We can organize records on disk in many ways. The best organization for a file is highly dependent on how we wish to use the file. Do we want to access the entire file, or do we only want to access one record in the file, or do we want to access several selected records? If data is in multiple files, do we need to join the records together to obtain the relationships we wish? How often do we join compared to how often we access the files without joining? Answering these questions for a single application is complex, and it is even more complex when we have to balance the requirements of several applications simultaneously. Because it is usually impossible to find a single organization that is best for all applications, we have to settle for a reasonable balance among the competing demands. This reasonable balance is itself difficult to find. We simply do the best we can and make adjustments as necessary.

1 Sequential Files

One straightforward way to organize a file on disk is to do it sequentially. In a sequential file, we place records in a block one after another until there is no room for another complete record in the block. We then fill the next available block—often the one that comes physically next on the disk, but sometimes one in another location with a pointer linking the sequential blocks. We then fill the next available block and so on until all the records have been placed in logically consecutive blocks. If we wish to access all the records, one after another, this is a highly efficient file organization.

Suppose, however, that we only wish to find one record. We may have to search the entire file, reading one block at a time to find the record. On the average, we would have to read in half the blocks of the file. If we are searching for a record with a given primary key value and if the sequential file is sorted on the primary key, we can reduce the time by using a binary search on the blocks of a file. If we have an index on the primary key, we can do the search even faster. Furthermore, indexes can also speed up the search for a record based on values other than the primary key.

An index is an auxiliary structure for a file that consists of an ordered sequence of value-pointer pairs, or, more generally, an ordered sequence of value-tuple/pointer-list pairs. A value-pointer pair, for example, might be a customer-ID value and a disk-address pointer that points to the block that includes the customer-ID value. A sequence of these pairs ordered by customer ID would be an index. More generally, the value can be a value-tuple because we may be searching on a composite key such as Name, Address or we may be searching on several attribute values such as Customer ID and Date to find an order. Since a value-tuple is also a value, we need not and do not distinguish these two cases in our discussion. The pointer part is more generally a pointer list because there may be more than one record in more than one block that contains the value of interest. As a specific example, we could have an index on Customer ID for our Order relation, which we discussed in Figure 2.10 in the last chapter. This would help us quickly find all orders placed by a customer even though these orders may be in several different blocks.

The general idea of an index is simple, but there are a number of factors to consider. One is whether the index itself is stored on disk or in main memory. If it is on disk, we must consider the number of disk accesses required to read the index into main memory. The number of disk accesses depends on how large the index is. If it is particularly large, we may organize it as a two- or three-level, or in general as an $n$-level, index tree so that we need not read all the blocks of the index to find the index value we are seeking. Pointers in the upper-level blocks of an index tree point to index blocks, and pointers in the leaf-level blocks point to file blocks.

When the values in the index pairs are primary keys, we call an index a primary-key index or sometimes just a primary index. When the values are not primary keys, we call an index a
secondary-key index or often just a secondary index. (Note that secondary keys may not be keys in the sense that they uniquely identify a tuple, but instead are merely keys used for accessing records.) If the file is sorted, it is almost always sorted on its primary key. Indeed, if it is sorted on any other key, for all intents and purposes we may as well consider it to be the primary key. Files sorted on a nonkey value are rare. If a file is sorted on a key, the index can be sparse, which means that it has a value-pointer pair for only one record in each block of the file. An index that has pointers for every record is called a dense index.

A sequential file sorted on its primary key, along with a sparse index on the primary key, is called an indexed sequential file. Figure 1 shows an example of an indexed sequential file organization for our Customer relation. The index holds the CustomerID value of the first record in each block. We can use the index to find a record with a given customer ID—say b12—as follows. First, we do a binary search on the index to find that b12 lies between a92 and b22. The record, if it exists, must therefore be in the second block. Hence, we make one access to the disk to retrieve the second block and then do a binary search on the records of the second block to retrieve the record. If we fail to find it, we need not look elsewhere, and we can report that the record does not exist.

Having an index on a file does not preclude us from having another index on the same file. We could, for example, also have an index on discount values for the file in Figure 1. This index would be a secondary index and must necessarily be dense. A value-pointer-list pair in this index would consist of one of the discount values v and a list of pointers to blocks containing any record with a v discount value. It is possible that every pointer list in this index could point to every block in the file. In this case, the index is not very useful. A secondary index on Address is also possible, and, unlike the index for discount values, would likely have a small number of pointers in its pointer list. In general, we can index every attribute and every combination of attributes, but it is not always wise to do so.
If we only wish to retrieve records, sequential files with indexes serve quite well. Updates, however, lead to several problems and potential pitfalls, which are often severe enough to cause us to use different file organizations.

Suppose we wish to delete a record. Although we are generally interested in reclaiming unused space, we should not reclaim the space by going through all blocks of a file and moving each record up by one. Not only is this slow because it accesses all blocks beyond the deletion, but it is also slow because it requires changing some (possibly all) pointers in all indexes on the file. A record is said to be pinned to a block if there is a pointer to the block for the record. Moving a pinned record out of a block requires an update of all pointers that reference the record. Thus, instead of reclaiming space in this way, we mark the record deleted, usually by setting a delete bit specifically set aside for this purpose.

Insertions are even more problematic than deletions. Suppose we wish to insert a record into an indexed sequential file. We can use the index to find the block in which we should place the record. If we are lucky, there will be enough space for the record either because at least one record has its delete bit set, or because we initially underfilled the blocks anticipating that we would later need the space. Even if the space is not in exactly the right place we can reorder the records in a block so that we can retain the sort order.

Sometimes, however, we will not be lucky, and there will be no space. In this case, we can use an overflow bucket, which consists of one or more blocks that contain all the overflow records of the indexed sequential file. Since we obtain these blocks for the overflow bucket as needed, we have no guarantee that they will be physically next to each other, and it is best to assume that they are scattered on disk. We can keep the records within the regular blocks of an indexed sequential file sorted by always bumping the last record in a block to the overflow bucket. Records in the overflow bucket, however, are not sorted. Attempting to keep them sorted could be time consuming because the overflow bucket may be several blocks long and randomly scattered on disk. When there is an overflow bucket, we might not find a record in the regular block in which it should be found. In this case, we search the overflow bucket sequentially, which may cause as many disk accesses as blocks in the bucket.

An overflow bucket can become long with respect to the size of an indexed sequential file. The longer an overflow bucket becomes, the more an indexed sequential file degenerates into an unordered file. Periodic reorganization, which rebuilds the indexed sequential file from scratch, may be necessary to keep the file efficient. This can be expensive because it not only requires that all the data be reorganized, but also requires that the primary index and any secondary indexes also be reorganized. For some applications, there may be times when the database is not in use. These times can be used for file reorganization. For other applications, however, it may be hard, if not impossible, to find convenient times to do the reorganization.

Before discussing some alternatives that alleviate these problems with indexed sequential files, we present one more idea. Record modification at first glance might not seem like much of a problem. Indeed, it is not a problem if the records are fixed in length, because then we can obtain the block that contains the record, change the field value, and rewrite the block. Since no space is added or deleted for fixed-length records, this always works.

Variable-length records can be useful in an application when similar objects relate to a non-uniform set of data. For example, a representation of persons in an application may have many attributes in common such as name, address, identification number, and so forth, but the representation may also have attributes such as college degrees, kinds of job skills, and hobbies that apply variously to different persons. An even more common use of variable-length records is for objects that each have an unspecified number of repeating groups of attribute values. In the application we discussed in chapter 2, for example, we might make a single record that would include all the information for an order. Instead of storing the information for Order, Item, and OrderInfo in Figure 2.10 in three separate files, we might have one file as Figure 2 shows.

In Figure 2, each order includes a set of nested tuples that represent the items ordered. For this organization, if we are interested in looking up a particular order, all the information resides
Together on disk. If we want the same information, but store it in three different files, we would have to look up the information in each file and join it together. If, on the other hand, we want to update an order by adding another item, we would likely be unable to easily add the information for either organization.

In general, we store variable-length records by reserving enough space to accommodate the variable part of any variable-length record of interest, or by chaining a variable number of fixed-length components together, or by a combination of these two techniques. In a combination technique, for our Order example in Figure 2, we could reserve space for $n$ items, where $n$ is reasonably small but is large enough to hold the number of items expected for most orders. The rest of the items could be placed sequentially at the end of an overflow bucket with a pointer from the initial record to the record in the overflow bucket where the list begins. We would thus be able to retrieve most orders with a one disk access, but for some we would need a second disk access and possibly more if the list were to span more than one overflow block.

2 Hashing

If we wish to find a single record given its primary key value, a good hashing technique provides the fastest way to retrieve it. Usually, only one disk access is necessary. Moreover, there is no auxiliary structure, like an index structure, and thus no need to worry about whether the structure is in memory or on disk and how much time it takes to access the auxiliary structure. Hashing also solves the problem of record insertion and deletion.

Closed static hashing, which we discuss here, is a standard hashing technique that can be used for database systems. Open hashing and dynamic hashing are more common, but beyond what we introduce here. For closed static hashing we reserve a fixed amount of space sufficient to hold at least as many records as we anticipate being in the file at any one time. We require that the file consists of fixed-length records, with a designated primary key. For simplicity of discussion, we ignore blocks and block boundaries, but it is not difficult to convert record offsets within the fixed space to block addresses.

Let $N$ be the maximum number of records that fit in the reserved space. Let $K$ be the set of possible key values. A hash function $h$ maps $K$ to $\{0 \ldots N-1\}$ (usually, $|K| \gg N$). If the fixed-length records each have $b$ bytes, then to insert a record $r$ whose primary key value is $k$, we compute $h(k)$ and try to place $r$ at location $bh(k)$ in the file. If no other record is already at location $bh(k)$, we place $r$ at this location. Otherwise, we search forward through the file, wrapping around to the beginning if necessary, until we find the next available record location, and we insert $r$ there. To retrieve a record $r$, which is known to be in the file and whose primary key value is $k$, we look first in location $bh(k)$. If it is not there, we search forward through the file.
To delete a record $r$, which is known to be in the file, we locate it and then mark it deleted.

In a hash file, every record location is either occupied by a record or is available, either because it contains a deleted record or because it has never held a record. We insert records in any available location, as just described. If we wish to locate a record for a given primary key value $k$, however, and if we are not sure whether a record for $k$ is in the file, we distinguish between locations with deleted records and locations that have never held a record. If in our forward search we encounter a location that has never held a record, we can stop and be sure that the record is not in the file.

Several factors determine how well closed static hashing works. Our chosen hash function $h$ should uniformly distribute the records over the space. If the hash function is perfect, so that there are never any collisions, we can always store and retrieve a record with at most one disk access. It is highly unlikely, however, that we can find a perfect hashing function. Usually, the number of key values far exceeds the maximum number of records $N$, and many key values map to each value between 0 and $N - 1$. Nevertheless, if we choose a good hash function that randomly and uniformly distributes the set of possible key values over $\{0 \ldots N - 1\}$, we can minimize the number of collisions.

Closed static hashing also depends on whether we can initially find enough contiguous available space and on how full we fill the space. If we cannot find enough contiguous space, we should use open hashing, which lets us use disk blocks that are not necessarily contiguous. If we completely fill the space, closed static hashing breaks down. Although we could provide an overflow bucket, this would cause a major degradation in performance. If we are likely to run into this problem because we cannot accurately estimate the expected maximum number of resident records, we should use dynamic hashing, which allows us to dynamically change how much space we use.

The simple forward search that we have described here is not always the best way to resolve collisions. However, it works well for database systems whose files are in secondary storage. This is because we always read a block of data, not just a single record. If the hash function is good and the fill factor is not too high, there is a good chance that the record will be in the block accessed even when there is a collision. When this happens, there is still only one disk access. Furthermore, if the record is beyond the end of the block, it is most likely in the physically next block on disk. Empirical studies have shown that if a hash function is well chosen and the fill factor is reasonable (about 75% or more), we can expect to retrieve a randomly chosen record with one disk access.

It is hard to beat hashing if we want to store and retrieve single records based only on their primary key. Unfortunately, application requirements are not usually so simplistic. Queries that require accessing a file by nonkey attribute values or by a range of values are not well suited for hashing. If we try to hash on a nonkey attribute, every record with the same value for the attribute hashes to the same location. This can cause our collision rate to skyrocket. For range queries, such as a query that requires us to find all the orders between January 1 and March 31, we would have to hash on every value in the range, even if there is no record with the date. This could drive us through nearly every block in the file multiple times. It would be far better to just search the entire hash file directly from beginning to end.

Besides individual queries that are not well suited to hashing, we often need to pose several different queries on the same file. This may cause us to access a file in many different ways, by many different attribute values or by many different combinations of attribute values. Here again, hashing is not helpful, because we can only hash a file in one way.

Indexing is better for all these situations. We can use secondary indexes for nonkey attributes. We can use a subtree in an index tree, rather than just a path in an index tree to handle range queries, and we can have multiple indexes on a file to satisfy different access needs. As discussed earlier, however, indexed sequential files are problematic, especially for highly dynamic files. We therefore need a better way to do our indexing. We discuss one of the possibilities next.
Figure 3: A $B^+$-tree index and data file.

3 $B^+$-tree Indexes

B-trees are $n$-way, balanced trees whose nonroot nodes are always at least half full. Unlike indexed sequential files, B-trees indefinitely maintain good retrieval, insertion, and deletion times and therefore never require reorganization from scratch. For files with frequent insertions and deletions, these advantages easily compensate for the overhead incurred by insertion and deletion operations plus the overhead of added space.

There are a number of B-tree variations. We only present the most common variation here. The variation we present here is a $B^+$-tree index for single-attribute keys over a file of unordered data records.

Figure 3 shows an example of a $B^+$-tree index on $ItemNr$ for our sample $Item$ file. To keep the example small, we let $n$, the number of pointers per node in the index, be 4 and the number of data records in a block of the data file be 5. (Since $B^+$-tree nodes are usually disk blocks, a more typical value for $n$ would be twenty to a hundred or so, and more than 5 item records would likely fit in a typical block.) Observe, in Figure 3, that the tree is balanced—every path from root to leaf has length 2. Observe also that each node in the tree has $n = 4$ pointers and $n - 1 = 3$ key values. For leaf nodes, we define “half full” in terms of values and require that each leaf node have at least $\lceil (n - 1)/2 \rceil$ values—every leaf node in Figure 3 has 2 or 3 values. For nonleaf nodes we define “half full” in terms of pointers and require that each nonleaf node have at least $\lceil n/2 \rceil$ pointers—every nonleaf node in Figure 3 has 2 or more pointers. The root node can be an exception and have fewer than $\lceil n/2 \rceil$ pointers—every nonleaf node in Figure 3 has 2 or more pointers. The root node can be an exception and have fewer than $\lceil n/2 \rceil$ pointers. When $n = 4$, of course, this is not possible, but if $n$ were 5, we could still have only one key value and 2 ($< \lceil 5/2 \rceil$) pointers in the root.

The leaf nodes and the nonleaf nodes in a $B^+$-tree index look similar, but they are not the same. The leaf nodes are a dense index for the data file, while the nonleaf nodes are a sparse index for the key values in the leaf nodes. The leaf nodes contain all the key values, sorted in ascending order. The pointer preceding a key value $k$ in a leaf node points to the block where the record whose key is $k$ resides. The last pointer in each node points to the next leaf node. Even though the records in the data file are unordered, this organization lets us access the entire
file in ascending order on the key by traversing the leaf nodes in order and following the record
pointers to the blocks where each record resides. Using a slight variation, we could speed up the
sequential search within a block by maintaining the records in sorted order within the block or by
maintaining a block-pointer and offset rather than just a block pointer in the dense index. The
trade-offs here are minimal and either are or are not worthwhile depending on the kinds of access
that are typical for the application.

The nonleaf nodes also contain key values sorted in ascending order, but not every key value.
A key value \( k \) in a nonleaf node is the smallest key value in the subtree whose root is pointed to by
the pointer succeeding \( k \). In Figure 3, for example, 85 is the key value in the root. Its succeeding
pointers point to the subtree with 90 and 92 as its key values. Lower down in this subtree, we see
the key value 85, which is the smallest key value in the entire subtree rooted at the node with 90
and 92 as its key values. The succeeding pointer of 90 points to the node (degenerate subtree)
whose smallest key value is 90, and the succeeding pointer of 92 points to the node whose smallest
key value is 92. The initial pointer of each nonleaf node points to the subtree whose values are all
less than the first key value. Thus, for every key value \( k \) in a nonleaf node, we can find all the key
values less than \( k \) by looking in the left subtree of \( k \) and all the key values greater than or equal
to \( k \) by looking in the right subtree of \( k \).

We search for a record using a B\(^+\)-tree index in a straightforward way. Suppose, for example,
that we wish to find the record whose ItemNr value is 91. Starting at the root, we see that
91 \( \geq \) 85. We thus follow the right pointer to the right subtree, whose key values are 90 and 92.
Since 91 falls between 90 and 92, we follow the pointer between 90 and 92 to find the leaf node
containing 91. We then follow the left pointer of 91 to the block containing the record for which
we are searching, read the block from disk, and search it sequentially until we find the record
whose key value is 91. Note that if we are searching for a record that does not exist, such as for
the record whose item number is 7, we can know for sure that it does not exist when we arrive at
the leaf node that should contain 7.

If the index is itself on disk, each node is (usually) in a separate disk block. Since we only
have to search at most one path in the B\(^+\)-tree to locate a record, we only have to access at most
\( \log_{\lceil n/2 \rceil} N \) \( n \)-way B\(^+\)-tree disk blocks, where \( N \) is the number of key values in the file. If we have
\( n = 100 \), for example, we could index at least 5,000 records and up to 990,000 records with a
3-level B\(^+\)-tree. Often we keep the root or the upper two levels of a B\(^+\)-tree in main memory.
This, of course, can further reduce the time to search a B\(^+\)-tree index, and can make access times
for B\(^+\)-tree comparable to access times for hashing.

Insertion and deletion algorithms for B\(^+\)-tree indexes are interesting because we must maintain
the balance in the B\(^+\)-tree. Some insertions and deletions are straightforward. If there is space in
a leaf node for the key value of a record to be inserted, we basically just drop the value in place
in the node and put the record anywhere there is space in the data file or in a new block of the
data file, if necessary. For example, if we insert a record for a new item whose item number is 37
in the B\(^+\)-tree organization in Figure 3, we see that the 37 and a pointer can go in the open space
in the 31-35 node, and that the record could go in the second block of the data file. If the key
value is 33, we do almost the same, except that we move the 35 and its pointer to the open space
and insert the 33 between the 31 and 35. If the key value must be the first value in a leaf node,
we insert it after moving all the others down one, and recursively propagate its value up the tree
as far as is necessary. Inserting 29, for example, in the 31-35 node changes the 31 in the 31-40
node to also be 29.

Deletion is much the same, so long as there are sufficient key values left in the leaf node to
maintain the requirement of the node being at least half full. Suppose we delete the record with
key value 61, for example. We first locate the record in the data file and mark it deleted. (By
the way, we also update our space management information to reflect the fact that there is now
more space in the data block where the 61 record was.) We then move the 70 and its pointer left
one slot to the space previously occupied by 61. (It should also be clear that we need to maintain
some information in each node that states how many key values are actually stored in the node,
Here, for example, we would reduce the count from 3 to 2, so that we know that any values in spaces beyond the count are garbage.

Insertion and deletion are far more complex when we need to insert into an already full node or delete from a node that is just half full. Since a complete discussion of these ideas is beyond our purpose here, we provide only a basic explanation and some examples. One important note about both insertion and deletion is that the worst-case time complexity for these operations is proportional to $\log \left(\frac{n}{2}\right)_N$, where $N$ is the number of key values in the file, and $n$ is the number of pointers in a node. This is because we travel down at most once to the one leaf where the insertion or deletion is to take place, and we travel up along the same path at most once, making adjustments along the way.

The basic idea for insertion into a full node is to split it into two nodes, each half full, and add a new key value into the parent. Of course, if the parent is also full, we need to split it and add a new key into its parent, and so on. If we need to insert into a full root, we split it and create a new root. In this special case, we also increase the height of the B$^+$-tree. Figure 4 shows what happens to the B$^+$-tree in Figure 3 when we insert 15 (Figure 4a) and then 47 (Figure 4b). Promoting key values from leaf nodes and from nonleaf nodes is different. When we promote a value from a leaf node, we leave a copy in the leaf; but when we promote a value from a nonleaf node, we do not leave a copy in the nonleaf node. In Figure 4a, we see that the 15, which was promoted from a leaf node, was copied into the proper nonleaf node, whereas the 40, which was promoted from a nonleaf node in Figure 4b, was moved to the root. Note, by the way, that the nonleaf node containing only 61 is half full, as required, because it has $\lceil n/2 \rceil$ pointers ($= 2$ for our example where $n = 4$).

The basic idea for deletion from a node that would become less than half full by a simple deletion is to combine siblings and, if necessary, redistribute values. Figure 5 shows what happens...
to the B⁺-tree in Figure 4b when we delete 23 (Figure 5a) and then 47 (Figure 5b). A simple deletion of 23 leaves the 15-23 leaf node with only one value, too few for a leaf node. Since either sibling has sufficient space, we choose one, the left sibling here, and move the 15 to the 11-12 node and discard the old 15-23 node. This, of course, requires an adjustment to the parent of the discarded node. Since there no longer is a node with 15 as its first element, we discard the 15 and its pointer and shift everything else in the node one place to the left. The result is Figure 5a. Continuing, we now delete 47, which makes the 40-47 node have too few values. Since we can combine with the sibling on the right, we do. This, however, now makes it so that the 61 is no longer the first value of any node. We therefore delete the 61 in the parent node, which causes this node to have too few values. We therefore try to combine the parent node with too few values with a sibling. Either sibling has sufficient space. Combining with the left sibling causes the 40 to be deleted from the root and placed in the combined node. After making the necessary adjustments to the root, we end up with the B⁺-tree in Figure 5b.

To show that redistribution of values is sometimes necessary, consider deleting 90 from the B⁺-tree in Figure 5b. Since neither node to the left or right of the 90-91 node has space for the remaining 91 value, we cannot combine nodes. Instead, we move some of the values from either sibling into the empty space in the 90-91 node. In particular, we move the 88 into the node and make the necessary adjustments to the parent. Figure 6 shows the result.
Figure 6: Deletion of 90 from the $B^+$-tree in Figure 5b causing redistribution.