INSERTION SORT

The basic step in this method is to insert a new record into a sorted sequence of *i* records in such a way that the resulting sequence of size *i* + 1 is also ordered.
Function *insert* accomplishes this insertion.

```
void insert(element e, element all, int i)
    {/* insert e into the ordered list a[1 : i] such that the resulting list a[1: i+1] is also
        ordered, the array a must have space allocated for at least i+2 elements */
        a[0] = e;
        while (e.key < a[i].key)
        {
            a[i+1] = a[i] ;
            i--;
        }
        a[i+1] = e;
    }
```

Program: Insertion into a sorted list

The use of *a*[0] enables us to simplify the while loop, avoiding a test for end of list (*i* < 1). In insertion sort, begin with the ordered sequence *a* [1] and successively insert the records *a* [2], *a* [3], ... , *a [n]*. Since each insertion leaves the resultant sequence ordered, the list with *n* records can be ordered making *n* - 1 insertions.
The details are given in function *insertionSort*.


```
void insertionSort(element all, int n)
    {   /* sort a[1: n] into nondecreasing order */
        int j;
        for (j = 2i j <= n : j++)
        {
            element temp = a[j];
            insert (temp, a, j-1);
        }
    }
```

Program: Insertion sort

Analysis of insertion Sort: In the worst case insert (e, a, i) makes i + 1 comparisons before making the insertion. Hence the complexity of Insert is O(i). Function insertionSort invokes insert for i = j - 1 = 1, 2, ... , n - 1. So, the complexity of insertionSort is

$$O(\sum_{i=1}^{n-1} (i+1)) = O(n^2).$$

The average time for insertionSort is $O(n^2)$

Example: Assume that n = 5 and the input key sequence is 5, 4, 3, 2, 1. After each iteration we have

j	[1]	[2]	[3]	[4]	[5]
–	**5**	4	3	2	1
2	**4**	**5**	3	2	1
3	**3**	**4**	**5**	2	1
4	**2**	**3**	**4**	**5**	1
5	**1**	**2**	**3**	**4**	**5**

Example: Assume that n = 5 and the input key sequence is 2, 3, 4, 5, 1. after each iteration we have

j	[1]	[2]	[3]	[4]	[5]
–	**2**	3	4	5	1
2	**2**	**3**	4	5	1
3	**2**	**3**	**4**	5	1
4	**2**	**3**	**4**	**5**	1
5	**1**	**2**	**3**	**4**	**5**

RADIX SORT

Radix sort is the method that many people intuitively use or begin to use when alphabetizing a large list of names. (Here the radix is 26, the 26 letters of the alphabet.) Specifically, the list of names is first sorted according to the first letter of each name. That is, the names are arranged in 26 classes, where the first class consists of those names that begin with "A," the second class consists of those names that begin with "B," and so on. During the second pass, each class is alphabetized according to the second letter of the name. And so on. If no name contains, for example, more than 12 letters, the names are alphabetized with at most 12 passes.

The radix sort is the method used by a card sorter. A card sorter contains 13 receiving pockets labelled as follows:

9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 11, 12, R (reject)

Each pocket other than R corresponds to a row on a card in which a hole can be punched. Decimal numbers, where the radix is 10, are punched in the obvious way and hence use only the first 10 pockets of the sorter. The sorter uses a radix reverse-digit sort on numbers. That is, suppose a card sorter is given a collection of cards where each card contains a 3-digit number punched in columns 1 to 3. The cards are first sorted according to the unit's digit. On the second pass, the cards are sorted according to the tens digit. On the third and last pass, the cards are sorted according to the hundreds digit.

Illustration with an example:

Suppose 9 cards are punched as follows:

348, 143, 361, 423, 538, 128, 321, 543, 366

Given to a card sorter, the numbers would be sorted in three phases, as pictured in

Input	0	1	2	3	4	5	6	7	8	9
348									348	
143				143						
361		361								
423				423						
538									538	
128									128	
321		321								
543				543						
366							366			

(a) First pass

Input	0	1	2	3	4	5	6	7	8	9
361							361			
321			321							
143					143					
423			423							
543					543					
366					543					
366							366			
348					348					
538				538						
128			128							

(b) Second pass

Input	0	1	2	3	4	5	6	7	8	9
321				321						
423					423					
128		128								
538						538				
143		143								
543						543				
348				348						
361				361						
366				366						

(c) Third pass

MODULE 5

Files and Their Organization

DATA HIERARCHY

Every file contains data which can be organized in a hierarchy to present a systematic organization.
The data hierarchy includes data items such as fields, records, files, and database. These terms are defined below.

- *Data field:* A *data field* is an elementary unit that stores a single fact. A data field is usually characterized by its type and size.
  Example: student's name is a data field that stores the name of students.

- *Record:* A *record* is a collection of related data fields which is seen as a single unit from the application point of view.
  Example: The student's record may contain data fields such as name, address, phone number, roll number, marks obtained, and so on.

- *File:* A *file* is a collection of related records.
  Example: A file of all the employees working in an organization

- *Directory*: A *directory* stores information of related files. A directory organizes information so that users can find it easily.
  Example: Below fig. shows how multiple related files are stored in a student directoy.

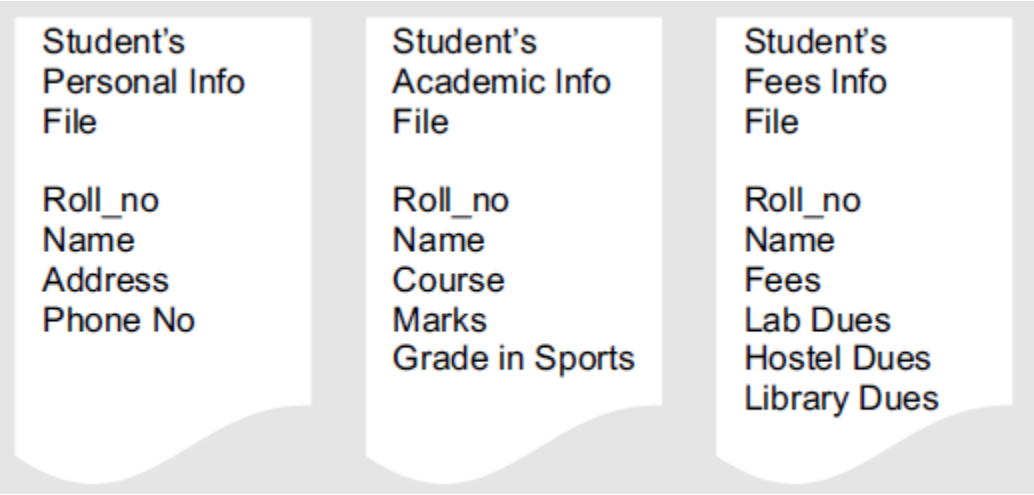


Figure Student directory

FILE ATTRIBUTES

File has a list of attributes associated with it that gives the operating system and the application software information about the file and how it is intended to be used.

The attributes are explained below

- **File name:** It is a string of characters that stores the name of a file. File naming conventions vary from one operating system to the other.
- **File position:** It is a pointer that points to the position at which the next read/write operation will be performed.
- **File structure:** It indicates whether the file is a text file or a binary file. In the text file, the numbers are stored as a string of characters. A binary file stores numbers in the same way as they are represented in the main memory.
- **File Access Method:** It indicates whether the records in a file can be accessed sequentially or randomly.
  In sequential access mode, records are read one by one. That is, if 60 records of students are stored in the STUDENT file, then to read the record of 39th student, you have to go through the record of the first 38 students.
  In random access, records can be accessed in any order.

- **Attributes Flag:** A file can have six additional attributes attached to it. These attributes are usually stored in a single byte, with each bit representing a specific attribute. If a particular bit is set to '1' then this means that the corresponding attribute is turned on.

Table Attribute flag

Attribute	Attribute Byte
Read-Only	00000001
Hidden	00000010
System	00000100
Volume Label	00001000
Directory	00010000
Archive	00100000

Above figure shows the list of attributes and their position in the attribute flag or attribute byte.

- **Read-only:** A file marked as read-only cannot be deleted or modified. Example: if an attempt is made to either delete or modify a read-only file, then a message 'access denied' is displayed on the screen.
- **Hidden:** A file marked as hidden is not displayed in the directory listing.
- **System:** A file marked as a system file indicates that it is an important file used by the system and should not be altered or removed from the disk.
- **Volume Label:** Every disk volume is assigned a label for identification. The label can be assigned at the time of formatting the disk or later through various tools such as the DOS command LABEL.
- **Directory:** In directory listing, the files and sub-directories of the current directory are differentiated by a directory-bit. This means that the files that have the directory-bit turned on are actually sub-directories containing one or more files.
- **Archive:** The archive bit is used as a communication link between programs that modify files and those that are used for backing up files. Most backup programs allow the user to do an incremental backup.

TEXT AND BINARY FILES

Text Files

- A text file, also known as a flat file or an ASCII file, is structured as a sequence of lines of alphabet, numerals, special characters.
- The data in a text file, whether numeric or non-numeric, is stored using its corresponding ASCII code.
- The end of a text file is denoted by placing a special character, called an end-of-file marker, after the last line in the text file.
- It is possible for humans to read text files which contain only ASCII text.
- Text files can be manipulated by any text editor, they do not provide efficient storage.

Binary Files

- A binary file contains any type of data encoded in binary form for computer storage and processing purposes.
- A binary file can contain text that is not broken up into lines.
- A binary file stores data in a format that is similar to the format in which the data is stored in the main memory. Therefore, a binary file is not readable by humans.
- Binary files contain formatting information that only certain applications or processors can understand.
- Binary files must be run on an appropriate software or processor so that the software or processor can transform the data in order to make it readable.
- Binary files provide efficient storage of data, but they can be read only through an appropriate program.

BASIC FILE OPERATIONS

The basic operations that can be performed on a file are given in below figure

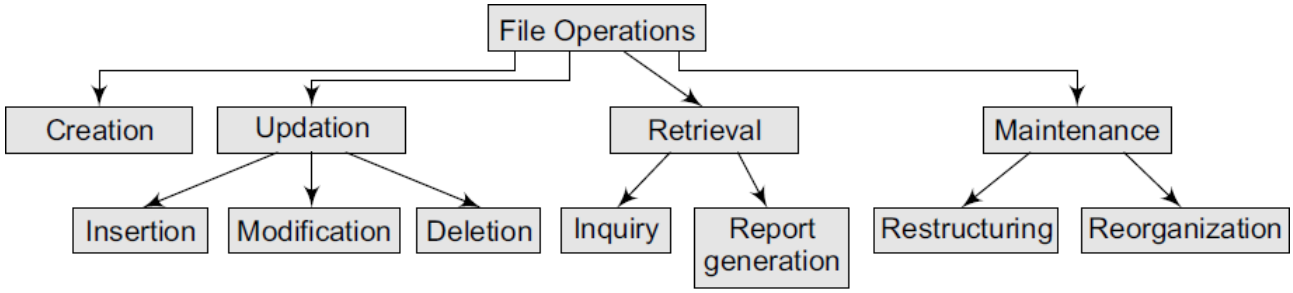


Figure File operations

Creating a File

A file is created by specifying its name and mode. Then the file is opened for writing records that are read from an input device. Once all the records have been written into the file, the file is closed. The file is now available for future read/write operations by any program that has been designed to use it in some way or the other.

Updating a File

Updating a file means changing the contents of the file to reflect a current picture of reality. A file can be updated in the following ways:

- Inserting a new record in the file. For example, if a new student joins the course, we need to add his record to the STUDENT file.
- Deleting an existing record. For example, if a student quits a course in the middle of the session, his record has to be deleted from the STUDENT file.
- Modifying an existing record. For example, if the name of a student was spelt incorrectly, then correcting the name will be a modification of the existing record.

Retrieving from a File

It means extracting useful data from a given file. Information can be retrieved from a file either for an inquiry or for report generation. An inquiry for some data retrieves low volume of data, while report generation may retrieve a large volume of data from the file.

Maintaining a File

It involves restructuring or re-organizing the file to improve the performance of the programs that access this file.

Restructuring a file keeps the file organization unchanged and changes only the structural aspects of the file.

Example: changing the field width or adding/deleting fields.

File reorganization may involve changing the entire organization of the file

FILE ORGANIZATION

Organization of records means the logical arrangement of records in the file and not the physical layout of the file as stored on a storage media.

The following considerations should be kept in mind before selecting an appropriate file organization method:

- Rapid access to one or more records
- Ease of inserting/updating/deleting one or more records without disrupting the speed of accessing record
- Efficient storage of records
- Using redundancy to ensure data integrity

1. Sequential Organization

A sequentially organized file stores the records in the order in which they were entered. Sequential files can be read only sequentially, starting with the first record in the file. Sequential file organization is the most basic way to organize a large collection of records in a file

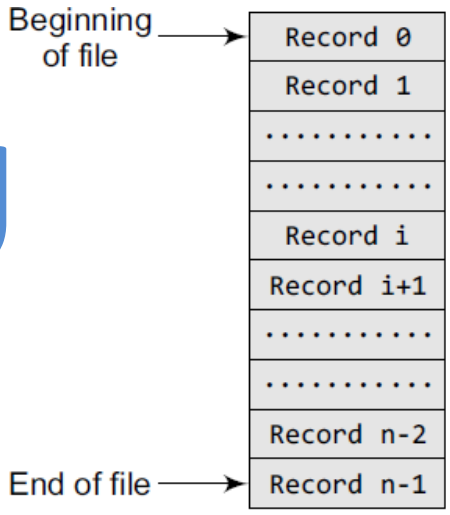


Figure Sequential file organization

Features

- Records are written in the order in which they are entered
- Records are read and written sequentially
- Deletion or updation of one or more records calls for replacing the original file with a new file that contains the desired changes
- Records have the same size and the same field format
- Records are sorted on a key value
- Generally used for report generation or sequential reading

Advantages

- Simple and easy to Handle
- No extra overheads involved
- Sequential files can be stored on magnetic disks as well as magnetic tapes
- Well suited for batch– oriented applications

Disadvantages

- Records can be read only sequentially. If i^{th} record has to be read, then all the i–1 records must be read
- Does not support update operation. A new file has to be created and the original file has to be replaced with the new file that contains the desired changes
- Cannot be used for interactive applications

2. Relative File Organization

Figure shows a schematic representation of a relative file which has been allocated space to store 100 records

Relative record number	Records stored in memory
0	Record 0
1	Record 1
2	FREE
3	FREE
4	Record 4
.....	.....
98	FREE
99	Record 99

Figure Relative file organization

If the records are of fixed length and we know the base address of the file and the length of the record, then any record i can be accessed using the following formula:

$$\text{Address of } i^{th} \text{ record} = \text{base\_address} + (i-1) * \text{record\_length}$$

Consider the base address of a file is 1000 and each record occupies 20 bytes, then the address of the 5th record can be given as:

$$1000 + (5-1) * 20$$
$$= 1000 + 80$$
$$= 1080$$

Features

- Provides an effective way to access individual records
- The record number represents the location of the record relative to the beginning of the file
- Records in a relative file are of fixed length
- Relative files can be used for both random as well as sequential access
- Every location in the table either stores a record or is marked as FREE

Advantages

- Ease of processing
- If the relative record number of the record that has to be accessed is known, then the record can be accessed instantaneously
- Random access of records makes access to relative files fast
- Allows deletions and updations in the same file
- Provides random as well as sequential access of records with low overhead
- New records can be easily added in the free locations based on the relative record number of the record to be inserted
- Well suited for interactive applications

Disadvantages

- Use of relative files is restricted to disk devices
- Records can be of fixed length only
- For random access of records, the relative record number must be known in advance

3. Indexed Sequential File Organization

The index sequential file organization can be visualized as shown in figure

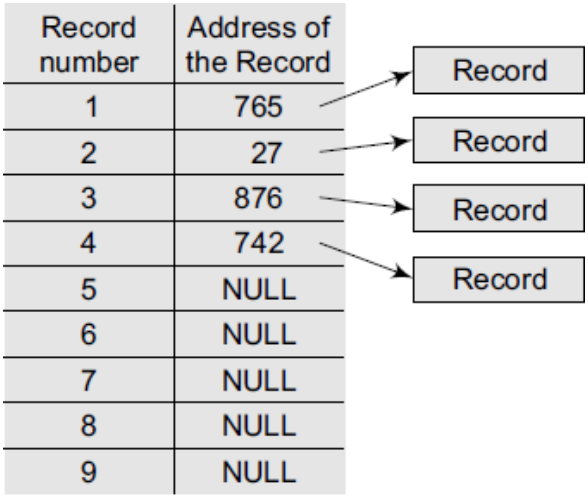


Figure Indexed sequential file organization

Features

- Provides fast data retrieval
- Records are of fixed length
- Index table stores the address of the records in the file
- The ith entry in the index table points to the ith record of the file
- While the index table is read sequentially to find the address of the desired record, a direct access is made to the address of the specified record in order to access it randomly
- Indexed sequential files perform well in situations where sequential access as well as random access is made to the data

Advantages

- The key improvement is that the indices are small and can be searched quickly, allowing the database to access only the records it needs
- Supports applications that require both batch and interactive processing
- Records can be accessed sequentially as well as randomly
- Updates the records in the same file

Disadvantages

- Indexed sequential files can be stored only on disks
- Needs extra space and overhead to store indices
- Handling these files is more complicated than handling sequential files
- Supports only fixed length records

INDEXING

the indexing technique based on factors such as access type, access time, insertion time, deletion time, and space overhead involved. There are two kinds of indices:

- *Ordered indices* that are sorted based on one or more key values
- *Hash indices* that are based on the values generated by applying a *hash function*

1. Ordered Indices

Indices are used to provide fast random access to records. An index of a file may be a primary index or a secondary index.

Primary Index

In a sequentially ordered file, the index whose search key specifies the sequential order of the file is defined as the primary index.

Example: suppose records of students are stored in a STUDENT file in a sequential order starting from roll number 1 to roll number 60. Now, if we want to search a record for, say, roll number 10, then the student's roll number is the primary index.

Secondary Index

An index whose search key specifies an order different from the sequential order of the file is called as the secondary index.

Example: If the record of a student is searched by his name, then the name is a secondary index. Secondary indices are used to improve the performance of queries on non-primary keys.

2. Dense and Sparse Indices

Dense index

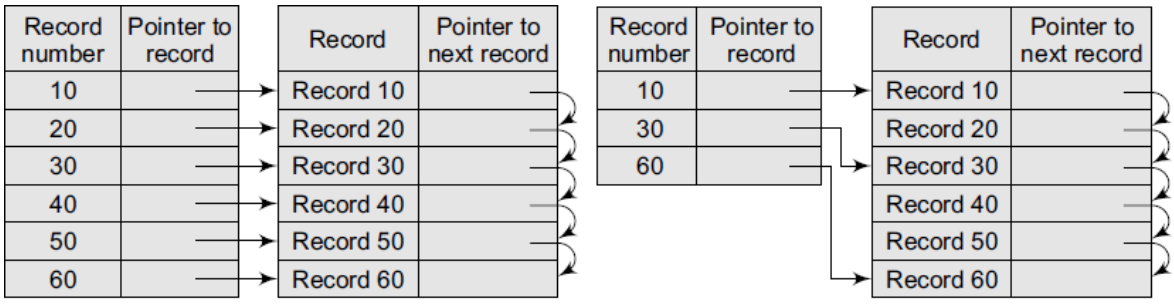
- In a dense index, the index table stores the address of every record in the file.
- Dense index would be more efficient to use than a sparse index if it fits in the memory
- By looking at the dense index, it can be concluded directly whether the record exists in the file or not.

Sparse index

- In a sparse index, the index table stores the address of only some of the records in the file.
- Sparse indices are easy to fit in the main memory,
- In a sparse index, to locate a record, first find an entry in the index table with the largest search key value that is either less than or equal to the search key value of the desired record. Then, start at that record pointed to by that entry in the index table and then proceed searching the record using the sequential pointers in the file, until the desired record is obtained.

Example: If we need to access record number 40, then record number 30 is the largest key value that is less than 40. So jump to the record pointed by record number 30 and move along the sequential pointer to reach record number 40.

Below figure shows a dense index and a sparse index for an indexed sequential file.



(a) Dense index (b) Sparse index

Figure Dense index and sparse index

3. Cylinder Surface Indexing

Cylinder surface indexing is a very simple technique used only for the primary key index of a sequentially ordered file.
The index file will contain two fields—cylinder index and several surface indices.
There are multiple cylinders, and each cylinder has multiple surfaces. If the file needs m cylinders for storage then the cylinder index will contain m entries.

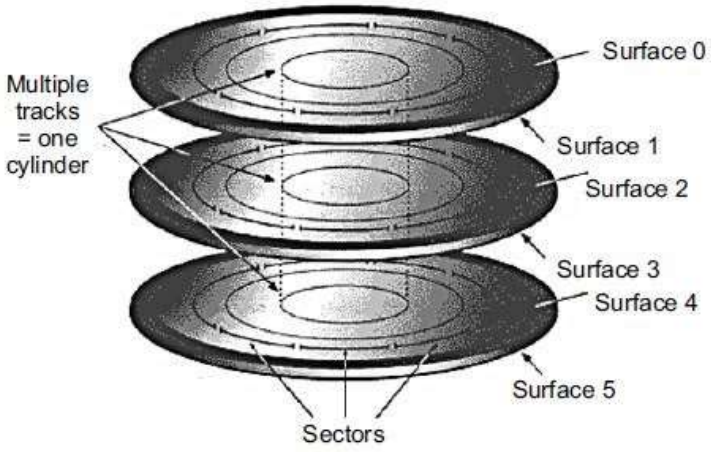


Figure Physical and logical organization of disk

When a record with a particular key value has to be searched, then the following steps are performed:

- First the cylinder index of the file is read into memory.
- Second, the cylinder index is searched to determine which cylinder holds the desired record. For this, either the binary search technique can be used or the cylinder index can be made to store an array of pointers to the starting of individual key values. In either case the search will take O (log m) time.
- After the cylinder index is searched, appropriate cylinder is determined.
- Depending on the cylinder, the surface index corresponding to the cylinder is then retrieved from the disk.
- Since the number of surfaces on a disk is very small, linear search can be used to determine surface index of the record.
- Once the cylinder and the surface are determined, the corresponding track is read and searched for the record with the desired key.

Hence, the total number of disk accesses is three—first, for accessing the cylinder index, second for accessing the surface index, and third for getting the track address.

4. Multi-level Indices

Consider very large files that may contain millions of records. For such files, a simple indexing technique will not suffice. In such a situation, we use multi-level indices.
Below figure shows a two-level multi-indexing. Three-level indexing and so, can also be used
In the figure, the main index table stores pointers to three inner index tables. The inner index tables are sparse index tables that in turn store pointers to the records.

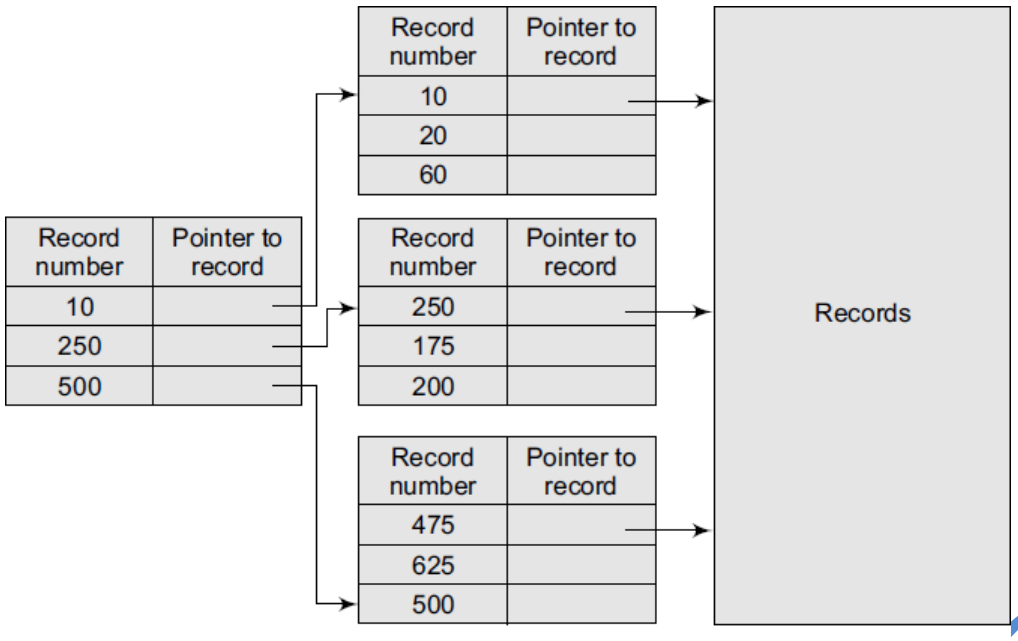


Figure Multi-level indices

5. Inverted Indices

- Inverted files are used in document retrieval systems for large textual databases.
- An inverted file reorganizes the structure of an existing data file in order to provide fast access to all records having one field falling within the set limits.
- When a term or keyword specified in the inverted file is identified, the record number is given and a set of records corresponding to the search criteria are created.
- For each keyword, an inverted file contains an inverted list that stores a list of pointers to all occurrences of that term in the main text. Therefore, given a keyword, the addresses of all the documents containing that keyword can easily be located.

There are two main variants of inverted indices:

- A record-level inverted index (inverted file index or inverted file) stores a list of references to documents for each word
- A word-level inverted index (full inverted index or inverted list) in addition to a list of references to documents for each word also contains the positions of each word within a document.

6. B-Tree (Balanced Tree) Indices

It is impractical to maintain the entire database in the memory, hence B-trees are used to index the data in order to provide fast access.
B-trees are used for its data retrieval speed, ease of maintenance, and simplicity.

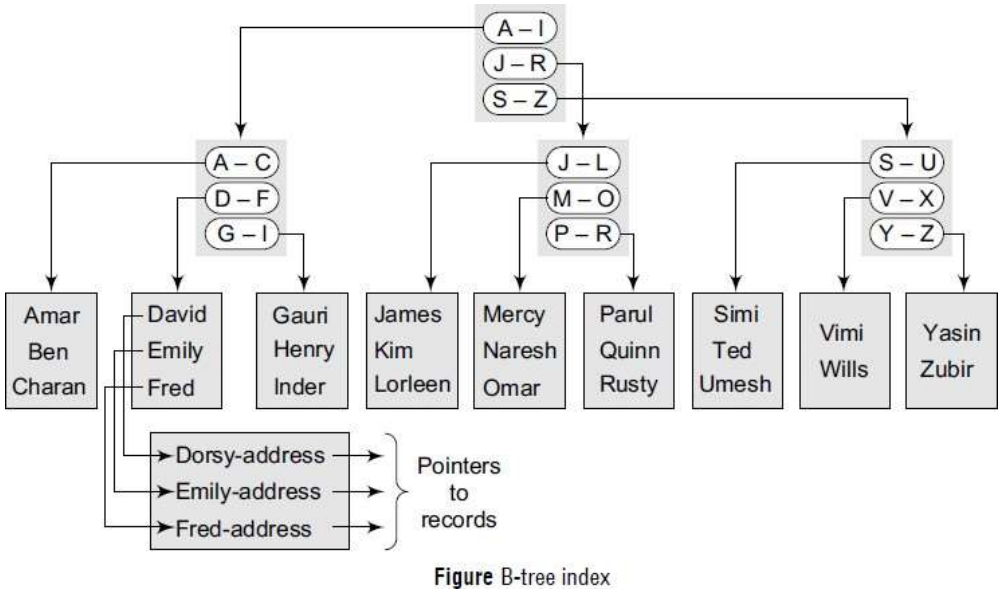


Figure B-tree index

- It forms a tree structure with the root at the top. The index consists of a B-tree (balanced tree) structure based on the values of the indexed column.
- In this example, the indexed column is name and the B-tree is created using all the existing names that are the values of the indexed column.
- The upper blocks of the tree contain index data pointing to the next lower block, thus forming a hierarchical structure. The lowest level blocks, also known as leaf blocks, contain pointers to the data rows stored in the table.

The B-tree structure has the following advantages:

- Since the leaf nodes of a B-tree are at the same depth, retrieval of any record from anywhere in the index takes approximately the same time.
- B-trees improve the performance of a wide range of queries that either searches a value having an exact match or for a value within specified range.
- B-trees provide fast and efficient algorithms to insert, update, and delete records that maintain the key order.
- B-trees perform well for small as well as large tables. Their performance does not degrade as the size of a table grows.
- B-trees optimize costly disk access.

7. Hashed Indices

Hashing is used to compute the address of a record by using a hash function on the search key value.
The hashed values map to the same address, then collision occurs and schemes to resolve these collisions are applied to generate a new address

Choosing a **good hash function** is critical to the success of this technique. By a good hash function, it mean two things.

1. First, a good hash function, irrespective of the number of search keys, gives an average-case lookup that is a small constant.
2. Second, the function distributes records uniformly and randomly among the buckets, where a bucket is defined as a unit of one or more records

The **worst hash function** is one that maps all the keys to the same bucket.

The drawback of using hashed indices includes:

- Though the number of buckets is fixed, the number of files may grow with time.
- If the number of buckets is too large, storage space is wasted.
- If the number of buckets is too small, there may be too many collisions.

The following operations are performed in a hashed file organization.

1. Insertion

To insert a record that has k_i as its search value, use the hash function $h(k_i)$ to compute the address of the bucket for that record.
If the bucket is free, store the record else use chaining to store the record.

2. Search

To search a record having the key value k_i, use $h(k_i)$ to compute the address of the bucket where the record is stored.
The bucket may contain one or several records, so check for every record in the bucket to retrieve the desired record with the given key value.

3. Deletion

To delete a record with key value k_i, use $h(k_i)$ to compute the address of the bucket where the record is stored. The bucket may contain one or several records so check for every record in the bucket, and then delete the record.