© The McGraw–Hill Companies, 2001

CHAPTER 12

Indexing and Hashing

Many queries reference only a small proportion of the records in a file. For example, a query like "Find all accounts at the Perryridge branch" or "Find the balance of account number A-101" references only a fraction of the account records. It is inefficient for the system to read every record and to check the *branch-name* field for the name "Perryridge," or the *account-number* field for the value A-101. Ideally, the system should be able to locate these records directly. To allow these forms of access, we design additional structures that we associate with files.

12.1 Basic Concepts

An index for a file in a database system works in much the same way as the index in this textbook. If we want to learn about a particular topic (specified by a word or a phrase) in this textbook, we can search for the topic in the index at the back of the book, find the pages where it occurs, and then read the pages to find the information we are looking for. The words in the index are in sorted order, making it easy to find the word we are looking for. Moreover, the index is much smaller than the book, further reducing the effort needed to find the words we are looking for.

Card catalogs in libraries worked in a similar manner (although they are rarely used any longer). To find a book by a particular author, we would search in the author catalog, and a card in the catalog tells us where to find the book. To assist us in searching the catalog, the library would keep the cards in alphabetic order by authors, with one card for each author of each book.

Database system indices play the same role as book indices or card catalogs in libraries. For example, to retrieve an *account* record given the account number, the database system would look up an index to find on which disk block the corresponding record resides, and then fetch the disk block, to get the *account* record.

Keeping a sorted list of account numbers would not work well on very large databases with millions of accounts, since the index would itself be very big; further,

even though keeping the index sorted reduces the search time, finding an account can still be rather time-consuming. Instead, more sophisticated indexing techniques may be used. We shall discuss several of these techniques in this chapter.

There are two basic kinds of indices:

- **Ordered indices**. Based on a sorted ordering of the values.
- **Hash indices**. Based on a uniform distribution of values across a range of buckets. The bucket to which a value is assigned is determined by a function, called a *hash function*.

We shall consider several techniques for both ordered indexing and hashing. No one technique is the best. Rather, each technique is best suited to particular database applications. Each technique must be evaluated on the basis of these factors:

- **Access types**: The types of access that are supported efficiently. Access types can include finding records with a specified attribute value and finding records whose attribute values fall in a specified range.
- **Access time**: The time it takes to find a particular data item, or set of items, using the technique in question.
- **Insertion time**: The time it takes to insert a new data item. This value includes the time it takes to find the correct place to insert the new data item, as well as the time it takes to update the index structure.
- **Deletion time**: The time it takes to delete a data item. This value includes the time it takes to find the item to be deleted, as well as the time it takes to update the index structure.
- **Space overhead**: The additional space occupied by an index structure. Provided that the amount of additional space is moderate, it is usually worthwhile to sacrifice the space to achieve improved performance.

We often want to have more than one index for a file. For example, libraries maintained several card catalogs: for author, for subject, and for title.

An attribute or set of attributes used to look up records in a file is called a **search key**. Note that this definition of *key* differs from that used in *primary key*, *candidate key*, and *superkey*. This duplicate meaning for *key* is (unfortunately) well established in practice. Using our notion of a search key, we see that if there are several indices on a file, there are several search keys.

12.2 Ordered Indices

To gain fast random access to records in a file, we can use an index structure. Each index structure is associated with a particular search key. Just like the index of a book or a library catalog, an ordered index stores the values of the search keys in sorted order, and associates with each search key the records that contain it.

The records in the indexed file may themselves be stored in some sorted order, just as books in a library are stored according to some attribute such as the Dewey deci-

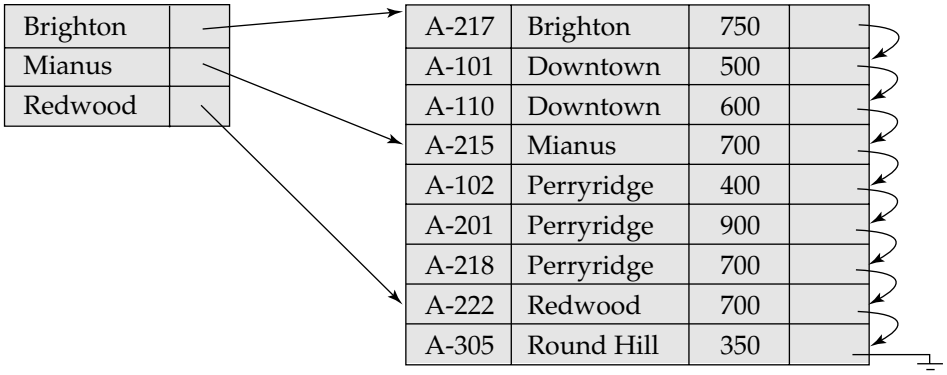


Figure 12.1 Sequential file for *account* records.

mal number. A file may have several indices, on different search keys. If the file containing the records is sequentially ordered, a **primary index** is an index whose search key also defines the sequential order of the file. (The term *primary index* is sometimes used to mean an index on a primary key. However, such usage is nonstandard and should be avoided.) Primary indices are also called **clustering indices**. The search key of a primary index is usually the primary key, although that is not necessarily so. Indices whose search key specifies an order different from the sequential order of the file are called **secondary indices**, or **nonclustering** indices.

12.2.1 Primary Index

In this section, we assume that all files are ordered sequentially on some search key. Such files, with a primary index on the search key, are called **index-sequential files**. They represent one of the oldest index schemes used in database systems. They are designed for applications that require both sequential processing of the entire file and random access to individual records.

Figure 12.1 shows a sequential file of *account* records taken from our banking example. In the example of Figure 12.1, the records are stored in search-key order, with *branch-name* used as the search key.

12.2.1.1 Dense and Sparse Indices

An **index record**, or **index entry**, consists of a search-key value, and pointers to one or more records with that value as their search-key value. The pointer to a record consists of the identifier of a disk block and an offset within the disk block to identify the record within the block.

There are two types of ordered indices that we can use:

- **Dense index**: An index record appears for every search-key value in the file. In a dense primary index, the index record contains the search-key value and a pointer to the first data record with that search-key value. The rest of the records with the same search key-value would be stored sequentially after the

first record, since, because the index is a primary one, records are sorted on the same search key.

Dense index implementations may store a list of pointers to all records with the same search-key value; doing so is not essential for primary indices.

- **Sparse index**: An index record appears for only some of the search-key values. As is true in dense indices, each index record contains a search-key value and a pointer to the first data record with that search-key value. To locate a record, we find the index entry with the largest search-key value that is less than or equal to the search-key value for which we are looking. We start at the record pointed to by that index entry, and follow the pointers in the file until we find the desired record.

Figures 12.2 and 12.3 show dense and sparse indices, respectively, for the *account* file. Suppose that we are looking up records for the Perryridge branch. Using the dense index of Figure 12.2, we follow the pointer directly to the first Perryridge record. We process this record, and follow the pointer in that record to locate the next record in search-key (*branch-name*) order. We continue processing records until we encounter a record for a branch other than Perryridge. If we are using the sparse index (Figure 12.3), we do not find an index entry for "Perryridge." Since the last entry (in alphabetic order) before "Perryridge" is "Mianus," we follow that pointer. We then read the *account* file in sequential order until we find the first Perryridge record, and begin processing at that point.

As we have seen, it is generally faster to locate a record if we have a dense index rather than a sparse index. However, sparse indices have advantages over dense indices in that they require less space and they impose less maintenance overhead for insertions and deletions.

There is a trade-off that the system designer must make between access time and space overhead. Although the decision regarding this trade-off depends on the specific application, a good compromise is to have a sparse index with one index entry per block. The reason this design is a good trade-off is that the dominant cost in pro-

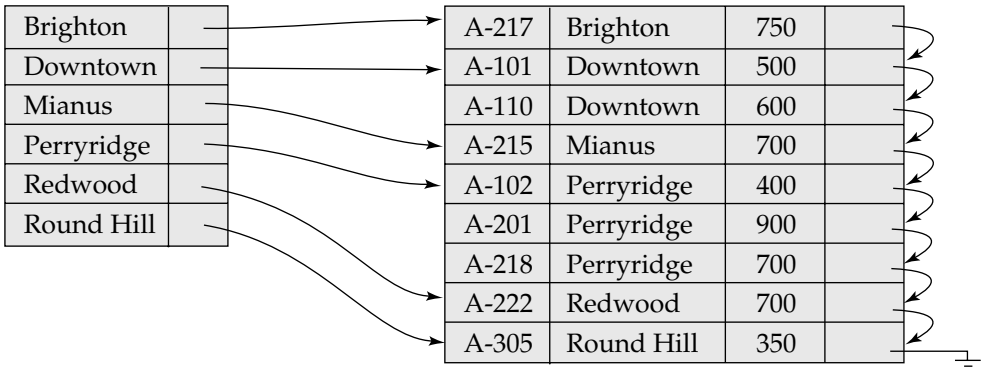


Figure 12.2 Dense index.

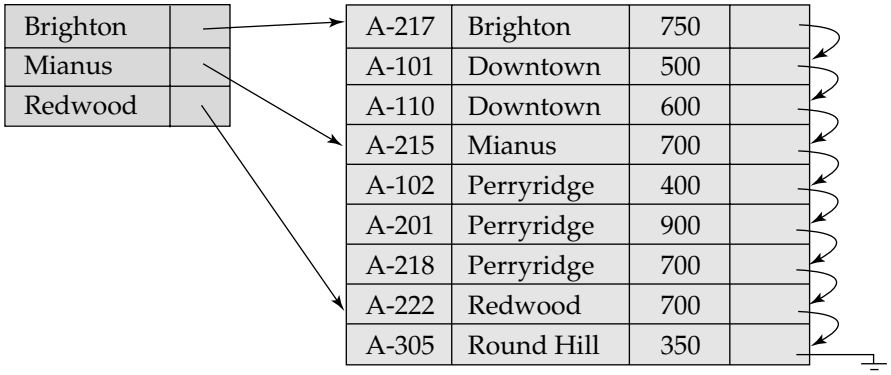


Figure 12.3 Sparse index.

cessing a database request is the time that it takes to bring a block from disk into main memory. Once we have brought in the block, the time to scan the entire block is negligible. Using this sparse index, we locate the block containing the record that we are seeking. Thus, unless the record is on an overflow block (see Section 11.7.1), we minimize block accesses while keeping the size of the index (and thus, our space overhead) as small as possible.

For the preceding technique to be fully general, we must consider the case where records for one search-key value occupy several blocks. It is easy to modify our scheme to handle this situation.

12.2.1.2 Multilevel Indices

Even if we use a sparse index, the index itself may become too large for efficient processing. It is not unreasonable, in practice, to have a file with 100,000 records, with 10 records stored in each block. If we have one index record per block, the index has 10,000 records. Index records are smaller than data records, so let us assume that 100 index records fit on a block. Thus, our index occupies 100 blocks. Such large indices are stored as sequential files on disk.

If an index is sufficiently small to be kept in main memory, the search time to find an entry is low. However, if the index is so large that it must be kept on disk, a search for an entry requires several disk block reads. Binary search can be used on the index file to locate an entry, but the search still has a large cost. If the index occupies b blocks, binary search requires as many as $\lceil \log_2(b) \rceil$ blocks to be read. ($\lceil x \rceil$ denotes the least integer that is greater than or equal to x; that is, we round upward.) For our 100-block index, binary search requires seven block reads. On a disk system where a block read takes 30 milliseconds, the search will take 210 milliseconds, which is long. Note that, if overflow blocks have been used, binary search will not be possible. In that case, a sequential search is typically used, and that requires b block reads, which will take even longer. Thus, the process of searching a large index may be costly.

To deal with this problem, we treat the index just as we would treat any other sequential file, and construct a sparse index on the primary index, as in Figure 12.4.

To locate a record, we first use binary search on the outer index to find the record for the largest search-key value less than or equal to the one that we desire. The pointer points to a block of the inner index. We scan this block until we find the record that has the largest search-key value less than or equal to the one that we desire. The pointer in this record points to the block of the file that contains the record for which we are looking.

Using the two levels of indexing, we have read only one index block, rather than the seven we read with binary search, if we assume that the outer index is already in main memory. If our file is extremely large, even the outer index may grow too large to fit in main memory. In such a case, we can create yet another level of index. Indeed, we can repeat this process as many times as necessary. Indices with two or more levels are called **multilevel** indices. Searching for records with a multilevel index requires significantly fewer I/O operations than does searching for records by binary search. Each level of index could correspond to a unit of physical storage. Thus, we may have indices at the track, cylinder, and disk levels.

A typical dictionary is an example of a multilevel index in the nondatabase world. The header of each page lists the first word alphabetically on that page. Such a book

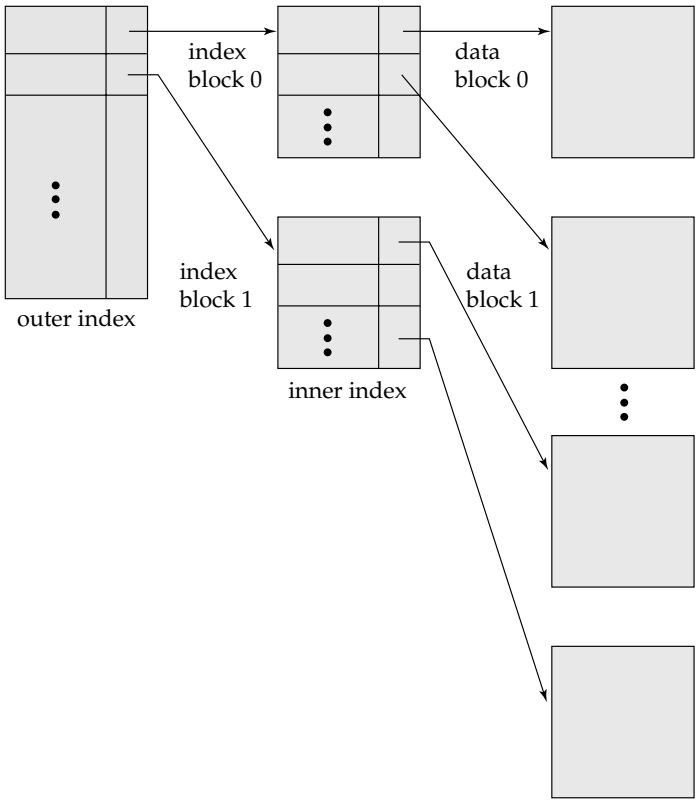


Figure 12.4 Two-level sparse index.

index is a multilevel index: The words at the top of each page of the book index form a sparse index on the contents of the dictionary pages.

Multilevel indices are closely related to tree structures, such as the binary trees used for in-memory indexing. We shall examine the relationship later, in Section 12.3.

12.2.1.3 Index Update

Regardless of what form of index is used, every index must be updated whenever a record is either inserted into or deleted from the file. We first describe algorithms for updating single-level indices.

- **Insertion**. First, the system performs a lookup using the search-key value that appears in the record to be inserted. Again, the actions the system takes next depend on whether the index is dense or sparse:
  - Dense indices:
    1. If the search-key value does not appear in the index, the system inserts an index record with the search-key value in the index at the appropriate position.
    2. Otherwise the following actions are taken:
       a. If the index record stores pointers to all records with the same search-key value, the system adds a pointer to the new record to the index record.
       b. Otherwise, the index record stores a pointer to only the first record with the search-key value. The system then places the record being inserted after the other records with the same search-key values.
  - Sparse indices: We assume that the index stores an entry for each block. If the system creates a new block, it inserts the first search-key value (in search-key order) appearing in the new block into the index. On the other hand, if the new record has the least search-key value in its block, the system updates the index entry pointing to the block; if not, the system makes no change to the index.
- **Deletion**. To delete a record, the system first looks up the record to be deleted. The actions the system takes next depend on whether the index is dense or sparse:
  - Dense indices:
    1. If the deleted record was the only record with its particular search-key value, then the system deletes the corresponding index record from the index.
    2. Otherwise the following actions are taken:
       a. If the index record stores pointers to all records with the same search-key value, the system deletes the pointer to the deleted record from the index record.

b. Otherwise, the index record stores a pointer to only the first record with the search-key value. In this case, if the deleted record was the first record with the search-key value, the system updates the index record to point to the next record.

- Sparse indices:
  1. If the index does not contain an index record with the search-key value of the deleted record, nothing needs to be done to the index.
  2. Otherwise the system takes the following actions:
     a. If the deleted record was the only record with its search key, the system replaces the corresponding index record with an index record for the next search-key value (in search-key order). If the next search-key value already has an index entry, the entry is deleted instead of being replaced.
     b. Otherwise, if the index record for the search-key value points to the record being deleted, the system updates the index record to point to the next record with the same search-key value.

Insertion and deletion algorithms for multilevel indices are a simple extension of the scheme just described. On deletion or insertion, the system updates the lowest-level index as described. As far as the second level is concerned, the lowest-level index is merely a file containing records—thus, if there is any change in the lowest-level index, the system updates the second-level index as described. The same technique applies to further levels of the index, if there are any.

12.2.2 Secondary Indices

Secondary indices must be dense, with an index entry for every search-key value, and a pointer to every record in the file. A primary index may be sparse, storing only some of the search-key values, since it is always possible to find records with intermediate search-key values by a sequential access to a part of the file, as described earlier. If a secondary index stores only some of the search-key values, records with intermediate search-key values may be anywhere in the file and, in general, we cannot find them without searching the entire file.

A secondary index on a candidate key looks just like a dense primary index, except that the records pointed to by successive values in the index are not stored sequentially. In general, however, secondary indices may have a different structure from primary indices. If the search key of a primary index is not a candidate key, it suffices if the index points to the first record with a particular value for the search key, since the other records can be fetched by a sequential scan of the file.

In contrast, if the search key of a secondary index is not a candidate key, it is not enough to point to just the first record with each search-key value. The remaining records with the same search-key value could be anywhere in the file, since the records are ordered by the search key of the primary index, rather than by the search key of the secondary index. Therefore, a secondary index must contain pointers to all the records.

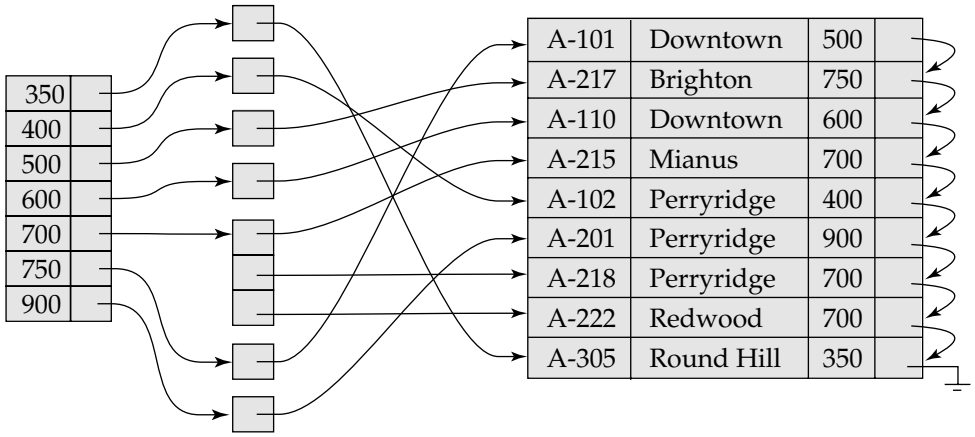


Figure 12.5 Secondary index on *account* file, on noncandidate key *balance*.

We can use an extra level of indirection to implement secondary indices on search keys that are not candidate keys. The pointers in such a secondary index do not point directly to the file. Instead, each points to a bucket that contains pointers to the file. Figure 12.5 shows the structure of a secondary index that uses an extra level of indirection on the *account* file, on the search key *balance*.

A sequential scan in primary index order is efficient because records in the file are stored physically in the same order as the index order. However, we cannot (except in rare special cases) store a file physically ordered both by the search key of the primary index, and the search key of a secondary index. Because secondary-key order and physical-key order differ, if we attempt to scan the file sequentially in secondary-key order, the reading of each record is likely to require the reading of a new block from disk, which is very slow.

The procedure described earlier for deletion and insertion can also be applied to secondary indices; the actions taken are those described for dense indices storing a pointer to every record in the file. If a file has multiple indices, whenever the file is modified, *every* index must be updated.

Secondary indices improve the performance of queries that use keys other than the search key of the primary index. However, they impose a significant overhead on modification of the database. The designer of a database decides which secondary indices are desirable on the basis of an estimate of the relative frequency of queries and modifications.

12.3 B+-Tree Index Files

The main disadvantage of the index-sequential file organization is that performance degrades as the file grows, both for index lookups and for sequential scans through the data. Although this degradation can be remedied by reorganization of the file, frequent reorganizations are undesirable.

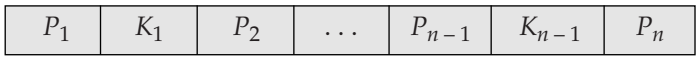


Figure 12.6 Typical node of a B$^+$-tree.

The **B$^+$-tree** index structure is the most widely used of several index structures that maintain their efficiency despite insertion and deletion of data. A B$^+$-tree index takes the form of a **balanced tree** in which every path from the root of the tree to a leaf of the tree is of the same length. Each nonleaf node in the tree has between $\lceil n/2 \rceil$ and n children, where n is fixed for a particular tree.

We shall see that the B$^+$-tree structure imposes performance overhead on insertion and deletion, and adds space overhead. The overhead is acceptable even for frequently modified files, since the cost of file reorganization is avoided. Furthermore, since nodes may be as much as half empty (if they have the minimum number of children), there is some wasted space. This space overhead, too, is acceptable given the performance benefits of the B$^+$-tree structure.

12.3.1 Structure of a B$^+$-Tree

A B$^+$-tree index is a multilevel index, but it has a structure that differs from that of the multilevel index-sequential file. Figure 12.6 shows a typical node of a B$^+$-tree. It contains up to $n - 1$ search-key values $K_1, K_2, \ldots, K_{n-1}$, and n pointers $P_1, P_2, \ldots, P_n$. The search-key values within a node are kept in sorted order; thus, if $i < j$, then $K_i < K_j$.

We consider first the structure of the leaf nodes. For $i = 1, 2, \ldots, n - 1$, pointer P_i points to either a file record with search-key value K_i or to a bucket of pointers, each of which points to a file record with search-key value K_i. The bucket structure is used only if the search key does not form a primary key, and if the file is not sorted in the search-key value order. Pointer P_n has a special purpose that we shall discuss shortly.

Figure 12.7 shows one leaf node of a B$^+$-tree for the *account* file, in which we have chosen n to be 3, and the search key is *branch-name*. Note that, since the account file is ordered by *branch-name*, the pointers in the leaf node point directly to the file.

Now that we have seen the structure of a leaf node, let us consider how search-key values are assigned to particular nodes. Each leaf can hold up to $n - 1$ values. We allow leaf nodes to contain as few as $\lceil (n-1)/2 \rceil$ values. The ranges of values in each leaf do not overlap. Thus, if L_i and L_j are leaf nodes and $i < j$, then every search-key value in L_i is less than every search-key value in L_j. If the B$^+$-tree index is to be a dense index, every search-key value must appear in some leaf node.

Now we can explain the use of the pointer P_n. Since there is a linear order on the leaves based on the search-key values that they contain, we use P_n to chain together the leaf nodes in search-key order. This ordering allows for efficient sequential processing of the file.

The nonleaf nodes of the B$^+$-tree form a multilevel (sparse) index on the leaf nodes. The structure of nonleaf nodes is the same as that for leaf nodes, except that all pointers are pointers to tree nodes. A nonleaf node may hold up to n pointers, and *must*

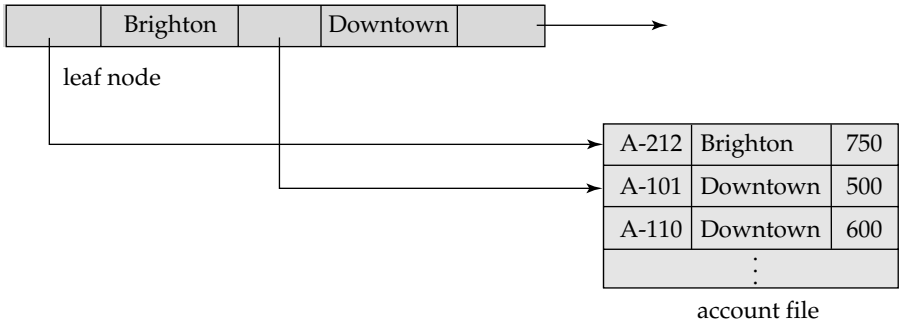


Figure 12.7 A leaf node for *account* B+-tree index ($n = 3$).

hold at least $\lceil n/2 \rceil$ pointers. The number of pointers in a node is called the *fanout* of the node.

Let us consider a node containing m pointers. For $i = 2, 3, \ldots, m - 1$, pointer P_i points to the subtree that contains search-key values less than K_i and greater than or equal to K_{i-1}. Pointer P_m points to the part of the subtree that contains those key values greater than or equal to K_{m-1}, and pointer P_1 points to the part of the subtree that contains those search-key values less than K_1.

Unlike other nonleaf nodes, the root node can hold fewer than $\lceil n/2 \rceil$ pointers; however, it must hold at least two pointers, unless the tree consists of only one node. It is always possible to construct a B+-tree, for any n, that satisfies the preceding requirements. Figure 12.8 shows a complete B+-tree for the *account* file ($n = 3$). For simplicity, we have omitted both the pointers to the file itself and the null pointers. As an example of a B+-tree for which the root must have less than $\lceil n/2 \rceil$ values, Figure 12.9 shows a B+-tree for the *account* file with $n = 5$.

These examples of B+-trees are all balanced. That is, the length of every path from the root to a leaf node is the same. This property is a requirement for a B+-tree. Indeed, the "B" in B+-tree stands for "balanced." It is the balance property of B+-trees that ensures good performance for lookup, insertion, and deletion.

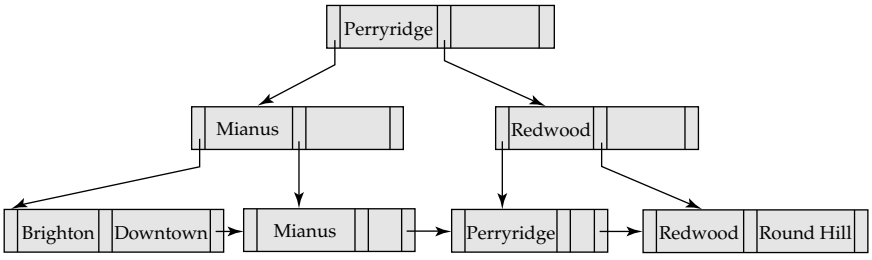


Figure 12.8 B+-tree for *account* file ($n = 3$).

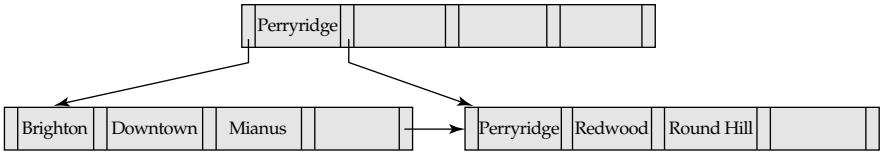


Figure 12.9 B$^+$-tree for *account* file with $n = 5$.

12.3.2 Queries on B$^+$-Trees

Let us consider how we process queries on a B$^+$-tree. Suppose that we wish to find all records with a search-key value of V. Figure 12.10 presents pseudocode for doing so. Intuitively, the procedure works as follows. First, we examine the root node, looking for the smallest search-key value greater than V. Suppose that we find that this search-key value is K_i. We then follow pointer P_i to another node. If we find no such value, then $k \geq K_{m-1}$, where m is the number of pointers in the node. In this case we follow P_m to another node. In the node we reached above, again we look for the smallest search-key value greater than V, and once again follow the corresponding pointer as above. Eventually, we reach a leaf node. At the leaf node, if we find search-key value K_i equals V, then pointer P_i directs us to the desired record or bucket. If the value V is not found in the leaf node, no record with key value V exists.

Thus, in processing a query, we traverse a path in the tree from the root to some leaf node. If there are K search-key values in the file, the path is no longer than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.

In practice, only a few nodes need to be accessed, Typically, a node is made to be the same size as a disk block, which is typically 4 kilobytes. With a search-key size of 12 bytes, and a disk-pointer size of 8 bytes, n is around 200. Even with a more conservative estimate of 32 bytes for the search-key size, n is around 100. With $n = 100$, if we have 1 million search-key values in the file, a lookup requires only

```
procedure find(value V)
    set C = root node
    while C is not a leaf node begin
        Let K_i = smallest search-key value, if any, greater than V
        if there is no such value then begin
            Let m = the number of pointers in the node
            set C = node pointed to by P_m
        end
            else set C = the node pointed to by P_i
    end
    if there is a key value K_i in C such that K_i = V
        then pointer P_i directs us to the desired record or bucket
        else no record with key value k exists
end procedure
```

Figure 12.10 Querying a B$^+$-tree.

$\lceil \log_{50}(1{,}000{,}000) \rceil = 4$ nodes to be accessed. Thus, at most four blocks need to be read from disk for the lookup. The root node of the tree is usually heavily accessed and is likely to be in the buffer, so typically only three or fewer blocks need to be read from disk.

An important difference between B$^+$-tree structures and in-memory tree structures, such as binary trees, is the size of a node, and as a result, the height of the tree. In a binary tree, each node is small, and has at most two pointers. In a B$^+$-tree, each node is large—typically a disk block—and a node can have a large number of pointers. Thus, B$^+$-trees tend to be fat and short, unlike thin and tall binary trees. In a balanced binary tree, the path for a lookup can be of length $\lceil \log_2(K) \rceil$, where K is the number of search-key values. With $K = 1{,}000{,}000$ as in the previous example, a balanced binary tree requires around 20 node accesses. If each node were on a different disk block, 20 block reads would be required to process a lookup, in contrast to the four block reads for the B$^+$-tree.

12.3.3 Updates on B$^+$-Trees

Insertion and deletion are more complicated than lookup, since it may be necessary to **split** a node that becomes too large as the result of an insertion, or to **coalesce** nodes (that is, combine nodes) if a node becomes too small (fewer than $\lceil n/2 \rceil$ pointers). Furthermore, when a node is split or a pair of nodes is combined, we must ensure that balance is preserved. To introduce the idea behind insertion and deletion in a B$^+$-tree, we shall assume temporarily that nodes never become too large or too small. Under this assumption, insertion and deletion are performed as defined next.

- **Insertion**. Using the same technique as for lookup, we find the leaf node in which the search-key value would appear. If the search-key value already appears in the leaf node, we add the new record to the file and, if necessary, add to the bucket a pointer to the record. If the search-key value does not appear, we insert the value in the leaf node, and position it such that the search keys are still in order. We then insert the new record in the file and, if necessary, create a new bucket with the appropriate pointer.
- **Deletion**. Using the same technique as for lookup, we find the record to be deleted, and remove it from the file. We remove the search-key value from the leaf node if there is no bucket associated with that search-key value or if the bucket becomes empty as a result of the deletion.

We now consider an example in which a node must be split. Assume that we wish to insert a record with a *branch-name* value of "Clearview" into the B$^+$-tree of Figure 12.8. Using the algorithm for lookup, we find that "Clearview" should appear in the node containing "Brighton" and "Downtown." There is no room to insert the search-key value "Clearview." Therefore, the node is *split* into two nodes. Figure 12.11 shows the two leaf nodes that result from inserting "Clearview" and splitting the node containing "Brighton" and "Downtown." In general, we take the n search-key

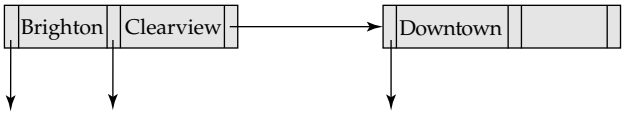


Figure 12.11 Split of leaf node on insertion of "Clearview."

values (the $n - 1$ values in the leaf node plus the value being inserted), and put the first $\lceil n/2 \rceil$ in the existing node and the remaining values in a new node.

Having split a leaf node, we must insert the new leaf node into the B$^+$-tree structure. In our example, the new node has "Downtown" as its smallest search-key value. We need to insert this search-key value into the parent of the leaf node that was split. The B$^+$-tree of Figure 12.12 shows the result of the insertion. The search-key value "Downtown" was inserted into the parent. It was possible to perform this insertion because there was room for an added search-key value. If there were no room, the parent would have had to be split. In the worst case, all nodes along the path to the root must be split. If the root itself is split, the entire tree becomes deeper.

The general technique for insertion into a B$^+$-tree is to determine the leaf node l into which insertion must occur. If a split results, insert the new node into the parent of node l. If this insertion causes a split, proceed recursively up the tree until either an insertion does not cause a split or a new root is created.

Figure 12.13 outlines the insertion algorithm in pseudocode. In the pseudocode, $L.K_i$ and $L.P_i$ denote the ith value and the ith pointer in node L, respectively. The pseudocode also makes use of the function $parent(L)$ to find the parent of a node L. We can compute a list of nodes in the path from the root to the leaf while initially finding the leaf node, and can use it later to find the parent of any node in the path efficiently. The pseudocode refers to inserting an entry (V, P) into a node. In the case of leaf nodes, the pointer to an entry actually precedes the key value, so the leaf node actually stores P before V. For internal nodes, P is stored just after V.

We now consider deletions that cause tree nodes to contain too few pointers. First, let us delete "Downtown" from the B$^+$-tree of Figure 12.12. We locate the entry for "Downtown" by using our lookup algorithm. When we delete the entry for "Downtown" from its leaf node, the leaf becomes empty. Since, in our example, $n = 3$ and $0 < \lceil (n-1)/2 \rceil$, this node must be eliminated from the B$^+$-tree. To delete a leaf node,

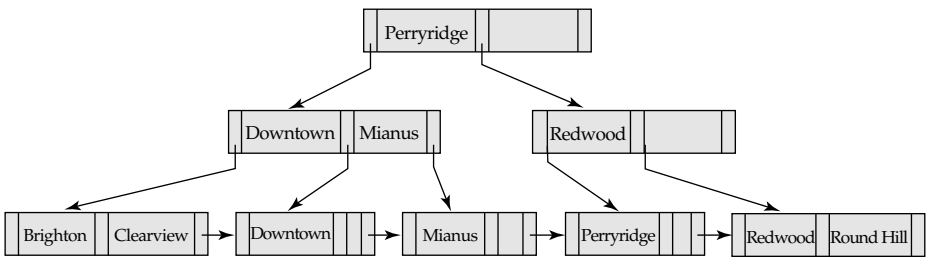


Figure 12.12 Insertion of "Clearview" into the B$^+$-tree of Figure 12.8.

```
procedure insert(value V, pointer P)
    find the leaf node L that should contain value V
    insert_entry(L, V, P)
end procedure

procedure insert_entry(node L, value V, pointer P)
    if (L has space for (V, P))
        then insert (V, P) in L
        else begin /* Split L */
            Create node L'
            Let V' be the value in L.K_1, ..., L.K_{n-1}, V such that exactly
                ⌈n/2⌉ of the values L.K_1, ..., L.K_{n-1}, V are less than V'
            Let m be the lowest value such that L.K_m ≥ V'
            /* Note: V' must be either L.K_m or V */
            if (L is a leaf) then begin
                move L.P_m, L.K_m, ..., L.P_{n-1}, L.K_{n-1} to L'
                if (V < V') then insert (P, V) in L
                else insert (P, V) in L'
            end
            else begin
                if (V = V') /* V is smallest value to go to L' */
                    then add P, L.K_m, ..., L.P_{n-1}, L.K_{n-1}, L.P_n to L'
                    else add L.P_m, ..., L.P_{n-1}, L.K_{n-1}, L.P_n to L'
                delete L.K_m, ..., L.P_{n-1}, L.K_{n-1}, L.P_n from L
                if (V < V') then insert (V, P) in L
                else if (V > V') then insert (V, P) in L'
                /* Case of V = V' handled already */
            end
            if (L is not the root of the tree)
                then insert_entry(parent(L), V', L');
                else begin
                    create a new node R with child nodes L and L' and
                        the single value V'
                    make R the root of the tree
                end
            if (L) is a leaf node then begin /* Fix next child pointers */
                set L'.P_n = L.P_n;
                set L.P_n = L'
            end
        end
end procedure
```

Figure 12.13 Insertion of entry in a B$^+$-tree.

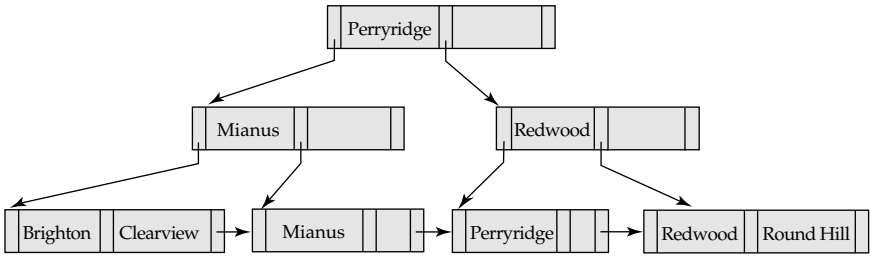


Figure 12.14 Deletion of "Downtown" from the B$^+$-tree of Figure 12.12.

we must delete the pointer to it from its parent. In our example, this deletion leaves the parent node, which formerly contained three pointers, with only two pointers. Since $2 \geq \lceil n/2 \rceil$, the node is still sufficiently large, and the deletion operation is complete. The resulting B$^+$-tree appears in Figure 12.14.

When we make a deletion from a parent of a leaf node, the parent node itself may become too small. That is exactly what happens if we delete "Perryridge" from the B$^+$-tree of Figure 12.14. Deletion of the Perryridge entry causes a leaf node to become empty. When we delete the pointer to this node in the latter's parent, the parent is left with only one pointer. Since $n = 3$, $\lceil n/2 \rceil = 2$, and thus only one pointer is too few. However, since the parent node contains useful information, we cannot simply delete it. Instead, we look at the sibling node (the nonleaf node containing the one search key, Mianus). This sibling node has room to accommodate the information contained in our now-too-small node, so we coalesce these nodes, such that the sibling node now contains the keys "Mianus" and "Redwood." The other node (the node containing only the search key "Redwood") now contains redundant information and can be deleted from its parent (which happens to be the root in our example). Figure 12.15 shows the result. Notice that the root has only one child pointer after the deletion, so it is deleted and its sole child becomes the root. So the depth of the B$^+$-tree has been decreased by 1.

It is not always possible to coalesce nodes. As an illustration, delete "Perryridge" from the B$^+$-tree of Figure 12.12. In this example, the "Downtown" entry is still part of the tree. Once again, the leaf node containing "Perryridge" becomes empty. The parent of the leaf node becomes too small (only one pointer). However, in this example, the sibling node already contains the maximum number of pointers: three. Thus, it cannot accommodate an additional pointer. The solution in this case is to **redistribute** the pointers such that each sibling has two pointers. The result appears in

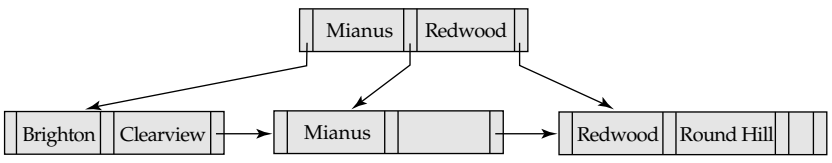


Figure 12.15 Deletion of "Perryridge" from the B$^+$-tree of Figure 12.14.

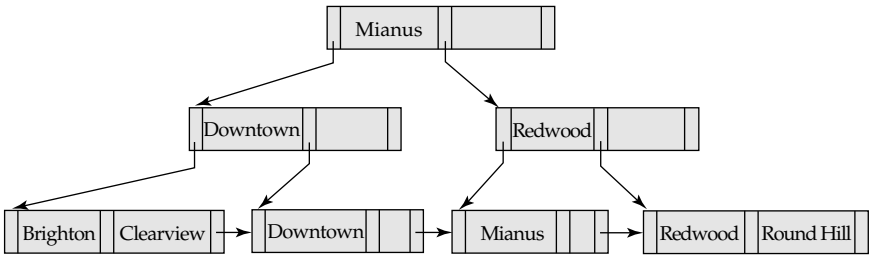


Figure 12.16 Deletion of "Perryridge" from the B+-tree of Figure 12.12.

Figure 12.16. Note that the redistribution of values necessitates a change of a search-key value in the parent of the two siblings.

In general, to delete a value in a B+-tree, we perform a lookup on the value and delete it. If the node is too small, we delete it from its parent. This deletion results in recursive application of the deletion algorithm until the root is reached, a parent remains adequately full after deletion, or redistribution is applied.

Figure 12.17 outlines the pseudocode for deletion from a B+-tree. The procedure swap_variables(L, L') merely swaps the values of the (pointer) variables L and L'; this swap has no effect on the tree itself. The pseudocode uses the condition "too few pointers/values." For nonleaf nodes, this criterion means less than $\lceil n/2 \rceil$ pointers; for leaf nodes, it means less than $\lceil (n-1)/2 \rceil$ values. The pseudocode redistributes entries by borrowing a single entry from an adjacent node. We can also redistribute entries by repartitioning entries equally between the two nodes. The pseudocode refers to deleting an entry (V, P) from a node. In the case of leaf nodes, the pointer to an entry actually precedes the key value, so the pointer P precedes the key value V. For internal nodes, P follows the key value V.

It is worth noting that, as a result of deletion, a key value that is present in an internal node of the B+-tree may not be present at any leaf of the tree.

Although insertion and deletion operations on B+-trees are complicated, they require relatively few I/O operations, which is an important benefit since I/O operations are expensive. It can be shown that the number of I/O operations needed for a worst-case insertion or deletion is proportional to $\log_{\lceil n/2 \rceil}(K)$, where n is the maximum number of pointers in a node, and K is the number of search-key values. In other words, the cost of insertion and deletion operations is proportional to the height of the B+-tree, and is therefore low. It is the speed of operation on B+-trees that makes them a frequently used index structure in database implementations.

12.3.4 B+-Tree File Organization

As mentioned in Section 12.3, the main drawback of index-sequential file organization is the degradation of performance as the file grows: With growth, an increasing percentage of index records and actual records become out of order, and are stored in overflow blocks. We solve the degradation of index lookups by using B+-tree indices on the file. We solve the degradation problem for storing the actual records by using the leaf level of the B+-tree to organize the blocks containing the actual records. We

```
procedure delete(value V, pointer P)
    find the leaf node L that contains (V, P)
    delete_entry(L, V, P)
end procedure
procedure delete_entry(node L, value V, pointer P)
    delete (V, P) from L
    if (L is the root and L has only one remaining child)
    then make the child of L the new root of the tree and delete L
    else if (L has too few values/pointers) then begin
        Let L' be the previous or next child of parent(L)
        Let V' be the value between pointers L and L' in parent(L)
        if (entries in L and L' can fit in a single node)
            then begin /* Coalesce nodes */
                if (L is a predecessor of L') then swap_variables(L, L')
                if (L is not a leaf)
                    then append V' and all pointers and values in L to L'
                    else append all (K_i, P_i) pairs in L to L'; set L'.P_n = L.P_n
                delete_entry(parent(L), V', L); delete node L
            end
        else begin /* Redistribution: borrow an entry from L' */
            if (L' is a predecessor of L) then begin
                if (L is a non-leaf node) then begin
                    let m be such that L'.P_m is the last pointer in L'
                    remove (L'.K_{m-1}, L'.P_m) from L'
                    insert (L'.P_m, V') as the first pointer and value in L,
                        by shifting other pointers and values right
                    replace V' in parent(L) by L'.K_{m-1}
                end
                else begin
                    let m be such that (L'.P_m, L'.K_m) is the last pointer/value
                        pair in L'
                    remove (L'.P_m, L'.K_m) from L'
                    insert (L'.P_m, L'.K_m) as the first pointer and value in L,
                        by shifting other pointers and values right
                    replace V' in parent(L) by L'.K_m
                end
            end
            else ... symmetric to the then case ...
        end
    end
end procedure
```

Figure 12.17 Deletion of entry from a B^+-tree.

use the B+-tree structure not only as an index, but also as an organizer for records in a file. In a **B+-tree file organization**, the leaf nodes of the tree store records, instead of storing pointers to records. Figure 12.18 shows an example of a B+-tree file organization. Since records are usually larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a nonleaf node. However, the leaf nodes are still required to be at least half full.

Insertion and deletion of records from a B+-tree file organization are handled in the same way as insertion and deletion of entries in a B+-tree index. When a record with a given key value v is inserted, the system locates the block that should contain the record by searching the B+-tree for the largest key in the tree that is $\leq v$. If the block located has enough free space for the record, the system stores the record in the block. Otherwise, as in B+-tree insertion, the system splits the block in two, and redistributes the records in it (in the B+-tree–key order) to create space for the new record. The split propagates up the B+-tree in the normal fashion. When we delete a record, the system first removes it from the block containing it. If a block B becomes less than half full as a result, the records in B are redistributed with the records in an adjacent block B'. Assuming fixed-sized records, each block will hold at least one-half as many records as the maximum that it can hold. The system updates the nonleaf nodes of the B+-tree in the usual fashion.

When we use a B+-tree for file organization, space utilization is particularly important, since the space occupied by the records is likely to be much more than the space occupied by keys and pointers. We can improve the utilization of space in a B+-tree by involving more sibling nodes in redistribution during splits and merges. The technique is applicable to both leaf nodes and internal nodes, and works as follows.

During insertion, if a node is full the system attempts to redistribute some of its entries to one of the adjacent nodes, to make space for a new entry. If this attempt fails because the adjacent nodes are themselves full, the system splits the node, and splits the entries evenly among one of the adjacent nodes and the two nodes that it obtained by splitting the original node. Since the three nodes together contain one more record than can fit in two nodes, each node will be about two-thirds full. More precisely, each node will have at least $\lfloor 2n/3 \rfloor$ entries, where n is the maximum number of entries that the node can hold. ($\lfloor x \rfloor$ denotes the greatest integer that is less than or equal to x; that is, we drop the fractional part, if any.)

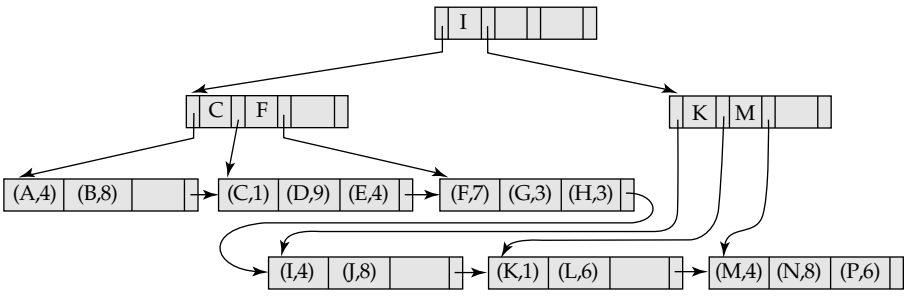


Figure 12.18 B+-tree file organization.

During deletion of a record, if the occupancy of a node falls below $\lfloor 2n/3 \rfloor$, the system attempts to borrow an entry from one of the sibling nodes. If both sibling nodes have $\lfloor 2n/3 \rfloor$ records, instead of borrowing an entry, the system redistributes the entries in the node and in the two siblings evenly between two of the nodes, and deletes the third node. We can use this approach because the total number of entries is $3\lfloor 2n/3 \rfloor - 1$, which is less than $2n$. With three adjacent nodes used for redistribution, each node can be guaranteed to have $\lfloor 3n/4 \rfloor$ entries. In general, if m nodes ($m - 1$ siblings) are involved in redistribution, each node can be guaranteed to contain at least $\lfloor (m - 1)n/m \rfloor$ entries. However, the cost of update becomes higher as more sibling nodes are involved in the redistribution.

12.4 B-Tree Index Files

B-tree indices are similar to B$^+$-tree indices. The primary distinction between the two approaches is that a B-tree eliminates the redundant storage of search-key values. In the B$^+$-tree of Figure 12.12, the search keys "Downtown," "Mianus," "Redwood," and "Perryridge" appear twice. Every search-key value appears in some leaf node; several are repeated in nonleaf nodes.

A B-tree allows search-key values to appear only once. Figure 12.19 shows a B-tree that represents the same search keys as the B$^+$-tree of Figure 12.12. Since search keys are not repeated in the B-tree, we may be able to store the index in fewer tree nodes than in the corresponding B$^+$-tree index. However, since search keys that appear in nonleaf nodes appear nowhere else in the B-tree, we are forced to include an additional pointer field for each search key in a nonleaf node. These additional pointers point to either file records or buckets for the associated search key.

A generalized B-tree leaf node appears in Figure 12.20a; a nonleaf node appears in Figure 12.20b. Leaf nodes are the same as in B$^+$-trees. In nonleaf nodes, the pointers P_i are the tree pointers that we used also for B$^+$-trees, while the pointers B_i are bucket or file-record pointers. In the generalized B-tree in the figure, there are $n - 1$ keys in the leaf node, but there are $m - 1$ keys in the nonleaf node. This discrepancy occurs because nonleaf nodes must include pointers B_i, thus reducing the number of

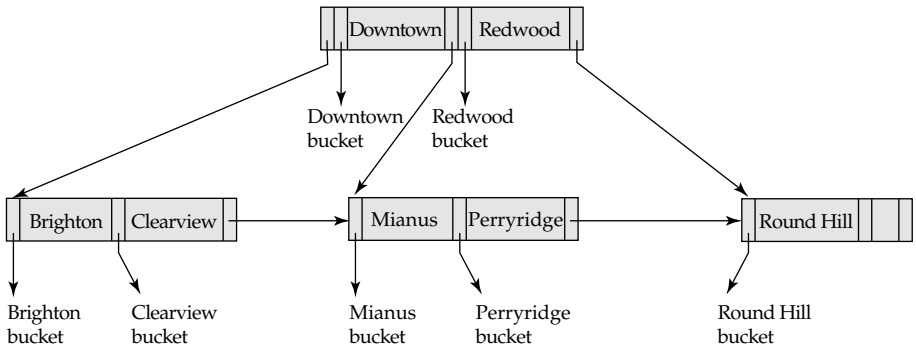


Figure 12.19 B-tree equivalent of B$^+$-tree in Figure 12.12.

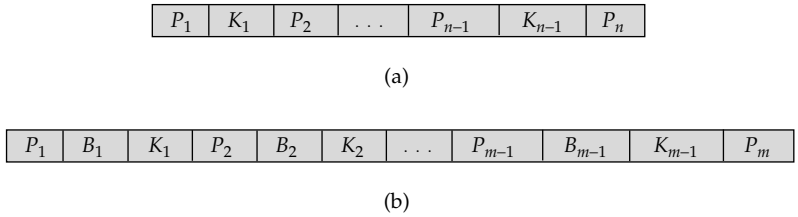


Figure 12.20 Typical nodes of a B-tree. (a) Leaf node. (b) Nonleaf node.

search keys that can be held in these nodes. Clearly, $m < n$, but the exact relationship between m and n depends on the relative size of search keys and pointers.

The number of nodes accessed in a lookup in a B-tree depends on where the search key is located. A lookup on a B$^+$-tree requires traversal of a path from the root of the tree to some leaf node. In contrast, it is sometimes possible to find the desired value in a B-tree before reaching a leaf node. However, roughly n times as many keys are stored in the leaf level of a B-tree as in the nonleaf levels, and, since n is typically large, the benefit of finding certain values early is relatively small. Moreover, the fact that fewer search keys appear in a nonleaf B-tree node, compared to B$^+$-trees, implies that a B-tree has a smaller fanout and therefore may have depth greater than that of the corresponding B$^+$-tree. Thus, lookup in a B-tree is faster for some search keys but slower for others, although, in general, lookup time is still proportional to the logarithm of the number of search keys.

Deletion in a B-tree is more complicated. In a B$^+$-tree, the deleted entry always appears in a leaf. In a B-tree, the deleted entry may appear in a nonleaf node. The proper value must be selected as a replacement from the subtree of the node containing the deleted entry. Specifically, if search key K_i is deleted, the smallest search key appearing in the subtree of pointer P_{i+1} must be moved to the field formerly occupied by K_i. Further actions need to be taken if the leaf node now has too few entries. In contrast, insertion in a B-tree is only slightly more complicated than is insertion in a B$^+$-tree.

The space advantages of B-trees are marginal for large indices, and usually do not outweigh the disadvantages that we have noted. Thus, many database system implementers prefer the structural simplicity of a B$^+$-tree. The exercises explore details of the insertion and deletion algorithms for B-trees.

12.5 Static Hashing

One disadvantage of sequential file organization is that we must access an index structure to locate data, or must use binary search, and that results in more I/O operations. File organizations based on the technique of **hashing** allow us to avoid accessing an index structure. Hashing also provides a way of constructing indices. We study file organizations and indices based on hashing in the following sections.

12.5.1 Hash File Organization

In a **hash file organization**, we obtain the address of the disk block containing a desired record directly by computing a function on the search-key value of the record. In our description of hashing, we shall use the term **bucket** to denote a unit of storage that can store one or more records. A bucket is typically a disk block, but could be chosen to be smaller or larger than a disk block.

Formally, let K denote the set of all search-key values, and let B denote the set of all bucket addresses. A **hash function** h is a function from K to B. Let h denote a hash function.

To insert a record with search key K_i, we compute $h(K_i)$, which gives the address of the bucket for that record. Assume for now that there is space in the bucket to store the record. Then, the record is stored in that bucket.

To perform a lookup on a search-key value K_i, we simply compute $h(K_i)$, then search the bucket with that address. Suppose that two search keys, K_5 and K_7, have the same hash value; that is, $h(K_5) = h(K_7)$. If we perform a lookup on K_5, the bucket $h(K_5)$ contains records with search-key values K_5 and records with search-key values K_7. Thus, we have to check the search-key value of every record in the bucket to verify that the record is one that we want.

Deletion is equally straightforward. If the search-key value of the record to be deleted is K_i, we compute $h(K_i)$, then search the corresponding bucket for that record, and delete the record from the bucket.

12.5.1.1 Hash Functions

The worst possible hash function maps all search-key values to the same bucket. Such a function is undesirable because all the records have to be kept in the same bucket. A lookup has to examine every such record to find the one desired. An ideal hash function distributes the stored keys uniformly across all the buckets, so that every bucket has the same number of records.

Since we do not know at design time precisely which search-key values will be stored in the file, we want to choose a hash function that assigns search-key values to buckets in such a way that the distribution has these qualities:

- The distribution is *uniform*. That is, the hash function assigns each bucket the same number of search-key values from the set of *all* possible search-key values.
- The distribution is *random*. That is, in the average case, each bucket will have nearly the same number of values assigned to it, regardless of the actual distribution of search-key values. More precisely, the hash value will not be correlated to any externally visible ordering on the search-key values, such as alphabetic ordering or ordering by the length of the search keys; the hash function will appear to be random.

As an illustration of these principles, let us choose a hash function for the *account* file using the search key *branch-name*. The hash function that we choose must have

the desirable properties not only on the example *account* file that we have been using, but also on an *account* file of realistic size for a large bank with many branches.

Assume that we decide to have 26 buckets, and we define a hash function that maps names beginning with the *i*th letter of the alphabet to the *i*th bucket. This hash function has the virtue of simplicity, but it fails to provide a uniform distribution, since we expect more branch names to begin with such letters as B and R than Q and X, for example.

Now suppose that we want a hash function on the search key *balance*. Suppose that the minimum balance is 1 and the maximum balance is 100,000, and we use a hash function that divides the values into 10 ranges, 1–10,000, 10,001–20,000 and so on. The distribution of search-key values is uniform (since each bucket has the same number of different *balance* values), but is not random. But records with balances between 1 and 10,000 are far more common than are records with balances between 90,001 and 100,000. As a result, the distribution of records is not uniform—some buckets receive more records than others do. If the function has a random distribution, even if there are such correlations in the search keys, the randomness of the distribution will make it very likely that all buckets will have roughly the same number of records, as long as each search key occurs in only a small fraction of the records. (If a single search key occurs in a large fraction of the records, the bucket containing it is likely to have more records than other buckets, regardless of the hash function used.)

Typical hash functions perform computation on the internal binary machine representation of characters in the search key. A simple hash function of this type first computes the sum of the binary representations of the characters of a key, then returns the sum modulo the number of buckets. Figure 12.21 shows the application of such a scheme, with 10 buckets, to the *account* file, under the assumption that the *i*th letter in the alphabet is represented by the integer *i*.

Hash functions require careful design. A bad hash function may result in lookup taking time proportional to the number of search keys in the file. A well-designed function gives an average-case lookup time that is a (small) constant, independent of the number of search keys in the file.

12.5.1.2 Handling of Bucket Overflows

So far, we have assumed that, when a record is inserted, the bucket to which it is mapped has space to store the record. If the bucket does not have enough space, a **bucket overflow** is said to occur. Bucket overflow can occur for several reasons:

- **Insufficient buckets**. The number of buckets, which we denote n_B, must be chosen such that $n_B > n_r/f_r$, where n_r denotes the total number of records that will be stored, and f_r denotes the number of records that will fit in a bucket. This designation, of course, assumes that the total number of records is known when the hash function is chosen.
- **Skew**. Some buckets are assigned more records than are others, so a bucket may overflow even when other buckets still have space. This situation is called bucket **skew**. Skew can occur for two reasons:

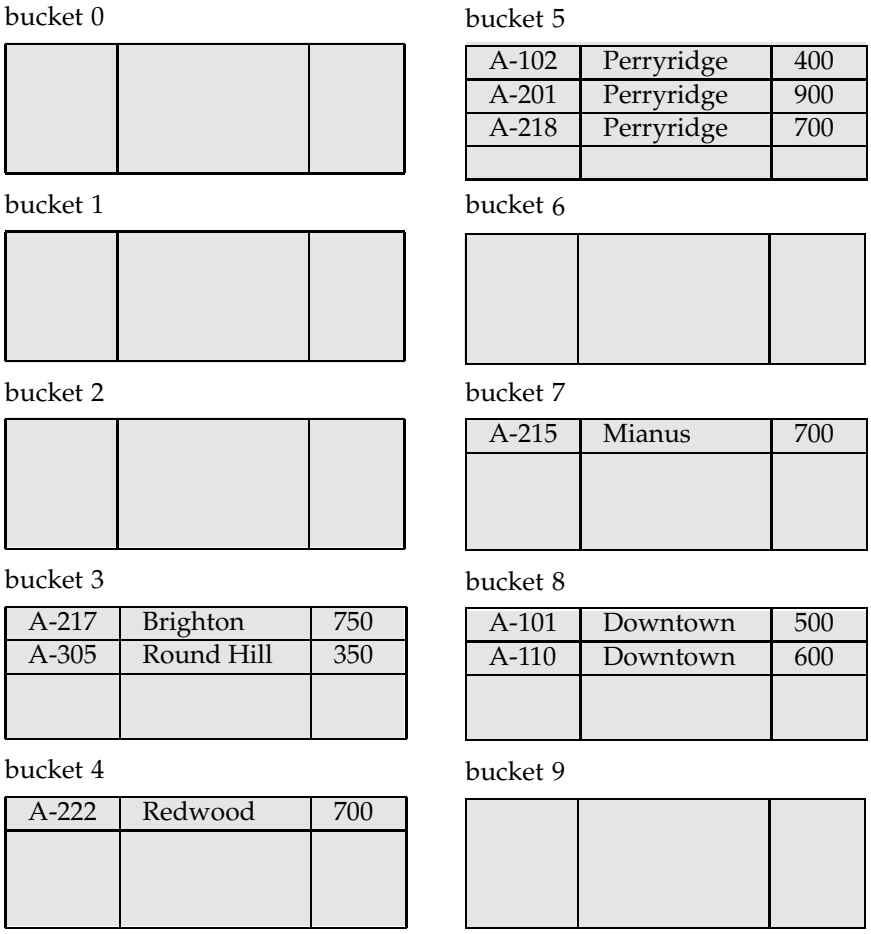


Figure 12.21 Hash organization of *account* file, with *branch-name* as the key.

1. Multiple records may have the same search key.
2. The chosen hash function may result in nonuniform distribution of search keys.

So that the probability of bucket overflow is reduced, the number of buckets is chosen to be $(n_r/f_r) * (1 + d)$, where d is a fudge factor, typically around 0.2. Some space is wasted: About 20 percent of the space in the buckets will be empty. But the benefit is that the probability of overflow is reduced.

Despite allocation of a few more buckets than required, bucket overflow can still occur. We handle bucket overflow by using **overflow buckets**. If a record must be inserted into a bucket b, and b is already full, the system provides an overflow bucket for b, and inserts the record into the overflow bucket. If the overflow bucket is also full, the system provides another overflow bucket, and so on. All the overflow buck-

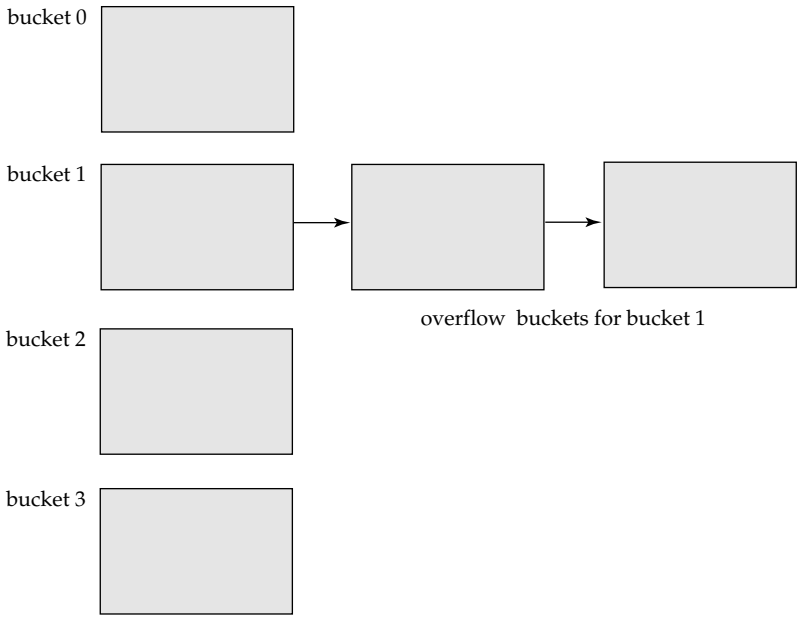


Figure 12.22 Overflow chaining in a hash structure.

ets of a given bucket are chained together in a linked list, as in Figure 12.22. Overflow handling using such a linked list is called **overflow chaining**.

We must change the lookup algorithm slightly to handle overflow chaining. As before, the system uses the hash function on the search key to identify a bucket b. The system must examine all the records in bucket b to see whether they match the search key, as before. In addition, if bucket b has overflow buckets, the system must examine the records in all the overflow buckets also.

The form of hash structure that we have just described is sometimes referred to as **closed hashing**. Under an alternative approach, called **open hashing**, the set of buckets is fixed, and there are no overflow chains. Instead, if a bucket is full, the system inserts records in some other bucket in the initial set of buckets B. One policy is to use the next bucket (in cyclic order) that has space; this policy is called *linear probing*. Other policies, such as computing further hash functions, are also used. Open hashing has been used to construct symbol tables for compilers and assemblers, but closed hashing is preferable for database systems. The reason is that deletion under open hashing is troublesome. Usually, compilers and assemblers perform only lookup and insertion operations on their symbol tables. However, in a database system, it is important to be able to handle deletion as well as insertion. Thus, open hashing is of only minor importance in database implementation.

An important drawback to the form of hashing that we have described is that we must choose the hash function when we implement the system, and it cannot be changed easily thereafter if the file being indexed grows or shrinks. Since the function h maps search-key values to a fixed set B of bucket addresses, we waste space if B is

made large to handle future growth of the file. If B is too small, the buckets contain records of many different search-key values, and bucket overflows can occur. As the file grows, performance suffers. We study later, in Section 12.6, how the number of buckets and the hash function can be changed dynamically.

12.5.2 Hash Indices

Hashing can be used not only for file organization, but also for index-structure creation. A **hash index** organizes the search keys, with their associated pointers, into a hash file structure. We construct a hash index as follows. We apply a hash function on a search key to identify a bucket, and store the key and its associated pointers in the bucket (or in overflow buckets). Figure 12.23 shows a secondary hash index on the *account* file, for the search key *account-number*. The hash function in the figure computes the sum of the digits of the account number modulo 7. The hash index has seven buckets, each of size 2 (realistic indices would, of course, have much larger

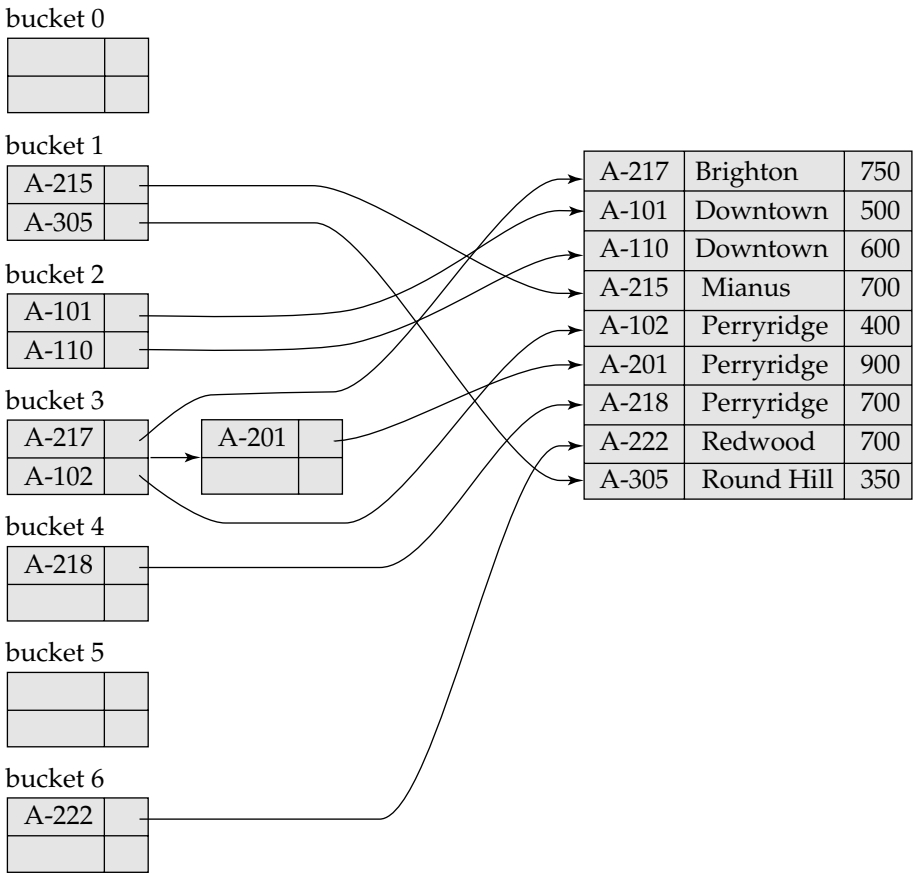


Figure 12.23 Hash index on search key *account-number* of *account* file.

bucket sizes). One of the buckets has three keys mapped to it, so it has an overflow bucket. In this example, *account-number* is a primary key for *account*, so each search-key has only one associated pointer. In general, multiple pointers can be associated with each key.

We use the term *hash index* to denote hash file structures as well as secondary hash indices. Strictly speaking, hash indices are only secondary index structures. A hash index is never needed as a primary index structure, since, if a file itself is organized by hashing, there is no need for a separate hash index structure on it. However, since hash file organization provides the same direct access to records that indexing provides, we pretend that a file organized by hashing also has a primary hash index on it.

12.6 Dynamic Hashing

As we have seen, the need to fix the set B of bucket addresses presents a serious problem with the static hashing technique of the previous section. Most databases grow larger over time. If we are to use static hashing for such a database, we have three classes of options:

1. Choose a hash function based on the current file size. This option will result in performance degradation as the database grows.
2. Choose a hash function based on the anticipated size of the file at some point in the future. Although performance degradation is avoided, a significant amount of space may be wasted initially.
3. Periodically reorganize the hash structure in response to file growth. Such a reorganization involves choosing a new hash function, recomputing the hash function on every record in the file, and generating new bucket assignments. This reorganization is a massive, time-consuming operation. Furthermore, it is necessary to forbid access to the file during reorganization.

Several **dynamic hashing** techniques allow the hash function to be modified dynamically to accommodate the growth or shrinkage of the database. In this section we describe one form of dynamic hashing, called **extendable hashing**. The bibliographical notes provide references to other forms of dynamic hashing.

12.6.1 Data Structure

Extendable hashing copes with changes in database size by splitting and coalescing buckets as the database grows and shrinks. As a result, space efficiency is retained. Moreover, since the reorganization is performed on only one bucket at a time, the resulting performance overhead is acceptably low.

With extendable hashing, we choose a hash function h with the desirable properties of uniformity and randomness. However, this hash function generates values over a relatively large range—namely, b-bit binary integers. A typical value for b is 32.

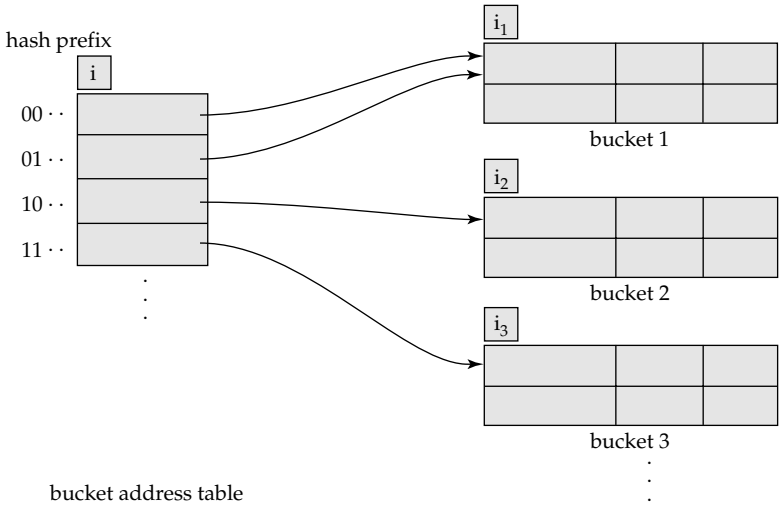


Figure 12.24 General extendable hash structure.

We do not create a bucket for each hash value. Indeed, 2^{32} is over 4 billion, and that many buckets is unreasonable for all but the largest databases. Instead, we create buckets on demand, as records are inserted into the file. We do not use the entire b bits of the hash value initially. At any point, we use i bits, where $0 \leq i \leq b$. These i bits are used as an offset into an additional table of bucket addresses. The value of i grows and shrinks with the size of the database.

Figure 12.24 shows a general extendable hash structure. The i appearing above the bucket address table in the figure indicates that i bits of the hash value $h(K)$ are required to determine the correct bucket for K. This number will, of course, change as the file grows. Although i bits are required to find the correct entry in the bucket address table, several consecutive table entries may point to the same bucket. All such entries will have a common hash prefix, but the length of this prefix may be less than i. Therefore, we associate with each bucket an integer giving the length of the common hash prefix. In Figure 12.24 the integer associated with bucket j is shown as i_j. The number of bucket-address-table entries that point to bucket j is

$$2^{(i - i_j)}$$

12.6.2 Queries and Updates

We now see how to perform lookup, insertion, and deletion on an extendable hash structure.

To locate the bucket containing search-key value K_l, the system takes the first i high-order bits of $h(K_l)$, looks at the corresponding table entry for this bit string, and follows the bucket pointer in the table entry.

To insert a record with search-key value K_l, the system follows the same procedure for lookup as before, ending up in some bucket—say, j. If there is room in the bucket,

the system inserts the record in the bucket. If, on the other hand, the bucket is full, it must split the bucket and redistribute the current records, plus the new one. To split the bucket, the system must first determine from the hash value whether it needs to increase the number of bits that it uses.

- If $i = i_j$, only one entry in the bucket address table points to bucket *j*. Therefore, the system needs to increase the size of the bucket address table so that it can include pointers to the two buckets that result from splitting bucket *j*. It does so by considering an additional bit of the hash value. It increments the value of *i* by 1, thus doubling the size of the bucket address table. It replaces each entry by two entries, both of which contain the same pointer as the original entry. Now two entries in the bucket address table point to bucket *j*. The system allocates a new bucket (bucket *z*), and sets the second entry to point to the new bucket. It sets i_j and i_z to *i*. Next, it rehashes each record in bucket *j* and, depending on the first *i* bits (remember the system has added 1 to *i*), either keeps it in bucket *j* or allocates it to the newly created bucket.

  The system now reattempts the insertion of the new record. Usually, the attempt will succeed. However, if all the records in bucket *j*, as well as the new record, have the same hash-value prefix, it will be necessary to split a bucket again, since all the records in bucket *j* and the new record are assigned to the same bucket. If the hash function has been chosen carefully, it is unlikely that a single insertion will require that a bucket be split more than once, unless there are a large number of records with the same search key. If all the records in bucket *j* have the same search-key value, no amount of splitting will help. In such cases, overflow buckets are used to store the records, as in static hashing.

- If $i > i_j$, then more than one entry in the bucket address table points to bucket *j*. Thus, the system can split bucket *j* without increasing the size of the bucket address table. Observe that all the entries that point to bucket *j* correspond to hash prefixes that have the same value on the leftmost i_j bits. The system allocates a new bucket (bucket *z*), and set i_j and i_z to the value that results from adding 1 to the original i_j value. Next, the system needs to adjust the entries in the bucket address table that previously pointed to bucket *j*. (Note that with the new value for i_j, not all the entries correspond to hash prefixes that have the same value on the leftmost i_j bits.) The system leaves the first half of the entries as they were (pointing to bucket *j*), and sets all the remaining entries to point to the newly created bucket (bucket *z*). Next, as in the previous case, the system rehashes each record in bucket *j*, and allocates it either to bucket *j* or to the newly created bucket *z*.

  The system then reattempts the insert. In the unlikely case that it again fails, it applies one of the two cases, $i = i_j$ or $i > i_j$, as appropriate.

Note that, in both cases, the system needs to recompute the hash function on only the records in bucket *j*.

To delete a record with search-key value K_l, the system follows the same procedure for lookup as before, ending up in some bucket—say, *j*. It removes both the

A-217	Brighton	750
A-101	Downtown	500
A-1 10	Downtown	600
A-215	Mianus	700
A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700
A-222	Redwood	700
A-305	Round Hill	350

Figure 12.25 Sample *account* file.

search key from the bucket and the record from the file. The bucket too is removed if it becomes empty. Note that, at this point, several buckets can be coalesced, and the size of the bucket address table can be cut in half. The procedure for deciding on which buckets can be coalesced and how to coalesce buckets is left to you to do as an exercise. The conditions under which the bucket address table can be reduced in size are also left to you as an exercise. Unlike coalescing of buckets, changing the size of the bucket address table is a rather expensive operation if the table is large. Therefore it may be worthwhile to reduce the bucket address table size only if the number of buckets reduces greatly.

Our example *account* file in Figure 12.25 illustrates the operation of insertion. The 32-bit hash values on *branch-name* appear in Figure 12.26. Assume that, initially, the file is empty, as in Figure 12.27. We insert the records one by one. To illustrate all the features of extendable hashing in a small structure, we shall make the unrealistic assumption that a bucket can hold only two records.

We insert the record (A-217, Brighton, 750). The bucket address table contains a pointer to the one bucket, and the system inserts the record. Next, we insert the record (A-101, Downtown, 500). The system also places this record in the one bucket of our structure.

When we attempt to insert the next record (Downtown, A-110, 600), we find that the bucket is full. Since $i = i_0$, we need to increase the number of bits that we use from the hash value. We now use 1 bit, allowing us $2^1 = 2$ buckets. This increase in

branch-name	h(*branch-name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001

Figure 12.26 Hash function for *branch-name*.

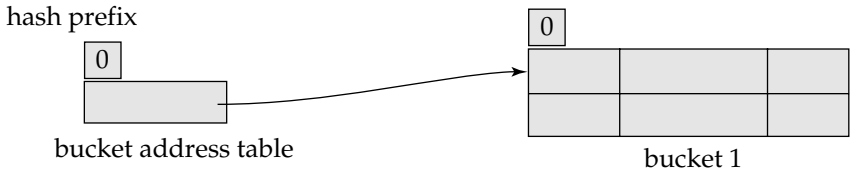


Figure 12.27 Initial extendable hash structure.

the number of bits necessitates doubling the size of the bucket address table to two entries. The system splits the bucket, placing in the new bucket those records whose search key has a hash value beginning with 1, and leaving in the original bucket the other records. Figure 12.28 shows the state of our structure after the split.

Next, we insert (A-215, Mianus, 700). Since the first bit of h(Mianus) is 1, we must insert this record into the bucket pointed to by the "1" entry in the bucket address table. Once again, we find the bucket full and $i = i_1$. We increase the number of bits that we use from the hash to 2. This increase in the number of bits necessitates doubling the size of the bucket address table to four entries, as in Figure 12.29. Since the bucket of Figure 12.28 for hash prefix 0 was not split, the two entries of the bucket address table of 00 and 01 both point to this bucket.

For each record in the bucket of Figure 12.28 for hash prefix 1 (the bucket being split), the system examines the first 2 bits of the hash value to determine which bucket of the new structure should hold it.

Next, we insert (A-102, Perryridge, 400), which goes in the same bucket as Mianus. The following insertion, of (A-201, Perryridge, 900), results in a bucket overflow, leading to an increase in the number of bits, and a doubling of the size of the bucket address table. The insertion of the third Perryridge record, (A-218, Perryridge, 700), leads to another overflow. However, this overflow cannot be handled by increasing the number of bits, since there are three records with exactly the same hash value. Hence the system uses an overflow bucket, as in Figure 12.30.

We continue in this manner until we have inserted all the *account* records of Figure 12.25. The resulting structure appears in Figure 12.31.

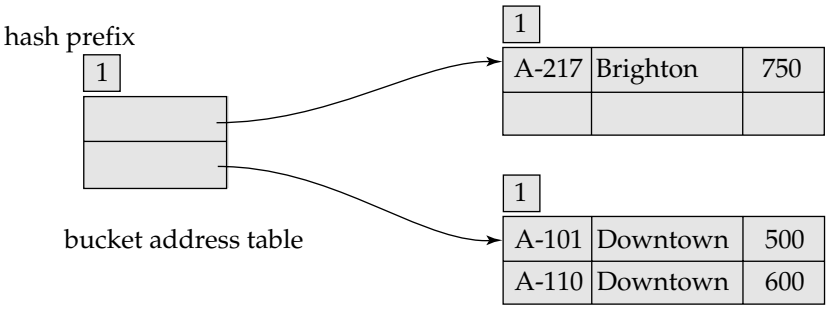


Figure 12.28 Hash structure after three insertions.

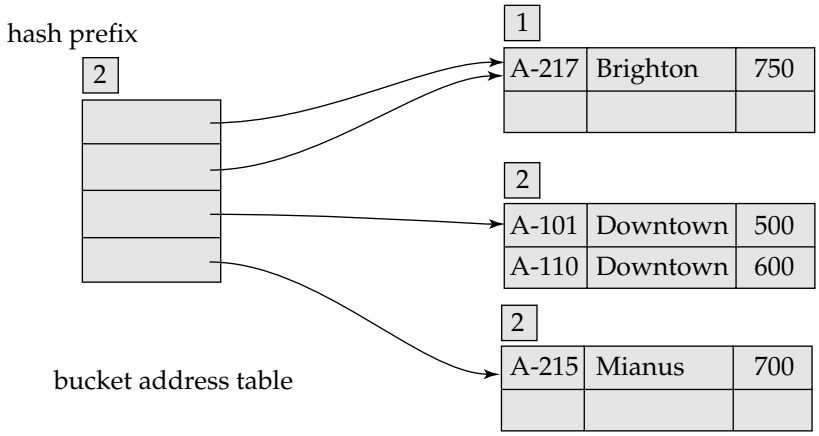


Figure 12.29 Hash structure after four insertions.

12.6.3 Comparison with Other Schemes

We now examine the advantages and disadvantages of extendable hashing, compared with the other schemes that we have discussed. The main advantage of extendable hashing is that performance does not degrade as the file grows. Furthermore, there is minimal space overhead. Although the bucket address table incurs additional overhead, it contains one pointer for each hash value for the current pre-

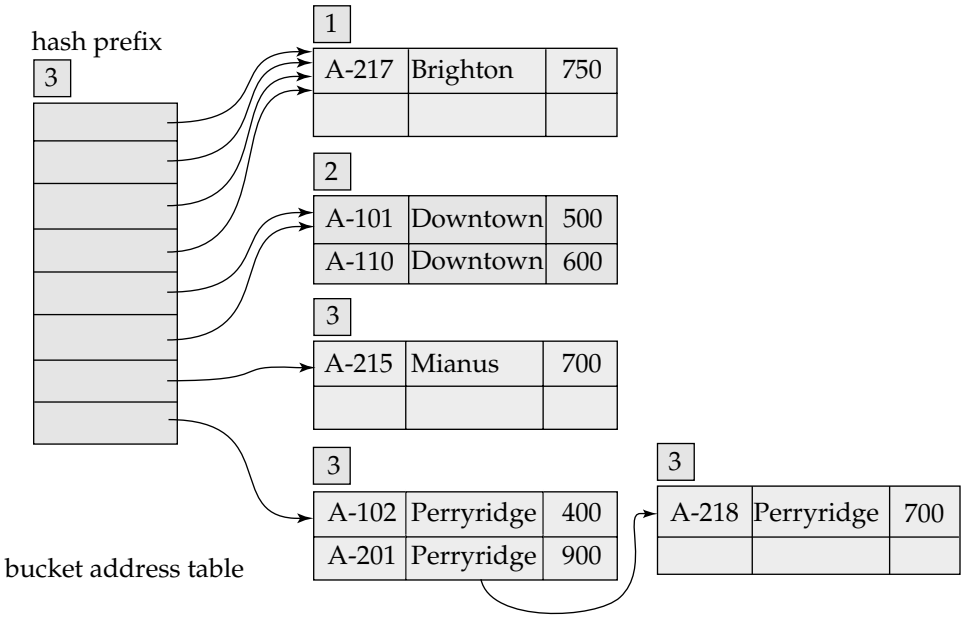


Figure 12.30 Hash structure after seven insertions.

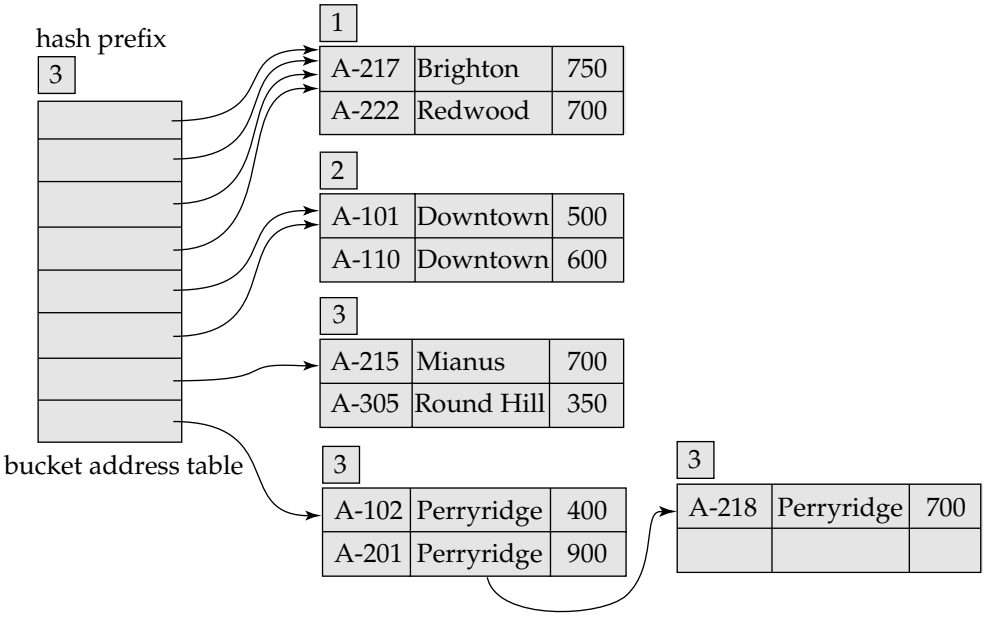


Figure 12.31 Extendable hash structure for the *account* file.

fix length. This table is thus small. The main space saving of extendable hashing over other forms of hashing is that no buckets need to be reserved for future growth; rather, buckets can be allocated dynamically.

A disadvantage of extendable hashing is that lookup involves an additional level of indirection, since the system must access the bucket address table before accessing the bucket itself. This extra reference has only a minor effect on performance. Although the hash structures that we discussed in Section 12.5 do not have this extra level of indirection, they lose their minor performance advantage as they become full.

Thus, extendable hashing appears to be a highly attractive technique, provided that we are willing to accept the added complexity involved in its implementation. The bibliographical notes reference more detailed descriptions of extendable hashing implementation. The bibliographical notes also provide references to another form of dynamic hashing called **linear hashing**, which avoids the extra level of indirection associated with extendable hashing, at the possible cost of more overflow buckets.

12.7 Comparison of Ordered Indexing and Hashing

We have seen several ordered-indexing schemes and several hashing schemes. We can organize files of records as ordered files, by using index-sequential organization or B^+-tree organizations. Alternatively, we can organize the files by using hashing. Finally, we can organize them as heap files, where the records are not ordered in any particular way.

Each scheme has advantages in certain situations. A database-system implementor could provide many schemes, leaving the final decision of which schemes to use to the database designer. However, such an approach requires the implementor to write more code, adding both to the cost of the system and to the space that the system occupies. Most database systems support B^+-trees and may additionally support some form of hash file organization or hash indices.

To make a wise choice of file organization and indexing techniques for a relation, the implementor or the database designer must consider the following issues:

- Is the cost of periodic reorganization of the index or hash organization acceptable?
- What is the relative frequency of insertion and deletion?
- Is it desirable to optimize average access time at the expense of increasing the worst-case access time?
- What types of queries are users likely to pose?

We have already examined the first three of these issues, first in our review of the relative merits of specific indexing techniques, and again in our discussion of hashing techniques. The fourth issue, the expected type of query, is critical to the choice of ordered indexing or hashing.

If most queries are of the form

select $A_1, A_2, \ldots, A_n$
from r
where $A_i = c$

then, to process this query, the system will perform a lookup on an ordered index or a hash structure for attribute A_i, for value c. For queries of this form, a hashing scheme is preferable. An ordered-index lookup requires time proportional to the log of the number of values in r for A_i. In a hash structure, however, the average lookup time is a constant independent of the size of the database. The only advantage to an index over a hash structure for this form of query is that the worst-case lookup time is proportional to the log of the number of values in r for A_i. By contrast, for hashing, the worst-case lookup time is proportional to the number of values in r for A_i. However, the worst-case lookup time is unlikely to occur with hashing, and hashing is preferable in this case.

Ordered-index techniques are preferable to hashing in cases where the query specifies a range of values. Such a query takes the following form:

select $A_1, A_2, ..., A_n$
from r
where $A_i \leq c_2$ **and** $A_i \geq c_1$

In other words, the preceding query finds all the records with A_i values between c_1 and c_2.

Let us consider how we process this query using an ordered index. First, we perform a lookup on value c_1. Once we have found the bucket for value c_1, we follow the pointer chain in the index to read the next bucket in order, and we continue in this manner until we reach c_2.

If, instead of an ordered index, we have a hash structure, we can perform a lookup on c_1 and can locate the corresponding bucket—but it is not easy, in general, to determine the next bucket that must be examined. The difficulty arises because a good hash function assigns values randomly to buckets. Thus, there is no simple notion of "next bucket in sorted order." The reason we cannot chain buckets together in sorted order on A_i is that each bucket is assigned many search-key values. Since values are scattered randomly by the hash function, the values in the specified range are likely to be scattered across many or all of the buckets. Therefore, we have to read all the buckets to find the required search keys.

Usually the designer will choose ordered indexing unless it is known in advance that range queries will be infrequent, in which case hashing would be chosen. Hash organizations are particularly useful for temporary files created during query processing, if lookups based on a key value are required, but no range queries will be performed.

12.8 Index Definition in SQL

The SQL standard does not provide any way for the database user or administrator to control what indices are created and maintained in the database system. Indices are not required for correctness, since they are redundant data structures. However, indices are important for efficient processing of transactions, including both update transactions and queries. Indices are also important for efficient enforcement of integrity constraints. For example, typical implementations enforce a key declaration (Chapter 6) by creating an index with the declared key as the search key of the index.

In principle, a database system can decide automatically what indices to create. However, because of the space cost of indices, as well as the effect of indices on update processing, it is not easy to automatically make the right choices about what indices to maintain. Therefore, most SQL implementations provide the programmer control over creation and removal of indices via data-definition-language commands.

We illustrate the syntax of these commands next. Although the syntax that we show is widely used and supported by many database systems, it is not part of the SQL:1999 standard. The SQL standards (up to SQL:1999, at least) do not support control of the physical database schema, and have restricted themselves to the logical database schema.

We create an index by the **create index** command, which takes the form

create index <index-name> **on** <relation-name> (<attribute-list>)

The *attribute-list* is the list of attributes of the relations that form the search key for the index.

To define an index name *b-index* on the *branch* relation with *branch-name* as the search key, we write

create index *b-index* **on** *branch* (*branch-name*)

If we wish to declare that the search key is a candidate key, we add the attribute **unique** to the index definition. Thus, the command

create unique index *b-index* **on** *branch* (*branch-name*)

declares *branch-name* to be a candidate key for *branch*. If, at the time we enter the **create unique index** command, *branch-name* is not a candidate key, the system will display an error message, and the attempt to create the index will fail. If the index-creation attempt succeeds, any subsequent attempt to insert a tuple that violates the key declaration will fail. Note that the **unique** feature is redundant if the database system supports the **unique** declaration of the SQL standard.

Many database systems also provide a way to specify the type of index to be used (such as B^+-tree or hashing). Some database systems also permit one of the indices on a relation to be declared to be clustered; the system then stores the relation sorted by the search-key of the clustered index.

The index name we specified for an index is required to drop an index. The **drop index** command takes the form:

drop index <index-name>

12.9 Multiple-Key Access

Until now, we have assumed implicitly that only one index (or hash table) is used to process a query on a relation. However, for certain types of queries, it is advantageous to use multiple indices if they exist.

12.9.1 Using Multiple Single-Key Indices

Assume that the *account* file has two indices: one for *branch-name* and one for *balance*. Consider the following query: "Find all account numbers at the Perryridge branch with balances equal to $1000." We write

select *loan-number*
from *account*
where *branch-name* = "Perryridge" **and** *balance* = 1000

There are three strategies possible for processing this query:

1. Use the index on *branch-name* to find all records pertaining to the Perryridge branch. Examine each such record to see whether *balance* = 1000.
2. Use the index on *balance* to find all records pertaining to accounts with balances of $1000. Examine each such record to see whether *branch-name* = "Perryridge."
3. Use the index on *branch-name* to find *pointers* to all records pertaining to the Perryridge branch. Also, use the index on *balance* to find pointers to all records

pertaining to accounts with a balance of $1000. Take the intersection of these two sets of pointers. Those pointers that are in the intersection point to records pertaining to both Perryridge and accounts with a balance of $1000.

The third strategy is the only one of the three that takes advantage of the existence of multiple indices. However, even this strategy may be a poor choice if all of the following hold:

- There are many records pertaining to the Perryridge branch.
- There are many records pertaining to accounts with a balance of $1000.
- There are only a few records pertaining to *both* the Perryridge branch and accounts with a balance of $1000.

If these conditions hold, we must scan a large number of pointers to produce a small result. An index structure called a "bitmap index" greatly speeds up the intersection operation used in the third strategy. Bitmap indices are outlined in Section 12.9.4.

12.9.2 Indices on Multiple Keys

An alternative strategy for this case is to create and use an index on a search key (*branch-name*, *balance*)—that is, the search key consisting of the branch name concatenated with the account balance. The structure of the index is the same as that of any other index, the only difference being that the search key is not a single attribute, but rather is a list of attributes. The search key can be represented as a tuple of values, of the form $(a_1, \ldots, a_n)$, where the indexed attributes are $A_1, \ldots, A_n$. The ordering of search-key values is the *lexicographic ordering*. For example, for the case of two attribute search keys, $(a_1, a_2) < (b_1, b_2)$ if either $a_1 < b_1$ or $a_1 = b_1$ and $a_2 < b_2$. Lexicographic ordering is basically the same as alphabetic ordering of words.

The use of an ordered-index structure on multiple attributes has a few shortcomings. As an illustration, consider the query

select *loan-number*
from *account*
where *branch-name* $<$ "Perryridge" **and** *balance* $= 1000$

We can answer this query by using an ordered index on the search key (*branch-name*, *balance*): For each value of *branch-name* that is less than "Perryridge" in alphabetic order, the system locates records with a *balance* value of 1000. However, each record is likely to be in a different disk block, because of the ordering of records in the file, leading to many I/O operations.

The difference between this query and the previous one is that the condition on *branch-name* is a comparison condition, rather than an equality condition.

To speed the processing of general multiple search-key queries (which can involve one or more comparison operations), we can use several special structures. We shall consider the *grid file* in Section 12.9.3. There is another structure, called the *R-tree*, that

can be used for this purpose. The R-tree is an extension of the B^+-tree to handle indexing on multiple dimensions. Since the R-tree is used primarily with geographical data types, we describe the structure in Chapter 23.

12.9.3 Grid Files

Figure 12.32 shows part of a **grid file** for the search keys *branch-name* and *balance* on the *account* file. The two-dimensional array in the figure is called the *grid array*, and the one-dimensional arrays are called *linear scales*. The grid file has a single grid array, and one linear scale for each search-key attribute.

Search keys are mapped to cells in this way. Each cell in the grid array has a pointer to a bucket that contains the search-key values and pointers to records. Only some of the buckets and pointers from the cells are shown in the figure. To conserve space, multiple elements of the array can point to the same bucket. The dotted boxes in the figure indicate which cells point to the same bucket.

Suppose that we want to insert in the grid-file index a record whose search-key value is ("Brighton", 500000). To find the cell to which the key is mapped, we independently locate the row and column to which the cell belongs.

We first use the linear scales on *branch-name* to locate the row of the cell to which the search key maps. To do so, we search the array to find the least element that is greater than "Brighton". In this case, it is the first element, so the search key maps to the row marked 0. If it were the ith element, the search key would map to row $i - 1$. If the search key is greater than or equal to all elements in the linear scale, it maps to

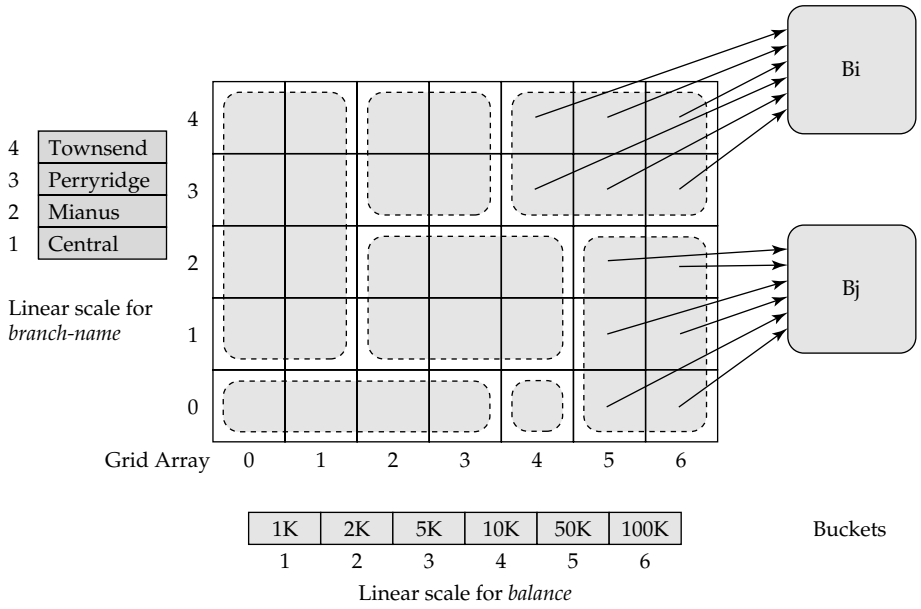


Figure 12.32 Grid file on keys *branch-name* and *balance* of the *account* file.

the final row. Next, we use the linear scale on *balance* to find out similarly to which column the search key maps. In this case, the balance 500000 maps to column 6.

Thus, the search-key value ("Brighton", 500000) maps to the cell in row 0, column 6. Similarly, ("Downtown", 60000) would map to the cell in row 1 column 5. Both cells point to the same bucket (as indicated by the dotted box), so, in both cases, the system stores the search-key values and the pointer to the record in the bucket labeled B_j in the figure.

To perform a lookup to answer our example query, with the search condition of

$$\textit{branch-name} < \text{"Perryridge"} \textbf{ and } \textit{balance} = 1000$$

we find all rows that can contain branch names less than "Perryridge", using the linear scale on *branch-name*. In this case, these rows are 0, 1, and 2. Rows 3 and beyond contain branch names greater than or equal to "Perryridge". Similarly, we find that only column 1 can contain a *balance* value of 1000. In this case, only column 1 satisfies this condition. Thus, only the cells in column 1, rows 0, 1, and 2, can contain entries that satisfy the search condition.

We therefore look up all entries in the buckets pointed to from these three cells. In this case, there are only two buckets, since two of the cells point to the same bucket, as indicated by the dotted boxes in the figure. The buckets may contain some search keys that do not satisfy the required condition, so each search key in the buckets must be tested again to see whether it satisfies the search condition. We have to examine only a small number of buckets, however, to answer this query.

We must choose the linear scales in such a way that the records are uniformly distributed across the cells. When a bucket—call it A—becomes full and an entry has to be inserted in it, the system allocates an extra bucket, B. If more than one cell points to A, the system changes the cell pointers so that some point to A and others to B. The entries in bucket A and the new entry are then redistributed between A and B according to the cells to which they map. If only one cell points to bucket A, B becomes an overflow bucket for A. To improve performance in such a situation, we must reorganize the grid file, with an expanded grid array and expanded linear scales. The process is much like the expansion of the bucket address table in extensible hashing, and is left for you to do as an exercise.

It is conceptually simple to extend the grid-file approach to any number of search keys. If we want our structure to be used for queries on n keys, we construct an n-dimensional grid array with n linear scales.

The grid structure is suitable also for queries involving one search key. Consider this query:

select *
from *account*
where *branch-name* = "Perryridge"

The linear scale on *branch-name* tells us that only cells in row 3 can satisfy this condition. Since there is no condition on *balance*, we examine all buckets pointed to by cells in row 3 to find entries pertaining to Perryridge. Thus, we can use a grid-file index on

two search keys to answer queries on either search key by itself, as well as to answer queries on both search keys. Thus, a single grid-file index can serve the role of three separate indices. If each index were maintained separately, the three together would occupy more space, and the cost of updating them would be high.

Grid files provide a significant decrease in processing time for multiple-key queries. However, they impose a space overhead (the grid directory can become large), as well as a performance overhead on record insertion and deletion. Further, it is hard to choose partitioning ranges for the keys such that the distribution of records is uniform. If insertions to the file are frequent, reorganization will have to be carried out periodically, and that can have a high cost.

12.9.4 Bitmap Indices

Bitmap indices are a specialized type of index designed for easy querying on multiple keys, although each bitmap index is built on a single key.

For bitmap indices to be used, records in a relation must be numbered sequentially, starting from, say, 0. Given a number n, it must be easy to retrieve the record numbered n. This is particularly easy to achieve if records are fixed in size, and allocated on consecutive blocks of a file. The record number can then be translated easily into a block number and a number that identifies the record within the block.

Consider a relation r, with an attribute A that can take on only one of a small number (for example, 2 to 20) values. For instance, a relation *customer-info* may have an attribute *gender*, which can take only values m (male) or f (female). Another example would be an attribute *income-level*, where income has been broken up into 5 levels: $L1$: $\$0 - 9999$, $L2$: $\$10,000 - 19,999$, $L3$: $20,000 - 39,999$, $L4$: $40,000 - 74,999$, and $L5$: $75,000 - \infty$. Here, the raw data can take on many values, but a data analyst has split the values into a small number of ranges to simplify analysis of the data.

12.9.4.1 Bitmap Index Structure

A **bitmap** is simply an array of bits. In its simplest form, a **bitmap index** on the attribute A of relation r consists of one bitmap for each value that A can take. Each bitmap has as many bits as the number of records in the relation. The ith bit of the bitmap for value v_j is set to 1 if the record numbered i has the value v_j for attribute A. All other bits of the bitmap are set to 0.

In our example, there is one bitmap for the value m and one for f. The ith bit of the bitmap for m is set to 1 if the *gender* value of the record numbered i is m. All other bits of the bitmap for m are set to 0. Similarly, the bitmap for f has the value 1 for bits corresponding to records with the value f for the *gender* attribute; all other bits have the value 0. Figure 12.33 shows an example of bitmap indices on a relation *customer-info*.

We now consider when bitmaps are useful. The simplest way of retrieving all records with value m (or value f) would be to simply read all records of the relation and select those records with value m (or f, respectively). The bitmap index doesn't really help to speed up such a selection.

record number	*name*	*gender*	*address*	*income -level*
0	John	m	Perryridge	L1
1	Diana	f	Brooklyn	L2
2	Mary	f	Jonestown	L1
3	Peter	m	Brooklyn	L4
4	Kathy	f	Perryridge	L3

Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income-level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

Figure 12.33 Bitmap indices on relation *customer-info*.

In fact, bitmap indices are useful for selections mainly when there are selections on multiple keys. Suppose we create a bitmap index on attribute *income-level*, which we described earlier, in addition to the bitmap index on *gender*.

Consider now a query that selects women with income in the range $10,000 - 19,999$. This query can be expressed as $\sigma_{gender=\text{f} \wedge income\text{-}level=L2}(r)$. To evaluate this selection, we fetch the bitmaps for *gender* value f and the bitmap for *income-level* value $L2$, and perform an **intersection** (logical-and) of the two bitmaps. In other words, we compute a new bitmap where bit i has value 1 if the ith bit of the two bitmaps are both 1, and has a value 0 otherwise. In the example in Figure 12.33, the intersection of the bitmap for $gender = \text{f}$ (01101) and the bitmap for $income\text{-}level = L1$ (10100) gives the bitmap 00100.

Since the first attribute can take 2 values, and the second can take 5 values, we would expect only about 1 in 10 records, on an average, to satisfy a combined condition on the two attributes. If there are further conditions, the fraction of records satisfying all the conditions is likely to be quite small. The system can then compute the query result by finding all bits with value 1 in the intersection bitmap, and retrieving the corresponding records. If the fraction is large, scanning the entire relation would remain the cheaper alternative.

Another important use of bitmaps is to count the number of tuples satisfying a given selection. Such queries are important for data analysis. For instance, if we wish to find out how many women have an income level $L2$, we compute the intersection of the two bitmaps, and then count the number of bits that are 1 in the intersection bitmap. We can thus get the desired result from the bitmap index, without even accessing the relation.

Bitmap indices are generally quite small compared to the actual relation size. Records are typically at least tens of bytes to hundreds of bytes long, whereas a single bit represents the record in a bitmap. Thus the space occupied by a single bitmap is usually less than 1 percent of the space occupied by the relation. For instance, if the record size for a given relation is 100 bytes, then the space occupied by a single bitmap would be $\frac{1}{8}$ of 1 percent of the space occupied by the relation. If an attribute A of the relation can take on only one of 8 values, a bitmap index on attribute A would consist of 8 bitmaps, which together occupy only 1 percent of the size of the relation.

Deletion of records creates gaps in the sequence of records, since shifting records (or record numbers) to fill gaps would be extremely expensive. To recognize deleted records, we can store an **existence bitmap**, in which bit i is 0 if record i does not exist and 1 otherwise. We will see the need for existence bitmaps in Section 12.9.4.2. Insertion of records should not affect the sequence numbering of other records. Therefore, we can do insertion either by appending records to the end of the file or by replacing deleted records.

12.9.4.2 Efficient Implementation of Bitmap Operations

We can compute the intersection of two bitmaps easily by using a **for** loop: the ith iteration of the loop computes the **and** of the ith bits of the two bitmaps. We can speed up computation of the intersection greatly by using bit-wise **and** instructions supported by most computer instruction sets. A *word* usually consists of 32 or 64 bits, depending on the architecture of the computer. A bit-wise **and** instruction takes two words as input and outputs a word where each bit is the logical **and** of the bits in corresponding positions of the input words. What is important to note is that a single bit-wise **and** instruction can compute the intersection of 32 or 64 bits *at once*.

If a relation had 1 million records, each bitmap would contain 1 million bits, or equivalently 128 Kbytes. Only 31,250 instructions are needed to compute the intersection of two bitmaps for our relation, assuming a 32-bit word length. Thus, computing bitmap intersections is an extremely fast operation.

Just like bitmap intersection is useful for computing the **and** of two conditions, bitmap union is useful for computing the **or** of two conditions. The procedure for bitmap union is exactly the same as for intersection, except we use bit-wise **or** instructions instead of bit-wise **and** instructions.

The complement operation can be used to compute a predicate involving the negation of a condition, such as **not** (*income-level* = $L1$). The complement of a bitmap is generated by complementing every bit of the bitmap (the complement of 1 is 0 and the complement of 0 is 1). It may appear that **not** (*income-level* = $L1$) can be implemented by just computing the complement of the bitmap for income level $L1$. If some records have been deleted, however, just computing the complement of a bitmap is not sufficient. Bits corresponding to such records would be 0 in the original bitmap, but would become 1 in the complement, although the records don't exist. A similar problem also arises when the value of an attribute is *null*. For instance, if the value of *income-level* is null, the bit would be 0 in the original bitmap for value $L1$, and 1 in the complement bitmap.

To make sure that the bits corresponding to deleted records are set to 0 in the result, the complement bitmap must be intersected with the existence bitmap to turn off the bits for deleted records. Similarly, to handle null values, the complement bitmap must also be intersected with the complement of the bitmap for the value *null*.[1]

Counting the number of bits that are 1 in a bitmap can be done fast by a clever technique. We can maintain an array with 256 entries, where the ith entry stores the

1. Handling predicates such as **is unknown** would cause further complications, which would in general require use of an extra bitmap to to track which operation results are unknown.

number of bits that are 1 in the binary representation of i. Set the total count initially to 0. We take each byte of the bitmap, use it to index into this array, and add the stored count to the total count. The number of addition operations would be $\frac{1}{8}$ of the number of tuples, and thus the counting process is very efficient. A large array (using $2^{16} = 65536$ entries), indexed by pairs of bytes, would give even higher speedup, but at a higher storage cost.

12.9.4.3 Bitmaps and B$^+$-Trees

Bitmaps can be combined with regular B$^+$-tree indices for relations where a few attribute values are extremely common, and other values also occur, but much less frequently. In a B$^+$-tree index leaf, for each value we would normally maintain a list of all records with that value for the indexed attribute. Each element of the list would be a record identifier, consisting of at least 32 bits, and usually more. For a value that occurs in many records, we store a bitmap instead of a list of records.

Suppose a particular value v_i occurs in $\frac{1}{16}$ of the records of a relation. Let N be the number of records in the relation, and assume that a record has a 64-bit number identifying it. The bitmap needs only 1 bit per record, or N bits in total. In contrast, the list representation requires 64 bits per record where the value occurs, or $64 * N/16 = 4N$ bits. Thus, a bitmap is preferable for representing the list of records for value v_i. In our example (with a 64-bit record identifier), if fewer than 1 in 64 records have a particular value, the list representation is preferable for identifying records with that value, since it uses fewer bits than the bitmap representation. If more than 1 in 64 records have that value, the bitmap representation is preferable.

Thus, bitmaps can be used as a compressed storage mechanism at the leaf nodes of B$^+$-trees, for those values that occur very frequently.

12.10 Summary

- Many queries reference only a small proportion of the records in a file. To reduce the overhead in searching for these records, we can construct *indices* for the files that store the database.
- Index-sequential files are one of the oldest index schemes used in database systems. To permit fast retrieval of records in search-key order, records are stored sequentially, and out-of-order records are chained together. To allow fast random access, we use an index structure.
- There are two types of indices that we can use: dense indices and sparse indices. Dense indices contain entries for every search-key value, whereas sparse indices contain entries only for some search-key values.
- If the sort order of a search key matches the sort order of a relation, an index on the search key is called a *primary index*. The other indices are called *secondary indices*. Secondary indices improve the performance of queries that use search keys other than the primary one. However, they impose an overhead on modification of the database.

- The primary disadvantage of the index-sequential file organization is that performance degrades as the file grows. To overcome this deficiency, we can use a B^+*-tree index*.
- A B^+-tree index takes the form of a *balanced* tree, in which every path from the root of the tree to a leaf of the tree is of the same length. The height of a B^+-tree is proportional to the logarithm to the base N of the number of records in the relation, where each nonleaf node stores N pointers; the value of N is often around 50 or 100. B^+-trees are much shorter than other balanced binary-tree structures such as AVL trees, and therefore require fewer disk accesses to locate records.
- Lookup on B^+-trees is straightforward and efficient. Insertion and deletion, however, are somewhat more complicated, but still efficient. The number of operations required for lookup, insertion, and deletion on B^+-trees is proportional to the logarithm to the base N of the number of records in the relation, where each nonleaf node stores N pointers.
- We can use B^+-trees for indexing a file containing records, as well as to organize records into a file.
- B-tree indices are similar to B^+-tree indices. The primary advantage of a B-tree is that the B-tree eliminates the redundant storage of search-key values. The major disadvantages are overall complexity and reduced fanout for a given node size. System designers almost universally prefer B^+-tree indices over B-tree indices in practice.
- Sequential file organizations require an index structure to locate data. File organizations based on hashing, by contrast, allow us to find the address of a data item directly by computing a function on the search-key value of the desired record. Since we do not know at design time precisely which search-key values will be stored in the file, a good hash function to choose is one that assigns search-key values to buckets such that the distribution is both uniform and random.
- *Static hashing* uses hash functions in which the set of bucket addresses is fixed. Such hash functions cannot easily accommodate databases that grow significantly larger over time. There are several *dynamic hashing techniques* that allow the hash function to be modified. One example is *extendable hashing*, which copes with changes in database size by splitting and coalescing buckets as the database grows and shrinks.
- We can also use hashing to create secondary indices; such indices are called *hash indices*. For notational convenience, we assume hash file organizations have an implicit hash index on the search key used for hashing.
- Ordered indices such as B^+-trees and hash indices can be used for selections based on equality conditions involving single attributes. When multiple

attributes are involved in a selection condition, we can intersect record identifiers retrieved from multiple indices.

- Grid files provide a general means of indexing on multiple attributes.
- Bitmap indices provide a very compact representation for indexing attributes with very few distinct values. Intersection operations are extremely fast on bitmaps, making them ideal for supporting queries on multiple attributes.

Review Terms

- Access types
- Access time
- Insertion time
- Deletion time
- Space overhead
- Ordered index
- Primary index
- Clustering index
- Secondary index
- Nonclustering index
- Index-sequential file
- Index record/entry
- Dense index
- Sparse index
- Multilevel index
- Sequential scan
- B^+-Tree index
- Balanced tree
- B^+-Tree file organization
- B-Tree index
- Static hashing
- Hash file organization
- Hash index
- Bucket
- Hash function
- Bucket overflow
- Skew
- Closed hashing
- Dynamic hashing
- Extendable hashing
- Multiple-key access
- Indices on multiple keys
- Grid files
- Bitmap index
- Bitmap operations
  - ☐ Intersection
  - ☐ Union
  - ☐ Complement
  - ☐ Existence bitmap

Exercises

12.1 When is it preferable to use a dense index rather than a sparse index? Explain your answer.

12.2 Since indices speed query processing, why might they not be kept on several search keys? List as many reasons as possible.

12.3 What is the difference between a primary index and a secondary index?

12.4 Is it possible in general to have two primary indices on the same relation for different search keys? Explain your answer.

12.5 Construct a B^+-tree for the following set of key values:

(2, 3, 5, 7, 11, 17, 19, 23, 29, 31)

Assume that the tree is initially empty and values are added in ascending order. Construct B^+-trees for the cases where the number of pointers that will fit in one node is as follows:

a. Four
b. Six
c. Eight

12.6 For each B^+-tree of Exercise 12.5, show the steps involved in the following queries:

a. Find records with a search-key value of 11.
b. Find records with a search-key value between 7 and 17, inclusive.

12.7 For each B^+-tree of Exercise 12.5, show the form of the tree after each of the following series of operations:

a. Insert 9.
b. Insert 10.
c. Insert 8.
d. Delete 23.
e. Delete 19.

12.8 Consider the modified redistribution scheme for B^+-trees described in page 463. What is the expected height of the tree as a function of n?

12.9 Repeat Exercise 12.5 for a B-tree.

12.10 Explain the distinction between closed and open hashing. Discuss the relative merits of each technique in database applications.

12.11 What are the causes of bucket overflow in a hash file organization? What can be done to reduce the occurrence of bucket overflows?

12.12 Suppose that we are using extendable hashing on a file that contains records with the following search-key values:

2, 3, 5, 7, 11, 17, 19, 23, 29, 31

Show the extendable hash structure for this file if the hash function is $h(x) = x$ mod 8 and buckets can hold three records.

12.13 Show how the extendable hash structure of Exercise 12.12 changes as the result of each of the following steps:

a. Delete 11.
b. Delete 31.
c. Insert 1.
d. Insert 15.

12.14 Give pseudocode for deletion of entries from an extendable hash structure, including details of when and how to coalesce buckets. Do not bother about reducing the size of the bucket address table.

12.15 Suggest an efficient way to test if the bucket address table in extendable hashing can be reduced in size, by storing an extra count with the bucket address table. Give details of how the count should be maintained when buckets are split, coalesced or deleted.

(Note: Reducing the size of the bucket address table is an expensive operation, and subsequent inserts may cause the table to grow again. Therefore, it is best not to reduce the size as soon as it is possible to do so, but instead do it only if the number of index entries becomes small compared to the bucket address table size.)

12.16 Why is a hash structure not the best choice for a search key on which range queries are likely?

12.17 Consider a grid file in which we wish to avoid overflow buckets for performance reasons. In cases where an overflow bucket would be needed, we instead reorganize the grid file. Present an algorithm for such a reorganization.

12.18 Consider the *account* relation shown in Figure 12.25.

a. Construct a bitmap index on the attributes *branch-name* and *balance*, dividing *balance* values into 4 ranges: below 250, 250 to below 500, 500 to below 750, and 750 and above.

b. Consider a query that requests all accounts in Downtown with a balance of 500 or more. Outline the steps in answering the query, and show the final and intermediate bitmaps constructed to answer the query.

12.19 Show how to compute existence bitmaps from other bitmaps. Make sure that your technique works even in the presence of null values, by using a bitmap for the value *null*.

12.20 How does data encryption affect index schemes? In particular, how might it affect schemes that attempt to store data in sorted order?

Bibliographical Notes

Discussions of the basic data structures in indexing and hashing can be found in Cormen et al. [1990]. B-tree indices were first introduced in Bayer [1972] and Bayer and McCreight [1972]. B^+-trees are discussed in Comer [1979], Bayer and Unterauer [1977] and Knuth [1973]. The bibliographic notes in Chapter 16 provides references to research on allowing concurrent accesses and updates on B^+-trees. Gray and Reuter [1993] provide a good description of issues in the implementation of B^+-trees.

Several alternative tree and treelike search structures have been proposed. **Tries** are trees whose structure is based on the "digits" of keys (for example, a dictionary thumb index, which has one entry for each letter). Such trees may not be balanced in the sense that B^+-trees are. Tries are discussed by Ramesh et al. [1989], Orenstein

[1982], Litwin [1981] and Fredkin [1960]. Related work includes the digital B-trees of Lomet [1981].

Knuth [1973] analyzes a large number of different hashing techniques. Several dynamic hashing schemes exist. Extendable hashing was introduced by Fagin et al. [1979]. Linear hashing was introduced by Litwin [1978] and Litwin [1980]; Larson [1982] presents a performance analysis of linear hashing. Ellis [1987] examined concurrency with linear hashing. Larson [1988] presents a variant of linear hashing. Another scheme, called dynamic hashing, was proposed by Larson [1978]. An alternative given by Ramakrishna and Larson [1989] allows retrieval in a single disk access at the price of a high overhead for a small fraction of database modifications. Partitioned hashing is an extension of hashing to multiple attributes, and is covered in Rivest [1976], Burkhard [1976] and Burkhard [1979].

The grid file structure appears in Nievergelt et al. [1984] and Hinrichs [1985]. Bitmap indices, and variants called **bit-sliced indices** and **projection indices** are described in O'Neil and Quass [1997]. They were first introduced in the IBM Model 204 file manager on the AS 400 platform. They provide very large speedups on certain types of queries, and are today implemented on most database systems. Recent research on bitmap indices includes Wu and Buchmann [1998], Chan and Ioannidis [1998], Chan and Ioannidis [1999], and Johnson [1999a].