# UNIT 4 FILE ORGANISATION IN DBMS

**Structure** **Page Nos.**



## 4.0 INTRODUCTION

In earlier units, we studied the concepts of database integrity and how to normalise the tables. Databases are used to store information. Normally, the principal operations we need to perform on database are those relating to:

- Creation of data
- Retrieving data
- Modifying
- Deleting some information which we are sure is no longer useful or valid.

We have seen that in terms of the logical operations to be performed on the data, relational tables provide a good mechanism for all of the above tasks. Therefore the storage of a database in a computer memory (on the hard disk, of course), is mainly concerned with the following issues:

- The need to store a set of tables, where each table can be stored as an independent file.
- The attributes in a table are closely related and, therefore, often accessed together. Therefore it makes sense to store the different attribute values in each record contiguously. In fact, is it necessary that the attributes must be stored in the same sequence, for each record of a table?
- It seems logical to store all the records of a table contiguously. However, since there is no prescribed order in which records must be stored in a table, we may choose the sequence in which we store the different records of a table.

We shall see that the point (iii) among these observations is quite useful. Databases are used to store information in the form of files of records and are typically stored on magnetic disks. This unit focuses on the file Organisation in DBMS, the access methods available and the system parameters associated with them. File Organisation is the way the files are arranged on the disk and access method is how the data can be retrieved based on the file Organisation.

# 4.1 OBJECTIVES

After going through this unit you should be able to:

- define storage of databases on hard disks;
- discuss the implementation of various file Organisation techniques;
- discuss the advantages and the limitation of the various file Organisation techniques;
- describe various indexes used in database systems, and
- define the multi-key file organisation.

# 4.2 PHYSICAL DATABASE DESIGN ISSUES

The database design involves the process of logical design with the help of E-R diagram, normalisation, etc., followed by the physical design.

**The Key issues in the Physical Database Design are:**

- The purpose of physical database design is to translate the logical description of data into the technical specifications for storing and retrieving data for the DBMS.
- The goal is to create a design for storing data that will provide adequate performance and ensure database integrity, security and recoverability.

**Some of the basic inputs required for Physical Database Design are:**

- Normalised relations
- Attribute definitions
- Data usage: entered, retrieved, deleted, updated
- Requirements for security, backup, recovery, retention, integrity
- DBMS characteristics.
- Performance criteria such as response time requirement with respect to volume estimates.

**However, for such data some of the Physical Database Design Decisions that are to be taken are:**

- Optimising attribute data types.
- Modifying the logical design.
- Specifying the file Organisation.
- Choosing indexes.

### Designing the fields in the data base

The following are the considerations one has to keep in mind while designing the fields in the data base.

- Choosing data type
- Coding, compression, encryption
- Controlling data integrity
- Default value
  - Range control
  - Null value control
  - Referential integrity
- Handling missing data
  - Substitute an estimate of the missing value
  - Trigger a report listing missing values
  - In programs, ignore missing data unless the value is significant.

**Physical Records**

These are the records that are stored in the secondary storage devices. For a database relation, physical records are the group of fields stored in adjacent memory locations

and retrieved together as a unit. Considering the page memory system, data page is the amount of data read or written in one I/O operation to and from secondary storage device to the memory and vice-versa. In this context we define a term blocking factor that is defined as the number of physical records per page.

### The issues relating to the Design of the Physical Database Files

Physical File is a file as stored on the disk. The main issues relating to physical files are:

- Constructs to link two pieces of data:
  - Sequential storage.
  - Pointers.
- File Organisation: How the files are arranged on the disk?
- Access Method: How the data can be retrieved based on the file Organisation?

Let us see in the next section how the data is stored on the hard disks.

## 4.3 STORAGE OF DATABASE ON HARD DISKS

At this point, it is worth while to note the difference between the terms file Organisation and the access method. A file organisation refers to the organisation 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, is the way how the data can be retrieved based on the file Organisation.

Mostly the databases are stored persistently on magnetic disks for the reasons given below:

- The databases being very large may not fit completely in the main memory.
- Storing the data permanently using the non-volatile storage and provide access to the users with the help of front end applications.
- Primary storage is considered to be very expensive and in order to cut short the cost of the storage per unit of data to substantially less.

Each hard drive is usually composed of a set of disk platters. Each disk platter has a layer of magnetic material deposited on its surface. The entire disk can contain a large amount of data, which is organised into smaller packages called BLOCKS (or pages). On most computers, one block is equivalent to 1 KB of data (= 1024 Bytes).

A block is the smallest unit of data transfer between the hard disk and the processor of the computer. Each block therefore has a fixed, assigned, address. Typically, the computer processor will submit a read/write request, which includes the address of the block, and the address of RAM in the computer memory area called a buffer (or cache) where the data must be stored / taken from. The processor then reads and modifies the buffer data as required, and, if required, writes the block back to the disk. Let us see how the tables of the database are stored on the hard disk.

### How are tables stored on Disk?

We realise that each record of a table can contain different amounts of data. This is because in some records, some attribute values may be 'null'. Or, some attributes may be of type varchar (), and therefore each record may have a different length string as the value of this attribute. Therefore, the record is stored with each subsequent attribute separated by the next by a special ASCII character called a field separator. Of course, in each block, we may place many records. Each record is separated from the next, again by another special ASCII character called the record separator. Let us see in the next section about the types of file Organisation briefly.

# 4.4 FILE ORGANISATION AND ITS TYPES

Just as arrays, lists, trees and other data structures are used to implement data Organisation in main memory, a number of strategies are used to support the Organisation of data in secondary memory. A file organisation is a technique to organise data in the secondary memory. In this section, we are concerned with obtaining data representation for files on external storage devices so that required functions (e.g. retrieval, update) may be carried out efficiently.
File Organisation is a way of arranging the records in a file when the file is stored on the disk. Data files are organized so as to facilitate access to records and to ensure their efficient storage. A tradeoff between these two requirements generally exists: if rapid access is required, more storage is required to make it possible. Selection of File Organisations is dependant on two factors as shown below:

- Typical DBMS applications need a small subset of the DB at any given time.
- When a portion of the data is needed it must be located on disk, copied to memory for processing and rewritten to disk if the data was modified.

A file of record is likely to be accessed and modified in a variety of ways, and different ways of arranging the records enable different operations over the file to be carried out efficiently. A DBMS supports several file Organisation techniques. The important task of the DBA is to choose a good Organisation for each file, based on its type of use.

The particular organisation most suitable for any application will depend upon such factors as the kind of external storage available, types of queries allowed, number of keys, mode of retrieval and mode of update. The *Figure1* illustrates different file organisations based on an access key.

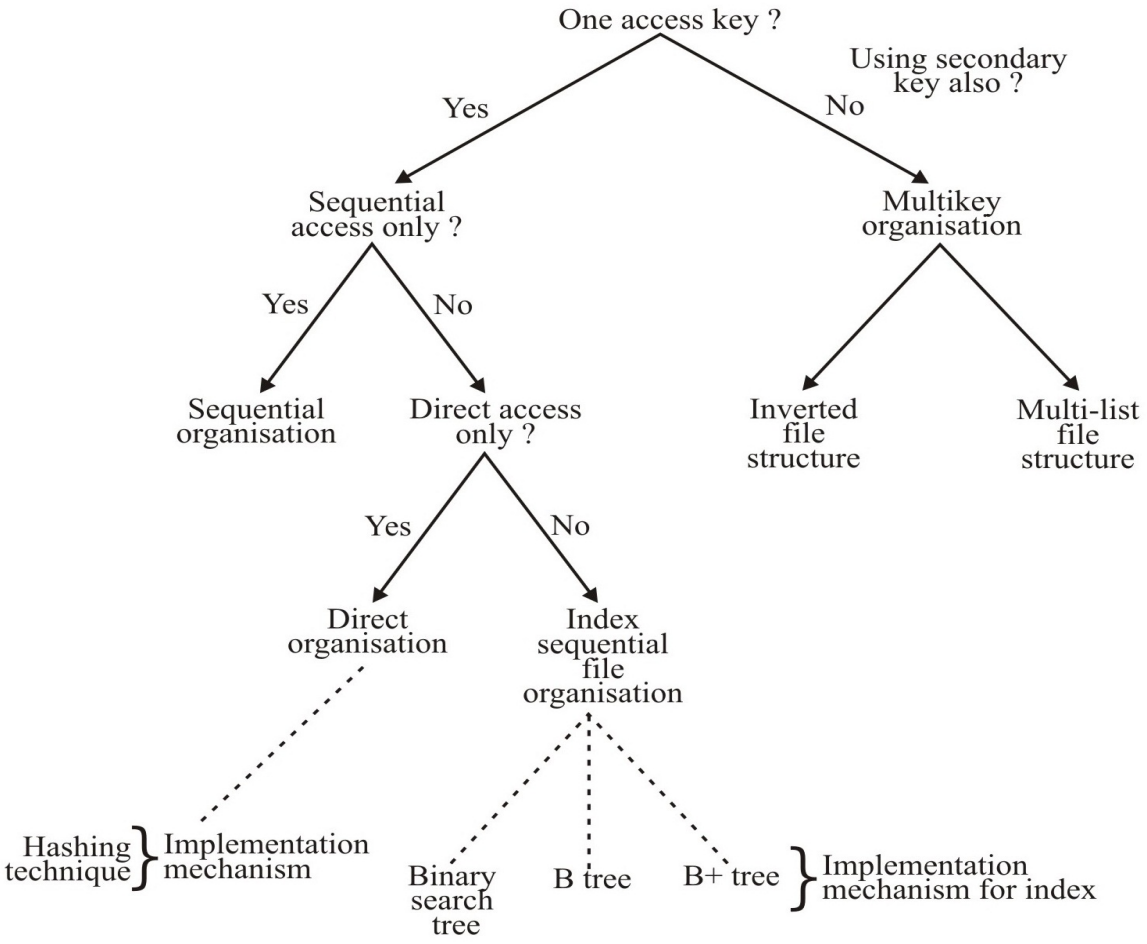


**Figure 1: File Organisation techniques**

Let us discuss some of these techniques in more detail:

### 4.4.1 Heap files (unordered file)

Basically these files are unordered files. It is the simplest and most basic type. These files consist of randomly ordered records. The records will have no particular order. The operations we can perform on the records are insert, retrieve and delete. The features of the heap file or the pile file Organisation are:

- New records can be inserted in any empty space that can accommodate them.
- When old records are deleted, the occupied space becomes empty and available for any new insertion.
- If updated records grow; they may need to be relocated (moved) to a new empty space. This needs to keep a list of empty space.

#### Advantages of heap files

1. This is a simple file Organisation method.
2. Insertion is somehow efficient.
3. Good for bulk-loading data into a table.
4. Best if file scans are common or insertions are frequent.

#### Disadvantages of heap files

1. Retrieval requires a linear search and is inefficient.
2. Deletion can result in unused space/need for reorganisation.

### 4.4.2 Sequential File Organisation

The most basic way to organise the collection of records in a file is to use sequential Organisation. Records of the file are stored in sequence by the primary key field values. They are accessible only in the order stored, i.e., in the primary key order. This kind of file Organisation works well for tasks which need to access nearly every record in a file, e.g., payroll. Let us see the advantages and disadvantages of it.
In a sequentially organised file records are written consecutively when the file is created and must be accessed consecutively when the file is later used for input *(Figure 2)*.

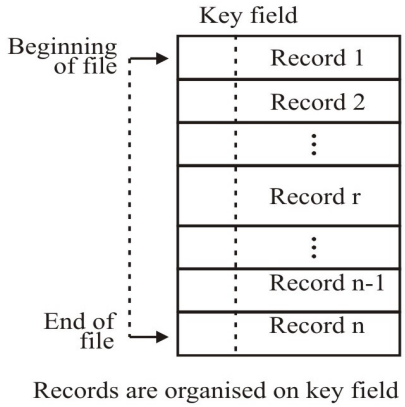


**Figure 2: Structure of sequential file**

A sequential file maintains the records in the logical sequence of its primary key values. Sequential files are inefficient for random access, however, are suitable for sequential access. A sequential file can be stored on devices like magnetic tape that allow sequential access.

On an average, to search a record in a sequential file would require to look into half of the records of the file. However, if a sequential file is stored on a disk (remember disks support direct access of its blocks) with keyword stored separately from the rest of record, then only those disk blocks need to be read that contains the desired record

or records. This type of storage allows binary search on sequential file blocks, thus, enhancing the speed of access.

Updating a sequential file usually creates a new file so that the record sequence on primary key is maintained. The update operation first copies the records till the record after which update is required into the new file and then the updated record is put followed by the remainder of records. Thus method of updating a sequential file automatically creates a backup copy.

Additions in the sequential files are also handled in a similar manner to update. Adding a record requires shifting of all records from the point of insertion to the end of file to create space for the new record. On the other hand deletion of a record requires a compression of the file space.

The basic advantages of sequential file is the sequential processing, as next record is easily accessible despite the absence of any data structure. However, simple queries are time consuming for large files. A single update is expensive as new file must be created, therefore, to reduce the cost per update, all updates requests are sorted in the order of the sequential file. This update file is then used to update the sequential file in a single go. The file containing the updates is sometimes referred to as a transaction file.

This process is called the batch mode of updating. In this mode each record of master sequential file is checked for one or more possible updates by comparing with the update information of transaction file. The records are written to new master file in the sequential manner. A record that require multiple update is written only when all the updates have been performed on the record. A record that is to be deleted is not written to new master file. Thus, a new updated master file will be created from the transaction file and old master file.

Thus, update, insertion and deletion of records in a sequential file require a new file creation. Can we reduce creation of this new file? Yes, it can easily be done if the original sequential file is created with holes which are empty records spaces as shown in the *Figure 3*. Thus, a reorganisation can be restricted to only a block that can be done very easily within the main memory. Thus, holes increase the performance of sequential file insertion and deletion. This organisation also support a concept of overflow area, which can store the spilled over records if a block is full. This technique is also used in index sequential file organisation. A detailed discussion on it can be found in the further readings.

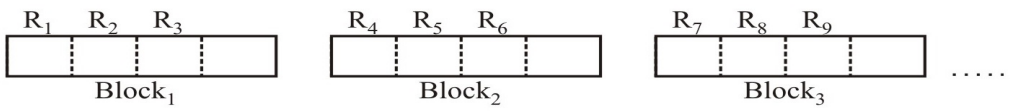


**Figure 3: A file with empty spaces for record insertions**

## Advantages of Sequential File Organisation

- It is fast and efficient when dealing with large volumes of data that need to be processed periodically (batch system).

## Disadvantages of sequential File Organisation

- Requires that all new transactions be sorted into the proper sequence for sequential access processing.
- Locating, storing, modifying, deleting, or adding records in the file require rearranging the file.
- This method is too slow to handle applications requiring immediate updating or responses.

### 4.4.3 Indexed (Indexed Sequential) File Organisation

It organises the file like a large dictionary, i.e., records are stored in order of the key but an index is kept which also permits a type of direct access. The records are stored sequentially by primary key values and there is an index built over the primary key field.

The retrieval of a record from a sequential file, on average, requires access to half the records in the file, making such inquiries not only inefficient but very time consuming for large files. To improve the query response time of a sequential file, a type of indexing technique can be added.

An index is a set of index value, address pairs. Indexing associates a set of objects to a set of orderable quantities, that are usually smaller in number or their properties. Thus, an index is a mechanism for faster search. Although the indices and the data blocks are kept together physically, they are logically distinct. Let us use the term an index file to describes the indexes and let us refer to data files as data records. An index can be small enough to be read into the main memory.

A sequential (or sorted on primary keys) file that is indexed on its primary key is called an index sequential file. The index allows for random access to records, while the sequential storage of the records of the file provides easy access to the sequential records. An additional feature of this file system is the over flow area. The overflow area provides additional space for record addition without the need to create.

### 4.4.4 Hashed File Organisation

Hashing is the most common form of purely random access to a file or database. It is also used to access columns that do not have an index as an optimisation technique. Hash functions calculate the address of the page in which the record is to be stored based on one or more fields in the record. The records in a hash file appear randomly distributed across the available space. It requires some hashing algorithm and the technique. Hashing Algorithm converts a primary key value into a record address. The most popular form of hashing is division hashing with chained overflow.

**Advantages of Hashed file Organisation**

1. Insertion or search on hash-key is fast.
2. Best if equality search is needed on hash-key.

**Disadvantages of Hashed file Organisation**

1. It is a complex file Organisation method.
2. Search is slow.
3. It suffers from disk space overhead.
4. Unbalanced buckets degrade performance.
5. Range search is slow.

## Check Your Progress 1

1) Mention the five operations which show the performance of a sequential file Organisation along with the comments.

……

……

2) What are Direct-Access systems? What can be the various strategies to achieve this?

……

……

……

3) What is file Organisation and what are the essential factors that are to be considered?

……

……

……

……

## 4.5 TYPES OF INDEXES

One of the term used during the file organisation is the term index. In this section, let us define this term in more detail.

We find the index of keywords at the end of each book. Notice that this index is a sorted list of keywords (index values) and page numbers (address) where the keyword can be found. In databases also an index is defined in a similar way, as the <index value, address> pair.

The basic advantage of having sorted index pages at the end of the book is that we can locate a desired keyword in the book. We could have used the topic and sub-topic listed in the table of contents, but it is not necessary that the given keyword can be found there; also they are not in any sorted sequence. If a keyword is not even found in the table of contents then we need to search each of the pages to find the required keyword, which is going to be very cumbersome. Thus, an index at the back of the book helps in locating the required keyword references very easily in the book.

The same is true for the databases that have very large number of records. A database index allows fast search on the index value in database records. It will be difficult to locate an attribute value in a large database, if index on that value is not provided. In such a case the value is to be searched record-by-record in the entire database which is cumbersome and time consuming. It is important to note here that for a large database the entire records cannot be kept in the main memory at a time, thus, data needs to be transferred from the secondary storage device which is more time consuming. Thus, without an index it may be difficult to search a database.

An index contains a pair consisting of index value and a list of pointers to disk block for the records that have the same index value. An index contains such information for every stored value of index attribute. An index file is very small compared to a data file that stores a relation. Also index entries are ordered, so that an index can be searched using an efficient search method like binary search. In case an index file is very large, we can create a multi-level index, that is index on index. Multi-level indexes are defined later in this section.

There are many types of indexes those are categorised as:

| | | |
|---|---|---|
| Primary index | Single level index | Spare index |
| Secondary index | Multi-level index | Dense index |
| Clustering index | | |

A primary index is defined on the attributes in the order of which the file is stored. This field is called the ordering field. A primary index can be on the primary key of a file. If an index is on attributes other than candidate key fields then several records may be related to one ordering field value. This is called clustering index. It is to be noted that there can be only one physical ordering field. Thus, a file can have either the primary index or clustering index, not both. Secondary indexes are defined on the

non-ordering fields. Thus there can be several secondary indexes in a file, but only one primary or clustering index.

## Primary index

A primary index is a file that contains a sorted sequence of records having two columns: the ordering key field; and a block address for that key field in the data file. The ordering key field for this index can be the primary key of the data file. Primary index contains one index entry for each value of the ordering key field. An entry in primary index file contains the index value of the first record of the data block and a pointer to that data block.

Let us discuss primary index with the help of an example. Let us assume a student database as (Assuming that one block stores only four student records.):

| | Enrolment Number | Name | City | Progra-mme |
|---|---|---|---|---|
| **BLOCK 1** | 2109348 | ANU VERMA | CHENNAI | CIC |
| | 2109349 | ABHISHEK KUMAR | CALCUTTA | MCA |
| | 2109351 | VIMAL KISHOR | KOCHI | BCA |
| | 2109352 | RANJEETA JULIE | KOCHI | CIC |
| **BLOCK 2** | 2109353 | MISS RAJIYA BANU | VARANASI | MBA |
| | 2238389 | JITENDAR KASWAN | NEW DELHI | MBA |
| | 2238390 | RITURAJ BHATI | VARANASI | MCA |
| | 2238411 | AMIT KUMAR JAIN | NEW DELHI | BCA |
| **BLOCK 3** | 2238412 | PAWAN TIWARI | AJMER | MCA |
| | 2238414 | SUPRIYA SWAMI | NEW DELHI | MCA |
| | 2238422 | KAMLESH KUMAR | MUMBAI | BSC |
| | 2258014 | DAVEN SINGHAL | MUMBAI | BCA |
| **BLOCK 4** | 2258015 | S SRIVASTAVA | MUMBAI | BCA |
| | 2258017 | SHWETA SINGH | NEW DELHI | BSC |
| | 2258018 | ASHISH TIWARI | MUMBAI | MCA |
| | 2258019 | SEEMA RANI | LUCKNOW | MBA |
| … | … | … | … | … |
| **BLOCK r** | 2258616 | NIDHI | AJMER | BCA |
| | 2258617 | JAGMEET SINGH | LUCKNOW | MCA |
| | 2258618 | PRADEEP KUMAR | NEW DELHI | BSC |
| | 2318935 | RAMADHAR | FARIDABAD | MBA |
| … | … | … | … | … |
| **BLOCK N-1** | 2401407 | BRIJMISHRA | BAREILLY | CIC |
| | 2401408 | AMIT KUMAR | BAREILLY | BSC |
| | 2401409 | MD. IMRAN SAIFI | AURANGABAD | BCA |
| | 2401623 | ARUN KUMAR | NEW DELHI | MCA |
| **BLOCK N** | 2401666 | ABHISHEK RAJPUT | MUMBAI | MCA |
| | 2409216 | TANNUJ SETHI | LUCKNOW | MBA |
| | 2409217 | SANTOSH KUMAR | ALMORA | BCA |
| | 2409422 | SAKSHI GINOTRA | MUMBAI | BSC |

**Figure 4: A Student file stored in the order of student enrolment numbers**

The primary index on this file would be on the ordering field – enrolment number. The primary index on this file would be:

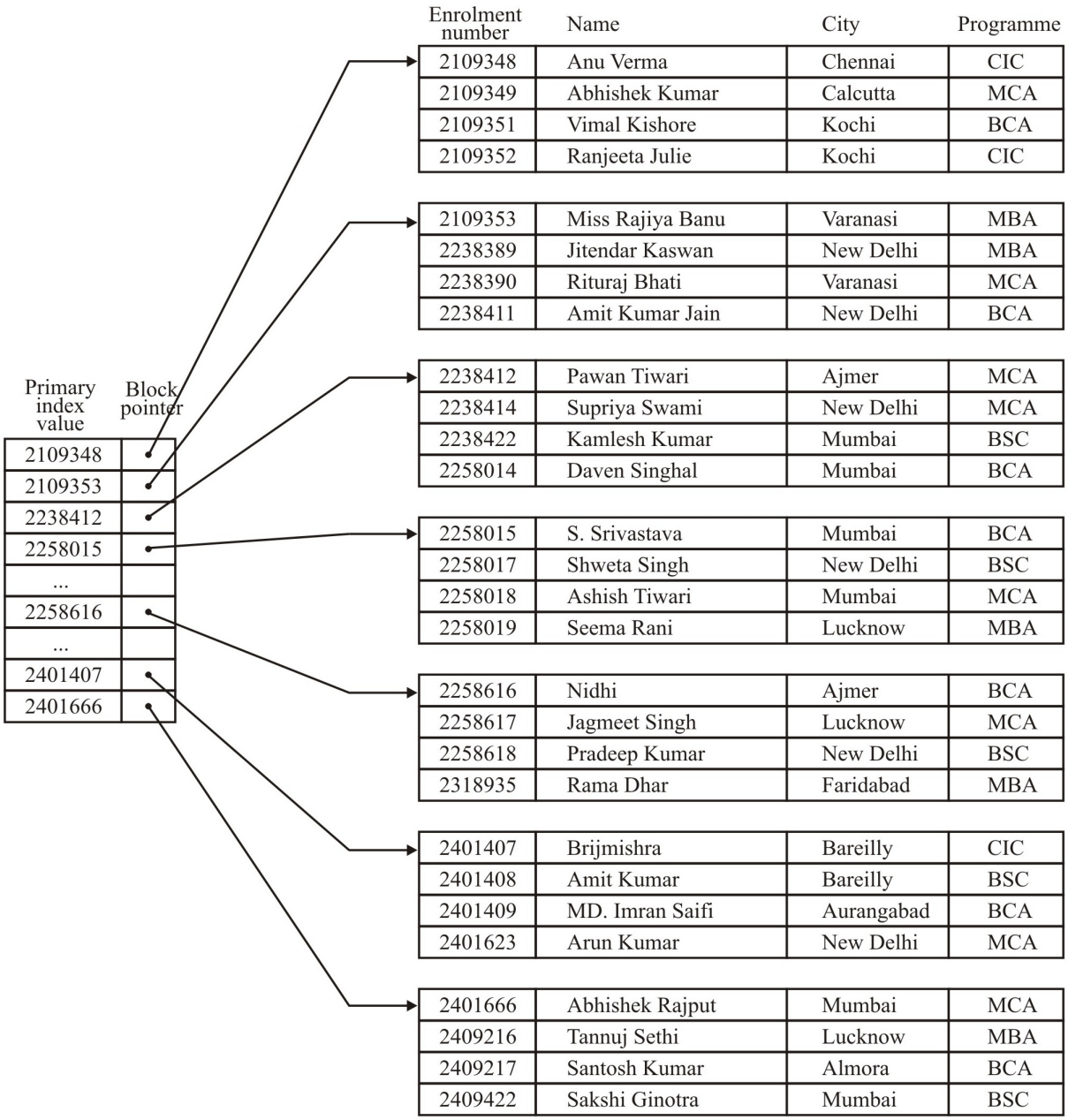


**Figure 5: The Student file and the Primary Index on Enrolment Number**

Please note the following points in the figure above.

- An index entry is defined as the attribute value, pointer to the block where that record is stored. The pointer physically is represented as the binary address of the block.
- Since there are four student records, which of the key value should be stored as the index value? We have used the first key value stored in the block as the index key value. This is also called the anchor value. All the records stored in the given block have ordering attribute value as the same or more than this anchor value.
- The number of entries in the primary index file is the same as the number of disk block in the ordered data file. Therefore, the size of the index file is small. Also notice that not all the records need to have an entry in the index file. This type of index is called non-dense index. Thus, the primary index is non-dense index.

- To locate the record of a student whose enrolment number is 2238422, we need to find two consecutive entries of indexes such that index value1 < 2238422 < index value 2. In the figure above we find the third and fourth index values as: 2238412 and 2258015 respectively satisfying the properties as above. Thus, the required student record must be found in Block 3.

But, does primary index enhance efficiency of searching? Let us explain this with the help of an example (Please note we will define savings as the number of block transfers as that is the most time consuming operation during searching).

**Example 1:** An ordered student file (ordering field is enrolment number) has 20,000 records stored on a disk having the Block size as 1 K. Assume that each student record is of 100 bytes, the ordering field is of 8 bytes, and block pointer is also of 8 bytes, find how many block accesses on average may be saved on using primary index.

**Answer**:

**Number of accesses without using Primary Index:**

Number of records in the file = 20000
Block size = 1024 bytes
Record size = 100 bytes
Number of records per block = integer value of [1024 / 100] = 10
Number of disk blocks acquired by the file = [Number of records / records per block]
= [20000/10] = 2000
Assuming a block level binary search, it would require $\log_2 2000$ = about11 block accesses.

**Number of accesses with Primary Index:**

Size of an index entry = 8+8 = 16 bytes

Number of index entries that can be stored per block

= integer value of [1024 / 16] = 64

Number of index entries = number of disk blocks = 2000

Number of index blocks = ceiling of [2000/ 64] = 32

Number of index block transfers to find the value in index blocks = $\log_2 32 = 5$

One block transfer will be required to get the data records using the index pointer after the required index value has been located. So total number of block transfers with primary index = 5 + 1 = 6.

Thus, the Primary index would save about 5 block transfers for the given case.

Is there any disadvantage of using primary index? Yes, a primary index requires the data file to be ordered, this causes problems during insertion and deletion of records in the file. This problem can be taken care of by selecting a suitable file organisation that allows logical ordering only.

**Clustering Indexes.**

It may be a good idea to keep records of the students in the order of the programme they have registered as most of the data file accesses may require student data of a particular programme only. An index that is created on an ordered file whose records of a file are physically ordered on a non-key field (that is the field does not have a distinct value for each record) is called a clustering index. *Figures 6 & 7* show the clustering indexes in the same file organised in different ways.

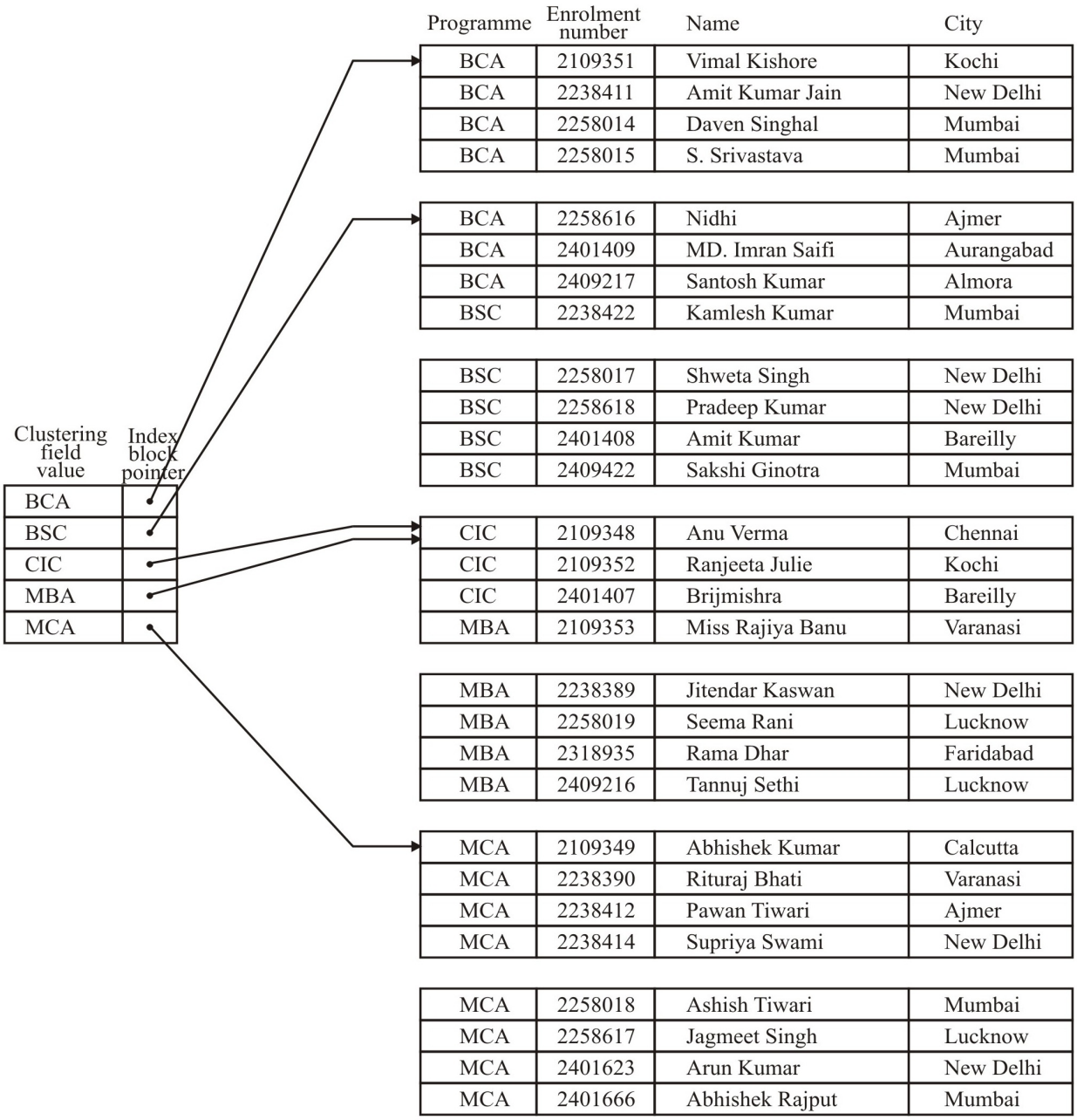


**Figure 6: A clustering Index on Programme in the Student file**

Please note the following points about the clustering index as shown in the *Figure 6*.

- The clustering index is an ordered file having the clustering index value and a block pointer to the first block where that clustering field value first appears.
- Clustering index is also a sparse index. The size of clustering index is smaller than primary index as far as number of entries is concerned.

In the Figure 6 the data file have blocks where multiple Programme students exist. We can improve upon this organisation by allowing only one Programme data in one block. Such an organisation and its clustering index is shown in the following *Figure 7*:

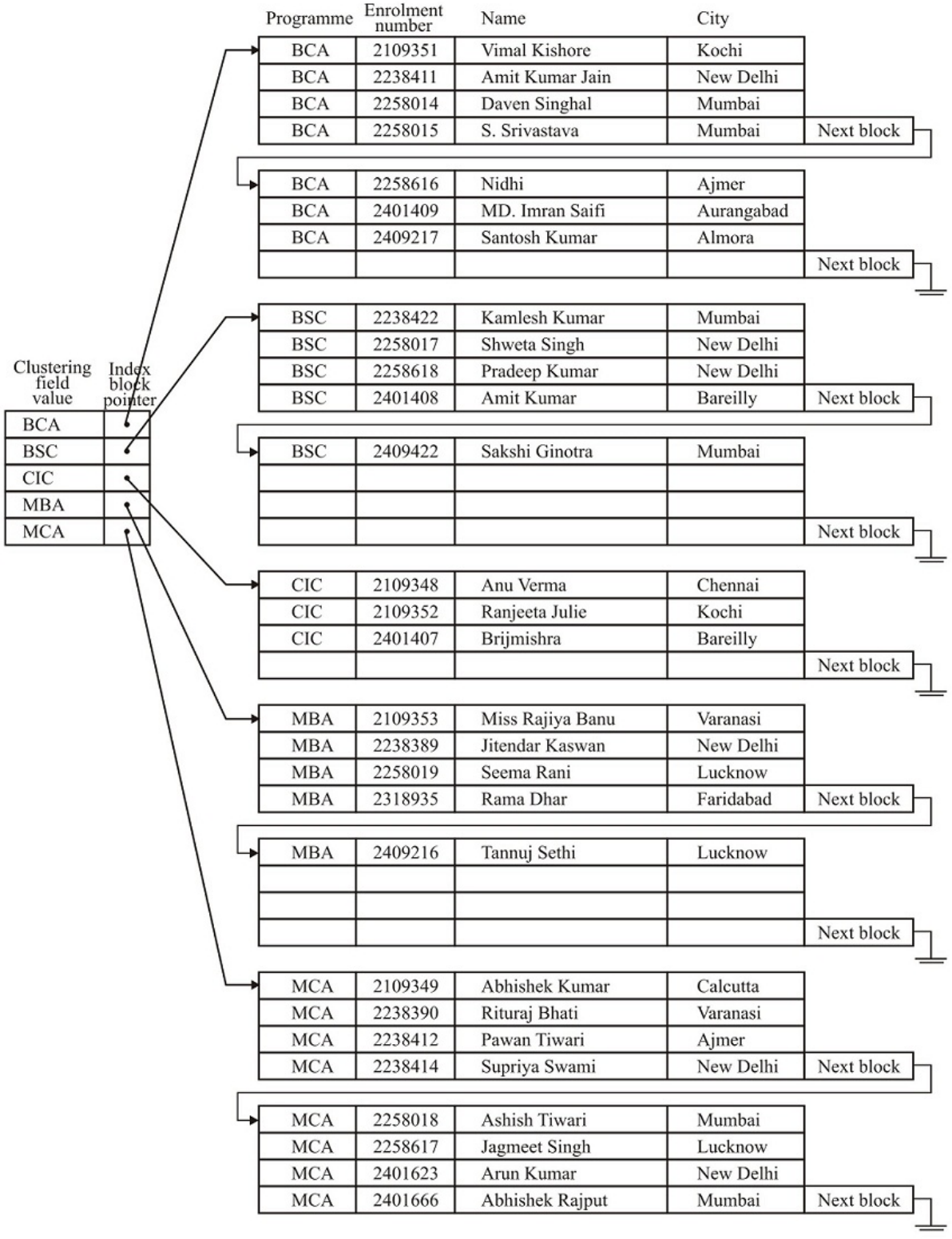


**Figure 7: Clustering index with separate blocks for each group of records with the same value for the clustering field**

Please note the following points in the tables:

- Data insertion and deletion is easier than in the earlier clustering files, even now it is cumbersome.
- The additional blocks allocated for an index entry are in the form of linked list blocks.

- Clustering index is another example of a non-dense index as it has one entry for every distinct value of the clustering index field and not for every record in the file.

**Secondary Indexes**

Consider the student database and its primary and clustering index (only one will be applicable at a time). Now consider the situation when the database is to be searched or accessed in the alphabetical order of names. Any search on a student name would require sequential data file search, thus, is going to be very time consuming. Such a search on an average would require reading of half of the total number of blocks. Thus, we need secondary indices in database systems. A secondary index is a file that contains records containing a secondary index field value which is not the ordering field of the data file, and a pointer to the block that contains the data record. Please note that although a data file can have only one primary index (as there can be only one ordering of a database file), it can have many secondary indices.

Secondary index can be defined on an alternate key or non-key attributes. A secondary index that is defined on the alternate key will be dense while secondary index on non-key attributes would require a bucket of pointers for one index entry. Let us explain them in more detail with the help of *Figures 8*.

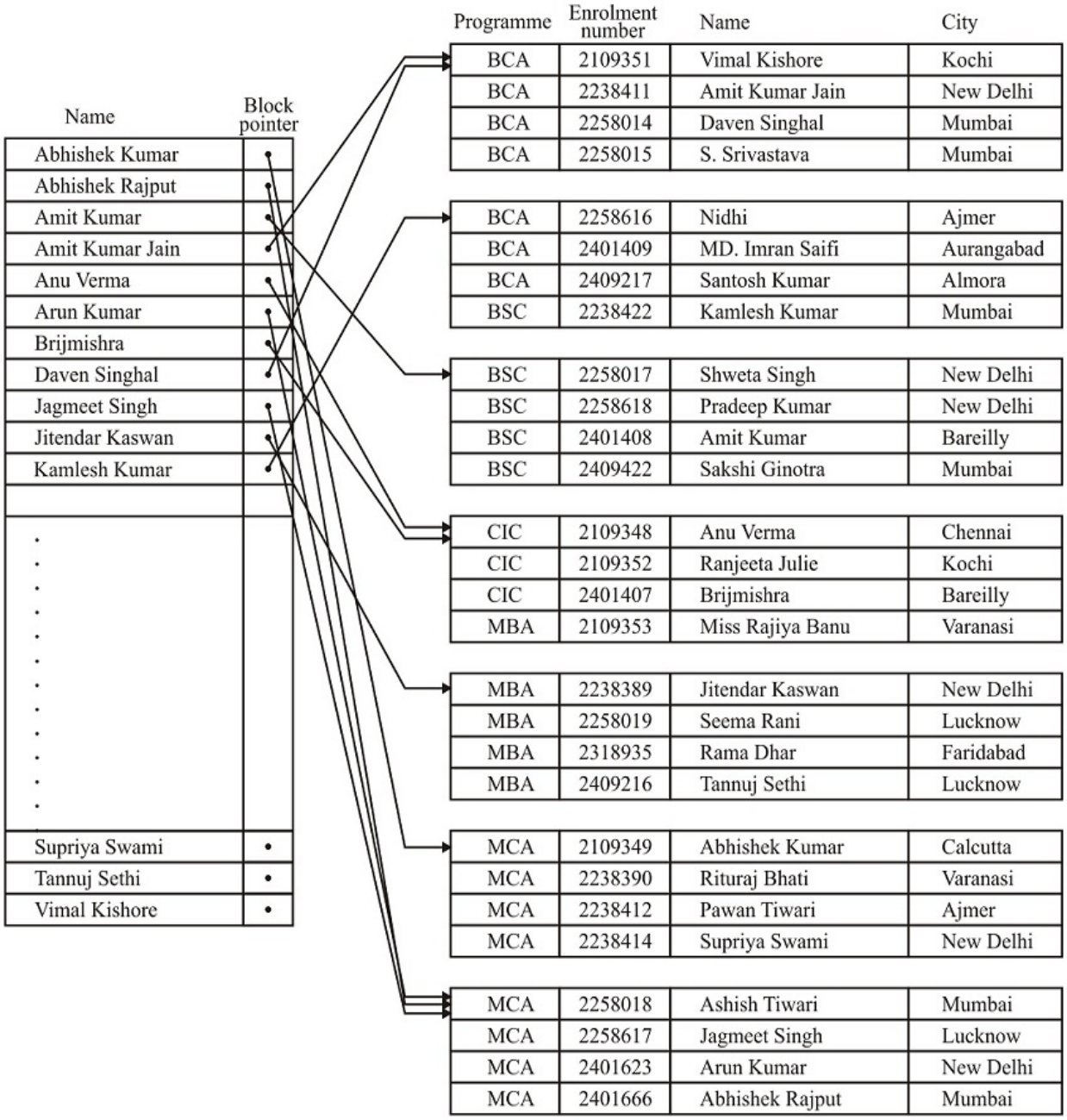




**Figure 8: A dense secondary Index on a non-ordering key field of a file (The file is organised on the clustering field "Programme"**

Please note the following in the *Figure 8*.

- The names in the data file are unique and thus are being assumed as the alternate key. Each name therefore is appearing as the secondary index entry.
- The pointers are block pointers, thus are pointing to the beginning of the block and not a record. For simplicity of the figure we have not shown all the pointers
- This type of secondary index file is dense index as it contains one entry for each record/district value.
- The secondary index is larger than the Primary index as we cannot use block anchor values here as the secondary index attributes are not the ordering attribute of the data file.
- To search a value in a data file using name, first the index file is (binary) searched to determine the block where the record having the desired key value can be found. Then this block is transferred to the main memory where the desired record is searched and accessed.
- A secondary index file is usually larger than that of primary index because of its larger number of entries. However, the secondary index improves the search time to a greater proportion than that of primary index. This is due to the fact that if primary index does not exist even then we can use binary search on the blocks as the records are ordered in the sequence of primary index value. However, if a secondary key does not exist then you may need to search the records sequentially. This fact is demonstrated with the help of Example 2.

**Example 2:** Let us reconsider the problem of Example 1 with a few changes. An un-ordered student file has 20,000 records stored on a disk having the Block size as 1 K. Assume that each student record is of 100 bytes, the secondary index field is of 8 bytes, and block pointer is also of 8 bytes, find how many block accesses on average may be saved on using secondary index on enrolment number.

**Answer:**

**Number of accesses without using Secondary Index:**

Number of records in the file = 20000
Block size = 1024 bytes
Record size = 100 bytes
Number of records per block = integer value of [1024 / 100] = 10
Number of disk blocks acquired by the file = [Number of records / records per block]
= [20000/10] = 2000
Since the file is un-ordered any search on an average will require about half of the above blocks to be accessed. Thus, average number of block accesses = 1000

**Number of accesses with Secondary Index:**

Size of an index entry = 8+8 = 16 bytes
Number of index entries that can be stored per block
= integer value of [1024 / 16] = 64
Number of index entries = number of records = 20000
Number of index blocks = ceiling of [20000/ 64] = 320
Number of index block transfers to find the value in index blocks =
ceiling of $[\log_2 320] = 9$

One block transfer will be required to get the data records using the index pointer after the required index value has been located. So total number of block transfers with secondary index = 9 + 1 = 10

Thus, the Secondary index would save about 1990 block transfers for the given case. This is a huge saving compared to primary index. Please also compare the size of secondary index to primary index.

Let us now see an example of a secondary index that is on an attribute that is not an alternate key.

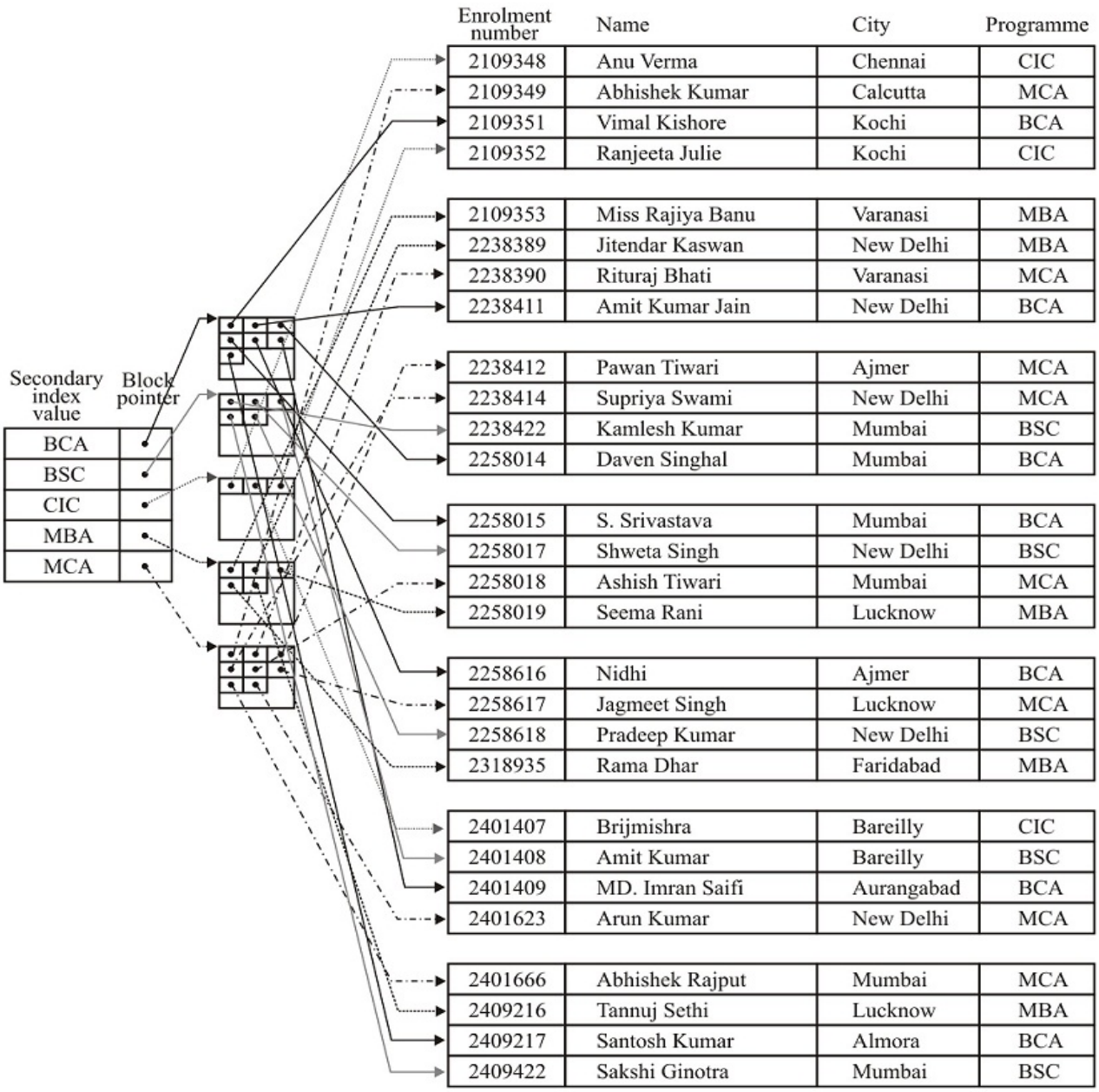


**Figure 9: A secondary index on a non-key field implemented using one level of indirection so that index entries are fixed length and have unique field values (The file is organised on the primary key.**

A secondary index that needs to be created on a field that is not a candidate key can be implemented using several ways. We have shown here the way in which a block of pointer records is kept for implementing such index. This method keeps the index entries at a fixed length. It also allows only a single entry for each index field value. However, this method creates an extra level of indirection to handle the multiple pointers. The algorithms for searching the index, inserting and deleting new values into an index are very simple in such a scheme. Thus, this is the most popular scheme for implementing such secondary indexes.

### Sparse and Dense Indexes

As discussed earlier an index is defined as the ordered <index value, address> pair. These indexes in principle are the same as that of indexes used at the back of the book. The key ideas of the indexes are:

- They are sorted on the order of the index value (ascending or descending) as per the choice of the creator.
- The indexes are logically separate files (just like separate index pages of the book).
- An index is primarily created for fast access of information.
- The primary index is the index on the ordering field of the data file whereas a secondary index is the index on any other field, thus, more useful.

## But what are sparse and dense indexes?

Sparse indices are those indices that do not include all the available values of a field. An index groups the records as per the index values. A sparse index is the one where the size of the group is one or more, while in a dense index the size of the group is 1. In other words a dense index contains one index entry for every value of the indexing attributes, whereas a sparse index also called non-dense index contains few index entires out of the available indexing attribute values. For example, the primary index on enrolment number is sparse, while secondary index on student name is dense.

### Multilevel Indexing Scheme

Consider the indexing scheme where the address of the block is kept in the index for every record, for a small file, this index would be small and can be processed efficiently in the main memory. However, for a large file the size of index can also be very large. In such a case, one can create a hierarchy of indexes with the lowest level index pointing to the records, while the higher level indexes point to the indexes on indexes. The following figure shows this scheme.

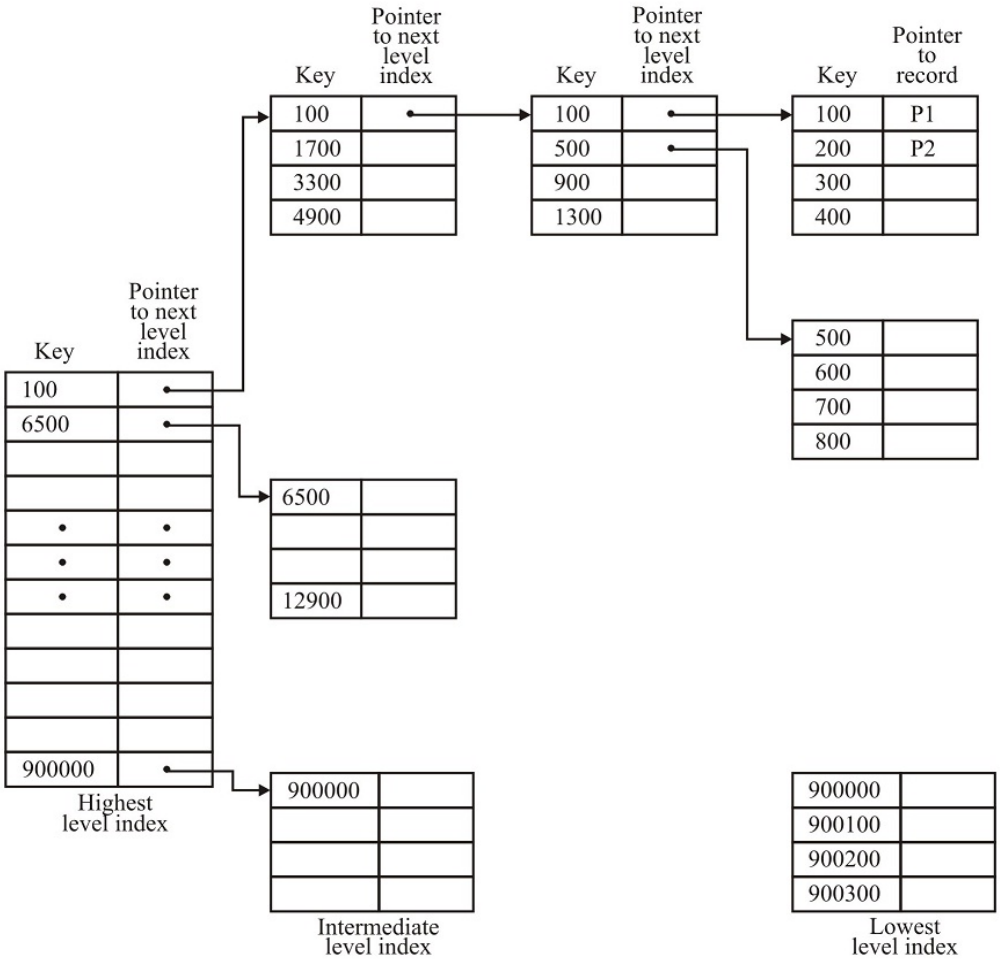


**Figure 10: Hierarchy of Indexes**

Please note the following points about the multi-level indexes:

- The lowest level index points to each record in the file; thus is costly in terms of space.
- Updating, insertion and deletion of records require changes to the multiple index files as well as the data file. Thus, maintenance of the multi-level indexes is also expensive.

After discussing so much about the indexes let us now turn our attention to how an index can be implemented in a database. The indexes are implemented through B Trees. The following section examines the index implementation in more detail.

## 4.6 INDEX AND TREE STRUCTURE

Let us discuss the data structure that is used for creating indexes.

**Can we use Binary Search Tree (BST) as Indexes?**

Les us first reconsider the binary search tree. A BST is a data structure that has a property that all the keys that are to the left of a node are smaller than the key value of the node and all the keys to the right are larger than the key value of the node.

To search a typical key value, you start from the root and move towards left or right depending on the value of key that is being searched. Since an index is a <value, address> pair, thus while using BST, we need to use the value as the key and address field must also be specified in order to locate the records in the file that is stored on the secondary storage devices. The following figure demonstrates the use of BST index for a University where a dense index exists on the enrolment number field.

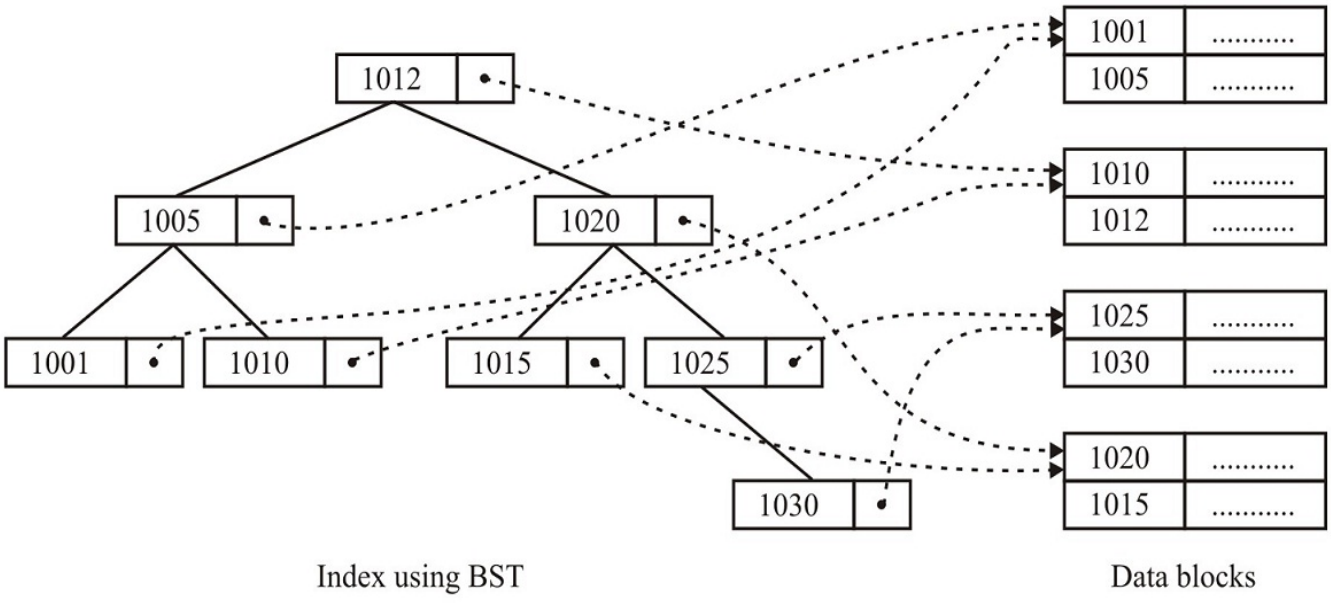


**Figure 11: The Index structure using Binary Search Tree**

Please note in the figure above that a key value is associated with a pointer to a record. A record consists of the key value and other information fields. However, we don't store these information fields in the binary search tree, as it would make a very large tree. Thus, to speed up searches and to reduce the tree size, the information fields of records are commonly stored into files on secondary storage devices. The connection between key values in the BST to its corresponding record in the file is established with the help of a pointer as shown in *Figure11*. Please note that the BST structure is key value, address pair.

Now, let us examine the suitability of BST as a data structure to implement index. A BST as a data structure is very much suitable for an index, if an index is to be contained completely in the primary memory. However, indexes are quite large in nature and require a combination of primary and secondary storage. As far as BST is concerned it might be stored level by level on a secondary storage which would require the additional problem of finding the correct sub-tree and also it may require a number of transfers, with the worst condition as one block transfer for each level of a tree being searched. This situation can be drastically remedied if we use B -Tree as data structure.

A B-Tree as an index has two advantages:

- It is completely balanced
- Each node of B-Tree can have a number of keys. Ideal node size would be if it is somewhat equal to the block size of secondary storage.

The question that needs to be answered here is what should be the order of B-Tree for an index. It ranges from 80-200 depending on various index structures and block size.

Let us recollect some basic facts about B-Trees indexes.

The basic B-tree structure was discovered by R.Bayer and E.McCreight (1970) of Bell Scientific Research Labs and has become one of the popular structures for organising an index structure. Many variations on the basic B-tree structure have been developed.

The B-tree is a useful balanced sort-tree for external sorting. There are strong uses of B-trees in a database system as pointed out by D. Comer (1979): "While no single scheme can be optimum for all applications, the techniques of organising a file and its index called the B-tree is the standard Organisation for indexes in a database system."

A B-tree of order N is a tree in which:

- Each node has a maximum of N children and a minimum of the ceiling of [N/2] children. However, the root node of the tree can have 2 to N children.
- Each node can have one fewer keys than the number of children, but a maximum of N-1 keys can be stored in a node.
- The keys are normally arranged in an increasing order. All keys in the sub tree to the left of a key are less than the key, and all the keys in the sub-tree to the right of a key are higher then the value of the key.
- If a new key is inserted into a full node, the node is split into two nodes, and the key with the median value is inserted in the parent node. If the parent node is the root, a new root node is created.
- All the leaves of B-tree are on the same level. There is no empty sub-tree above the level of the leaves. Thus a B-tree is completely balanced.

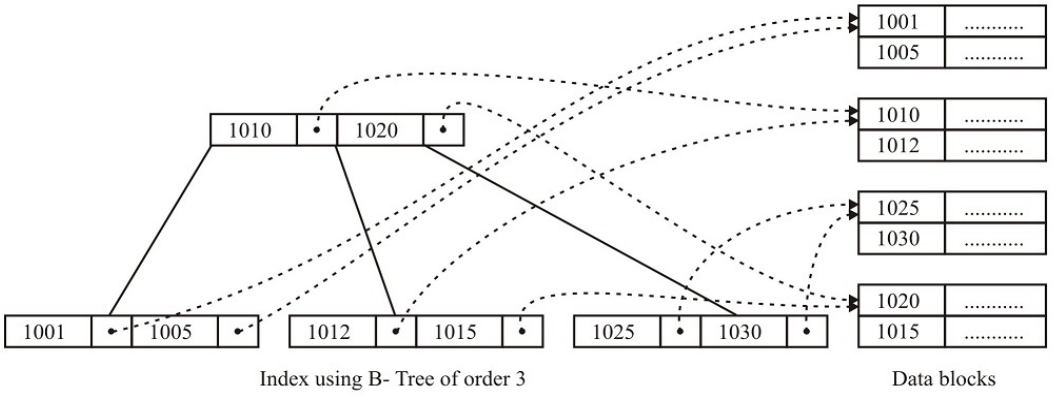


**Figure 12: A B-Tree as an index**

A B-Tree index is shown in Figure 12. The B-Tree has a very useful variant called B+Tree, which have all the key values at the leaf level also in addition to the higher level. For example, the key value 1010 in *Figure 12* will also exist at leaf level. In addition, these lowest level leaves are linked through pointers. Thus, B+tree is a very useful structure for index-sequential organisation. You can refer to further readings for more detail on these topics.

# 4.7 MULTI-KEY FILE ORGANISATION

Till now we have discussed file organisations having the single access key. But is it possible to have file organisations that allows access of records on more than one key field? In this section, we will introduce two basic file Organisation schemes that allow records to be accessed by more than one key field, thus, allowing multiple access paths each having a different key. These are called multi-key file Organisations. These file organisation techniques are at the heart of database implementation.

There are numerous techniques that have been used to implement multi-key file Organisation. Most of these techniques are based on building indexes to provide direct access by the key value. Two of the commonest techniques for this Organisation are:

- Multi-list file Organisation
- Inverted file Organisation

Let us discuss these techniques in more detail. But first let us discuss the need for the Multiple access paths.

### 4.7.1 Need for Multiple Access Paths

In practice, most of the online information systems require the support of multi-key files. For example, consider a banking database application having many different kinds of users such as:

- Teller
- Loan officers
- Brach manager
- Account holders

All of these users access the bank data however in a different way. Let us assume a sample data format for the Account relation in a bank as:

Account Relation:

| Account Number | Account Holder Name | Branch Code | Account type | Balance | Permissible Loan Limit |
|---|---|---|---|---|---|
| | | | | | |

A teller may access the record above to check the balance at the time of withdrawal. S/he needs to access the account on the basis of branch code and account number. A loan approver may be interested in finding the potential customer by accessing the records in decreasing order of permissible loan limits. A branch manager may need to find the top ten most preferred customers in each category of account so may access the database in the order of account type and balance. The account holder may be interested in his/her own record. Thus, all these applications are trying to refer to the same data but using different key values. Thus, all the applications as above require the database file to be accessed in different format and order. What may be the most efficient way to provide faster access to all such applications? Let us discuss two approaches for this:

- By Replicating Data
- By providing indexes.

### Replicating Data

One approach that may support efficient access to different applications as above may be to provide access to each of the applications from different replicated files of the data. Each of the file may be organised in a different way to serve the requirements of a different application. For example, for the problem above, we may provide an indexed sequential account file having account number as the key to bank teller and the account holders. A sequential file in the order of permissible loan limit to the Loan officers and a sorted sequential file in the order of balance to branch manager.
All of these files thus differ in the organisation and would require different replica for different applications. However, the Data replication brings in the problems of inconsistency under updating environments. Therefore, a better approach for data access for multiple keys has been proposed.

### Support by Adding Indexes

Multiple indexes can be used to access a data file through multiple access paths. In such a scheme only one copy of the data is kept, only the number of paths is added with the help of indexes. Let us discuss two important approaches in this category: Multi-List file organisation and Inverted file organisation.

### 4.7.2 Multi-list file Organisation

Multi-list file organisation is a multi-index linked file organisation. A linked file organisation is a logical organisation where physical ordering of records is not of concern. In linked organisation the sequence of records is governed by the links that determine the next record in sequence. Linking of records can be unordered but such a linking is very expensive for searching of information from a file. Therefore, it may be a good idea to link records in the order of increasing primary key. This will facilitate insertion and deletion algorithms. Also this greatly helps the search performance. In addition to creating order during linking, search through a file can be further facilitated by creating primary and secondary indexes. All these concepts are supported in the multi-list file organisation. Let us explain these concepts further with the help of an example.

Consider the employee data as given in *Figure 13*. The record numbers are given as alphabets for better description. Assume that the Empid is the key field of the data records. Let us explain the Multi-list file organisation for the data file.

| Record Number | Empid | Name | Job | Qualification | Gender | City | Married/ Single | Salary |
|---|---|---|---|---|---|---|---|---|
| A | 800 | Jain | Software Engineer | B. Tech. | Male | New Delhi | Single | 15,000/- |
| B | 500 | Inder | Software Manager | B. Tech. | Female | New Delhi | Married | 18,000/- |
| C | 900 | Rashi | Software Manager | MCA | Female | Mumbai | Single | 16,000/- |
| D | 700 | Gurpreet | Software Engineer | B. Tech. | Male | Mumbai | Married | 12,000/- |
| E | 600 | Meena | Software Manager | MCA | Female | Mumbai | Single | 13,000/- |

**Figure 13: Sample data for Employee file**

Since, the primary key of the file is Empid, therefore the linked order of records should be defined as B (500), E(600), D(700), A(800), C(900). However, as the file size will grow the search performance of the file would deteriorate. Therefore, we can create a primary index on the file (please note that in this file the records are in the logical sequence and tied together using links and not physical placement, therefore, the primary index will be a linked index file rather than block indexes).

Let us create a primary index for this file having the Empid values in the range:

>= 500 but < 700
> = 700 but < 900
>= 900 but < 1100

The index file for the example data as per Figure 13 is shown in *Figure 14*.

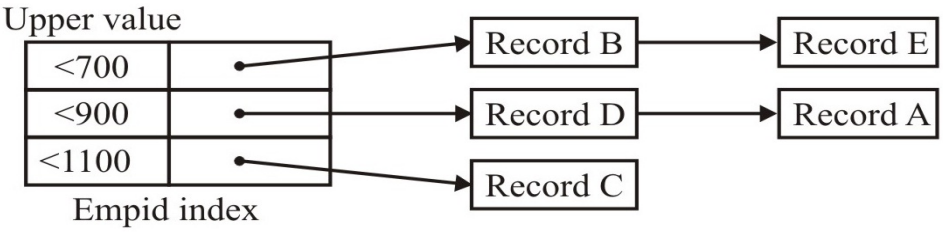


**Figure 14: Linking together all the records in the same index value.**

Please note that in the *Figure 14*, those records that fall in the same index value range of Empid are linked together. These lists are smaller than the total range and thus will improve search performance.

This file can be supported by many more indexes that will enhance the search performance on various fields, thus, creating a multi-list file organisation. *Figure 15* shows various indexes and lists corresponding to those indexes. For simplicity we have just shown the links and not the complete record. Please note the original order of the nodes is in the order of Empid's.

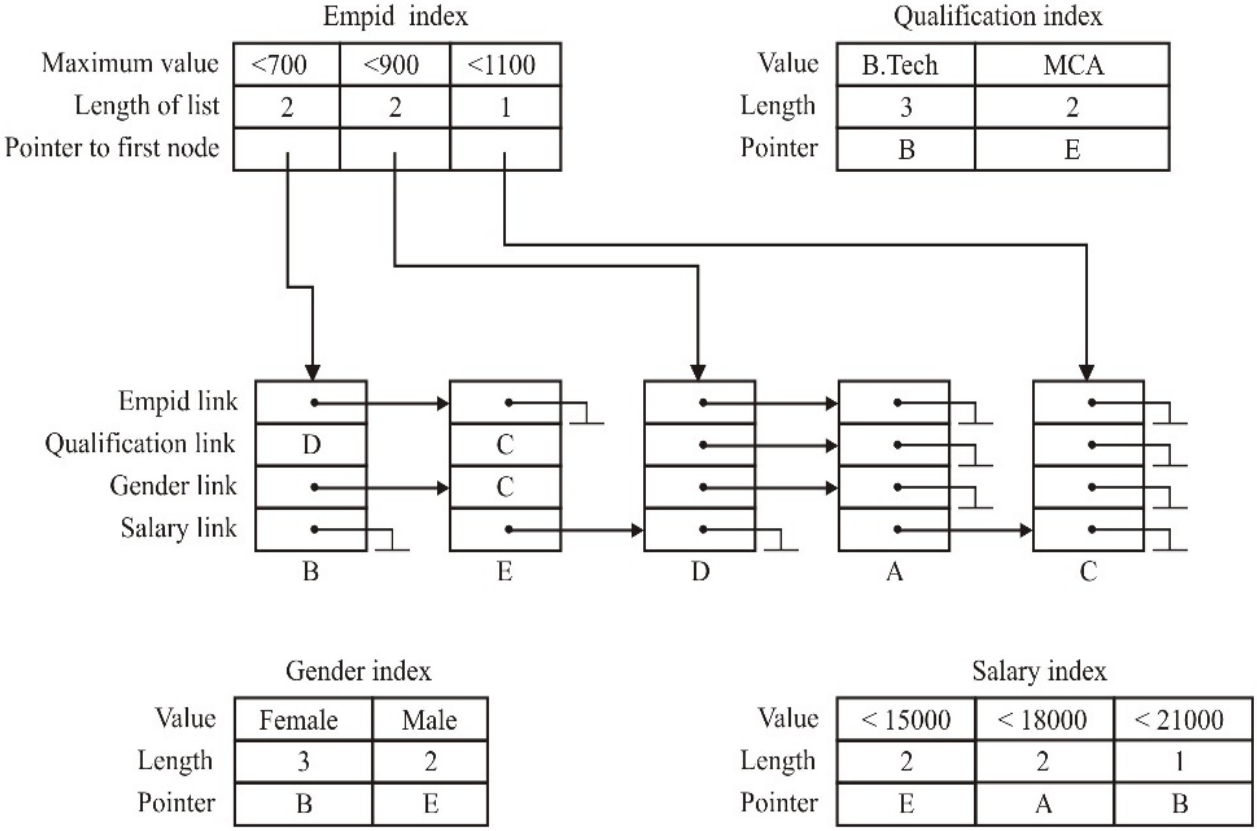


**Figure 15: Multi-list representation for figure 13**

An interesting addition that has been done in the indexing scheme of multi-list organisation is that an index entry contains the length of each sub-list, in addition to the index value and the initial pointer to list. This information is useful when the query contains a Boolean expression. For example, if you need to find the list of Female employees who have MCA qualifications, you can find the results in two ways. Either you go to the Gender index, and search the Female index list for MCA qualification or you search the qualification index to find MCA list and in MCA list search for Female candidates. Since the size of MCA list is 2 while the size of Female list is 3 so the preferable search will be through MCA index list. Thus, the information about the length of the list may help in reducing the search time in the complex queries.

Consider another situation when the Female and MCA lists both are of about a length of 1000 and only 10 Female candidates are MCA. To enhance the search performance of such a query a combined index on both the fields can be created. The values for this index may be the Cartesian product of the two attribute values.

Insertion and deletion into multi-list is as easy/hard as is the case of list data structures. In fact, the performance of this structure is not good under heavy insertion and deletion of records. However, it is a good structure for searching records in case the appropriate index exist.

### 4.7.3 Inverted File Organisation

Inverted file organisation is one file organisation where the index structure is most important. In this organisation the basic structure of file records does not matter much. This file organisation is somewhat similar to that of multi-list file organisation with the key difference that in multi-list file organisation index points to a list, whereas in inverted file organisation the index itself contains the list. Thus, maintaining the proper index through proper structures is an important issue in the design of inverted file organisation. Let us show inverted file organisation with the help of data given in *Figure 13*.

Let us assume that the inverted file organisation for the data shown contains dense index. *Figure 16* shows how the data can be represented using inverted file organisation.

Empid index

| | |
|---|---|
| 500 | B |
| 600 | E |
| 700 | D |
| 800 | A |
| 900 | C |

Qualification index

| | |
|---|---|
| B.Tech | B,C,D |
| MCA | A,E |

Gender index

| | |
|---|---|
| Female | B,C,E |
| Male | A,D |

Salary index

| | |
|---|---|
| 12000 | D |
| 13000 | E |
| 15000 | A |
| 16000 | C |
| 18000 | B |

**Figure 16: Some of the indexes for fully inverted file**

Please note the following points for the inverted file organisation:

- The index entries are of variable lengths as the number of records with the same key value is changing, thus, maintenance of index is more complex than that of multi-list file organisation.
- The queries that involve Boolean expressions require accesses only for those records that satisfy the query in addition to the block accesses needed for the indices. For example, the query about Female, MCA employees can be solved by the Gender and Qualification index. You just need to take intersection of record numbers on the two indices. (Please refer to *Figure 16*). Thus, any complex query requiring Boolean expression can be handled easily through the help of indices.

Let us now finally differentiate between the two-multi-list and inverted file organisation**.**

***Similarities:***

**Both organisations can support:**

- An index for primary and secondary key
- The pointers to data records may be direct or indirect and may or may not be sorted.

***Differences:***

The indexes in the two organisations differ as:

- In a Multi-list organisation an index entry points to the first data record in the list, whereas in inverted index file an index entry has address pointers to all the data records related to it.
- A multi-list index has fixed length records, whereas an inverted index contains variable length records

However, the data records do not change in an inverted file organisation whereas in the multi-list file organisation a record contains the links, one per created index.

***Some of the implications of these differences are:***

- An index in a multi-list organisation can be managed easily as it is of fixed length.
- The query response of inverted file approach is better than multi-list as the query can be answered only by the index. This also helps in reducing block accesses.

## 4.8 IMPORTANCE OF FILE ORGANISATION IN DATABASE

To implement a database efficiently, there are several design tradeoffs needed. One of the most important ones is the file Organisation. For example, if there were to be an application that required only sequential batch processing, then the use of indexing techniques would be pointless and wasteful.

There are several important consequences of an inappropriate file Organisation being used in a database. Thus using replication would be wasteful of space besides posing the problem of inconsistency in the data. The wrong file Organisation can also–

- Mean much larger processing time for retrieving or modifying the required record
- Require undue disk access that could stress the hardware

Needless to say, these could have many undesirable consequences at the user level, such as making some applications impractical.

### Check Your Progress 2

1) What is the difference between BST-Tree and B tree indexes?

……

……

2) Why is a B+ tree a better structure than a B-tree for implementation of an indexed sequential file?

……

……

……

3) **State or True or False** T/ F

a) A primary index is index on the primary key of an unordered data file ☐

b) A clustering index involves ordering on the non-key attributes. ☐

c) A primary index is a dense index. ☐

d) A non-dense index involves all the available attribute values of ☐

the index field.

e) Multi-level indexes enhance the block accesses than simple indexes. ☐

f) A file containing 40,000 student records of 100 bytes each having the 1k-block size. It has a secondary index on its alternate key of size 16 bits per index entry. For this file search through the secondary index will require 20 block transfers on an average. ☐

g) A multi-list file increases the size of the record as the link information is added to each record. ☐

h) An inverted file stores the list of pointers to records within the index. ☐

i) Multi-list organisation is superior than that of inverted organisation for queries having Boolean expression. ☐

## 4.9 SUMMARY

In this unit, we discussed the physical database design issues in which we had addressed the file Organisation and file access method. After that, we discussed the various file Organisations: Heap, Sequential, Indexed Sequential and Hash, their advantages and their disadvantages.

An index is a very important component of a database system as one of the key requirements of DBMS is efficient access to data, therefore, various types of indexes that may exist in database systems were explained in detail. Some of these are: Primary index, clustering index and secondary index. The secondary index results in better search performance, but adds on the task of index updating. In this unit, we also discussed two multi-key file organisations viz. multi-list and inverted file organisations. These are very useful for better query performance.

## 4.10 SOLUTIONS / ANSWERS

### Check Your Progress 1

1)

| Operation | Comments |
|---|---|
| File Creation | It will be efficient if transaction records are ordered by record key |
| Record Location | As it follows the sequential approach it is inefficient. On an average, half of the records in the file must be processed |
| Record Creation | It has to browse through all the records. Entire file must be read and write. Efficiency improves with greater number of additions. This operation could be combined with deletion and modification transactions to achieve greater efficiency. |
| Record Deletion | Entire file must be read and write. Efficiency improves with greater number of deletions. This operation could be combined with addition and modification transactions to achieve greater efficiency. |
| Record Modification | Very efficient if the number of records to be modified is high and the records in the transaction file are ordered by the record key. |

2) Direct-access systems do not search the entire file; rather they move directly to the needed record. To be able to achieve this, several strategies like relative addressing, hashing and indexing can be used.

3) It is a technique for physically arranging the records of a file on secondary storage devices. When choosing the file Organisation, you should consider the following factors:

1. Data retrieval speed
2. Data processing speed
3. Efficiency usage of storage space
4. Protection from failures and data loss
5. Scalability
6. Security

## Check Your Progress 2

1) In a B+ tree the leaves are linked together to form a sequence set; interior nodes exist only for the purposes of indexing the sequence set (not to index into data/records). The insertion and deletion algorithm differ slightly.

2) Sequential access to the keys of a B-tree is much slower than sequential access to the keys of a B+ tree, since the latter have all the keys linked in sequential order in the leave.

3) (a) False
(b) True
(c) False
(d) False
(e) False
(f) False
(g) True
(h) True
(i) False