MODULE 3

Physical Data Organization

****

SYLLABUS

SQL DML (Data Manipulation Language), Physical Data Organization

SQL DML (Data Manipulation Language) - SQL queries on single and multiple tables, Nested queries (correlated and non-correlated), Aggregation and grouping, Views, assertions, Triggers, SQL data types.

Physical Data Organization - Review of terms: physical and logical records, blocking factor, pinned and unpinned organization. Heap files, Indexing, Singe level indices, numerical examples, Multi-level-indices, numerical examples, B-Trees & B+-Trees (structure only, algorithms not required), Extendible Hashing, Indexing on multiple keys – grid files

****

Physical Files

Physical files contain the actual data that is stored on the system, and a description of how data is to be presented to or received from a program. They contain only one record format, and one or more members. Records in database files can be externally or program-described.

A physical file can have a keyed sequence access path. This means that data is presented to a program in a sequence based on one or more key fields in the file.

Logical files

Logical files do not contain data. They contain a description of records found in one or more physical files. A logical file is a view or representation of one or more physical files. Logical files that contain more than one format are referred to as multi-format logical files. If your program processes a logical file which contains more than one record format, you can use a read by record format to set the format you wish to use.

Physical File	Logical File
It occupies the portion of memory. It contains the original data.	It does not occupy memory space. It does not contain data.
A physical file contains one record format.	It can contain upto 32 record formats.
It can exist without logical file.	It cannot exist without physical file.
If there is a logical file for physical file, the physical file cannot be deleted until and unless we delete the logical file.	If there is a logical file for a physical file, the logical file can be deleted without deleting the physical file.

Typical database applications need only a small portion of the database at a time for processing. Whenever a certain portion of the data is needed, it must be located on disk, copied to main memory for processing, and then rewritten to the disk if the data is changed. The data stored on disk is organized as files of records. Each record is a collection of data values that can be interpreted as facts about entities, their attributes, and their relationships. Records should be stored on disk in a manner that makes it possible to locate them efficiently when they are needed

File organizations

File organization refers to the organization of the data of a file into records, blocks, and access structures; this includes the way records and blocks are placed on the storage medium and interlinked. An access method, on the other hand, provides a group of operations that can be applied to a file. In general, it is possible to apply several access methods to a file organization

Primary file organizations

Primary file organizations determines how the file records are physically placed on the disk, and hence how the records can be accessed.

1. Heap file (or unordered file) places the records on disk in no particular order by appending new records at the end of the file
2. Sorted file (or sequential file) keeps the records ordered by the value of a particular field (called the sort key).
3. Hashed file uses a hash function applied to a particular field (called the hash key) to determine a record's placement on disk.
4. Other primary file organizations, such as B-trees, use tree structures

Secondary organization

A secondary organization or auxiliary access structure allows efficient access to file records based on alternate fields than those that have been used for the primary file organization

Records and Record Types

Data is usually stored in the form of records. Each record consists of a collection of related data values or items, where each value is formed of one or more bytes and corresponds to a particular field of the record. Records usually describe entities and their attributes.

A collection of field names and their corresponding data types constitutes a record type or record format definition. A data type, associated with each field, specifies the types of values a field can take

Files, Fixed-Length Records, and Variable-Length Records

A file is a sequence of records. In many cases, all records in a file are of the same record type.

If every record in the file has exactly the same size (in bytes), the file is said to be made up of fixed-length records. If different records in the file have different sizes, the file is said to be made up of variable-length records.

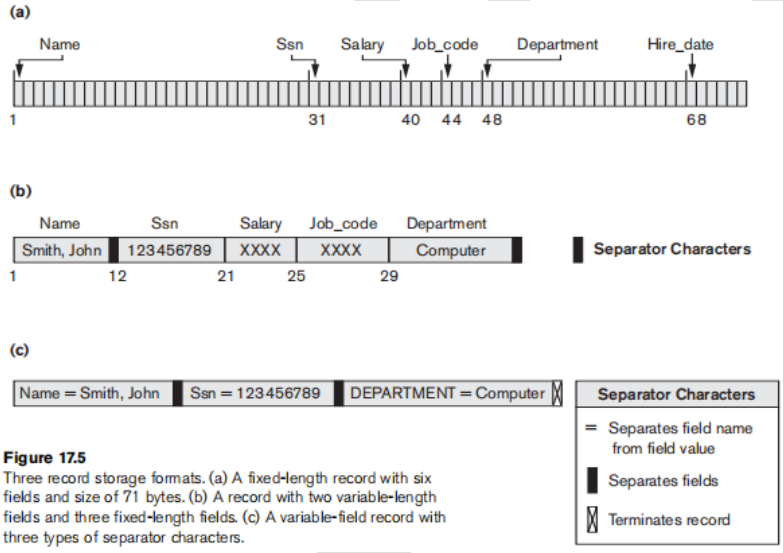


Figure 17.5
Three record storage formats. (a) A fixed-length record with six fields and size of 71 bytes. (b) A record with two variable-length fields and three fixed-length fields. (c) A variable-field record with three types of separator characters.

Spanned and Unspanned Organization

A block is the unit of data transfer between disk and memory. When the block size is larger than the record size, each block will contain numerous records, although some files may have unusually large records that cannot fit in one block.

Part of the record can be stored on one block and the rest on another. A pointer at the end of the first block points to the block containing the remainder of the record. This organization is called

spanned because records can span more than one block. Whenever a record is larger than a block, we must use a spanned organization.If records are not allowed to cross block boundaries, the organization is called unspanned.

Blocking factor for the file

The records of a file must be allocated to disk blocks because a block is the unit of data transfer between disk and memory. When the block size is larger than the record size, each block will contain numerous records, although some files may have unusually large records that cannot fit in one block. Suppose that the block size is B bytes. For a file of fixed-length records of size R bytes, with $B \geq R$, we can fit $bfr = \lfloor B / R \rfloor$ records per block. The value bfr is called the blocking factor for the file.

In general, R may not divide B exactly, so we have some unused space in each block equal to

$B - (bfr * R)$ bytes

To utilize this unused space, we can store part of a record on one block and the rest on another. A pointer at the end of the first block points to the block containing the remainder of the record in case it is not the next consecutive block on disk. This organization is called spanned because records can span more than one block. Whenever a record is larger than a block, we must use a spanned organization. If records are not allowed to cross block boundaries, the organization is called unspanned. If the average record is large, it is advantageous to use spanning to reduce the lost space in each block

For variable-length records using spanned organization, each block may store a different number of records. In this case, the blocking factor bfr represents the average number of records per block for the file. We can use bfr to calculate the number of blocks b needed for a file of r records:

$b = \lceil (r/bfr) \rceil$ blocks

where the $\lceil (x) \rceil$ (ceiling function) rounds the value x up to the next integer.

Allocating File Blocks on Disk

- Contiguous allocation,
  - the file blocks are allocated to consecutive disk blocks.
  - This makes reading the whole file very fast using double buffering, but it makes expand_x0002_ing the file difficult.
- Linked allocation
  - each file block contains a pointer to the next file block.

- This makes it easy to expand the file but makes it slow to read the whole file.

- Clusters allocation
  - A combination of the two allocates clusters of consecutive disk blocks, and the clusters are linked.
  - Clusters are sometimes called file segments or extents.
- Indexed allocation
  - where one or more index blocks contain pointers to the actual file blocks

File Headers

A file header or file descriptor contains information about a file that is needed by the system programs that access the file records. The header includes information to determine the disk addresses of the file blocks as well as to record format descriptions, which may include field lengths and the order of fields within a record for fixed-length unspanned records and field type codes, separator characters, and record type codes for variable-length records.

To search for a record on disk, one or more blocks are copied into main memory buffers. Programs then search for the desired record or records within the buffers, using the information in the file header. If the address of the block that contains the desired record is not known, the search programs must do a linear search through the file blocks.

Each file block is copied into a buffer and searched until the record is located or all the file blocks have been searched unsuccessfully. This can be very time-consuming for a large file. The goal of a good file organization is to locate the block that contains a desired record with a minimal number of block transfers.

Operation on Files

Typical file operations include:

- OPEN: Readies the file for access, and associates a pointer that will refer to a current file record at each point in time.
- FIND: Searches for the first file record that satisfies a certain condition, and makes it the current file record.
- FINDNEXT: Searches for the next file record (from the current record) that satisfies a certain condition, and makes it the current file record.
- READ: Reads the current file record into a program variable.
- INSERT: Inserts a new record into the file & makes it the current file record.

- DELETE: Removes the current file record from the file, usually by marking the record to indicate that it is no longer valid.
- MODIFY: Changes the values of some fields of the current file record.
- CLOSE: Terminates access to the file.
- REORGANIZE: Reorganizes the file records.
- For example, the records marked deleted are physically removed from the file or a new organization of the file records is created.
- READ_ORDERED: Read the file blocks in order of a specific field of the file.

Files of Unordered Records (Heap Files)

Records are placed in the file in the order in which they are inserted, so new records are inserted at the end of the file. This organization is often used with additional access paths, such as the secondary indexes

Inserting a new record is very efficient. The last disk block of the file is copied into a buffer, the new record is added, and the block is then rewritten back to disk. The address of the last file block is kept in the file header.

Searching a record using any search condition involves a linear search through the file block by block an expensive procedure. If only one record satisfies the search condition, then, on the average, a program will read into memory and search half the file blocks before it finds the record. For a file of b blocks, this requires searching (b/2) blocks, on average. If no records or several records satisfy the search condition, the program must read and search all b blocks in the file

In case of deletion a program must first find its block, copy the block into a buffer, delete the record from the buffer, and finally rewrite the block back to the disk. This leaves unused space in the disk block. Deleting a large number of records in this way results in wasted storage space.

Another technique used for record deletion is to have an extra byte or bit, called a deletion marker, stored with each record. A record is deleted by setting the deletion marker to a certain value. A different value for the marker indicates a valid (not deleted) record. Search programs consider only valid records in a block when conducting their search. Both of these deletion techniques require periodic reorganization of the file to reclaim the unused space of deleted records

We can use either spanned or unspanned organization for an unordered file, and it may be used with either fixed-length or variable-length records. Modifying a variable-length record may require deleting the old record and inserting a modified record because the modified record may

not fit in its old space on disk. For a file of unordered fixed-length records using unspanned blocks and contiguous allocation, it is straightforward to access any record by its position in the file.

If the file records are numbered 0, 1, 2, ..., r − 1 and the records in each block are numbered 0, 1, ..., bfr − 1, where bfr is the blocking factor, then the ith record of the file is located in block ⌊(i/bfr)⌋ and is the (i mod bfr)th record in that block. Such a file is often called a relative or direct file because records can easily be accessed directly by their relative positions. Accessing a record by its position does not help locate a record based on a search condition; however, it facilitates the construction of access paths on the file, such as the indexes

Files of Ordered Records (Sorted Files)

We can physically order the records of a file on disk based on the values of one of their fields called the ordering field. This leads to an ordered or sequential file. If the ordering field is also a key field of the file a field guaranteed to have a unique value in each record then the field is called the ordering key for the file.

Figure 17.7
Some blocks of an ordered (sequential) file of EMPLOYEE records with Name as the ordering key field.

	Name	Ssn	Birth_date	Job	Salary	Sex
Block 1	Aaron, Ed					
	Abbott, Diane					
	⋮					
	Acosta, Marc					
Block 2	Adams, John					
	Adams, Robin					
	⋮					
	Akers, Jan					
Block 3	Alexander, Ed					
	Alfred, Bob					
	⋮					
	Allen, Sam					
Block 4	Allen, Troy					
	Anders, Keith					
	⋮					
	Anderson, Rob					
Block 5	Anderson, Zach					
	Angeli, Joe					
	⋮					
	Archer, Sue					

Ordered records advantages over unordered files

First, reading the records in order of the ordering key values becomes extremely efficient because no sorting is required. Second, finding the next record from the current one in order of the ordering key usually requires no additional block accesses because the next record is in the same block as the current one. Third, using a search condition based on the value of an ordering key field results in faster access when the binary search technique is used, which constitutes an

improvement over linear searches, although it is not often used for disk files. Ordered files are blocked and stored on contiguous cylinders to minimize the seek time

A binary search for disk files can be done on the blocks rather than on the records.

Suppose that the file has b blocks numbered 1, 2, ..., b; the records are ordered by ascending value of their ordering key field; and we are searching for a record whose ordering key field value is K.

A binary search usually accesses log2(b) blocks, whether the record is found or not an improvement over linear searches, where, on the average, (b/2) blocks are accessed when the record is found and b blocks are accessed when the record is not found. Ordering does not provide any advantages for random or ordered access of the records based on values of the other nonordering fields of the file. In these cases, we do a linear search for random access. To access the records in order based on a nonordering field, it is necessary to create another sorted copy in a different order of the file

Inserting and deleting records are expensive operations for an ordered file because the records must remain physically ordered.

To insert a record, we must find its correct position in the file, based on its ordering field value, and then make space in the file to insert the record in that position. For a large file this can be very time consuming because, on the average, half the records of the file must be moved to make space for the new record. This means that half the file blocks must be read and rewritten after records are moved among them. For record deletion, the problem is less severe if deletion markers and periodic reorganization are used.

Average Access Times

The following table shows the average access time to access a specific record for a given type of file

TABLE 13.2 AVERAGE ACCESS TIMES FOR BASIC FILE ORGANIZATIONS

TYPE OF ORGANIZATION	ACCESS/SEARCH METHOD	AVERAGE TIME TO ACCESS A SPECIFIC RECORD
Heap (Unordered)	Sequential scan (Linear Search)	$b/2$
Ordered	Sequential scan	$b/2$
Ordered	Binary Search	$\log_2 b$

Hashing Techniques

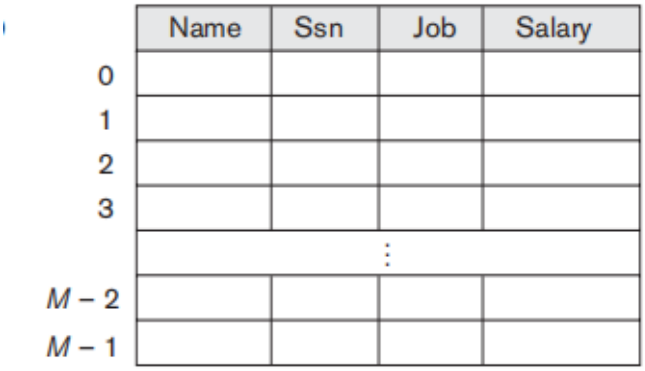
The search condition must be an equality condition on a single field, called the hash field.

In most cases, the hash field is also a key field of the file, in which case it is called the hash key. The idea behind hashing is to provide a function h, called a hash function or randomizing function, which is applied to the hash field value of a record and yields the address of the disk block in which the record is stored.

A search for the record within the block can be carried out in a main memory buffer. For most records, we need only a single-block access to retrieve that record.

Internal Hashing

For internal files, hashing is typically implemented as a hash table through the use of an array of records. Suppose that the array index range is from 0 to M – 1, ; then we have M slots whose addresses correspond to the array indexes. We choose a hash function that transforms the hash field value into an integer between 0 and M − 1. One common hash function is the h(K) = K mod M function, which returns the remainder of an integer hash field value K after division by M; this value is then used for the record address.



Other hashing functions available are

1. Folding
   - involves applying an arithmetic function such as addition or a logical function such as exclusive or to different portions of the hash field value to calculate the hash address (for example, with an address space from 0 to 999 to store 1,000 keys, a 6-digit key 235469 may be folded and stored at the address: (235+964) mod 1000 = 199).
2. picking some digits of the hash field value
   - for instance, the third, fifth, and eighth digits—to form the hash address (for example, storing 1,000 employees with Social Security numbers of 10 digits into a hash file with

1,000 positions would give the Social Security number 301-67-8923 a hash value of 172 by this hash function)

The problem with most hashing functions is that they do not guarantee that distinct values will hash to distinct addresses, because the hash field space the number of possible values a hash field can take is usually much larger than the address space the number of available addresses for records. The hashing function maps the hash field space to the address space

A collision occurs when the hash field value of a record that is being inserted hashes to an address that already contains a different record. In this situation, we must insert the new record in some other position, since its hash address is occupied. The process of finding another position is called collision resolution. There are numerous methods for collision resolution, including the following:

1. Open addressing. Proceeding from the occupied position specified by the hash address, the program checks the subsequent positions in order until an unused (empty) position is found.
2. Chaining. For this method, various overflow locations are kept, usually by extending the array with a number of overflow positions. Additionally, a pointer field is added to each record location. A collision is resolved by placing the new record in an unused overflow location and setting the pointer of the occupied hash address location to the address of that overflow location.
3. Multiple hashing. The program applies a second hash function if the first results in a collision. If another collision results, the program uses open addressing or applies a third hash function and then uses open addressing if necessary

The goal of a good hashing function is to distribute the records uniformly over the address space so as to minimize collisions while not leaving many unused locations. Simulation and analysis studies have shown that it is usually best to keep a hash table between 70 and 90 percent full so that the number of collisions remains low and we do not waste too much space. Hence, if we expect to have r records to store in the table, we should choose M locations for the address space such that (r/M) is between 0.7 and 0.9. It may also be useful to choose a prime number for M, since it has been demonstrated that this distributes the hash addresses better over the address space when the mod hashing function is used. Other hash functions may require M to be a power of 2.

External Hashing for Disk Files

Hashing for disk files is called external hashing. To suit the characteristics of disk storage, the target address space is made of buckets, each of which holds multiple records. A bucket is either one disk block or a cluster of contiguous disk blocks. The hashing function maps a key into a relative bucket number, rather than assigning an absolute block address to the bucket. A table maintained in the file header converts the bucket number into the corresponding disk block address

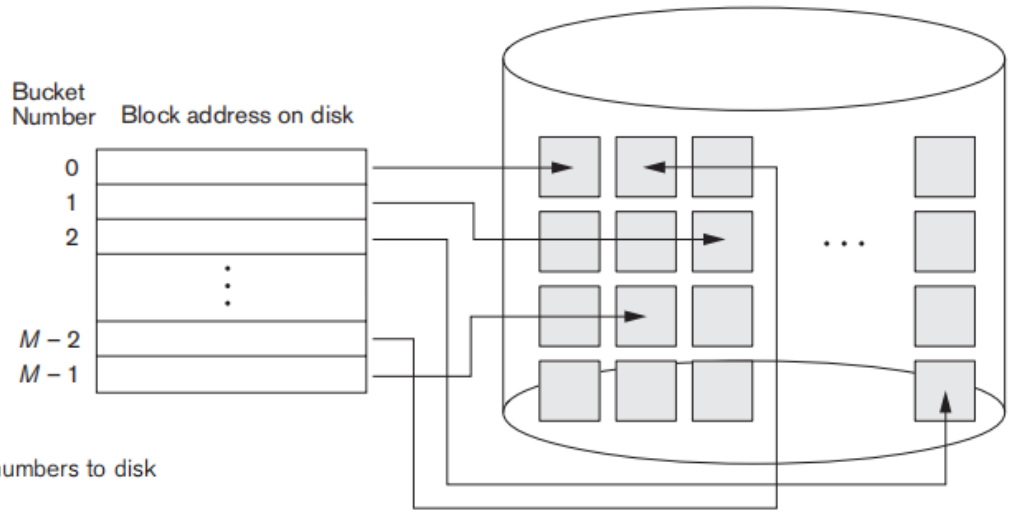


Figure 17.9
Matching bucket numbers to disk block addresses.

The collision problem is less severe with buckets, because as many records as will fit in a bucket can hash to the same bucket without causing problems. However, we must make provisions for the case where a bucket is filled to capacity and a new record being inserted hashes to that bucket. We can use a variation of chaining in which a pointer is maintained in each bucket to a linked list of overflow records for the bucket. The pointers in the linked list should be record pointers, which include both a block address and a relative record position within the block.

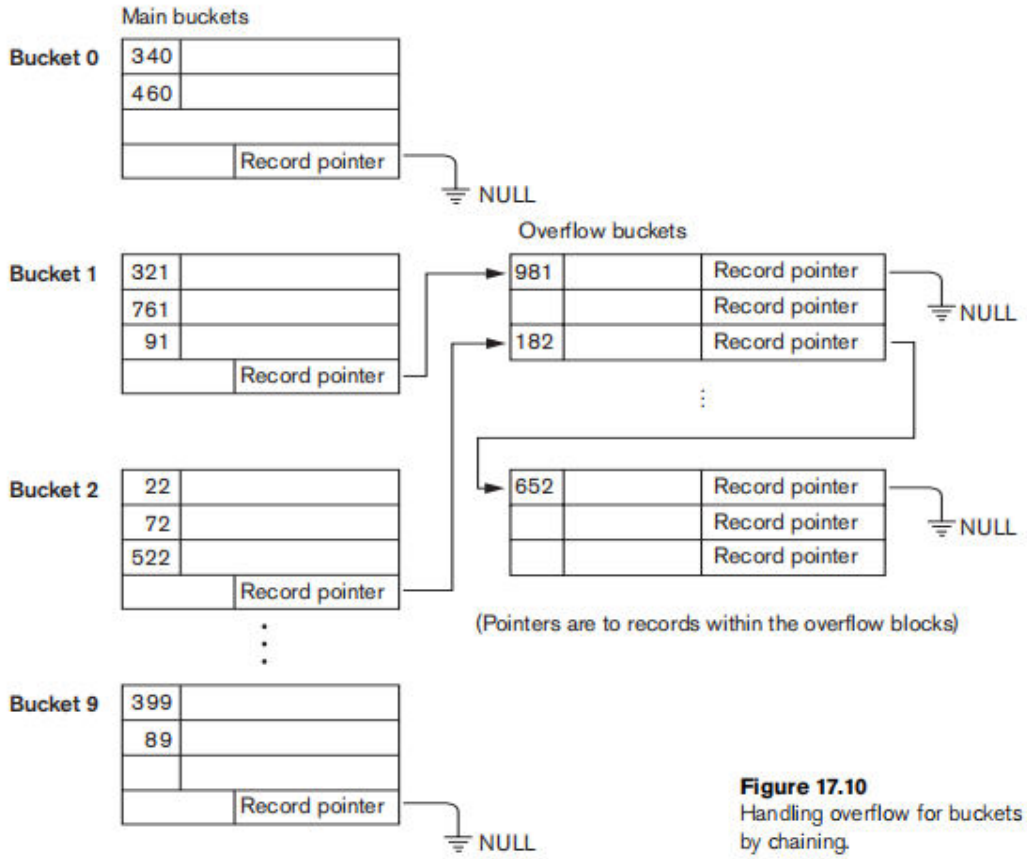


Figure 17.10
Handling overflow for buckets by chaining.

The hashing scheme described so far is called static hashing because a fixed number of buckets M is allocated. This can be a serious drawback for dynamic files. Suppose that we allocate M buckets for the address space and let m be the maximum number of records that can fit in one bucket; then at most (m * M) records will fit in the allocated space. If the number of records turns out to be substantially fewer than (m * M), we are left with a lot of unused space. On the other hand, if the number of records increases to substantially more than (m * M), numerous collisions will result and retrieval will be slowed down because of the long lists of overflow records. In either case, we may have to change the number of blocks M allocated and then use a new hashing function (based on the new value of M) to redistribute the records. These reorganizations can be quite time-consuming for large files. Newer dynamic file organizations based on hashing allow the number of buckets to vary dynamically with only localized reorganization

When using external hashing, searching for a record given a value of some field other than the hash field is as expensive as in the case of an unordered file. Record deletion can be implemented by removing the record from its bucket. If the bucket has an overflow chain, we can move one of the overflow records into the bucket to replace the deleted record. If the record to be deleted is already in overflow, we simply remove it from the linked list. Notice that removing an overflow

record implies that we should keep track of empty positions in overflow. This is done easily by maintaining a linked list of unused overflow locations.

Modifying a specific record's field value depends on two factors: the search condition to locate that specific record and the field to be modified. If the search condition is an equality comparison on the hash field, we can locate the record efficiently by using the hashing function; otherwise, we must do a linear search. A non hash field can be modified by changing the record and rewriting it in the same bucket. Modifying the hash field means that the record can move to another bucket, which requires deletion of the old record followed by insertion of the modified record.

Dynamic and Extendible Hashing Techniques

Hashing techniques are adapted to allow the dynamic growth and shrinking of the number of file records.These techniques include the following: dynamic hashing, extendible hashing, and linear hashing.Both dynamic and extendible hashing use the binary representation of the hash value h(K) in order to access a directory.

In dynamic hashing the directory is a binary tree. In extendible hashing the directory is an array of size 2d where d is called the global depth.

Extendible Hashing

In extendible hashing, a type of directory—an array of 2d bucket addresses—is maintained, where d is called the global depth of the directory. The integer value corresponding to the first (high-order) d bits of a hash value is used as an index to the array to determine a directory entry, and the address in that entry determines the bucket in which the corresponding records are stored. However, there does not have to be a distinct bucket for each of the 2d directory locations. Several directory locations with the same first d bits for their hash values may contain the same bucket address if all the records that hash to these locations fit in a single bucket. A local depth d —stored with each bucket—specifies the number of bits on which the bucket contents are based.

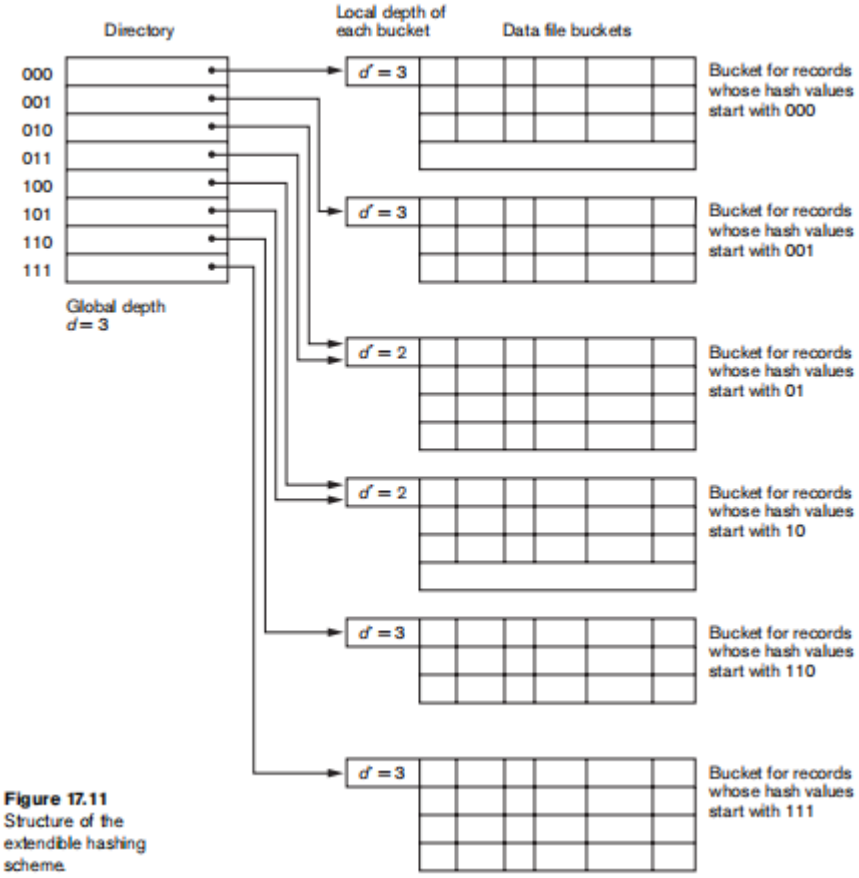


Figure 17.11
Structure of the extendible hashing scheme.

The value of d can be increased or decreased by one at a time, thus doubling or halving the number of entries in the directory array. Doubling is needed if a bucket, whose local depth d is equal to the global depth d, overflows. Halving occurs if d > d for all the buckets after some deletions occur. Most record retrievals require two block accesses—one to the directory and the other to the bucket.

To illustrate bucket splitting, suppose that a new inserted record causes overflow in the bucket whose hash values start with 01—the third bucket. The records will be distributed between two buckets: the first contains all records whose hash values start with 010, and the second all those whose hash values start with 011. Now the two directory locations for 010 and 011 point to the two new distinct buckets. Before the split, they pointed to the same bucket. The local depth d of the two new buckets is 3, which is one more than the local depth of the old bucket.

If a bucket that overflows and is split used to have a local depth d equal to the global depth d of the directory, then the size of the directory must now be doubled so that we can use an extra bit to distinguish the two new buckets. For example, if the bucket for records whose hash values start with 111 overflows, the two new buckets need a directory with global depth d = 4, because the two buckets are now labeled 1110 and 1111, and hence their local depths are both 4. The

directory size is hence doubled, and each of the other original locations in the directory is also split into two locations, both of which have the same pointer value as did the original location.
The main advantage of extendible hashing that makes it attractive is that the performance of the file does not degrade as the file grows, as opposed to static external hashing where collisions increase and the corresponding chaining effectively increases the average number of accesses per key. Additionally, no space is allocated in extendible hashing for future growth, but additional buckets can be allocated dynamically as needed. The space overhead for the directory table is negligible. The maximum directory size is 2k, where k is the number of bits in the hash value.
Another advantage is that splitting causes minor reorganization in most cases, since only the records in one bucket are redistributed to the two new buckets. The only time reorganization is more expensive is when the directory has to be doubled (or halved). A disadvantage is that the directory must be searched before accessing the buckets themselves, resulting in two block accesses instead of one in static hashing. This performance penalty is considered minor and thus the scheme is considered quite desirable for dynamic files.

Index Structures

Indexing is a data structure technique to efficiently retrieve records from the database files based on some attributes on which the indexing has been done.
An index on a database table provides a convenient mechanism for locating a row (data record) without scanning the entire table and thus greatly reduces the time it takes to process a query.
The index is usually specified on one field of the file.
One form of an index is a file of entries <field value, pointer to record>, which is ordered by field value. The index file usually occupies considerably less disk blocks than the data file because its entries are much smaller.

Types of Ordered Indexes

There are several types of ordered indexes.

- A **primary index**
  - is specified on the ordering key field of an ordered file of records.
  - An ordering key field is used to physically order the file records on disk, and every record has a unique value for that field.
- Clustered Index
  - If the ordering field is not a key field that is, if numerous records in the file can have the same value for the ordering field another type of index, called a **clustering index**, can be used.

- The data file is called a clustered file.
- Notice that a file can have at most one physical ordering field, so it can have at most one primary index or one clustering index, but not both.

- Secondary index,
  - It can be specified on any nonordering field of a file.
  - A data file can have several secondary indexes in addition to its primary access method

Primary Indexes

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

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

<K(1) = (Aaron, Ed), P(1) = address of block 1>

<K(2) = (Adams, John), P(2) = address of block 2>

<K(3) = (Alexander, Ed), P(3) = address of block 3>

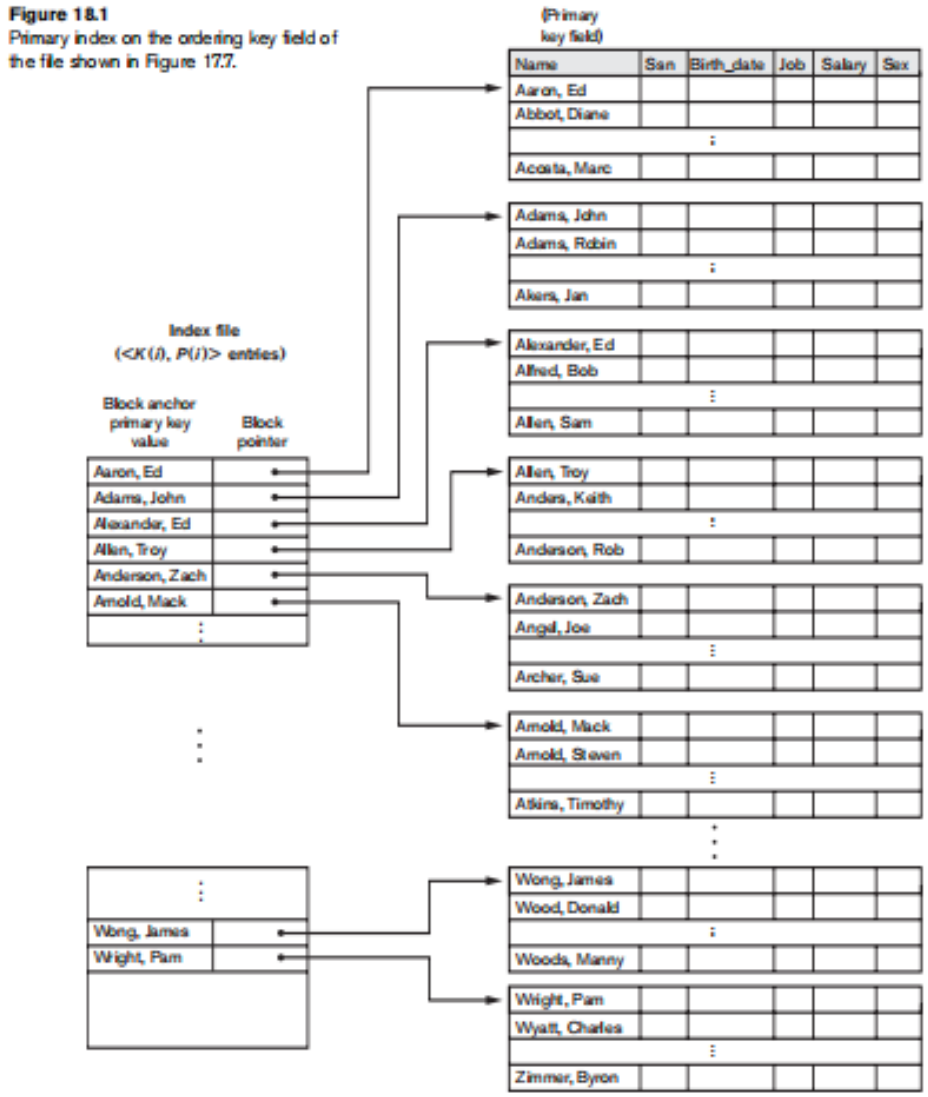


Figure 18.1
Primary index on the ordering key field of the file shown in Figure 17.7.

The total number of entries in the index is the same as the number of disk blocks in the ordered data file. The first record in each block of the data file is called the anchor record of the block, or simply the block anchor

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

The index file for a primary index occupies a much smaller space than does the data file, for two reasons. First, there are fewer index entries than there are records in the data file. Second, each index entry is typically smaller in size than a data record because it has only two fields;

consequently, more index entries than data records can fit in one block. Therefore, a binary search on the index file requires fewer block accesses than a binary search on the data file.
the binary search for an ordered data file required log2b block accesses. But if the primary index file contains only bi blocks, then to locate a record with a search key value requires a binary search of that index and access to the block containing that record: a total of log2bi + 1 accesses. A record whose primary key value is K lies in the block whose address is P(i), where $K(i) \leq K < K(i + 1)$. The ith block in the data file contains all such records because of the physical ordering of the file records on the primary key field. To retrieve a record, given the value K of its primary key field, we do a binary search on the index file to find the appropriate index entry i, and then retrieve the data file block whose address is P(i).

Example 1.
Suppose that we have an ordered file with r = 30,000 records stored on a disk with block size B = 1024 bytes. File records are of fixed size and are unspanned, with record length R = 100 bytes. The blocking factor for the file would be bfr = $\lfloor$ (B/R)$\rfloor$ = $\lfloor$ (1024/100)$\rfloor$ = 10 records per block. The number of blocks needed for the file is b = $\lceil$ (r/bfr)$\rceil$ = $\lceil$ (30000/10)$\rceil$ = 3000 blocks. A binary search on the data file would need approximately $\lceil$ log2b$\rceil$ = $\lceil$ (log23000)$\rceil$ = 12 block accesses.
Now suppose that the ordering key field of the file is V = 9 bytes long, a block pointer is P = 6 bytes long, and we have constructed a primary index for the file. The size of each index entry is Ri = (9 + 6) = 15 bytes, so the blocking factor for the index is bfri = $\lfloor$ (B/Ri)$\rfloor$ = $\lfloor$ (1024/15)$\rfloor$ = 68 entries per block. The total number of index entries ri is equal to the number of blocks in the data file, which is 3000. The number of index blocks is hence bi = $\lceil$ (ri/bfri)$\rceil$ = $\lceil$ (3000/68)$\rceil$ = 45 blocks. To perform a binary search on the index file would need $\lceil$ (log2bi)$\rceil$ = $\lceil$ (log245)$\rceil$ = 6 block accesses. To search for a record using the index, we need one additional block access to the data file for a total of 6 + 1 = 7 block accesses—an improvement over binary search on the data file, which required 12 disk block accesses.

Clustering Indexes

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

A clustering index is also an ordered file with two fields; the first field is of the same type as the clustering field of the data file, and the second field is a disk block pointer. There is one entry in the clustering index for each distinct value of the clustering field, and it contains the value and a pointer to the first block in the data file that has a record with that value for its clustering field.

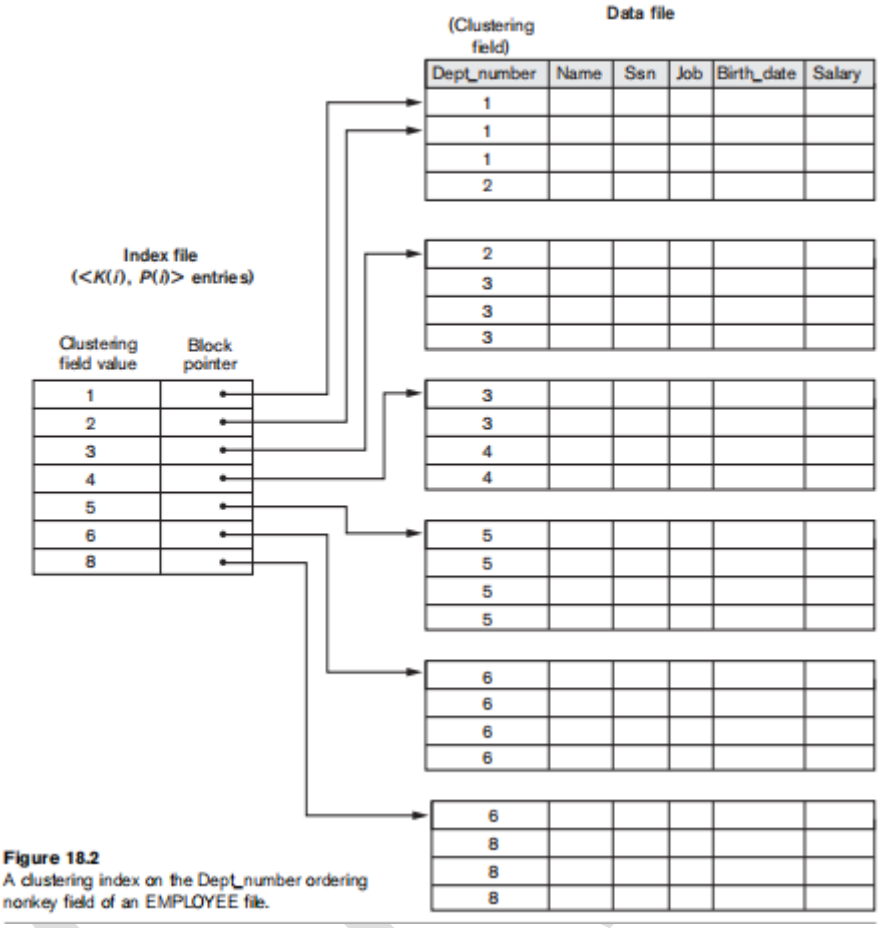


Figure 18.2
A clustering index on the Dept_number ordering nonkey field of an EMPLOYEE file.

The record insertion and deletion still cause problems because the data records are physically ordered. To alleviate the problem of insertion, it is common to reserve a whole block (or a cluster of contiguous blocks) for each value of the clustering field; all records with that value are placed in the block (or block cluster). This makes insertion and deletion relatively straightforward.

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

An index is somewhat similar to dynamic hashing and to the directory structures used for extendible hashing. Both are searched to find a pointer to the data block containing the desired

record. A main difference is that an index search uses the values of the search field itself, whereas a hash directory search uses the binary hash value that is calculated by applying the hash function to the search field.

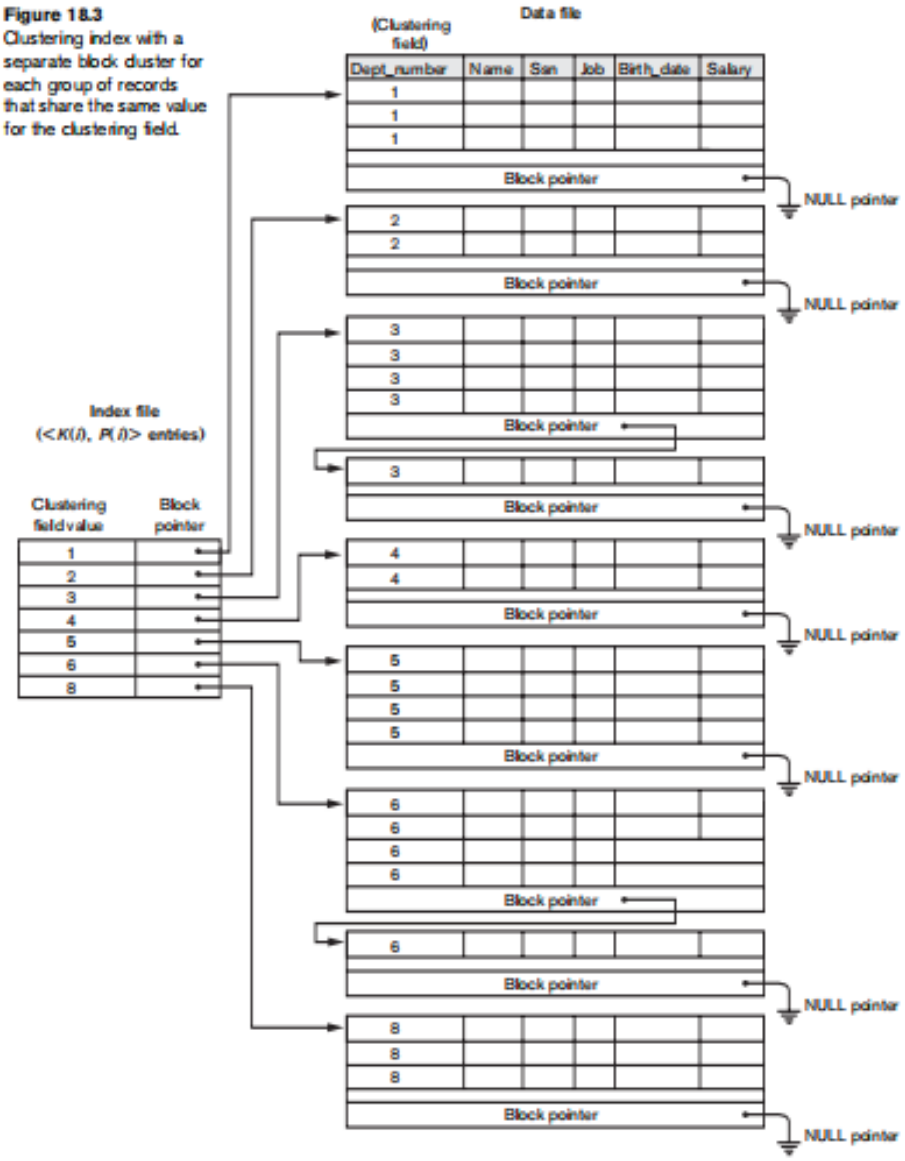


Figure 18.3
Clustering index with a separate block cluster for each group of records that share the same value for the clustering field.

Question

Suppose we have an ordered file with 30,0000 records and stored on a disk of block size 4096 bytes and records are of size 100 byte, unspanned organization. The ordered file is based on non key field (department_name). How many block access if using a primary index file, with an ordering key field of the file 5 bytes and block pointer size 6 bytes. There are 1000 department and 300 employees per department.

Ans. No of record=300000

Block size=4096 b

Record size=100 b

Block ptr=6 b

Clustered Index

Without index:

No of blocks= ⌐300000/40¬ =7500

No. of block access= log2 7500=13

With clustered index

No of index record= ⌐4096/(6+5)¬ =4096/11=372

Total no of index record=1000 department

No of clustered index blocks= ⌐ 1000/372¬ =3 blocks

No of clock access= ⌐log2 3¬ +1= 2+1=3 block access

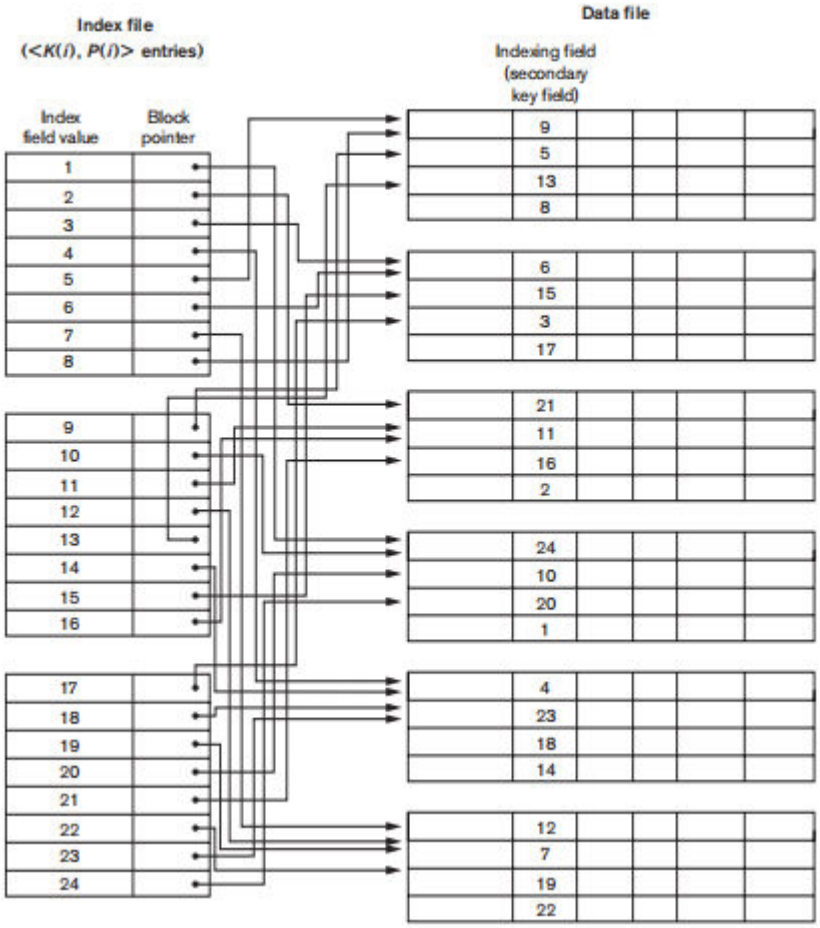
Secondary Indexes

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

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

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

Figure 18.4
A dense secondary index (with block pointers) on a nonordering key field of a file.



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

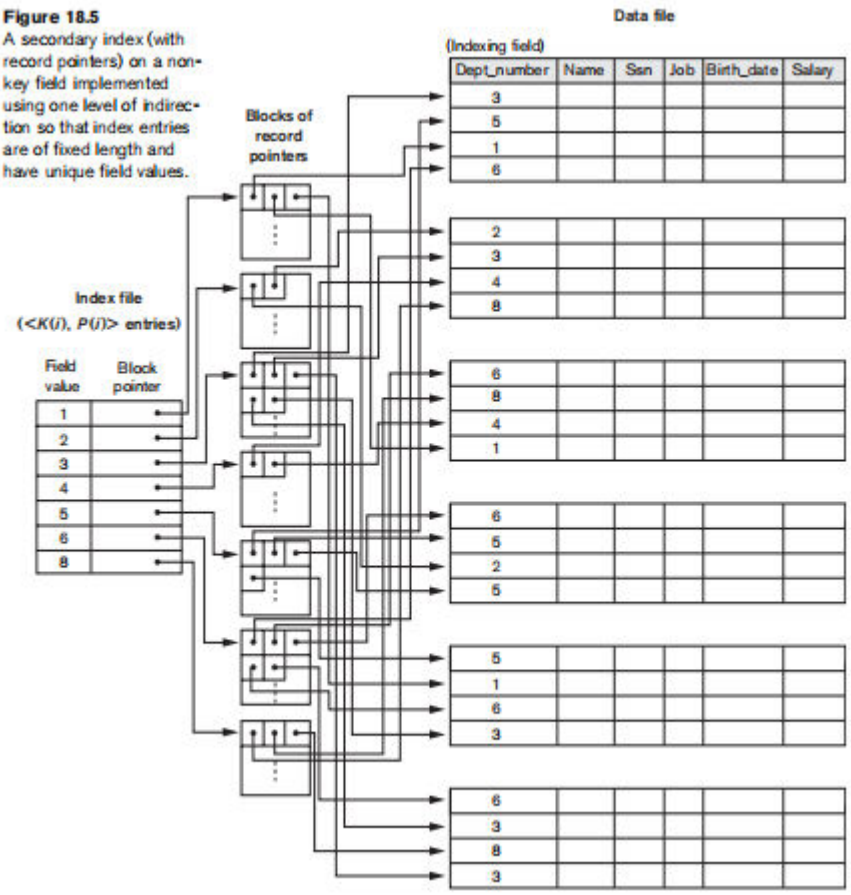
Example 2

Consider the file of Example 1 with r = 30,000 fixed-length records of size R = 100 bytes stored on a disk with block size B = 1024 bytes. The file has b = 3000 blocks, as calculated in Example 1. Suppose we want to search for a record with a specific value for the secondary key—a nonordering key field of the file that is V = 9 bytes long. Without the secondary index, to do a linear search on the file would require b/2 = 3000/2 = 1500 block accesses on the average. Suppose that we construct a secondary index on that nonordering key field of the file. Aa block pointer is P = 6 bytes long, so each index entry is Ri = (9 + 6) = 15 bytes, and the blocking factor for the index is bfri = $\lfloor$ (B/Ri)$\rfloor$ = $\lfloor$ (1024/15)$\rfloor$ = 68 entries per block. In a dense secondary index such as this, the total number of index entries ri is equal to the number of records in the data

file, which is 30,000. The number of blocks needed for the index is hence bi = ⌈ (ri/bfri)⌉ = ⌈ (3000/68)⌉ = 442 blocks. A binary search on this secondary index needs ⌈ (log2bi)⌉ = ⌈ (log2442)⌉ = 9 block accesses. To search for a record using the index, we need an additional block access to the data file for a total of 9 + 1 = 10 block accesses—a vast improvement over the 1500 block accesses needed on the average for a linear search, but slightly worse than the 7 block accesses required for the primary index. This difference arose because the primary index was nondense and hence shorter, with only 45 blocks in length.

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

- Option 1 is to include duplicate index entries with the same K(i) value—one for each record. This would be a dense index.
- Option 2 is to have variable-length records for the index entries, with a repeating field for the pointer. We keep a list of pointers <P(i, 1), ..., P(i, k)> in the index entry for K(i)—one pointer to each block that contains a record whose indexing field value equals K(i). In either option 1 or option 2, the binary search algorithm on the index must be modified appropriately to account for a variable number of index entries per index key value.
- Option 3, which is more commonly used, is to keep the index entries themselves at a fixed length and have a single entry for each index field value, but to create an extra level of indirection to handle the multiple pointers. In this nondense scheme, the pointer P(i) in index entry <K(i), P(i)> points to a disk block, which contains a set of record pointers; each record pointer in that disk block points to one of the data file records with value K(i) for the indexing field. If some value K(i) occurs in too many records, so that their record pointers cannot fit in a single disk block, a cluster or linked list of blocks is used.



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

Summary

To conclude this section, we summarize the discussion of index types in two tables. Table 18.1 shows the index field characteristics of each type of ordered single-level index discussed primary, clustering, and secondary. Table 18.2 summarizes the properties of each type of index by comparing the number of index entries and specifying which indexes are dense and which use block anchors of the data file.

Table 18.1 Types of Indexes Based on the Properties of the Indexing Field

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

Table 18.2 Properties of Index Types

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

Multilevel Indexes

The indexing schemes we have described thus far involve an ordered index file. A binary search is applied to the index to locate pointers to a disk block or to a record (or records) in the file having a specific index field value. *A binary search requires approximately (log_2 bi) block accesses for an index with bi blocks because each step of the algorithm reduces the part of the index file that we continue to search by a factor of 2.* This is why we take the log function to the base 2.

The idea behind a multilevel index is to reduce the part of the index that we continue to search by bfri, the blocking factor for the index, which is larger than 2. Hence, the search space is reduced much faster. The value bfri is called the fan-out of the multilevel index, and we will refer to it by the symbol fo. Whereas we divide the record search space into two halves at each step during a binary search, we divide it n-ways (where n = the fan-out) at each search step using the multilevel index. Searching a multilevel index requires approximately (logfobi) block accesses, which is a substantially smaller number than for a binary search if the fan-out is larger than 2. In most cases, the fan-out is much larger than 2.

A multilevel index considers the index file, which we will now refer to as the first(or base) level of a multilevel index, as an ordered file with a distinct value for each K(i). Therefore, by considering the first-level index file as a sorted data file, we can create a primary index for the first level; this index to the first level is called the second level of the multilevel index. Because the second level is a primary index, we can use block anchors so that the second level has one entry for each block of the first level.

The blocking factor bfri for the second level and for all subsequent levels is the same as that for the first-level index because all index entries are the same size; each has one field value and one block address. If the first level has r1 entries, and the blocking factor which is also the fan-out for

the index is bfri = fo, then the first level needs $\lceil (r1/fo) \rceil$ blocks, which is therefore the number of entries r2 needed at the second level of the index.

We can repeat this process for the second level. The third level, which is a primary index for the second level, has an entry for each second-level block, so the number of third-level entries is r3 = $\lceil (r2/fo) \rceil$. Notice that we require a second level only if the first level needs more than one block of disk storage, and, similarly, we require a third level only if the second level needs more than one block. We can repeat the preceding process until all the entries of some index level t fit in a single block. This block at the tth level is called the top index level.4 Each level reduces the number of entries at the previous level by a factor of fo—the index fan-out—so we can use the formula $1 \leq (r1/((fo)t))$ to calculate t. Hence, a multilevel index with r1 first-level entries will have approximately t levels, where t = $\lceil (\log fo(r1)) \rceil$. When searching the index, a single disk block is retrieved at each level. Hence, t disk blocks are accessed for an index search, where t is the number of index levels.

Example 3.

Suppose that the dense secondary index of Example 2 is converted into a multilevel index. We calculated the index blocking factor bfri = 68 index entries per block, which is also the fan-out fo for the multilevel index; the number of firstlevel blocks b1 = 442 blocks was also calculated. The number of second-level blocks will be b2 = $\lceil (b1/fo) \rceil$ = $\lceil (442/68) \rceil$ = 7 blocks, and the number of third-level blocks will be b3 = $\lceil (b2/fo) \rceil$ = $\lceil (7/68) \rceil$ = 1 block. Hence, the third level is the top level of the index, and t = 3. To access a record by searching the multilevel index, we must access one block at each level plus one block from the data file, so we need t + 1 = 3 + 1 = 4 block accesses

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

Problem:

Consider a file of 16384 records. Each record is 32 bytes long and its key field is of size 6 bytes. The file is ordered on a non-key field, and the file organization is unspanned. The file is stored in a file system with block size 1024 bytes, and the size of a block pointer is 10 bytes. If the secondary index is built on the key field of the file, and a multi-level index scheme is used to store the secondary index, the number of first-level and second-level blocks in the multi-level index are?

ANS.

Number of records in file = 16384

Record size = 32 bytes

Key Size = 6 bytes

Block Size on file system = 1024 bytes

Size of Block Pointer = 10 bytes

Size of a record or index Entry = 10 + 6 = 16

- ordered on a non-key field
- secondary index is built on the key field of the file,
- multi-level index scheme is used

Without Indexing

No of data record per block=1024/32=32

No of data block=16384/32=512

Average block access without indexing=b/2=512/2=256

Indexing First Level:

No of index record=16384

secondary index is built on the key field of the file non ordered

No of index record per block =1024/(6+10)=64

No of index blocks=16384/64=256

Indexing Second Level

No of index records=256

ordered on a non-key field

No of index record per block=1024/16=64

No of index block =256/64=4 blocks

Dynamic Multilevel Indexes Using

B-Trees and B+-Trees

B-trees and B+-trees are special cases of the well-known search data structure known as a tree. A tree is formed of nodes. Each node in the tree, except for a special node called the root, has one parent node and zero or more child nodes. The root node has no parent. A node that does not have any child nodes is called a leaf node; a nonleaf node is called an internal node. The level of a node is always one more than the level of its parent, with the level of the root node being zero. A subtree of a node consists of that node and all its descendant nodes its child nodes, the child nodes of its child nodes, and so on

Search Trees

A search tree is a special type of tree that is used to guide the search for a record, given the value of one of the record's fields. A search tree of order p is a tree such that each node contains at most p − 1 search values and p pointers in the order <P1, K1, P2, K2, ..., Pq−1, Kq−1, Pq>, where q ≤ p. Each Pi is a pointer to a child node (or a NULL pointer), and each Ki is a search value from some ordered set of values. All search values are assumed to be unique.

Two constraints must hold at all times on the search tree:

1. Within each node, K1 < K2 < ... < Kq−1.
2. For all values X in the subtree pointed at by Pi , we have Ki−1 < X < Ki for 1 < i < q; X < Ki for i = 1; and Ki−1 < X for i = q

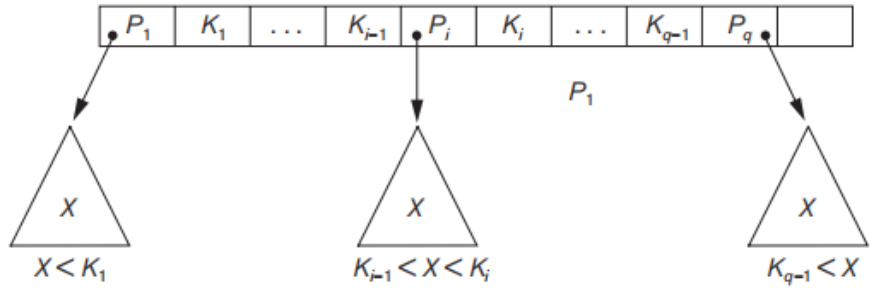


Figure 18.8
A node in a search tree with pointers to subtrees below it.

Whenever we search for a value X, we follow the appropriate pointer Pi according to the formulas in condition 2 above.

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

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

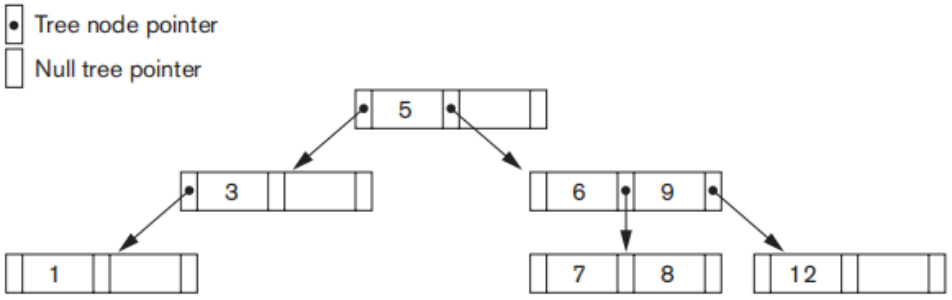


Figure 18.9
A search tree of order $p = 3$.

Binary Search Tree insertion and deletion do not guarantee that a search tree is balanced, meaning that all of its leaf nodes are at the same level.7 The tree in Figure 18.7 is not balanced because it has leaf nodes at levels 1, 2, and 3. The goals for balancing a search tree are as follows:

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

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

B-Trees

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

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

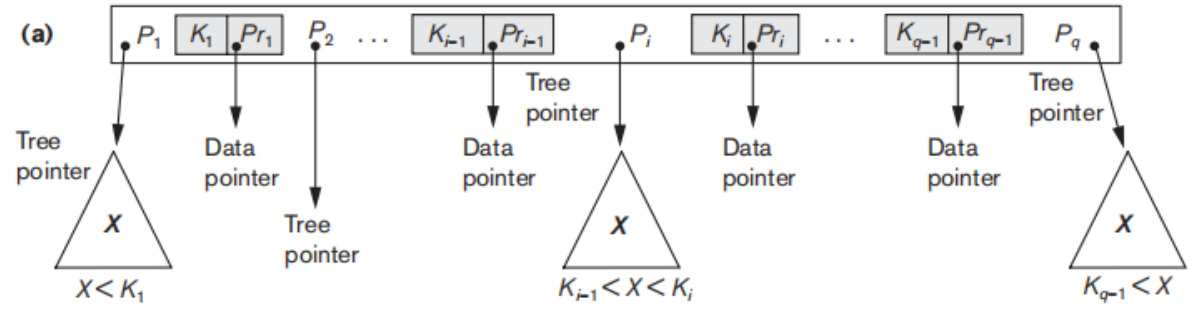
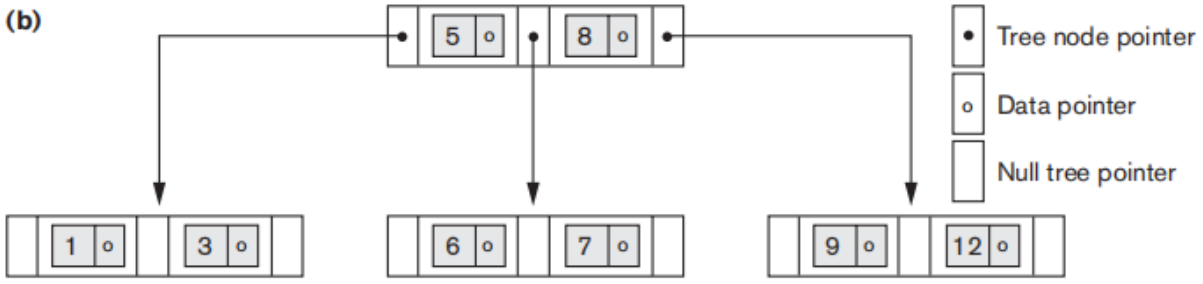


Fig (a) A node in a B-tree with q – 1 search values

1. Each internal node in the B-tree is of the form <P1, <K1, Pr1>, P2, <K2, Pr2>, ..., <Kq–1, Prq–1>, Pq> where q ≤ p. Each **Pi is a tree pointer** a pointer to another node in the Btree. Each Pri is a data pointer, a pointer to the record whose search **key field value** is equal to **Ki** (or to the data file block containing that record).
2. Within each node, K1 < K2 < ... < Kq−1.
3. For all search key field values X in the subtree pointed at by Pi (the ith subtree), we have: Ki–1 < X < Ki for 1 < i < q; X < Ki for i = 1; and Ki–1 < X for i = q.
4. Each node has **at most p tree pointers**.
5. Each node, **except the root** and leaf nodes, has at least **⌈ (p/2)⌉ tree pointers**. The **root node has at least two tree pointers** unless it is the only node in the tree.
6. A node with **q tree pointers, q ≤** p, has **q – 1 search key field** values (and hence has q – 1 data pointers).
7. All **leaf nodes** are at the **same level.** Leaf nodes have the **same structure** as internal nodes except that all of their tree pointers Pi are NULL.



(b) A B-tree of order p = 3.The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.

A B-tree starts with a single root node (which is also a leaf node) at level 0 (zero). Once the root node is full with p – 1 search key values and we attempt to insert another entry in the tree, the root node splits into two nodes at level 1. Only the middle value is kept in the root node, and the rest of the values are split evenly between the other two nodes. When a nonroot node is full and a new entry is inserted into it, that node is split into two nodes at the same level, and the middle entry is moved to the parent node along with two pointers to the new split nodes. If
the parent node is full, it is also split. Splitting can propagate all the way to the root node, creating a new level if the root is split.

If deletion of a value causes a node to be less than half full, it is combined with its neighboring nodes, and this can also propagate all the way to the root. Hence, deletion can reduce the number of tree levels.

B-trees are sometimes used as primary file organizations. In this case, whole records are stored within the B-tree nodes rather than just the <search key, record pointer> entries. This works well for files with a relatively small number of records and a small record size

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

B+-Trees

In a B+-tree, data pointers are stored *only at the leaf nodes* of the tree; hence, the structure of leaf nodes differs from the structure of internal nodes. The leaf nodes have an entry for *every* value of the search field, along with a data pointer to the record.

The structure of the *internal nodes* of a B+ tree of order *p* is as follows:

1. Each internal node is of the form <*P*1, *K*1, *P*2, *K*2, ..., *Pq* – 1, *Kq* –1, *Pq*> where $q \leq p$ and each *Pi* is a **tree pointer**.
2. Within each internal node, $K1 < K2 < ... < Kq-1$.
3. For all search field values *X* in the subtree pointed at by *Pi*, we have $Ki-1 < X \leq Ki$ for $1 < i < q$; $X \leq Ki$ for $i = 1$; and $Ki-1 < X$ for $i = q$.
4. Each internal node has at most *p* tree pointers.
5. Each internal node, except the root, has at least $\lceil (p/2) \rceil$ tree pointers. The root node has at least two tree pointers if it is an internal node.
6. An internal node with *q* pointers, $q \leq p$, has $q - 1$ search field values.

The structure of the *leaf nodes* of a B+-tree of order *p* (Figure 18.11(b)) is as follows:

1. Each leaf node is of the form <<*K*1, *Pr*1>, <*K*2, *Pr*2>, ..., <*Kq*–1, *Prq*–1>, *P*next> where $q \leq p$, each *Pri* is a data pointer, and *P*next points to the next *leaf node* of the B+-tree.
2. Within each leaf node, $K1 \leq K2 ... , Kq-1, q \leq p$.
3. Each *Pri* is a **data pointer** that points to the record whose search field value is *Ki* or to a file block containing the record
4. Each leaf node has at least $\lceil (p/2) \rceil$ values.
5. All leaf nodes are at the same level.

(a)

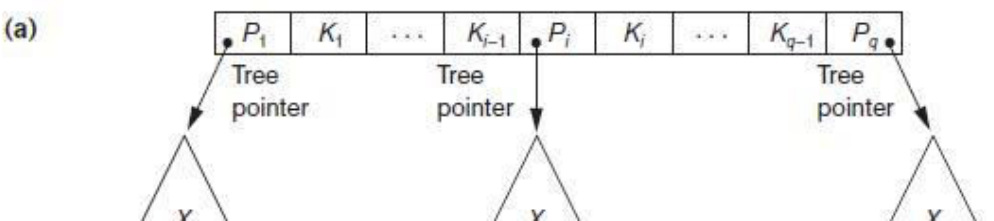


Figure 5.8: The nodes of a B+-tree. (a) Internal node of a B+-tree with $q - 1$ search values. (b) Leaf node of a B+-tree with $q - 1$ search values and $q - 1$ data pointers.

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

Example of an Insertion in a B+-tree

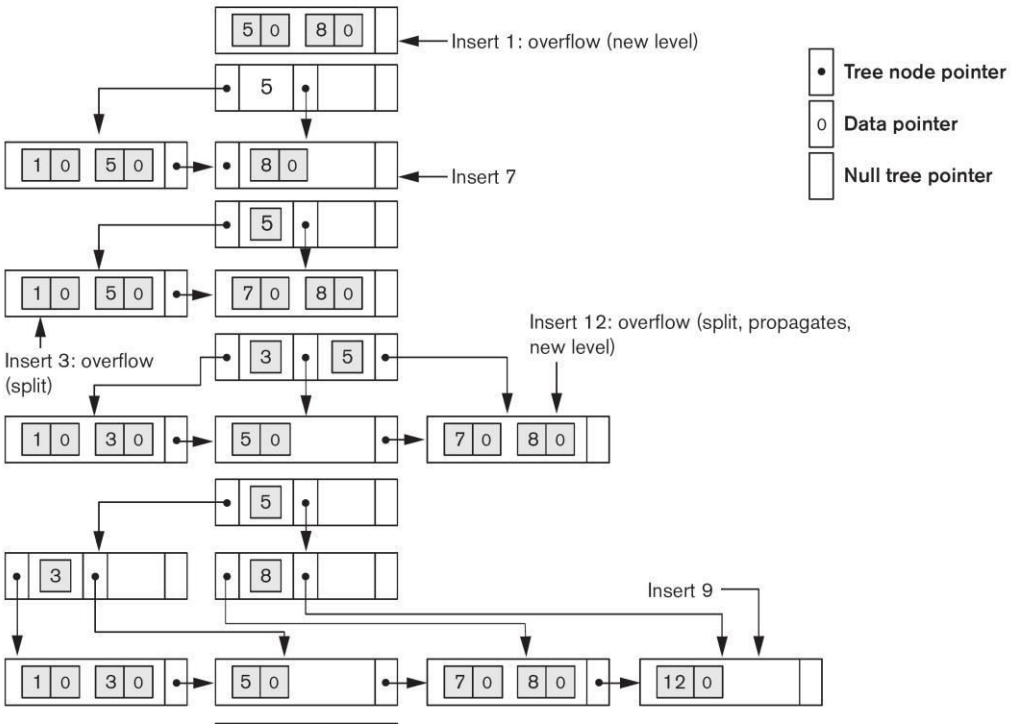
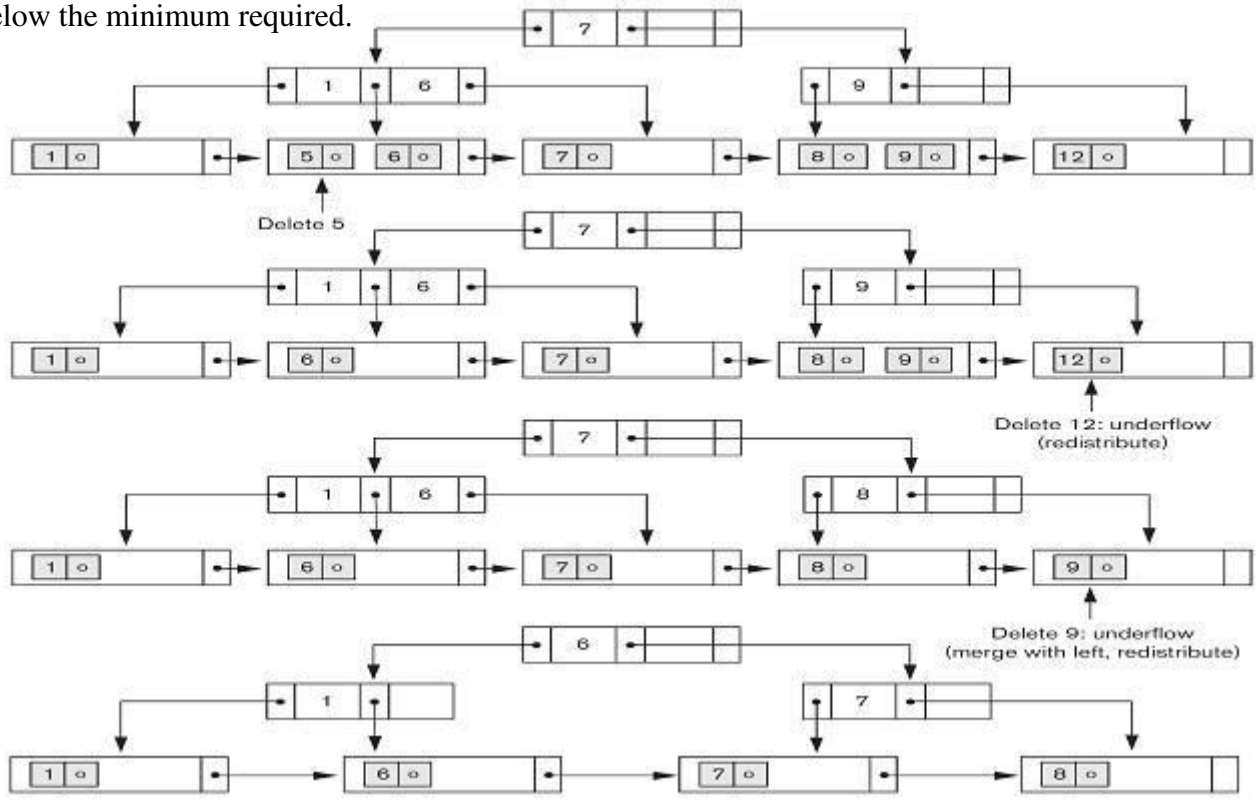


Figure 5.9: An example of insertion in a B+-tree with *p* = 3 and *p*leaf = 2. The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.

Example of a Deletion in a B+-tree

When an entry is deleted, it is always removed from the leaf level. If it happens to occur in an internal node, it must also be removed from there. In the latter case, the value to its left in the leaf node must replace it in the internal node because that value is now the rightmost entry in the subtree. Deletion may cause underflow by reducing the number of entries in the leaf node to below the minimum required.



In this case, we try to find a sibling leaf node—a leaf node directly to the left or to the right of the node with underflow—and redistribute the entries among the node and to sibling so that both are at least half full; otherwise, the node is merged with its siblings and the number of leaf nodes is reduced.

Most multi-level indexes use B-tree or B+-tree data structures because of the insertion and deletion problem. This leaves space in each tree node (disk block) to allow for new index entries. These data structures are variations of search trees that allow efficient insertion and deletion of new search values. In B-Tree and B+-Tree data structures, each node corresponds to a disk block. Each node is kept between half-full and completely full. An insertion into a node that is not full is quite efficient. If a node is full the insertion causes a split into two nodes. Splitting may propagate to other tree levels. A deletion is quite efficient if a node does not become less than half full. If a deletion causes a node to become less than half full, it must be merged with neighboring nodes

Difference between B-tree and B+-tree

In a B-tree, pointers to data records exist at all levels of the tree. In a B+-tree, all pointers to data records exists at the leaf-level nodes. A B+-tree can have less levels (or higher capacity of search values) than the corresponding B-tree.

Indexes on Multiple Keys

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

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

1. Assuming Dno has an index, but Age does not, access the records having Dno = 4 using the index, and then select from among them those records that satisfy Age = 59.
2. Alternately, if Age is indexed but Dno is not, access the records having Age = 59 using the index, and then select from among them those records that satisfy Dno = 4.

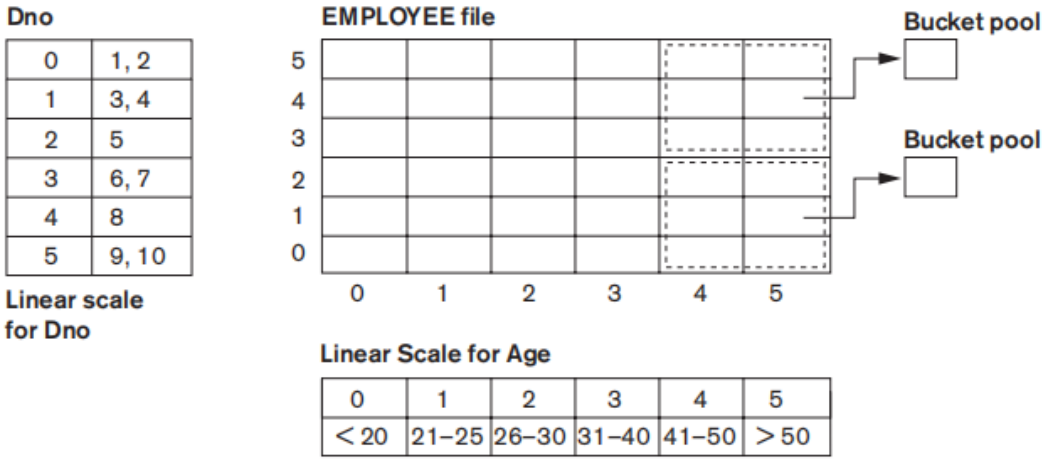
3. If indexes have been created on both Dno and Age, both indexes may be used; each gives a set of records or a set of pointers (to blocks or records). An intersection of these sets of records or pointers yields those records or pointers that satisfy both conditions.

All of these alternatives eventually give the correct result. However, if the set of records that meet each condition (Dno = 4 or Age = 59) individually are large, yet only a few records satisfy the combined condition, then none of the above is an efficient technique for the given search request. A number of possibilities exist that would treat the combination < Dno, Age> or < Age, Dno> as a search key made up of multiple attributes.

Grid Files

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

Figure 18.14
Example of a grid array on Dno and Age attributes.



Thus our request for Dno = 4 and Age = 59 maps into the cell (1, 5) corresponding to the grid array. The records for this combination will be found in the corresponding bucket. This method is

particularly useful for range queries that would map into a set of cells corresponding to a group of values along the linear scales. If a range query corresponds to a match on the some of the grid cells, it can be processed by accessing exactly the buckets for those grid cells. For example, a query for Dno ≤ 5and Age > 40 refers to the data in the top bucket shown in Figure. The grid file concept can be applied to any number of search keys. For example, for n search keys, the grid array would have n dimensions. The grid array thus allows a partitioning of the file along the dimensions of the search key attributes and provides an access by combinations of values along those dimensions. Grid files perform well in terms of reduction in time for multiple key access. However, they represent a space overhead in terms of the grid array structure. Moreover, with dynamic files, a frequent reorganization of the file adds to the maintenance cost.