chapter 17

Indexing Structures for Files and Physical Database Design

In this chapter, we assume that a file already exists with some primary organization such as the unordered, ordered, or hashed organizations that were described in Chapter 16. We will describe additional auxiliary **access structures** called **indexes**, which are used to speed up the retrieval of records in response to certain search conditions. The index structures are additional files on disk that provide **secondary access paths**, which provide alternative ways to access the records without affecting the physical placement of records in the primary data file on disk. They enable efficient access to records based on the **indexing fields** that are used to construct the index. Basically, *any field* of the file can be used to create an index, and *multiple indexes* on different fields—as well as indexes on *multiple fields*—can be constructed on the same file. A variety of indexes are possible; each of them uses a particular data structure to speed up the search. To find a record or records in the data file based on a search condition on an indexing field, the index is searched, which leads to pointers to one or more disk blocks in the data file where the required records are located. The most prevalent types of indexes are based on ordered files (single-level indexes) and use tree data structures (multilevel indexes, B^+-trees) to organize the index. Indexes can also be constructed based on hashing or other search data structures. We also discuss indexes that are vectors of bits called *bitmap indexes*.

We describe different types of single-level ordered indexes—primary, secondary, and clustering—in Section 17.1. By viewing a single-level index as an ordered file, one can develop additional indexes for it, giving rise to the concept of multilevel indexes. A popular indexing scheme called **ISAM (indexed sequential access method)** is based on this idea. We discuss multilevel tree-structured indexes in Section 17.2. In Section 17.3, we describe B-trees and B^+-trees, which are data structures that are commonly used in DBMSs to implement dynamically changing

multilevel indexes. B^+-trees have become a commonly accepted default structure for generating indexes on demand in most relational DBMSs. Section 17.4 is devoted to alternative ways to access data based on a combination of multiple keys. In Section 17.5, we discuss hash indexes and introduce the concept of logical indexes, which give an additional level of indirection from physical indexes and allow the physical index to be flexible and extensible in its organization. In Section 17.6, we discuss multikey indexing and bitmap indexes used for searching on one or more keys. Section 17.7 covers physical design and Section 7.8 summarizes the chapter.

17.1 Types of Single-Level Ordered Indexes

The idea behind an ordered index is similar to that behind the index used in a textbook, which lists important terms at the end of the book in alphabetical order along with a list of page numbers where the term appears in the book. We can search the book index for a certain term in the textbook to find a list of *addresses*—page numbers in this case—and use these addresses to locate the specified pages first and then *search* for the term on each specified page. The alternative, if no other guidance is given, would be to sift slowly through the whole textbook word by word to find the term we are interested in; this corresponds to doing a *linear search*, which scans the whole file. Of course, most books do have additional information, such as chapter and section titles, which help us find a term without having to search through the whole book. However, the index is the only exact indication of the pages where each term occurs in the book.

For a file with a given record structure consisting of several fields (or attributes), an index access structure is usually defined on a single field of a file, called an **indexing field** (or **indexing attribute**).[1] The index typically stores each value of the index field along with a list of pointers to all disk blocks that contain records with that field value. The values in the index are *ordered* so that we can do a *binary search* on the index. If both the data file and the index file are ordered, and since the index file is typically much smaller than the data file, searching the index using a binary search is a better option. Tree-structured multilevel indexes (see Section 17.2) implement an extension of the binary search idea that reduces the search space by two-way partitioning at each search step to an *n*-ary partitioning approach that divides the search space in the file *n*-ways at each stage.

There are several types of ordered indexes. A **primary index** is specified on the *ordering key field* of an **ordered file** of records. Recall from Section 16.7 that an ordering key field is used to *physically order* the file records on disk, and every record has a *unique value* for that field. If the ordering field is not a key field—that is, if numerous records in the file can have the same value for the ordering field—another type of index, called a **clustering index**, can be used. The data file is called a **clustered file** in this latter case. Notice that a file can have at most one physical ordering field, so it can have at most one primary index or one clustering index, *but*

[1]We use the terms *field* and *attribute* interchangeably in this chapter.

not both. A third type of index, called a **secondary index**, can be specified on any *nonordering* field of a file. A data file can have several secondary indexes in addition to its primary access method. We discuss these types of single-level indexes in the next three subsections.

17.1.1 Primary Indexes

A **primary index** is an ordered file whose records are of fixed length with two fields, and it acts like an access structure to efficiently search for and access the data records in a data file. The first field is of the same data type as the ordering key field—called the **primary key**—of the data file, and the second field is a pointer to a disk block (a block address). There is one **index entry** (or **index record**) in the index file for each *block* in the data file. Each index entry has the value of the primary key field for the *first* record in a block and a pointer to that block as its two field values. We will refer to the two field values of index entry i as $<K(i), P(i)>$. In the rest of this chapter, we refer to different types of index **entries** $< K(i), X >$ as follows:

- X may be the physical address of a block (or page) in the file, as in the case of $P(i)$ above.
- X may be the record address made up of a block address and a record id (or offset) within the block.
- X may be a logical address of the block or of the record within the file and is a relative number that would be mapped to a physical address (see further explanation in Section 17.6.1).

To create a primary index on the ordered file shown in Figure 16.7, we use the Name field as primary key, because that is the ordering key field of the file (assuming that each value of Name is unique). Each entry in the index has a Name value and a pointer. The first three index entries are as follows:

$<K(1) =$ (Aaron, Ed), $P(1) =$ address of block 1$>$
$<K(2) =$ (Adams, John), $P(2) =$ address of block 2$>$
$<K(3) =$ (Alexander, Ed), $P(3) =$ address of block 3$>$

Figure 17.1 illustrates this primary index. The total number of entries in the index is the same as the *number of disk blocks* in the ordered data file. The first record in each block of the data file is called the **anchor record** of the block, or simply the **block anchor**.[2]

Indexes can also be characterized as dense or sparse. A **dense index** has an index entry for *every search key value* (and hence every record) in the data file. A **sparse** (or **nondense**) **index**, on the other hand, has index entries for only some of the search values. A sparse index has fewer entries than the number of records in the file. Thus, a primary index is a nondense (sparse) index, since it includes an

[2]We can use a scheme similar to the one described here, with the last record in each block (rather than the first) as the block anchor. This slightly improves the efficiency of the search algorithm.

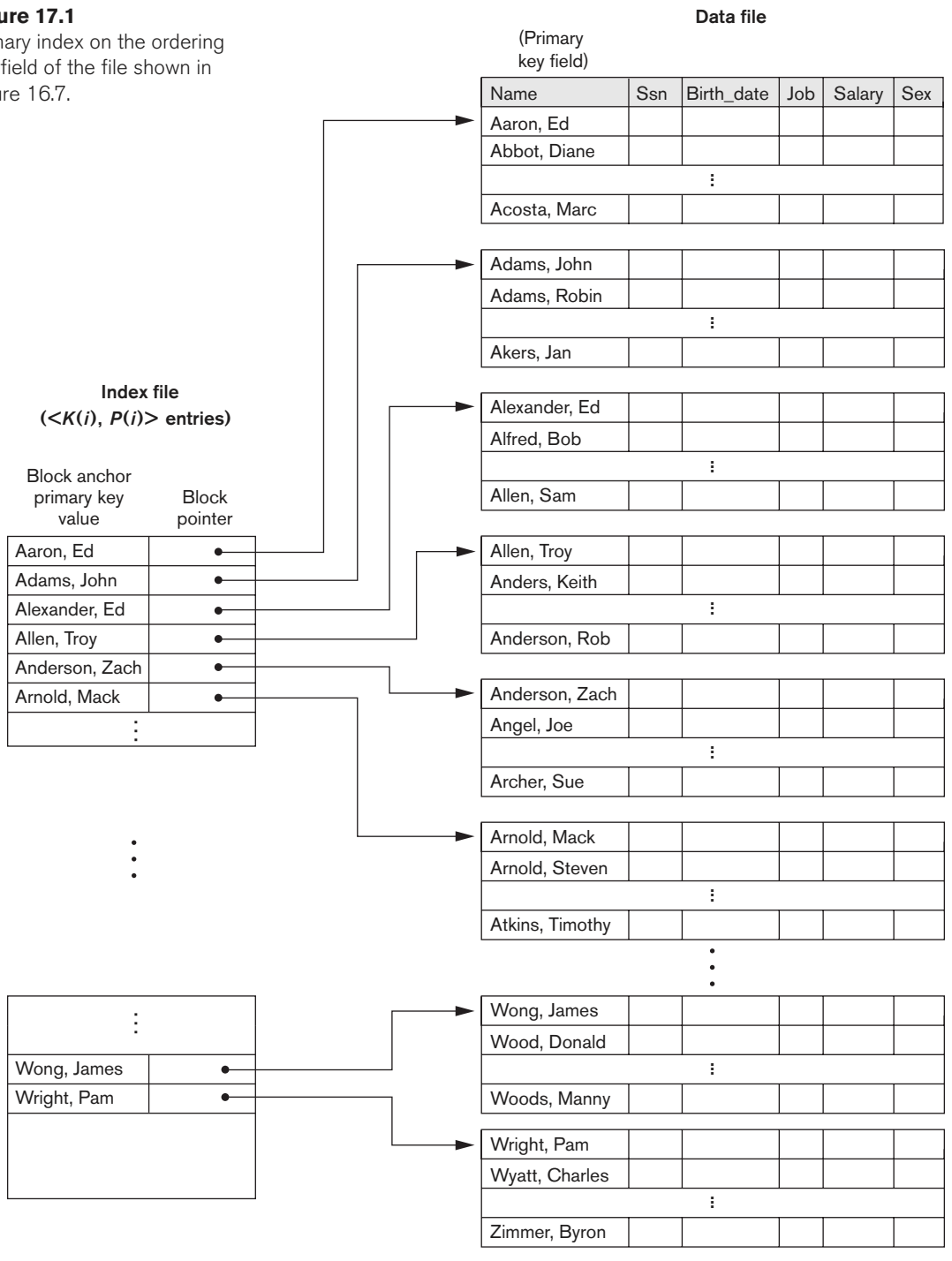


Figure 17.1
Primary index on the ordering key field of the file shown in Figure 16.7.

entry for each disk block of the data file and the keys of its anchor record rather than for every search value (or every record).[3]

The index file for a primary index occupies a much smaller space than does the data file, for two reasons. First, there are *fewer index entries* than there are records in the data file. Second, each index entry is typically *smaller in size* than a data record because it has only two fields, both of which tend to be short in size; consequently, more index entries than data records can fit in one block. Therefore, a binary search on the index file requires fewer block accesses than a binary search on the data file. Referring to Table 16.3, note that the binary search for an ordered data file required $\log_2 b$ block accesses. But if the primary index file contains only b_i blocks, then to locate a record with a search key value requires a binary search of that index and access to the block containing that record: a total of $\log_2 b_i + 1$ accesses.

A record whose primary key value is K lies in the block whose address is $P(i)$, where $K(i) \leq K < K(i + 1)$. The ith block in the data file contains all such records because of the physical ordering of the file records on the primary key field. To retrieve a record, given the value K of its primary key field, we do a binary search on the index file to find the appropriate index entry i, and then retrieve the data file block whose address is $P(i)$.[4] Example 1 illustrates the saving in block accesses that is attainable when a primary index is used to search for a record.

Example 1. Suppose that we have an ordered file with $r = 300{,}000$ records stored on a disk with block size $B = 4{,}096$ bytes.[5] File records are of fixed size and are unspanned, with record length $R = 100$ bytes. The blocking factor for the file would be $bfr = \lfloor (B/R) \rfloor = \lfloor (4{,}096/100) \rfloor = 40$ records per block. The number of blocks needed for the file is $b = \lceil (r/bfr) \rceil = \lceil (300{,}000/40) \rceil = 7{,}500$ blocks. A binary search on the data file would need approximately $\lceil \log_2 b \rceil = \lceil (\log_2 7{,}500) \rceil = 13$ block accesses.

Now suppose that the ordering key field of the file is $V = 9$ bytes long, a block pointer is $P = 6$ bytes long, and we have constructed a primary index for the file. The size of each index entry is $R_i = (9 + 6) = 15$ bytes, so the blocking factor for the index is $bfr_i = \lfloor (B/R_i) \rfloor = \lfloor (4{,}096/15) \rfloor = 273$ entries per block. The total number of index entries r_i is equal to the number of blocks in the data file, which is 7,500. The number of index blocks is hence $b_i = \lceil (r_i/bfr_i) \rceil = \lceil (7{,}500/273) \rceil = 28$ blocks. To perform a binary search on the index file would need $\lceil (\log_2 b_i) \rceil = \lceil (\log_2 28) \rceil = 5$ block accesses. To search for a record using the index, we need one additional block access to the data file for a total of 5 + 1 = 6 block accesses—an improvement over binary search on the data file, which required 13 disk block accesses. Note that the index with 7,500 entries of 15 bytes each is rather small (112,500 or 112.5 Kbytes) and would typically be kept in main memory thus requiring negligible time to search with binary search. In that case we simply make one block access to retrieve the record.

---

[3]The sparse primary index has been called clustered (primary) index in some books and articles.

[4]Notice that the above formula would not be correct if the data file were ordered on a *nonkey field*; in that case the same index value in the block anchor could be repeated in the last records of the previous block.

[5]Most DBMS vendors, including Oracle, are using 4K or 4,096 bytes as a standard block/page size.

A major problem with a primary index—as with any ordered file—is insertion and deletion of records. With a primary index, the problem is compounded because if we attempt to insert a record in its correct position in the data file, we must not only move records to make space for the new record but also change some index entries, since moving records will change the *anchor records* of some blocks. Using an unordered overflow file, as discussed in Section 16.7, can reduce this problem. Another possibility is to use a linked list of overflow records for each block in the data file. This is similar to the method of dealing with overflow records described with hashing in Section 16.8.2. Records within each block and its overflow linked list can be sorted to improve retrieval time. Record deletion is handled using deletion markers.

17.1.2 Clustering Indexes

If file records are physically ordered on a nonkey field—which *does not* have a distinct value for each record—that field is called the **clustering field** and the data file is called a **clustered file.** We can create a different type of index, called a **clustering index**, to speed up retrieval of all the records that have the same value for the clustering field. This differs from a primary index, which requires that the ordering field of the data file have a *distinct value* for each record.

A clustering index is also an ordered file with two fields; the first field is of the same type as the clustering field of the data file, and the second field is a disk block pointer. There is one entry in the clustering index for each *distinct value* of the clustering field, and it contains the value and a pointer to the *first block* in the data file that has a record with that value for its clustering field. Figure 17.2 shows an example. Notice that record insertion and deletion still cause problems because the data records are physically ordered. To alleviate the problem of insertion, it is common to reserve a whole block (or a cluster of contiguous blocks) for *each value* of the clustering field; all records with that value are placed in the block (or block cluster). This makes insertion and deletion relatively straightforward. Figure 17.3 shows this scheme.

A clustering index is another example of a *nondense* index because it has an entry for every *distinct value* of the indexing field, which is a nonkey by definition and hence has duplicate values rather than a unique value for every record in the file.

Example 2. Suppose that we consider the same ordered file with r = 300,000 records stored on a disk with block size B = 4,096 bytes. Imagine that it is ordered by the attribute Zipcode and there are 1,000 zip codes in the file (with an average 300 records per zip code, assuming even distribution across zip codes.) The index in this case has 1,000 index entries of 11 bytes each (5-byte Zipcode and 6-byte block pointer) with a blocking factor $bfr_i = \lfloor (B/R_i) \rfloor = \lfloor (4{,}096/11) \rfloor = 372$ index entries per block. The number of index blocks is hence $b_i = \lceil (r_i/bfr_i) \rceil = \lceil (1{,}000/372) \rceil = 3$ blocks. To perform a binary search on the index file would need $\lceil (\log_2 b_i) \rceil = \lceil (\log_2 3) \rceil = 2$ block accesses. Again, this index would typically be loaded in main memory (occupies 11,000 or 11 Kbytes) and takes negligible time to search in memory. One block access to the data file would lead to the first record with a given zip code.

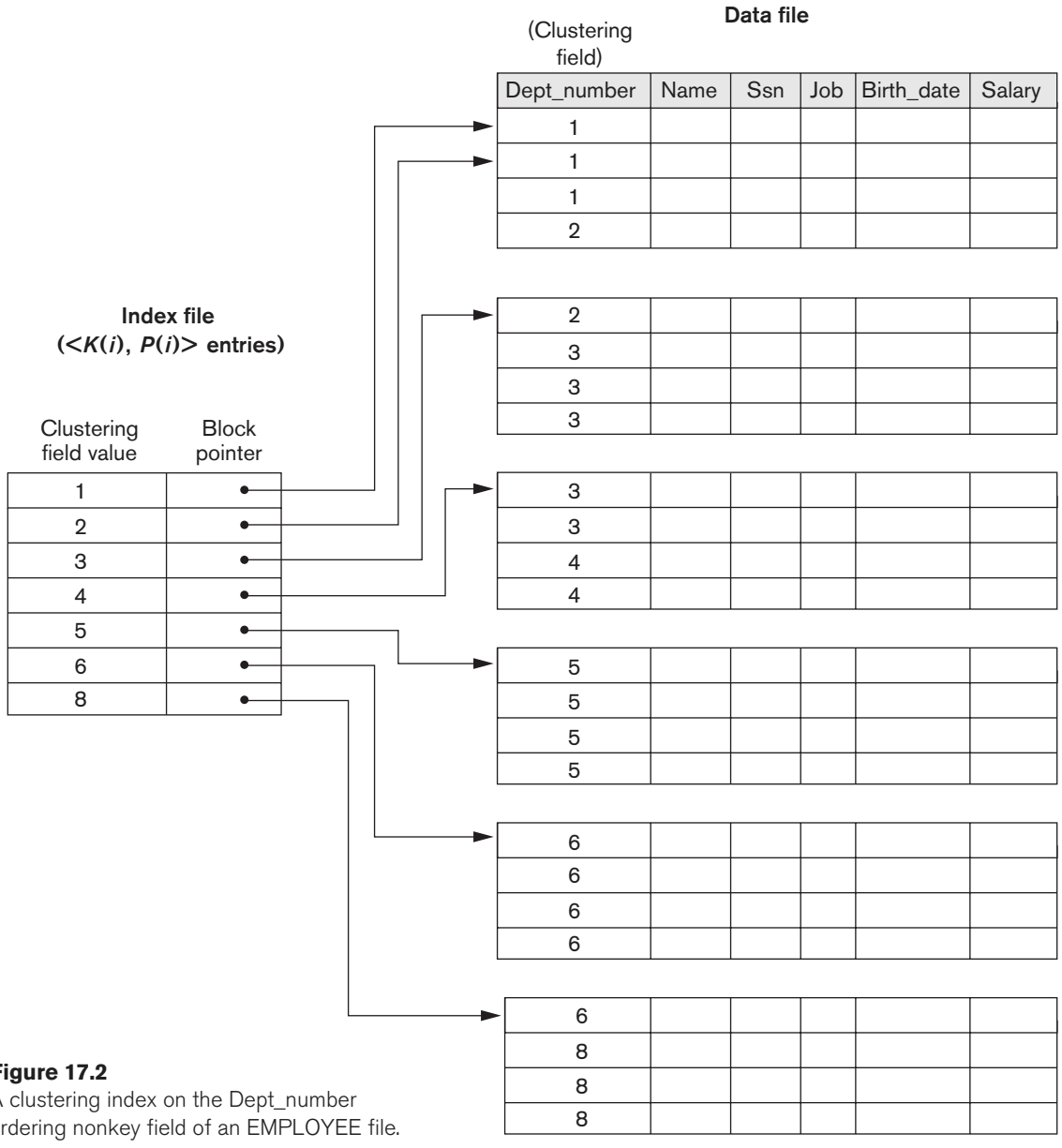


Figure 17.2
A clustering index on the Dept_number ordering nonkey field of an EMPLOYEE file.

There is some similarity between Figures 17.1, 17.2, and 17.3 and Figures 16.11 and 16.12. An index is somewhat similar to dynamic hashing (described in Section 16.8.3) and to the directory structures used for extendible hashing. Both are searched to find a pointer to the data block containing the desired record. A main difference is that an index search uses the values of the search field itself, whereas a hash directory search uses the binary hash value that is calculated by applying the hash function to the search field.

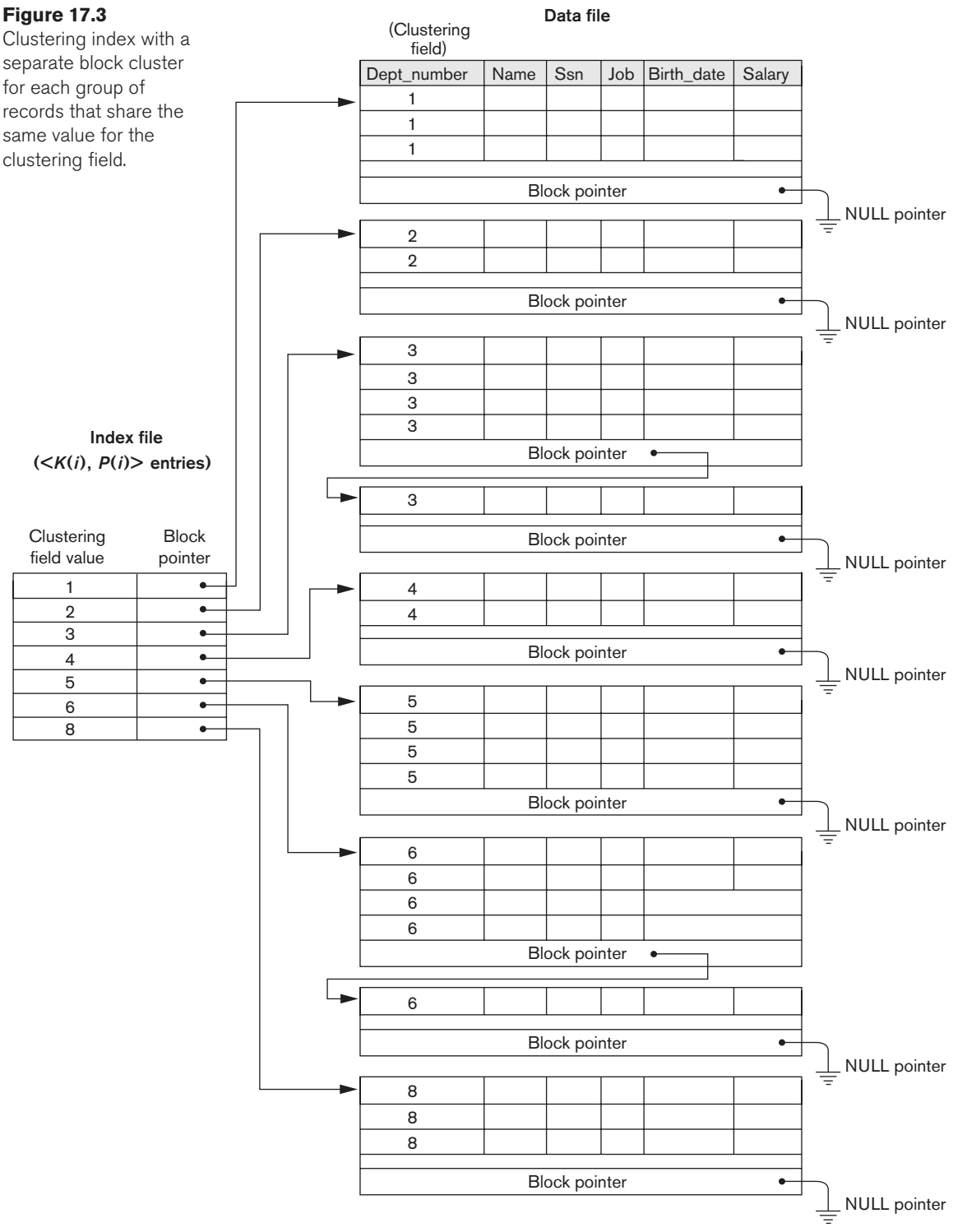


Figure 17.3
Clustering index with a separate block cluster for each group of records that share the same value for the clustering field.

17.1.3 Secondary Indexes

A **secondary index** provides a secondary means of accessing a data file for which some primary access already exists. The data file records could be ordered, unordered, or hashed. The secondary index may be created on a field that is a candidate key and has a unique value in every record, or on a nonkey field with duplicate values. The index is again an ordered file with two fields. The first field is of the same data type as some *nonordering field* of the data file that is an **indexing field**. The second field is either a *block* pointer or a *record* pointer. *Many* secondary indexes (and hence, indexing fields) can be created for the same file—each represents an additional means of accessing that file based on some specific field.

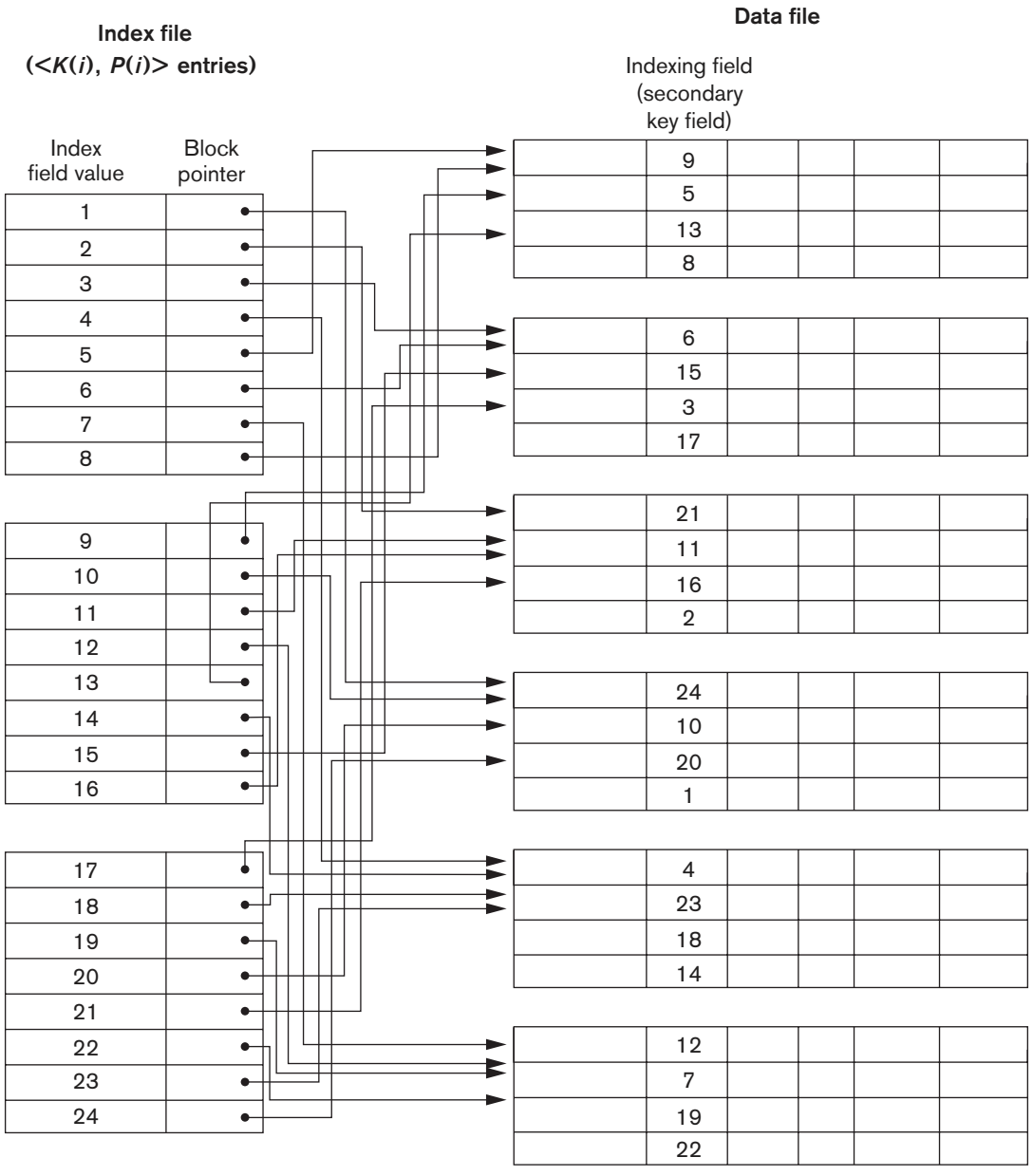
First we consider a secondary index access structure on a key (unique) field that has a *distinct value* for every record. Such a field is sometimes called a **secondary** key; in the relational model, this would correspond to any UNIQUE key attribute or to the primary key attribute of a table. In this case there is one index entry for *each record* in the data file, which contains the value of the field for the record and a pointer either to the block in which the record is stored or to the record itself. Hence, such an index is **dense**.

Again we refer to the two field values of index entry i as $<K(i), P(i)>$. The entries are **ordered** by value of $K(i)$, so we can perform a binary search. Because the records of the data file are *not* physically ordered by values of the secondary key field, we *cannot* use block anchors. That is why an index entry is created for each record in the data file, rather than for each block, as in the case of a primary index. Figure 17.4 illustrates a secondary index in which the pointers $P(i)$ in the index entries are *block pointers,* not record pointers. Once the appropriate disk block is transferred to a main memory buffer, a search for the desired record within the block can be carried out.

A secondary index usually needs more storage space and longer search time than does a primary index, because of its larger number of entries. However, the *improvement* in search time for an arbitrary record is much greater for a secondary index than for a primary index, since we would have to do a *linear search* on the data file if the secondary index did not exist. For a primary index, we could still use a binary search on the main file, even if the index did not exist. Example 3 illustrates the improvement in number of blocks accessed.

Example 3. Consider the file of Example 1 with $r = 300{,}000$ fixed-length records of size $R = 100$ bytes stored on a disk with block size $B = 4{,}096$ bytes. The file has $b = 7{,}500$ blocks, as calculated in Example 1. Suppose we want to search for a record with a specific value for the secondary key—a nonordering key field of the file that is $V = 9$ bytes long. Without the secondary index, to do a linear search on the file would require $b/2 = 7{,}500/2 = 3{,}750$ block accesses on the average. Suppose that we construct a secondary index on that *nonordering key* field of the file. As in Example 1, a block pointer is $P = 6$ bytes long, so each index entry is $R_i = (9 + 6) = 15$ bytes, and the blocking factor for the index is $bfr_i = \lfloor (B/R_i) \rfloor = \lfloor (4{,}096/15) \rfloor = 273$ index entries per block. In a dense secondary index such as this, the total number of index entries r_i is equal to the *number of records* in the data file, which is 300,000. The number of blocks needed for the index is hence $b_i = \lceil (r_i/bfr_i) \rceil = \lceil (300{,}000/273) \rceil = 1{,}099$ blocks.

Figure 17.4
A dense secondary index (with block pointers) on a nonordering key field of a file.



A binary search on this secondary index needs $\lceil (\log_2 b_i) \rceil = \lceil (\log_2 1{,}099) \rceil = 11$ block accesses. To search for a record using the index, we need an additional block access to the data file for a total of 11 + 1 = 12 block accesses—a vast improvement over the 3,750 block accesses needed on the average for a linear search, but slightly worse than the 6 block accesses required for the primary index. This difference arose because the primary index was nondense and hence shorter, with only 28 blocks in length as opposed to the 1,099 blocks dense index here.

We can also create a secondary index on a *nonkey, nonordering field* of a file. In this case, numerous records in the data file can have the same value for the indexing field. There are several options for implementing such an index:

- Option 1 is to include duplicate index entries with the same $K(i)$ value—one for each record. This would be a dense index.
- Option 2 is to have variable-length records for the index entries, with a repeating field for the pointer. We keep a list of pointers $<P(i, 1), \ldots, P(i, k)>$ in the index entry for $K(i)$—one pointer to each block that contains a record whose indexing field value equals $K(i)$. In either option 1 or option 2, the binary search algorithm on the index must be modified appropriately to account for a variable number of index entries per index key value.
- Option 3, which is more commonly used, is to keep the index entries themselves at a fixed length and have a single entry for each *index field value*, but to create *an extra level of indirection* to handle the multiple pointers. In this nondense scheme, the pointer $P(i)$ in index entry $<K(i), P(i)>$ points to a disk block, which contains a *set of record pointers*; each record pointer in that disk block points to one of the data file records with value $K(i)$ for the indexing field. If some value $K(i)$ occurs in too many records, so that their record pointers cannot fit in a single disk block, a cluster or linked list of blocks is used. This technique is illustrated in Figure 17.5. Retrieval via the index requires one or more additional block accesses because of the extra level, but the algorithms for searching the index and (more importantly) for inserting of new records in the data file are straightforward. The binary search algorithm is directly applicable to the index file since it is ordered. For range retrievals such as retrieving records where $V_1 \leq K \leq V_2$, block pointers may be used in the pool of pointers for each value instead of the record pointers. Then a union operation can be used on the pools of block pointers corresponding to the entries from V_1 to V_2 in the index to eliminate duplicates and the resulting blocks can be accessed. In addition, retrievals on complex selection conditions may be handled by referring to the record pointers from multiple non-key secondary indexes, without having to retrieve many unnecessary records from the data file (see Exercise 17.24).

Notice that a secondary index provides a **logical ordering** on the records by the indexing field. If we access the records in order of the entries in the secondary index, we get them in order of the indexing field. The primary and clustering indexes assume that the field used for **physical ordering** of records in the file is the same as the indexing field.

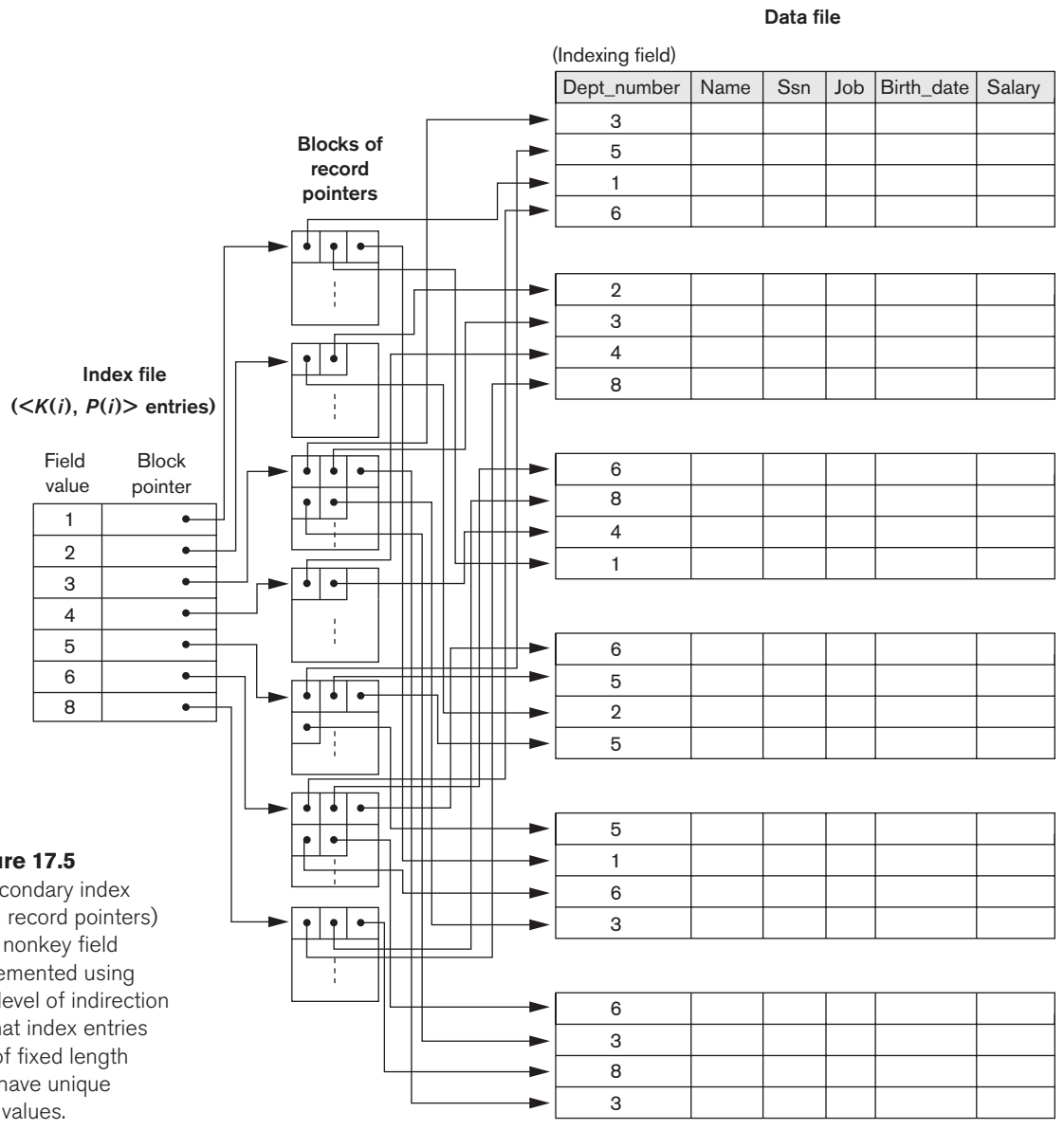


Figure 17.5
A secondary index (with record pointers) on a nonkey field implemented using one level of indirection so that index entries are of fixed length and have unique field values.

17.1.4 Summary

To conclude this section, we summarize the discussion of index types in two tables. Table 17.1 shows the index field characteristics of each type of ordered single-level index discussed—primary, clustering, and secondary. Table 17.2 summarizes the properties of each type of index by comparing the number of index entries and specifying which indexes are dense and which use block anchors of the data file.

Table 17.1 Types of Indexes Based on the Properties of the Indexing Field

	Index Field Used for Physical Ordering of the File	Index Field Not Used for Physical Ordering of the File
Indexing field is key	Primary index	Secondary index (Key)
Indexing field is nonkey	Clustering index	Secondary index (NonKey)

Table 17.2 Properties of Index Types

Type of Index	Number of (First-Level) Index Entries	Dense or Nondense (Sparse)	Block Anchoring on the Data File
Primary	Number of blocks in data file	Nondense	Yes
Clustering	Number of distinct index field values	Nondense	Yes/no[a]
Secondary (key)	Number of records in data file	Dense	No
Secondary (nonkey)	Number of records[b] or number of distinct index field values[c]	Dense or Nondense	No

[a]Yes if every distinct value of the ordering field starts a new block; no otherwise.
[b]For option 1.
[c]For options 2 and 3.

17.2 Multilevel Indexes

The indexing schemes we have described thus far involve an ordered index file. A binary search is applied to the index to locate pointers to a disk block or to a record (or records) in the file having a specific index field value. A binary search requires approximately ($\log_2 b_i$) block accesses for an index with b_i blocks because each step of the algorithm reduces the part of the index file that we continue to search by a factor of 2. This is why we take the log function to the base 2. The idea behind a **multilevel index** is to reduce the part of the index that we continue to search by bfr_i, the blocking factor for the index, which is larger than 2. Hence, the search space is reduced much faster. The value bfr_i is called the **fan-out** of the multilevel index, and we will refer to it by the symbol **fo**. Whereas we divide the *record search space* into two halves at each step during a binary search, we divide it n-ways (where n = the fan-out) at each search step using the multilevel index. Searching a multilevel index requires approximately ($\log_{fo} b_i$) block accesses, which is a substantially smaller number than for a binary search if the fan-out is larger than 2. In most cases, the fan-out is much larger than 2. Given a blocksize of 4,096, which is most common in today's DBMSs, the fan-out depends on how many (key + block pointer) entries fit within a block. With a 4-byte block pointer (which would accommodate $2^{32} - 1 = 4.2 * 10^9$ blocks) and a 9-byte key such as SSN, the fan-out comes to 315.

A multilevel index considers the index file, which we will now refer to as the **first** (or **base**) **level** of a multilevel index, as an *ordered file* with a *distinct value* for each

$K(i)$. Therefore, by considering the first-level index file as a sorted data file, we can create a primary index for the first level; this index to the first level is called the **second level** of the multilevel index. Because the second level is a primary index, we can use block anchors so that the second level has one entry for *each block* of the first level. The blocking factor bfr_i for the second level—and for all subsequent levels—is the same as that for the first-level index because all index entries are the same size; each has one field value and one block address. If the first level has r_1 entries, and the blocking factor—which is also the fan-out—for the index is $bfr_i = fo$, then the first level needs $\lceil (r_1/fo) \rceil$ blocks, which is therefore the number of entries r_2 needed at the second level of the index.

We can repeat this process for the second level. The **third level**, which is a primary index for the second level, has an entry for each second-level block, so the number of third-level entries is $r_3 = \lceil (r_2/fo) \rceil$. Notice that we require a second level only if the first level needs more than one block of disk storage, and, similarly, we require a third level only if the second level needs more than one block. We can repeat the preceding process until all the entries of some index level t fit in a single block. This block at the tth level is called the **top** index level.[6] Each level reduces the number of entries at the previous level by a factor of fo—the index fan-out—so we can use the formula $1 \leq (r_1/((fo)^t))$ to calculate t. Hence, a multilevel index with r_1 first-level entries will have approximately t levels, where $t = \lceil (\log_{fo}(r_1)) \rceil$. When searching the index, a single disk block is retrieved at each level. Hence, t disk blocks are accessed for an index search, where t is the *number of index levels*.

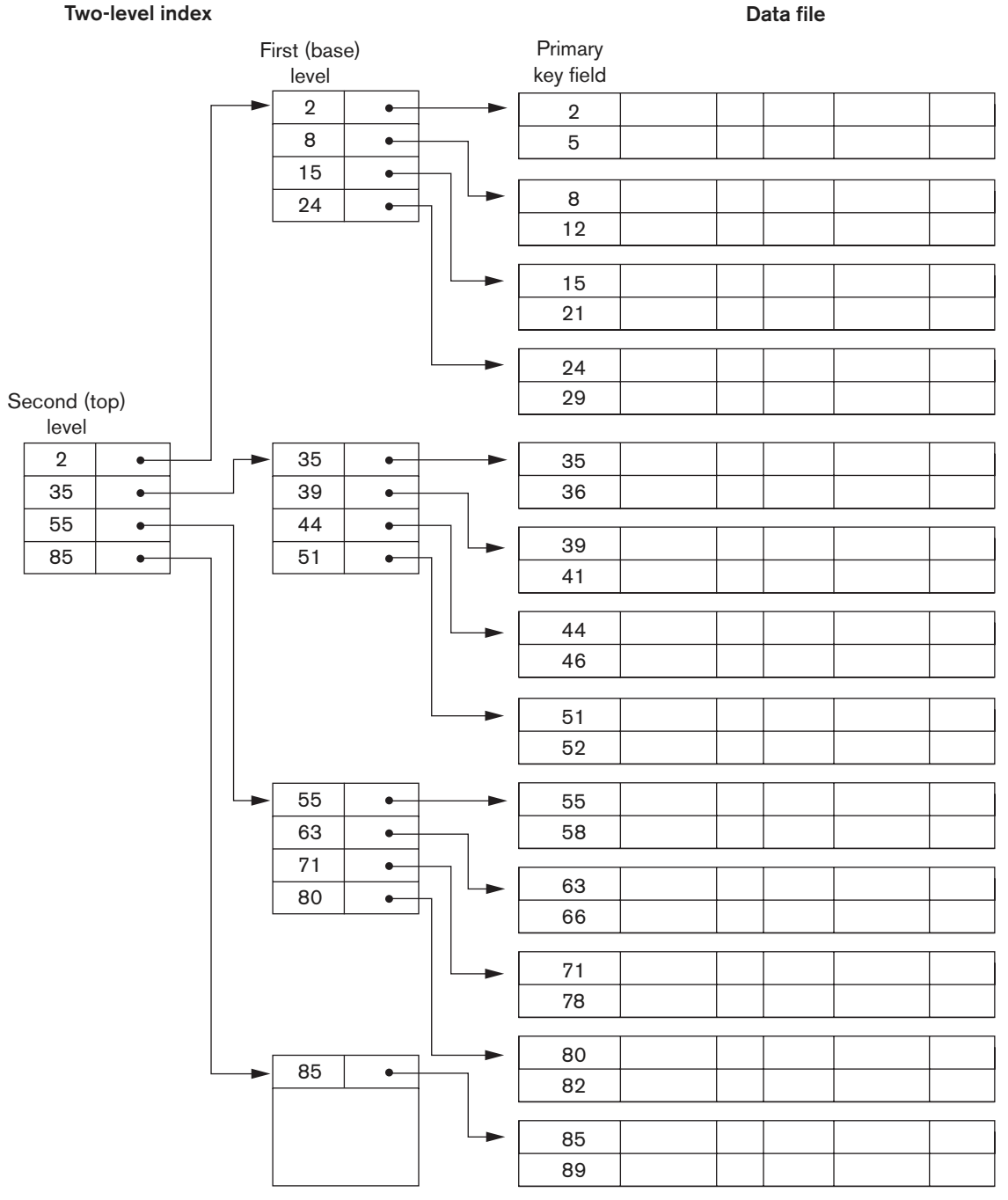
The multilevel scheme described here can be used on any type of index—whether it is primary, clustering, or secondary—as long as the first-level index has *distinct values for K(i) and fixed-length entries.* Figure 17.6 shows a multilevel index built over a primary index. Example 3 illustrates the improvement in number of blocks accessed when a multilevel index is used to search for a record.

Example 4. Suppose that the dense secondary index of Example 3 is converted into a multilevel index. We calculated the index blocking factor $bfr_i = 273$ index entries per block, which is also the fan-out fo for the multilevel index; the number of first-level blocks $b_1 = 1{,}099$ blocks was also calculated. The number of second-level blocks will be $b_2 = \lceil (b_1/fo) \rceil = \lceil (1{,}099/273) \rceil = 5$ blocks, and the number of third-level blocks will be $b_3 = \lceil (b_2/fo) \rceil = \lceil (5/273) \rceil = 1$ block. Hence, the third level is the top level of the index, and $t = 3$. To access a record by searching the multilevel index, we must access one block at each level plus one block from the data file, so we need $t + 1 = 3 + 1 = 4$ block accesses. Compare this to Example 3, where 12 block accesses were needed when a single-level index and binary search were used.

Notice that we could also have a multilevel primary index, which would be non-dense. Exercise 17.18(c) illustrates this case, where we *must* access the data block from the file before we can determine whether the record being searched for is in the file. For a dense index, this can be determined by accessing the first index level

[6]The numbering scheme for index levels used here is the reverse of the way levels are commonly defined for tree data structures. In tree data structures, t is referred to as level 0 (zero), $t - 1$ is level 1, and so on.

Figure 17.6
A two-level primary index resembling ISAM (indexed sequential access method) organization.



(without having to access a data block), since there is an index entry for *every* record in the file.

A common file organization used in business data processing is an ordered file with a multilevel primary index on its ordering key field. Such an organization is called an **indexed sequential file** and was used in a large number of early IBM systems. IBM's **ISAM** organization incorporates a two-level index that is closely related to the organization of the disk in terms of cylinders and tracks (see Section 16.2.1). The first level is a cylinder index, which has the key value of an anchor record for each cylinder of a disk pack occupied by the file and a pointer to the track index for the cylinder. The track index has the key value of an anchor record for each track in the cylinder and a pointer to the track. The track can then be searched sequentially for the desired record or block. Insertion is handled by some form of overflow file that is merged periodically with the data file. The index is re-created during file reorganization.

Algorithm 17.1 outlines the search procedure for a record in a data file that uses a nondense multilevel primary index with t levels. We refer to entry i at level j of the index as $<K_j(i), P_j(i)>$, and we search for a record whose primary key value is K. We assume that any overflow records are ignored. If the record is in the file, there must be some entry at level 1 with $K_1(i) \leq K < K_1(i+1)$ and the record will be in the block of the data file whose address is $P_1(i)$. Exercise 17.23 discusses modifying the search algorithm for other types of indexes.

Algorithm 17.1. Searching a Nondense Multilevel Primary Index with t Levels

```
(*We assume the index entry to be a block anchor that is the first key per block*)
p ← address of top-level block of index;
for j ← t step − 1 to 1 do
    begin
        read the index block (at jth index level) whose address is p;
        search block p for entry i such that K_j (i) ≤ K < K_j(i + 1)
    (* if K_j(i)
        is the last entry in the block, it is sufficient to satisfy K_j(i) ≤ K *);
        p ← P_j(i ) (* picks appropriate pointer at jth index level *)
    end;
    read the data file block whose address is p;
    search block p for record with key = K;
```

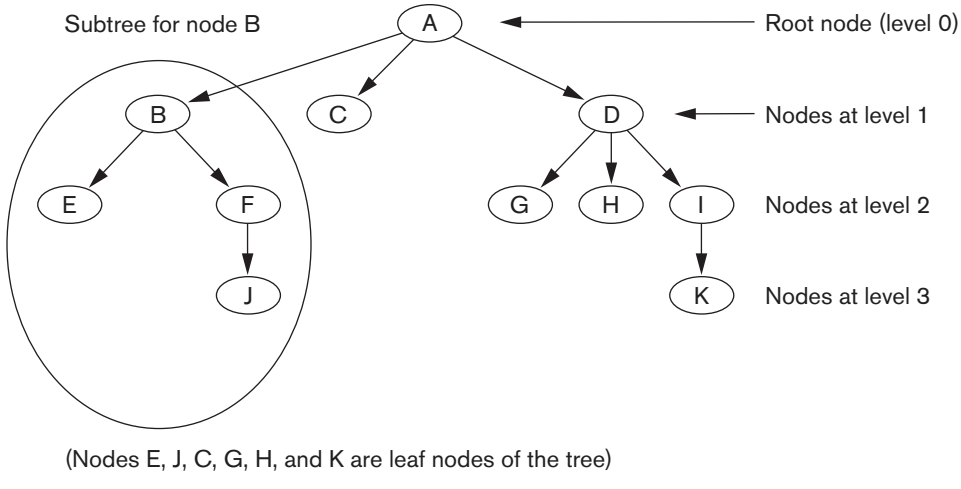
As we have seen, a multilevel index reduces the number of blocks accessed when searching for a record, given its indexing field value. We are still faced with the problems of dealing with index insertions and deletions, because all index levels are *physically ordered files.* To retain the benefits of using multilevel indexing while reducing index insertion and deletion problems, designers adopted a multilevel index called a **dynamic multilevel index** that leaves some space in each of its blocks for inserting new entries and uses appropriate insertion/deletion algorithms for creating and deleting new index blocks when the data file grows and shrinks. It is often implemented by using data structures called B-trees and B^+-trees, which we describe in the next section.

17.3 Dynamic Multilevel Indexes Using B-Trees and B+-Trees

B-trees and B+-trees are special cases of the well-known search data structure known as a **tree**. We briefly introduce the terminology used in discussing tree data structures. A **tree** is formed of **nodes**. Each node in the tree, except for a special node called the **root**, has one **parent** node and zero or more **child** nodes. The root node has no parent. A node that does not have any child nodes is called a **leaf** node; a nonleaf node is called an **internal** node. The **level** of a node is always one more than the level of its parent, with the level of the root node being *zero*.[7] A **subtree** of a node consists of that node and all its **descendant** nodes—its child nodes, the child nodes of its child nodes, and so on. A precise recursive definition of a subtree is that it consists of a node *n* and the subtrees of all the child nodes of *n*. Figure 17.7 illustrates a tree data structure. In this figure the root node is A, and its child nodes are B, C, and D. Nodes E, J, C, G, H, and K are leaf nodes. Since the leaf nodes are at different levels of the tree, this tree is called **unbalanced**.

In Section 17.3.1, we introduce search trees and then discuss B-trees, which can be used as dynamic multilevel indexes to guide the search for records in a data file. B-tree nodes are kept between 50 and 100 percent full, and pointers to the data blocks are stored in both internal nodes and leaf nodes of the B-tree structure. In Section 17.3.2 we discuss B+-trees, a variation of B-trees in which pointers to the data blocks of a file are stored only in leaf nodes, which can lead to fewer levels and

Figure 17.7
A tree data structure that shows an unbalanced tree.



[7]This standard definition of the level of a tree node, which we use throughout Section 17.3, is different from the one we gave for multilevel indexes in Section 17.2.

higher-capacity indexes. In the DBMSs prevalent in the market today, the common structure used for indexing is B^+-trees.

17.3.1 Search Trees and B-Trees

A **search tree** is a special type of tree that is used to guide the search for a record, given the value of one of the record's fields. The multilevel indexes discussed in Section 17.2 can be thought of as a variation of a search tree; each node in the multilevel index can have as many as *fo* pointers and *fo* key values, where *fo* is the index fan-out. The index field values in each node guide us to the next node, until we reach the data file block that contains the required records. By following a pointer, we restrict our search at each level to a subtree of the search tree and ignore all nodes not in this subtree.

Search Trees. A search tree is slightly different from a multilevel index. A **search tree of order** p is a tree such that each node contains *at most* $p - 1$ search values and p pointers in the order $<P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q>$, where $q \leq p$. Each P_i is a pointer to a child node (or a NULL pointer), and each K_i is a search value from some ordered set of values. All search values are assumed to be unique.[8] Figure 17.8 illustrates a node in a search tree. Two constraints must hold at all times on the search tree:

1. Within each node, $K_1 < K_2 < \ldots < K_{q-1}$.
2. For all values X in the subtree pointed at by P_i, we have $K_{i-1} < X < K_i$ for $1 < i < q$; $X < K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$ (see Figure 17.8).

Whenever we search for a value X, we follow the appropriate pointer P_i according to the formulas in condition 2 above. Figure 17.9 illustrates a search tree of order $p = 3$ and integer search values. Notice that some of the pointers P_i in a node may be NULL pointers.

We can use a search tree as a mechanism to search for records stored in a disk file. The values in the tree can be the values of one of the fields of the file, called the

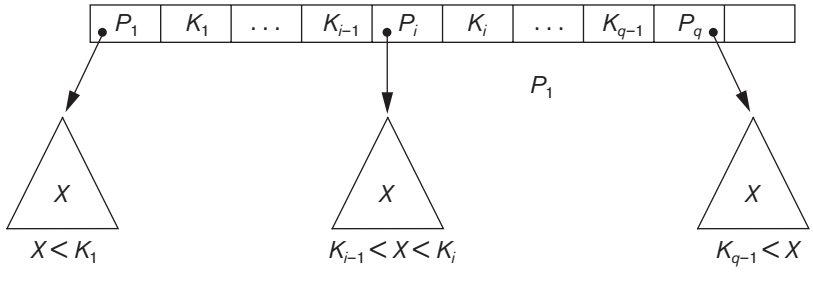


Figure 17.8
A node in a search tree with pointers to subtrees below it.

[8]This restriction can be relaxed. If the index is on a nonkey field, duplicate search values may exist and the node structure and the navigation rules for the tree may be modified.

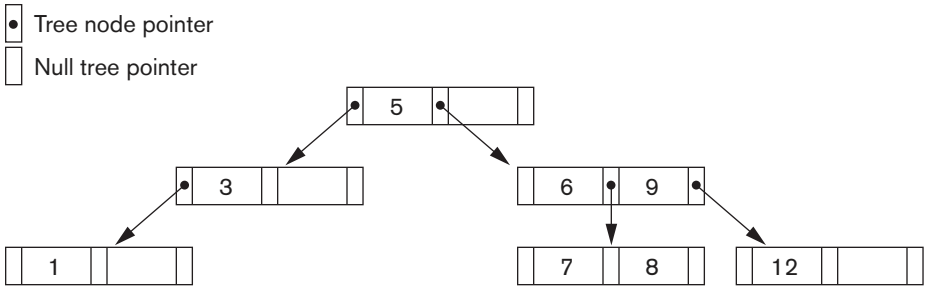


Figure 17.9
A search tree of order $p = 3$.

search field (which is the same as the index field if a multilevel index guides the search). Each key value in the tree is associated with a pointer to the record in the data file having that value. Alternatively, the pointer could be to the disk block containing that record. The search tree itself can be stored on disk by assigning each tree node to a disk block. When a new record is inserted in the file, we must update the search tree by inserting an entry in the tree containing the search field value of the new record and a pointer to the new record.

Algorithms are necessary for inserting and deleting search values into and from the search tree while maintaining the preceding two constraints. In general, these algorithms do not guarantee that a search tree is **balanced**, meaning that all of its leaf nodes are at the same level.[9] The tree in Figure 17.7 is not balanced because it has leaf nodes at levels 1, 2, and 3. The goals for balancing a search tree are as follows:

- To guarantee that nodes are evenly distributed, so that the depth of the tree is minimized for the given set of keys and that the tree does not get skewed with some nodes being at very deep levels
- To make the search speed uniform, so that the average time to find any random key is roughly the same

Minimizing the number of levels in the tree is one goal, another implicit goal is to make sure that the index tree does not need too much restructuring as records are inserted into and deleted from the main file. Thus we want the nodes to be as full as possible and do not want any nodes to be empty if there are too many deletions. Record deletion may leave some nodes in the tree nearly empty, thus wasting storage space and increasing the number of levels. The B-tree addresses both of these problems by specifying additional constraints on the search tree.

B-Trees. The B-tree has additional constraints that ensure that the tree is always balanced and that the space wasted by deletion, if any, never becomes excessive. The algorithms for insertion and deletion, though, become more complex in order to maintain these constraints. Nonetheless, most insertions and deletions are simple processes; they become complicated only under special circumstances—namely, whenever we attempt an insertion into a node that is already full or a deletion from

[9]The definition of *balanced* is different for binary trees. Balanced binary trees are known as *AVL trees*.

a node that makes it less than half full. More formally, a **B-tree of order** p, when used as an access structure on a *key field* to search for records in a data file, can be defined as follows:

1. Each internal node in the B-tree (Figure 17.10(a)) is of the form

   $$<P_1, <K_1, Pr_1>, P_2, <K_2, Pr_2>, \ldots, <K_{q-1}, Pr_{q-1}>, P_q>$$

   where $q \leq p$. Each P_i is a **tree pointer**—a pointer to another node in the B-tree. Each Pr_i is a **data pointer**[10]—a pointer to the record whose search key field value is equal to K_i (or to the data file block containing that record).
2. Within each node, $K_1 < K_2 < \ldots < K_{q-1}$.
3. For all search key field values X in the subtree pointed at by P_i (the ith subtree, see Figure 17.10(a)), we have:

   $$K_{i-1} < X < K_i \text{ for } 1 < i < q; X < K_i \text{ for } i = 1; \text{ and } K_{i-1} < X \text{ for } i = q$$

4. Each node has at most p tree pointers.
5. Each node, except the root and leaf nodes, has at least $\lceil (p/2) \rceil$ tree pointers. The root node has at least two tree pointers unless it is the only node in the tree.
6. A node with q tree pointers, $q \leq p$, has $q - 1$ search key field values (and hence has $q - 1$ data pointers).
7. All leaf nodes are at the same level. Leaf nodes have the same structure as internal nodes except that all of their *tree pointers* P_i are NULL.

Figure 17.10(b) illustrates a B-tree of order $p = 3$. Notice that all search values K in the B-tree are unique because we assumed that the tree is used as an access structure on a key field. If we use a B-tree *on a nonkey field*, we must change the definition of the file pointers Pr_i to point to a block—or a cluster of blocks—that contain the pointers to the file records. This extra level of indirection is similar to option 3, discussed in Section 17.1.3, for secondary indexes.

A B-tree starts with a single root node (which is also a leaf node) at level 0 (zero). Once the root node is full with $p - 1$ search key values and we attempt to insert another entry in the tree, the root node splits into two nodes at level 1. Only the middle value is kept in the root node, and the rest of the values are split evenly between the other two nodes. When a nonroot node is full and a new entry is inserted into it, that node is split into two nodes at the same level, and the middle entry is moved to the parent node along with two pointers to the new split nodes. If the parent node is full, it is also split. Splitting can propagate all the way to the root node, creating a new level if the root is split. We do not discuss algorithms for B-trees in detail in this text,[11] but we outline search and insertion procedures for B^+-trees in the next section.

[10]A data pointer is either a block address or a record address; the latter is essentially a block address and a record offset within the block.

[11]For details on insertion and deletion algorithms for B-trees, consult Ramakrishnan and Gehrke (2003).

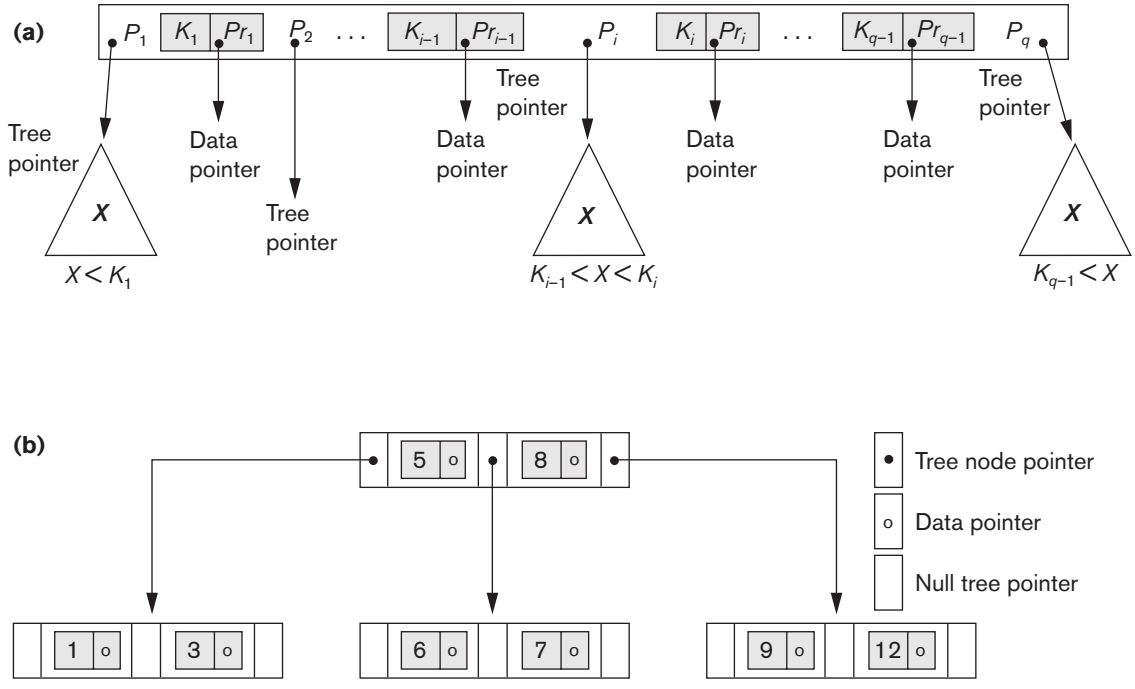


Figure 17.10
B-tree structures. (a) A node in a B-tree with $q - 1$ search values. (b) A B-tree of order $p = 3$. The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.

If deletion of a value causes a node to be less than half full, it is combined with its neighboring nodes, and this can also propagate all the way to the root. Hence, deletion can reduce the number of tree levels. It has been shown by analysis and simulation that, after numerous random insertions and deletions on a B-tree, the nodes are approximately 69% full when the number of values in the tree stabilizes. This is also true of B^+-trees. If this happens, node splitting and combining will occur only rarely, so insertion and deletion become quite efficient. If the number of values grows, the tree will expand without a problem—although splitting of nodes may occur, so some insertions will take more time. Each B-tree node can have *at most p* tree pointers, $p - 1$ data pointers, and $p - 1$ search key field values (see Figure 17.10(a)).

In general, a B-tree node may contain additional information needed by the algorithms that manipulate the tree, such as the number of entries q in the node and a pointer to the parent node. Next, we illustrate how to calculate the number of blocks and levels for a B-tree.

Example 5. Suppose that the search field is a nonordering key field, and we construct a B-tree on this field with $p = 23$. Assume that each node of the B-tree is 69% full. Each node, on the average, will have $p * 0.69 = 23 * 0.69$ or approximately

16 pointers and, hence, 15 search key field values. The **average fan-out** *fo* = 16. We can start at the root and see how many values and pointers can exist, on the average, at each subsequent level:

Root:	1 node	15 key entries	16 pointers
Level 1:	16 nodes	240 key entries	256 pointers
Level 2:	256 nodes	3,840 key entries	4,096 pointers
Level 3:	4,096 nodes	61,440 key entries	

At each level, we calculated the number of key entries by multiplying the total number of pointers at the previous level by 15, the average number of entries in each node. Hence, for the given block size (512 bytes), record/data pointer size (7 bytes), tree/block pointer size (6 bytes), and search key field size (9bytes), a two-level B-tree of order 23 with 69% occupancy holds 3,840 + 240 + 15 = 4,095 entries on the average; a three-level B-tree holds 65,535 entries on the average.

B-trees are sometimes used as **primary file organizations**. In this case, *whole records* are stored within the B-tree nodes rather than just the <search key, record pointer> entries. This works well for files with a relatively *small number of records* and a *small record size.* Otherwise, the fan-out and the number of levels become too great to permit efficient access.

In summary, B-trees provide a multilevel access structure that is a balanced tree structure in which each node is at least half full. Each node in a B-tree of order p can have at most $p - 1$ search values.

17.3.2 B^+-Trees

Most implementations of a dynamic multilevel index use a variation of the B-tree data structure called a **B^+-tree**. In a B-tree, every value of the search field appears once at some level in the tree, along with a data pointer. In a B^+-tree, data pointers are stored *only at the leaf nodes* of the tree; hence, the structure of leaf nodes differs from the structure of internal nodes. The leaf nodes have an entry for *every* value of the search field, along with a data pointer to the record (or to the block that contains this record) if the search field is a key field. For a nonkey search field, the pointer points to a block containing pointers to the data file records, creating an extra level of indirection.

The leaf nodes of the B^+-tree are usually linked to provide ordered access on the search field to the records. These leaf nodes are similar to the first (base) level of an index. Internal nodes of the B^+-tree correspond to the other levels of a multilevel index. Some search field values from the leaf nodes are *repeated* in the internal nodes of the B^+-tree to guide the search. The structure of the *internal nodes* of a B^+-tree of order p (Figure 17.11(a)) is as follows:

1. Each internal node is of the form

   $$<P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q>$$

   where $q \leq p$ and each P_i is a **tree pointer**.

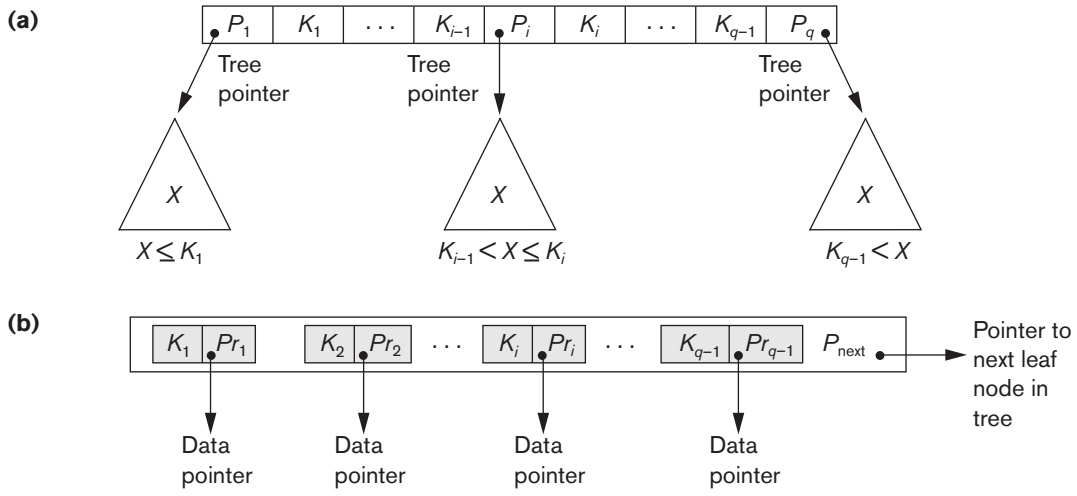


Figure 17.11
The nodes of a B+-tree. (a) Internal node of a B+-tree with $q - 1$ search values. (b) Leaf node of a B+-tree with $q - 1$ search values and $q - 1$ data pointers.

2. Within each internal node, $K_1 < K_2 < \ldots < K_{q-1}$.
3. For all search field values X in the subtree pointed at by P_i, we have $K_{i-1} < X \leq K_i$ for $1 < i < q$; $X \leq K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$ (see Figure 17.11(a)).[12]
4. Each internal node has at most p tree pointers.
5. Each internal node, except the root, has at least $\lceil (p/2) \rceil$ tree pointers. The root node has at least two tree pointers if it is an internal node.
6. An internal node with q pointers, $q \leq p$, has $q - 1$ search field values.

The structure of the *leaf nodes* of a B+-tree of order p (Figure 17.11(b)) is as follows:

1. Each leaf node is of the form

   $<<K_1, Pr_1>, <K_2, Pr_2>, \ldots, <K_{q-1}, Pr_{q-1}>, P_{\text{next}}>$

   where $q \leq p$, each Pr_i is a data pointer, and P_{next} points to the next *leaf node* of the B+-tree.
2. Within each leaf node, $K_1 \leq K_2 \ldots, K_{q-1}, q \leq p$.
3. Each Pr_i is a **data pointer** that points to the record whose search field value is K_i or to a file block containing the record (or to a block of record pointers that point to records whose search field value is K_i if the search field is not a key).
4. Each leaf node has at least $\lceil (p/2) \rceil$ values.
5. All leaf nodes are at the same level.

[12]Our definition follows Knuth (1998). One can define a B+-tree differently by exchanging the $<$ and $\leq$ symbols ($K_{i-1} \leq X < K_i$; $K_{q-1} \leq$ X), but the principles remain the same.

The pointers in internal nodes are *tree pointers* to blocks that are tree nodes, whereas the pointers in leaf nodes are *data pointers* to the data file records or blocks—except for the P_{next} pointer, which is a tree pointer to the next leaf node. By starting at the leftmost leaf node, it is possible to traverse leaf nodes as a linked list, using the P_{next} pointers. This provides ordered access to the data records on the indexing field. A $P_{previous}$ pointer can also be included. For a B^+-tree on a nonkey field, an extra level of indirection is needed similar to the one shown in Figure 17.5, so the *Pr* pointers are block pointers to blocks that contain a set of record pointers to the actual records in the data file, as discussed in option 3 of Section 17.1.3.

Because entries in the *internal nodes* of a B^+-tree include search values and tree pointers without any data pointers, more entries can be packed into an internal node of a B^+-tree than for a similar B-tree. Thus, for the same block (node) size, the order p will be larger for the B^+-tree than for the B-tree, as we illustrate in Example 6. This can lead to fewer B^+-tree levels, improving search time. Because the structures for internal and for leaf nodes of a B^+-tree are different, the order p can be different. We will use p to denote the order for *internal nodes* and p_{leaf} to denote the order for *leaf nodes*, which we define as being the maximum number of data pointers in a leaf node.

Example 6. To calculate the order p of a B^+-tree, suppose that the search key field is $V = 9$ bytes long, the block size is $B = 512$ bytes, a record pointer is $Pr = 7$ bytes, and a block pointer/tree pointer is $P = 6$ bytes. An internal node of the B^+-tree can have up to p tree pointers and $p - 1$ search field values; these must fit into a single block. Hence, we have:

$$(p * P) + ((p - 1) * V) \leq B$$
$$(p * 6) + ((p - 1) * 9) \leq 512$$
$$(15 * p) \leq 512$$

We can choose p to be the largest value satisfying the above inequality, which gives $p = 34$. This is larger than the value of 23 for the B-tree (it is left to the reader to compute the order of the B-tree assuming same size pointers), resulting in a larger fan-out and more entries in each internal node of a B^+-tree than in the corresponding B-tree. The leaf nodes of the B^+-tree will have the same number of values and pointers, except that the pointers are data pointers and a next pointer. Hence, the order p_{leaf} for the leaf nodes can be calculated as follows:

$$(p_{leaf} * (Pr + V)) + P \leq B$$
$$(p_{leaf} * (7 + 9)) + 6 \leq 512$$
$$(16 * p_{leaf}) \leq 506$$

It follows that each leaf node can hold up to $p_{leaf} = 31$ key value/data pointer combinations, assuming that the data pointers are record pointers.

As with the B-tree, we may need additional information—to implement the insertion and deletion algorithms—in each node. This information can include the type of node (internal or leaf), the number of current entries q in the node, and pointers to the parent and sibling nodes. Hence, before we do the above calculations for p

and p_{leaf}, we should reduce the block size by the amount of space needed for all such information. The next example illustrates how we can calculate the number of entries in a B+-tree.

Example 7. Suppose that we construct a B+-tree on the field in Example 6. To calculate the approximate number of entries in the B+-tree, we assume that each node is 69% full. On the average, each internal node will have 34 * 0.69 or approximately 23 pointers, and hence 22 values. Each leaf node, on the average, will hold 0.69 * p_{leaf} = 0.69 * 31 or approximately 21 data record pointers. A B+-tree will have the following average number of entries at each level:

Root:	1 node	22 key entries	23 pointers
Level 1:	23 nodes	506 key entries	529 pointers
Level 2:	529 nodes	11,638 key entries	12,167 pointers
Leaf level:	12,167 nodes	255,507 data record pointers	

For the block size, pointer size, and search field size as in Example 6, a three-level B+-tree holds up to 255,507 record pointers, with the average 69% occupancy of nodes. Note that we considered the leaf node differently from the nonleaf nodes and computed the data pointers in the leaf node to be 12,167 * 21 based on 69% occupancy of the leaf node, which can hold 31 keys with data pointers. Compare this to the 65,535 entries for the corresponding B-tree in Example 5. Because a B-tree includes a data/record pointer along with each search key at all levels of the tree, it tends to accommodate less number of keys for a given number of index levels. This is the main reason that B+-trees are preferred to B-trees as indexes to database files. Most DBMSs, such as Oracle, are creating all indexes as B+-trees.

Search, Insertion, and Deletion with B+-Trees. Algorithm 17.2 outlines the procedure using the B+-tree as the access structure to search for a record. Algorithm 17.3 illustrates the procedure for inserting a record in a file with a B+-tree access structure. These algorithms assume the existence of a key search field, and they must be modified appropriately for the case of a B+-tree on a nonkey field. We illustrate insertion and deletion with an example.

Algorithm 17.2. Searching for a Record with Search Key Field Value *K*, Using a B+-Tree

```
n ← block containing root node of B+-tree;
read block n;
while (n is not a leaf node of the B+-tree) do
    begin
    q ← number of tree pointers in node n;
    if K ≤ n.K1 (*n.Ki refers to the ith search field value in node n*)
        then n ← n.P1 (*n.Pi refers to the ith tree pointer in node n*)
        else if K > n.Kq–1
            then n ← n.Pq
```

```
            else begin
                search node n for an entry i such that n.K_{i-1} < K ≤ n.K_i;
                    n ← n.P_i
                    end;
        read block n
        end;
search block n for entry (K_i, Pr_i) with K = K_i; (* search leaf node *)
if found
    then read data file block with address Pr_i and retrieve record
    else the record with search field value K is not in the data file;
```

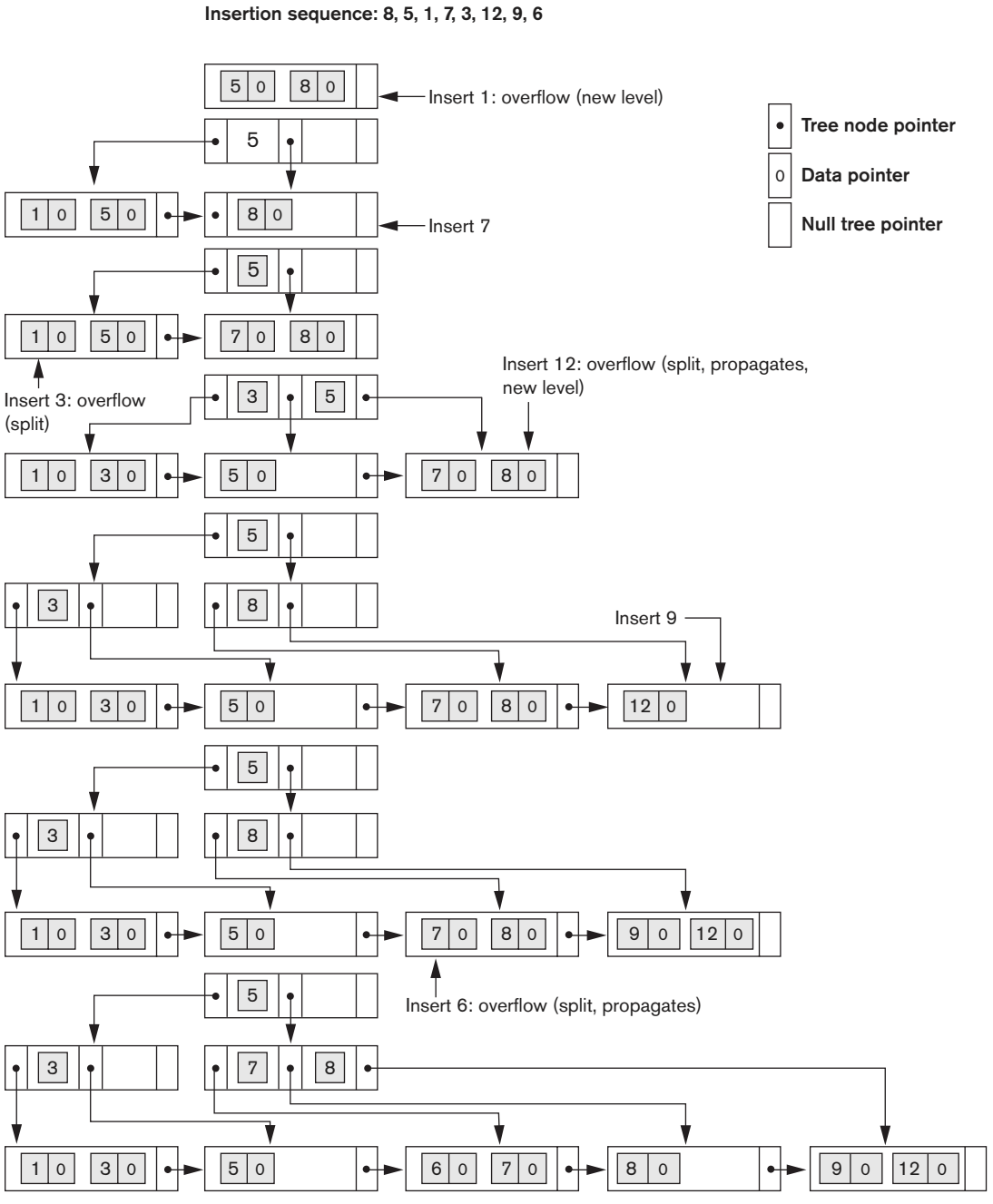
Algorithm 17.3. Inserting a Record with Search Key Field Value K in a B^+-Tree of Order p

```
n ← block containing root node of B+-tree;
read block n; set stack S to empty;
while (n is not a leaf node of the B+-tree) do
    begin
    push address of n on stack S;
        (*stack S holds parent nodes that are needed in case of split*)
    q ← number of tree pointers in node n;
    if K ≤ n.K_1 (*n.K_i refers to the ith search field value in node n*)
        then n ← n.P_1 (*n.P_i refers to the ith tree pointer in node n*)
        else if K ← n.K_{q-1}
            then n ← n.P_q
            else begin
                search node n for an entry i such that n.K_{i-1} < K ≤ n.K_i;
                n ← n.P_i
                end;
        read block n
    end;
search block n for entry (K_i,Pr_i) with K = K_i; (*search leaf node n*)
if found
    then record already in file; cannot insert
    else (*insert entry in B+-tree to point to record*)
        begin
        create entry (K, Pr) where Pr points to the new record;
        if leaf node n is not full
            then insert entry (K, Pr) in correct position in leaf node n
            else begin (*leaf node n is full with p_leaf record pointers; is split*)
                copy n to temp (*temp is an oversize leaf node to hold extra entries*);
                insert entry (K, Pr) in temp in correct position;
                (*temp now holds p_leaf + 1 entries of the form (K_i, Pr_i)*)
                new ← a new empty leaf node for the tree; new.P_next ← n.P_next ;
                j ← ⌈ (p_leaf + 1)/2 ⌉ ;
                n ← first j entries in temp (up to entry (K_j, Pr_j)); n.P_next ← new;
```

```
            new ← remaining entries in temp; K ← K_j ;
            (*now we must move (K, new) and insert in parent internal node;
              however, if parent is full, split may propagate*)
            finished ← false;
            repeat
            if stack S is empty
                then (←no parent node; new root node is created for the tree*)
                    begin
                    root ← a new empty internal node for the tree;
                    root ← <n, K, new>; finished ← true;
                    end
                else begin
                    n ← pop stack S;
                    if internal node n is not full
                        then
                            begin (*parent node not full; no split*)
                            insert (K, new) in correct position in internal node n;
                            finished ← true
                            end
                        else begin (*internal node n is full with p tree pointers;
                                    overflow condition; node is split*)
                        copy n to temp (*temp is an oversize internal node*);
                        insert (K, new) in temp in correct position;
                        (*temp now has p + 1 tree pointers*)
                        new ← a new empty internal node for the tree;
                        j ← ⌊((p + 1)/2⌋ ;
                        n ← entries up to tree pointer P_j in temp;
                        (*n contains <P_1, K_1, P_2, K_2, ... , P_{j–1}, K_{j–1}, P_j >*)
                        new ← entries from tree pointer P_{j+1} in temp;
                        (*new contains < P_{j+1}, K_{j+1}, ... , K_{p–1}, P_p, K_p, P_{p+1} >*)
                        K ← K_j
                        (*now we must move (K, new) and insert in
                          parentinternal node*)
                    end
                end
            until finished
            end;
    end;
```

Figure 17.12 illustrates insertion of records in a B+-tree of order $p = 3$ and $p_{leaf} = 2$. First, we observe that the root is the only node in the tree, so it is also a leaf node. As soon as more than one level is created, the tree is divided into internal nodes and leaf nodes. Notice that *every key value must exist at the leaf level*, because all data pointers are at the leaf level. However, only some values exist in internal nodes to guide the search. Notice also that every value appearing in an internal node also appears as *the rightmost value* in the leaf level of the subtree pointed at by the tree pointer to the left of the value.

Figure 17.12
An example of insertion in a B^+-tree with $p = 3$ and $p_{leaf} = 2$.



When a *leaf node* is full and a new entry is inserted there, the node *overflows* and must be split. The first $j = \lceil ((p_{\text{leaf}} + 1)/2) \rceil$ entries in the original node are kept there, and the remaining entries are moved to a new leaf node. The jth search value is replicated in the parent internal node, and an extra pointer to the new node is created in the parent. These must be inserted in the parent node in their correct sequence. If the parent internal node is full, the new value will cause it to overflow also, so it must be split. The entries in the internal node up to P_j—the jth tree pointer after inserting the new value and pointer, where $j = \lfloor ((p + 1)/2) \rfloor$—are kept, whereas the jth search value is moved to the parent, not replicated. A new internal node will hold the entries from P_{j+1} to the end of the entries in the node (see Algorithm 17.3). This splitting can propagate all the way up to create a new root node and hence a new level for the B+-tree.

Figure 17.13 illustrates deletion from a B+-tree. When an entry is deleted, it is always removed from the leaf level. If it happens to occur in an internal node, it must also be removed from there. In the latter case, the value to its left in the leaf node must replace it in the internal node because that value is now the rightmost entry in the subtree. Deletion may cause **underflow** by reducing the number of entries in the leaf node to below the minimum required. In this case, we try to find a sibling leaf node—a leaf node directly to the left or to the right of the node with underflow—and redistribute the entries among the node and its **sibling** so that both are at least half full; otherwise, the node is merged with its siblings and the number of leaf nodes is reduced. A common method is to try to **redistribute** entries with the left sibling; if this is not possible, an attempt to redistribute with the right sibling is made. If this is also not possible, the three nodes are merged into two leaf nodes. In such a case, underflow may propagate to **internal** nodes because one fewer tree pointer and search value are needed. This can propagate and reduce the tree levels.

Notice that implementing the insertion and deletion algorithms may require parent and sibling pointers for each node, or the use of a stack as in Algorithm 17.3. Each node should also include the number of entries in it and its type (leaf or internal). Another alternative is to implement insertion and deletion as recursive procedures.[13]

Variations of B-Trees and B+-Trees. To conclude this section, we briefly mention some variations of B-trees and B+-trees. In some cases, constraint 5 on the B-tree (or for the internal nodes of the B+–tree, except the root node), which requires each node to be at least half full, can be changed to require each node to be at least two-thirds full. In this case the B-tree has been called a **B*-tree**. In general, some systems allow the user to choose a **fill factor** between 0.5 and 1.0, where the latter means that the B-tree (index) nodes are to be completely full. It is also possible to specify two fill factors for a B+-tree: one for the leaf level and one for the internal nodes of the tree. When the index is first constructed, each node is filled up

[13]For more details on insertion and deletion algorithms for B+-trees, consult Ramakrishnan and Gehrke (2003).

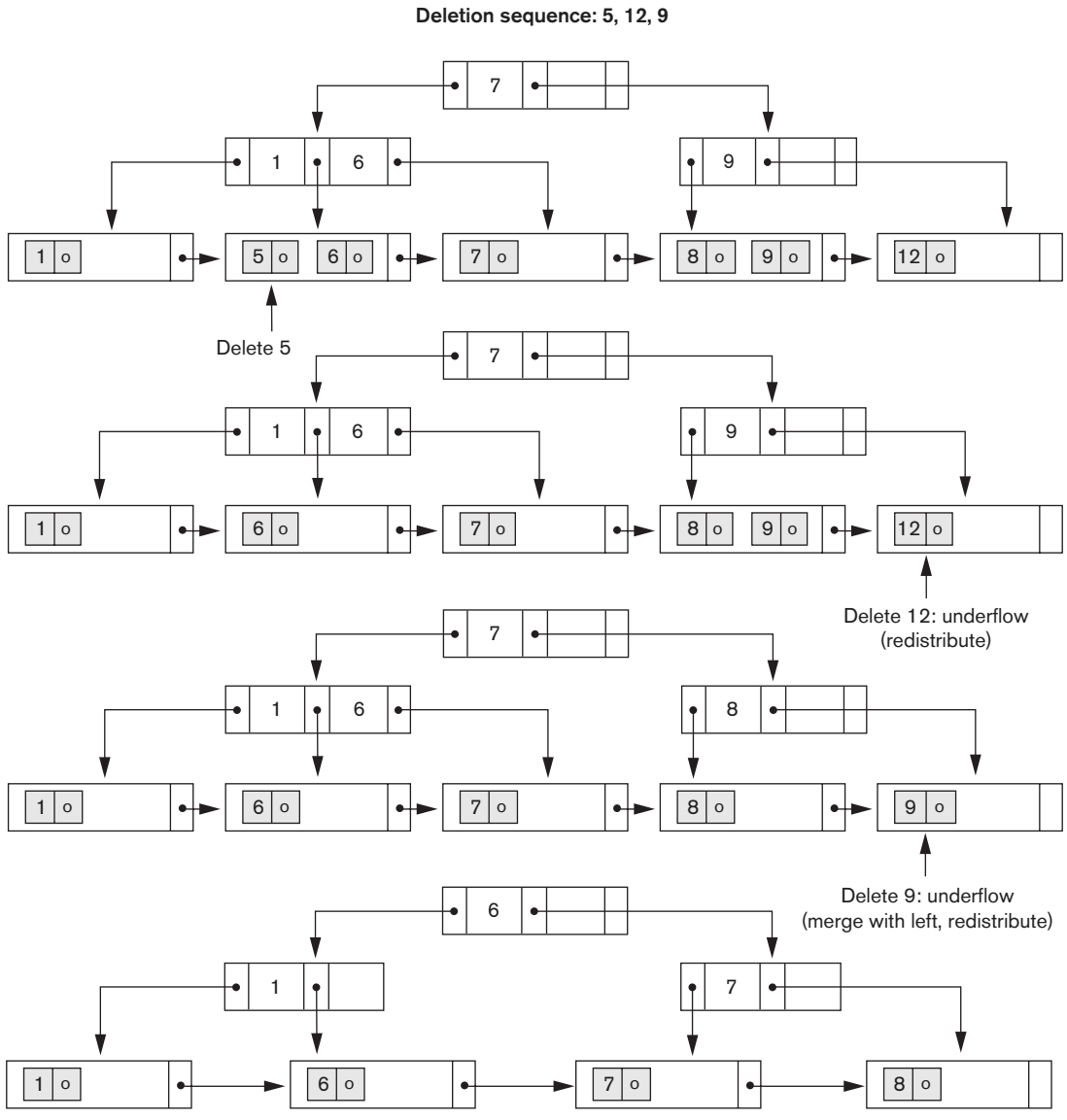


Figure 17.13
An example of deletion from a B^+-tree.

to approximately the fill factors specified. Some investigators have suggested relaxing the requirement that a node be half full, and instead allow a node to become completely empty before merging, to simplify the deletion algorithm. Simulation studies show that this does not waste too much additional space under randomly distributed insertions and deletions.

17.4 Indexes on Multiple Keys

In our discussion so far, we have assumed that the primary or secondary keys on which files were accessed were single attributes (fields). In many retrieval and update requests, multiple attributes are involved. If a certain combination of attributes is used frequently, it is advantageous to set up an access structure to provide efficient access by a key value that is a combination of those attributes.

For example, consider an EMPLOYEE file containing attributes Dno (department number), Age, Street, City, Zip_code, Salary and Skill_code, with the key of Ssn (Social Security number). Consider the query: *List the employees in department number 4 whose age is 59*. Note that both Dno and Age are nonkey attributes, which means that a search value for either of these will point to multiple records. The following alternative search strategies may be considered:

1. Assuming Dno has an index, but Age does not, access the records having Dno = 4 using the index, and then select from among them those records that satisfy Age = 59.
2. Alternately, if Age is indexed but Dno is not, access the records having Age = 59 using the index, and then select from among them those records that satisfy Dno = 4.
3. If indexes have been created on both Dno and Age, both indexes may be used; each gives a set of records or a set of pointers (to blocks or records). An intersection of these sets of records or pointers yields those records or pointers that satisfy both conditions.

All of these alternatives eventually give the correct result. However, if the set of records that meet each condition (Dno = 4 or Age = 59) individually are large, yet only a few records satisfy the combined condition, then none of the above is an efficient technique for the given search request. Note also that queries such as "find the minimum or maximum age among all employees" can be answered just by using the index on Age, without going to the data file. Finding the maximum or minimum age within Dno = 4, however, would not be answerable just by processing the index alone. Also, listing the departments in which employees with Age = 59 work will also not be possible by processing just the indexes. A number of possibilities exist that would treat the combination <Dno, Age> or <Age, Dno> as a search key made up of multiple attributes. We briefly outline these techniques in the following sections. We will refer to keys containing multiple attributes as **composite keys**.

17.4.1 Ordered Index on Multiple Attributes

All the discussion in this chapter so far still applies if we create an index on a search key field that is a combination of <Dno, Age>. The search key is a pair of values <4, 59> in the above example. In general, if an index is created on attributes $<A_1, A_2, \ldots, A_n>$, the search key values are tuples with n values: $<v_1, v_2, \ldots, v_n>$.

A lexicographic ordering of these tuple values establishes an order on this composite search key. For our example, all of the department keys for department number

3 precede those for department number 4. Thus <3, *n*> precedes <4, *m*> for any values of *m* and *n*. The ascending key order for keys with Dno = 4 would be <4, 18>, <4, 19>, <4, 20>, and so on. Lexicographic ordering works similarly to ordering of character strings. An index on a composite key of *n* attributes works similarly to any index discussed in this chapter so far.

17.4.2 Partitioned Hashing

Partitioned hashing is an extension of static external hashing (Section 16.8.2) that allows access on multiple keys. It is suitable only for equality comparisons; range queries are not supported. In partitioned hashing, for a key consisting of *n* components, the hash function is designed to produce a result with *n* separate hash addresses. The bucket address is a concatenation of these *n* addresses. It is then possible to search for the required composite search key by looking up the appropriate buckets that match the parts of the address in which we are interested.

For example, consider the composite search key <Dno, Age>. If Dno and Age are hashed into a 3-bit and 5-bit address respectively, we get an 8-bit bucket address. Suppose that Dno = 4 has a hash address '100' and Age = 59 has hash address '10101'. Then to search for the combined search value, Dno = 4 and Age = 59, one goes to bucket address 100 10101; just to search for all employees with Age = 59, all buckets (eight of them) will be searched whose addresses are '000 10101', '001 10101', … and so on. An advantage of partitioned hashing is that it can be easily extended to any number of attributes. The bucket addresses can be designed so that high-order bits in the addresses correspond to more frequently accessed attributes. Additionally, no separate access structure needs to be maintained for the individual attributes. The main drawback of partitioned hashing is that it cannot handle range queries on any of the component attributes. Additionally, most hash functions do not maintain records in order by the key being hashed. Hence, accessing records in lexicographic order by a combination of attributes such as <Dno, Age> used as a key would not be straightforward or efficient.

17.4.3 Grid Files

Another alternative is to organize the EMPLOYEE file as a grid file. If we want to access a file on two keys, say Dno and Age as in our example, we can construct a grid array with one linear scale (or dimension) for each of the search attributes. Figure 17.14 shows a grid array for the EMPLOYEE file with one linear scale for Dno and another for the Age attribute. The scales are made in a way as to achieve a uniform distribution of that attribute. Thus, in our example, we show that the linear scale for Dno has Dno = 1, 2 combined as one value 0 on the scale, whereas Dno = 5 corresponds to the value 2 on that scale. Similarly, Age is divided into its scale of 0 to 5 by grouping ages so as to distribute the employees uniformly by age. The grid array shown for this file has a total of 36 cells. Each cell points to some bucket address where the records corresponding to that cell are stored. Figure 17.14 also shows the assignment of cells to buckets (only partially).

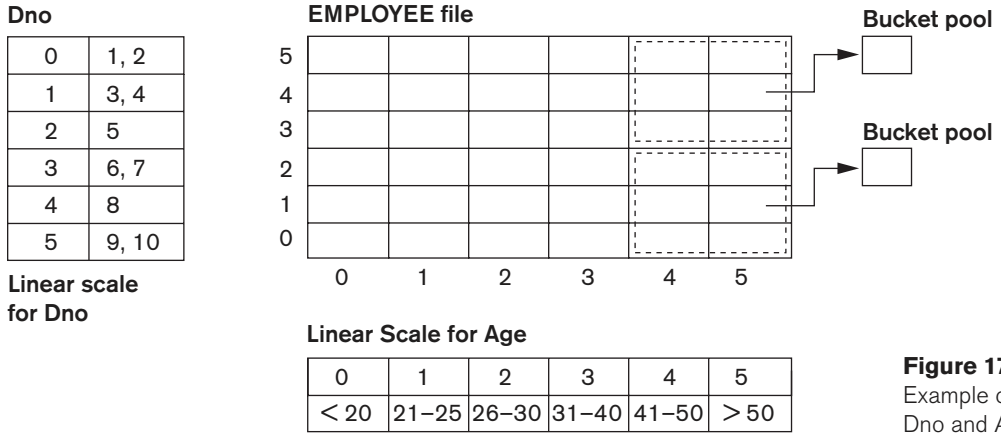


Figure 17.14
Example of a grid array on Dno and Age attributes.

Thus our request for Dno = 4 and Age = 59 maps into the cell (1, 5) corresponding to the grid array. The records for this combination will be found in the corresponding bucket. This method is particularly useful for range queries that would map into a set of cells corresponding to a group of values along the linear scales. If a range query corresponds to a match on the some of the grid cells, it can be processed by accessing exactly the buckets for those grid cells. For example, a query for Dno ≤ 5 and Age > 40 refers to the data in the top bucket shown in Figure 17.14.

The grid file concept can be applied to any number of search keys. For example, for *n* search keys, the grid array would have *n* dimensions. The grid array thus allows a partitioning of the file along the dimensions of the search key attributes and provides an access by combinations of values along those dimensions. Grid files perform well in terms of reduction in time for multiple key access. However, they represent a space overhead in terms of the grid array structure. Moreover, with dynamic files, a frequent reorganization of the file adds to the maintenance cost.[14]

17.5 Other Types of Indexes

17.5.1 Hash Indexes

It is also possible to create access structures similar to indexes that are based on *hashing*. The **hash index** is a secondary structure to access the file by using hashing on a search key other than the one used for the primary data file organization. The index entries are of the type <*K*, *Pr*> or <*K*, *P*>, where *Pr* is a pointer to the record containing the key, or *P* is a pointer to the block containing the record for that key. The index file with these index entries can be organized as a dynamically expandable hash file, using one of the techniques described in Section 16.8.3; searching for an entry uses the hash search algorithm on *K*. Once an entry is found, the pointer *Pr*

[14]Insertion/deletion algorithms for grid files may be found in Nievergelt et al. (1984).

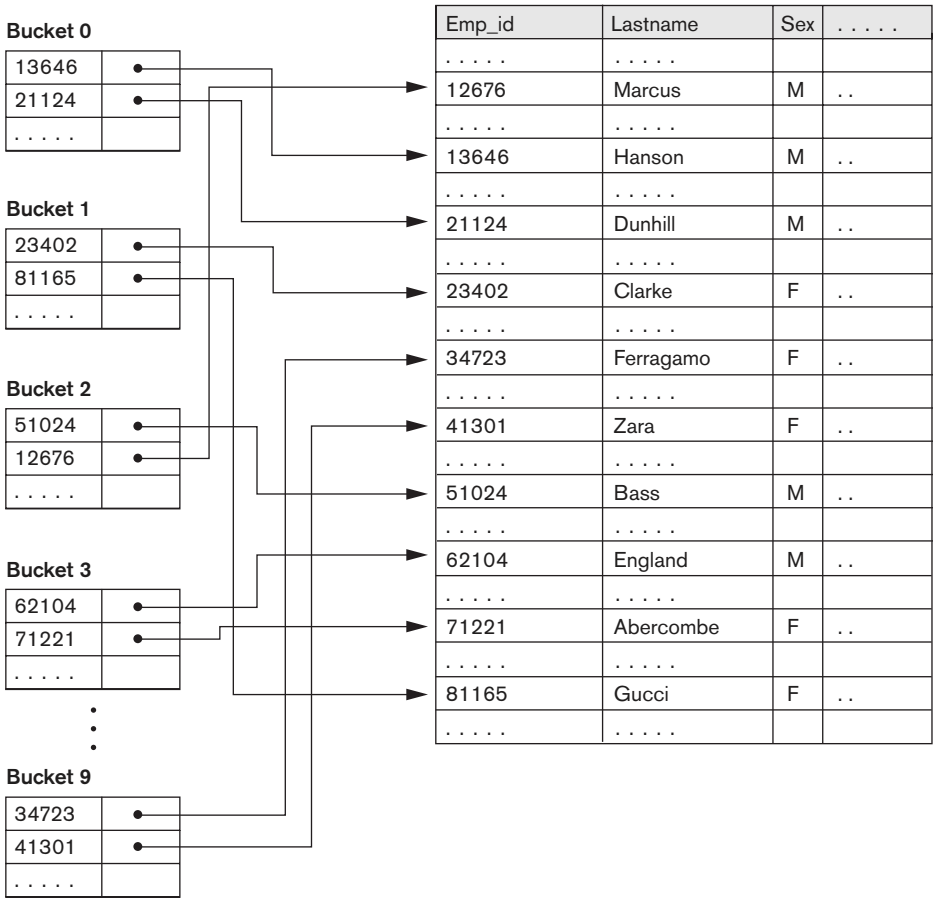


Figure 17.15
Hash-based indexing.

(or *P*) is used to locate the corresponding record in the data file. Figure 17.15 illustrates a hash index on the Emp_id field for a file that has been stored as a sequential file ordered by Name. The Emp_id is hashed to a bucket number by using a hashing function: the sum of the digits of Emp_id modulo 10. For example, to find Emp_id 51024, the hash function results in bucket number 2; that bucket is accessed first. It contains the index entry < 51024, *Pr* >; the pointer *Pr* leads us to the actual record in the file. In a practical application, there may be thousands of buckets; the bucket number, which may be several bits long, would be subjected to the directory schemes discussed in the context of dynamic hashing in Section 16.8.3. Other search structures can also be used as indexes.

17.5.2 Bitmap Indexes

The **bitmap index** is another popular data structure that facilitates querying on multiple keys. Bitmap indexing is used for relations that contain a large number of rows. It creates an index for one or more columns, and each value or value range in

EMPLOYEE

Row_id	Emp_id	Lname	Sex	Zipcode	Salary_grade
0	51024	Bass	M	94040	..
1	23402	Clarke	F	30022	..
2	62104	England	M	19046	..
3	34723	Ferragamo	F	30022	..
4	81165	Gucci	F	19046	..
5	13646	Hanson	M	19046	..
6	12676	Marcus	M	30022	..
7	41301	Zara	F	94040	..

Bitmap index for Sex

M	F
10100110	01011001

Bitmap index for Zipcode

Zipcode 19046	Zipcode 30022	Zipcode 94040
00101100	01010010	10000001

Figure 17.16
Bitmap indexes for Sex and Zipcode.

those columns is indexed. Typically, a bitmap index is created for those columns that contain a fairly small number of unique values. To build a bitmap index on a set of records in a relation, the records must be numbered from 0 to n with an id (a record id or a row id) that can be mapped to a physical address made of a block number and a record offset within the block.

A bitmap index is built on **one particular value** of a particular field (the column in a relation) and is just an array of bits. Thus, for a given field, there is one separate bitmap index (or a vector) maintained corresponding to each unique value in the database. Consider a bitmap index for the column C and a value V for that column. For a relation with n rows, it contains n bits. The ith bit is set to 1 if the row i has the value V for column C; otherwise it is set to a 0. If C contains the valueset $<v_1, v_2, \ldots, v_m>$ with m distinct values, then m bitmap indexes would be created for that column. Figure 17.16 shows the relation EMPLOYEE with columns Emp_id, Lname, Sex, Zipcode, and Salary_grade (with just eight rows for illustration) and a bitmap index for the Sex and Zipcode columns. As an example, if the bitmap for Sex = F, the bits for Row_ids 1, 3, 4, and 7 are set to 1, and the rest of the bits are set to 0, the bitmap indexes could have the following query applications:

- For the query $C_1 = V_1$, the corresponding bitmap for value V_1 returns the Row_ids containing the rows that qualify.
- For the query $C_1 = V_1$ and $C_2 = V_2$ (a multikey search request), the two corresponding bitmaps are retrieved and intersected (logically AND-ed) to yield the set of Row_ids that qualify. In general, k bitvectors can be intersected to deal with k equality conditions. Complex AND-OR conditions can also be supported using bitmap indexing.
  - For the query $C_1 = V_1$ or $C_2 = V_2$ or $C_3 = V_3$ (a multikey search request), the three corresponding bitmaps for three different attributes are retrieved and unioned (logically OR-ed) to yield the set of Row ids that qualify.

- To retrieve a count of rows that qualify for the condition $C_1 = V_1$, the "1" entries in the corresponding bitvector are counted.
- Queries with negation, such as $C_1 \neg = V_1$, can be handled by applying the Boolean *complement* operation on the corresponding bitmap.

Consider the example relation EMPLOYEE in Figure 17.16 with bitmap indexes on Sex and Zipcode. To find employees with Sex = F and Zipcode = 30022, we intersect the bitmaps "01011001" and "01010010" yielding Row_ids 1 and 3. Employees who do not live in Zipcode = 94040 are obtained by complementing the bitvector "10000001" and yields Row_ids 1 through 6. In general, if we assume uniform distribution of values for a given column, and if one column has 5 distinct values and another has 10 distinct values, the join condition on these two can be considered to have a selectivity of 1/50 (= 1/5 * 1/10). Hence, only about 2% of the records would actually have to be retrieved. If a column has only a few values, like the Sex column in Figure 17.16, retrieval of the Sex = M condition on average would retrieve 50% of the rows; in such cases, it is better to do a complete scan rather than use bitmap indexing.

In general, bitmap indexes are efficient in terms of the storage space that they need. If we consider a file of 1 million rows (records) with record size of 100 bytes per row, each bitmap index would take up only one bit per row and hence would use 1 million bits or 125 Kbytes. Suppose this relation is for 1 million residents of a state, and they are spread over 200 ZIP Codes; the 200 bitmaps over Zipcodes contribute 200 bits (or 25 bytes) worth of space per row; hence, the 200 bitmaps occupy only 25% as much space as the data file. They allow an exact retrieval of all residents who live in a given ZIP Code by yielding their Row_ids.

When records are deleted, renumbering rows and shifting bits in bitmaps becomes expensive. Another bitmap, called the **existence bitmap**, can be used to avoid this expense. This bitmap has a 0 bit for the rows that have been deleted but are still physically present and a 1 bit for rows that actually exist. Whenever a row is inserted in the relation, an entry must be made in all the bitmaps of all the columns that have a bitmap index; rows typically are appended to the relation or may replace deleted rows to minimize the impact on the reorganization of the bitmaps. This process still constitutes an indexing overhead.

Large bitvectors are handled by treating them as a series of 32-bit or 64-bit vectors, and corresponding AND, OR, and NOT operators are used from the instruction set to deal with 32- or 64-bit input vectors in a single instruction. This makes bitvector operations computationally very efficient.

Bitmaps for B$^+$-Tree Leaf Nodes. Bitmaps can be used on the leaf nodes of B$^+$-tree indexes as well as to point to the set of records that contain each specific value of the indexed field in the leaf node. When the B$^+$-tree is built on a nonkey search field, the leaf record must contain a list of record pointers alongside each value of the indexed attribute. For values that occur very frequently, that is, in a large percentage of the relation, a bitmap index may be stored instead of the pointers. As an

example, for a relation with n rows, suppose a value occurs in 10% of the file records. A bitvector would have n bits, having the "1" bit for those Row_ids that contain that search value, which is $n/8$ or $0.125n$ bytes in size. If the record pointer takes up 4 bytes (32 bits), then the $n/10$ record pointers would take up $4 * n/10$ or $0.4n$ bytes. Since $0.4n$ is more than 3 times larger than $0.125n$, it is better to store the bitmap index rather than the record pointers. Hence for search values that occur more frequently than a certain ratio (in this case that would be 1/32), it is beneficial to use bitmaps as a compressed storage mechanism for representing the record pointers in B^+-trees that index a nonkey field.

17.5.3 Function-Based Indexing

In this section, we discuss a new type of indexing, called **function-based indexing**, that has been introduced in the Oracle relational DBMS as well as in some other commercial products.[15]

The idea behind function-based indexing is to create an index such that the value that results from applying some function on a field or a collection of fields becomes the key to the index. The following examples show how to create and use function-based indexes.

Example 1. The following statement creates a function-based index on the EMPLOYEE table based on an uppercase representation of the Lname column, which can be entered in many ways but is always queried by its uppercase representation.

```
CREATE INDEX upper_ix ON Employee (UPPER(Lname));
```

This statement will create an index based on the function `UPPER(Lname)`, which returns the last name in uppercase letters; for example, `UPPER('Smith')` will return 'SMITH'.

Function-based indexes ensure that Oracle Database system will use the index rather than perform a full table scan, even when a function is used in the search predicate of a query. For example, the following query will use the index:

```
SELECT First_name, Lname
FROM Employee
WHERE UPPER(Lname)= "SMITH".
```

Without the function-based index, an Oracle Database might perform a full table scan, since a B^+-tree index is searched only by using the column value directly; the use of any function on a column prevents such an index from being used.

Example 2. In this example, the EMPLOYEE table is supposed to contain two fields—salary and commission_pct (commission percentage)—and an index is being created on the sum of salary and commission based on the commission_pct.

```
CREATE INDEX income_ix
ON Employee(Salary + (Salary*Commission_pct));
```

[15]Rafi Ahmed contributed most of this section.

The following query uses the `income_ix` index even though the fields salary and commission_pct are occurring in the reverse order in the query when compared to the index definition.

```
SELECT First_name, Lname
FROM Employee
WHERE ((Salary*Commission_pct) + Salary ) > 15000;
```

Example 3. This is a more advanced example of using function-based indexing to define conditional uniqueness. The following statement creates a unique function-based index on the ORDERS table that prevents a customer from taking advantage of a promotion id ("blowout sale") more than once. It creates a composite index on the Customer_id and Promotion_id fields together, and it allows only one entry in the index for a given Customer_id with the Promotion_id of "2" by declaring it as a unique index.

```
CREATE UNIQUE INDEX promo_ix ON Orders
(CASE WHEN Promotion_id = 2 THEN Customer_id ELSE NULL END,
CASE WHEN Promotion_id = 2 THEN Promotion_id ELSE NULL END);
```

Note that by using the `CASE` statement, the objective is to remove from the index any rows where Promotion_id is not equal to 2. Oracle Database does not store in the B^+-tree index any rows where all the keys are NULL. Therefore, in this example, we map both Customer_id and Promotion_id to NULL unless Promotion_id is equal to 2. The result is that the index constraint is violated only if Promotion_id is equal to 2, for two (attempted insertions of) rows with the same Customer_id value.

17.6 Some General Issues Concerning Indexing

17.6.1 Logical versus Physical Indexes

In the earlier discussion, we have assumed that the index entries <*K*, *Pr*> (or <*K*, *P*>) always include a physical pointer *Pr* (or *P*) that specifies the physical record address on disk as a block number and offset. This is sometimes called a **physical index**, and it has the disadvantage that the pointer must be changed if the record is moved to another disk location. For example, suppose that a primary file organization is based on linear hashing or extendible hashing; then, each time a bucket is split, some records are allocated to new buckets and hence have new physical addresses. If there was a secondary index on the file, the pointers to those records would have to be found and updated, which is a difficult task.

To remedy this situation, we can use a structure called a **logical index**, whose index entries are of the form <*K*, K_p>. Each entry has one value *K* for the secondary indexing field matched with the value K_p of the field used for the primary file organization. By searching the secondary index on the value of *K*, a program can locate the corresponding value of K_p and use this to access the record through the primary file organization, using a primary index if available. Logical indexes thus introduce an

additional level of indirection between the access structure and the data. They are used when physical record addresses are expected to change frequently. The cost of this indirection is the extra search based on the primary file organization.

17.6.2 Index Creation

Many RDBMSs have a similar type of command for creating an index, although it is not part of the SQL standard. The general form of this command is:

```
CREATE [ UNIQUE ] INDEX <index name>
ON <table name> ( <column name> [ <order> ] { , <column name> [ <order> ] } )
[ CLUSTER ] ;
```

The keywords UNIQUE and CLUSTER are optional. The keyword CLUSTER is used when the index to be created should also sort the data file records on the indexing attribute. Thus, specifying CLUSTER on a key (unique) attribute would create some variation of a primary index, whereas specifying CLUSTER on a nonkey (nonunique) attribute would create some variation of a clustering index. The value for <order> can be either ASC (ascending) or DESC (descending), and it specifies whether the data file should be ordered in ascending or descending values of the indexing attribute. The default is ASC. For example, the following would create a clustering (ascending) index on the nonkey attribute Dno of the EMPLOYEE file:

```
CREATE INDEX DnoIndex
ON EMPLOYEE (Dno)
CLUSTER ;
```

Index Creation Process: In many systems, an index is not an integral part of the data file but can be created and discarded dynamically. That is why it is often called an *access structure.* Whenever we expect to access a file frequently based on some search condition involving a particular field, we can request the DBMS to create an index on that field as shown above for the DnoIndex. Usually, a secondary index is created to avoid physical ordering of the records in the data file on disk.

The main advantage of secondary indexes is that—theoretically, at least—they can be created in conjunction with *virtually any primary record organization.* Hence, a secondary index could be used to complement other primary access methods such as ordering or hashing, or it could even be used with mixed files. To create a B^+-tree secondary index on some field of a file, if the file is large and contains millions of records, neither the file nor the index would fit in main memory. Insertion of a large number of entries into the index is done by a process called **bulk loading** the index. We must go through all records in the file to create the entries at the leaf level of the tree. These entries are then sorted and filled according to the specified fill factor; simultaneously, the other index levels are created. It is more expensive and much harder to create primary indexes and clustering indexes dynamically, because the records of the data file must be physically sorted on disk in order of the indexing field. However, some systems allow users to create these indexes dynamically on their files by sorting the file during index creation.

Indexing of Strings: There are a couple of issues that are of particular concern when indexing strings. Strings can be variable length (e.g., VARCHAR data type in SQL; see Chapter 6) and strings may be too long limiting the fan-out. If a B^+-tree index is to be built with a string as a search key, there may be an uneven number of keys per index node and the fan-out may vary. Some nodes may be forced to split when they become full regardless of the number of keys in them. The technique of **prefix compression** alleviates the situation. Instead of storing the entire string in the intermediate nodes, it stores only the prefix of the search key adequate to distinguish the keys that are being separated and directed to the subtree. For example, if Lastname was a search key and we were looking for "Navathe", the nonleaf node may contain "Nac" for Nachamkin and "Nay" for Nayuddin as the two keys on either side of the subtree pointer that we need to follow.

17.6.3 Tuning Indexes

The initial choice of indexes may have to be revised for the following reasons:

- Certain queries may take too long to run for lack of an index.
- Certain indexes may not get utilized at all.
- Certain indexes may undergo too much updating because the index is on an attribute that undergoes frequent changes.

Most DBMSs have a command or trace facility, which can be used by the DBA to ask the system to show how a query was executed—what operations were performed in what order and what secondary access structures (indexes) were used. By analyzing these execution plans (we will discuss this term further in Chapter 18), it is possible to diagnose the causes of the above problems. Some indexes may be dropped and some new indexes may be created based on the tuning analysis.

The goal of tuning is to dynamically evaluate the requirements, which sometimes fluctuate seasonally or during different times of the month or week, and to reorganize the indexes and file organizations to yield the best overall performance. Dropping and building new indexes is an overhead that can be justified in terms of performance improvements. Updating of a table is generally suspended while an index is dropped or created; this loss of service must be accounted for.

Besides dropping or creating indexes and changing from a nonclustered to a clustered index and vice versa, **rebuilding the index** may improve performance. Most RDBMSs use B^+-trees for an index. If there are many deletions on the index key, index pages may contain wasted space, which can be claimed during a rebuild operation. Similarly, too many insertions may cause overflows in a clustered index that affect performance. Rebuilding a clustered index amounts to reorganizing the entire table ordered on that key.

The available options for indexing and the way they are defined, created, and reorganized vary from system to system. As an illustration, consider the sparse and dense indexes we discussed in Section 17.1. A sparse index such as a primary index will have one index pointer for each page (disk block) in the data file; a

dense index such as a unique secondary index will have an index pointer for each record. Sybase provides clustering indexes as sparse indexes in the form of B^+-trees, whereas INGRES provides sparse clustering indexes as ISAM files and dense clustering indexes as B^+-trees. In some versions of Oracle and DB2, the option of setting up a clustering index is limited to a dense index, and the DBA has to work with this limitation.

17.6.4 Additional Issues Related to Storage of Relations and Indexes

Using an Index for Managing Constraints and Duplicates: It is common to use an index to enforce a *key constraint* on an attribute. While searching the index to insert a new record, it is straightforward to check at the same time whether another record in the file—and hence in the index tree—has the same key attribute value as the new record. If so, the insertion can be rejected.

If an index is created on a nonkey field, *duplicates* occur; handling of these duplicates is an issue the DBMS product vendors have to deal with and affects data storage as well as index creation and management. Data records for the duplicate key may be contained in the same block or may span multiple blocks where many duplicates are possible. Some systems add a row id to the record so that records with duplicate keys have their own unique identifiers. In such cases, the B^+-tree index may regard a <key, Row_id> combination as the de facto key for the index, turning the index into a unique index with no duplicates. The deletion of a key K from such an index would involve deleting all occurrences of that key K—hence the deletion algorithm has to account for this.

In actual DBMS products, deletion from B^+-tree indexes is also handled in various ways to improve performance and response times. Deleted records may be marked as deleted and the corresponding index entries may also not be removed until a garbage collection process reclaims the space in the data file; the index is rebuilt online after garbage collection.

Inverted Files and Other Access Methods: A file that has a secondary index on every one of its fields is often called a **fully inverted file**. Because all indexes are secondary, new records are inserted at the end of the file; therefore, the data file itself is an unordered (heap) file. The indexes are usually implemented as B^+-trees, so they are updated dynamically to reflect insertion or deletion of records. Some commercial DBMSs, such as Software AG's Adabas, use this method extensively.

We referred to the popular IBM file organization called ISAM in Section 17.2. Another IBM method, the **virtual storage access method** (**VSAM**), is somewhat similar to the B^+–tree access structure and is still being used in many commercial systems.

Using Indexing Hints in Queries: DBMSs such as Oracle have a provision for allowing hints in queries that are suggested alternatives or indicators to the query

processor and optimizer for expediting query execution. One form of hints is called indexing hints; these hints suggest the use of an index to improve the execution of a query. The hints appear as a special comment (which is preceded by +) and they override all optimizer decisions, but they may be ignored by the optimizer if they are invalid, irrelevant, or improperly formulated. We do not get into a detailed discussion of indexing hints, but illustrate with an example query.

For example, to retrieve the SSN, Salary, and department number for employees working in department numbers with Dno less than 10:

```
SELECT /*+ INDEX (EMPLOYEE emp_dno_index ) */ Emp_ssn, Salary, Dno
FROM EMPLOYEE
WHERE Dno < 10;
```

The above query includes a hint to use a valid index called emp_dno_index (which is an index on the EMPLOYEE relation on Dno).

Column-Based Storage of Relations: There has been a recent trend to consider a column-based storage of relations as an alternative to the traditional way of storing relations row by row. Commercial relational DBMSs have offered B^+-tree indexing on primary as well as secondary keys as an efficient mechanism to support access to data by various search criteria and the ability to write a row or a set of rows to disk at a time to produce write-optimized systems. For data warehouses (to be discussed in Chapter 29), which are read-only databases, the column-based storage offers particular advantages for read-only queries. Typically, the column-store RDBMSs consider storing each column of data individually and afford performance advantages in the following areas:

- Vertically partitioning the table column by column, so that a two-column table can be constructed for every attribute and thus only the needed columns can be accessed
- Using column-wise indexes (similar to the bitmap indexes discussed in Section 17.5.2) and join indexes on multiple tables to answer queries without having to access the data tables
- Using materialized views (see Chapter 7) to support queries on multiple columns

Column-wise storage of data affords additional freedom in the creation of indexes, such as the bitmap indexes discussed earlier. The same column may be present in multiple projections of a table and indexes may be created on each projection. To store the values in the same column, strategies for data compression, null-value suppression, dictionary encoding techniques (where distinct values in the column are assigned shorter codes), and run-length encoding techniques have been devised. MonetDB/X100, C-Store, and Vertica are examples of such systems Some popular systems (like Cassandra, Hbase, and Hypertable) have used column-based storage effectively with the concept of **wide column-stores**. The storage of data in such systems will be explained in the context of NOSQL systems that we will discuss in Chapter 24.

17.7 Physical Database Design in Relational Databases

In this section, we discuss the physical design factors that affect the performance of applications and transactions, and then we comment on the specific guidelines for RDBMSs in the context of what we discussed in Chapter 16 and this chapter so far.

17.7.1 Factors That Influence Physical Database Design

Physical design is an activity where the goal is not only to create the appropriate structuring of data in storage, but also to do so in a way that guarantees good performance. For a given conceptual schema, there are many physical design alternatives in a given DBMS. It is not possible to make meaningful physical design decisions and performance analyses until the database designer knows the mix of queries, transactions, and applications that are expected to run on the database. This is called the **job mix** for the particular set of database system applications. The database administrators/designers must analyze these applications, their expected frequencies of invocation, any timing constraints on their execution speed, the expected frequency of update operations, and any unique constraints on attributes. We discuss each of these factors next.

A. Analyzing the Database Queries and Transactions. Before undertaking the physical database design, we must have a good idea of the intended use of the database by defining in a high-level form the queries and transactions that are expected to run on the database. For each **retrieval query**, the following information about the query would be needed:

1. The files (relations) that will be accessed by the query
2. The attributes on which any selection conditions for the query are specified
3. Whether the selection condition is an equality, inequality, or a range condition
4. The attributes on which any join conditions or conditions to link multiple tables or objects for the query are specified
5. The attributes whose values will be retrieved by the query

The attributes listed in items 2 and 4 above are candidates for the definition of access structures, such as indexes, hash keys, or sorting of the file.

For each **update operation** or **update transaction**, the following information would be needed:

1. The files that will be updated
2. The type of operation on each file (insert, update, or delete)
3. The attributes on which selection conditions for a delete or update are specified
4. The attributes whose values will be changed by an update operation

Again, the attributes listed in item 3 are candidates for access structures on the files, because they would be used to locate the records that will be updated or deleted. On

the other hand, the attributes listed in item 4 are candidates for *avoiding an access structure*, since modifying them will require updating the access structures.

B. Analyzing the Expected Frequency of Invocation of Queries and Transactions. Besides identifying the characteristics of expected retrieval queries and update transactions, we must consider their expected rates of invocation. This frequency information, along with the attribute information collected on each query and transaction, is used to compile a cumulative list of the expected frequency of use for all queries and transactions. This is expressed as the expected frequency of using each attribute in each file as a selection attribute or a join attribute, over all the queries and transactions. Generally, for large volumes of processing, the informal *80–20 rule* can be used: approximately 80% of the processing is accounted for by only 20% of the queries and transactions. Therefore, in practical situations, it is rarely necessary to collect exhaustive statistics and invocation rates on all the queries and transactions; it is sufficient to determine the 20% or so most important ones.

C. Analyzing the Time Constraints of Queries and Transactions. Some queries and transactions may have stringent performance constraints. For example, a transaction may have the constraint that it should terminate within 5 seconds on 95% of the occasions when it is invoked, and that it should never take more than 20 seconds. Such timing constraints place further priorities on the attributes that are candidates for access paths. The selection attributes used by queries and transactions with time constraints become higher-priority candidates for primary access structures for the files, because the primary access structures are generally the most efficient for locating records in a file.

D. Analyzing the Expected Frequencies of Update Operations. A minimum number of access paths should be specified for a file that is frequently updated, because updating the access paths themselves slows down the update operations. For example, if a file that has frequent record insertions has 10 indexes on 10 different attributes, each of these indexes must be updated whenever a new record is inserted. The overhead for updating 10 indexes can slow down the insert operations.

E. Analyzing the Uniqueness Constraints on Attributes. Access paths should be specified on all *candidate key* attributes—or sets of attributes—that are either the primary key of a file or unique attributes. The existence of an index (or other access path) makes it sufficient to search only the index when checking this uniqueness constraint, since all values of the attribute will exist in the leaf nodes of the index. For example, when inserting a new record, if a key attribute value of the new record *already exists in the index*, the insertion of the new record should be rejected, since it would violate the uniqueness constraint on the attribute.

Once the preceding information is compiled, it is possible to address the physical database design decisions, which consist mainly of deciding on the storage structures and access paths for the database files.

17.7.2 Physical Database Design Decisions

Most relational systems represent each base relation as a physical database file. The access path options include specifying the type of primary file organization for each relation and the attributes that are candidates for defining individual or composite indexes. At most, one of the indexes on each file may be a primary or a clustering index. Any number of additional secondary indexes can be created.

Design Decisions about Indexing. The attributes whose values are required in equality or range conditions (selection operation) are those that are keys or that participate in join conditions (join operation) requiring access paths, such as indexes.

The performance of queries largely depends upon what indexes or hashing schemes exist to expedite the processing of selections and joins. On the other hand, during insert, delete, or update operations, the existence of indexes adds to the overhead. This overhead must be justified in terms of the gain in efficiency by expediting queries and transactions.

The physical design decisions for indexing fall into the following categories:

1. **Whether to index an attribute.** The general rules for creating an index on an attribute are that the attribute must either be a key (unique), or there must be some query that uses that attribute either in a selection condition (equality or range of values) or in a join condition. One reason for creating multiple indexes is that some operations can be processed by just scanning the indexes, without having to access the actual data file.
2. **What attribute or attributes to index on.** An index can be constructed on a single attribute, or on more than one attribute if it is a composite index. If multiple attributes from one relation are involved together in several queries, (for example, (Garment_style_#, Color) in a garment inventory database), a multiattribute (composite) index is warranted. The ordering of attributes within a multiattribute index must correspond to the queries. For instance, the above index assumes that queries would be based on an ordering of colors within a Garment_style_# rather than vice versa.
3. **Whether to set up a clustered index.** At most, one index per table can be a primary or clustering index, because this implies that the file be physically ordered on that attribute. In most RDBMSs, this is specified by the keyword CLUSTER. (If the attribute is a *key*, *a primary index* is created, whereas a *clustering index* is created if the attribute is *not a key*.) If a table requires several indexes, the decision about which one should be the primary or clustering index depends upon whether keeping the table ordered on that attribute is needed. Range queries benefit a great deal from clustering. If several attributes require range queries, relative benefits must be evaluated before deciding which attribute to cluster on. If a query is to be answered by doing an index search only (without retrieving data records), the corresponding index should *not* be clustered, since the main benefit of clustering is achieved

when retrieving the records themselves. A clustering index may be set up as a multiattribute index if range retrieval by that composite key is useful in report creation (for example, an index on Zip_code, Store_id, and Product_id may be a clustering index for sales data).

4. **Whether to use a hash index over a tree index.** In general, RDBMSs use B^+-trees for indexing. However, ISAM and hash indexes are also provided in some systems. B^+-trees support both equality and range queries on the attribute used as the search key. Hash indexes work well with equality conditions, particularly during joins to find a matching record(s), but they do not support range queries.
5. **Whether to use dynamic hashing for the file.** For files that are very volatile—that is, those that grow and shrink continuously—one of the dynamic hashing schemes discussed in Section 16.9 would be suitable. Currently, such schemes are not offered by many commercial RDBMSs.

17.8 Summary

In this chapter, we presented file organizations that involve additional access structures, called indexes, to improve the efficiency of retrieval of records from a data file. These access structures may be used *in conjunction with* the primary file organizations discussed in Chapter 16, which are used to organize the file records themselves on disk.

Three types of ordered single-level indexes were introduced: primary, clustering, and secondary. Each index is specified on a field of the file. Primary and clustering indexes are constructed on the physical ordering field of a file, whereas secondary indexes are specified on nonordering fields as additional access structures to improve performance of queries and transactions. The field for a primary index must also be a key of the file, whereas it is a nonkey field for a clustering index. A single-level index is an ordered file and is searched using a binary search. We showed how multilevel indexes can be constructed to improve the efficiency of searching an index. An example is IBM's popular indexed sequential access method (ISAM), which is a multilevel index based on the cylinder/track configuration on disk.

Next we showed how multilevel indexes can be implemented as B-trees and B^+-trees, which are dynamic structures that allow an index to expand and shrink dynamically. The nodes (blocks) of these index structures are kept between half full and completely full by the insertion and deletion algorithms. Nodes eventually stabilize at an average occupancy of 69% full, allowing space for insertions without requiring reorganization of the index for the majority of insertions. B^+-trees can generally hold more entries in their internal nodes than can B-trees, so they may have fewer levels or hold more entries than does a corresponding B-tree.

We gave an overview of multiple key access methods, and we showed how an index can be constructed based on hash data structures. We introduced the concept of

partitioned hashing, which is an extension of external hashing to deal with multiple keys. We also introduced **grid files**, which organize data into buckets along multiple dimensions, We discussed the **hash index** in some detail—it is a secondary structure to access the file by using hashing on a search key other than that used for the primary organization. **Bitmap indexing** is another important type of indexing used for querying by multiple keys and is particularly applicable on fields with a small number of unique values. Bitmaps can also be used at the leaf nodes of B^+ tree indexes as well. We also discussed function-based indexing, which is being provided by relational vendors to allow special indexes on a function of one or more attributes.

We introduced the concept of a logical index and compared it with the physical indexes we described before. They allow an additional level of indirection in indexing in order to permit greater freedom for movement of actual record locations on disk. We discussed index creation in SQL, the process of bulk loading of index files and indexing of strings. We discussed circumstances that point to tuning of indexes. Then we reviewed some general topics related to indexing, including managing constraints, using inverted indexes, and using indexing hints in queries; we commented on column-based storage of relations, which is becoming a viable alternative for storing and accessing large databases. Finally, we discussed physical database design of relational databases, which involves decisions related to storage and accessing of data that we have been discussing in the current and the previous chapter. This discussion was divided into factors that influence the design and the types of decisions regarding whether to index an attribute, what attributes to include in an index, clustered versus nonclustered indexes, hashed indexes, and dynamic hashing.

Review Questions

17.1. Define the following terms: *indexing field*, *primary key field*, *clustering field*, *secondary key field*, *block anchor*, *dense index*, and *nondense (sparse) index*.

17.2. What are the differences among primary, secondary, and clustering indexes? How do these differences affect the ways in which these indexes are implemented? Which of the indexes are dense, and which are not?

17.3. Why can we have at most one primary or clustering index on a file, but several secondary indexes?

17.4. How does multilevel indexing improve the efficiency of searching an index file?

17.5. What is the order p of a B-tree? Describe the structure of B-tree nodes.

17.6. What is the order p of a B^+-tree? Describe the structure of both internal and leaf nodes of a B^+-tree.

17.7. How does a B-tree differ from a B^+-tree? Why is a B^+-tree usually preferred as an access structure to a data file?

17.8. Explain what alternative choices exist for accessing a file based on multiple search keys.

17.9. What is partitioned hashing? How does it work? What are its limitations?

17.10. What is a grid file? What are its advantages and disadvantages?

17.11. Show an example of constructing a grid array on two attributes on some file.

17.12. What is a fully inverted file? What is an indexed sequential file?

17.13. How can hashing be used to construct an index?

17.14. What is bitmap indexing? Create a relation with two columns and sixteen tuples and show an example of a bitmap index on one or both.

17.15. What is the concept of function-based indexing? What additional purpose does it serve?

17.16. What is the difference between a logical index and a physical index?

17.17. What is column-based storage of a relational database?

Exercises

17.18. Consider a disk with block size $B = 512$ bytes. A block pointer is $P = 6$ bytes long, and a record pointer is $P_R = 7$ bytes long. A file has $r = 30{,}000$ EMPLOYEE records of *fixed length*. Each record has the following fields: Name (30 bytes),Ssn (9 bytes), Department_code (9 bytes), Address (40 bytes), Phone (10 bytes), Birth_date (8 bytes), Sex (1 byte), Job_code (4 bytes), and Salary (4 bytes, real number). An additional byte is used as a deletion marker.

a. Calculate the record size R in bytes.

b. Calculate the blocking factor *bfr* and the number of file blocks b, assuming an unspanned organization.

c. Suppose that the file is *ordered* by the key field Ssn and we want to construct a *primary index* on Ssn. Calculate (i) the index blocking factor bfr_i (which is also the index fan-out *fo*); (ii) the number of first-level index entries and the number of first-level index blocks; (iii) the number of levels needed if we make it into a multilevel index; (iv) the total number of blocks required by the multilevel index; and (v) the number of block accesses needed to search for and retrieve a record from the file—given its Ssn value—using the primary index.

d. Suppose that the file is *not ordered* by the key field Ssn and we want to construct a *secondary index* on Ssn. Repeat the previous exercise (part c) for the secondary index and compare with the primary index.

e. Suppose that the file is *not ordered* by the nonkey field Department_code and we want to construct a *secondary index* on Department_code, using

option 3 of Section 17.1.3, with an extra level of indirection that stores record pointers. Assume there are 1,000 distinct values of Department_code and that the EMPLOYEE records are evenly distributed among these values. Calculate (i) the index blocking factor bfr_i (which is also the index fan-out *fo*); (ii) the number of blocks needed by the level of indirection that stores record pointers; (iii) the number of first-level index entries and the number of first-level index blocks; (iv) the number of levels needed if we make it into a multilevel index; (v) the total number of blocks required by the multilevel index and the blocks used in the extra level of indirection; and (vi) the approximate number of block accesses needed to search for and retrieve all records in the file that have a specific Department_code value, using the index.

f. Suppose that the file is *ordered* by the nonkey field Department_code and we want to construct a *clustering index* on Department_code that uses block anchors (every new value of Department_code starts at the beginning of a new block). Assume there are 1,000 distinct values of Department_code and that the EMPLOYEE records are evenly distributed among these values. Calculate (i) the index blocking factor bfr_i (which is also the index fan-out *fo*); (ii) the number of first-level index entries and the number of first-level index blocks; (iii) the number of levels needed if we make it into a multilevel index; (iv) the total number of blocks required by the multilevel index; and (v) the number of block accesses needed to search for and retrieve all records in the file that have a specific Department_code value, using the clustering index (assume that multiple blocks in a cluster are contiguous).

g. Suppose that the file is *not* ordered by the key field Ssn and we want to construct a B^+-tree access structure (index) on Ssn. Calculate (i) the orders p and p_{leaf} of the B^+-tree; (ii) the number of leaf-level blocks needed if blocks are approximately 69% full (rounded up for convenience); (iii) the number of levels needed if internal nodes are also 69% full (rounded up for convenience); (iv) the total number of blocks required by the B^+-tree; and (v) the number of block accesses needed to search for and retrieve a record from the file—given its Ssn value—using the B^+-tree.

h. Repeat part g, but for a B-tree rather than for a B^+-tree. Compare your results for the B-tree and for the B^+-tree.

17.19. A PARTS file with Part# as the key field includes records with the following Part# values: 23, 65, 37, 60, 46, 92, 48, 71, 56, 59, 18, 21, 10, 74, 78, 15, 16, 20, 24, 28, 39, 43, 47, 50, 69, 75, 8, 49, 33, 38. Suppose that the search field values are inserted in the given order in a B^+-tree of order $p = 4$ and $p_{\text{leaf}} = 3$; show how the tree will expand and what the final tree will look like.

17.20. Repeat Exercise 17.19, but use a B-tree of order $p = 4$ instead of a B^+-tree.

17.21. Suppose that the following search field values are deleted, in the given order, from the B^+-tree of Exercise 17.19; show how the tree will shrink and show the final tree. The deleted values are 65, 75, 43, 18, 20, 92, 59, 37.

17.22. Repeat Exercise 17.21, but for the B-tree of Exercise 17.20.

17.23. Algorithm 17.1 outlines the procedure for searching a nondense multilevel primary index to retrieve a file record. Adapt the algorithm for each of the following cases:

a. A multilevel secondary index on a nonkey nonordering field of a file. Assume that option 3 of Section 17.1.3 is used, where an extra level of indirection stores pointers to the individual records with the corresponding index field value.

b. A multilevel secondary index on a nonordering key field of a file.

c. A multilevel clustering index on a nonkey ordering field of a file.

17.24. Suppose that several secondary indexes exist on nonkey fields of a file, implemented using option 3 of Section 17.1.3; for example, we could have secondary indexes on the fields Department_code, Job_code, and Salary of the EMPLOYEE file of Exercise 17.18. Describe an efficient way to search for and retrieve records satisfying a complex selection condition on these fields, such as (Department_code = 5 AND Job_code = 12 AND Salary = 50,000), using the record pointers in the indirection level.

17.25. Adapt Algorithms 17.2 and 17.3, which outline search and insertion procedures for a B^+-tree, to a B-tree.

17.26. It is possible to modify the B^+-tree insertion algorithm to delay the case where a new level is produced by checking for a possible *redistribution* of values among the leaf nodes. Figure 17.17 illustrates how this could be done for our example in Figure 17.12; rather than splitting the leftmost leaf node when 12 is inserted, we do a *left redistribution* by moving 7 to the leaf node to its left (if there is space in this node). Figure 17.17 shows how the tree would look when redistribution is considered. It is also possible to consider *right redistribution*. Try to modify the B^+-tree insertion algorithm to take redistribution into account.

17.27. Outline an algorithm for deletion from a B^+-tree.

17.28. Repeat Exercise 17.27 for a B-tree.

Selected Bibliography

Indexing: Bayer and McCreight (1972) introduced B-trees and associated algorithms. Comer (1979) provides an excellent survey of B-trees and their history, and variations of B-trees. Knuth (1998) provides detailed analysis of many search techniques, including B-trees and some of their variations. Nievergelt (1974) discusses the use of binary search trees for file organization. Textbooks on file structures, including Claybrook (1992), Smith and Barnes (1987), and Salzberg (1988); the algorithms and data structures textbook by Wirth (1985); as well as the database textbook by Ramakrihnan and Gehrke (2003) discuss indexing in detail and may be

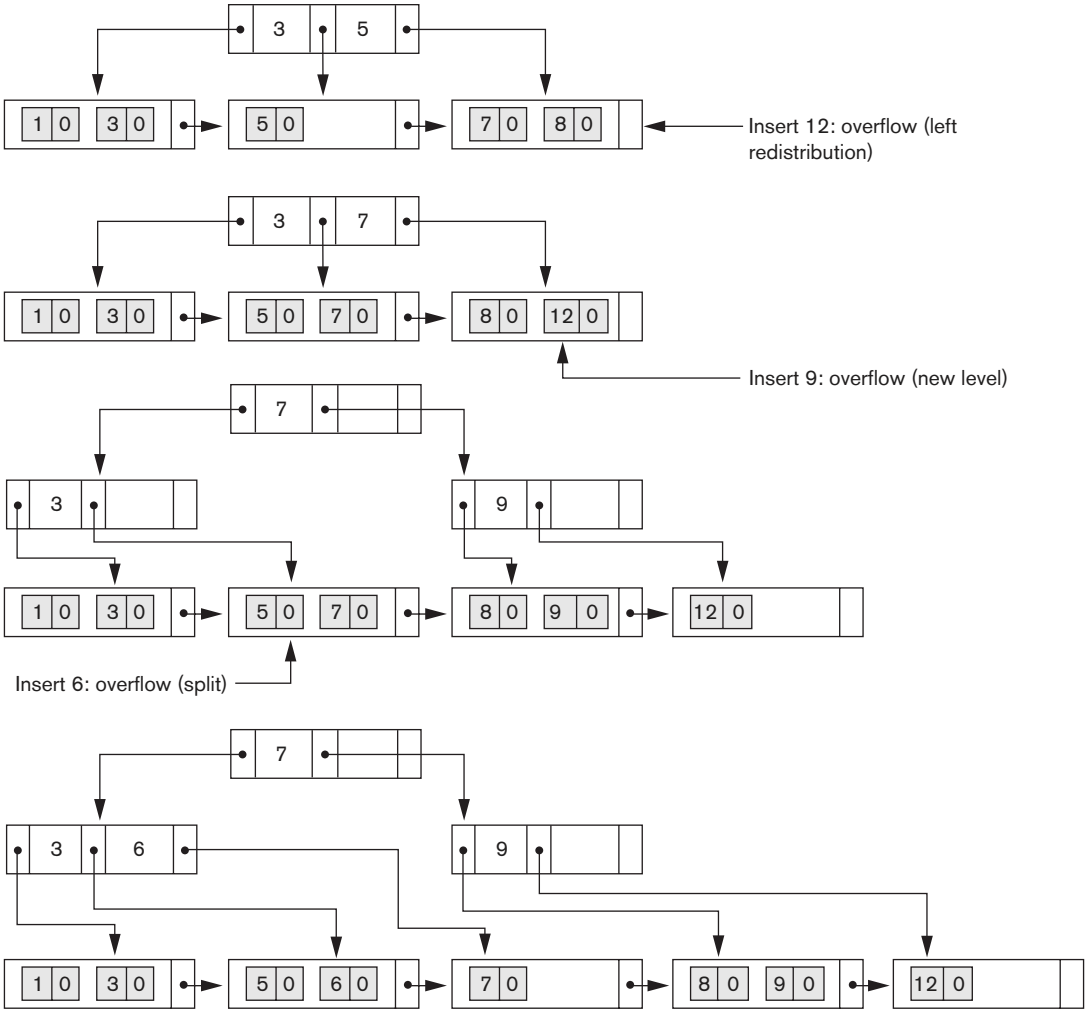


Figure 17.17
B^+-tree insertion with left redistribution.

consulted for search, insertion, and deletion algorithms for B-trees and B^+-trees. Larson (1981) analyzes index-sequential files, and Held and Stonebraker (1978) compare static multilevel indexes with B-tree dynamic indexes. Lehman and Yao (1981) and Srinivasan and Carey (1991) did further analysis of concurrent access to B-trees. The books by Wiederhold (1987), Smith and Barnes (1987), and Salzberg (1988), among others, discuss many of the search techniques described in this chapter. Grid files are introduced in Nievergelt et al. (1984). Partial-match retrieval, which uses partitioned hashing, is discussed in Burkhard (1976, 1979).

New techniques and applications of indexes and B^+-trees are discussed in Lanka and Mays (1991), Zobel et al. (1992), and Faloutsos and Jagadish (1992). Mohan

and Narang (1992) discuss index creation. The performance of various B–tree and B^+-tree algorithms is assessed in Baeza-Yates and Larson (1989) and Johnson and Shasha (1993). Buffer management for indexes is discussed in Chan et al. (1992). Column-based storage of databases was proposed by Stonebraker et al. (2005) in the C-Store database system; MonetDB/X100 by Boncz et al. (2008) is another implementation of the idea. Abadi et al. (2008) discuss the advantages of column stores over row-stored databases for read-only database applications.

Physical Database Design: Wiederhold (1987) covers issues related to physical design. O'Neil and O'Neil (2001) provides a detailed discussion of physical design and transaction issues in reference to commercial RDBMSs. Navathe and Kerschberg (1986) discuss all phases of database design and point out the role of data dictionaries. Rozen and Shasha (1991) and Carlis and March (1984) present different models for the problem of physical database design. Shasha and Bonnet (2002) offer an elaborate discussion of guidelines for database tuning. Niemiec (2008) is one among several books available for Oracle database administration and tuning; Schneider (2006) is focused on designing and tuning MySQL databases.