Chapter 4

Index Structures

Having seen the options available for representing records, we must now consider how whole relations, or the extents of classes, are represented. It is not sufficient simply to scatter the records that represent tuples of the relation or objects of the extent among various blocks. To see why, ask how we would answer even the simplest query, such as `SELECT * FROM R`. We would have to examine every block in the storage system, and we would have to rely on there being:

1. Enough information in block headers to identify where in the block records begin.

2. Enough information in record headers to tell what relation the record belongs to.

A slightly better organization is to reserve some blocks, perhaps several whole cylinders, for a given relation. All blocks in those cylinders may be assumed to hold records that represent tuples of our relation. Now, at least we can find the tuples of the relation without scanning the entire data store.

However, this organization offers no help should we want to answer the next-simplest query: "find a tuple given the value of its primary key." For example, name is the primary key of the `MovieStar` relation from Fig. 3.1. A query like

```
SELECT *
FROM MovieStar
WHERE name = 'Jim Carrey';
```

requires us to scan all the blocks on which MovieStar tuples could be found. To facilitate queries such as this one, we often create one or more *indexes* on a relation. As suggested in Fig. 4.1, an index is any data structure that takes as input a property of records — typically the value of one or more fields — and finds the records with that property "quickly." In particular, an index lets us find a records without having to look at more than a small fraction of all

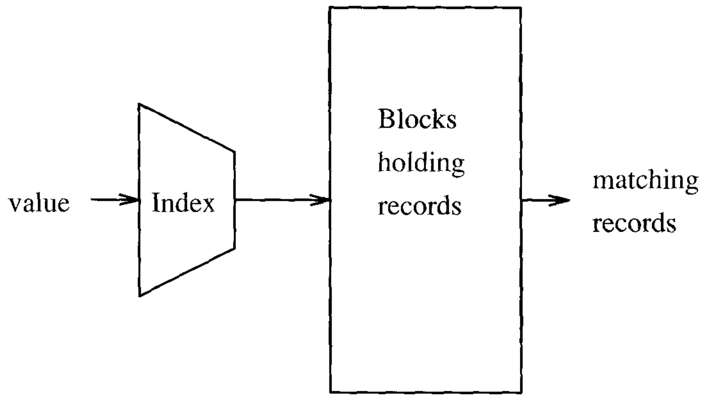

Figure 4.1: An index takes a value for some field(s) and finds records with the matching value

possible records. The field(s) on whose values the index is based is called the *search key,* or just "key" if the index is understood.

There are many different data structures that serve as indexes. In the remainder of this chapter we consider methods for designing and implementing indexes:

1. Simple indexes on sorted files.
2. Secondary indexes on unsorted files.
3. B-trees, a commonly used way to build indexes on any file.
4. Hash tables, another useful and important index structure.

4.1 Indexes on Sequential Files

We begin our study of index structures by considering what is probably the simplest structure: A sorted file, called the *data file,* is given another file, called the *index file,* consisting of key-pointer pairs. A search key K in the index file is associated with a pointer to a data-file record that has search key K. These indexes can be "dense," meaning there is an entry in the index file for every record of the data file, or "sparse," meaning that only some of the data records are represented in the index file, often one index pair per block of the data file.

4.1.1 Sequential Files

One of the simplest index types relies on the file being sorted on the attribute(s) of the index. Such a file is called a *sequential file.* This structure is especially

Keys and More Keys

The term "key" has several meanings, and this book uses "key" in each of these ways when the situation warrants it. You surely are familiar with the use of "key" to mean "primary key of a relation." These keys are declared in SQL and require that the relation not have two tuples that agree on the attribute or attributes of the primary key.

In Section 2.3.4 we learned about "sort keys," the attribute(s) on which a file of records is sorted. Now, we shall speak of "search keys," the attribute(s) for which we are given values and asked to search, through an index," for tuples with matching values. We try to use the appropriate adjective — "primary," "sort," or "search" — when the meaning of "key" is unclear. However, notice in sections such as 4.1.2 and 4.1.3 that there are many times when the three kinds of keys are one and the same.

useful when the search key is the primary key of the relation, although it can be used for other attributes. Figure 4.2 suggests a relation represented as a sequential file.

In this file, the tuples are sorted by their primary key. We imagine that keys are integers; we show only the key field, and we make the atypical assumption that there is room for only two records in one block. For instance, the first block of the file holds the records with keys 10 and 20. In this and many other examples, we use integers that are sequential multiples of 10 as keys, although there is surely no requirement that keys be multiples of 10 or that records with all multiples of 10 appear.

4.1.2 Dense Indexes

Now that we have our records sorted, we can build on them a *dense index*, which is a sequence of blocks holding only the keys of the records and pointers to the records themselves; the pointers are addresses in the sense discussed in Section 3.3. The index is called "dense" because every key from the data file is represented in the index. In comparison, "sparse" indexes, to be discussed in Section 4.1.3, normally keep only one key per data block in the index.

The index blocks of the dense index maintain these keys in the same sorted order as in the file itself. Since keys and pointers presumably take much less space than complete records, we expect to use many fewer blocks for the index than for the file itself. The index is especially advantageous when it, but not the data file, can fit in main memory. Then, by using the index, we can find any record given its search key, with only one disk I/O per lookup.

Example 4.1: Figure 4.3 suggests a dense index on a sorted file that begins as Fig. 4.2. For convenience, we have assumed that the file continues with a

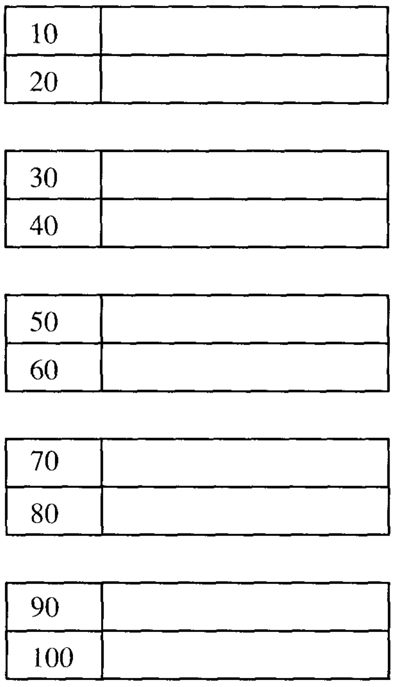

Figure 4.2: A sequential file

key every 10 integers, although in practice we would not expect to find such a regular pattern of keys. We have also assumed that index blocks can hold only four key-pointer pairs. Again, in practice we would find typically that there were many more pairs per block, perhaps hundreds.

The first index block contains pointers to the first four records, the second block has pointers to the next four, and so on. For reasons that we shall discuss in Section 4.1.6, in practice we may not want to fill all the index blocks completely. □

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

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

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

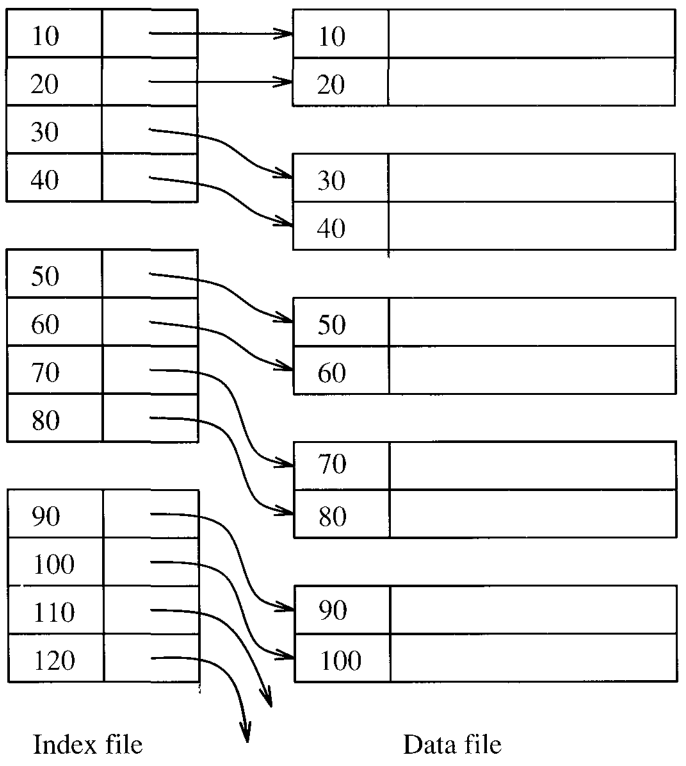

Figure 4.3: A dense index (left) on a sequential data file (light)

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

Example 4.2 : Imagine a relation of 1,000,000 tuples that fit ten to a 4096-byte block. The total space required by the data is over 400 megabytes, probably far too much to keep in main memory. However, suppose that the key field is 30 bytes, and pointers are 8 bytes. Then with a reasonable amount of block-header space we can keep 100 key-pointer pairs in a 4096-byte block.

A dense index therefore requires 10,000 blocks, or 40 megabytes. We might be able to allocate main-memory buffers for these blocks, depending on what else we needed in main memory, and how much main memory there was. Further, $\log_2(10000)$ is about 13, so we only need to access 13 or 14 blocks in a binary search for a key. And since all binary searches would start out accessing only a small subset of the blocks (the block in the middle, those at the 1/4 and 3/4 points, those at 1/8, 3/8, 5/8, and 7/8, and so on), even if we could not afford to keep the whole index in memory, we might be able to keep the most important blocks in main memory, thus retrieving the record for any key with

Locating Index Blocks

Wo have assumed that some mechanism exists for locating the index blocks, from which the individual tuples (if the index is dense) or blocks of the data file (if the index is sparse) can be found. Many ways of locating the index can be used. For example, if the index is small, we may store it in reserved locations of memory or disk. If the index is larger, we can build another layer of index on top of it as we discuss in Section 4.1.4 and keep that in fixed locations. The ultimate extension of this idea is the B-tree of Section 4.3, where we need to know the location of only a single root block.

significantly fewer than 14 disk I/O's. □

4.1.3 Sparse Indexes

If a dense index is too large, we can use a similar structure, called a *sparse index,* that uses less space at the expense of somewhat more time to find a record given its key. A sparse index, as seen in Fig. 4.4, holds only one key-pointer per data block. The key is for the first record on the data block.

Example 4.3: As in Example 4.1, we assume that the data file is sorted, and keys are all the integers divisible by 10, up to some large number. We also continue to assume that four key-pointer pairs fit on an index block. Thus, the first index block has entries for the first keys on the first four blocks, which are 10, 30, 50, and 70. Continuing the assumed pattern of keys, the second index block has the first keys of the fifth through eighth blocks, which we assume are 90, 110, 130, and 150. We also show a third index block with first keys from the hypothetical ninth through twelfth data blocks. □

Example 4.4: A sparse index can require many fewer blocks than a dense index. Using the more realistic parameters of Example 4.2, since there are 100,000 data blocks, and 100 key-pointer pairs fit on one index block, we need only 1000 index blocks if a sparse index is used. Now the index uses only four megabytes, an amount that could plausibly be allocated in main memory.

On the other hand, the dense index allows us to answer queries of the form "does there exist a record with key value K?" without having to retrieve the block containing the record. The fact that K exists in the dense index is enough to guarantee the existence of the record with key K. On the other hand, the same query, using a sparse index, requires a disk I/O to retrieve the block on which key K *might* be found. □

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

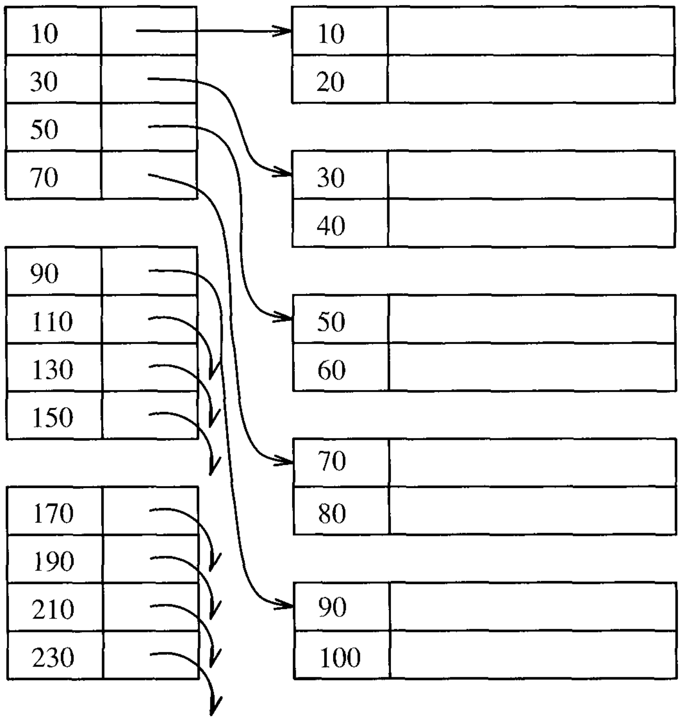

Figure 4.4: A sparse index on a sequential file

key, we may again be able to use binary search to locate this entry. We follow the associated pointer to a data block. Now, we must search this block for the record with key K. Of course the block must have enough format information that the records and their contents can be identified. Any of the techniques from Sections 3.2 and 3.4 can be used, as appropriate.

4.1.4 Multiple Levels of Index

An index can itself cover many blocks, as we saw in Examples 4.2 and 4.4. If these blocks are not in some place where we know we can find them, e.g., designated cylinders of a disk, then we may need another data structure to find them. Even if we can locate the index blocks, and we can use a binary search to find the desired index entry, we still may need to do many disk I/O's to get to the record we want.

By putting an index on the index, we can make the use of the first level of index more efficient. Figure 4.5 extends Fig. 4.4 by adding a second index level (as before, we assume the unusual pattern of keys every 10 integers). The same idea would let us place a third-level index on the second level, and so on. However, this idea has its limits, and we might consider using the B-tree structure described in Section 4.3 in preference to building many levels of index.

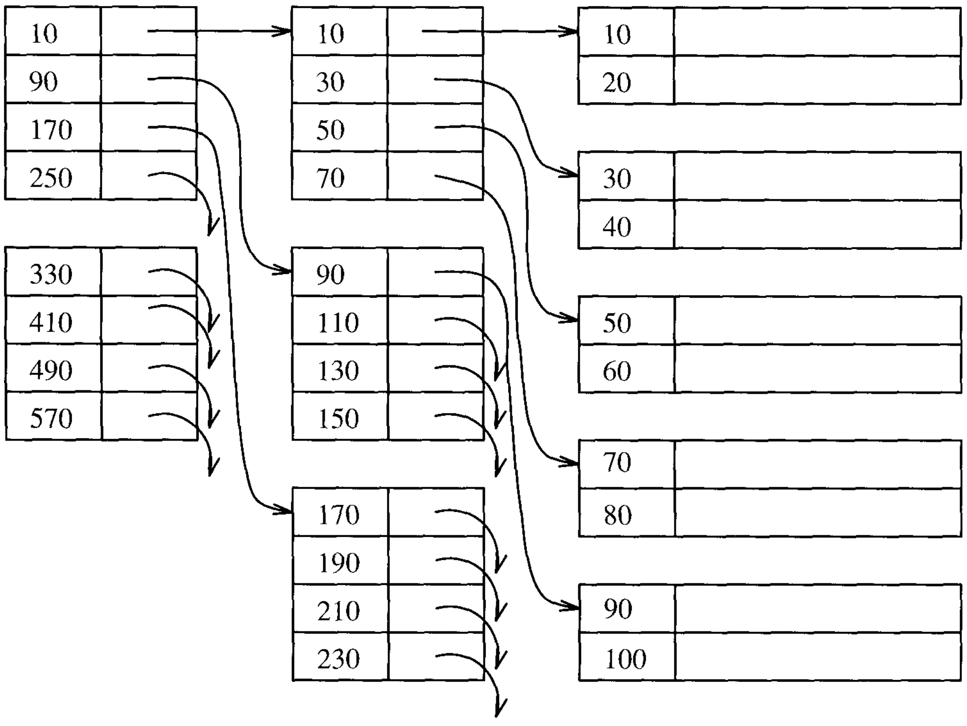

Figure 4.5: Adding a second level of sparse index

In this example, the first-level index is sparse, although we could have chosen a dense index for the first level. However, the second and higher levels must be sparse. The reason is that a dense index on an index would have exactly as many key-pointer pairs as the first-level index, and therefore would take exactly as much space as the first-level index. A second-level dense index thus introduces additional structure for no advantage.

Example 4.5: Continuing with a study of the hypothetical relation of Example 4.4, suppose we put a second-level index on the first-level sparse index. Since the first-level index occupies 1000 blocks, and we can fit 100 key-pointer pairs in a block, we need 10 blocks for the second-level index.

It is very likely that these 10 blocks can remain buffered in memory. If so, then to find the record with a given key K, we look up in the second-level index to find the largest key less than or equal to K. The associated pointer leads to a block B of the first-level index that will surely guide us to the desired record. We read block B into memory if it is not already there; this read is the first disk I/O we need to do. We look in block B for the greatest key less than or equal to K, and that key gives us a data block that will contain the record with key K if such a record exists. That block requires a second disk I/O, and we are done, having used only two I/O's. □

4.1.5 Indexes With Duplicate Search Keys

Until this point we have supposed that the search key, upon which the index is based, was also a key of the relation, so there could be at most one record with any key value. However, indexes are often used for nonkey attributes, so it is possible that more than one record has a given key value. If we sort the records by the search key, leaving records with equal search key in any order, then we can adapt the ideas mentioned earlier to search keys that are not keys of the relation.

Perhaps the simplest extension of previous ideas is to have a dense index with one entry with key K for each record of the data file that has search key K. That is, we allow duplicate search keys in the index file. Finding all the records with a given search key K is thus simple: Look for the first K in the index file, find all the other K's, which must immediately follow, and pursue all the associated pointers to find the records with search key K.

A slightly more efficient approach is to have only one record in the dense index for each search key K. This key is associated with a pointer to the first of the records with K. To find the others, move forward in the data file to find any additional records with K; these must follow immediately in the sorted order of the data file. Figure 4.6 illustrates this idea.

Example 4.6: Suppose we want to find all the records with search key 20 in Fig. 4.6 We find the 20 entry in the index and follow its pointer to the first record with search key 20. We then search forward in the data file. Since we are at the last record of the second block of this file, we move forward to the third block.[1] We find the first record of this block has 20, but the second has 30. Thus, we need search no further; we have found the two records with search key 20. □

Figure 4.7 shows a sparse index on the same data file as Fig. 4.6. The sparse index is quite conventional; it has key-pointer pairs corresponding to the first search key on each block of the data file.

To find the records with search key K *in* this data structure, we find the last entry of the index, call it E_1, that has a key less than or equal to K. We then move towards the front of the index until we either come to the first entry or we come to an entry E_2 with a key strictly less than K. All the data blocks that might have a record with search key K are pointed to by the index entries from $E'2$ to E_1, inclusive.

[1]To find the next block of the data file, we could chain the blocks in a linked list, i.e , give each block a pointer to the next. We could also go back to the index and follow the next pointer of the index to the next data-file block.

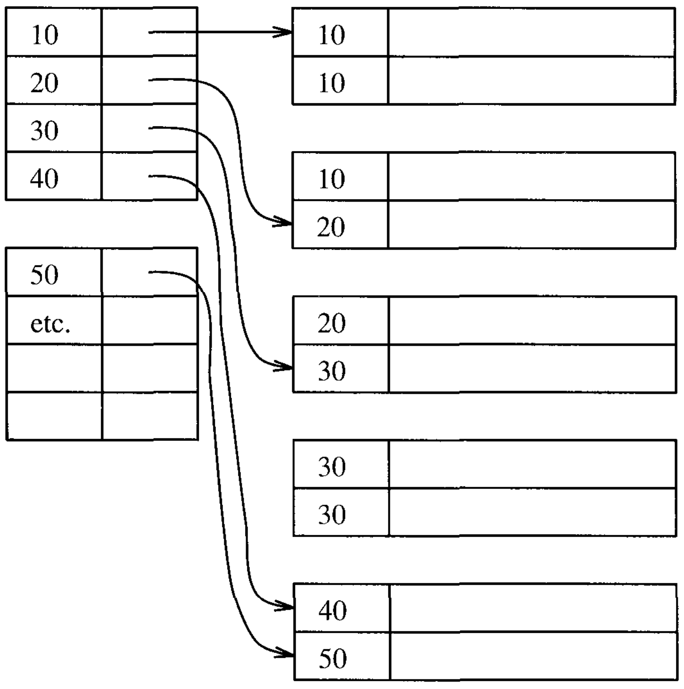

Figure 4.6: A dense index when duplicate search keys are allowed

Example 4.7: Suppose we want to look up key 20 in Fig. 4.7. The third entry in the first index block is E_1; it is the last entry with a key <20. When we search backward, we immediately find an entry with a key smaller than 20. Thus, the second entry of the first index block is E_2. The two associated pointers take us to the second and third data blocks, and it is on these two blocks that we find records with search key 20.

For another example, if $K = 10$, then E_1 is the second entry of the first index block, and E_2 doesn't exist because we never find a smaller key. Thus, we follow the pointers in all index entries up to and including the second. That takes us to the first two data blocks, where we find all of the records with search key 10. □

A slightly different scheme is shown in Fig. 4.8. There, the index entry for a data block holds the smallest search key that is *new;* i.e., it did not appear in a previous block. If there is no new search key in a block, then its index entry holds the lone search key found in that block. Under this scheme, we can find the records with search key K by looking in the index for the first entry whose key is either

a) Equal to K, or

b) Less than K, but the next key is greater than K.

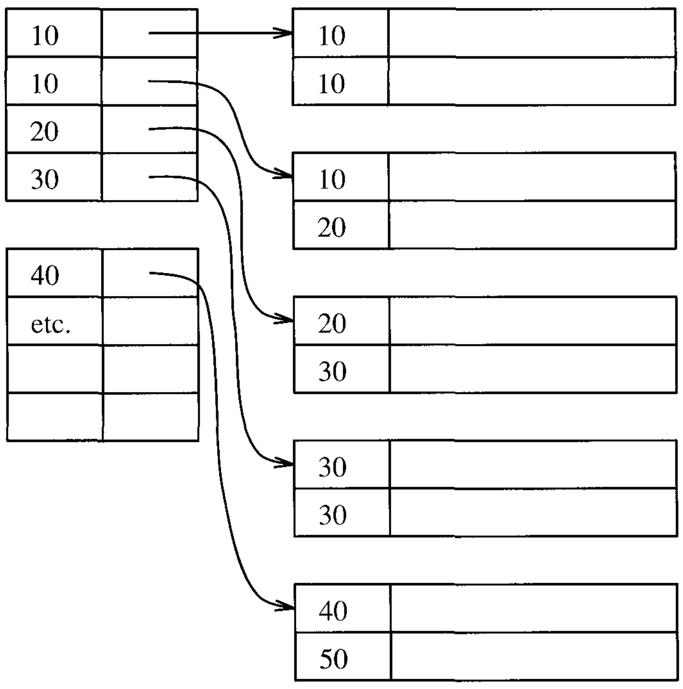

Figure 4.7: A sparse index indicating the lowest search key in each block

We follow the pointer in this entry, and if we find at least one record with search key K in that block, then we search forward through additional blocks until we find all records with search key K.

Example 4.8: Suppose that K — 20 in the structure of Fig. 4.8. The second index entry is indicated by the above rule, and its pointer leads us to the first block with 20. We must search forward, since the following block also has a 20.

If K = 30, the rule indicates the third entry. Its pointer leads us to the third data block, where the records with search key 30 begin. Finally, if K = 25, then part (b) of the selection rule indicates the second index entry. We are thus led to the second data block. If there were any records with search key 25, at least one would have to follow the records with 20 on that block, because we know that the first new key in the third data block is 30. Since there are no 25's, we fail in our search. □

4.1.6 Managing Indexes During Data Modifications

Until this point, we have shown data files and indexes as if they were sequences of blocks, fully packed with records of the appropriate type. Since data evolves with time, we expect that records will be inserted, deleted, and sometimes updated. As a result, an organization like a sequential file will evolve so that

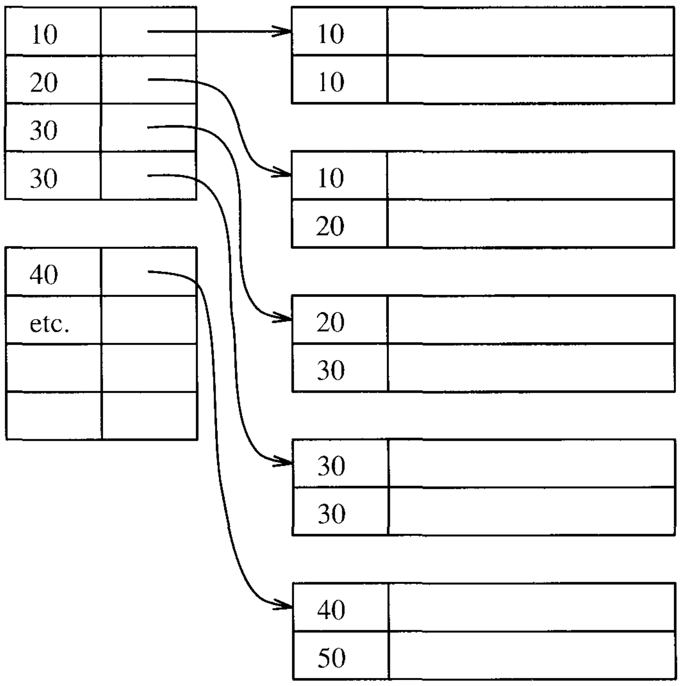

Figure 4.8: A sparse index indicating the lowest new search kev in each block

what once fit in one block no longer does. We can *use* the techniques discussed in Section 3.5 to reorganize the data file. Recall that the three big ideas from that section arc:

1. Create overflow blocks if extra space is needed, or delete overflow blocks if enough records are deleted that the space is no longer needed. Overflow blocks do not have entries in a sparse index. Rather, they should be considered as extensions of their primary block.

2. Instead of overflow blocks, we may be able to insert new blocks in the sequential order. If we do, then the new block needs an entry in a sparse index. We should remember that changing an index can create the same kinds of problems on the index file that insertions and deletions to the data file create. If we create new index blocks, then these blocks must be located somehow, e.g., with another level of index as in Section 4.1.4.

3 When there is no room to insert a tuple into a block, we can sometimes slide tuples to adjacent blocks. Conversely, if adjacent blocks grow too empty, they can be combined.

However, when changes occur to the data file, we must often change the index to adapt. The correct approach depends on whether the index is dense or sparse, and on which of the three actions discussed above is used. However, one general principle should be remembered:

- An index file is an example of a sequential file; the key-pointer pairs can be treated as records sorted by the value of the search key. Thus, the same strategies used to maintain data files in the face of modifications can be applied to its index file.

In Fig. 4.9, we summarize the actions that must be taken on a sparse or dense index when seven different actions on the data file are taken. These seven actions include creating or deleting empty overflow blocks, creating or deleting empty blocks of the sequential file, inserting, deleting, and moving records. Notice that we assume only empty blocks can be created or destroyed. In particular, if we want to delete a block that contains records, we must first delete the records or move them to another block.

Action	Dense Index	Sparse Index
Create empty overflow block	none	none
Delete empty overflow block	none	none
Create empty sequential block	none	insert
Delete empty sequential block	none	delete
Insert record	insert	update(?)
Delete record	delete	update(?)
Slide record	update	update(?)

Figure 4.9: How actions on the sequential file affect the index file

In this table, we notice the following:

- Creating or destroying an empty overflow block has no effect on either type of index. It has no effect on a dense index, because that index refers to records. It has no effect on a sparse index, because it is only the primary blocks, not the overflow blocks, that have entries in the sparse index.

- Creating or destroying blocks of the sequential file has no effect on a dense index, again because that index refers to records, not blocks. It *does* affect a sparse index, since we must insert or delete an index entry for the block created or destroyed, respectively.

- Inserting or deleting records results in the same action on a dense index, as a key-pointer pair for that record is inserted or deleted. However, there is typically no effect on a sparse index. The exception is when the record is the first of its block, in which case the corresponding key value in the sparse index must be updated. Thus, we have put a question mark after

Preparing for Evolution of Data

Since it is common for relations or class extents to grow with time, it is often wise to distribute extra space among blocks — both data and index blocks. If blocks are, say, 75% full to begin with, then we can run for some time before having to create overflow blocks or slide records between blocks. The advantage to having no overflow blocks, or few overflow blocks, is that the average record access then requires only one disk I/O. The more overflow blocks, the higher will be the average number of blocks we need to look at in order to find a given record.

"update" for these actions in the table of Fig. 4.9, indicating that the update is possible, but not certain.

- Similarly, sliding a record, whether within a block or between blocks, results in an update to the corresponding entry of a dense index, but only affects a sparse index if the moved record was or becomes the first of its block.

We shall illustrate the family of algorithms implied *by* these rules in a series of examples. These examples involve both sparse and dense indexes and both "record sliding" and overflow-block approaches.

Example 4.9 : First, let us consider the deletion of a record from a sequential file with a dense index. We begin with the file and index of Fig. 4.3. Suppose that the record with key 30 is deleted. Figure 4.10 shows the result of the deletion.

First, the record 30 is deleted from the sequential file. We assume that there are possible pointers from outside the block to records in the block, so we have elected not to slide the remaining record, 40, forward in the block. Rather, we suppose that a tombstone has been left in place of the record 30.

In the index, we delete the key-pointer pair for 30. We suppose that there cannot be pointers to index records from outside, so there is no need to leave a tombstone for the pair. Therefore, we have taken the option to consolidate the index block and move following records forward. □

Example 4.10: Now, let us consider two deletions from a file with a sparse index. We begin with the structure of Fig. 4.4 and again suppose that the record with key 30 is deleted. We also assume that there is no impediment to sliding records around in blocks — either we know there are no pointers to records from anywhere, or we are using an offset table as in Fig. 3.17 to support such sliding.

The effect of the deletion of record 30 is shown in Fig. 4.11. The record has been deleted, and the following record, 40, slides forward to consolidate the

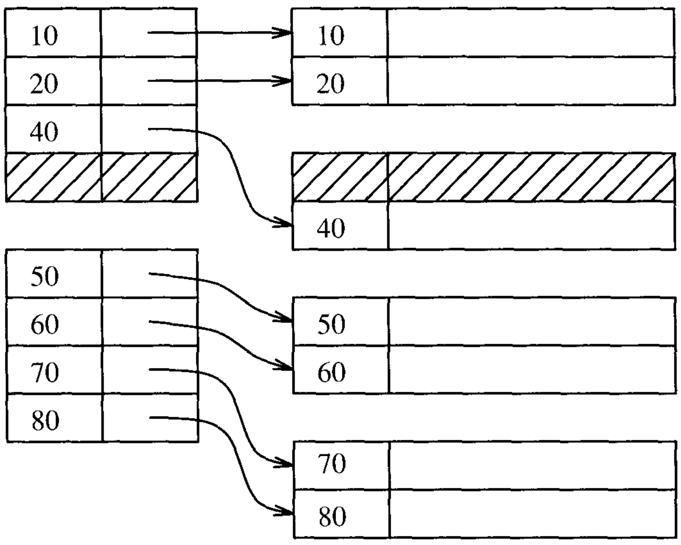

Figure 4.10: Deletion of record with search key 30 in a dense index

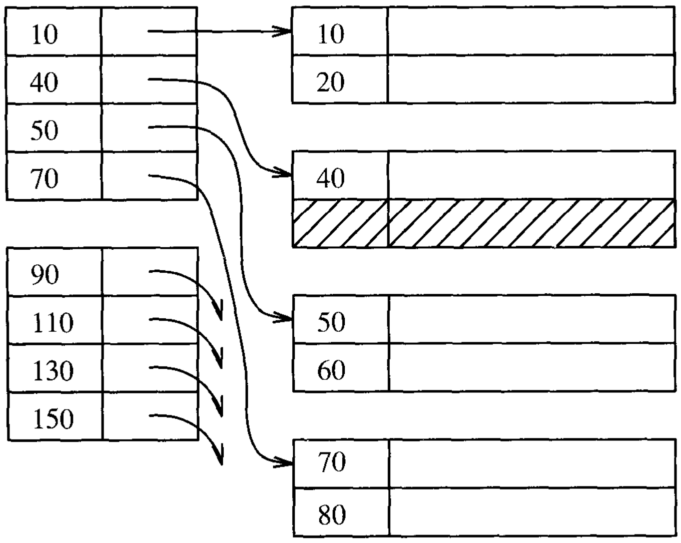

Figure 4.11: Deletion of record with search key 30 in a sparse index

block at the front. Since 40 is now the first key on the second data block, we need to update the index record for that block. We see in Fig. 4.11 that the key associated with the pointer to the second data block has been updated from 30 to 40.

Now, suppose that record 40 is also deleted. We see the effect of this action in Fig. 4.12. The second data block now has no records at all. If the sequential file is stored on arbitrary blocks (rather than, say, consecutive blocks of a cylinder), then we may link the unused block to a list of available space.

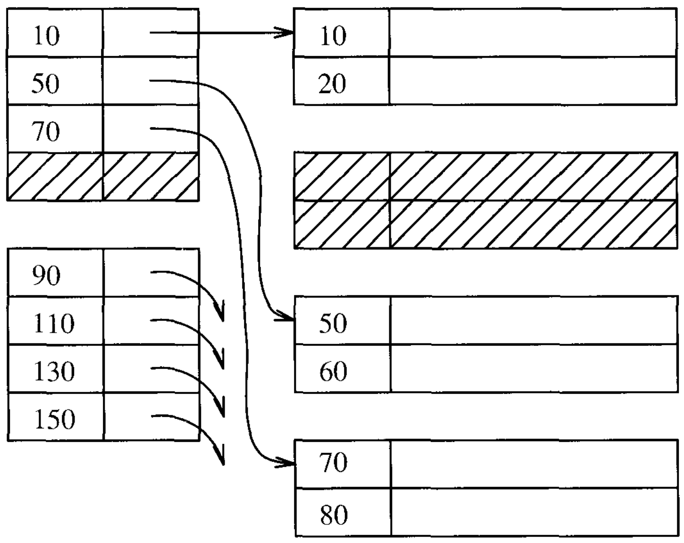

Figure 4.12: Deletion of record with search key 40 in a sparse index

We complete the deletion of record 40 by adjusting the index. Since the second data block no longer exists, we delete its entry from the index. We also show in Fig 4.12 the first index block having been consolidated by moving forward the following pairs. That step is optional. □

Example 4.11: Now, let us consider the effect of an insertion. Begin at Fig. 4.11, where we have just deleted record 30 from the file with a sparse index, but the record 40 remains. We now insert a record with key 15. Consulting the sparse index, we find that this record belongs in the first data block. But that block is full; it holds records 10 and 20.

One thing we can do is look for a nearby block with some extra space, and in this case we find it in the second data block. We thus slide blocks backward in the file to make room for record 15. The result is shown in Fig. 4.13. Record 20 has been moved from the first to the second data block, and 15 put in its place. To fit record 20 on the second block and keep records sorted, we slide record 40 back in the second block and put 20 ahead of it.

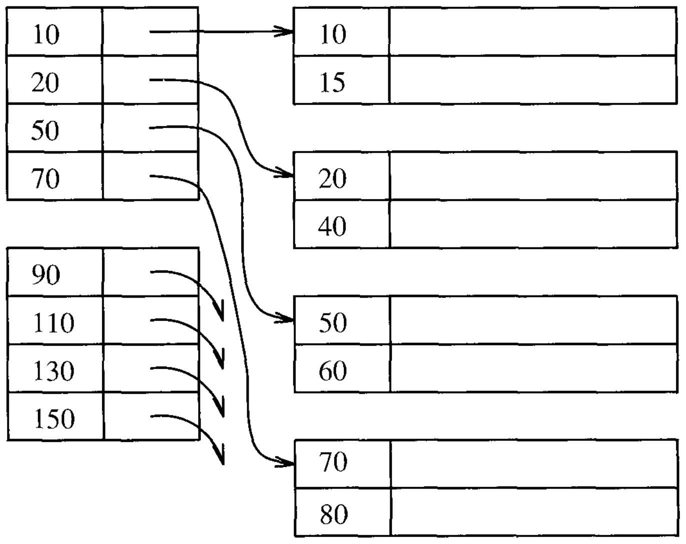

Figure 4.13: Insertion into a file with a sparse index, using immediate reorganization

Our last step is to modify the index entries of the changed blocks. Wo might have to change the key in the index pair for block 1, but we do not in this case, because the inserted record is not the first in its block. We do, however, change the key in the index entry for the second data block, since the first record of that block, which used to be 40, is now 20. □

Example 4.12: The problem with the strategy exhibited in Example 4.11 is that we were lucky to find an empty space in an adjacent data block. Had the record with key 30 not been deleted previously, we would have searched in vain for an empty space. In principle, we would have had to slide every record from 20 to the end of the file back until we got to the end of the file and could create an additional block.

Because of this risk, it is often wiser to allow overflow blocks to supplement the space of a primary block that has too many records. Figure 4.14 shows the effect of inserting a record with key 15 into the structure of Fig. 4.11. As in Example 4.11, the first data block has too many records. Instead of sliding records to the second block, we create an overflow block for the data block. We have shown in Fig. 4.14 a "nub" on each block, representing a place in the block header where a pointer to an overflow block may be placed. Any number of overflow blocks may be linked in a chain using these pointer spaces.

In our example, record 15 is inserted in its rightful place, after record 10. Record 20 slides to the overflow block to make room. No changes to the index are necessary, since the first record in data block 1 has not changed. Notice that no index entry is made for the overflow block, which is considered an extension

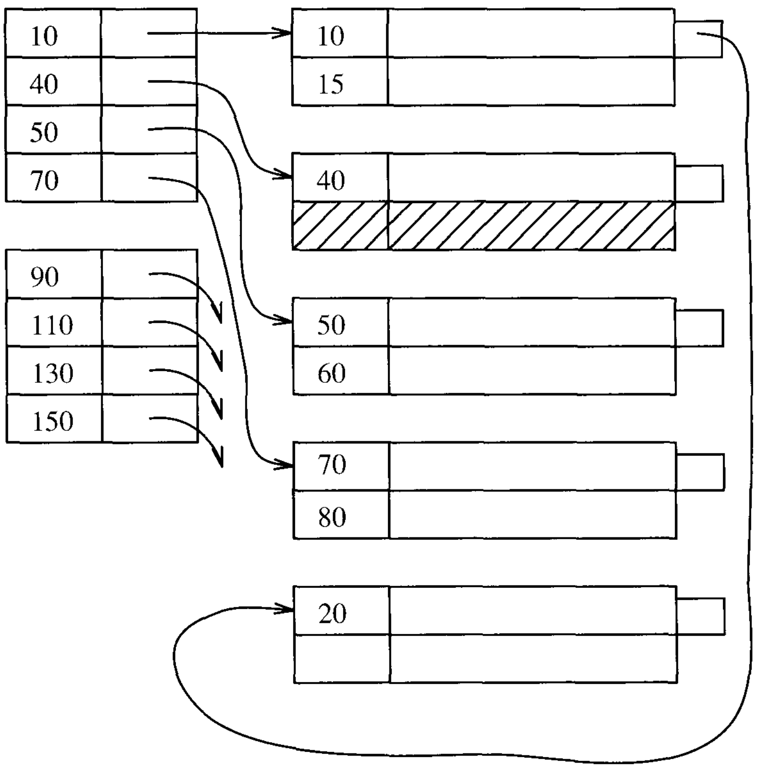

Figure 4.14: Insertion into a file with a sparse index, using overflow blocks

of data block 1, not a block of the sequential file on its own. C

4.1.7 Exercises for Section 4.1

* **Exercise 4.1.1:** Suppose blocks hold either three records, or ten key-pointer pairs. As a function of *n,* the number of records, how many blocks do we need to hold a data file and:

a) A dense index?

b) A sparse index?

Exercise 4.1.2: Repeat Exercise 4.1.1 if blocks can hold up to 30 records or 200 key-pointer pairs, but neither data- nor index-blocks are allowed to be more than 80% full.

! **Exercise 4.1.3:** Repeat Exercise 4.1.1 if we use as many levels of index as is appropriate, until the final level of index has only one block.

***!! Exercise 4.1.4:** Suppose that blocks hold three records or ten key-pointer pairs, as in Exercise 4.1.1, but duplicate search keys are possible. To be specific, 1/3 of all search keys in the database appear in one record, 1/3 appear in exactly two records, and 1/3 appear in exactly three records. Suppose we have a dense index, but there is only one key-pointer pair per search-key value, to the first of the records that has that key. If no blocks are in memory initially, compute the average number of disk I/O's needed to find all the records with a given search key K. You may assume that the location of the index block containing key K is known, although it is on disk.

! Exercise 4.1.5: Repeat Exercise 4.1.4 for:

a) A dense index with a key-pointer pair for each record, including those with duplicated keys.

b) A sparse index indicating the lowest key on each data block, as in Fig. 4.7.

c) A sparse index indicating the lowest *new* key on each data block, as in Fig. 4.8.

! Exercise 4.1.6: If we have a dense index on the primary key attribute of a relation, then it is possible to have pointers to tuples (or the records that represent those tuples) go to the index entry rather than to the record itself. What are the advantages of each approach?

Exercise 4.1.7: Continue the changes to Fig. 4.13 if we next delete the records with keys 60, 70, and 80, then insert records with keys 21, 22, and so on, up to 29. Assume that extra space is obtained by:

* a) Adding overflow blocks to either the data file or index file.

b) Sliding records as far back as necessary, adding additional blocks to the end of the data file and/or index file if needed.

c) Inserting new data or index blocks into the middle of these files as necessary.

***! Exercise 4.1.8:** Suppose that we handle insertions into a data file of n records by creating overflow blocks as needed. Also, suppose that the data blocks are currently half full on the average. If we insert new records at random, how many records do we have to insert before the average number of data blocks (including overflow blocks if necessary) that we need to examine to find a record with a given key reaches 2? Assume that on a lookup, we search the block pointed to by the index first, and only search overflow blocks, in order, until we find the record, which is definitely in one of the blocks of the chain.

4.2 Secondary Indexes

The data structures described in Section 4.1 are called *primary indexes*, because they determine the location of the indexed records. In Section 4.1, the location was determined by the fact that the underlying file was sorted on the search key. Section 4.4 will discuss another common example of a primary index: a hash table in which the search key determines the "bucket" into which the record goes.

However, frequently we want several indexes on a relation, to facilitate a variety of queries. For instance, consider again the MovieStar relation declared in Fig. 3.1. Since we declared name to be the primary key, we expect that the DBMS will create a primary index structure to support queries that specify the name of the star. However, suppose we also want to use our database to acknowledge stars on milestone birthdays. We may then run queries like

```
SELECT name, address
FROM MovieStar
WHERE birthdate = DATE '1950-01-01';
```

We need a *secondary index* on birthdate to help with such queries. In an SQL system, we might call for such an index by an explicit command such as

```
CREATE INDEX BDIndex ON MovieStar(birthdate);
```

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

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

- Thus, secondary indexes are always dense.

4.2.1 Design of Secondary Indexes

A secondary index is a dense index, usually with duplicates. As before, this index consists of key-pointer pairs; the "key" is a search key and need not be unique. Pairs in the index file are sorted by key value, to help find the entries given a key. If we wish to place a second level of index on this structure, then that index would be sparse, for the reasons discussed in Section 4.1.4.

Example 4.13: Figure 4.15 shows a typical secondary index. The data file is shown with two records per block, as has been our standard for illustration. The records have only their search key shown; this attribute is integer valued, and as before we have taken the values to be multiples of 10. Notice that, unlike the data file in Section 4.1.5, here the data is not sorted by the search key.

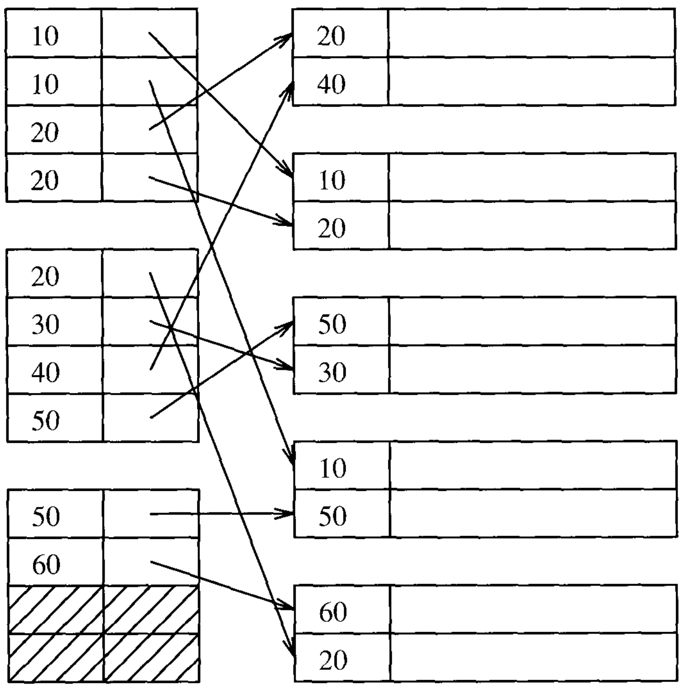

Figure 4.15: A secondary index

However, the keys in the index file *are* sorted. The result is that the pointers in one index block can go to many different data blocks, instead of one or a few consecutive blocks. For example, to retrieve all the records with search key 20, we not only have to look at two index blocks, but we are sent by their pointers to three different data blocks. Thus, using a secondary index may result in many more disk I/O's than if we get the same number of records via a primary index. However, there is no help for this problem; we cannot control the order of tuples in the data block, because they are presumably ordered according to some other attribute(s).

It would be possible to add a second level of index to Fig. 4.15. This level would be sparse, with pairs corresponding to the first key or first new key of each index block, as discussed in Section 4.1.4. D

4.2.2 Applications of Secondary Indexes

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

A second common structure needing secondary indexes is the *clustered file*. In this structure, two or more relations are stored with their records intermixed. An example will illustrate why this organization makes good sense in special situations.

Example 4.14: Suppose we have two relations, whose schemas we may describe briefly as

```
Movie(title, year, length, studioName)
Studio(name, address, president)
```

Attributes title and year together are the key for Movie, while name is the key for Studio. Attribute studioName in Movie is a foreign key referencing name in Studio. Suppose further that a common form of query is:

```
SELECT title, year
FROM Movie
WHERE studioName = 'zzz';
```

Here, *zzz* is intended to represent the name of a particular studio, e.g. 'Disney'.

If we are convinced that the above is a typical query, then instead of ordering Movie tuples by the primary key title and year, we can order the tuples by studioName. We could then place on this sequential file a primary index with duplicates, as was discussed in Section 4.1.5. The value of doing so is that when we query for the movies by a given studio, we find all our answers on a few blocks, perhaps one more than the minimum number on which they could possibly fit. That minimizes disk I/O's for this query and thus makes the answering of this query form very efficient.

However, merely sorting the Movie tuples by an attribute other than its primary key will not help if we need to relate information about movies to information about studios, such as:

```
SELECT president
FROM Movie, Studio
WHERE title = 'Star Wars' AND
      Movie.studioName = Studio.name
```

i.e., find the president of the studio that made "Star Wars," or:

```
SELECT title, year
FROM Movie, Studio
WHERE address LIKE '%Hollywood%' AND
      Movie.studioName = Studio.name
```

i.e., find all the movies that were made in Hollywood. For these queries, we need to join Movie and Studio.

If we are sure that joins on the studio name between relations Movie and Studio will be common, we can make those joins efficient by choosing a *clustered file structure,* where the Movie tuples are placed with Studio tuples in the same sequence of blocks. More specifically, we place after each Studio tuple all the Movie tuples for the movies made by that studio. The pattern is suggested in Fig. 4.16.

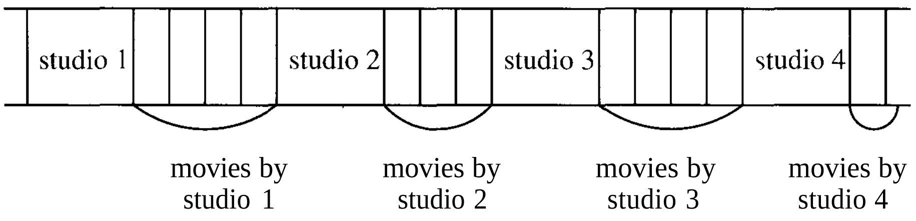

Figure 4.16: A clustered file with each studio clustered with the movies made by that studio

Now, if we want the president of the studio that made a particular movie, we have a good chance of finding the record for the studio and the movie on the same block, saving an I/O step. If we want the movies made by certain studios, we again will tend to find those movies on the same block as one of the studios, saving I/O's.

If these queries are to be efficient, however, we need to find the given movie or given studio efficiently. Thus, we need a secondary index on `Movie.title` to find the movie (or movies, since two or more movies with the same title can exist) with that title, wherever they may be among the blocks that hold Movie and Studio tuples. We also need an index on `Studio.name` to find the tuple for a given studio. □

4.2.3 Indirection in Secondary Indexes

There is some wasted space, perhaps a significant amount of wastage, in the structure suggested by Fig. 4.15. If a search-key value appears n times in the data file, then the value is written n times in the index file. It would be better if we could write the key value once for all the pointers to data records with that value.

A convenient way to avoid repeating values is to use a level of indirection, called *buckets*, between the secondary index file and the data file. As shown in Fig. 4.17, there is one pair for each search key K. The pointer of this pair goes to a position in a "bucket file," which holds the "bucket" for K. Following this position, until the next position pointed to by the index, are pointers to all the records with search-key value K.

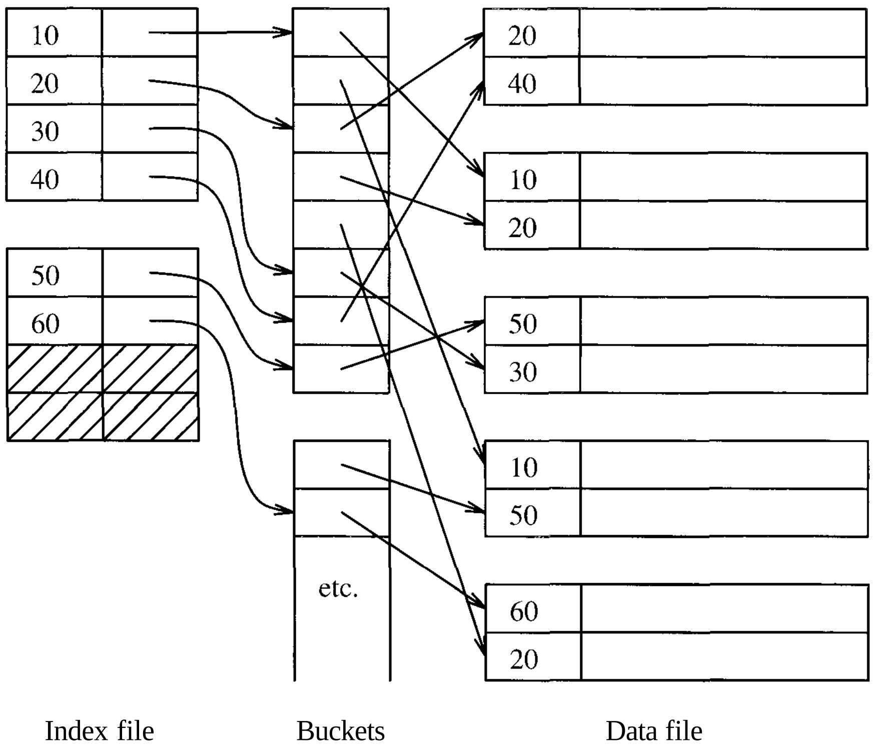

Figure 4.17: Saving space by using indirection in a secondary index

Example 4.15 : For instance, if we follow the pointer from search key 50 in the index file of Fig. 4.17 to the intermediate "bucket" file. This pointer happens to take us to the last pointer of one block of the bucket file. We search forward, to the first pointer of the next block. We stop at that point, because the next pointer of the index file, associated with search key 60, points to the second pointer of the second block of the bucket file. □

The scheme suggested *by* Fig. 4.17 will save space as long as search-key values are larger than pointers. However, even when the keys and pointers are comparable in size, there is an important advantage to using indirection with secondary indexes: often, we can use the pointers in the buckets to help answer queries without ever looking at most of the records in the data file. Specifically, when there are several conditions to a query, and each condition has a secondary index to help it, we can find the bucket pointers that satisfy all the conditions by intersecting sets of pointers in memory, and retrieving only the records pointed to by the surviving pointers. We thus save the I/O cost of retrieving records that satisfy some, but not all, of the conditions.[2]

[2]We could also use this pointer-intersection trick if we got the pointers directly from the

Example 4.16 : Consider the relation

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

of Example 4.14. Suppose we have secondary indexes with indirect buckets on both studioName and year, and we are asked the query

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

that is, find all the Disney movies made in 1995.

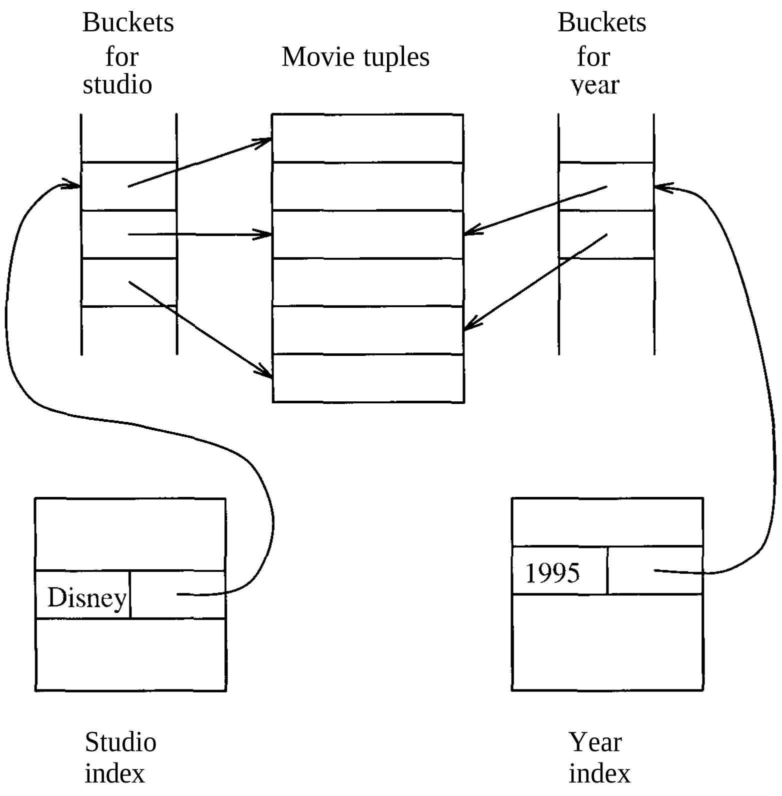

Figure 4.18: Intersecting buckets in main memory

Figure 4.18 shows how we can answer this query using the indexes. Using the index on studioName, we find the pointers to all records for Disney movies,

index, rather than from buckets. However, the use of buckets often saves disk I/O's, since the pointers use less space than key-pointer pairs.

but we do not yet bring any of those records from disk to memory. Instead, using the index on year, we find the pointers to all the movies of 1995. We then intersect the two sets of pointers, getting exactly the movies that were made by Disney in 1995. Now, we need to retrieve from disk all the blocks holding one or more of these movies, thus retrieving the minimum possible number of data-blocks. □

4.2.4 Document Retrieval and Inverted Indexes

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

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

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

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

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

- Instead of creating a separate index for each attribute (i.e., for each word), the indexes are combined into one, called an *inverted index*. This index uses indirect buckets for space efficiency, as was discussed in Section 4.2.3.

Example 4.17: An inverted index is illustrated in Fig. 4.19. In place of a data file of records is a collection of documents, each of which may be stored on one or more disk blocks. The inverted index itself consists of a set of word-pointer pairs; the words are in effect the search key for the index. The inverted index is kept in a sequence of blocks, just like any of the indexes discussed so far. However, in some document-retrieval applications, the data may be more static than the typical database, so there may be no provision for overflow of blocks or changes to the index in general.

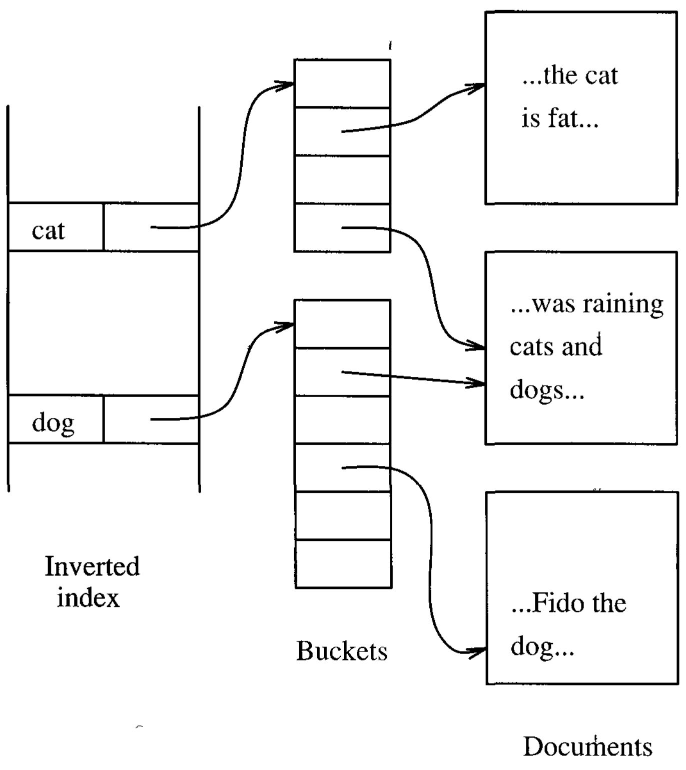

Figure 4.19: An inverted index on documents

The pointers refer to positions in a "bucket" file. For instance, we have shown in Fig. 4.19 the word "cat" with a pointer to the bucket file. Following the position of the bucket file pointed to are pointers to all the documents that contain the word "cat." We have shown some of these in the figure. Similarly, the word "dog" is shown pointing to a list of pointers to all the documents with "dog." □

Pointers in the bucket file can be:

1. Pointers to the document itself.

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

Once we have the idea of using "buckets" of pointers to occurrences of each word, we may also want to extend the idea to include in the bucket array some information about the occurrence. Now, the bucket file itself becomes a collection of records with important structure. Early uses of the idea distinguished

More About Information Retrieval

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

1. *Stemming.* We remove suffixes to find the "stem" of each word, before entering its occurrence into the index. For example, plural nouns can be treated as their singular versions. Thus, in Example 4.17, the inverted index evidently uses stemming, since the search for word "dog" got us not only documents with "dog," but also a document with the word "dogs."

2. *Stop words.* The most common words, such as "the" or "and," arc called *stop words* and are often not included in the inverted index. The reason is that the several hundred most common words appear in too many documents to make them useful as a way to find documents about specific subjects. Eliminating stop words also reduces the size of the index significantly.

occurrences of a word in the title of a document, the abstract, and the body of text. With the growth of documents on the Web, especially documents using HTML, XML, or another markup language, we can also indicate the markings associated with words. For instance, we can distinguish words appearing in titles headers, tables, or anchors, as well as words appearing in different fonts or sixes.

Example 4.18: Figure 4.20 illustrates a bucket file that has been used to indicate occurrences of words in HTML documents. The first column indicates the type of occurrence, i.e., its marking, if any. The second and third columns are together the pointer to the occurrence. The third column indicates the document, and the second column gives the number of the word in the document.

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

a) Mention dogs in the title, and

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

Insertion and Deletion From Buckets

We show buckets in figures such as Fig. 4.19 as compacted arrays of appropriate size. In practice, they are records with a single field (the pointer) and are stored in blocks like any other collection of records. Thus, when we insert or delete pointers, we may use any of the techniques seen so far, such as leaving extra space in blocks for expansion of the file, overflow blocks, and possibly moving records within or among blocks. In the latter case, we must be careful to change the pointer from the inverted index to the bucket file, as we move the records it points to.

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

4.2.5 Exercises for Section 4.2

* **Exercise 4.2.1:** As insertions and deletions are made on a data file, a secondary index file needs to change as well. Suggest some ways that the secondary index can be kept up to date as the data file changes.

! Exercise **4.2.2:** Suppose we have blocks that can hold three records or ten key-pointer pairs, as in Exercise 4.1.1. Let these blocks be used for a data file and a secondary index on search key K. For each K-value v present in the file, there are either 1, 2, or three records with v in field K. Exactly 1/3 of the values appear once, 1/3 appear twice, and 1/3 appear three times. Suppose further that the index blocks and data blocks are all on disk, but there is a structure that allows us to take any K-value v and get pointers to all the index blocks that have search-key value v in one or more records (perhaps there is a second level of index in main memory). Calculate the average number of disk I/O's necessary to retrieve all the records with search-key value v.

***! Exercise 4.2.3** : Consider a clustered file organization like Fig. 4.16, and suppose that ten records, either studio records or movie records, will fit on one block. Also assume that the number of movies per studio is uniformly distributed between 1 and m. As a function of m, what is the averge number of disk I/0's needed to retrieve a studio and all its movies? What would the number be if movies were randomly distributed over a large number of blocks?

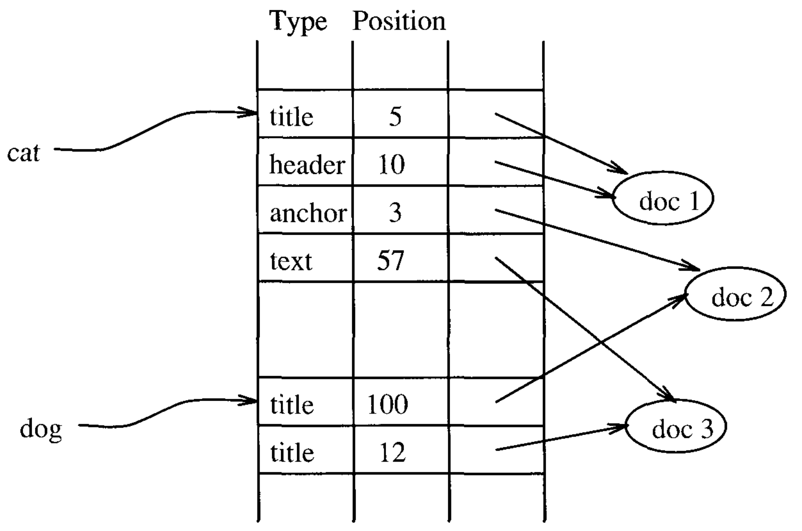

Figure 4.20: Storing more information in the inverted index

Exercise 4.2.4: Suppose that blocks can hold either three records, ten key-pointer pairs, or fifty pointers. If we use the indirect buckets scheme of Fig. 4.17:

* a) If the average search-key value appears in 10 records, how many blocks do we need to hold 3000 records and its secondary index structure? How many blocks would be needed if we did *not* use buckets?

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

! **Exercise 4.2.5** : On the assumptions of Exercise 4.2.4(a), what is the average number of disk I/O's to find and retrieve the ten records with a given search-key value, both with and without the bucket structure? Assume nothing is in memory to begin, but it is possible to locate index or bucket blocks without incurring additional I/O's beyond what is needed to retrieve these blocks into memory.

Exercise 4.2.6: Suppose that as in Exercise 4.2.4, a block can hold either three records, ten key-pointer pairs, or fifty pointers. Let there be secondary indexes on `studioName` and year of the relation Movie, as in Example 4.16. Suppose there are 51 Disney movies, and 101 movies made in 1995. Only one of these movies was a Disney movie. Compute the number of disk I/0's needed to answer the query of Example 4.16 (find the Disney movies made in 1995) if we:

* a) Use buckets for both secondary indexes, retrieve the pointers from the buckets, intersect them in main memory, and retrieve only the one record for the Disney movie of 1995.

b) Do not use buckets, use the index on `studioName` to get the pointers to Disney movies, retrieve them, and select those that were made in 1995. Assume no two Disney movie records are on the same block.

c) Proceed as in (b), but starting with the index on year. Assume no two movies of 1995 are on the same block.

Exercise 4.2.7: Suppose we have a repository of 1000 documents, and we wish to build an inverted index with 10,000 words. A block can hold ten word-pointer pairs or 50 pointers to either a document or a position within a document. The distribution of words is Zipfian (see the box on "The Zipfian Distribution" in Section 7.4.3); the number of occurrences of the ith most frequent word is $100000/\sqrt{i}$, for $i = 1, 2, \ldots, 10000$.

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

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

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

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

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

Exercise 4.2.8: If we use an augmented inverted index, such as in Fig. 4.20, we can perform a number of other kinds of searches. Suggest how this index could be used to find:

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

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

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

4.3 B-Trees

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

- B-trees automatically maintain as many levels of index as is appropriate for the size of the file being indexed.

- B-trees manage the space on the blocks they use so that every block is between half used and completely full. No overflow blocks are ever needed for the index.

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

4.3.1 The Structure of B-trees

As implied by the name, a B-tree organizes its blocks into a tree. The tree is *balanced*, meaning that all paths from the root to a leaf have the same length. Typically, there are three layers in a B-tree: the root, an intermediate layer, and leaves, but any number of layers is possible. To help visualize B-trees, you may wish to look ahead at Figs. 4.21 and 4.22, which show nodes of a B-tree, and Fig. 4.23, which shows a small, complete B-tree.

There is a parameter n associated with each B-tree index, and this parameter determines the layout of all blocks of the B-tree. Each block will have space for n search-key values and $n + 1$ pointers. In a sense, a B-tree block is similar to the index blocks introduced in Section 4.1, except that the B-tree block has an extra pointer, along with n key-pointer pairs. We pick n to be as large as will allow $n + 1$ pointers and n keys to fit in one block.

Example 4.19: Suppose our blocks are 4096 bytes. Also let keys be integers of 4 bytes and let pointers be 8 bytes. If there is no header information kept on the blocks, then we want to find the largest integer value of n such that $4n + 8(n + 1) < 4096$. That value is $n = 340$. n

There are several important rules that constrain what can appear in the blocks of a B-tree.

- At the root, there are at least two used pointers.[3] All pointers point to B-tree blocks at the level below.

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

- At a leaf, the last pointer points to the next leaf block to the right, i.e., to the block with the next higher keys. Among the other n pointers in a leaf block, at least $\lfloor \frac{n+1}{2} \rfloor$ of these pointers are used and point to data records; unused pointers may be thought of as null and do not point anywhere. The ith pointer, if it is used, points to a record with the ith key.

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

- Suppose we draw a B-tree in the conventional manner for trees, with the children of a given node placed in order from left (the "first child")to right (the "last child"). Then if we look at the nodes of the B-tree at any one level, from left to right, the keys at those nodes appear in nondecreasing order.

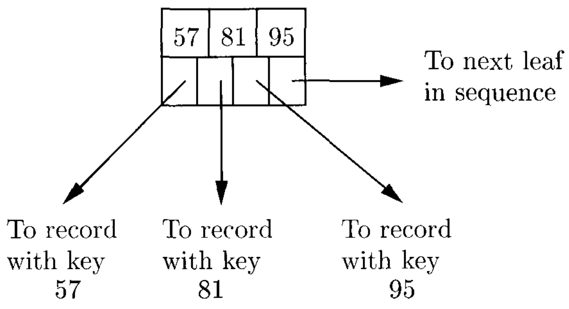

Figure 4.21: A typical leaf of a B+ tree

Example 4.20: In this and our running examples of B-trees, we shall use $n = 3$. That is, blocks have room for three keys and four pointers, which are atypically small numbers. Keys are integers. Figure 4.21 shows a leaf that is completely used. There are three keys, 57, 81, and 95. The first three pointers go to records with these keys. The last pointer, as is always the case with leaves, points to the next leaf to the right in the order of keys; it would be null if this leaf were the last in sequence.

A leaf is not necessarily full, but in our example with $n = 3$, there must be at least two key-pointer pairs. That is, the key 95 in Fig. 4.21 might be missing, and with it the third of the pointers, the one labeled "to record with key 95."

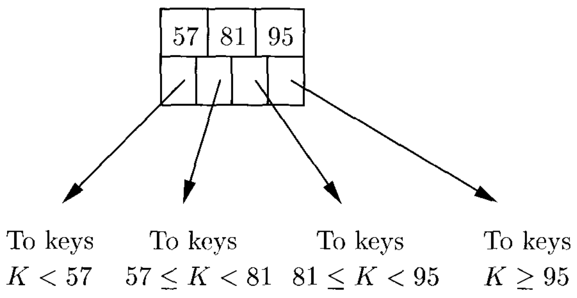

Figure 4.22: A typical interior node of a B+tree

Figure 4.22 shows a typical interior node. There are three keys; we have picked the same keys as in our leaf example: 57, 81, and 95.[4] There are also four pointers in this node. The first points to a part of the B-tree from which we can reach only records with keys less than 57, the first of the keys. The second pointer leads to records with keys between the first and second keys of the B-tree block, the third pointer for those records between the second and third keys of the block, and the fourth pointer lets us reach records with keys equal to or above the third key of the block.

As with our example leaf, it is not necessarily the case that all slots for keys and pointers are occupied. However, with $n = 3$, at least one key and two pointers must be present in an interior node. The most extreme case of missing elements would be if the only key were 57, and only the first two pointers were used. In that case, the first pointer would be to keys less than 57, and the second pointer would be to keys greater than or equal to 57. □

Example 4.21: Figure 4.23 shows a complete, three-level B+ tree,[5] using the nodes described in Example 4.20. We have assumed that the data file consists of records whose keys are all the primes from 2 to 47. Notice that at the leaves, each of these keys appears once, in order. All leaf blocks have two or three key-pointer pairs, plus a pointer to the next leaf in sequence. The keys are in sorted order as we look across the leaves from left to right.

The root has only two pointers, the minimum possible number, although it could have up to four. The one key at the root separates those keys reachable

[4] Although the keys are the same, the leaf of Fig. 4.21 and the interior node of Fig. 4.22 have no relationship. In fact, they could never appear in the same B-tree.

[5] Remember all B-trees discussed in this section are B+ trees, but we shall, in the future, omit the "+" when referring to them.

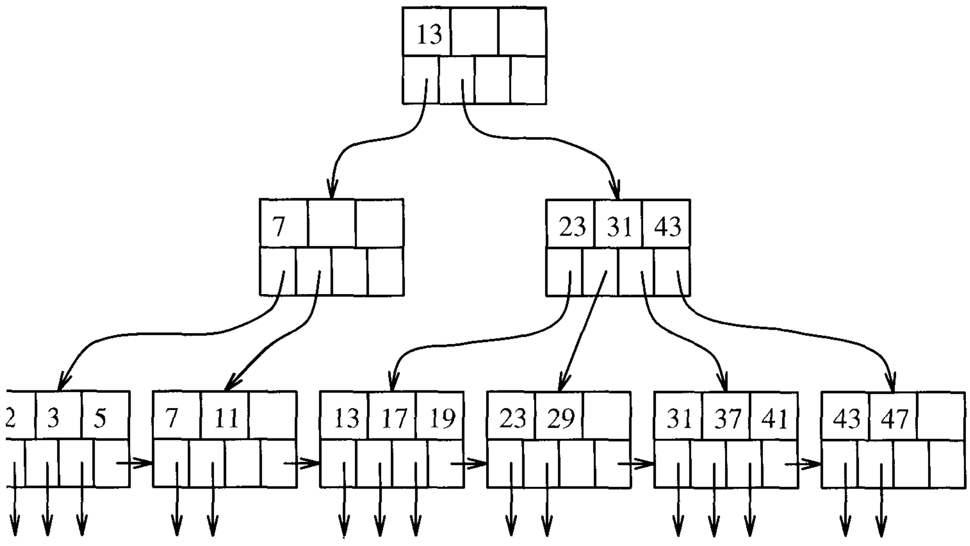

Figure 4.23: A B+ tree

via the first pointer from those reachable via the second. That is, keys up to 12 could be found in the first subtree of the root, and keys 13 and up are in the second subtree.

If we look at the first child of the root, with key 7, we again find two pointers, one to keys less than 7 and the other to keys 7 and above. Note that the second pointer in this node gets us only to keys 7 and 11, not to *all* keys > 7, such as 13 (although we could reach the larger keys by following the next-block pointers in the leaves).

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

4.3.2 Applications of B-trees

The B-tree is a powerful tool for building indexes. The sequence of pointers to records at the leaves can play the role of any of the pointer sequences coming out of an index file that we learned about in Sections 4.1 or 4.2. Here are some examples:

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

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

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

There are additional applications of B-tree variants that allow multiple occurrences of the search key[6] at the leaves. Figure 4.24 suggests what such a B-tree might look like. The extension is analogous to the indexes with duplicates that we discussed in Section 4.1.5.

If we do allow duplicate occurrences of a search key, then we need to change slightly the definition of what the keys at interior nodes mean, which we discussed in Section 4.3.1. Now, suppose there are keys $K_1, K_2, \ldots, K_n$ at an interior node. Then K_i will be the smallest new key that appears in the part of the subtree accessible from the $(i+1)$st pointer. By "new," we mean that there are no occurrences of K_i in the portion of the tree to the left of the $(i+1)$st subtree, but at least one occurrence of K_i in that subtree. Note that in some situations, there will be no such key, in which case K_i can be taken to be null. Its associated pointer is still necessary, as it points to a significant portion of the tree that happens to have only one key value within it.

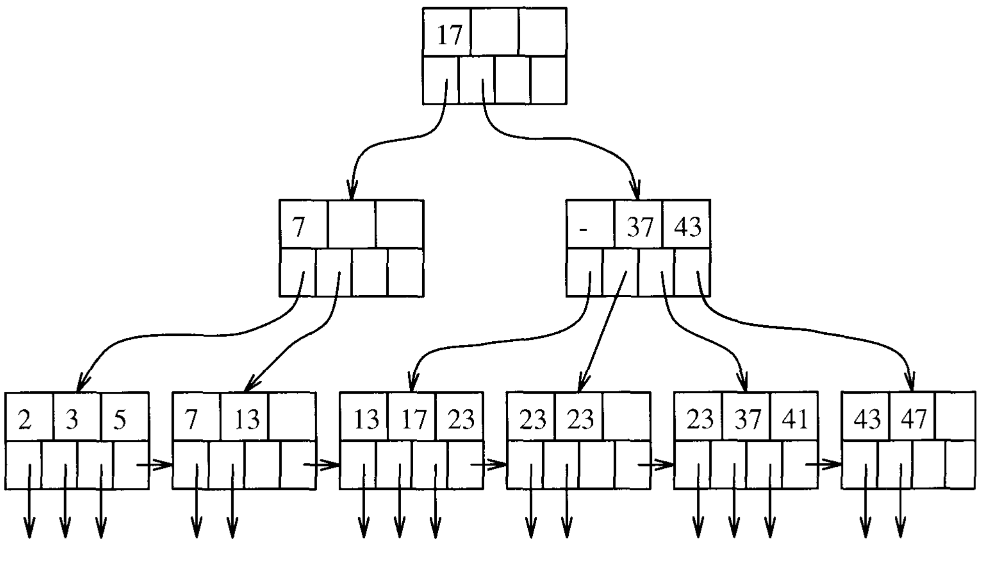

Figure 4.24: A B-tree with duplicate keys

Example 4.22: Figure 4.24 shows a B-tree similar to Fig. 4.23, but with duplicate values. In particular, key 11 has been replaced by 13, and keys 19,

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

29, and 31 have all been replaced by 23. As a result, the key at the root is 17, not 13. The reason is that, although 13 is the lowest key in the second subtree of the root, it is not a *new* key for that subtree, since it also appeals in the fust subtree.

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

- We would not reach the second child of the root searching for 13; we would be directed at the root to its first child instead.
- If we are looking for any key between 24 and 36, we are directed to the third leaf, but when we don't find even one occurrence of what we are looking for, we know not to search further right. For example, if there were a key 24 among the leaves, it would either be on the 4th leaf, in which case the null key in the second child of the root would be 24 instead, or it would be in the 5th leaf, in which case the key 37 at the second child of the root would be 24.

□

4.3.3 Lookup in B-Trees

We now revert to our original assumption that there are no duplicate keys at the leaves. This assumption makes the discussion of B-tree operations simpler, but is not essential for these operations. Suppose we have a B-tree index and we want to find a record with search-key value K. We search for K recursively, starting at the root and ending at a leaf. The search procedure is:

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

INDUCTION: If we are at an interior node with keys $K_1, K_2, \ldots, K_n$, follow the rules given in Section 4.3.1 to decide which of the children of this node should next be examined. That is, there is only one child that could lead to a leaf with key K. If $K < K_1$, then it is the first child, if $K_1 < K < K_2$, it is the second child, and so on. Recursively apply the search procedure at this child.

Example 4.23 : Suppose we have the B-tree of Fig. 4.23, and we want to find a record with search key 40. We start at the root, where there is one key, 13. Since 13 < 40, we follow the second pointer, which leads us to the second-level node with keys 23, 31, and 43.

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

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

4.3.4 Range Queries

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

```
SELECT *
FROM R
WHERE R.k > 40;
```

or

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

If we want to find all keys in the range [a, *b]* at the leaves of a B-tree, we do a lookup to find the key *a.* Whether or not it exists, we are led to a leaf where *a* could be, and we search the leaf for keys that are *a* or greater. Each such key we find has an associated pointer to one of the records whose key is in the desired range.

If we do not find a key higher than *b*, we use the pointer in the current leaf to the next leaf, and keep examining keys and following the associated pointers, until we either

1. Find a key higher than 6, at which point we stop, or

2. Reach the end of the leaf, in which case we go to the next leaf and repeat the process.

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

Example 4.24 : Suppose we have the B-tree of Fig. 4.23, and we are given the range $(10, 25)$ to search for. We look for key 10, which leads us to the second leaf. The first key is less than 10, but the second, 11, is at least 10. We follow its associated pointer to get the record with key 11.

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

4.3.5 Insertion Into B-Trees

We see some of the advantage of B-trees over the simpler multilevel indexes introduced in Section 4.1 4 when we consider how to insert a new key into a B-tree. The corresponding record will be inserted into the file being indexed by the B-tree, using any of the methods discussed in Section 4.1; here we consider how the B-tree changes in response. The insertion is in principle recursive:

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

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

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

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

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

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

Example 4.25: Let us insert key 40 into the B-tree of Fig. 4.23. We find the proper leaf for the insertion by the lookup procedure of Section 4.3.3. As found in Example 4.23, the insertion goes into the fifth leaf. Since *n* = 3, but this leaf now has four key-pointer pairs — 31, 37, 40, and 41 — we need to split the leaf. Our first step is to create a new node and move the highest two keys. 40 and 41, along with their pointers, to that node. Figure 4.25 shows this split.

Notice that although we now show the nodes on four ranks, there are still only three levels to the tree, and the seven leaves occupy the last two ranks of the diagram. They are linked by their last pointers, which still form a chain from left to right.

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

We start with pointers to the last five leaves and the list of keys representing the least keys of the last four of these leaves. That is, we have pointers P_1, P_2, P_3, P_4, P_5 to the leaves whose least keys are 13, 23, 31, 40, and 43, and we have the key sequence 23, 31, 40, 43 to separate these pointers. The first three pointers and first two keys remain with the split interior node, while the last two pointers and last key go to the new node. The remaining key, 40, represents the least key accessible via the new node.

Figure 4.26 shows the completion of the insert of key 40. The root now has three children; the last two are the split interior node. Notice that the key 40, which marks the lowest of the keys reachable via the second of the split nodes. has been installed in the root to separate the keys of the root's second and third children. □

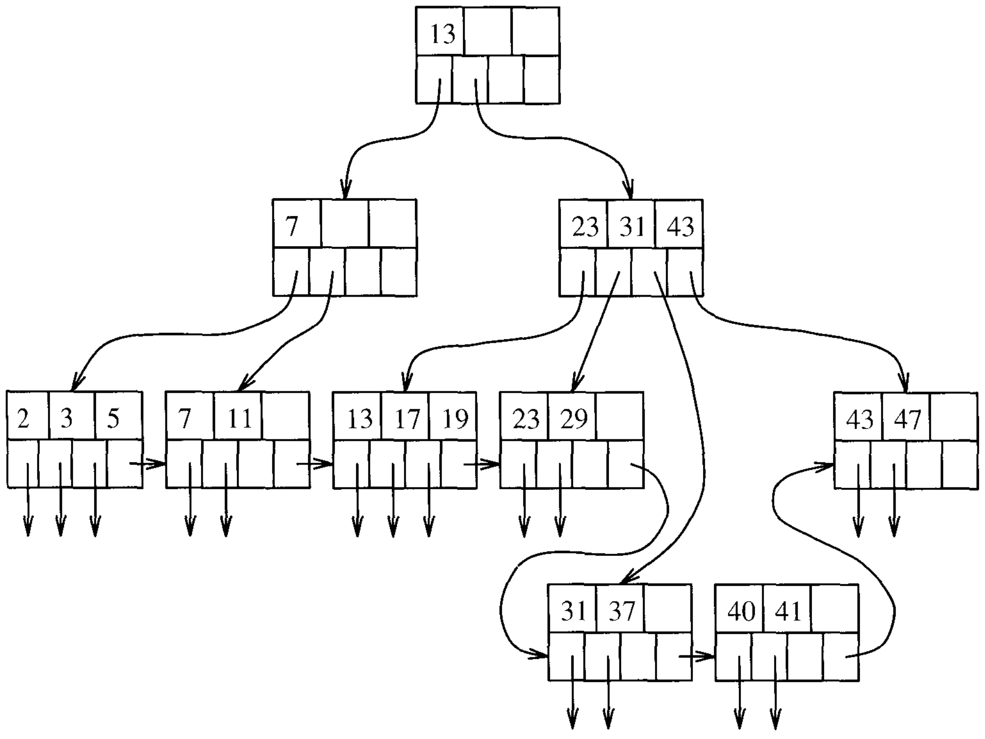

Figure 4.25: Beginning the insertion of key 40

4.3.6 Deletion From B-Trees

If we are to delete a record with a given key K, we must first locate that record and its key-pointer pair in a leaf of the B-tree. This part of the deletion process is essentially a lookup, as in Section 4.3.3. We then delete the record itself from the data file and we delete the key-pointer pair from the B-tree.

If the B-tree node from which a deletion occurred still has at least the minimum number of keys and pointers, then there is nothing more to be done.[7] However, it is possible that the node was right at the minimum occupancy before the deletion, so after deletion the constraint on the number of keys is violated. We then need to do one of two things for a node N whose contents are subminimum; one case requires a recursive deletion up the tree:

1. If one of the adjacent siblings of node N has more than the minimum number of keys and pointers, then one key-pointer pair can be moved to N, keeping the order of keys intact. Possibly, the keys at the parent of N

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

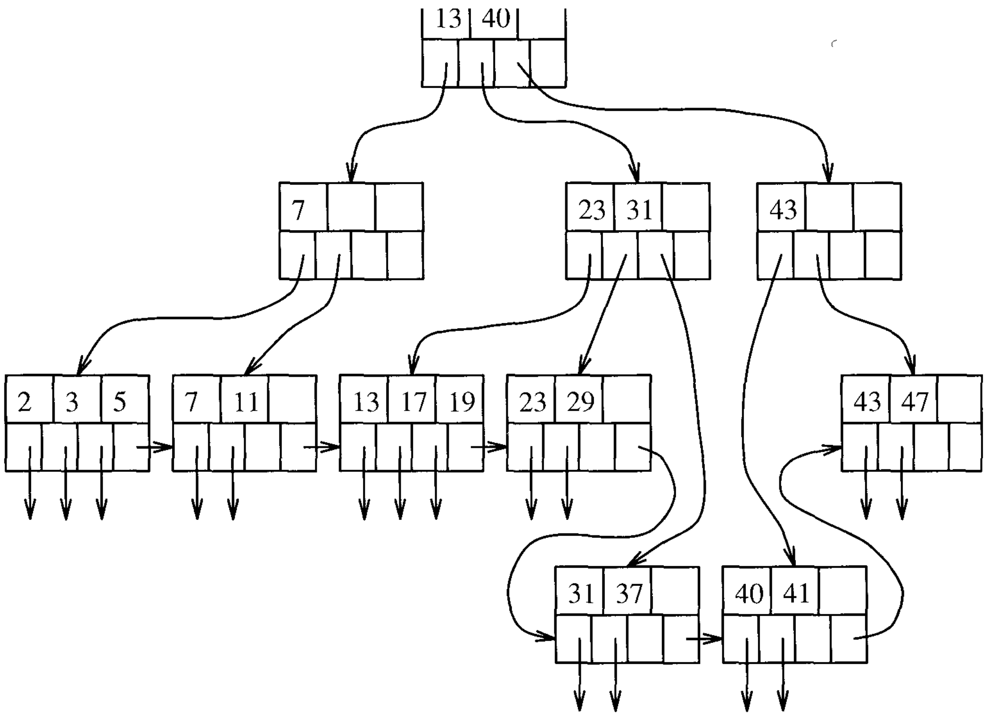

Figure 4.26: Completing the insertion of key 40

must be adjusted to reflect the new situation. For instance, if the right sibling of N, say node M, provides an extra key and pointer, then it must be the smallest key that is moved from M to N. At the parent of M and N, there is a key that represents the smallest key accessible via M; that key must be raised.

2. The hard case is when neither adjacent sibling can be used to provide an extra key for N. However, in that case, we have two adjacent nodes, N and one of its siblings M, one with the minimum number of keys and one with less than that. Therefore, together they have no more keys and pointers than are allowed in a single node (which is why half-full was chosen as the minimum allowable occupancy of B-tree nodes). We merge these two nodes, effectively deleting one of them. We need to adjust the keys at the parent, and then delete a key and pointer at the parent. If the parent is still full enough, then we are done. If not. then we recursively apply the deletion algorithm at the parent.

Example 4.26: Let us begin with the original B-tree of Fig. 4.23, before the insertion of key 40. Suppose we delete key 7. This key is found in the second leaf. We delete it, its associated pointer, and the record that pointer points to.

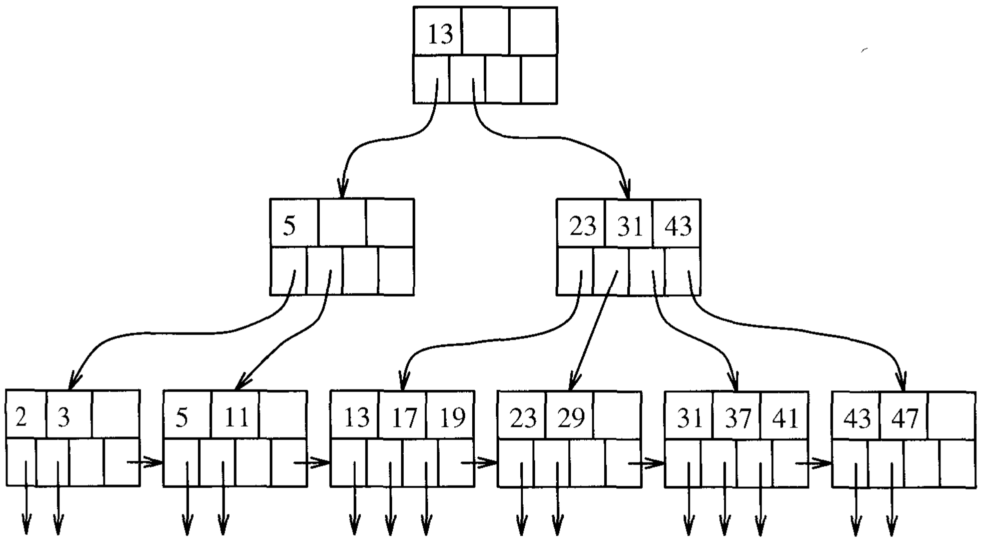

Figure 4.27: Deletion of key 7

Unfortunately, the second leaf now has only one key, and we need at least two in every leaf. But we are saved by the sibling to the left, the first loaf, because that leaf has an extra key-pointer pair. We may therefore move the highest key, 5, and its associated pointer to the second leaf. The resulting B-tree is shown in Fig. 4.27. Notice that because the lowest key in the second leaf is now 5, the key in the parent of the first two leaves has been changed from 7 to 5.

Next, suppose we delete key 11. This deletion has the same effect on the second leaf; it again reduces the number of its keys below the minimum. This time, however, we cannot borrow from the first leaf, because the latter is down to the minimum number of keys. Additionally, there is no sibling to the right from which to borrow.[8] Thus, we need to merge the second leaf with a sibling, namely the first leaf.

The three remaining key-pointer pairs from the first two leaves fit in one leaf, so we move 5 to the first leaf and delete the second leaf. The pointers and keys in the parent are adjusted to reflect the new situation at its children; specifically, the two pointers are replaced by one (to the remaining leaf) and the key 5 is no longer relevant and is deleted. The situation is now as shown in Fig.4.28.

Unfortunately, the deletion of a leaf has adversely affected the parent, which is the left child of the root. That node, as we see in Fig. 4.28, now has no keys

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

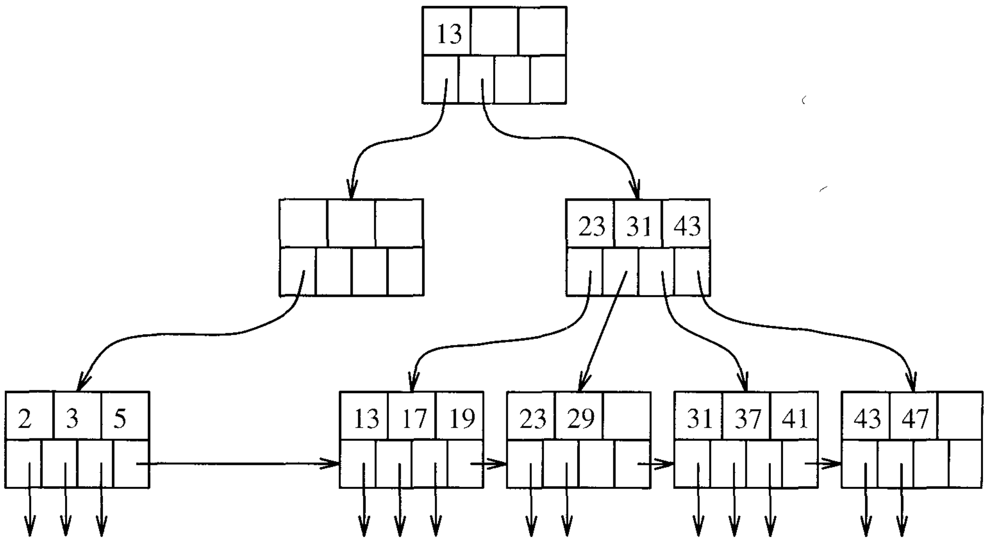

Figure 4.28: Beginning the deletion of key 11

and only one pointer. Thus, we try to obtain an extra key and pointer from an adjacent sibling. This time we have the easy case, since the other child of the root can afford to give up its smallest key and a pointer.

The change is shown in Fig. 4.29. The pointer to the leaf with keys 13, 17, and 19 has been moved from the second child of the root to the first child. We have also changed some keys at the interior nodes. The key 13, which used to reside at the root and represented the smallest key accessible via the pointer that was transferred, is now needed at the first child of the root. On the other hand, the key 23, which used to separate the first and second children of the second child of the root now represents the smallest key accessible from the second child of the root. It therefore is placed at the root itself. □

4.3.7 Efficiency of B-Trees

B-trees allow lookup, insertion, and deletion of records using very few disk I/O's per file operation. First, we should observe that if n, the number of keys per block is reasonably large, say 10 or more, then it will be a rare event that calls for splitting or merging of blocks. Further, when such an operation is needed, it almost always is limited to the leaves, so only two leaves and their parent are affected. Thus, we can essentially neglect the I/O cost of B-tree reorganizations.

However, every search for the record(s) with a given search key requires us to go from the root down to a leaf, to find a pointer to the record. Since we are only reading B-tree blocks, the number of disk I/0's will be the number of levels the B-tree has, plus the one (for lookup) or two (for insert or delete)

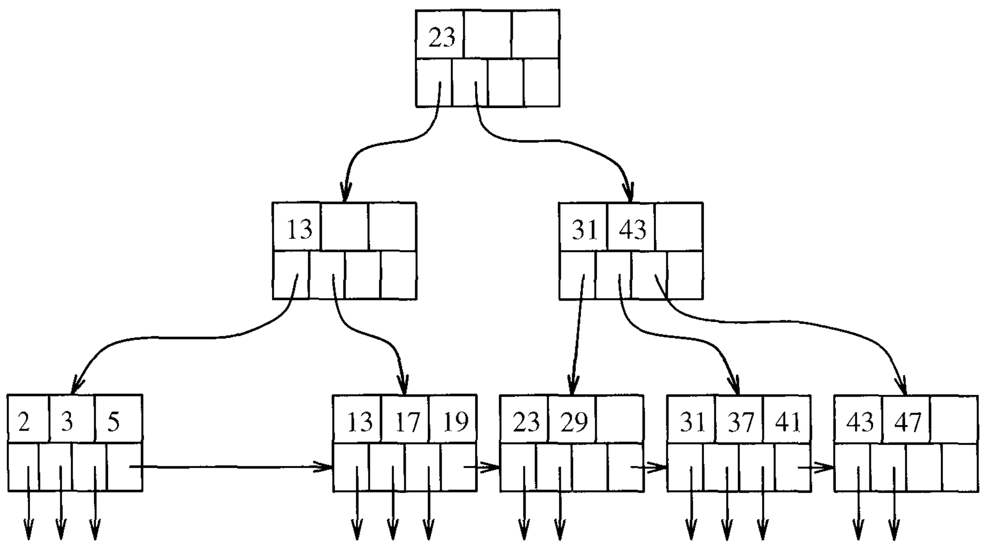

Figure 4.29: Completing the deletion of key 11

disk I/O's needed for manipulation of the record itself. We must thus ask: how many levels does a B-tree have? For the typical sizes of keys, pointers, and blocks, three levels are sufficient for all but the largest databases. Thus, we shall generally take 3 as the number of levels of a B-tree. The following example illustrates why.

Example 4.27: Recall our analysis in Example 4.19, where we determined that 340 key-pointer pairs could fit in one block for our example data. Suppose that the average block has an occupancy midway between the minimum and maximum, i.e., a typical block has 255 pointers. With a root, 255 children, and 255^2 = 65025 leaves, we shall have among those leaves 255^3, or about 16.6 million pointers to records. That is, files with up to 16.6 million records can be accommodated by a 3-level B-tree. □

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

4.3.8 Exercises for Section 4.3

Exercise **4.3.1:** Suppose that blocks can hold either ten records or 99 keys and 100 pointers. Also assume that the average B-tree node is 70% full; i.e., it

Should We Delete From B-Trees?

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

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

will have 69 keys and 70 pointers. We can use B-trees as part of several different structures. For each structure described below, determine (i) the total number of blocks needed for a 1,000,000-record file, and (ii) the average number of disk I/O's to retrieve a record given its search key. You may assume nothing is in memory initially, and the search key is the primary key for the records.

* a) The data file is a sequential file, sorted on the search key, with 10 records per block. The B-tree is a dense index.

b) The same as (a), but the data file consists of records in no particular order, packed 10 to a block.

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

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

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

Exercise 4.3.2: Repeat Exercise 4.3.1 in the case that the query is a range query that is matched by 1000 records.

Exercise 4.3.3 : Suppose pointers are 4 bytes long, and keys are 12 bytes long. How many keys and pointers will a block of 16,384 bytes have?

Exercise 4.3.4: What are the minimum numbers of keys and pointers in B-tree *(i)* interior nodes and *(ii)* leaves, when:

* a) $n = 10$; i.e., a block holds 10 keys and 11 pointers.

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

Exercise 4.3.5: Execute the following operations on Fig. 4.23. Describe the changes for operations that modify the tree.

a) Lookup the record with key 41.

b) Lookup the record with key 40.

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

d) Lookup all records with keys less than 30.

e) Lookup all records with keys greater than 30.

f) Insert a record with key 1.

g) Insert records with keys 14 through 16.

h) Delete the record with key 23.

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

! **Exercise 4.3.6**: We mentioned that the leaf of Fig. 4.21 and the interior node of Fig. 4.22 could never appear in the same B-tree. Explain why.

Exercise 4.3.7: When duplicate keys are allowed in a B-tree, there are some necessary modifications to the algorithms for lookup, insertion, and deletion that we described in this section. Give the changes for:

* a) Lookup.

b) Insertion.

c) Deletion.

! **Exercise 4.3.8:** In Example 4.26 we suggested that it would be possible to borrow keys from a nonsibling to the right (or left) if we used a more complicated algorithm for maintaining keys at interior nodes. Describe a suitable algorithm that rebalances by borrowing from adjacent nodes at a level, regardless of whether they are siblings of the node that has too many or too few key-pointer pairs.

Exercise 4.3.9: If we use the 3-key, 4-pointer nodes of our examples in this section, how many different B-trees are there when the data file has:

*! a) 6 records.

!! b) 10 records.

!! c) 15 records.

! Exercise 4.3.10: Suppose we have B-tree nodes with room for three keys and four pointers, as in the examples of this section. Suppose also that when we split a leaf, we divide the pointers 2 and 2, while when we split an interior node, the first 3 pointers go with the first (left) node, and the last 2 pointers go with the second (right) node. We start with a leaf containing pointers to records with keys 1, 2, and 3. We then add in order, records with keys 4, 5, 6, and so on. At the insertion of what key will the B-tree first reach four levels?

!! **Exercise 4.3.11:** Consider an index organized as a B+ tree. The leaf nodes contain pointers to a total of N records, and each block that makes up the index has m pointers. We wish to choose the value of m that will minimize search times on a particular disk with the following characteristics:

i. The time to lead a given block into memory can be approximated by $70 + .05m$ milliseconds. The 70 milliseconds represent the seek and latency components of the read, and the $.05m$ milliseconds is the transfer time. That is, as m becomes larger, the block will be larger, and it will take more time to read it into memory.

ii. Once the block is in memory, a binary search is used to find the correct pointer. Thus, the time to process a block in main memory is $a + b\log_2 m$ milliseconds, for some constants a and b.

iii. The main memory time constant a is much smaller than the diskseek and latency time of 70 milliseconds.

iv. The index is full, so that the number of blocks that must be examined per search is $\log_m N$.

Answer the following:

a) What value of m minimizes the time to search for a given record?

b) What happens as the seek and latency constant (70ms) decreases? For instance, if this constant is cut in half, how does the optimum m value change?

4.4 Hash Tables

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

4.4.1 Secondary-Storage Hash Tables

A hash table that holds a very large number of records, so many that they must be kept mainly in secondary storage, differs from the main-memory version in small but important ways. First, the bucket array consists of blocks, rather than pointers to the headers of lists. Records that are hashed by the hash function h to a certain bucket are put in the block for that bucket. If a bucket *overflows*, meaning that it cannot hold all the records that belong in that bucket, then a chain of *overflow blocks* can be added to the bucket to hold more records.

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

Example 4.28: Figure 4.30 shows a hash table. To keep our illustrations manageable, we assume that a block can hold only two records, and that $B = 4$; i.e., the hash function h returns values from 0 to 3. We show certain records populating the hash table. Keys are letters a through / in Fig. 4.30. We assume that $h(d) = 0$, $h(c) = h(e) = 1$, $h(b) = 2$, and $h(a) = h(f) = 3$. Thus, the six records are distributed into blocks as shown. □

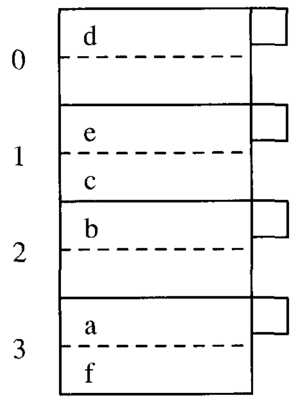

Figure 4.30: A hash table

Note that we show each block in Fig. 4.30 with a "nub" at the right end.

Choice of Hash Function

The hash function should "hash" the key so the resulting integer is a seemingly random function of the key. Thus, buckets will tend to have equal numbers of records, which improves the average time to access a record, as we shall discuss in Section 4.4.4. Also, the hash function should be easy to compute, since we shall compute it many times.

- A common choice of hash function when keys are integers is to compute the remainder of K/B, where K is the key value and B is the number of buckets. Often, B is chosen to be a prime, although there are reasons to make B a power of 2, as we discuss starting in Section 4.4.5.
- For character-string search keys, we may treat each character as an integer, surn these integers, and take the remainder when the sum is divided by B.

This nub represents additional information in the block's header. We shall use it to chain overflow blocks together, and starting in Section 4.4.5, we shall use it to keep other critical information about the block.

4.4.2 Insertion Into a Hash Table

When a new record with search key K must be inserted, we compute $h(K)$. If the bucket numbered $h(K)$ has space, then we insert the record into the block for this bucket, or into one of the overflow blocks on its chain if there is no room in the first block. If none of the blocks of the chain for bucket $h(K)$ has room, we add a new overflow block to the chain and store the new record there.

Example 4.29: Suppose we add to the hash table of Fig. 4.30 a record with key g, and $h(g) = 1$. Then we must add the new record to the bucket numbered 1, which is the second bucket from the top. However, the block for that bucket already has two records. Thus, we add a new block and chain it to the original block for bucket 1. The record with key g goes in that block, as shown in Fig. 4.31. □

4.4.3 Hash-Table Deletion

Deletion of the record (or records) with search key K follows the same pattern. We go to the bucket numbered $h(K)$ and search for records with that search key. Any that we find are deleted. If we are able to move records around among

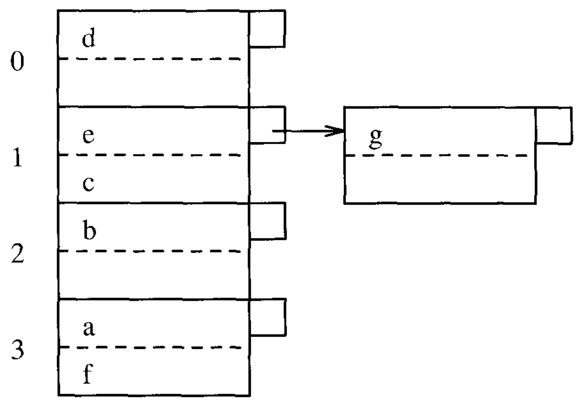

Figure 4.31: Adding an additional block to a hash-table bucket

blocks, then after deletion we may optionally consolidate the blocks of a chain into one fewer block.[9]

Example 4.30: Figure 4.32 shows the result of deleting the record with key *c* from the hash table of Fig. 4.31. Recall $h(c) = 1$, so we go to the bucket numbered 1 (i.e., the second bucket) and search all its blocks to find a record (or records if the search key were not the primary key) with key c. We find it in the first block of the chain for bucket 1. Since there is now room to move the record with key *g* from the second block of the chain to the first, we can do so and remove the second block.

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

4.4.4 Efficiency of Hash Table Indexes

Ideally, there are enough buckets that most of the buckets consist of a single block. If so, then the typical lookup takes only one disk I/O, and insertion or deletion from the file takes only two disk I/O's. That number is significantly better than straightforward sparse or dense indexes, or B-tree indexes (although hash tables do not support range queries as B-trees do; see Section 4.3.4).

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

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

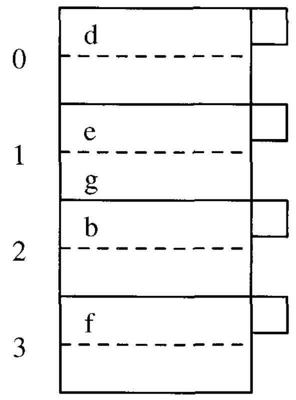

Figure 4.32: Result of deletions from a hash table

The hash tables we have examined so far are called *static hash tables,* because B, the number of buckets, never changes. However, there are several kinds of *dynamic hash tables,* where B is allowed to vary, so B approximates the number of records divided by the number of records that can fit on a block; i.e., there is about one block per bucket. We shall discuss two such methods:

1. Extensible hashing in Section 4.4.5, and
2. Linear hashing in Section 4.4.7.

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

4.4.5 Extensible Hash Tables

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

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

smaller number of bits, say i bits, from the beginning of this sequence. That is, the bucket array will have 2^i entries when i is the number of bits used.

Example 4.31: Figure 4.33 shows a small extensible hash table. We suppose, for simplicity of the example, that $k = 4$; i.e., the hash function produces a sequence of only four bits. At the moment, only one of these bits is used, as indicated by $i = 1$ in the box above the bucket array. The bucket array therefore has only two entries, one for 0 and one for 1.

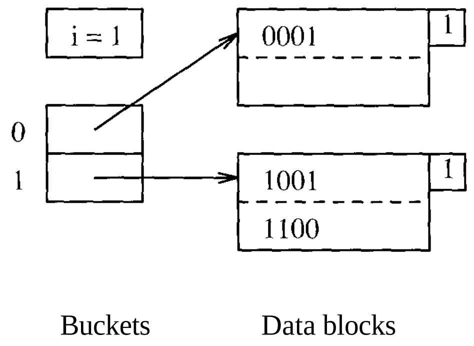

Figure 4.33: An extensible hash table

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

We should notice the number 1 appearing in the "nub" of each of the blocks in Fig. 4.33. This number, which would actually appear in the block header, indicates how many bits of the hash function's sequence is used to determine membership of records in this block. In the situation of Example 4.31, there is only one bit considered for all blocks and records, but as we shall see, the number of bits considered for various blocks can differ as the hash table grows. That is, the bucket array size is determined by the maximum number of bits we are now using, but some blocks may use fewer.

4.4.6 Insertion Into Extensible Hash Tables

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

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

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

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

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

Example 4.32: Suppose we insert into the table of Fig. 4.33 a record whose key hashes to the sequence 1010. Since the first bit is 1, this record belongs in the second block. However, that block is already full, so it needs to be split. We find that $j = i = 1$ in this case, so we first need to double the bucket array, as shown in Fig. 4.34. We have also set $i = 2$ in this figure.

Notice that the two entries beginning with 0 each point to the block for records whose hashed keys begin with 0, and that block still has the integer 1 in its "nub" to indicate that only the first bit determines membership in the block. However, the block for records beginning with 1 needs to be split, so we

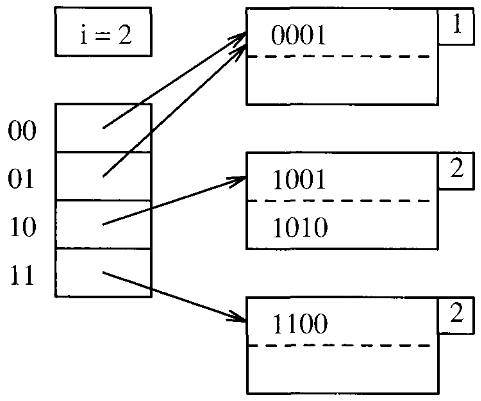

Figure 4.34: Now, two bits of the hash function are used

partition its records into those beginning 10 and those beginning 11. A 2 in each of these blocks indicates that two bits are used to determine membership. Fortunately, the split is successful; since each of the two new blocks gets at least one record, we do not have to split recursively.

Now suppose we insert records whose keys hash to 0000 and 0111. These both go in the first block of Fig. 4.34, which then overflows. Since only one bit is used to determine membership in this block, while $i = 2$, we do not have to adjust the bucket array. We simply split the block, with 0000 and 0001 staying, and 0111 going to the new block. The entry for 01 in the bucket array is made to point to the new block. Again, we have been fortunate that the records did not all go in one of the new blocks, so we have no need to split recursively.

Now suppose a record whose key hashes to 1000 is inserted. The block for 10 overflows. Since it already uses two bits to determine membership, it is time to split the bucket array again and set $i = 3$. Figure 4.35 shows the data structure at this point. Notice that the block for 10 has been split into blocks for 100 and 101, while the other blocks continue to use only two bits to determine membership. □

4.4.7 Linear Hash Tables

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

1. When the bucket array needs to be doubled in size, there is a substantial amount of work to be done (when i is large). This work interrupts access

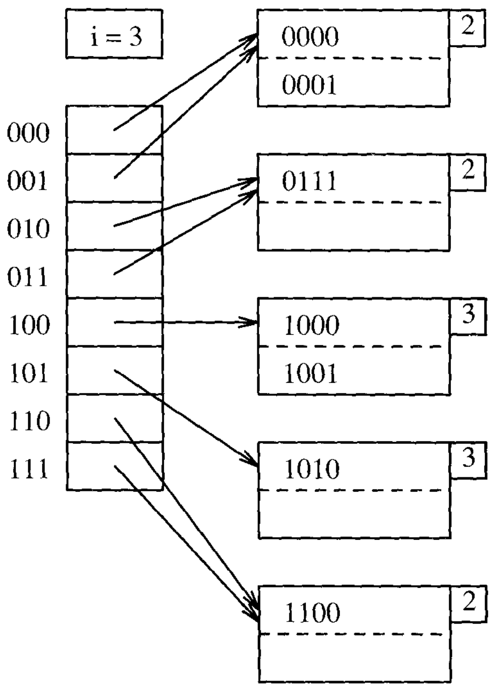

Figure 4.35: The hash table now uses three bits of the hash function

to the data file, or makes certain insertions appear to take a very large amount of time.

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

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

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

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

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

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

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

Example 4.33: Figure 4.36 shows a linear hash table with $n = 2$. We currently are using only one bit of the hash value to determine the buckets of records. Following the pattern established in Example 4.31, we assume the hash function *h* produces 4 bits, and we represent records by the value produced by *h* when applied to the search key of the record.

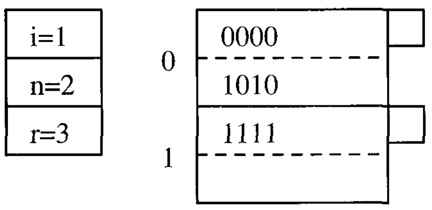

Figure 4.36: A linear hash table

We see in Fig. 4.36 the two buckets, each consisting of one block. The buckets are numbered 0 and 1. All records whose hash value ends in 0 go in the first bucket, and those whose hash value ends in 1 go in the second.

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

4.4.8 Insertion Into Linear Hash Tables

When we insert a new record, we determine its bucket by the algorithm outlined in Section 4.4.7. That is, we compute $h(K)$, where K is the key of the record, and determine the correct number of bits at the end of bit sequence $h(K)$ to use as the bucket number. We put the record either in that bucket, or (if the bucket number is n or greater) in the bucket with the leading bit changed from 1 to 0. If there is no room in the bucket, then we create an overflow block, add it to the chain for that bucket, and put the record there.

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

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

Example 4.34: We shall continue with Example 4.33 and consider what happens when a record whose key hashes to 0101 is inserted. Since this bit sequence ends in 1, the record goes into the second bucket of Fig. 4.36. There is room for the record, so no overflow block is created.

However, since there are now 4 records in 2 buckets, we exceed the ratio 1.7, and we must therefore raise n to 3. Since $\lceil \log_2 3 \rceil = 2$, we should begin to think of buckets 0 and 1 as 00 and 01, but no change to the data structure is necessary. We add to the table the next bucket, which would have number 10. Then, we split the bucket 00, that bucket whose number differs from the added bucket only in the first bit. When we do the split, the record whose key hashes to 0000 stays in 00. since it ends with 00, while the record whose key hashes to 1010 goes to 10 because it ends that way. The resulting hash table is shown in Fig. 4.37.

Next, let us suppose we add a record whose search key hashes to 0001. The last two bits are 01, so we put it in this bucket, which currently exists. Unfortunately, the bucket's block is full, so we add an overflow block. The three records are distributed among the two blocks of the bucket; we chose to keep them in numerical order of their hashed keys, but order is not important. Since the ratio of records to buckets for the table as a whole is 5/3, and this ratio is less than 1.7, we do not create a new bucket. The result is seen in Fig. 4.38.

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

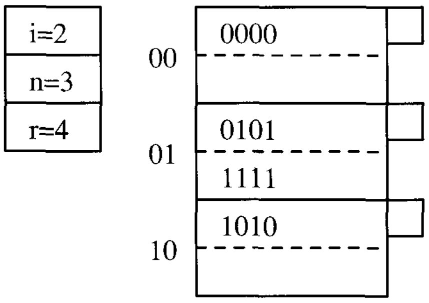

Figure 4.37: Adding a third bucket

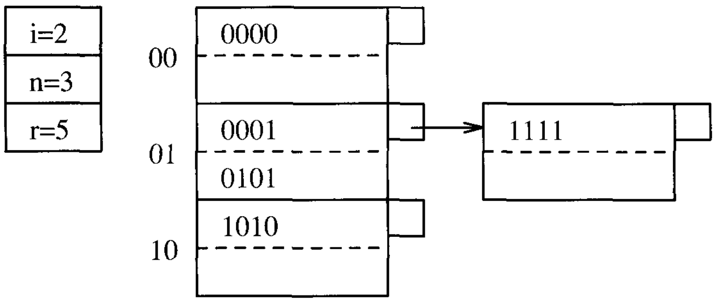

Figure 4.38: Overflow blocks are used if necessary

However, the ratio of the number of records to buckets has exceeded 1.7, so we must create a new bucket, numbered 11. Coincidentally, this bucket is the one we wanted for the new record. We split the four records in bucket 01, with 0001 and 0101 remaining, and 0111 and 1111 going to the new bucket. Since bucket 01 now has only two records, we can delete the overflow block. The hash table is now as shown in Fig. 4.39.

Notice that the next time we insert a record into Fig. 4.39, we shall exceed the 1.7 ratio of records to buckets. Then, we shall raise n to 5 and i becomes 3. □

Example 4.35: Lookup in a linear hash table follows the procedure we described for selecting the bucket in which an inserted record belongs. If the record we wish to look up is not in that bucket, it cannot be anywhere. For illustration, consider the situation of Fig. 4.37, where we have $i = 2$ and $n = 3$.

First, suppose we want to look up a record whose key hashes to 1010. Since $i = 2$, we look at the last two bits, 10, which we interpret as a binary integer, namely $m = 2$. Since $m < n$, the bucket numbered 10 exists, and we look there. Notice that just because we find a record with hash value 1010 doesn't mean that this record is the one we want; we need to check the complete key of that record to be sure.

Second, consider the lookup of a record whose key hashes to 1011. Now, we

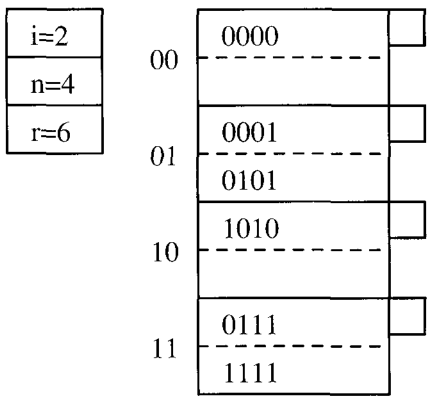

Figure 4.39: Adding a fourth bucket

must look in the bucket whose number is 11. Since that bucket number as a binary integer is $m = 3$, and $m > n$, the bucket 11 does not exist. We redirect to bucket 01 by changing the leading 1 to 0. However, bucket 01 has no record whose key has hash value 1011, and therefore surely our desired record is not in the hash table. □

4.4.9 Exercises for Section 4.4

Exercise 4.4.1: Show what happens to the buckets in Fig. 4.30 if the following insertions and deletions occur:

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

ii. Records a and b are deleted.

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

iv. Records c and d are deleted.

Exercise 4.4.2 : We did not discuss how deletions can be carried out in a linear or extensible hash table. The mechanics of locating the record(s) to be deleted should be obvious. What method would you suggest for executing the deletion? In particular, what are the advantages and disadvantages of restructuring the table if its smaller size after deletion allows for compression of certain blocks?

! Exercise **4.4.3:** The material of this section assumes that search keys are unique. However, only small modifications are needed to allow the techniques to work for search keys with duplicates. Describe the necessary changes to insertion, deletion, and lookup algorithms, and suggest the major problems that arise when there are duplicates in:

* a) A simple hash table.

b) An extensible hash table.

c) A linear hash table.

! Exercise **4.4.4:** Some hash functions do not work as well as theoretically possible. Suppose that we use the hash function on integer keys i defined by $h(i) = i^2 \bmod B$.

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

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

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

Exercise **4.4.5:** In an extensible hash table with n records per block, what is the probability that an overflowing block will have to be handled recursively; i.e., all members of the block will go into the same one of the two blocks created in the split?

Exercise **4.4.6:** Suppose keys arc hashed to four-bit sequences, as in our examples of extensible and linear hashing in this section. However, also suppose that blocks can hold three records, rather than the two-record blocks of our examples. If we start with a hash table with two empty blocks (corresponding to 0 and 1), show the organization after we insert records with keys:

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

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

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

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

* Exercise **4.4.7:** Suppose we use a linear or extensible hashing scheme, but there are pointers to records from outside. These pointers prevent us from moving records between blocks, as is sometimes required by these hashing methods. Suggest several ways that we could modify the structure to allow pointers from outside.

!! Exercise 4.4.8 : A linear-hashing scheme with blocks that hold k records uses a threshold constant c, such that the current number of buckets n and the current number of records r are related by $r = ckn$. For instance, in Example 4.33 we used $k = 2$ and $c = 0.85$, so there were 1.7 records per bucket; i.e., $r = 1.7n$.

a) Suppose for convenience that each key occurs exactly its expected number of times.[10] As a function of c, k, and n, how many blocks, including overflow blocks, are needed for the structure?

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

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

*! **Exercise 4.4.9:** Suppose we have a file of 1,000,000 records that we want to hash into a table with 1000 buckets. 100 records will fit in a block, and we wish to keep blocks as full as possible, but not allow two buckets to share a block. What are the minimum and maximum number of blocks that we could need to store this hash table?

4.5 Summary of Chapter 4

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

✦ *Dense Indexes:* These indexes have a key-pointer pair for every record in the data file. The pairs are kept in sorted order of their key values.

✦ *Sparse Indexes:* These indexes have one key-pointer pair for each block of the data file. The key associated with a pointer to a block is the first key found on that block.

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

✦ *Expanding Files:* As a data file and its index file(s) grow, some provision for adding additional blocks to the file must be made. Adding overflow blocks to the original blocks is one possibility. Inserting additional blocks in the sequence for the data or index file may be possible, unless the file itself is required to be in sequential blocks of the disk.

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

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

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

✦ *Range Queries*: Queries in which we ask for all records whose search-key value lies in a given range are facilitated by indexed sequential files and B-tree indexes, although not by hash-table indexes.

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

✦ *Dynamic Hashing*: Since performance of a hash table degrades if there are too many records in one bucket, the number of buckets may need to grow as time goes on. Two important methods of allowing graceful growth are extensible and linear hashing. Both begin by hashing search-key values to long bit-strings and use a varying number of those bits to determine the bucket for records.

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

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

4.6 References for Chapter 4

The B-tree was the original idea of Bayer and McCreight [2]. Unlike the B+ tree described here, this formulation had pointers to records at the interior nodes as well as at the leaves. [3] is a survey of B-tree varieties.

Hashing as a data structure goes back to Peterson [8]. Extensible hashing was developed by [4], while linear hashing is from [7]. The book by Knuth [6] contains much information on data structures, including techniques for selecting hash functions and designing hash tables, as well as a number of ideas concerning B-tree variants. The B+ tree formulation (without key values at interior nodes) appeared in the 1973 edition of [6].

Secondary indexes and other techniques for retrieval of documents are covered by [9]. Also, [5] and [1] are surveys of index methods for text documents.

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

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

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

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

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

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

7. W. Litwin, "Linear hashing: a new tool for file and table addressing," *Proc. Intl. Conf. on Very Large Databases* (1980) pp. 212-223.

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

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