-trees comes from the fact that values must be provided for all attributes of the search key, or the index is useless. So far in this chapter, we took advantage of a one-dimensional search-key space in several ways:

- Indexes on sequential files and B-trees both take advantage of having a single linear order for the keys.

- 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.

In the balance of this chapter, we shall look at index structures that are suitable for multidimensional data. In these structures, any nonempty subset of the fields that form the dimensions can be given values, and some speedup will result.

### 14.4.1 Applications of Multidimensional Indexes

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 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.

One important application of multidimensional indexes involves geographic data. A *geographic information system* stores objects in a (typically) two-dimensional space. The objects may be points or shapes. Often, these databases

In Fig. 14.35 we see the data from Example 14.27 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. □

### 14.5.6 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 14.5.4 that a correlation between age and salary would cause most points of Fig. 14.32 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 required to support only 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.

### 14.5.7 Exercises for Section 14.5

**Exercise 14.5.1:** In Fig. 14.36 are specifications for twelve of the thirteen PC's introduced in Fig. 2.21. 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.

! 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.

| *model* | *speed* | *ram* | *hd* |
|---|---|---|---|
| 1001 | 2.66 | 1024 | 250 |
| 1002 | 2.10 | 512 | 250 |
| 1003 | 1.42 | 512 | 80 |
| 1004 | 2.80 | 1024 | 250 |
| 1005 | 3.20 | 512 | 250 |
| 1006 | 3.20 | 1024 | 320 |
| 1007 | 2.20 | 1024 | 200 |
| 1008 | 2.20 | 2048 | 250 |
| 1009 | 2.00 | 1024 | 250 |
| 1010 | 2.80 | 2048 | 300 |
| 1011 | 1.86 | 2048 | 160 |
| 1012 | 2.80 | 1024 | 160 |

Figure 14.36: Some PC's and their characteristics

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

**Exercise 14.5.2:** 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. 14.36 well.

**Exercise 14.5.3:** Suppose we place the data of Fig. 14.36 in a grid file with dimensions for speed and ram only. The partitions are at speeds of 2.00, 2.20, and 2.80, and at ram of 1024 and 2048. Suppose also that only two points can fit in one bucket. Suggest good splits if we insert a point with speed 2.5 and ram 1536.

! **Exercise 14.5.4:** Suppose we wish to place the data of Fig. 14.36 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 14.5.5:** 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 330 < x AND x < 400 AND 620 < y AND y < 860;
```

! b) We wish to perform a nearest-neighbor query for the point $(110, 245)$. 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 $(115, 230)$. What other buckets must be searched to verify that this point is the closest?

!! **Exercise 14.5.6:** Suppose we have a hash table whose buckets are numbered 0 to $2^n - 1$; 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 specify either 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.

## 14.6 Tree 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, we shall 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.

### 14.6.1 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 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. 14.37 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,

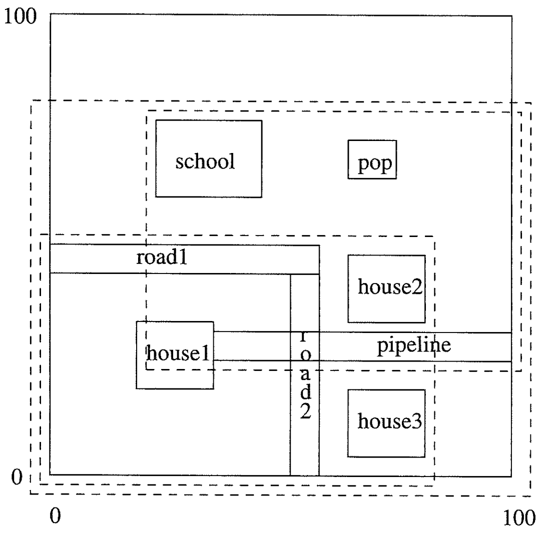

Figure 14.48: Extending a region to accommodate new data

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. 14.47, 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. 14.48. We also must change the description of the region in the top node of Fig. 14.47 from $((0,0),\ (60,50))$ to $((0,0),\ (80,50))$. □

## 14.6.9 Exercises for Section 14.6

**Exercise 14.6.1:** Show a multiple-key index for the data of Fig. 14.36 if the indexes are on:

a) Ram then hard-disk.

b) Speed, then ram.

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

**Exercise 14.6.2:** Place the data of Fig. 14.36 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 att

**Exercise 14.6.3:**
attributes $x$ and $y$
$y$ ranges from 1 to
of $y$, and for each
there are thus 500
help us to answer

```
SELECT z
FROM R
WHERE x =
```

where $C$ and $D$ a
pairs, and we wis
higher-level index
Also assume that

a) How many
if the first i

b) How many
if the first i

! c) Suppose you
blocks woul
you wanted

**Exercise 14.6.4**
are required to a
Assume data is
be found within

**Exercise 14.6.5**
to:

a) The block

b) The block

**Exercise 14.6.**
insert the points

**! Exercise 14.6.**
we execute a pa
specified, then

a) Explain w

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

**Exercise 14.6.3:** 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 50 different values of $y$, and for each $y$ there are records with 5 different values of $x$. Note that there are thus 5000 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 $y$?

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

! c) Suppose you were allowed to buffer 6 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 14.6.4:** For the structure of Exercise 14.6.3(a), how many disk I/O's are required to answer the range query in which $20 \leq x \leq 35$ and $200 \leq y \leq 350$. Assume data is distributed uniformly; i.e., the expected number of points will be found within any given range.

**Exercise 14.6.5:** In the tree of Fig. 14.39, what new points would be directed to:

a) The block with points $(45, 60)$ and $(50, 75)$?

b) The block with points $(25, 400)$ and $(45, 350)$?

**Exercise 14.6.6:** Show a possible evolution of the tree of Fig. 14.41 if we insert the points $(40, 200)$ and then $(20, 175)$.

! **Exercise 14.6.7:** 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 14.6.8:** 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 14.6.9:** 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 14.6.10:** Place the data of Fig. 14.36 in a quad tree with dimensions speed and ram. Assume the range for speed is 1.00 to 5.00, and for ram it is 500 to 3500. No leaf of the quad tree should have more than two points.

**Exercise 14.6.11:** Repeat Exercise 14.6.10 with the addition of a third dimension, hard-disk, that ranges from 0 to 500.

## 14.7 Bitmap Indexes

Let us now turn to a type of index that is rather different from those 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 $i$th 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 $i$th record has $v$ in field $F$, and it has 0 there if not.

**Example 14.39:** 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:

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

Lastly, 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.

### 14.7.5 Exercises for Section 14.7

**Exercise 14.7.1:** For the data of Fig. 14.36, show the bitmap indexes for the attributes: (a) speed (b) ram (c) hd, both in (*i*) uncompressed form, and (*ii*) compressed form using the scheme of Section 14.7.2.

**Exercise 14.7.2:** Using the bitmaps of Example 14.41, find the jewelry buyers with an age in the range 40–60 and a salary in the range 100–200.

**Exercise 14.7.3:** Consider a file of 100,000 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 100,000 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 14.7.4:** Encode, using the scheme of Section 14.7.2, the following bitmaps:

a) 0110000000100000000100.

b) 10000000010000001001010001.

c) 0001000000000000001000010000.

!! **Exercise 14.7.5:** We suggested in Section 14.7.2 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?

## 14.8 Summary of Chapter 14

✦ *Sequential Files*: Several simple file organizations begin by sorting the data file according to some sort key and placing an index on this file.