13.4.9 Coping With Multiple Disk Crashes

There is a theory of error-correcting codes that allows us to deal with any number of disk crashes — data or redundant — if we use enough redundant disks. This strategy leads to the highest RAID "level," *RAID level 6.* We shall give only a simple example here, where two simultaneous crashes are correctable, and the strategy is based on the simplest error-correcting code, known as a *Hamming code.*

In our description we focus on a system with seven disks, numbered 1 through 7. The first four are data disks, and disks 5 through 7 are redundant. The relationship between data and redundant disks is summarized by the 3×7 matrix of 0's and 1's in Fig. 13.10. Notice that:

a) Every possible column of three 0's and 1's, except for the all-0 column, appears in the matrix of Fig. 13.10.

b) The columns for the redundant disks have a single 1.

c) The columns for the data disks each have at least two 1's.

	Data				Redundant		
Disk number	1	2	3	4	5	6	7
	1	1	1	0	1	0	0
	1	1	0	1	0	1	0
	1	0	1	1	0	0	1

Figure 13.10: Redundancy pattern for a system that can recover from two simultaneous disk crashes

The meaning of each of the three rows of 0's and 1's is that if we look at the corresponding bits from all seven disks, and restrict our attention to those disks that have 1 in that row, then the modulo-2 sum of these bits must be 0. Put another way, the disks with 1 in a given row of the matrix are treated as if they were the entire set of disks in a RAID level 4 scheme. Thus, we can compute the bits of one of the redundant disks by finding the row in which that disk has 1, and taking the modulo-2 sum of the corresponding bits of the other disks that have 1 in the same row.

For the matrix of Fig. 13.10, this rule implies:

1. The bits of disk 5 are the modulo-2 sum of the corresponding bits of disks 1, 2, and 3.

2. The bits of disk 6 are the modulo-2 sum of the corresponding bits of disks 1, 2, and 4.

3. The bits of disk 7 are the modulo-2 sum of the corresponding bits of disks 1, 3, and 4.

We shall see shortly that the particular choice of bits in this matrix gives us a simple rule by which we can recover from two simultaneous disk crashes.

Reading

We may read data from any data disk normally. The redundant disks can be ignored.

Writing

The idea is similar to the writing strategy outlined in Section 13.4.8, but now several redundant disks may be involved. To write a block of some data disk, we compute the modulo-2 sum of the new and old versions of that block. These bits are then added, in a modulo-2 sum, to the corresponding blocks of all those redundant disks that have 1 in a row in which the written disk also has 1.

Example 13.13: Let us again assume that blocks are only eight bits long, and focus on the first blocks of the seven disks involved in our RAID level 6 example. First, suppose the data and redundant first blocks are as given in Fig. 13.11. Notice that the block for disk 5 is the modulo-2 sum of the blocks for the first three disks, the sixth row is the modulo-2 sum of rows 1, 2, and 4, and the last row is the modulo-2 sum of rows 1, 3, and 4.

Disk	Contents
1)	11110000
2)	10101010
3)	00111000
4)	01000001
5)	01100010
6)	00011011
7)	10001001

Figure 13.11: First blocks of all disks

Suppose we rewrite the first block of disk 2 to be 00001111. If we sum this sequence of bits modulo-2 with the sequence 10101010 that is the old value of this block, we get 10100101. If we look at the column for disk 2 in Fig. 13.10, we find that this disk has 1's in the first two rows, but not the third. Since redundant disks 5 and 6 have 1 in rows 1 and 2, respectively, we must perform the sum modulo-2 operation on the current contents of their first blocks and the sequence 10100101 just calculated. That is, we flip the values of positions 1, 3, 6, and 8 of these two blocks. The resulting contents of the first blocks of all disks is shown in Fig. 13.12. Notice that the new contents continue to satisfy the constraints implied by Fig. 13.10: the modulo-2 sum of corresponding blocks that have 1 in a particular row of the matrix of Fig. 13.10 is still all 0's. □

Disk	Contents
1)	11110000
2)	00001111
3)	00111000
4)	01000001
5)	11000111
6)	10111110
7)	10001001

Figure 13.12: First blocks of all disks after rewriting disk 2 and changing the redundant disks

Failure Recovery

Now, let us see how the redundancy scheme outlined above can be used to correct up to two simultaneous disk crashes. Let the failed disks be a and b. Since all columns of the matrix of Fig. 13.10 are different, we must be able to find some row r in which the columns for a and b are different. Suppose that a has 0 in row r, while b has 1 there.

Then we can compute the correct b by taking the modulo-2 sum of corresponding bits from all the disks other than b that have 1 in row r. Note that a is not among these, so none of these disks have failed. Having recomputed b, we must recompute a, with all other disks available. Since every column of the matrix of Fig. 13.10 has a 1 in some row, we can use this row to recompute disk a by taking the modulo-2 sum of bits of those other disks with a 1 in this row.

Disk	Contents
1)	11110000
2)	????????
3)	00111000
4)	01000001
5)	????????
6)	10111110
7)	10001001

Figure 13.13: Situation after disks 2 and 5 fail

Example 13.14: Suppose that disks 2 and 5 fail at about the same time. Consulting the matrix of Fig. 13.10, we find that the columns for these two disks differ in row 2, where disk 2 has 1 but disk 5 has 0. We may thus reconstruct disk 2 by taking the modulo-2 sum of corresponding bits of disks 1, 4, and 6, the other three disks with 1 in row 2. Notice that none of these three disks has failed. For instance, following from the situation regarding the first blocks in Fig. 13.12, we would initially have the data of Fig. 13.13 available after disks 2 and 5 failed.

If we take the modulo-2 sum of the contents of the blocks of disks 1, 4, and 6, we find that the block for disk 2 is 00001111. This block is correct as can be verified from Fig. 13.12. The situation is now as in Fig. 13.14.

Disk	Contents
1)	11110000
2)	00001111
3)	00111000
4)	01000001
5)	????????
6)	10111110
7)	10001001

Figure 13.14: After recovering disk 2

Now, we see that disk 5's column in Fig. 13.10 has a 1 in the first row. We can therefore recompute disk 5 by taking the modulo-2 sum of corresponding bits from disks 1, 2, and 3, the other three disks that have 1 in the first row. For block 1, this sum is 11000111. Again, the correctness of this calculation can be confirmed by Fig. 13.12. □

13.4.10 Exercises for Section 13.4

Exercise 13.4.1: Compute the parity bit for the following bit sequences:

a) 00111011.

b) 00000000.

c) 10101101.

Exercise 13.4.2: We can have two parity bits associated with a string if we follow the string by one bit that is a parity bit for the odd positions and a second that is the parity bit for the even positions. For each of the strings in Exercise 13.4.1, find the two bits that serve in this way.

Additional Observations About RAID Level 6

1. We can combine the ideas of RAID levels 5 and 6, by varying the choice of redundant disks according to the block or cylinder number. Doing so will avoid bottlenecks when writing; the scheme described in Section 13.4.9 will cause bottlenecks at the redundant disks.

2. The scheme described in Section 13.4.9 is not restricted to four data disks. The number of disks can be one less than any power of 2, say $2^k - 1$. Of these disks, k are redundant, and the remaining $2^k - k - 1$ are data disks, so the redundancy grows roughly as the logarithm of the number of data disks. For any k, we can construct the matrix corresponding to Fig. 13.10 by writing all possible columns of k 0's and 1's, except the all-0's column. The columns with a single 1 correspond to the redundant disks, and the columns with more than one 1 are the data disks.

Exercise 13.4.3: Suppose we use mirrored disks as in Example 13.8, the failure rate is 4% per year, and it takes 8 hours to replace a disk. What is the mean time to a disk failure involving loss of data?

! **Exercise 13.4.4:** Suppose that a disk has probability F of failing in a given year, and it takes H hours to replace a disk.

a) If we use mirrored disks, what is the mean time to data loss, as a function of F and H?

b) If we use a RAID level 4 or 5 scheme, with N disks, what is the mean time to data loss?

!! **Exercise 13.4.5:** Suppose we use three disks as a mirrored group; i.e., all three hold identical data. If the yearly probability of failure for one disk is F, and it takes H hours to restore a disk, what is the mean time to data loss?

Exercise 13.4.6: Suppose we are using a RAID level 4 scheme with four data disks and one redundant disk. As in Example 13.9 assume blocks are a single byte. Give the block of the redundant disk if the corresponding blocks of the data disks are:

a) 01010110, 11000000, 00111011, and 11111011.

b) 11110000, 11111000, 00111111, and 00000001.

Error-Correcting Codes and RAID Level 6

There is a theory that guides our selection of a suitable matrix, like that of Fig. 13.10, to determine the content of redundant disks. A *code* of length n is a set of bit-vectors (called *code words*) of length n. The *Hamming distance* between two code words is the number of positions in which they differ, and the *minimum distance* of a code is the smallest Hamming distance of any two different code words.

If C is any code of length n, we can require that the corresponding bits on n disks have one of the sequences that are members of the code. As a very simple example, if we are using a disk and its mirror, then $n = 2$, and we can use the code $C = \{00, 11\}$. That is, the corresponding bits of the two disks must be the same. For another example, the matrix of Fig. 13.10 defines the code consisting of the 16 bit-vectors of length 7 that have arbitrary values for the first four bits and have the remaining three bits determined by the rules for the three redundant disks.

If the minimum distance of a code is d, then disks whose corresponding bits are required to be a vector in the code will be able to tolerate $d - 1$ simultaneous disk crashes. The reason is that, should we obscure $d - 1$ positions of a code word, and there were two different ways these positions could be filled in to make a code word, then the two code words would have to differ in at most the $d - 1$ positions. Thus, the code could not have minimum distance d. As an example, the matrix of Fig. 13.10 actually defines the well-known *Hamming code*, which has minimum distance 3. Thus, it can handle two disk crashes.

Exercise 13.4.7: Using the same RAID level 4 scheme as in Exercise 13.4.6, suppose that data disk 1 has failed. Recover the block of that disk under the following circumstances:

a) The contents of disks 2 through 4 are 01010110, 11000000, and 00111011, while the redundant disk holds 11111011.

b) The contents of disks 2 through 4 are 11110000, 11111000, and 00111111, while the redundant disk holds 00000001.

Exercise 13.4.8: Suppose the block on the first disk in Exercise 13.4.6 is changed to 10101010. What changes to the corresponding blocks on the other disks must be made?

Exercise 13.4.9: Suppose we have the RAID level 6 scheme of Example 13.13, and the blocks of the four data disks are 00111100, 11000111, 01010101, and 10000100, respectively.

a) What are the corresponding blocks of the redundant disks?

b) If the third disk's block is rewritten to be 10000000, what steps must be taken to change other disks?

Exercise 13.4.10: Describe the steps taken to recover from the following failures using the RAID level 6 scheme with seven disks: (a) disks 1 and 7, (b) disks 1 and 4, (c) disks 3 and 6.

13.5 Arranging Data on Disk

We now turn to the matter of how disks are used store databases. A data element such as a tuple or object is represented by a *record*, which consists of consecutive bytes in some disk block. Collections such as relations are usually represented by placing the records that represent their data elements in one or more blocks. It is normal for a disk block to hold only elements of one relation, although there are organizations where blocks hold tuples of several relations. In this section, we shall cover the basic layout techniques for both records and blocks.

13.5.1 Fixed-Length Records

The simplest sort of record consists of fixed-length *fields*, one for each attribute of the represented tuple. Many machines allow more efficient reading and writing of main memory when data begins at an address that is a multiple of 4 or 8; some even require us to do so. Thus, it is common to begin all fields at a multiple of 4 or 8, as appropriate. Space not used by the previous field is wasted. Note that, even though records are kept in secondary, not main, memory, they are manipulated in main memory. Thus it is necessary to lay out the record so it can be moved to main memory and accessed efficiently there.

Often, the record begins with a *header*, a fixed-length region where information about the record itself is kept. For example, we may want to keep in the record:

1. A pointer to the schema for the data stored in the record. For example, a tuple's record could point to the schema for the relation to which the tuple belongs. This information helps us find the fields of the record.

2. The length of the record. This information helps us skip over records without consulting the schema.

3. Timestamps indicating the time the record was last modified, or last read. This information may be useful for implementing database transactions as will be discussed in Chapter 18.

4. Pointers to the fields of the record. This information can substitute for schema information, and it will be seen to be important when we consider variable-length fields in Section 13.7.

```
CREATE TABLE MovieStar(
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    gender CHAR(1),
    birthdate DATE
);
```

Figure 13.15: A SQL table declaration

Example 13.15: Figure 13.15 repeats our running `MovieStar` schema. Let us assume all fields must start at a byte that is a multiple of four. Tuples of this relation have a header and the following four fields:

1. The first field is for `name`, and this field requires 30 bytes. If we assume that all fields begin at a multiple of 4, then we allocate 32 bytes for the `name`.

2. The next attribute is `address`. A `VARCHAR` attribute requires a fixed-length segment of bytes, with one more byte than the maximum length (for the string's endmarker). Thus, we need 256 bytes for `address`.

3. Attribute `gender` is a single byte, holding either the character `'M'` or `'F'`. We allocate 4 bytes, so the next field can start at a multiple of 4.

4. Attribute `birthdate` is a SQL `DATE` value, which is a 10-byte string. We shall allocate 12 bytes to its field, to keep subsequent records in the block aligned at multiples of 4.

. The header of the record will hold:

a) A pointer to the record schema.

b) The record length.

c) A timestamp indicating when the record was created.

We shall assume each of these items is 4 bytes long. Figure 13.16 shows the layout of a record for a `MovieStar` tuple. The length of the record is 316 bytes. □

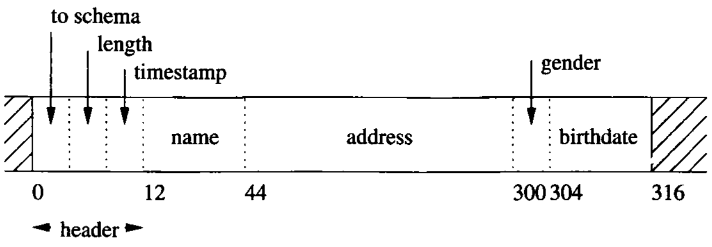

Figure 13.16: Layout of records for tuples of the `MovieStar` relation

13.5.2 Packing Fixed-Length Records into Blocks

Records representing tuples of a relation are stored in blocks of the disk and moved into main memory (along with their entire block) when we need to access or update them. The layout of a block that holds records is suggested in Fig. 13.17.

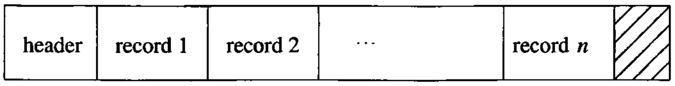

Figure 13.17: A typical block holding records

In addition to the records, there is a *block header* holding information such as:

1. Links to one or more other blocks that are part of a network of blocks such as those that will be described in Chapter 14 for creating indexes to the tuples of a relation.

2. Information about the role played by this block in such a network.

3. Information about which relation the tuples of this block belong to.

4. A "directory" giving the offset of each record in the block.

5. Timestamp(s) indicating the time of the block's last modification and/or access.

By far the simplest case is when the block holds tuples from one relation, and the records for those tuples have a fixed format. In that case, following the header, we pack as many records as we can into the block and leave the remaining space unused.

Example 13.16: Suppose we are storing records with the layout developed in Example 13.15. These records are 316 bytes long. Suppose also that we use 4096-byte blocks. Of these bytes, say 12 will be used for a block header, leaving 4084 bytes for data. In this space we can fit twelve records of the given 316-byte format, and 292 bytes of each block are wasted space. □

13.5.3 Exercises for Section 13.5

Exercise 13.5.1: Suppose a record has the following fields in this order: A character string of length 15, an integer of 2 bytes, a SQL date, and a SQL time (no decimal point). How many bytes does the record take if:

a) Fields can start at any byte.

b) Fields must start at a byte that is a multiple of 4.

c) Fields must start at a byte that is a multiple of 8.

Exercise 13.5.2: Repeat Exercise 13.5.1 for the list of fields: a real of 8 bytes, a character string of length 17, a single byte, and a SQL date.

Exercise 13.5.3: Assume fields are as in Exercise 13.5.1, but records also have a record header consisting of two 4-byte pointers and a character. Calculate the record length for the three situations regarding field alignment (a) through (c) in Exercise 13.5.1.

Exercise 13.5.4: Repeat Exercise 13.5.2 if the records also include a header consisting of an 8-byte pointer, and ten 2-byte integers.

13.6 Representing Block and Record Addresses

When in main memory, the address of a block is the virtual-memory address of its first byte, and the address of a record within that block is the virtual-memory address of the first byte of that record. However, in secondary storage, the block is not part of the application's virtual-memory address space. Rather, a sequence of bytes describes the location of the block within the overall system of data accessible to the DBMS: the device ID for the disk, the cylinder number, and so on. A record can be identified by giving its block address and the offset of the first byte of the record within the block.

In this section, we shall begin with a discussion of address spaces, especially as they pertain to the common "client-server" architecture for DBMS's (see Section 9.2.4). We then discuss the options for representing addresses, and finally look at "pointer swizzling," the ways in which we can convert addresses in the data server's world to the world of the client application programs.

13.6.1 Addresses in Client-Server Systems

Commonly, a database system consists of a *server* process that provides data from secondary storage to one or more *client* processes that are applications using the data. The server and client processes may be on one machine, or the server and the various clients can be distributed over many machines.

The client application uses a conventional "virtual" address space, typically 32 bits, or about 4 billion different addresses. The operating system or DBMS decides which parts of the address space are currently located in main memory, and hardware maps the virtual address space to physical locations in main memory. We shall not think further of this virtual-to-physical translation, and shall think of the client address space as if it were main memory itself.

The server's data lives in a *database address space.* The addresses of this space refer to blocks, and possibly to offsets within the block. There are several ways that addresses in this address space can be represented:

1. *Physical Addresses.* These are byte strings that let us determine the place within the secondary storage system where the block or record can be found. One or more bytes of the physical address are used to indicate each of:

   (a) The host to which the storage is attached (if the database is stored across more than one machine),

   (b) An identifier for the disk or other device on which the block is located,

   (c) The number of the cylinder of the disk,

   (d) The number of the track within the cylinder,

   (e) The number of the block within the track, and

   (f) (In some cases) the offset of the beginning of the record within the block.

2. *Logical Addresses.* Each block or record has a "logical address," which is an arbitrary string of bytes of some fixed length. A *map table*, stored on disk in a known location, relates logical to physical addresses, as suggested in Fig. 13.18.

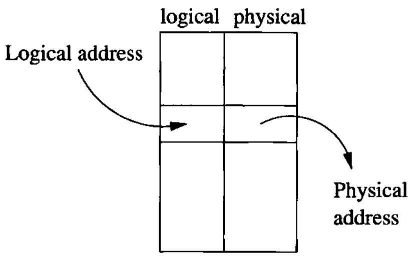

Figure 13.18: A map table translates logical to physical addresses

Notice that physical addresses are long. Eight bytes is about the minimum we could use if we incorporate all the listed elements, and some systems use many more bytes. For example, imagine a database of objects that is designed to last for 100 years. In the future, the database may grow to encompass one million machines, and each machine might be fast enough to create one object every nanosecond. This system would create around 2^{77} objects, which requires a minimum of ten bytes to represent addresses. Since we would probably prefer to reserve some bytes to represent the host, others to represent the storage unit, and so on, a rational address notation would use considerably more than 10 bytes for a system of this scale.

13.6.2 Logical and Structured Addresses

One might wonder what the purpose of logical addresses could be. All the information needed for a physical address is found in the map table, and following logical pointers to records requires consulting the map table and then going to the physical address. However, the level of indirection involved in the map table allows us considerable flexibility. For example, many data organizations require us to move records around, either within a block or from block to block. If we use a map table, then all pointers to the record refer to this map table, and all we have to do when we move or delete the record is to change the entry for that record in the table.

Many combinations of logical and physical addresses are possible as well, yielding *structured* address schemes. For instance, one could use a physical address for the block (but not the offset within the block), and add the key value for the record being referred to. Then, to find a record given this structured address, we use the physical part to reach the block containing that record, and we examine the records of the block to find the one with the proper key.

A similar, and very useful, combination of physical and logical addresses is to keep in each block an *offset table* that holds the offsets of the records within the block, as suggested in Fig. 13.19. Notice that the table grows from the front end of the block, while the records are placed starting at the end of the block. This strategy is useful when the records need not be of equal length. Then, we do not know in advance how many records the block will hold, and we do not have to allocate a fixed amount of the block header to the table initially.

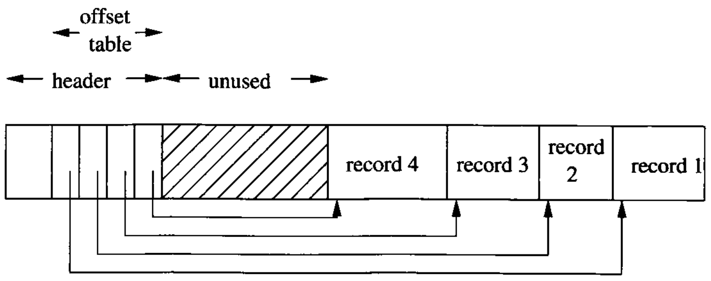

Figure 13.19: A block with a table of offsets telling us the position of each record within the block

The address of a record is now the physical address of its block plus the offset of the entry in the block's offset table for that record. This level of indirection within the block offers many of the advantages of logical addresses, without the need for a global map table.

- We can move the record around within the block, and all we have to do is change the record's entry in the offset table; pointers to the record will still be able to find it.
- We can even allow the record to move to another block, if the offset table entries are large enough to hold a *forwarding address* for the record, giving its new location.
- Finally, we have an option, should the record be deleted, of leaving in its offset-table entry a *tombstone*, a special value that indicates the record has been deleted. Prior to its deletion, pointers to this record may have been stored at various places in the database. After record deletion, following a pointer to this record leads to the tombstone, whereupon the pointer can either be replaced by a null pointer, or the data structure otherwise modified to reflect the deletion of the record. Had we not left the tombstone, the pointer might lead to some new record, with surprising, and erroneous, results.

13.6.3 Pointer Swizzling

Often, pointers or addresses are part of records. This situation is not typical for records that represent tuples of a relation, but it is common for tuples that represent objects. Also, modern object-relational database systems allow attributes of pointer type (called references), so even relational systems need the ability to represent pointers in tuples. Finally, index structures are composed of blocks that usually have pointers within them. Thus, we need to study the management of pointers as blocks are moved between main and secondary memory.

As we mentioned earlier, every block, record, object, or other referenceable data item has two forms of address: its *database address* in the server's address space, and a *memory address* if the item is currently copied in virtual memory. When in secondary storage, we surely must use the database address of the item. However, when the item is in the main memory, we can refer to the item by either its database address or its memory address. It is more efficient to put memory addresses wherever an item has a pointer, because these pointers can be followed using a single machine instruction.

In contrast, following a database address is much more time-consuming. We need a table that translates from all those database addresses that are currently in virtual memory to their current memory address. Such a *translation table* is suggested in Fig. 13.20. It may look like the map table of Fig. 13.18 that translates between logical and physical addresses. However:

a) Logical and physical addresses are both representations for the database address. In contrast, memory addresses in the translation table are for copies of the corresponding object in memory.

b) All addressable items in the database have entries in the map table, while only those items currently in memory are mentioned in the translation table.

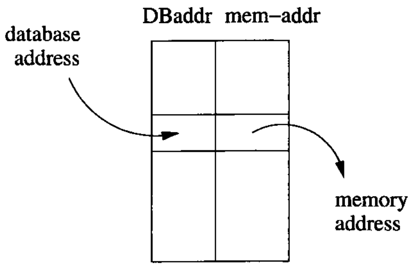

Figure 13.20: The translation table turns database addresses into their equivalents in memory

To avoid the cost of translating repeatedly from database addresses to memory addresses, several techniques have been developed that are collectively known as *pointer swizzling*. The general idea is that when we move a block from secondary to main memory, pointers within the block may be "swizzled," that is, translated from the database address space to the virtual address space. Thus, a pointer actually consists of:

1. A bit indicating whether the pointer is currently a database address or a (swizzled) memory address.

2. The database or memory pointer, as appropriate. The same space is used for whichever address form is present at the moment. Of course, not all the space may be used when the memory address is present, because it is typically shorter than the database address.

Example 13.17: Figure 13.21 shows a simple situation in which the Block 1 has a record with pointers to a second record on the same block and to a record on another block. The figure also shows what might happen when Block 1 is copied to memory. The first pointer, which points within Block 1, can be swizzled so it points directly to the memory address of the target record.

However, if Block 2 is not in memory at this time, then we cannot swizzle the second pointer; it must remain unswizzled, pointing to the database address of its target. Should Block 2 be brought to memory later, it becomes theoretically possible to swizzle the second pointer of Block 1. Depending on the swizzling strategy used, there may or may not be a list of such pointers that are in memory, referring to Block 2; if so, then we have the option of swizzling the pointer at that time. □

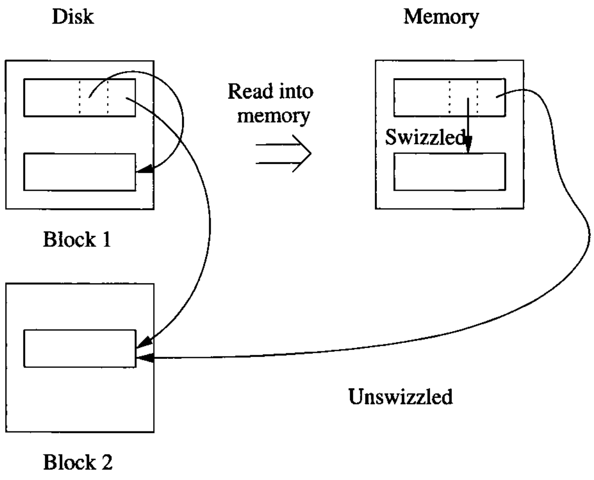

Figure 13.21: Structure of a pointer when swizzling is used

Automatic Swizzling

There are several strategies we can use to determine when to swizzle pointers. If we use *automatic swizzling*, then as soon as a block is brought into memory, we locate all its pointers and addresses and enter them into the translation table if they are not already there. These pointers include both the pointers *from* records in the block to elsewhere and the addresses of the block itself and/or its records, if these are addressable items. We need some mechanism to locate the pointers within the block. For example:

1. If the block holds records with a known schema, the schema will tell us where in the records the pointers are found.

2. If the block is used for one of the index structures we shall discuss in Chapter 14, then the block will hold pointers at known locations.

3. We may keep within the block header a list of where the pointers are.

When we enter into the translation table the addresses for the block just moved into memory, and/or its records, we know where in memory the block has been buffered. We may thus create the translation-table entry for these database addresses straightforwardly. When we insert one of these database addresses A into the translation table, we may find it in the table already, because its block is currently in memory. In this case, we replace A in the block just moved to memory by the corresponding memory address, and we set the "swizzled" bit to true. On the other hand, if A is not yet in the translation table, then its block has not been copied into main memory. We therefore cannot swizzle this pointer and leave it in the block as a database pointer.

Suppose that during the use of this data, we follow a pointer P and we find that P is still unswizzled, i.e., in the form of a database pointer. We consult the translation table to see if database address P currently has a memory equivalent. If not, block B must be copied into a memory buffer. Once B is in memory, we can "swizzle" P by replacing its database form by the equivalent memory form.

Swizzling on Demand

Another approach is to leave all pointers unswizzled when the block is first brought into memory. We enter its address, and the addresses of its pointers, into the translation table, along with their memory equivalents. If we follow a pointer P that is inside some block of memory, we swizzle it, using the same strategy that we followed when we found an unswizzled pointer using automatic swizzling.

The difference between on-demand and automatic swizzling is that the latter tries to get all the pointers swizzled quickly and efficiently when the block is loaded into memory. The possible time saved by swizzling all of a block's pointers at one time must be weighed against the possibility that some swizzled pointers will never be followed. In that case, any time spent swizzling and unswizzling the pointer will be wasted.

An interesting option is to arrange that database pointers look like invalid memory addresses. If so, then we can allow the computer to follow any pointer as if it were in its memory form. If the pointer happens to be unswizzled, then the memory reference will cause a hardware trap. If the DBMS provides a function that is invoked by the trap, and this function "swizzles" the pointer in the manner described above, then we can follow swizzled pointers in single instructions, and only need to do something more time consuming when the pointer is unswizzled.

No Swizzling

Of course it is possible never to swizzle pointers. We still need the translation table, so the pointers may be followed in their unswizzled form. This approach does offer the advantage that records cannot be pinned in memory, as discussed in Section 13.6.5, and decisions about which form of pointer is present need not be made.

Programmer Control of Swizzling

In some applications, it may be known by the application programmer whether the pointers in a block are likely to be followed. This programmer may be able to specify explicitly that a block loaded into memory is to have its pointers swizzled, or the programmer may call for the pointers to be swizzled only as needed. For example, if a programmer knows that a block is likely to be accessed heavily, such as the root block of a B-tree (discussed in Section 14.2), then the pointers would be swizzled. However, blocks that are loaded into memory, used once, and then likely dropped from memory, would not be swizzled.

13.6.4 Returning Blocks to Disk

When a block is moved from memory back to disk, any pointers within that block must be "unswizzled"; that is, their memory addresses must be replaced by the corresponding database addresses. The translation table can be used to associate addresses of the two types in either direction, so in principle it is possible to find, given a memory address, the database address to which the memory address is assigned.

However, we do not want each unswizzling operation to require a search of the entire translation table. While we have not discussed the implementation of this table, we might imagine that the table of Fig. 13.20 has appropriate indexes. If we think of the translation table as a relation, then the problem of finding the memory address associated with a database address x can be expressed as the query:

```
SELECT memAddr
FROM TranslationTable
WHERE dbAddr = x;
```

For instance, a hash table using the database address as the key might be appropriate for an index on the `dbAddr` attribute; Chapter 14 suggests possible data structures.

If we want to support the reverse query,

```
SELECT dbAddr
FROM TranslationTable
WHERE memAddr = y;
```

then we need to have an index on attribute `memAddr` as well. Again, Chapter 14 suggests data structures suitable for such an index. Also, Section 13.6.5 talks about linked-list structures that in some circumstances can be used to go from a memory address to all main-memory pointers to that address.

13.6.5 Pinned Records and Blocks

A block in memory is said to be *pinned* if it cannot at the moment be written back to disk safely. A bit telling whether or not a block is pinned can be located in the header of the block. There are many reasons why a block could be pinned, including requirements of a recovery system as discussed in Chapter 17. Pointer swizzling introduces an important reason why certain blocks must be pinned.

If a block B_1 has within it a swizzled pointer to some data item in block B_2, then we must be very careful about moving block B_2 back to disk and reusing its main-memory buffer. The reason is that, should we follow the pointer in B_1, it will lead us to the buffer, which no longer holds B_2; in effect, the pointer has become dangling. A block, like B_2, that is referred to by a swizzled pointer from somewhere else is therefore pinned.

When we write a block back to disk, we not only need to "unswizzle" any pointers in that block. We also need to make sure it is not pinned. If it is pinned, we must either unpin it, or let the block remain in memory, occupying space that could otherwise be used for some other block. To unpin a block that is pinned because of swizzled pointers from outside, we must "unswizzle" any pointers to it. Consequently, the translation table must record, for each database address whose data item is in memory, the places in memory where swizzled pointers to that item exist. Two possible approaches are:

1. Keep the list of references to a memory address as a linked list attached to the entry for that address in the translation table.

2. If memory addresses are significantly shorter than database addresses, we can create the linked list in the space used for the pointers themselves. That is, each space used for a database pointer is replaced by

   (a) The swizzled pointer, and

   (b) Another pointer that forms part of a linked list of all occurrences of this pointer.

   Figure 13.22 suggests how two occurrences of a memory pointer y could be linked, starting at the entry in the translation table for database address x and its corresponding memory address y.

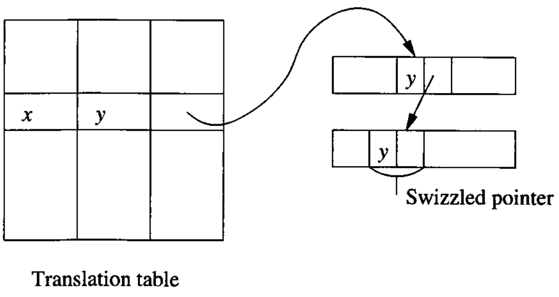

Figure 13.22: A linked list of occurrences of a swizzled pointer

13.6.6 Exercises for Section 13.6

Exercise 13.6.1: If we represent physical addresses for the Megatron 747 disk by allocating a separate byte or bytes to each of the cylinder, track within a cylinder, and block within a track, how many bytes do we need? Make a reasonable assumption about the maximum number of blocks on each track; recall that the Megatron 747 has a variable number of sectors/track.

Exercise 13.6.2: Repeat Exercise 13.6.1 for the Megatron 777 disk described in Exercise 13.2.1

Exercise 13.6.3: If we wish to represent record addresses as well as block addresses, we need additional bytes. Assuming we want addresses for a single Megatron 747 disk as in Exercise 13.6.1, how many bytes would we need for record addresses if we:

a) Included the number of the byte within a block as part of the physical address.

b) Used structured addresses for records. Assume that the stored records have a 4-byte integer as a key.

Exercise 13.6.4: Today, IP addresses have four bytes. Suppose that block addresses for a world-wide address system consist of an IP address for the host, a device number between 1 and 1000, and a block address on an individual device (assumed to be a Megatron 747 disk). How many bytes would block addresses require?

Exercise 13.6.5: In IP version 6, IP addresses are 16 bytes long. In addition, we may want to address not only blocks, but records, which may start at any byte of a block. However, devices will have their own IP address, so there will be no need to represent a device within a host, as we suggested was necessary in Exercise 13.6.4. How many bytes would be needed to represent addresses in these circumstances, again assuming devices were Megatron 747 disks?

! **Exercise 13.6.6:** Suppose we wish to represent the addresses of blocks on a Megatron 747 disk logically, i.e., using identifiers of k bytes for some k. We also need to store on the disk itself a map table, as in Fig. 13.18, consisting of pairs of logical and physical addresses. The blocks used for the map table itself are not part of the database, and therefore do not have their own logical addresses in the map table. Assuming that physical addresses use the minimum possible number of bytes for physical addresses (as calculated in Exercise 13.6.1), and logical addresses likewise use the minimum possible number of bytes for logical addresses, how many blocks of 4096 bytes does the map table for the disk occupy?

! **Exercise 13.6.7:** Suppose that we have 4096-byte blocks in which we store records of 100 bytes. The block header consists of an offset table, as in Fig. 13.19, using 2-byte pointers to records within the block. On an average day, two records per block are inserted, and one record is deleted. A deleted record must have its pointer replaced by a "tombstone," because there may be dangling pointers to it. For specificity, assume the deletion on any day always occurs before the insertions. If the block is initially empty, after how many days will there be no room to insert any more records?

Exercise 13.6.8: Suppose that if we swizzle all pointers automatically, we can perform the swizzling in half the time it would take to swizzle each one separately. If the probability that a pointer in main memory will be followed at least once is p, for what values of p is it more efficient to swizzle automatically than on demand?

! **Exercise 13.6.9:** Generalize Exercise 13.6.8 to include the possibility that we never swizzle pointers. Suppose that the important actions take the following times, in some arbitrary time units:

i. On-demand swizzling of a pointer: 30.

ii. Automatic swizzling of pointers: 20 per pointer.

iii. Following a swizzled pointer: 1.

iv. Following an unswizzled pointer: 10.

Suppose that in-memory pointers are either not followed (probability $1 - p$) or are followed k times (probability p). For what values of k and p do no-swizzling, automatic-swizzling, and on-demand-swizzling each offer the best average performance?

13.7 Variable-Length Data and Records

Until now, we have made the simplifying assumptions that records have a fixed schema, and that the schema is a list of fixed-length fields. However, in practice, we also may wish to represent:

1. *Data items whose size varies.* For instance, in Fig. 13.15 we considered a `MovieStar` relation that had an address field of up to 255 bytes. While there might be some addresses that long, the vast majority of them will probably be 50 bytes or less. We could save more than half the space used for storing `MovieStar` tuples if we used only as much space as the actual address needed.

2. *Repeating fields.* If we try to represent a many-many relationship in a record representing an object, we shall have to store references to as many objects as are related to the given object.

3. *Variable-format records.* Sometimes we do not know in advance what the fields of a record will be, or how many occurrences of each field there will be. An important example is a record that represents an XML element, which might have no constraints at all, or might be allowed to have repeating subelements, optional attributes, and so on.

4. *Enormous fields.* Modern DBMS's support attributes whose values are very large. For instance, a movie record might have a field that is a 2-gigabyte MPEG encoding of the movie itself, as well as more mundane fields such as the title of the movie.

13.7.1 Records With Variable-Length Fields

If one or more fields of a record have variable length, then the record must contain enough information to let us find any field of the record. A simple but effective scheme is to put all fixed-length fields ahead of the variable-length fields. We then place in the record header:

1. The length of the record.

2. Pointers to (i.e., offsets of) the beginnings of all the variable-length fields other than the first (which we know must immediately follow the fixed-length fields).

Example 13.18: Suppose we have movie-star records with name, address, gender, and birthdate. We shall assume that the gender and birthdate are fixed-length fields, taking 4 and 12 bytes, respectively. However, both name and address will be represented by character strings of whatever length is appropriate. Figure 13.23 suggests what a typical movie-star record would look like. Note that no pointer to the beginning of the name is needed; that field begins right after the fixed-length portion of the record. □

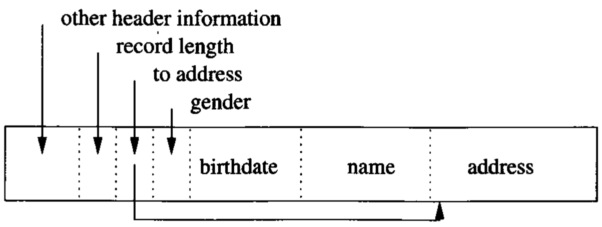

Figure 13.23: A `MovieStar` record with `name` and `address` implemented as variable-length character strings

Representing Null Values

Tuples often have fields that may be NULL. The record format of Fig. 13.23 offers a convenient way to represent NULL values. If a field such as `address` is null, then we put a null pointer in the place where the pointer to an address goes. Then, we need no space for an address, except the place for the pointer. This arrangement can save space on average, even if `address` is a fixed-length field but frequently has the value NULL.

13.7.2 Records With Repeating Fields

A similar situation occurs if a record contains a variable number of occurrences of a field F, but the field itself is of fixed length. It is sufficient to group all occurrences of field F together and put in the record header a pointer to the first. We can locate all the occurrences of the field F as follows. Let the number of bytes devoted to one instance of field F be L. We then add to the offset for the field F all integer multiples of L, starting at 0, then L, $2L$, $3L$, and so on. Eventually, we reach the offset of the field following F or the end of the record, whereupon we stop.

Example 13.19: Suppose we redesign our movie-star records to hold only the name and address (which are variable-length strings) and pointers to all the movies of the star. Figure 13.24 shows how this type of record could be represented. The header contains pointers to the beginning of the address field (we assume the name field always begins right after the header) and to the first of the movie pointers. The length of the record tells us how many movie pointers there are. □

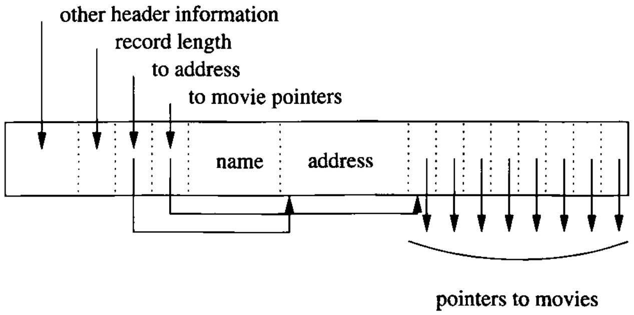

Figure 13.24: A record with a repeating group of references to movies

An alternative representation is to keep the record of fixed length, and put the variable-length portion — be it fields of variable length or fields that repeat an indefinite number of times — on a separate block. In the record itself we keep:

1. Pointers to the place where each repeating field begins, and

2. Either how many repetitions there are, or where the repetitions end.

Figure 13.25 shows the layout of a record for the problem of Example 13.19, but with the variable-length fields `name` and `address`, and the repeating field `starredIn` (a set of movie references) kept on a separate block or blocks.

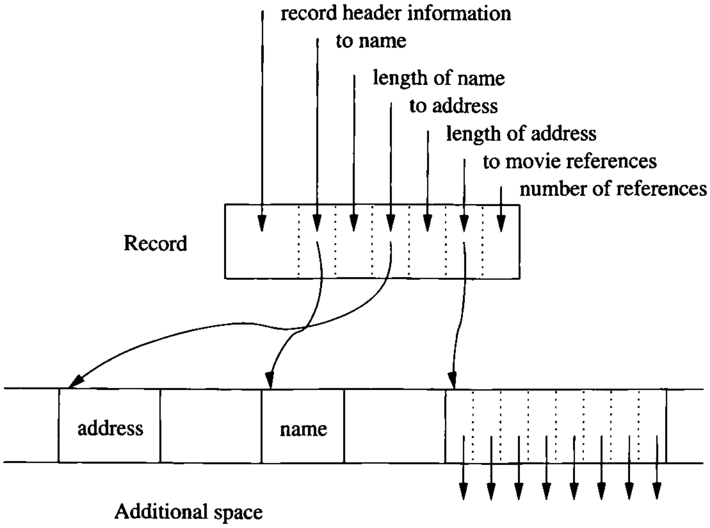

Figure 13.25: Storing variable-length fields separately from the record

There are advantages and disadvantages to using indirection for the variable-length components of a record:

- Keeping the record itself fixed-length allows records to be searched more efficiently, minimizes the overhead in block headers, and allows records to be moved within or among blocks with minimum effort.

- On the other hand, storing variable-length components on another block increases the number of disk I/O's needed to examine all components of a record.

A compromise strategy is to keep in the fixed-length portion of the record enough space for:

1. Some reasonable number of occurrences of the repeating fields,

2. A pointer to a place where additional occurrences could be found, and

3. A count of how many additional occurrences there are.

If there are fewer than this number, some of the space would be unused. If there are more than can fit in the fixed-length portion, then the pointer to additional space will be nonnull, and we can find the additional occurrences by following this pointer.

13.7.3 Variable-Format Records

An even more complex situation occurs when records do not have a fixed schema. We mentioned an example: records that represent XML elements. For another example, medical records may contain information about many tests, but there are thousands of possible tests, and each patient has results for relatively few of them. If the outcome of each test is an attribute, we would prefer that the record for each tuple hold only the attributes for which the outcome is nonnull.

The simplest representation of variable-format records is a sequence of *tagged fields*, each of which consists of the value of the field preceded by information about the role of this field, such as:

1. The attribute or field name,

2. The type of the field, if it is not apparent from the field name and some readily available schema information, and

3. The length of the field, if it is not apparent from the type.

Example 13.20: Suppose movie stars may have additional attributes such as movies directed, former spouses, restaurants owned, and a number of other known but unusual pieces of information. In Fig. 13.26 we see the beginning of a hypothetical movie-star record using tagged fields. We suppose that single-byte codes are used for the various possible field names and types. Appropriate codes are indicated on the figure, along with lengths for the two fields shown, both of which happen to be of type string. □

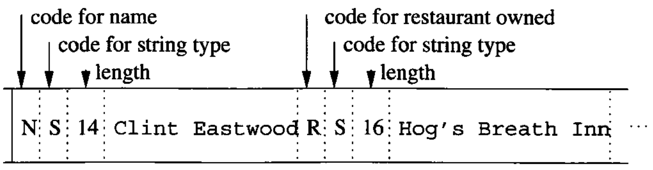

Figure 13.26: A record with tagged fields

13.7.4 Records That Do Not Fit in a Block

Today, DBMS's frequently are used to manage datatypes with large values; often values do not fit in one block. Typical examples are video or audio "clips." Often, these large values have a variable length, but even if the length is fixed for all values of the type, we need special techniques to represent values that are larger than blocks. In this section we shall consider a technique called "spanned records." The management of extremely large values (megabytes or gigabytes) is addressed in Section 13.7.5.

Spanned records also are useful in situations where records are smaller than blocks, but packing whole records into blocks wastes significant amounts of space. For instance, the wasted space in Example 13.16 was only 7%, but if records are just slightly larger than half a block, the wasted space can approach 50%. The reason is that then we can pack only one record per block.

The portion of a record that appears in one block is called a *record fragment.* A record with two or more fragments is called *spanned*, and records that do not cross a block boundary are *unspanned.*

If records can be spanned, then every record and record fragment requires some extra header information:

1. Each record or fragment header must contain a bit telling whether or not it is a fragment.

2. If it is a fragment, then it needs bits telling whether it is the first or last fragment for its record.

3. If there is a next and/or previous fragment for the same record, then the fragment needs pointers to these other fragments.

Example 13.21: Figure 13.27 suggests how records that were about 60% of a block in size could be stored with three records for every two blocks. The header for record fragment $2a$ contains an indicator that it is a fragment, an indicator that it is the first fragment for its record, and a pointer to next fragment, $2b$. Similarly, the header for $2b$ indicates it is the last fragment for its record and holds a back-pointer to the previous fragment $2a$. □

13.7.5 BLOBs

Now, let us consider the representation of truly large values for records or fields of records. The common examples include images in various formats (e.g., GIF, or JPEG), movies in formats such as MPEG, or signals of all sorts: audio, radar, and so on. Such values are often called *binary, large objects*, or BLOBs. When a field has a BLOB as value, we must rethink at least two issues.

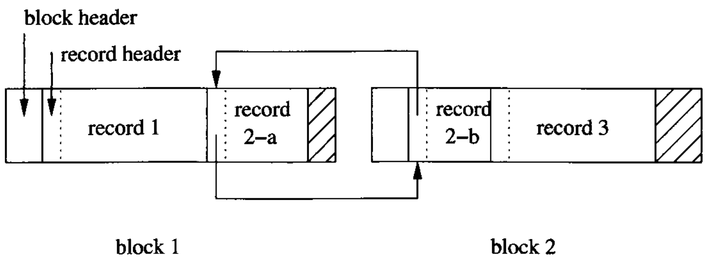

Figure 13.27: Storing spanned records across blocks

Storage of BLOBs

A BLOB must be stored on a sequence of blocks. Often we prefer that these blocks are allocated consecutively on a cylinder or cylinders of the disk, so the BLOB may be retrieved efficiently. However, it is also possible to store the BLOB on a linked list of blocks.

Moreover, it is possible that the BLOB needs to be retrieved so quickly (e.g., a movie that must be played in real time), that storing it on one disk does not allow us to retrieve it fast enough. Then, it is necessary to *stripe* the BLOB across several disks, that is, to alternate blocks of the BLOB among these disks. Thus, several blocks of the BLOB can be retrieved simultaneously, increasing the retrieval rate by a factor approximately equal to the number of disks involved in the striping.

Retrieval of BLOBs

Our assumption that when a client wants a record, the block containing the record is passed from the database server to the client in its entirety may not hold. We may want to pass only the "small" fields of the record, and allow the client to request blocks of the BLOB one at a time, independently of the rest of the record. For instance, if the BLOB is a 2-hour movie, and the client requests that the movie be played, the BLOB could be shipped several blocks at a time to the client, at just the rate necessary to play the movie.

In many applications, it is also important that the client be able to request interior portions of the BLOB without having to receive the entire BLOB. Examples would be a request to see the 45th minute of a movie, or the ending of an audio clip. If the DBMS is to support such operations, then it requires a suitable index structure, e.g., an index by seconds on a movie BLOB.

13.7.6 Column Stores

An alternative to storing tuples as records is to store each column as a record. Since an entire column of a relation may occupy far more than a single block, these records may span many blocks, much as long files do. If we keep the values in each column in the same order, then we can reconstruct the relation from the column records. Alternatively, we can keep tuple ID's or integers with each value, to tell which tuple the value belongs to.

Example 13.22: Consider the relation

X	Y
a	b
c	d
e	f

The column for X can be represented by the record (a, c, e) and the column for Y can be represented by the record (b, d, f). If we want to indicate the tuple to which each value belongs, then we can represent the two columns by the records $\big((1,a),\ (2,c),\ (3,e)\big)$ and $\big((1,b),\ (2,d),\ (3,f)\big)$, respectively. No matter how many tuples the relation above had, the columns would be represented by variable-length records of values or repeating groups of tuple ID's and values. □

If we store relations by columns, it is often possible to compress data, the the values all have a known type. For example, an attribute gender in a relation might have type CHAR(1), but we would use four bytes in a tuple-based record, because it is more convenient to have all components of a tuple begin at word boundaries. However, if all we are storing is a sequence of gender values, then it would make sense to store the column by a sequence of bits. If we did so, we would compress the data by a factor of 32.

However, in order for column-based storage to make sense, it must be the case that most queries call for examination of all, or a large fraction of the values in each of several columns. Recall our discussion in Section 10.6 of "analytic" queries, which are the common kind of queries with the desired characteristic. These "OLAP" queries may benefit from organizing the data by columns.

13.7.7 Exercises for Section 13.7

Exercise 13.7.1: A patient record consists of the following fixed-length fields: the patient's date of birth, social-security number, and patient ID, each 10 bytes long. It also has the following variable-length fields: name, address, and patient history. If pointers within a record require 4 bytes, and the record length is a 4-byte integer, how many bytes, exclusive of the space needed for the variable-length fields, are needed for the record? You may assume that no alignment of fields is required.

Exercise 13.7.2: Suppose records are as in Exercise 13.7.1, and the variable-length fields name, address, and history each have a length that is uniformly distributed. For the name, the range is 10–50 bytes; for address it is 20–80 bytes, and for history it is 0–1000 bytes. What is the average length of a patient record?

The Merits of Data Compression

One might think that with storage so cheap, there is little advantage to compressing data. However, storing data in fewer disk blocks enables us to read and write the data faster, since we use fewer disk I/O's. When we need to read entire columns, then storage by compressed columns can result in significant speedups. However, if we want to read or write only a single tuple, then column-based storage can lose. The reason is that in order to decompress and find the value for the one tuple we want, we need to read the entire column. In contrast, tuple-based storage allows us to read only the block containing the tuple. An even more extreme case is when the data is not only compressed, but encrypted.

In order to make access of single values efficient, we must both compress and encrypt on a block-by-block basis. The most efficient compression methods generally perform better when they are allowed to compress large amounts of data as a group, and they do not lend themselves to block-based decompression. However, in special cases such as the compression of a gender column discussed in Section 13.7.6, we can in fact do block-by-block compression that is as good as possible.

Exercise 13.7.3: Suppose that the patient records of Exercise 13.7.1 are augmented by an additional repeating field that represents cholesterol tests. Each cholesterol test requires 16 bytes for a date and an integer result of the test. Show the layout of patient records if:

a) The repeating tests are kept with the record itself.

b) The tests are stored on a separate block, with pointers to them in the record.

Exercise 13.7.4: Starting with the patient records of Exercise 13.7.1, suppose we add fields for tests and their results. Each test consists of a test name, a date, and a test result. Assume that each such test requires 40 bytes. Also, suppose that for each patient and each test a result is stored with probability p.

a) Assuming pointers and integers each require 4 bytes, what is the average number of bytes devoted to test results in a patient record, assuming that all test results are kept within the record itself, as a variable-length field?

b) Repeat (a), if test results are represented by pointers within the record to test-result fields kept elsewhere.

! c) Suppose we use a hybrid scheme, where room for k test results are kept within the record, and additional test results are found by following a pointer to another block (or chain of blocks) where those results are kept. As a function of p, what value of k minimizes the amount of storage used for test results?

!! d) The amount of space used by the repeating test-result fields is not the only issue. Let us suppose that the figure of merit we wish to minimize is the number of bytes used, plus a penalty of 10,000 if we have to store some results on another block (and therefore will require a disk I/O for many of the test-result accesses we need to do. Under this assumption, what is the best value of k as a function of p?

!! **Exercise 13.7.5:** Suppose blocks have 1000 bytes available for the storage of records, and we wish to store on them fixed-length records of length r, where $500 < r \leq 1000$. The value of r includes the record header, but a record fragment requires an additional 16 bytes for the fragment header. For what values of r can we improve space utilization by spanning records?

!! **Exercise 13.7.6:** An MPEG movie uses about one gigabyte per hour of play. If we carefully organized several movies on a Megatron 747 disk, how many could we deliver with only small delay (say 100 milliseconds) from one disk. Use the timing estimates of Example 13.2, but remember that you can choose how the movies are laid out on the disk.

13.8 Record Modifications

Insertions, deletions, and updates of records often create special problems. These problems are most severe when the records change their length, but they come up even when records and fields are all of fixed length.

13.8.1 Insertion

First, let us consider insertion of new records into a relation. If the records of a relation are kept in no particular order, we can just find a block with some empty space, or get a new block if there is none, and put the record there.

There is more of a problem when the tuples must be kept in some fixed order, such as sorted by their primary key (e.g., see Section 14.1.1). If we need to insert a new record, we first locate the appropriate block for that record. Suppose first that there is space in the block to put the new record. Since records must be kept in order, we may have to slide records around in the block to make space available at the proper point. If we need to slide records, then the block organization that we showed in Fig. 13.19, which we reproduce here as Fig. 13.28, is useful. Recall from our discussion in Section 13.6.2 that we may create an "offset table" in the header of each block, with pointers to the location of each record in the block. A pointer to a record from outside the block is a "structured address," that is, the block address and the location of the entry for the record in the offset table.

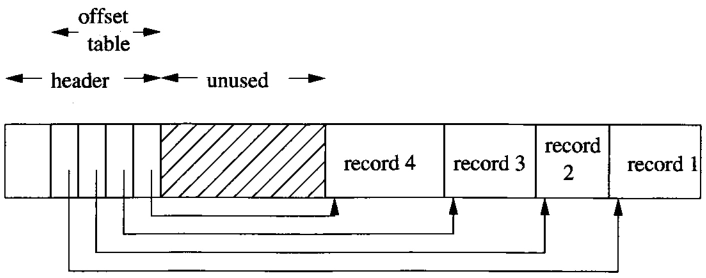

Figure 13.28: An offset table lets us slide records within a block to make room for new records

If we can find room for the inserted record in the block at hand, then we simply slide the records within the block and adjust the pointers in the offset table. The new record is inserted into the block, and a new pointer to the record is added to the offset table for the block. However, there may be no room in the block for the new record, in which case we have to find room outside the block. There are two major approaches to solving this problem, as well as combinations of these approaches.

1. *Find space on a "nearby" block.* For example, if block B_1 has no available space for a record that needs to be inserted in sorted order into that block, then look at the following block B_2 in the sorted order of the blocks. If there is room in B_2, move the highest record(s) of B_1 to B_2, leave forwarding addresses (recall Section 13.6.2) and slide the records around on both blocks.

2. *Create an overflow block.* In this scheme, each block B has in its header a place for a pointer to an *overflow* block where additional records that theoretically belong in B can be placed. The overflow block for B can point to a second overflow block, and so on. Figure 13.29 suggests the structure. We show the pointer for overflow blocks as a nub on the block, although it is in fact part of the block header.

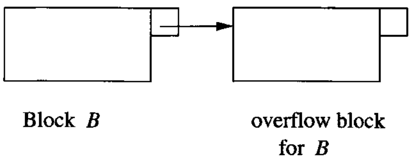

Figure 13.29: A block and its first overflow block

13.8.2 Deletion

When we delete a record, we may be able to reclaim its space. If we use an offset table as in Fig. 13.28 and records can slide around the block, then we can compact the space in the block so there is always one unused region in the center, as suggested by that figure.

If we cannot slide records, we should maintain an available-space list in the block header. Then we shall know where, and how large, the available regions are, when a new record is inserted into the block. Note that the block header normally does not need to hold the entire available space list. It is sufficient to put the list head in the block header, and use the available regions themselves to hold the links in the list, much as we did in Fig. 13.22.

There is one additional complication involved in deletion, which we must remember regardless of what scheme we use for reorganizing blocks. There may be pointers to the deleted record, and if so, we don't want these pointers to dangle or wind up pointing to a new record that is put in the place of the deleted record. The usual technique, which we pointed out in Section 13.6.2, is to place a ***tombstone*** in place of the record. This tombstone is permanent; it must exist until the entire database is reconstructed.

Where the tombstone is placed depends on the nature of record pointers. If pointers go to fixed locations from which the location of the record is found, then we put the tombstone in that fixed location. Here are two examples:

1. We suggested in Section 13.6.2 that if the offset-table scheme of Fig. 13.28 were used, then the tombstone could be a null pointer in the offset table, since pointers to the record were really pointers to the offset table entries.

2. If we are using a map table, as in Fig. 13.18, to translate logical record addresses to physical addresses, then the tombstone can be a null pointer in place of the physical address.

If we need to replace records by tombstones, we should place the bit that serves as a tombstone at the very beginning of the record. Then, only this bit must remain where the record used to begin, and subsequent bytes can be reused for another record, as suggested by Fig. 13.30.

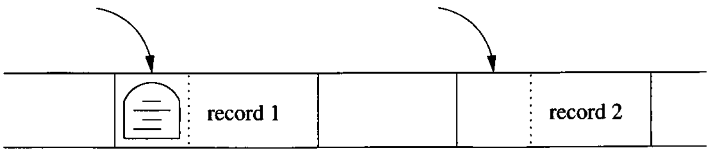

Figure 13.30: Record 1 can be replaced, but the tombstone remains; record 2 has no tombstone and can be seen when we follow a pointer to it

13.8.3 Update

When a fixed-length record is updated, there is no effect on the storage system, because we know it can occupy exactly the same space it did before the update. However, when a variable-length record is updated, we have all the problems associated with both insertion and deletion, except that it is never necessary to create a tombstone for the old version of the record.

If the updated record is longer than the old version, then we may need to create more space on its block. This process may involve sliding records or even the creation of an overflow block. If variable-length portions of the record are stored on another block, as in Fig. 13.25, then we may need to move elements around that block or create a new block for storing variable-length fields. Conversely, if the record shrinks because of the update, we have the same opportunities as with a deletion to recover or consolidate space.

13.8.4 Exercises for Section 13.8

Exercise 13.8.1: Relational database systems have always preferred to use fixed-length tuples if possible. Give three reasons for this preference.

13.9 Summary of Chapter 13

✦ *Memory Hierarchy*: A computer system uses storage components ranging over many orders of magnitude in speed, capacity, and cost per bit. From the smallest/most expensive to largest/cheapest, they are: cache, main memory, secondary memory (disk), and tertiary memory.

✦ *Disks/Secondary Storage*: Secondary storage devices are principally magnetic disks with multigigabyte capacities. Disk units have several circular platters of magnetic material, with concentric tracks to store bits. Platters rotate around a central spindle. The tracks at a given radius from the center of a platter form a cylinder.

✦ *Blocks and Sectors*: Tracks are divided into sectors, which are separated by unmagnetized gaps. Sectors are the unit of reading and writing from the disk. Blocks are logical units of storage used by an application such as a DBMS. Blocks typically consist of several sectors.

✦ *Disk Controller*: The disk controller is a processor that controls one or more disk units. It is responsible for moving the disk heads to the proper cylinder to read or write a requested track. It also may schedule competing requests for disk access and buffers the blocks to be read or written.

✦ *Disk Access Time*: The latency of a disk is the time between a request to read or write a block, and the time the access is completed. Latency is caused principally by three factors: the seek time to move the heads to the proper cylinder, the rotational latency during which the desired block rotates under the head, and the transfer time, while the block moves under the head and is read or written.

- ✦ *Speeding Up Disk Access*: There are several techniques for accessing disk blocks faster for some applications. They include dividing the data among several disks (striping), mirroring disks (maintaining several copies of the data, also to allow parallel access), and organizing data that will be accessed together by tracks or cylinders.

- ✦ *Elevator Algorithm*: We can also speed accesses by queueing access requests and handling them in an order that allows the heads to make one sweep across the disk. The heads stop to handle a request each time it reaches a cylinder containing one or more blocks with pending access requests.

- ✦ *Disk Failure Modes*: To avoid loss of data, systems must be able to handle errors. The principal types of disk failure are intermittent (a read or write error that will not reoccur if repeated), permanent (data on the disk is corrupted and cannot be properly read), and the disk crash, where the entire disk becomes unreadable.

- ✦ *Checksums*: By adding a parity check (extra bit to make the number of 1's in a bit string even), intermittent failures and permanent failures can be detected, although not corrected.

- ✦ *Stable Storage*: By making two copies of all data and being careful about the order in which those copies are written, a single disk can be used to protect against almost all permanent failures of a single sector.

- ✦ *RAID*: These schemes allow data to survive a disk crash. RAID level 4 adds a disk whose contents are a parity check on corresponding bits of all other disks, level 5 varies the disk holding the parity bit to avoid making the parity disk a writing bottleneck. Level 6 involves the use of error-correcting codes and may allow survival after several simultaneous disk crashes.

- ✦ *Records*: Records are composed of several fields plus a record header. The header contains information about the record, possibly including such matters as a timestamp, schema information, and a record length. If the record has varying-length fields, the header may also help locate those fields.

- ✦ *Blocks*: Records are generally stored within blocks. A block header, with information about that block, consumes some of the space in the block, with the remainder occupied by one or more records. To support insertions, deletions and modifications of records, we can put in the block header an offset table that has pointers to each of the records in the block.

- *Spanned Records*: Generally, a record exists within one block. However, if records are longer than blocks, or we wish to make use of leftover space within blocks, then we can break records into two or more fragments, one on each block. A fragment header is then needed to link the fragments of a record.

- *BLOBs*: Very large values, such as images and videos, are called BLOBs (binary, large objects). These values must be stored across many blocks and may require specialized storage techniques such as reserving a cylinder or striping the blocks of the BLOB.

- *Database Addresses*: Data managed by a DBMS is found among several storage devices, typically disks. To locate blocks and records in this storage system, we can use physical addresses, which are a description of the device number, cylinder, track, sector(s), and possibly byte within a sector. We can also use logical addresses, which are arbitrary character strings that are translated into physical addresses by a map table.

- *Pointer Swizzling*: When disk blocks are brought to main memory, the database addresses need to be translated to memory addresses, if pointers are to be followed. The translation is called swizzling, and can either be done automatically, when blocks are brought to memory, or on-demand, when a pointer is first followed.

- *Tombstones*: When a record is deleted, pointers to it will dangle. A tombstone in place of (part of) the deleted record warns the system that the record is no longer there.

- *Pinned Blocks*: For various reasons, including the fact that a block may contain swizzled pointers, it may be unacceptable to copy a block from memory back to its place on disk. Such a block is said to be pinned. If the pinning is due to swizzled pointers, then they must be unswizzled before returning the block to disk.

13.10 References for Chapter 13

The RAID idea can be traced back to [8] on disk striping. The name and error-correcting capability is from [7]. The model of disk failures in Section 13.4 appears in unpublished work of Lampson and Sturgis [5].

There are several useful surveys of disk-related material. A study of RAID systems is in [2]. [10] surveys algorithms suitable for the secondary storage model (block model) of computation. [3] is an important study of how one optimizes a system involving processor, memory, and disk, to perform specific tasks.

References [4] and [11] have more information on record and block structures. [9] discusses column stores as an alternative to the conventional record structures. Tombstones as a technique for dealing with deletion is from [6]. [1] covers data representation issues, such as addresses and swizzling in the context of object-oriented DBMS's.

1. R. G. G. Cattell, *Object Data Management*, Addison-Wesley, Reading MA, 1994.

2. P. M. Chen et al., "RAID: high-performance, reliable secondary storage," *Computing Surveys* **26**:2 (1994), pp. 145–186.

3. J. N. Gray and F. Putzolo, "The five minute rule for trading memory for disk accesses and the 10 byte rule for trading memory for CPU time," *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pp. 395–398, 1987.

4. D. E. Knuth, *The Art of Computer Programming, Vol. I, Fundamental Algorithms, Third Edition*, Addison-Wesley, Reading MA, 1997.

5. B. Lampson and H. Sturgis, "Crash recovery in a distributed data storage system," Technical report, Xerox Palo Alto Research Center, 1976.

6. D. Lomet, "Scheme for invalidating free references," *IBM J. Research and Development* **19**:1 (1975), pp. 26–35.

7. D. A. Patterson, G. A. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks," *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pp. 109–116, 1988.

8. K. Salem and H. Garcia-Molina, "Disk striping," *Proc. Second Intl. Conf. on Data Engineering*, pp. 336–342, 1986.

9. M. Stonebraker et al., "C-Store: a column-oriented DBMS," *Proc. Thirty-first Intl. Conf. on Very Large Database Systems*" (2005).

10. J. S. Vitter, "External memory algorithms," *Proc. Seventeenth Annual ACM Symposium on Principles of Database Systems*, pp. 119–128, 1998.

11. G. Wiederhold, *File Organization for Database Design*, McGraw-Hill, New York, 1987.

Chapter 14

Index Structures

It is not sufficient simply to scatter the records that represent tuples of a relation among various blocks. To see why, think how we would answer the simple query `SELECT * FROM R`. We would have to examine every block in the storage system to find the tuples of R. A better idea is to reserve some blocks, perhaps several whole cylinders, for R. Now, at least we can find the tuples of R without scanning the entire data store.

However, this organization offers little help for a query like

```
SELECT * FROM R WHERE a=10;
```

Section 8.4 introduced us to the importance of creating *indexes* to speed up queries that specify values for one or more attributes. As suggested in Fig. 14.1, an index is any data structure that takes the value of one or more fields and finds the records with that value "quickly." In particular, an index lets us find a record without having to look at more than a small fraction of all possible records. The field(s) on whose values the index is based is called the *search key*, or just "key" if the index is understood.

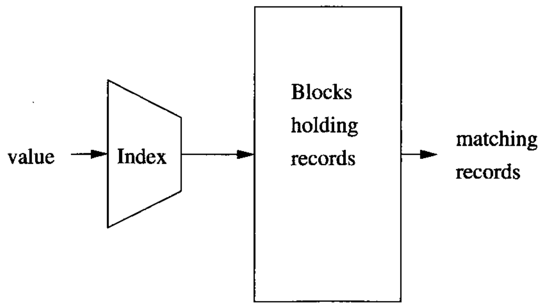

Figure 14.1: An index takes a value for some field(s) and finds records with the matching value

Different Kinds of "Keys"

There are many meanings of the term "key." We used it in Section 2.3.6 to mean the primary key of a relation. We shall also speak of "sort keys," the attribute(s) on which a file of records is sorted. We just introduced "search keys," the attribute(s) for which we are given values and asked to search, through an index, for tuples with matching values. We try to use the appropriate adjective — "primary," "sort," or "search" — when the meaning of "key" is unclear. However, in many cases, the three kinds of keys are one and the same.

In this chapter, we shall introduce the most common form of index in database systems: the B-tree. We shall also discuss hash tables in secondary storage, which is another important index structure. Finally, we consider other index structures that are designed to handle multidimensional data. These structures support queries that specify values or ranges for several attributes at once.

14.1 Index-Structure Basics

In this section, we introduce concepts that apply to all index structures. Storage structures consist of *files*, which are similar to the files used by operating systems. A *data file* may be used to store a relation, for example. The data file may have one or more *index files.* Each index file associates values of the search key with pointers to data-file records that have that value for the attribute(s) of the search key.

Indexes can be "dense," meaning there is an entry in the index file for every record of the data file. They can be "sparse," meaning that only some of the data records are represented in the index, often one index entry per block of the data file. Indexes can also be "primary" or "secondary." A primary index determines the location of the records of the data file, while a secondary index does not. For example, it is common to create a primary index on the primary key of a relation and to create secondary indexes on some of the other attributes.

We conclude the section with a study of information retrieval from documents. The ideas of the section are combined to yield "inverted indexes," which enable efficient retrieval of documents that contain one or more given keywords. This technique is essential for answering search queries on the Web, for instance.

14.1.1 Sequential Files

A *sequential file* is created by sorting the tuples of a relation by their primary key. The tuples are then distributed among blocks, in this order.

Example 14.1: Fig 14.2 shows a sequential file on the right. We imagine that keys are integers; we show only the key field, and we make the atypical assumption that there is room for only two records in one block. For instance, the first block of the file holds the records with keys 10 and 20. In this and several other examples, we use integers that are sequential multiples of 10 as keys, although there is surely no requirement that keys form an arithmetic sequence. □

Although in Example 14.1 we supposed that records were packed as tightly as possible into blocks, it is common to leave some space initially in each block to accomodate new tuples that may be added to a relation. Alternatively, we may accomodate new tuples with overflow blocks, as we suggested in Section 13.8.1.

14.1.2 Dense Indexes

If records are sorted, we can build on them a *dense index*, which is a sequence of blocks holding only the keys of the records and pointers to the records themselves; the pointers are addresses in the sense discussed in Section 13.6. The index blocks of the dense index maintain these keys in the same sorted order as in the file itself. Since keys and pointers presumably take much less space than complete records, we expect to use many fewer blocks for the index than for the file itself. The index is especially advantageous when it, but not the data file, can fit in main memory. Then, by using the index, we can find any record given its search key, with only one disk I/O per lookup.

Example 14.2: Figure 14.2 suggests a dense index on a sorted file. The first index block contains pointers to the first four records (an atypically small number of pointers for one block), the second block has pointers to the next four, and so on. □

The dense index supports queries that ask for records with a given search-key value. Given key value K, we search the index blocks for K, and when we find it, we follow the associated pointer to the record with key K. It might appear that we need to examine every block of the index, or half the blocks of the index, on average, before we find K. However, there are several factors that make the index-based search more efficient than it seems.

1. The number of index blocks is usually small compared with the number of data blocks.

2. Since keys are sorted, we can use binary search to find K. If there are n blocks of the index, we only look at $\log_2 n$ of them.

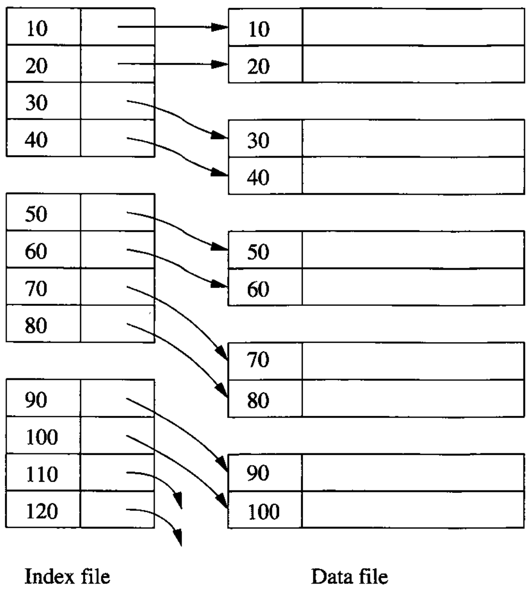

Figure 14.2: A dense index (left) on a sequential data file (right)

3. The index may be small enough to be kept permanently in main memory buffers. If so, the search for key K involves only main-memory accesses, and there are no expensive disk I/O's to be performed.

14.1.3 Sparse Indexes

A sparse index typically has only one key-pointer pair per block of the data file. It thus uses less space than a dense index, at the expense of somewhat more time to find a record given its key. You can only use a sparse index if the data file is sorted by the search key, while a dense index can be used for any search key. Figure 14.3 shows a sparse index with one key-pointer per data block. The keys are for the first records on each data block.

Example 14.3: As in Example 14.2, we assume that the data file is sorted, and keys are all the integers divisible by 10, up to some large number. We also continue to assume that four key-pointer pairs fit on an index block. Thus, the first sparse-index block has entries for the first keys on the first four blocks, which are 10, 30, 50, and 70. Continuing the assumed pattern of keys, the second index block has the first keys of the fifth through eighth blocks, which we assume are 90, 110, 130, and 150. We also show a third index block with first keys from the hypothetical ninth through twelfth data blocks. □

To find the record with search-key value K, we search the sparse index for the largest key less than or equal to K. Since the index file is sorted by key, a

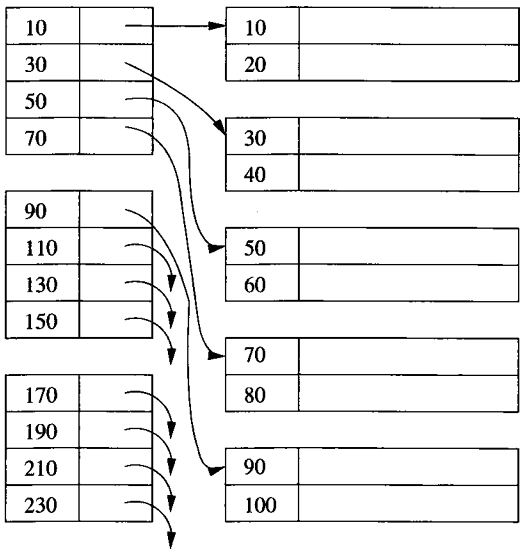

Figure 14.3: A sparse index on a sequential file

binary search can locate this entry. We follow the associated pointer to a data block. Now, we must search this block for the record with key K. Of course the block must have enough format information that the records and their contents can be identified. Any of the techniques from Sections 13.5 and 13.7 can be used.

14.1.4 Multiple Levels of Index

An index file can cover many blocks. Even if we use binary search to find the desired index entry, we still may need to do many disk I/O's to get to the record we want. By putting an index on the index, we can make the use of the first level of index more efficient.

Figure 14.4 extends Fig. 14.3 by adding a second index level (as before, we assume keys are every multiple of 10). The same idea would let us place a third-level index on the second level, and so on. However, this idea has its limits, and we prefer the B-tree structure described in Section 14.2 over building many levels of index.

In this example, the first-level index is sparse, although we could have chosen a dense index for the first level. However, the second and higher levels must be sparse. The reason is that a dense index on an index would have exactly as many key-pointer pairs as the first-level index, and therefore would take exactly as much space as the first-level index.

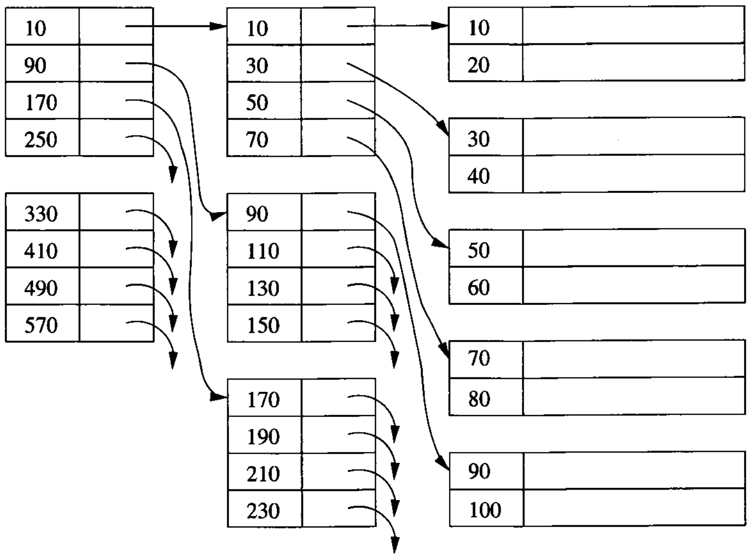

Figure 14.4: Adding a second level of sparse index

14.1.5 Secondary Indexes

A secondary index serves the purpose of any index: it is a data structure that facilitates finding records given a value for one or more fields. However, the secondary index is distinguished from the primary index in that a secondary index does not determine the placement of records in the data file. Rather, the secondary index tells us the current locations of records; that location may have been decided by a primary index on some other field. An important consequence of the distinction between primary and secondary indexes is that:

- Secondary indexes are always dense. It makes no sense to talk of a sparse, secondary index. Since the secondary index does not influence location, we could not use it to predict the location of any record whose key was not mentioned in the index file explicitly.

Example 14.4: Figure 14.5 shows a typical secondary index. The data file is shown with two records per block, as has been our standard for illustration. The records have only their search key shown; this attribute is integer valued, and as before we have taken the values to be multiples of 10. Notice that, unlike the data file in Fig. 14.2, here the data is not sorted by the search key.

However, the keys in the index file *are* sorted. The result is that the pointers in one index block can go to many different data blocks, instead of one or a few consecutive blocks. For example, to retrieve all the records with search key 20, we not only have to look at two index blocks, but we are sent by their pointers to three different data blocks. Thus, using a secondary index may result in

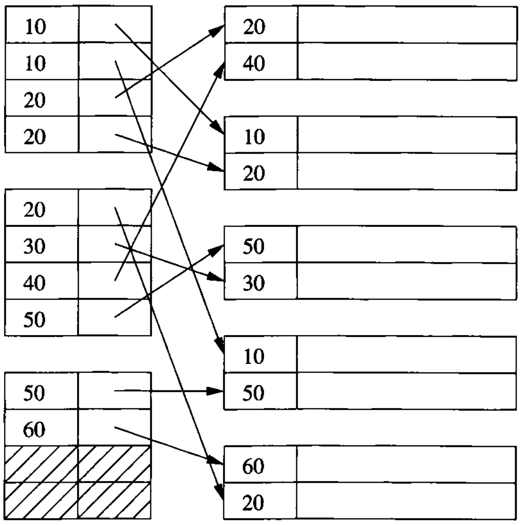

Figure 14.5: A secondary index

many more disk I/O's than if we get the same number of records via a primary index. However, there is no help for this problem; we cannot control the order of tuples in the data block, because they are presumably ordered according to some other attribute(s). □

14.1.6 Applications of Secondary Indexes

Besides supporting additional indexes on relations that are organized as sequential files, there are some data structures where secondary indexes are needed for even the primary key. One of these is the "heap" structure, where the records of the relation are kept in no particular order.

A second common structure needing secondary indexes is the *clustered file*. Suppose there are relations R and S, with a many-one relationship from the tuples of R to tuples of S. It may make sense to store each tuple of R with the tuple of S to which it is related, rather than according to the primary key of R. An example will illustrate why this organization makes good sense in special situations.

Example 14.5: Consider our standard movie and studio relations:

```
Movie(title, year, length, genre, studioName, producerC#)
Studio(name, address, presC#)
```

Suppose further that the most common form of query is:

```
SELECT title, year
FROM Movie, Studio
WHERE presC# = zzz AND Movie.studioName = Studio.name;
```

Here, *zzz* represents any possible certificate number for a studio president. That is, given the president of a studio, we need to find all the movies made by that studio.

If we are convinced that the above query is typical, then instead of ordering `Movie` tuples by the primary key `title` and `year`, we can create a *clustered file structure* for both relations `Studio` and `Movie`, as suggested by Fig. 14.6. Following each `Studio` tuple are all the `Movie` tuples for all the movies owned by that studio.

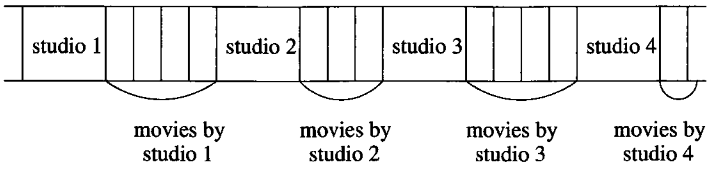

Figure 14.6: A clustered file with each studio clustered with the movies made by that studio

If we create an index for `Studio` with search key `presC#`, then whatever the value of *zzz* is, we can quickly find the tuple for the proper studio. Moreover, all the `Movie` tuples whose value of attribute `studioName` matches the value of `name` for that studio will follow the studio's tuple in the clustered file. As a result, we can find the movies for this studio by making almost as few disk I/O's as possible. The reason is that the desired `Movie` tuples are packed almost as densely as possible onto the following blocks. However, an index on any attribute(s) of `Movie` would have to be a secondary index. □

14.1.7 Indirection in Secondary Indexes

There is some wasted space, perhaps a significant amount of wastage, in the structure suggested by Fig. 14.5. If a search-key value appears n times in the data file, then the value is written n times in the index file. It would be better if we could write the key value once for all the pointers to data records with that value.

A convenient way to avoid repeating values is to use a level of indirection, called ***buckets***, between the secondary index file and the data file. As shown in Fig. 14.7, there is one pair for each search key K. The pointer of this pair goes to a position in a "bucket file," which holds the "bucket" for K. Following this position, until the next position pointed to by the index, are pointers to all the records with search-key value K.

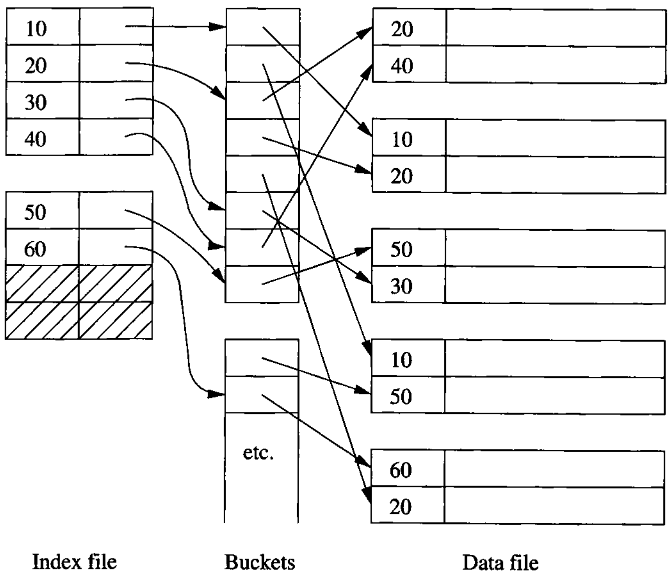

Figure 14.7: Saving space by using indirection in a secondary index

Example 14.6: For instance, let us follow the pointer from search key 50 in the index file of Fig. 14.7 to the intermediate "bucket" file. This pointer happens to take us to the last pointer of one block of the bucket file. We search forward, to the first pointer of the next block. We stop at that point, because the next pointer of the index file, associated with search key 60, points to the next record in the bucket file. □

The scheme of Fig. 14.7 saves space as long as search-key values are larger than pointers, and the average key appears at least twice. However, even if not, there is an important advantage to using indirection with secondary indexes: often, we can use the pointers in the buckets to help answer queries without ever looking at most of the records in the data file. Specifically, when there are several conditions to a query, and each condition has a secondary index to help it, we can find the bucket pointers that satisfy all the conditions by intersecting sets of pointers in memory, and retrieving only the records pointed to by the surviving pointers. We thus save the I/O cost of retrieving records that satisfy some, but not all, of the conditions.[1]

Example 14.7: Consider the usual `Movie` relation:

```
Movie(title, year, length, genre, studioName, producerC#)
```

[1]We also could use this pointer-intersection trick if we got the pointers directly from the index, rather than from buckets.

Suppose we have secondary indexes with indirect buckets on both `studioName` and `year`, and we are asked the query

```
SELECT title
FROM Movie
WHERE studioName = 'Disney' AND year = 2005;
```

that is, find all the Disney movies made in 2005.

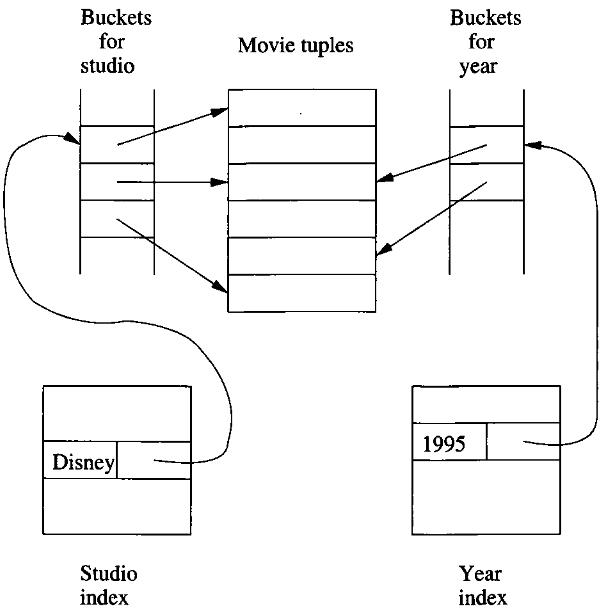

Figure 14.8: Intersecting buckets in main memory

Figure 14.8 shows how we can answer this query using the indexes. Using the index on `studioName`, we find the pointers to all records for Disney movies, but we do not yet bring any of those records from disk to memory. Instead, using the index on `year`, we find the pointers to all the movies of 2005. We then intersect the two sets of pointers, getting exactly the movies that were made by Disney in 2005. Finally, we retrieve from disk all data blocks holding one or more of these movies, thus retrieving the minimum possible number of blocks. □

14.1.8 Document Retrieval and Inverted Indexes

For many years, the information-retrieval community has dealt with the storage of documents and the efficient retrieval of documents with a given set of keywords. With the advent of the World-Wide Web and the feasibility of keeping all documents on-line, the retrieval of documents given keywords has become one of the largest database problems. While there are many kinds of queries that one can use to find relevant documents, the simplest and most common form can be seen in relational terms as follows:

- A document may be thought of as a tuple in a relation `Doc`. This relation has very many attributes, one corresponding to each possible word in a document. Each attribute is boolean — either the word is present in the document, or it is not. Thus, the relation schema may be thought of as

  ```
  Doc(hasCat, hasDog, ... )
  ```

  where `hasCat` is true if and only if the document has the word "cat" at least once.

- There is a secondary index on each of the attributes of `Doc`. However, we save the trouble of indexing those tuples for which the value of the attribute is `FALSE`; instead, the index leads us to only the documents for which the word is present. That is, the index has entries only for the search-key value `TRUE`.

- Instead of creating a separate index for each attribute (i.e., for each word), the indexes are combined into one, called an *inverted index.* This index uses indirect buckets for space efficiency, as was discussed in Section 14.1.7.

Example 14.8 : An inverted index is illustrated in Fig. 14.9. In place of a data file of records is a collection of documents, each of which may be stored on one or more disk blocks. The inverted index itself consists of a set of word-pointer pairs; the words are in effect the search key for the index. The inverted index is kept in a sequence of blocks, just like any of the indexes discussed so far.

The pointers refer to positions in a "bucket" file. For instance, we have shown in Fig. 14.9 the word "cat" with a pointer to the bucket file. That pointer leads us to the beginning of a list of pointers to all the documents that contain the word "cat." We have shown some of these in the figure. Similarly, the word "dog" is shown leading to a list of pointers to all the documents with "dog." □

Pointers in the bucket file can be:

1. Pointers to the document itself.

2. Pointers to an occurrence of the word. In this case, the pointer might be a pair consisting of the first block for the document and an integer indicating the number of the word in the document.

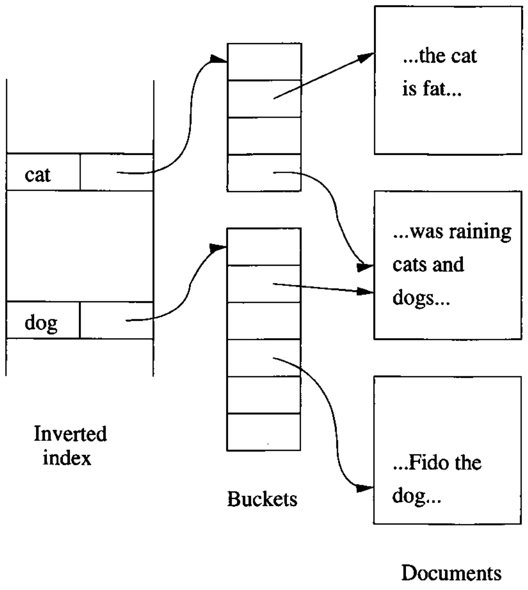

Figure 14.9: An inverted index on documents

When we use "buckets" of pointers to occurrences of each word, we may extend the idea to include in the bucket array some information about each occurrence. Now, the bucket file itself becomes a collection of records with important structure. Early uses of the idea distinguished occurrences of a word in the title of a document, the abstract, and the body of text. With the growth of documents on the Web, especially documents using HTML, XML, or another markup language, we can also indicate the markings associated with words. For instance, we can distinguish words appearing in titles, headers, tables, or anchors, as well as words appearing in different fonts or sizes.

Example 14.9: Figure 14.10 illustrates a bucket file that has been used to indicate occurrences of words in HTML documents. The first column indicates the type of occurrence, i.e., its marking, if any. The second and third columns are together the pointer to the occurrence. The third column indicates the document, and the second column gives the number of the word in the document.

We can use this data structure to answer various queries about documents without having to examine the documents in detail. For instance, suppose we want to find documents about dogs that compare them with cats. Without a deep understanding of the meaning of the text, we cannot answer this query precisely. However, we could get a good hint if we searched for documents that

a) Mention dogs in the title, and

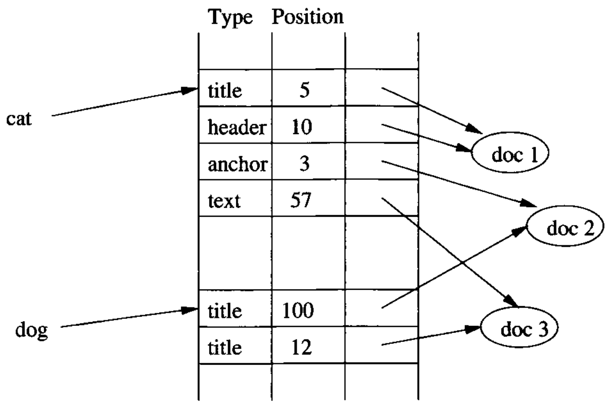

Figure 14.10: Storing more information in the inverted index

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 three records, or ten 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 30 records or 200 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 ten records, either studio records 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 three records, ten key-pointer pairs, or fifty 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 3000 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 14.1.6:** On the assumptions of Exercise 14.1.5(a), what is the average number of disk I/O's to find and retrieve the ten 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: Suppose we have a repository of 1000 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 ith most frequent word is $100000/\sqrt{i}$, for $i = 1, 2, \ldots, 10000$.

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

b) 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?

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

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

Exercise 14.1.8: 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.

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:

- B-trees automatically maintain as many levels of index as is appropriate for the size of the file being indexed.
- B-trees manage the space on the blocks they use so that every block is between half used and completely full.

In the following discussion, we shall talk about "B-trees," but the details will all be for the B+ tree variant. Other types of B-tree are discussed in exercises.

14.2.1 The Structure of B-trees

A B-tree organizes its blocks into a tree that is *balanced*, meaning that all paths from the root to a leaf have the same length. Typically, there are three layers in a B-tree: the root, an intermediate layer, and leaves, but any number of layers is possible. To help visualize B-trees, you may wish to look ahead at Figs. 14.11 and 14.12, which show nodes of a B-tree, and Fig. 14.13, which shows an entire B-tree.

There is a parameter n associated with each B-tree index, and this parameter determines the layout of all blocks of the B-tree. Each block will have space for n search-key values and $n+1$ pointers. In a sense, a B-tree block is similar to the index blocks introduced in Section 14.1.2, except that the B-tree block has an extra pointer, along with n key-pointer pairs. We pick n to be as large as will allow $n+1$ pointers and n keys to fit in one block.

Example 14.10: Suppose our blocks are 4096 bytes. Also let keys be integers of 4 bytes and let pointers be 8 bytes. If there is no header information kept on the blocks, then we want to find the largest integer value of n such that $4n + 8(n+1) \leq 4096$. That value is $n = 340$. □

There are several important rules about what can appear in the blocks of a B-tree:

- The keys in leaf nodes are copies of keys from the data file. These keys are distributed among the leaves in sorted order, from left to right.
- At the root, there are at least two used pointers.[2] All pointers point to B-tree blocks at the level below.
- At a leaf, the last pointer points to the next leaf block to the right, i.e., to the block with the next higher keys. Among the other n pointers in a leaf block, at least $\lfloor (n+1)/2 \rfloor$ of these pointers are used and point to data records; unused pointers are null and do not point anywhere. The ith pointer, if it is used, points to a record with the ith key.

[2]Technically, there is a possibility that the entire B-tree has only one pointer because it is an index into a data file with only one record. In this case, the entire tree is a root block that is also a leaf, and this block has only one key and one pointer. We shall ignore this trivial case in the descriptions that follow.

- At an interior node, all $n + 1$ pointers can be used to point to B-tree blocks at the next lower level. At least $\lceil (n + 1)/2 \rceil$ of them are actually used (but if the node is the root, then we require only that at least 2 be used, regardless of how large n is). If j pointers are used, then there will be $j - 1$ keys, say $K_1, K_2, \ldots, K_{j-1}$. The first pointer points to a part of the B-tree where some of the records with keys less than K_1 will be found. The second pointer goes to that part of the tree where all records with keys that are at least K_1, but less than K_2 will be found, and so on. Finally, the jth pointer gets us to the part of the B-tree where some of the records with keys greater than or equal to K_{j-1} are found. Note that some records with keys far below K_1 or far above K_{j-1} may not be reachable from this block at all, but will be reached via another block at the same level.

- All used pointers and their keys appear at the beginning of the block, with the exception of the $(n + 1)$st pointer in a leaf, which points to the next leaf.

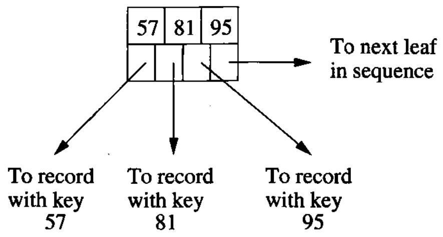

Figure 14.11: A typical leaf of a B-tree

Example 14.11: Our running example of B-trees will use $n = 3$. That is, blocks have room for three keys and four pointers, which are atypically small numbers. Keys are integers. Figure 14.11 shows a leaf that is completely used. There are three keys, 57, 81, and 95. The first three pointers go to records with these keys. The last pointer, as is always the case with leaves, points to the next leaf to the right in the order of keys; it would be null if this leaf were the last in sequence.

A leaf is not necessarily full, but in our example with $n = 3$, there must be at least two key-pointer pairs. That is, the key 95 in Fig. 14.11 might be missing, and if so, the third pointer would be null.

Figure 14.12 shows a typical interior node. There are three keys, 14, 52, and 78. There are also four pointers in this node. The first points to a part of the B-tree from which we can reach only records with keys less than 14 — the first of the keys. The second pointer leads to all records with keys between the first and second keys of the B-tree block; the third pointer is for those records

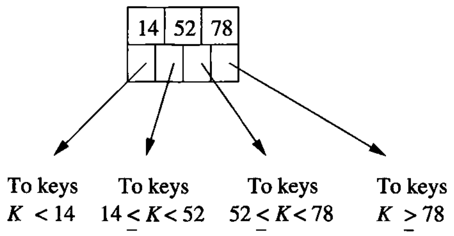

Figure 14.12: A typical interior node of a B-tree

between the second and third keys of the block, and the fourth pointer lets us reach some of the records with keys equal to or above the third key of the block.

As with our example leaf, it is not necessarily the case that all slots for keys and pointers are occupied. However, with $n = 3$, at least the first key and the first two pointers must be present in an interior node. □

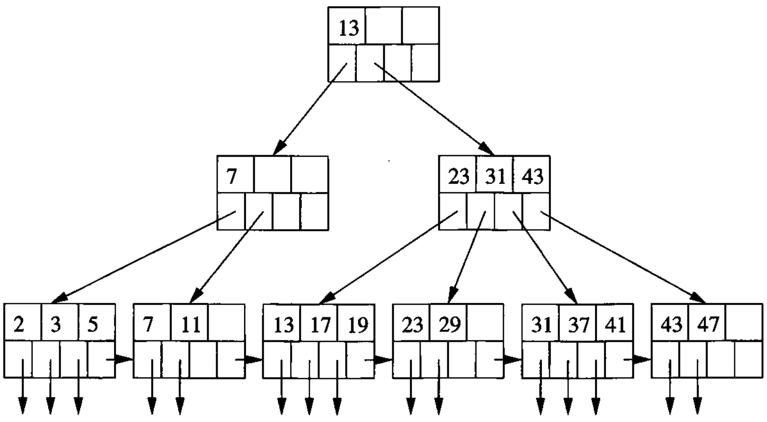

Figure 14.13: A B-tree

Example 14.12: Figure 14.13 shows an entire three-level B-tree, with $n = 3$, as in Example 14.11. We have assumed that the data file consists of records whose keys are all the primes from 2 to 47. Notice that at the leaves, each of these keys appears once, in order. All leaf blocks have two or three key-pointer pairs, plus a pointer to the next leaf in sequence. The keys are in sorted order as we look across the leaves from left to right.

The root has only two pointers, the minimum possible number, although it could have up to four. The one key at the root separates those keys reachable via the first pointer from those reachable via the second. That is, keys up to 12 could be found in the first subtree of the root, and keys 13 and up are in the second subtree.

If we look at the first child of the root, with key 7, we again find two pointers, one to keys less than 7 and the other to keys 7 and above. Note that the second pointer in this node gets us only to keys 7 and 11, not to *all* keys ≥ 7, such as 13.

Finally, the second child of the root has all four pointer slots in use. The first gets us to some of the keys less than 23, namely 13, 17, and 19. The second pointer gets us to all keys K such that $23 \leq K < 31$; the third pointer lets us reach all keys K such that $31 \leq K < 43$, and the fourth pointer gets us to some of the keys ≥ 43 (in this case, to all of them). □

14.2.2 Applications of B-trees

The B-tree is a powerful tool for building indexes. The sequence of pointers at the leaves of a B-tree can play the role of any of the pointer sequences coming out of an index file that we learned about in Section 14.1. Here are some examples:

1. The search key of the B-tree is the primary key for the data file, and the index is dense. That is, there is one key-pointer pair in a leaf for every record of the data file. The data file may or may not be sorted by primary key.

2. The data file is sorted by its primary key, and the B-tree is a sparse index with one key-pointer pair at a leaf for each block of the data file.

3. The data file is sorted by an attribute that is not a key, and this attribute is the search key for the B-tree. For each key value K that appears in the data file there is one key-pointer pair at a leaf. That pointer goes to the first of the records that have K as their sort-key value.

There are additional applications of B-tree variants that allow multiple occurrences of the search key[3] at the leaves. Figure 14.14 suggests what such a B-tree might look like.

If we do allow duplicate occurrences of a search key, then we need to change slightly the definition of what the keys at interior nodes mean, which we discussed in Section 14.2.1. Now, suppose there are keys $K_1, K_2, \ldots, K_n$ at an interior node. Then K_i will be the smallest new key that appears in the part of the subtree accessible from the $(i+1)$st pointer. By "new," we mean that there are no occurrences of K_i in the portion of the tree to the left of the $(i+1)$st subtree, but at least one occurrence of K_i in that subtree. Note that in some situations, there will be no such key, in which case K_i can be taken to be null. Its associated pointer is still necessary, as it points to a significant portion of the tree that happens to have only one key value within it.

[3]Remember that a "search key" is not necessarily a "key" in the sense of being unique.

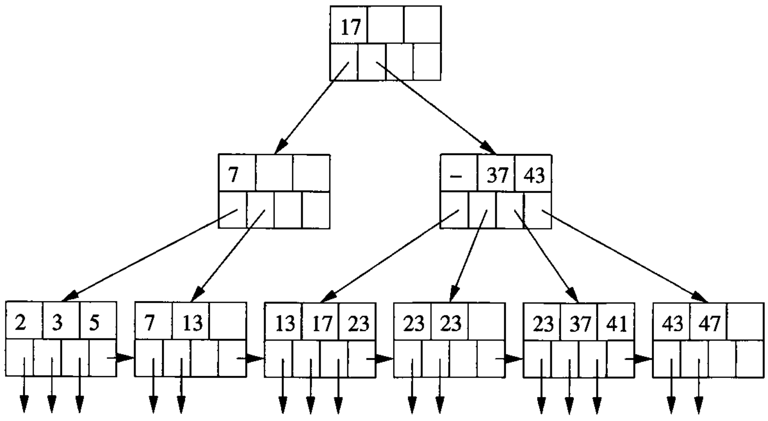

Figure 14.14: A B-tree with duplicate keys

Example 14.13: Figure 14.14 shows a B-tree similar to Fig. 14.13, but with duplicate values. In particular, key 11 has been replaced by 13, and keys 19, 29, and 31 have all been replaced by 23. As a result, the key at the root is 17, not 13. The reason is that, although 13 is the lowest key in the second subtree of the root, it is not a *new* key for that subtree, since it also appears in the first subtree.

We also had to make some changes to the second child of the root. The second key is changed to 37, since that is the first new key of the third child (fifth leaf from the left). Most interestingly, the first key is now null. The reason is that the second child (fourth leaf) has no new keys at all. Put another way, if we were searching for any key and reached the second child of the root, we would never want to start at its second child. If we are searching for 23 or anything lower, we want to start at its first child, where we will either find what we are looking for (if it is 17), or find the first of what we are looking for (if it is 23). Note that:

- We would not reach the second child of the root searching for 13; we would be directed at the root to its first child instead.

- If we are looking for any key between 24 and 36, we are directed to the third leaf, but when we don't find even one occurrence of what we are looking for, we know not to search further right. For example, if there were a key 24 among the leaves, it would either be on the 4th leaf, in which case the null key in the second child of the root would be 24 instead, or it would be in the 5th leaf, in which case the key 37 at the second child of the root would be 24.

□

14.2.3 Lookup in B-Trees

We now revert to our original assumption that there are no duplicate keys at the leaves. We also suppose that the B-tree is a dense index, so every search-key value that appears in the data file will also appear at a leaf. These assumptions make the discussion of B-tree operations simpler, but is not essential for these operations. In particular, modifications for sparse indexes are similar to the changes we introduced in Section 14.1.3 for indexes on sequential files.

Suppose we have a B-tree index and we want to find a record with search-key value K. We search for K recursively, starting at the root and ending at a leaf. The search procedure is:

BASIS: If we are at a leaf, look among the keys there. If the ith key is K, then the ith pointer will take us to the desired record.

INDUCTION: If we are at an interior node with keys $K_1, K_2, \ldots, K_n$, follow the rules given in Section 14.2.1 to decide which of the children of this node should next be examined. That is, there is only one child that could lead to a leaf with key K. If $K < K_1$, then it is the first child, if $K_1 \leq K < K_2$, it is the second child, and so on. Recursively apply the search procedure at this child.

Example 14.14: Suppose we have the B-tree of Fig. 14.13, and we want to find a record with search key 40. We start at the root, where there is one key, 13. Since $13 \leq 40$, we follow the second pointer, which leads us to the second-level node with keys 23, 31, and 43.

At that node, we find $31 \leq 40 < 43$, so we follow the third pointer. We are thus led to the leaf with keys 31, 37, and 41. If there had been a record in the data file with key 40, we would have found key 40 at this leaf. Since we do not find 40, we conclude that there is no record with key 40 in the underlying data.

Note that had we been looking for a record with key 37, we would have taken exactly the same decisions, but when we got to the leaf we would find key 37. Since it is the second key in the leaf, we follow the second pointer, which will lead us to the data record with key 37. □

14.2.4 Range Queries

B-trees are useful not only for queries in which a single value of the search key is sought, but for queries in which a range of values are asked for. Typically, *range queries* have a term in the WHERE-clause that compares the search key with a value or values, using one of the comparison operators other than = or <>. Examples of range queries using a search-key attribute k are:

```
SELECT * FROM R   SELECT * FROM R
WHERE R.k > 40;   WHERE R.k >= 10 AND R.k <= 25;
```

If we want to find all keys in the range $[a, b]$ at the leaves of a B-tree, we do a lookup to find the key a. Whether or not it exists, we are led to a leaf where

a could be, and we search the leaf for keys that are a or greater. Each such key we find has an associated pointer to one of the records whose key is in the desired range. As long as we do not find a key greater than b in the current block, we follow the pointer to the next leaf and repeat our search for keys in the range $[a, b]$.

The above search algorithm also works if b is infinite; i.e., there is only a lower bound and no upper bound. In that case, we search all the leaves from the one that would hold key a to the end of the chain of leaves. If a is $-\infty$ (that is, there is an upper bound on the range but no lower bound), then the search for "minus infinity" as a search key will always take us to the first leaf. The search then proceeds as above, stopping only when we pass the key b.

Example 14.15: Suppose we have the B-tree of Fig. 14.13, and we are given the range $(10, 25)$ to search for. We look for key 10, which leads us to the second leaf. The first key is less than 10, but the second, 11, is at least 10. We follow its associated pointer to get the record with key 11.

Since there are no more keys in the second leaf, we follow the chain to the third leaf, where we find keys 13, 17, and 19. All are less than or equal to 25, so we follow their associated pointers and retrieve the records with these keys. Finally, we move to the fourth leaf, where we find key 23. But the next key of that leaf, 29, exceeds 25, so we are done with our search. Thus, we have retrieved the five records with keys 11 through 23. □

14.2.5 Insertion Into B-Trees

We see some of the advantages of B-trees over simpler multilevel indexes when we consider how to insert a new key into a B-tree. The corresponding record will be inserted into the file being indexed by the B-tree, using any of the methods discussed in Section 14.1; here we consider how the B-tree changes. The insertion is, in principle, recursive:

- We try to find a place for the new key in the appropriate leaf, and we put it there if there is room.
- If there is no room in the proper leaf, we split the leaf into two and divide the keys between the two new nodes, so each is half full or just over half full.
- The splitting of nodes at one level appears to the level above as if a new key-pointer pair needs to be inserted at that higher level. We may thus recursively apply this strategy to insert at the next level: if there is room, insert it; if not, split the parent node and continue up the tree.
- As an exception, if we try to insert into the root, and there is no room, then we split the root into two nodes and create a new root at the next higher level; the new root has the two nodes resulting from the split as its children. Recall that no matter how large n (the number of slots for

keys at a node) is, it is always permissible for the root to have only one key and two children.

When we split a node and insert it into its parent, we need to be careful how the keys are managed. First, suppose N is a leaf whose capacity is n keys. Also suppose we are trying to insert an $(n+1)$st key and its associated pointer. We create a new node M, which will be the sibling of N, immediately to its right. The first $\lceil (n+1)/2 \rceil$ key-pointer pairs, in sorted order of the keys, remain with N, while the other key-pointer pairs move to M. Note that both nodes N and M are left with a sufficient number of key-pointer pairs — at least $\lfloor (n+1)/2 \rfloor$ pairs.

Now, suppose N is an interior node whose capacity is n keys and $n+1$ pointers, and N has just been assigned $n+2$ pointers because of a node splitting below. We do the following:

1. Create a new node M, which will be the sibling of N, immediately to its right.

2. Leave at N the first $\lceil (n+2)/2 \rceil$ pointers, in sorted order, and move to M the remaining $\lfloor (n+2)/2 \rfloor$ pointers.

3. The first $\lceil n/2 \rceil$ keys stay with N, while the last $\lfloor n/2 \rfloor$ keys move to M. Note that there is always one key in the middle left over; it goes with neither N nor M. The leftover key K indicates the smallest key reachable via the first of M's children. Although this key doesn't appear in N or M, it is associated with M, in the sense that it represents the smallest key reachable via M. Therefore K will be inserted into the parent of N and M to divide searches between those two nodes.

Example 14.16: Let us insert key 40 into the B-tree of Fig. 14.13. We find the proper leaf for the insertion by the lookup procedure of Section 14.2.3. As found in Example 14.14, the insertion goes into the fifth leaf. Since this leaf now has four key-pointer pairs — 31, 37, 40, and 41 — we need to split the leaf. Our first step is to create a new node and move the highest two keys, 40 and 41, along with their pointers, to that node. Figure 14.15 shows this split.

Notice that although we now show the nodes on four ranks to save space, there are still only three levels to the tree. The seven leaves are linked by their last pointers, which still form a chain from left to right.

We must now insert a pointer to the new leaf (the one with keys 40 and 41) into the node above it (the node with keys 23, 31, and 43). We must also associate with this pointer the key 40, which is the least key reachable through the new leaf. Unfortunately, the parent of the split node is already full; it has no room for another key or pointer. Thus, it too must be split.

We start with pointers to the last five leaves and the list of keys representing the least keys of the last four of these leaves. That is, we have pointers P_1, P_2, P_3, P_4, P_5 to the leaves whose least keys are 13, 23, 31, 40, and 43, and

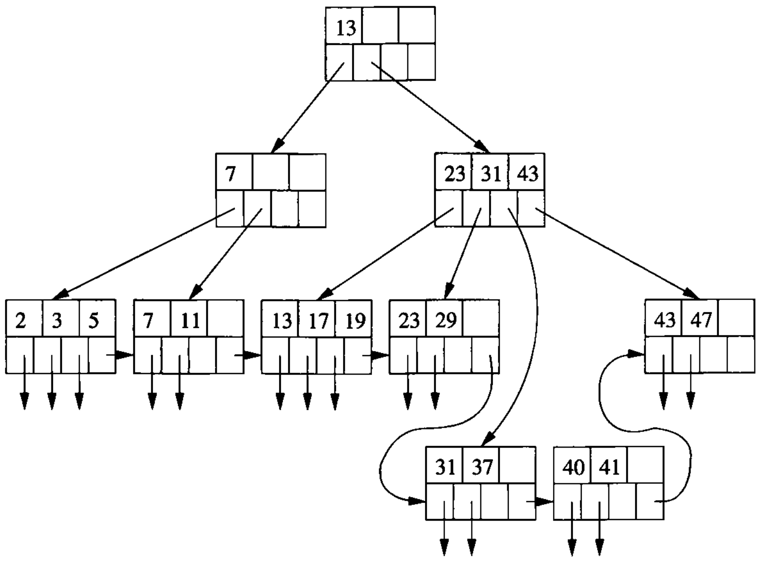

Figure 14.15: Beginning the insertion of key 40

we have the key sequence 23, 31, 40, 43 to separate these pointers. The first three pointers and first two keys remain with the split interior node, while the last two pointers and last key go to the new node. The remaining key, 40, represents the least key accessible via the new node.

Figure 14.16 shows the completion of the insert of key 40. The root now has three children; the last two are the split interior node. Notice that the key 40, which marks the lowest of the keys reachable via the second of the split nodes, has been installed in the root to separate the keys of the root's second and third children. □

14.2.6 Deletion From B-Trees

If we are to delete a record with a given key K, we must first locate that record and its key-pointer pair in a leaf of the B-tree. This part of the deletion process is essentially a lookup, as in Section 14.2.3. We then delete the record itself from the data file, and we delete the key-pointer pair from the B-tree.

If the B-tree node from which a deletion occurred still has at least the minimum number of keys and pointers, then there is nothing more to be done.[4] However, it is possible that the node was right at the minimum occupancy before the deletion, so after deletion the constraint on the number of keys is

[4]If the data record with the least key at a leaf is deleted, then we have the option of raising the appropriate key at one of the ancestors of that leaf, but there is no requirement that we do so; all searches will still go to the appropriate leaf.

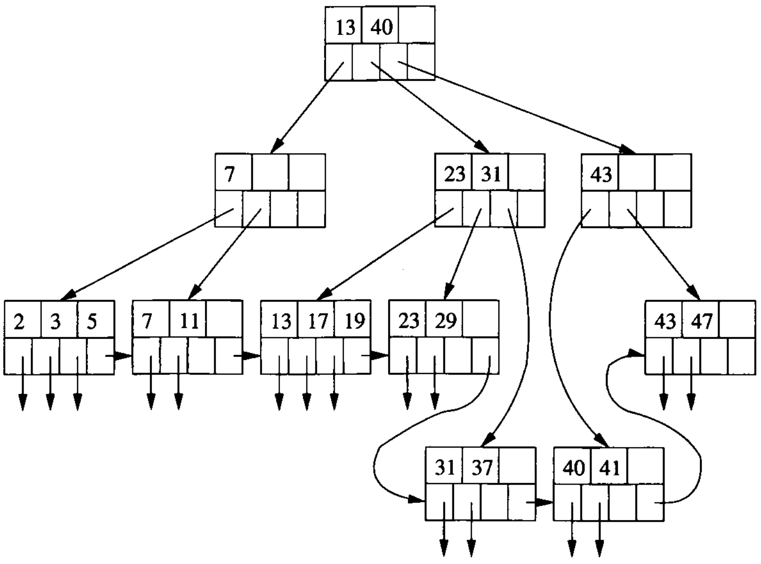

Figure 14.16: Completing the insertion of key 40

violated. We then need to do one of two things for a node N whose contents are subminimum; one case requires a recursive deletion up the tree:

1. If one of the adjacent siblings of node N has more than the minimum number of keys and pointers, then one key-pointer pair can be moved to N, keeping the order of keys intact. Possibly, the keys at the parent of N must be adjusted to reflect the new situation. For instance, if the right sibling of N, say node M, provides an extra key and pointer, then it must be the smallest key that is moved from M to N. At the parent of M and N, there is a key that represents the smallest key accessible via M; that key must be increased to reflect the new M.

2. The hard case is when neither adjacent sibling can be used to provide an extra key for N. However, in that case, we have two adjacent nodes, N and a sibling M; the latter has the minimum number of keys and the former has fewer than the minimum. Therefore, together they have no more keys and pointers than are allowed in a single node. We merge these two nodes, effectively deleting one of them. We need to adjust the keys at the parent, and then delete a key and pointer at the parent. If the parent is still full enough, then we are done. If not, then we recursively apply the deletion algorithm at the parent.

Example 14.17: Let us begin with the original B-tree of Fig. 14.13, before the insertion of key 40. Suppose we delete key 7. This key is found in the second leaf. We delete it, its associated pointer, and the record that pointer points to.

The second leaf now has only one key, and we need at least two in every leaf. But we are saved by the sibling to the left, the first leaf, because that leaf has an extra key-pointer pair. We may therefore move the highest key, 5, and its associated pointer to the second leaf. The resulting B-tree is shown in Fig. 14.17. Notice that because the lowest key in the second leaf is now 5, the key in the parent of the first two leaves has been changed from 7 to 5.

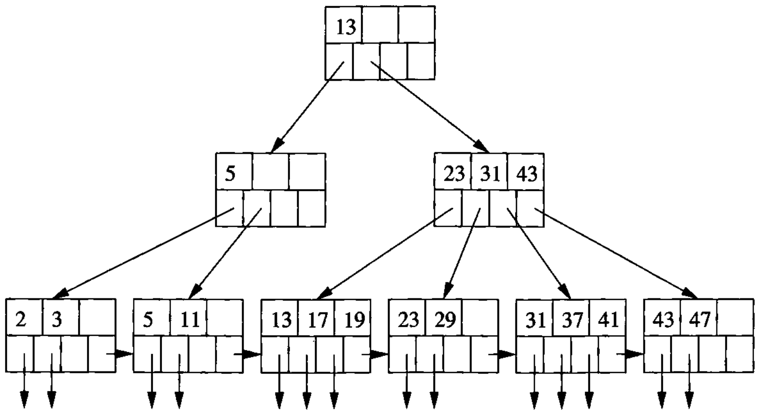

Figure 14.17: Deletion of key 7

Next, suppose we delete key 11. This deletion has the same effect on the second leaf; it again reduces the number of its keys below the minimum. This time, however, we cannot take a key from the first leaf, because the latter is down to the minimum number of keys. Additionally, there is no sibling to the right from which to take a key.[5] Thus, we need to merge the second leaf with a sibling, namely the first leaf.

The three remaining key-pointer pairs from the first two leaves fit in one leaf, so we move 5 to the first leaf and delete the second leaf. The pointers and keys in the parent are adjusted to reflect the new situation at its children; specifically, the two pointers are replaced by one (to the remaining leaf) and the key 5 is no longer relevant and is deleted. The situation is now as shown in Fig. 14.18.

The deletion of a leaf has adversely affected the parent, which is the left child of the root. That node, as we see in Fig. 14.18, now has no keys and only one pointer. Thus, we try to obtain an extra key and pointer from an adjacent sibling. This time we have the easy case, since the other child of the root can afford to give up its smallest key and a pointer.

The change is shown in Fig. 14.19. The pointer to the leaf with keys 13, 17,

[5]Notice that the leaf to the right, with keys 13, 17, and 19, is not a sibling, because it has a different parent. We could take a key from that node anyway, but then the algorithm for adjusting keys throughout the tree becomes more complex. We leave this enhancement as an exercise.

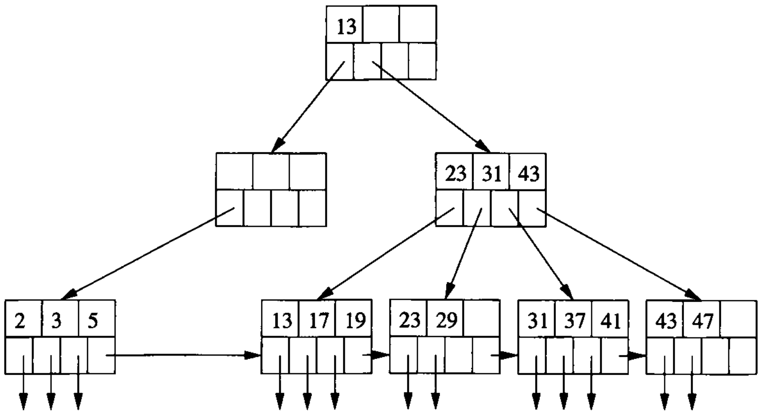

Figure 14.18: Beginning the deletion of key 11

and 19 has been moved from the second child of the root to the first child. We have also changed some keys at the interior nodes. The key 13, which used to reside at the root and represented the smallest key accessible via the pointer that was transferred, is now needed at the first child of the root. On the other hand, the key 23, which used to separate the first and second children of the second child of the root now represents the smallest key accessible from the second child of the root. It therefore is placed at the root itself. □

14.2.7 Efficiency of B-Trees

B-trees allow lookup, insertion, and deletion of records using very few disk I/O's per file operation. First, we should observe that if n, the number of keys per block, is reasonably large, then splitting and merging of blocks will be rare events. Further, when such an operation is needed, it almost always is limited to the leaves, so only two leaves and their parent are affected. Thus, we can essentially neglect the disk-I/O cost of B-tree reorganizations.

However, every search for the record(s) with a given search key requires us to go from the root down to a leaf, to find a pointer to the record. Since we are only reading B-tree blocks, the number of disk I/O's will be the number of levels the B-tree has, plus the one (for lookup) or two (for insert or delete) disk I/O's needed for manipulation of the record itself. We must thus ask: how many levels does a B-tree have? For the typical sizes of keys, pointers, and blocks, three levels are sufficient for all but the largest databases. Thus, we shall generally take 3 as the number of levels of a B-tree. The following example illustrates why.

Example 14.18: Recall our analysis in Example 14.10, where we determined that 340 key-pointer pairs could fit in one block for our example data. Suppose

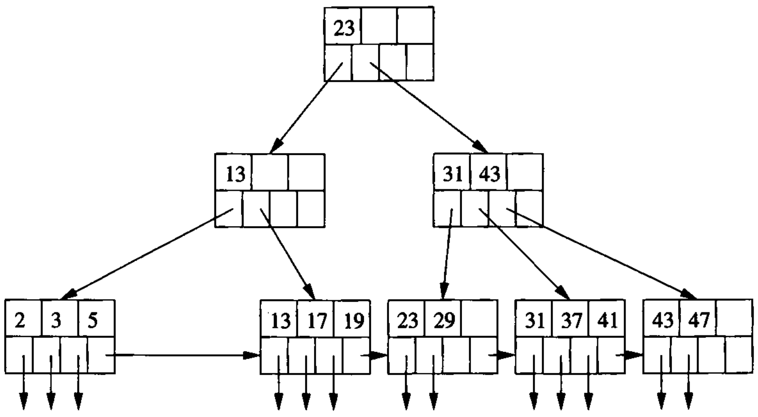

Figure 14.19: Completing the deletion of key 11

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 1,000,000-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 10 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 10 to a block.

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) 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.

e) 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.

Exercise 14.2.2: Repeat Exercise 14.2.1 in the case that the query is a range query that is matched by 1000 records.

Exercise 14.2.3: Suppose pointers are 4 bytes long, and keys are 12 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) $n = 10$; i.e., a block holds 10 keys and 11 pointers.

b) $n = 11$; i.e., a block holds 11 keys and 12 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 41.

b) Lookup the record with key 40.

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

d) Lookup all records with keys less than 30.

e) Lookup all records with keys greater than 30.

f) Insert a record with key 1.

g) Insert records with keys 14 through 16.

h) Delete the record with key 23.

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

Exercise 14.2.6: 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.

! **Exercise 14.2.7:** 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 14.2.8:** 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 14.2.9:** 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?

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 $h(K)$.

14.3.1 Secondary-Storage Hash Tables

A hash table that holds a very large number of records, so many that they must be kept mainly in secondary storage, differs from the main-memory version in small but important ways. First, the bucket array consists of blocks, rather than pointers to the headers of lists. Records that are hashed by the hash function h to a certain bucket are put in the block for that bucket. If a bucket has too many records, a chain of overflow blocks can be added to the bucket to hold more records.

We shall assume that the location of the first block for any bucket i can be found given i. For example, there might be a main-memory array of pointers to blocks, indexed by the bucket number. Another possibility is to put the first block for each bucket in fixed, consecutive disk locations, so we can compute the location of bucket i from the integer i.

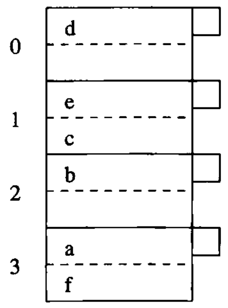

Figure 14.20: A hash table

Example 14.19: Figure 14.20 shows a hash table. To keep our illustrations manageable, we assume that a block can hold only two records, and that $B = 4$; i.e., the hash function h returns values from 0 to 3. We show certain records populating the hash table. Keys are letters a through f in Fig. 14.20. We assume that $h(d) = 0$, $h(c) = h(e) = 1$, $h(b) = 2$, and $h(a) = h(f) = 3$. Thus, the six records are distributed into blocks as shown. □

Note that we show each block in Fig. 14.20 with a "nub" at the right end. This nub represents additional information in the block's header. We shall use it to chain overflow blocks together, and starting in Section 14.3.5, we shall use it to keep other critical information about the block.

14.3.2 Insertion Into a Hash Table

When a new record with search key K must be inserted, we compute $h(K)$. If the bucket numbered $h(K)$ has space, then we insert the record into the block for this bucket, or into one of the overflow blocks on its chain if there is no room

Choice of Hash Function

The hash function should "hash" the key so the resulting integer is a seemingly random function of the key. Thus, buckets will tend to have equal numbers of records, which improves the average time to access a record, as we shall discuss in Section 14.3.4. Also, the hash function should be easy to compute, since we shall compute it many times.

A common choice of hash function when keys are integers is to compute the remainder of K/B, where K is the key value and B is the number of buckets. Often, B is chosen to be a prime, although there are reasons to make B a power of 2, as we discuss starting in Section 14.3.5. For character-string search keys, we may treat each character as an integer, sum these integers, and take the remainder when the sum is divided by B.

in the first block. If none of the blocks of the chain for bucket $h(K)$ has room, we add a new overflow block to the chain and store the new record there.

Example 14.20: Suppose we add to the hash table of Fig. 14.20 a record with key g, and $h(g) = 1$. Then we must add the new record to the bucket numbered 1. However, the block for that bucket already has two records. Thus, we add a new block and chain it to the original block for bucket 1. The record with key g goes in that block, as shown in Fig. 14.21. □

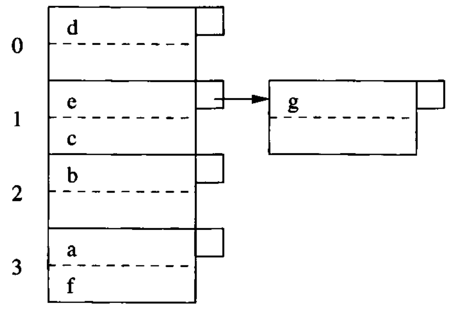

Figure 14.21: Adding an additional block to a hash-table bucket

14.3.3 Hash-Table Deletion

Deletion of the record (or records) with search key K follows the same pattern as insertion. We go to the bucket numbered $h(K)$ and search for records with that search key. Any that we find are deleted. If we are able to move records around among blocks, then after deletion we may optionally consolidate the blocks of a bucket into one fewer block.[6]

Example 14.21 : Figure 14.22 shows the result of deleting the record with key c from the hash table of Fig. 14.21. Recall $h(c) = 1$, so we go to the bucket numbered 1 (i.e., the second bucket) and search all its blocks to find a record (or records if the search key were not the primary key) with key c. We find it in the first block of the chain for bucket 1. Since there is now room to move the record with key g from the second block of the chain to the first, we can do so and remove the second block.

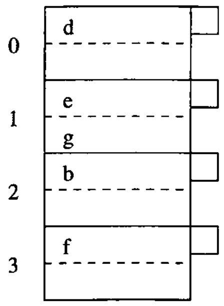

Figure 14.22: Result of deletions from a hash table

We also show the deletion of the record with key a. For this key, we found our way to bucket 3, deleted it, and "consolidated" the remaining record at the beginning of the block. □

14.3.4 Efficiency of Hash Table Indexes

Ideally, there are enough buckets that most of them fit on one block. If so, then the typical lookup takes only one disk I/O, and insertion or deletion from the file takes only two disk I/O's. That number is significantly better than straightforward sparse or dense indexes, or B-tree indexes (although hash tables do not support range queries as B-trees do; see Section 14.2.4).

However, if the file grows, then we shall eventually reach a situation where there are many blocks in the chain for a typical bucket. If so, then we need to search long lists of blocks, taking at least one disk I/O per block. Thus, there is a good reason to try to keep the number of blocks per bucket low.

The hash tables we have examined so far are called ***static hash tables***, because B, the number of buckets, never changes. However, there are several kinds of ***dynamic hash tables***, where B is allowed to vary so it approximates the number

[6]A risk of consolidating blocks of a chain whenever possible is that an oscillation, where we alternately insert and delete records from a bucket, will cause a block to be created or destroyed at each step.

of records divided by the number of records that can fit on a block; i.e., there is about one block per bucket. We shall discuss two such methods:

1. Extensible hashing in Section 14.3.5, and
2. Linear hashing in Section 14.3.7.

The first grows B by doubling it whenever it is deemed too small, and the second grows B by 1 each time statistics of the file suggest some growth is needed.

14.3.5 Extensible Hash Tables

Our first approach to dynamic hashing is called *extensible hash tables.* The major additions to the simpler static hash table structure are:

1. There is a level of indirection for the buckets. That is, an array of pointers to blocks represents the buckets, instead of the array holding the data blocks themselves.
2. The array of pointers can grow. Its length is always a power of 2, so in a growing step the number of buckets doubles.
3. However, there does not have to be a data block for each bucket; certain buckets can share a block if the total number of records in those buckets can fit in the block.
4. The hash function h computes for each key a sequence of k bits for some large k, say 32. However, the bucket numbers will at all times use some smaller number of bits, say i bits, from the beginning or end of this sequence. The bucket array will have 2^i entries when i is the number of bits used.

Example 14.22: Figure 14.23 shows a small extensible hash table. We suppose, for simplicity of the example, that $k = 4$; i.e., the hash function produces a sequence of only four bits. At the moment, only one of these bits is used, as indicated by $i = 1$ in the box above the bucket array. The bucket array therefore has only two entries, one for 0 and one for 1.

The bucket array entries point to two blocks. The first holds all the current records whose search keys hash to a bit sequence that begins with 0, and the second holds all those whose search keys hash to a sequence beginning with 1. For convenience, we show the keys of records as if they were the entire bit sequence to which the hash function converts them. Thus, the first block holds a record whose key hashes to 0001, and the second holds records whose keys hash to 1001 and 1100. □

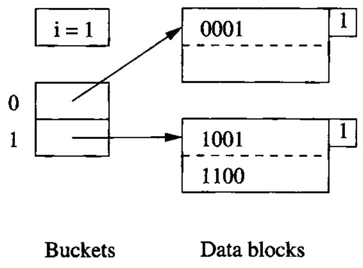

Figure 14.23: An extensible hash table

We should notice the number 1 appearing in the "nub" of each of the blocks in Fig. 14.23. This number, which would actually appear in the block header, indicates how many bits of the hash function's sequence is used to determine membership of records in this block. In the situation of Example 14.22, there is only one bit considered for all blocks and records, but as we shall see, the number of bits considered for various blocks can differ as the hash table grows. That is, the bucket array size is determined by the maximum number of bits we are now using, but some blocks may use fewer.

14.3.6 Insertion Into Extensible Hash Tables

Insertion into an extensible hash table begins like insertion into a static hash table. To insert a record with search key K, we compute $h(K)$, take the first i bits of this bit sequence, and go to the entry of the bucket array indexed by these i bits. Note that we can determine i because it is kept as part of the data structure.

We follow the pointer in this entry of the bucket array and arrive at a block B. If there is room to put the new record in block B, we do so and we are done. If there is no room, then there are two possibilities, depending on the number j, which indicates how many bits of the hash value are used to determine membership in block B (recall the value of j is found in the "nub" of each block in figures).

1. If $j < i$, then nothing needs to be done to the bucket array. We:
   (a) Split block B into two.
   (b) Distribute records in B to the two blocks, based on the value of their $(j + 1)$st bit — records whose key has 0 in that bit stay in B and those with 1 there go to the new block.
   (c) Put $j + 1$ in each block's "nub" (header) to indicate the number of bits used to determine membership.
   (d) Adjust the pointers in the bucket array so entries that formerly pointed to B now point either to B or the new block, depending on their $(j + 1)$st bit.

Note that splitting block B may not solve the problem, since by chance all the records of B may go into one of the two blocks into which it was split. If so, we need to repeat the process on the overfull block, using the next higher value of j and the block that is still overfull.

2. If $j = i$, then we must first increment i by 1. We double the length of the bucket array, so it now has 2^{i+1} entries. Suppose w is a sequence of i bits indexing one of the entries in the previous bucket array. In the new bucket array, the entries indexed by both $w0$ and $w1$ (i.e., the two numbers derived from w by extending it with 0 or 1) each point to the same block that the w entry used to point to. That is, the two new entries share the block, and the block itself does not change. Membership in the block is still determined by whatever number of bits was previously used. Finally, we proceed to split block B as in case 1. Since i is now greater than j, that case applies.

Example 14.23: Suppose we insert into the table of Fig. 14.23 a record whose key hashes to the sequence 1010. Since the first bit is 1, this record belongs in the second block. However, that block is already full, so it needs to be split. We find that $j = i = 1$ in this case, so we first need to double the bucket array, as shown in Fig. 14.24. We have also set $i = 2$ in this figure.

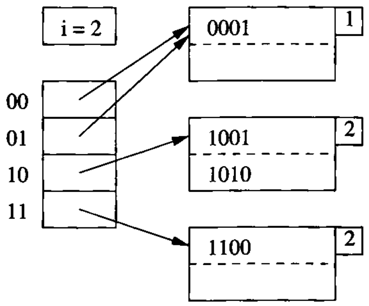

Figure 14.24: Now, two bits of the hash function are used

Notice that the two entries beginning with 0 each point to the block for records whose hashed keys begin with 0, and that block still has the integer 1 in its "nub" to indicate that only the first bit determines membership in the block. However, the block for records beginning with 1 needs to be split, so we partition its records into those beginning 10 and those beginning 11. A 2 in each of these blocks indicates that two bits are used to determine membership. Fortunately, the split is successful; since each of the two new blocks gets at least one record, we do not have to split recursively.

Now suppose we insert records whose keys hash to 0000 and 0111. These both go in the first block of Fig. 14.24, which then overflows. Since only one bit is used to determine membership in this block, while $i = 2$, we do not have to adjust the bucket array. We simply split the block, with 0000 and 0001 staying, and 0111 going to the new block. The entry for 01 in the bucket array is made to point to the new block. Again, we have been fortunate that the records did not all go in one of the new blocks, so we have no need to split recursively.

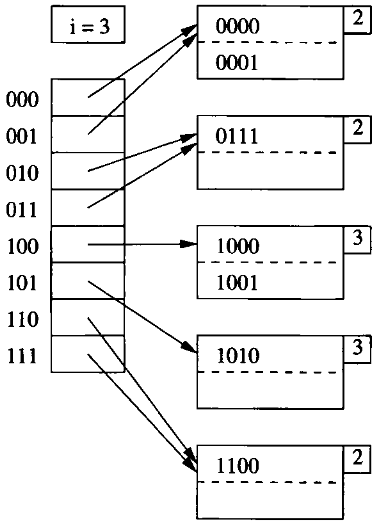

Figure 14.25: The hash table now uses three bits of the hash function

Now suppose a record whose key hashes to 1000 is inserted. The block for 10 overflows. Since it already uses two bits to determine membership, it is time to split the bucket array again and set $i = 3$. Figure 14.25 shows the data structure at this point. Notice that the block for 10 has been split into blocks for 100 and 101, while the other blocks continue to use only two bits to determine membership. □

14.3.7 Linear Hash Tables

Extensible hash tables have some important advantages. Most significant is the fact that when looking for a record, we never need to search more than one data block. We also have to examine an entry of the bucket array, but if the bucket array is small enough to be kept in main memory, then there is no disk I/O needed to access the bucket array. However, extensible hash tables also suffer from some defects:

1. When the bucket array needs to be doubled in size, there is a substantial amount of work to be done (when i is large). This work interrupts access to the data file, or makes certain insertions appear to take a long time.

2. When the bucket array is doubled in size, it may no longer fit in main memory, or may crowd out other data that we would like to hold in main memory. As a result, a system that was performing well might suddenly start using many more disk I/O's per operation.

3. If the number of records per block is small, then there is likely to be one block that needs to be split well in advance of the logical time to do so. For instance, if there are two records per block as in our running example, there might be one sequence of 20 bits that begins the keys of three records, even though the total number of records is much less than 2^{20}. In that case, we would have to use $i = 20$ and a million-bucket array, even though the number of blocks holding records was much smaller than a million.

Another strategy, called *linear hashing*, grows the number of buckets more slowly. The principal new elements we find in linear hashing are:

- The number of buckets n is always chosen so the average number of records per bucket is a fixed fraction, say 80%, of the number of records that fill one block.

- Since blocks cannot always be split, overflow blocks are permitted, although the average number of overflow blocks per bucket will be much less than 1.

- The number of bits used to number the entries of the bucket array is $\lceil \log_2 n \rceil$, where n is the current number of buckets. These bits are always taken from the *right* (low-order) end of the bit sequence that is produced by the hash function.

- Suppose i bits of the hash function are being used to number array entries, and a record with key K is intended for bucket $a_1a_2\cdots a_i$; that is, $a_1a_2\cdots a_i$ are the last i bits of $h(K)$. Then let $a_1a_2\cdots a_i$ be m, treated as an i-bit binary integer. If $m < n$, then the bucket numbered m exists, and we place the record in that bucket. If $n \leq m < 2^i$, then the bucket m does not yet exist, so we place the record in bucket $m - 2^{i-1}$, that is, the bucket we would get if we changed a_1 (which must be 1) to 0.

Example 14.24: Figure 14.26 shows a linear hash table with $n = 2$. We currently are using only one bit of the hash value to determine the buckets of records. Following the pattern established in Example 14.22, we assume the hash function h produces 4 bits, and we represent records by the value produced by h when applied to the search key of the record.

We see in Fig. 14.26 the two buckets, each consisting of one block. The buckets are numbered 0 and 1. All records whose hash value ends in 0 go in the first bucket, and those whose hash value ends in 1 go in the second.

Also part of the structure are the parameters i (the number of bits of the hash function that currently are used), n (the current number of buckets), and r

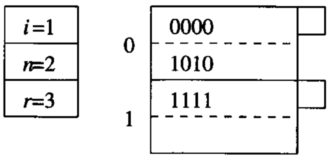

Figure 14.26: A linear hash table

(the current number of records in the hash table). The ratio r/n will be limited so that the typical bucket will need about one disk block. We shall adopt the policy of choosing n, the number of buckets, so that there are no more than $1.7n$ records in the file; i.e., $r \leq 1.7n$. That is, since blocks hold two records, the average occupancy of a bucket does not exceed 85% of the capacity of a block. □

14.3.8 Insertion Into Linear Hash Tables

When we insert a new record, we determine its bucket by the algorithm outlined in Section 14.3.7. We compute $h(K)$, where K is the key of the record, and we use the i bits at the end of bit sequence $h(K)$ as the bucket number, m. If $m < n$, we put the record in bucket m, and if $m \geq n$, we put the record in bucket $m - 2^{i-1}$. If there is no room in the designated bucket, then we create an overflow block, add it to the chain for that bucket, and put the record there.

Each time we insert, we compare the current number of records r with the threshold ratio of r/n, and if the ratio is too high, we add the next bucket to the table. Note that the bucket we add bears no relationship to the bucket into which the insertion occurs! If the binary representation of the number of the bucket we add is $1a_2 \cdots a_i$, then we split the bucket numbered $0a_2 \cdots a_i$, putting records into one or the other bucket, depending on their last i bits. Note that all these records will have hash values that end in $a_2 \cdots a_i$, and only the ith bit from the right end will vary.

The last important detail is what happens when n exceeds 2^i. Then, i is incremented by 1. Technically, all the bucket numbers get an additional 0 in front of their bit sequences, but there is no need to make any physical change, since these bit sequences, interpreted as integers, remain the same.

Example 14.25: We shall continue with Example 14.24 and consider what happens when a record whose key hashes to 0101 is inserted. Since this bit sequence ends in 1, the record goes into the second bucket of Fig. 14.26. There is room for the record, so no overflow block is created.

However, since there are now 4 records in 2 buckets, we exceed the ratio 1.7, and we must therefore raise n to 3. Since $\lceil \log_2 3 \rceil = 2$, we should begin to think of buckets 0 and 1 as 00 and 01, but no change to the data structure is necessary. We add to the table the next bucket, which would have number 10. Then, we split the bucket 00, that bucket whose number differs from the added bucket only in the first bit. When we do the split, the record whose key hashes to 0000 stays in 00, since it ends with 00, while the record whose key hashes to 1010 goes to 10 because it ends that way. The resulting hash table is shown in Fig. 14.27.

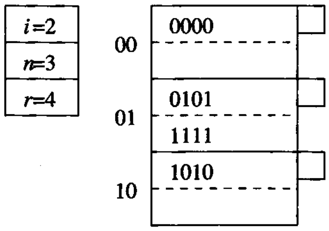

Figure 14.27: Adding a third bucket

Next, let us suppose we add a record whose search key hashes to 0001. The last two bits are 01, so we put it in this bucket, which currently exists. Unfortunately, the bucket's block is full, so we add an overflow block. The three records are distributed among the two blocks of the bucket; we chose to keep them in numerical order of their hashed keys, but order is not important. Since the ratio of records to buckets for the table as a whole is 5/3, and this ratio is less than 1.7, we do not create a new bucket. The result is seen in Fig. 14.28.

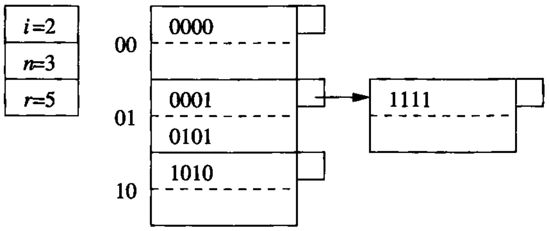

Figure 14.28: Overflow blocks are used if necessary

Finally, consider the insertion of a record whose search key hashes to 0111. The last two bits are 11, but bucket 11 does not yet exist. We therefore redirect this record to bucket 01, whose number differs by having a 0 in the first bit. The new record fits in the overflow block of this bucket.

However, the ratio of the number of records to buckets has exceeded 1.7, so we must create a new bucket, numbered 11. Coincidentally, this bucket is the one we wanted for the new record. We split the four records in bucket 01, with 0001 and 0101 remaining, and 0111 and 1111 going to the new bucket. Since bucket 01 now has only two records, we can delete the overflow block. The hash table is now as shown in Fig. 14.29.

Notice that the next time we insert a record into Fig. 14.29, we shall exceed

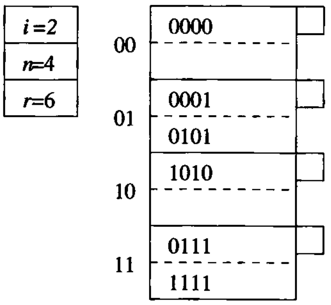

Figure 14.29: Adding a fourth bucket

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: 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 14.3.2: 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 14.3.3:** 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 14.3.4:** 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 14.3.5: 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 14.3.6: 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) $0000, 0001, \ldots, 1111$, and the method of hashing is extensible hashing.

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

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

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

Exercise 14.3.7: 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 14.3.8:** 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 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 λ 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 1000 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 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. 14.30.

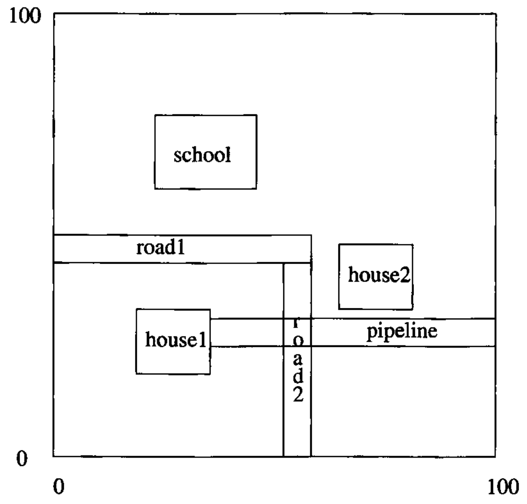

Figure 14.30: Some objects in 2-dimensional space

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:

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. If shapes are represented, then we may ask for the shapes that are partially or wholly within the range. These queries generalize the one-dimensional range queries that we considered in Section 14.2.4.

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-I queries.* 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.

14.4.2 Executing Range Queries Using Conventional Indexes

Now, let us consider to what extent one-dimensional indexes help in answering range queries. Suppose for simplicity that there are two dimensions, x and y. 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. Then, we intersect these pointers, using the idea of Section 14.1.7. 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/O's for finding our way down the B-trees (see Section 14.2.7). To this amount we must add the disk I/O's needed to retrieve all the matching records, however many they may be.

Example 14.26: 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, $450 \leq x \leq 550$ and $450 \leq y \leq 550$. 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 14.2.7, it is generally feasible to keep the root of any B-tree in main memory. Section 14.2.4 showed how to access the 100,000 pointers in either dimension by examining 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. □

14.4.3 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, as suggested by Fig. 14.31.

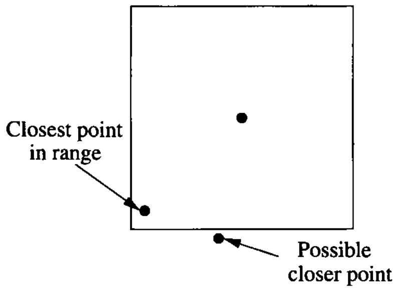

Figure 14.31: The point is in the range, but there could be a closer point outside the range

The general technique we shall use for answering nearest-neighbor queries is to begin by estimating a range in which the nearest point is likely to be found, and executing the corresponding range query. If no points are found within that range, we repeat with a larger range, until eventually we find at least one point. We then consider whether there is the possibility that a closer point exists, but that point is outside the range just used, as in Fig. 14.31. If so, we increase the range once more and retrieve all points in the larger range, to check.

14.4.4 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 one-dimensional index structures. With the hash-based schemes — grid files and partitioned hash functions in Section 14.5 — 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 14.6, trees often will 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.

14.5 Hash 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.

14.5.1 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 14.27: Let us introduce a running example for multidimensional indexes: "who buys gold jewelry?" Imagine a database of customers who have bought gold jewelry. To make things simple, 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. 14.32 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. 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. 14.32 represents points with $40 \leq age < 55$ and $90 \leq salary < 225$. □

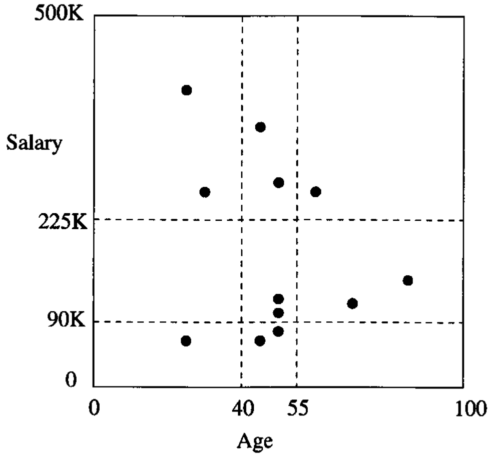

Figure 14.32: A grid file

14.5.2 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.

Example 14.28: Figure 14.33 shows the data of Fig. 14.32 placed in buckets. Since the grids in both dimensions divide the space into three regions, the bucket array is a 3 × 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 buckets 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. □

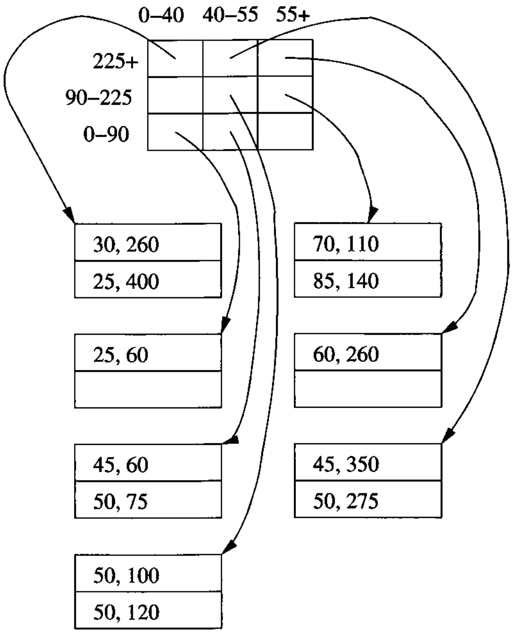

Figure 14.33: A grid file representing the points of Fig. 14.32

14.5.3 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.

Accessing Buckets of a Grid File

While finding the proper coordinates for a point in a three-by-three grid like Fig. 14.33 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.

2. Reorganize the structure by adding or moving the grid lines. This approach is similar to the dynamic hashing techniques discussed in Section 14.3, but there are additional problems because the contents of buckets are linked across a dimension. That is, adding a grid line splits all the 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 a dimension along which to split or a point at which to split, without making many empty buckets or leaving several very full ones.

Example 14.29: Suppose someone 52 years old with an income of \$200K buys gold jewelry. This customer belongs in the central rectangle of Fig. 14.32. 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 points of each of the other buckets for age 40–55 are 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 also splits 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, except for the one we had to split. 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. 14.34. □

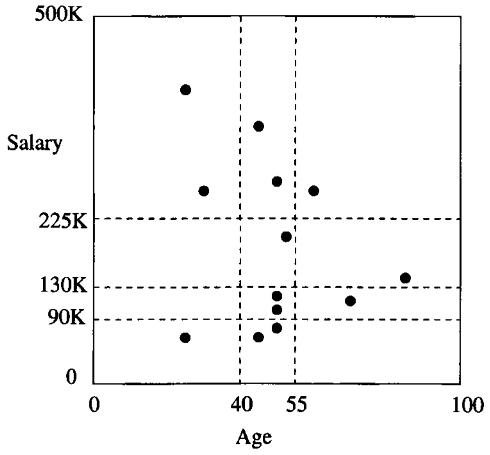

Figure 14.34: Insertion of the point (52, 200) followed by splitting of buckets

14.5.4 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 in the high-dimensional case is that the number of buckets grows exponentially with the number of dimensions. 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. 14.32 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 (as per the box "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 a row or column, but only a small fraction of all the buckets will be accessed.

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. 14.32. 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/O'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. 14.31. 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 14.30: Suppose we are looking in Fig. 14.32 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. □

14.5.5 Partitioned Hash Functions

Hash functions can take a list of values as arguments, although typically there is only one argument. For instance, if a is an integer-valued attribute and b is a character-string-valued attribute, then we could compute $h(a, b)$ by adding the value of a to the value of the ASCII code for each character of b, dividing by the number of buckets, and taking the remainder.

However, such a hash table could be used only 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, and $\sum_{i=1}^{n} k_i = k$. 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 $(v_1, v_2, \ldots, v_n)$ for the n attributes is computed by concatenating the bit sequences: $h_1(v_1)h_2(v_2)\cdots h_n(v_n)$.

Example 14.31: If we have a hash table with 10-bit bucket numbers (1024 buckets), we could devote four bits to attribute a and the remaining six bits to attribute b. Suppose we have a tuple with a-value A and b-value B, perhaps with other attributes that are not involved in the hash. If $h_a(A) = 0101$ and $h_b(B) = 111000$, then this tuple hashes to bucket 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 are 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 are of the form $0101\cdots$, where the $\cdots$ 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 14.32: Suppose we have the "gold jewelry" data of Example 14.27, 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.

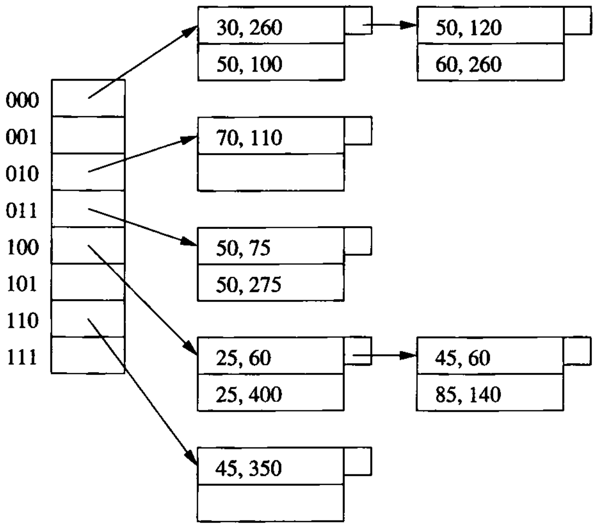

Figure 14.35: A partitioned hash table

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. 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:** 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 14.5.3: 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. 14.36 well.

Exercise 14.5.4: 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 14.5.5: 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 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,

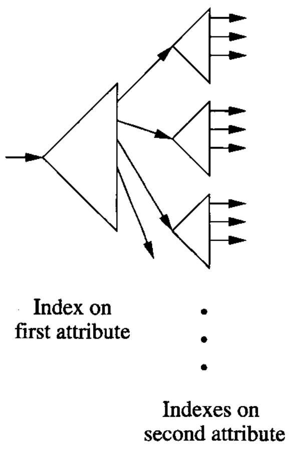

Figure 14.37: Using nested indexes on different keys

then the index we reach by following key 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.

Example 14.33: Figure 14.38 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. 14.38. At the right of Fig. 14.38 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: salaries 75, 100, 120, and 275. □

In a multiple-key index, some of the second- or higher-level indexes may be very small. For example, Fig 14.38 has four second-level 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.

14.6.2 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. We use the root index to find the one subindex that leads to the points we want. On the

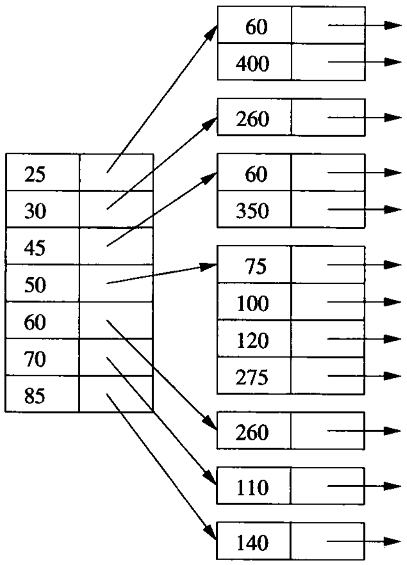

Figure 14.38: Multiple-key indexes for age/salary data

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 — B-trees or indexed-sequential files, for instance. 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.

Nearest-Neighbor Queries

These queries can be answered by a series of range queries, as described in Section 14.4.3.

14.6.3 *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.

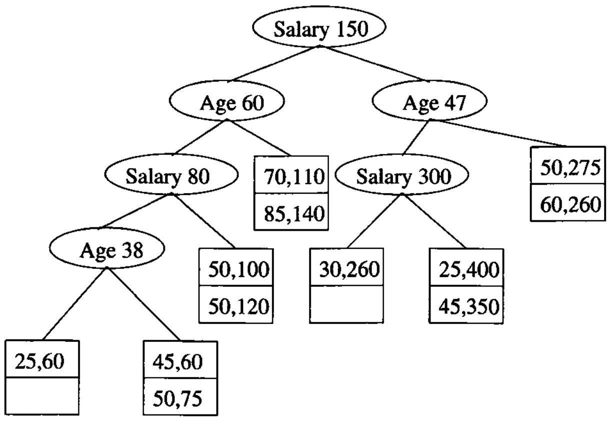

Figure 14.39: A *kd*-tree

Example 14.34: In Fig. 14.39 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 14.40 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. □

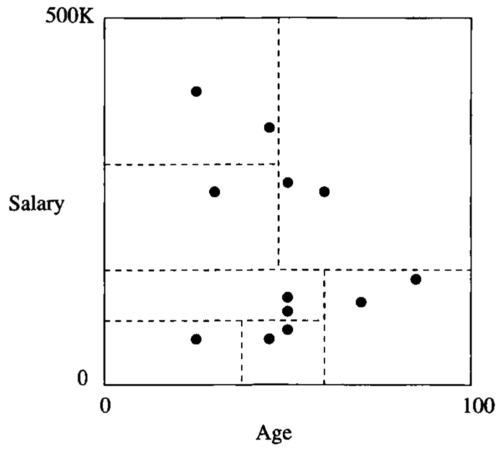

Figure 14.40: The partitions implied by the tree of Fig. 14.39

14.6.4 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 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.[8]

Example 14.35: Suppose someone 35 years old with a salary of $500K buys gold jewelry. Starting at the root, since the salary is at least $150K 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. 14.41. □

[8]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.

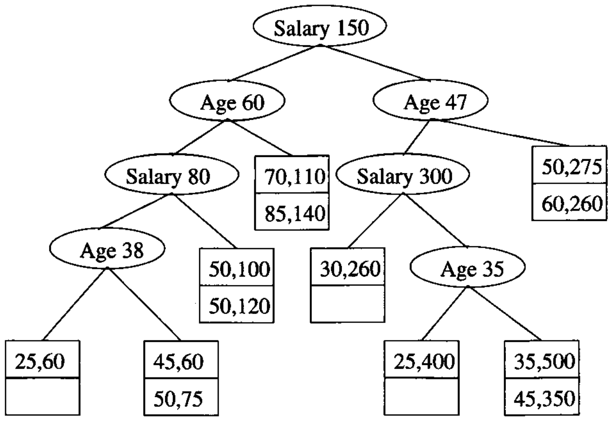

Figure 14.41: Tree after insertion of (35, 500)

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 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. 14.39, 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. For example, if the tree is perfectly balanced and the index has two dimensions, one of which is 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. 14.39 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.

14.6.5 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 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. Two approaches to the twin problems of long paths and unused space are:

1. *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.

2. *Group Interior Nodes Into Blocks.* We could pack many interior nodes, each with two children, into a single block. 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.

14.6.6 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 14.36: Figure 14.42 shows the gold-jewelry data points organized into regions that correspond to nodes of a quad tree. For ease 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.

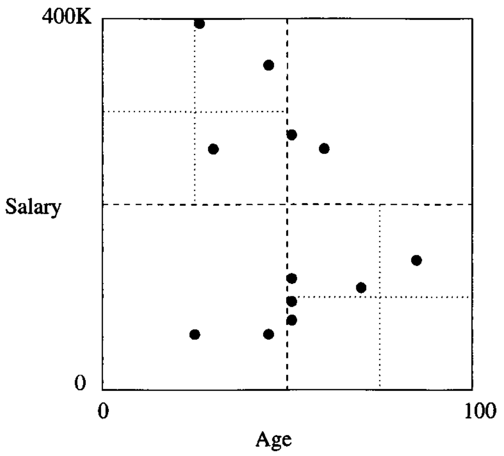

Figure 14.42: Data organized in a quad tree

Figure 14.43 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.

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. 14.42. 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.

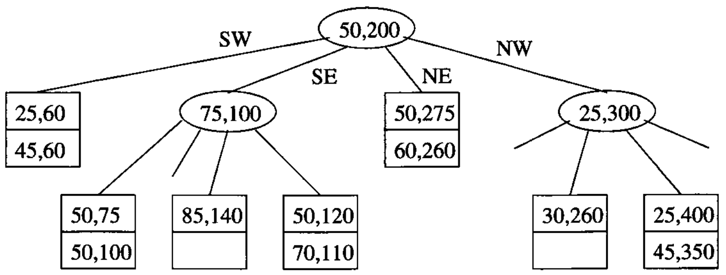

Figure 14.43: A quad tree

The remaining two quadrants each have more than two points. Both are split into subquadrants, as suggested by the dotted lines in Fig. 14.42. Each of the resulting quadrants has at most two 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 14.6.4 for *kd*-trees. The algorithms for quad trees resemble those for *kd*-trees.

14.6.7 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. 14.44. 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 14.44: 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. The subregions are allowed to overlap, although it is desirable to keep the overlap small.

Figure 14.45 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 each data region that lies within the large region is wholly contained within one of the small regions.

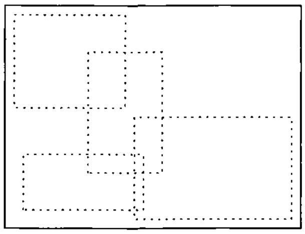

Figure 14.45: The region of an R-tree node and subregions of its children

14.6.8 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. 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 14.37: Let us consider the addition of a new region to the map of Fig. 14.30. Suppose that leaves have room for six regions. Further suppose that the six regions of Fig. 14.30 are together on one leaf, whose region is represented by the outer (solid) rectangle in Fig. 14.46.

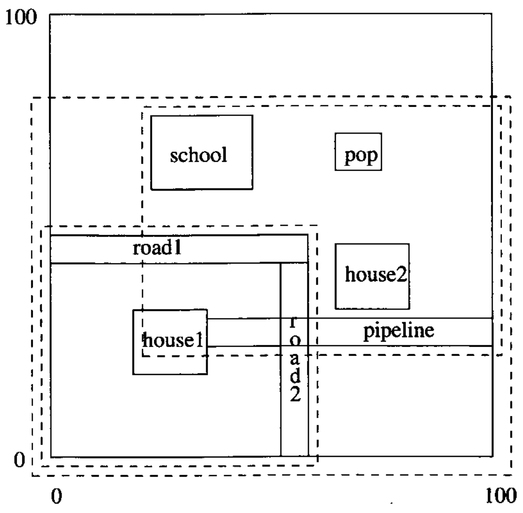

Figure 14.46: Splitting the set of objects

Now, suppose the local cellular phone company adds a POP (point of presence, or base station) at the position shown in Fig. 14.46. Since the seven data regions do not fit on one leaf, we shall split the leaf, with four in one leaf and three in the other. Our options are many; we have picked in Fig. 14.46 the division (indicated by the inner, dashed rectangles) that minimizes the overlap, while splitting the leaves as evenly as possible.

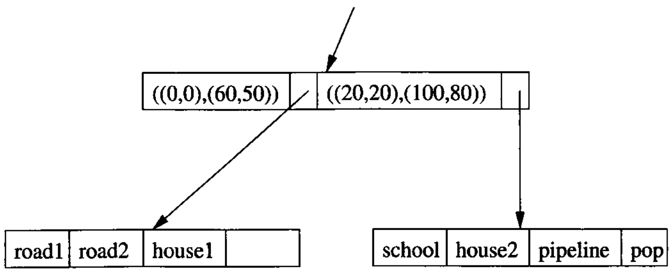

Figure 14.47: An R-tree

We show in Fig. 14.47 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 14.38: Suppose we inserted another house below house2, with lower-left coordinates $(70, 5)$ and upper-right coordinates $(80, 15)$. Since this house is

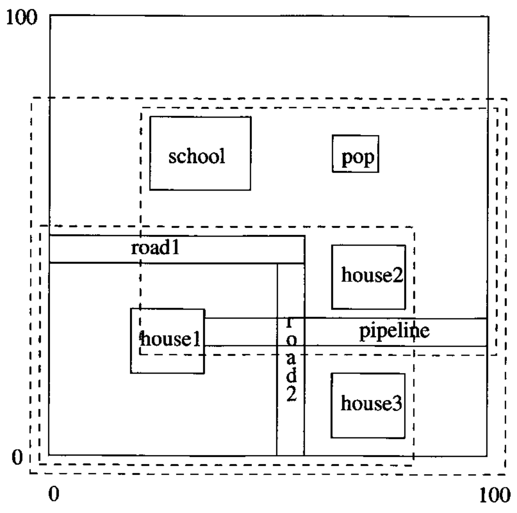

Figure 14.48: Extending a region to accommodate new data

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) Speed, then ram.

b) Ram then hard-disk.

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

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 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 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 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 point $(30, 260)$?

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

Exercise 14.6.6: Show a possible evolution of the tree of Fig. 14.41 if we insert the points $(20, 110)$ and then $(40, 400)$.

! **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: 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 14.6.9: Repeat Exercise 14.6.8 with the addition of a third dimension, hard-disk, that ranges from 0 to 400.

! **Exercise 14.6.10:** 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 14.6.11:** 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?

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 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 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:

Value	Vector
`foo`	100100
`bar`	010010
`baz`	001001

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

14.7.1 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 14.7.2.

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 14.7.4.

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 14.7, 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 14.40: Recall Example 14.7, where we queried the `Movie` relation with the query

```
SELECT title FROM Movie
WHERE studioName = 'Disney' AND year = 2005;
```

Suppose there are bitmap indexes on both attributes `studioName` and `year`. Then we can intersect the vectors for `year` = 2005 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 2005.

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 14.7. 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 14.7.4 and making sure the bit-vectors do not occupy too much space in Section 14.7.2. □

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 14.41: Consider the gold-jewelry data first introduced in Example 14.27. 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. □

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.

14.7.2 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 less 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.

We shall study 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 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.[9]

Example 14.42: If $i = 13$, then $j = 4$; that is, we need 4 bits in the binary

[9]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.

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 a 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 14.43: 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.

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 0's will not be recovered. Since we presumably know the number of records in the file, the additional 0'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 0's.

Example 14.44: Let us convert some of the bit-vectors from Example 14.42 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 0'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 14.44 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, occurs once. 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, the total number of bits to represent the index is at most $2n\log_2 n$. In comparison, the uncompressed bit-vectors for this data would require n^2 bits.

14.7.3 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 14.45: Consider the encoded bit-vectors we obtained in Example 14.44 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 0's and the 1 at position 8 that comes from the bit-vector for age 30. The 1 at position 9 from age 25's bit-vector is produced. Neither bit-vector produces any more 1's for the output. We conclude that the OR of these bit-vectors is 100000011. Technically, we must append 000, since uncompressed bit-vectors are of length twelve in this example. □

14.7.4 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

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.

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. The techniques of Section 13.7 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 seen already 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.

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 i 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 14.7.2, 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.

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 20–40 and a salary in the range 0–100.

Exercise 14.7.3: 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 14.7.4:** 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?

Exercise 14.7.5: Encode, using the scheme of Section 14.7.2, the following bitmaps:

a) 0110000000100000100.

b) 10000010000001001101.

c) 0001000000000010000010000.

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.

- *Dense and Sparse Indexes*: Dense indexes have a key-pointer pair for every record in the data file, while sparse indexes have one key-pointer pair for each block of the data file.

- *Multilevel Indexes*: It is sometimes useful to put an index on the index file itself, an index file on that, and so on. Higher levels of index must be sparse.

- *Secondary Indexes*: An index on a search key K can be created even if the data file is not sorted by K. Such an index must be dense.

- *Inverted Indexes*: The relation between documents and the words they contain is often represented by an index structure with word-pointer pairs. The pointer goes to a place in a "bucket" file where is found a list of pointers to places where that word occurs.

- *B-trees*: These structures are essentially multilevel indexes, with graceful growth capabilities. Blocks with n keys and $n + 1$ pointers are organized in a tree, with the leaves pointing to records. All nonroot blocks are between half-full and completely full at all times.

- *Hash Tables*: We can create hash tables out of blocks in secondary memory, much as we can create main-memory hash tables. A hash function maps search-key values to buckets, effectively partitioning the records of a data file into many small groups (the buckets). Buckets are represented by a block and possible overflow blocks.

- *Extensible Hashing*: This method allows the number of buckets to double whenever any bucket has too many records. It uses an array of pointers to blocks that represent the buckets. To avoid having too many blocks, several buckets can be represented by the same block.

- *Linear Hashing*: This method grows the number of buckets by 1 each time the ratio of records to buckets exceeds a threshold. Since the population of a single bucket cannot cause the table to expand, overflow blocks for buckets are needed in some situations.

- *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.

- *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.

- *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.

- *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.

- *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.

14.9 References for Chapter 14

The B-tree was the original idea of Bayer and McCreight [2]. Unlike the B+ tree described here, this formulation had pointers to records at the interior nodes as well as at the leaves. [8] is a survey of B-tree varieties.

Hashing as a data structure goes back to Peterson [19]. Extensible hashing was developed by [9], while linear hashing is from [15]. The book by Knuth [14] contains much information on data structures, including techniques for selecting hash functions and designing hash tables, as well as a number of ideas concerning B-tree variants. The B+ tree formulation (without key values at interior nodes) appeared in the 1973 edition of [14].

Secondary indexes and other techniques for retrieval of documents are covered by [23]. Also, [10] and [1] are surveys of index methods for text documents.

The *kd*-tree is from [4]. Modifications suitable for secondary storage appeared in [5] and [21]. Partitioned hashing and its use in partial-match retieval is from [20] and [7]. However, the design idea from Exercise 14.5.6 is from [22].

Grid files first appeared in [16] and the quad tree in [11]. The R-tree is from [13], and two extensions [24] and [3] 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 [17]. [18] is a recent expansion of the idea.

There are a number of surveys of multidimensional storage structures. One of the earliest is [6]. More recent surveys are found in [25] and [12]. The former also includes surveys of several other important database topics.

1. R. Baeza-Yates, "Integrating contents and structure in text retrieval," *SIGMOD Record* **25**:1 (1996), pp. 67–79.

2. R. Bayer and E. M. McCreight, "Organization and maintenance of large ordered indexes," *Acta Informatica* **1**:3 (1972), pp. 173–189.

3. N. Beckmann, 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.

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

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

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

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

8. D. Comer, "The ubiquitous B-tree," *Computing Surveys* **11**:2 (1979), pp. 121–137.

9. R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible hashing — a fast access method for dynamic files," *ACM Trans. on Database Systems* **4**:3 (1979), pp. 315–344.

10. C. Faloutsos, "Access methods for text," *Computing Surveys* **17**:1 (1985), pp. 49–74.

11. 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.

12. V. Gaede and O. Gunther, "Multidimensional access methods," *Computing Surveys* **30**:2 (1998), pp. 170–231.

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

14. D. E. Knuth, *The Art of Computer Programming, Vol. III, Sorting and Searching, Second Edition*, Addison-Wesley, Reading MA, 1998.

15. W. Litwin, "Linear hashing: a new tool for file and table addressing," *Intl. Conf. on Very Large Databases*, pp. 212–223, 1980.

16. 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.

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

18. 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.

19. W. W. Peterson, "Addressing for random access storage," *IBM J. Research and Development* **1**:2 (1957), pp. 130–146.

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

21. 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.

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

23. G. Salton, *Introduction to Modern Information Retrieval*, McGraw-Hill, New York, 1983.

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

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

Chapter 15

Query Execution

The broad topic of query processing will be covered in this chapter and Chapter 16. The *query processor* is the group of components of a DBMS that turns user queries and data-modification commands into a sequence of database operations and executes those operations. Since SQL lets us express queries at a very high level, the query processor must supply much detail regarding how the query is to be executed. Moreover, a naive execution strategy for a query may take far more time than necessary.

Figure 15.1 suggests the division of topics between Chapters 15 and 16. In this chapter, we concentrate on query execution, that is, the algorithms that manipulate the data of the database. We focus on the operations of the extended relational algebra, described in Section 5.2. Because SQL uses a bag model, we also assume that relations are bags, and thus use the bag versions of the operators from Section 5.1.

We shall cover the principal methods for execution of the operations of relational algebra. These methods differ in their basic strategy; scanning, hashing, sorting, and indexing are the major approaches. The methods also differ on their assumption as to the amount of available main memory. Some algorithms assume that enough main memory is available to hold at least one of the relations involved in an operation. Others assume that the arguments of the operation are too big to fit in memory, and these algorithms have significantly different costs and structures.

Preview of Query Compilation

To set the context for query execution, we offer a very brief outline of the content of the next chapter. Query compilation is divided into the three major steps shown in Fig. 15.2.

a) *Parsing.* A *parse tree* for the query is constructed.

b) *Query Rewrite.* The parse tree is converted to an initial query plan, which is usually an algebraic representation of the query. This initial plan is then

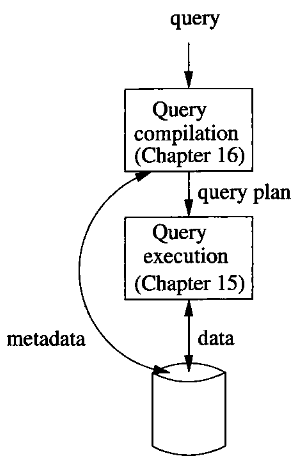

Figure 15.1: The major parts of the query processor

transformed into an equivalent plan that is expected to require less time to execute.

c) *Physical Plan Generation.* The abstract query plan from (b), often called a *logical query plan*, is turned into a *physical query plan* by selecting algorithms to implement each of the operators of the logical plan, and by selecting an order of execution for these operators. The physical plan, like the result of parsing and the logical plan, is represented by an expression tree. The physical plan also includes details such as how the queried relations are accessed, and when and if a relation should be sorted.

Parts (b) and (c) are often called the *query optimizer*, and these are the hard parts of query compilation. To select the best query plan we need to decide:

1. Which of the algebraically equivalent forms of a query leads to the most efficient algorithm for answering the query?

2. For each operation of the selected form, what algorithm should we use to implement that operation?

3. How should the operations pass data from one to the other, e.g., in a pipelined fashion, in main-memory buffers, or via the disk?

Each of these choices depends on the metadata about the database. Typical metadata that is available to the query optimizer includes: the size of each relation; statistics such as the approximate number and frequency of different values for an attribute; the existence of certain indexes; and the layout of data on disk.

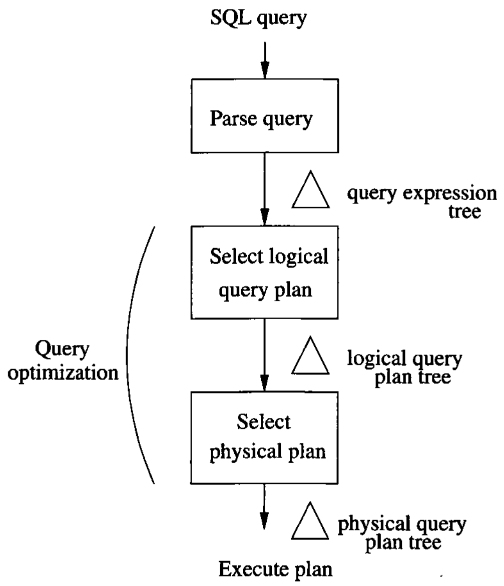

Figure 15.2: Outline of query compilation

15.1 Introduction to Physical-Query-Plan Operators

Physical query plans are built from operators, each of which implements one step of the plan. Often, the physical operators are particular implementations for one of the operations of relational algebra. However, we also need physical operators for other tasks that do not involve an operation of relational algebra. For example, we often need to "scan" a table, that is, bring into main memory each tuple of some relation. The relation is typically an operand of some other operation.

In this section, we shall introduce the basic building blocks of physical query plans. Later sections cover the more complex algorithms that implement operators of relational algebra efficiently; these algorithms also form an essential part of physical query plans. We also introduce here the "iterator" concept, which is an important method by which the operators comprising a physical query plan can pass requests for tuples and answers among themselves.

15.1.1 Scanning Tables

Perhaps the most basic thing we can do in a physical query plan is to read the entire contents of a relation R. A variation of this operator involves a simple predicate, where we read only those tuples of the relation R that satisfy the predicate. There are two basic approaches to locating the tuples of a relation R.

1. In many cases, the relation R is stored in an area of secondary memory, with its tuples arranged in blocks. The blocks containing the tuples of R are known to the system, and it is possible to get the blocks one by one. This operation is called *table-scan*.

2. If there is an index on any attribute of R, we may be able to use this index to get all the tuples of R. For example, a sparse index on R, as discussed in Section 14.1.3, can be used to lead us to all the blocks holding R, even if we don't know otherwise which blocks these are. This operation is called *index-scan*.

We shall take up index-scan again in Section 15.6.2, when we talk about implementing selection. However, the important observation for now is that we can use the index not only to get *all* the tuples of the relation it indexes, but to get only those tuples that have a particular value (or sometimes a particular range of values) in the attribute or attributes that form the search key for the index.

15.1.2 Sorting While Scanning Tables

There are a number of reasons why we might want to sort a relation as we read its tuples. For one, the query could include an `ORDER BY` clause, requiring that a relation be sorted. For another, some approaches to implementing relational-algebra operations require one or both arguments to be sorted relations. These algorithms appear in Section 15.4 and elsewhere.

The physical-query-plan operator *sort-scan* takes a relation R and a specification of the attributes on which the sort is to be made, and produces R in that sorted order. There are several ways that sort-scan can be implemented. If relation R must be sorted by attribute a, and there is a B-tree index on a, then a scan of the index allows us to produce R in the desired order. If R is small enough to fit in main memory, then we can retrieve its tuples using a table scan or index scan, and then use a main-memory sorting algorithm. If R is too large to fit in main memory, then we can use a multiway merge-sort, as will be discussed Section 15.4.1.

15.1.3 The Computation Model for Physical Operators

A query generally consists of several operations of relational algebra, and the corresponding physical query plan is composed of several physical operators. Since choosing physical-plan operators wisely is an essential of a good query processor, we must be able to estimate the "cost" of each operator we use. We shall use the number of disk I/O's as our measure of cost for an operation. This measure is consistent with our view (see Section 13.3.1) that it takes longer to get data from disk than to do anything useful with it once the data is in main memory.

When comparing algorithms for the same operations, we shall make an assumption that may be surprising at first:

- We assume that the arguments of any operator are found on disk, but the result of the operator is left in main memory.

If the operator produces the final answer to a query, and that result is indeed written to disk, then the cost of doing so depends only on the size of the answer, and not on how the answer was computed. We can simply add the final write-back cost to the total cost of the query. However, in many applications, the answer is not stored on disk at all, but printed or passed to some formatting program. Then, the disk I/O cost of the output either is zero or depends upon what some unknown application program does with the data. In either case, the cost of writing the answer does not influence our choice of algorithm for executing the operator.

Similarly, the result of an operator that forms part of a query (rather than the whole query) often is not written to disk. In Section 15.1.6 we shall discuss "iterators," where the result of one operator O_1 is constructed in main memory, perhaps a small piece at a time, and passed as an argument to another operator O_2. In this situation, we never have to write the result of O_1 to disk, and moreover, we save the cost of reading from disk an argument of O_2.

15.1.4 Parameters for Measuring Costs

Now, let us introduce the parameters (sometimes called statistics) that we use to express the cost of an operator. Estimates of cost are essential if the optimizer is to determine which of the many query plans is likely to execute fastest. Section 16.5 will show how to exploit these cost estimates.

We need a parameter to represent the portion of main memory that the operator uses, and we require other parameters to measure the size of its argument(s). Assume that main memory is divided into buffers, whose size is the same as the size of disk blocks. Then M will denote the number of main-memory buffers available to an execution of a particular operator.

Sometimes, we can think of M as the entire main memory, or most of the main memory. However, we shall also see situations where several operations share the main memory, so M could be much smaller than the total main memory. In fact, as we shall discuss in Section 15.7, the number of buffers available to an operation may not be a predictable constant, but may be decided during execution, based on what other processes are executing at the same time. If so, M is really an estimate of the number of buffers available to the operation.

Next, let us consider the parameters that measure the cost of accessing argument relations. These parameters, measuring size and distribution of data in a relation, are often computed periodically to help the query optimizer choose physical operators.

We shall make the simplifying assumption that data is accessed one block at a time from disk. In practice, one of the techniques discussed in Section 13.3 might be able to speed up the algorithm if we are able to read many blocks of the relation at once, and they can be read from consecutive blocks on a track. There are three parameter families, B, T, and V:

- When describing the size of a relation R, we most often are concerned with the number of blocks that are needed to hold all the tuples of R. This number of blocks will be denoted $B(R)$, or just B if we know that relation R is meant. Usually, we assume that R is *clustered*; that is, it is stored in B blocks or in approximately B blocks.

- Sometimes, we also need to know the number of tuples in R, and we denote this quantity by $T(R)$, or just T if R is understood. If we need the number of tuples of R that can fit in one block, we can use the ratio T/B.

- Finally, we shall sometimes want to refer to the number of distinct values that appear in a column of a relation. If R is a relation, and one of its attributes is a, then $V(R,a)$ is the number of distinct values of the column for a in R. More generally, if $[a_1, a_2, \ldots, a_n]$ is a list of attributes, then $V(R, [a_1, a_2, \ldots, a_n])$ is the number of distinct n-tuples in the columns of R for attributes $a_1, a_2, \ldots, a_n$. Put formally, it is the number of tuples in $\delta(\pi_{a_1,a_2,\ldots,a_n}(R))$.

15.1.5 I/O Cost for Scan Operators

As a simple application of the parameters that were introduced, we can represent the number of disk I/O's needed for each of the table-scan operators discussed so far. If relation R is clustered, then the number of disk I/O's for the table-scan operator is approximately B. Likewise, if R fits in main-memory, then we can implement sort-scan by reading R into memory and performing an in-memory sort, again requiring only B disk I/O's.

However, if R is not clustered, then the number of required disk I/O's is generally much higher. If R is distributed among tuples of other relations, then a table-scan for R may require reading as many blocks as there are tuples of R; that is, the I/O cost is T. Similarly, if we want to sort R, but R fits in memory, then T disk I/O's are what we need to get all of R into memory.

Finally, let us consider the cost of an index-scan. Generally, an index on a relation R occupies many fewer than $B(R)$ blocks. Therefore, a scan of the entire R, which takes at least B disk I/O's, will require significantly more I/O's than does examining the entire index. Thus, even though index-scan requires examining both the relation and its index,

- We continue to use B or T, respectively, to estimate the cost of accessing a clustered or unclustered relation in its entirety, using an index.

However, if we only want part of R, we often are able to avoid looking at the entire index and the entire R. We shall defer analysis of these uses of indexes to Section 15.6.2.

15.1.6 Iterators for Implementation of Physical Operators

Many physical operators can be implemented as an *iterator*, which is a group of three methods that allows a consumer of the result of the physical operator to get the result one tuple at a time. The three methods forming the iterator for an operation are:

1. `Open()`. This method starts the process of getting tuples, but does not get a tuple. It initializes any data structures needed to perform the operation and calls `Open()` for any arguments of the operation.

2. `GetNext()`. This method returns the next tuple in the result and adjusts data structures as necessary to allow subsequent tuples to be obtained. In getting the next tuple of its result, it typically calls `GetNext()` one or more times on its argument(s). If there are no more tuples to return, `GetNext()` returns a special value `NotFound`, which we assume cannot be mistaken for a tuple.

3. `Close()`. This method ends the iteration after all tuples, or all tuples that the consumer wanted, have been obtained. Typically, it calls `Close()` on any arguments of the operator.

When describing iterators and their methods, we shall assume that there is a "class" for each type of iterator (i.e., for each type of physical operator implemented as an iterator), and the class defines `Open()`, `GetNext()`, and `Close()` methods on instances of the class.

Example 15.1: Perhaps the simplest iterator is the one that implements the table-scan operator. The iterator is implemented by a class `TableScan`, and a table-scan operator in a query plan is an instance of this class parameterized by the relation R we wish to scan. Let us assume that R is a relation clustered in some list of blocks, which we can access in a convenient way; that is, the notion of "get the next block of R" is implemented by the storage system and need not be described in detail. Further, we assume that within a block there is a directory of records (tuples), so it is easy to get the next tuple of a block or tell that the last tuple has been reached.

Figure 15.3 sketches the three methods for this iterator. We imagine a block pointer b and a tuple pointer t that points to a tuple within block b. We assume that both pointers can point "beyond" the last block or last tuple of a block, respectively, and that it is possible to identify when these conditions occur. Notice that `Close()` in this example does nothing. In practice, a `Close()` method for an iterator might clean up the internal structure of the DBMS in various ways. It might inform the buffer manager that certain buffers are no