INTRODUCTION

It is not only important that a database is designed well, but part of the design process also involves ensuring that the structure of the developed system is efficient enough to satisfy users' requirements now and into the future.

Database tuning is a set of techniques that can be used to improve performance. It is an important and complex subject to which a number of units are devoted in this module. The Physical Storage unit laid the foundations, and this unit introduces indexing techniques which are a major subtopic of database tuning. We will be looking into design issues of indexes, and the appropriate ways of using indexes. Indexes play a similar role in database systems as they do in books in that they are used to speed up access to information. File structures can be affected by different indexing techniques, and they in turn will affect the performance of the databases.

It is worth emphasising again the importance of file organisations and their related access methods. Tuning techniques can help improve performance, but only to the extent that is allowed by a particular file organisation. For example, if you have a Ford car, you may obtain a better performance if you have the car's engine tuned. However, no matter how hard you tune it, you will never be able to get a Ferrari's performance out of it.

Indexes can help database developers build efficient file structures and offer effective access methods. When properly used and tuned, the database performance can be improved further. In fact, indexes are probably the single most important mechanism explicitly available to database developers and administrators for tuning the performance of a database.

CONTEXT

Physical file structures are concerned with that how records are organised in files, and based on a specific structure, what access methods may be used. A heap file, for example, provides no special access structure, and therefore, we can only have sequential (linear) access to records. A hash file, on the other hand, organises records using one or more hash functions and the placement of records is based on the values of their hash field. This structure allows us to directly access records if the hash field values are known.

In this document, we will introduce some new access structures called *indexes*, which are used to speed up the retrieval of records if certain requirements on search conditions are met. An index for a file of records works just like an index catalogue in a library. In a normal library environment, for example, there should be catalogues such as author indexes and book title indexes. A user may use one of these indexes to quickly find the location of a required book, if he/she knows the author(s) or title of the book.

Each index (access structure) offers an *access path* to records. Some types of indexes, called *secondary access paths*, do not affect the physical placement of records on disk; rather, they provide alternative search paths for locating the records efficiently based on the indexing fields. Other types of indexes can only be constructed on a file with a particular primary organisation.

In general, any of the structures discussed in Physical Storage can be used to construct a secondary access path. However, the most common types of indexes are based on sorted file organisations (e.g., single-level indexes) and tree data structures (e.g., multilevel indexes, B-trees, and B^+-trees). Indexes can also be created using hashing techniques.

The focus of our study in this document will be on the following:

❑ Primary indexes.
❑ Clustering indexes.
❑ Secondary indexes.
❑ Multilevel indexes.
❑ B-tree and B^+-tree structures.

It must be emphasised that different indexes have their own advantages and disadvantages. There is no universally efficient index. Each technique is best suited for a particular type of database application.

The merits of indexes can be measured in the following aspects:

- ❑ *Access types*: the kind of access methods that can be supported efficiently (e.g., value-based search or range search).
- ❑ *Access time*: time needed to locate a particular record or a set of records.
- ❑ *Insertion efficiency*: how efficient is an insertion operation.
- ❑ *Deletion efficiency*: how efficient is a deletion operation.
- ❑ *Storage overhead*: the additional storage requirement by an index structure.

It is worth noting that a file of records can have more than one index, just like for books there can be different indexes such as author index and title index.

An index access structure is usually constructed on a single field of record in a file, called *an indexing field*. Such an index typically stores each value of the indexing field along with a list of pointers to all disk blocks that contain records with that field value. The values in the index are usually sorted (ordered) so that we can perform an efficient binary search on the index.

To give you an intuitive understanding of an index, we look at library indexes again. For example, an author index in a library will have entries for all authors whose books are stored in the library. AUTHOR is the indexing field and all the names are sorted according to alphabetical order. For a particular author, the index entry will contain locations (i.e., pointers) of all the books this author has written. If you know the name of the author, you will be able to use this index to find his/her books quickly. What happens if you do not have an index to use? This is similar to using a heap file and linear search. You will have to browse through the whole library looking for the book.

An index file is much smaller than the data file, and therefore, searching the index using a binary search can be carried out quickly. Multilevel indexing goes one step further in the sense that it eliminates the need for a binary search by building indexes to the index itself. We will be discussing these techniques later on in the document.

Single-Level Ordered Indexes

There are several types of single-level ordered indexes. A *primary index* is an index specified on the *ordering key field* of a sorted file of records. If the records are sorted *not* on the key field, but on a *non-key field*, an index can still be built which is called a *clustering index*. The difference lies in the fact that different records have different values in the key field, but for a non-key field, some records may share the same value. It must be noted that a file can have at most one physical ordering field. Thus, it can have at most one primary index or one clustering index, *but not both*.

A third type of indexes, called *secondary index*, can be specified on any non-ordering field of a file. A file can have several secondary indexes in addition to its primary access path (i.e., primary index or clustering index). As we mentioned earlier, secondary indexes do not affect the physical organisation of records.

Primary indexes

A *primary index* is built for a file (the *data file*) sorted on its *key field*, and itself is another sorted file (the *index file*) whose records (*index records*) are of fixed-length with two fields. The first field is of the same data type as the ordering key field of the data file, and the second field is a pointer to a disk block (i.e., the block address).

The ordering key field is called the *primary key* of the data file. There is one *index entry* (i.e., 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 use the following notation to refer to an index entry *i* in the index file:

<K(i), P(i)>

K(i) is the primary key value, and P(i) is the corresponding pointer (i.e., block address).

For example, to build a primary index on the sorted file shown in figure 1, we use the ID# as primary key, because that is the ordering key field of the data file. Each entry in the index has an ID# value and a pointer. The first three index entries of such an index file are as follows:

<K(1) = 9701654, P(1) = address of block 1>
<K(2) = 9702381, P(2) = address of block 2>
<K(3) = 9703501, P(3) = address of block 3>

Figure 2 depicts this primary index. The total number of entries in the index is the same as the number of disk blocks in the data file. In this example, there are *b* blocks.

The first record in each block of the data file is called the *anchor record* of that block. A variation to such a primary index scheme is that we could use the last record of a block as the block anchor. However, the two schemes are very similar and there is no significant difference in performance. Thus, it is sufficient to discuss just one of them.

A primary index is an example of a *sparse index* in the sense that it contains an entry for each disk block rather than for every record in the data file. A *dense index*, on the other hand, contains an entry for every data record. In the case of a dense index, it does not require the data file to be a sorted file. Instead, it can be built on any file organisation (typically, a heap file).

By definition, an index file is just a special type of data file of fixed-length records with two fields. We use the term *index file* to refer to data files storing index entries. The general term *data file* is used to refer to files containing the actual data records such as STUDENT.

A file of STUDENT records:

ID# NAME ADDRESS COURSE LEVEL

Block *1*

9701654				
9701890				
......				
9702317				

Block *2*

9702381				
9702399				
......				
9703478				

Block *3*

9703501				
9703569				
......				
9801220				

....

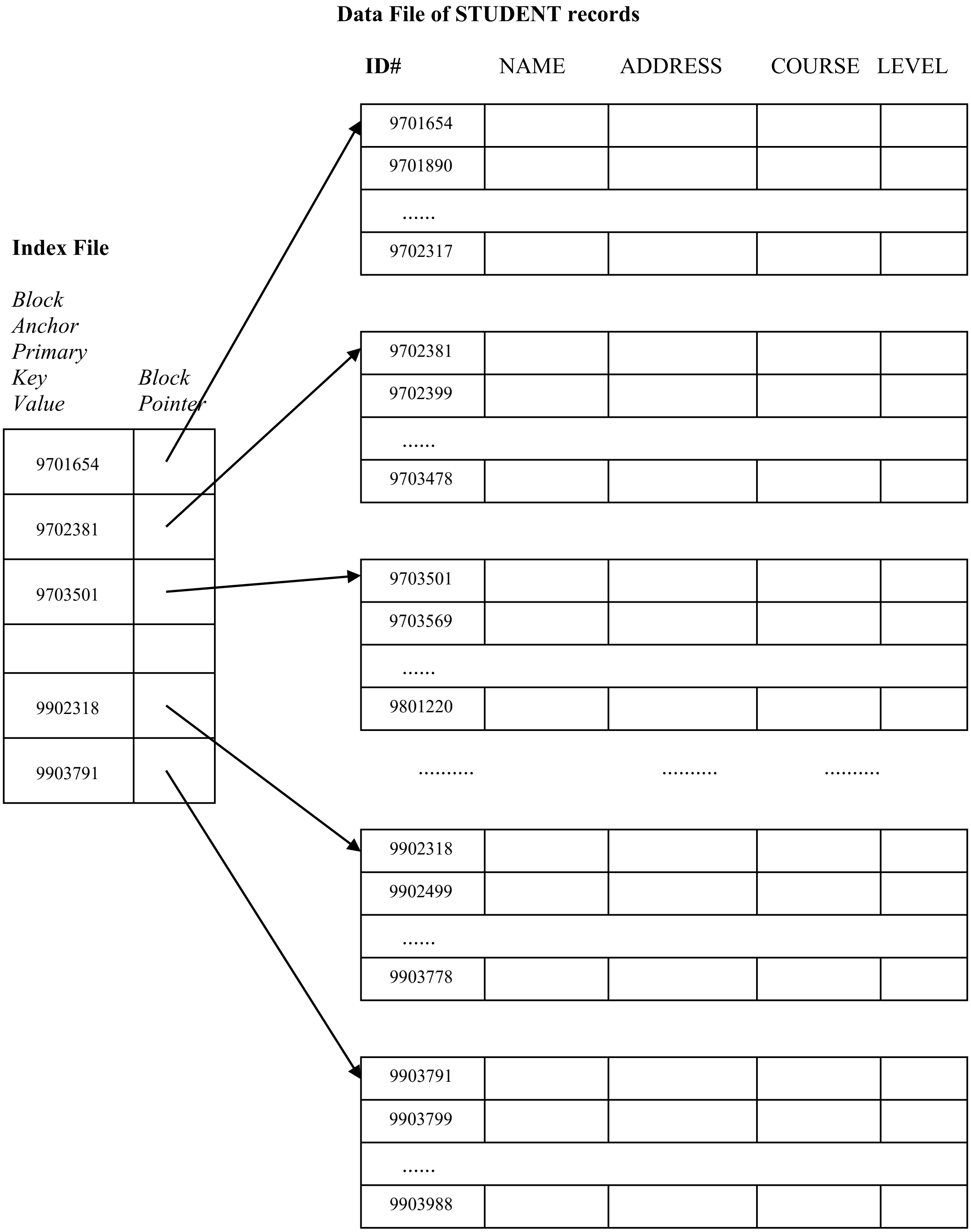
Block *b –1*

9902318				
9902499				
......				
9903778				

Block *b*

9903791				
9903799				
......				
9903988				

Figure 1 A sorted STUDENT file with ID# as the ordering key



Data File of STUDENT records

ID#	NAME	ADDRESS	COURSE	LEVEL
9701654				
9701890				
……				
9702317				

ID#	NAME	ADDRESS	COURSE	LEVEL
9702381				
9702399				
……				
9703478				

ID#	NAME	ADDRESS	COURSE	LEVEL
9703501				
9703569				
……				
9801220				

………. ………. ……….

ID#	NAME	ADDRESS	COURSE	LEVEL
9902318				
9902499				
……				
9903778				

ID#	NAME	ADDRESS	COURSE	LEVEL
9903791				
9903799				
……				
9903988				

Index File

Block Anchor Primary Key Value	*Block Pointer*
9701654	
9702381	
9703501	
9902318	
9903791	

Figure 2 Primary index on the STUDENT file in Figure 1

Performance issues

The index file for a primary index needs significantly fewer blocks than does the file for data records for the following two reasons:

- There are fewer index entries than there are records in the data file, because an entry exists for each block rather than for each record.
- Each index entry is typically smaller in size than a data record because it has only two fields. Consequently more index entries can fit into one block. A binary search on the index file hence requires fewer block accesses than a binary search on the data file.

If a record whose primary key value is K is in the data file, then it has to be in the block whose address is P(i), where $K(i) \le K < K(i+1)$. The *i*th block in the data file contains all such records because of the physical ordering of the records based on the primary key field. For example, look at the first three entries in the index in figure 2

<K(1) = 9701654, P(1) = address of block 1>
<K(2) = 9702381, P(2) = address of block 2>
<K(3) = 9703501, P(3) = address of block 3>

The record with ID#=9702399 is in the 2nd block because $K(2) \le 9702399 < K(3)$. In fact, all the records with an ID# value between K(2) and K(3) must be in block 2, if they are in the data file at all.

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 use the block address contained in the pointer P(i) to retrieve the data block. The following example explains the performance improvement, in terms of the number of block accesses, that can be obtained through the use of a primary index.

Example 1:

Suppose that we have an ordered file with r = 40,000 records stored on a disk with block size B = 1024 bytes. File records are of fixed-length and are unspanned, with a record size R = 100 bytes. The blocking factor for the data file would be $bfr = \lfloor (B/R) \rfloor = \lfloor (1024/100) \rfloor = 10$ records per block. The number of blocks needed for this file is $b = \lceil (r/bfr) \rceil = \lceil (40000/10) \rceil = 4000$ blocks. A binary search on the data file would need approximately $\lceil (\log_2 b) \rceil = \lceil (\log_2 4000) \rceil = 12$ block accesses

Now suppose that the ordering key field of the file is V = 11 bytes long, a block pointer (block address) is P = 8 bytes long, and a primary index has been constructed for the file. The size of each index entry is $R_i = (11 + 8) = 19$ bytes, so the blocking factor for the index file is $bfr_i = \lfloor (B/R_i) \rfloor = \lfloor (1024/19) \rfloor = 53$ entries per block. The total number of index entries r_i is equal to the number of blocks in the data file, which is 4000. Thus, the number of blocks needed for the index file is $b_i = \lceil (r_i/bfr_i) \rceil = \lceil (4000/53) \rceil = 76$ blocks. To perform a binary search on the index file would need $\lceil (\log_2 b_i) \rceil = \lceil (\log_2 76) \rceil = 7$ block accesses. To search for the actual data record using the index, one additional block access is needed. In total, we need 7 + 1 = 8 block accesses, which is an improvement over binary search on the data file, which required 12 block accesses.

A major problem with a primary index is insertion and deletion of records. We have seen similar problems with sorted file organisations. However, they are more serious with the index structure because, if we attempt to insert a record in its correct position in the data file, we not only have to move records to make space for the newcomer but also have to change some index entries, since moving records may change some block anchors. Deletion of records cause similar problems in the other direction.

Since a primary index file is much smaller than the data file, storage overhead is not a serious problem.

Clustering indexes

If records of a file are physically ordered on a non-key field which may not have a unique value for each record, that field is called the *clustering field*. Based on the clustering field values, a *clustering index* can be built to speed up retrieval of records that have the same value for the clustering field. (Remember the difference between a primary index and a clustering index.)

A clustering index is also a sorted file of fixed-length records 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 block pointer. There is one entry in the clustering index for *each distinct value* of the clustering field, containing the value and a pointer to the *first* block in the data file that holds *at least one record* with that value for its clustering field. Figure 3 illustrates an example of the STUDENT file (sorted by their LEVEL rather than ID#) with a clustering index.

In figure 3, there are four distinct values for LEVEL: 0, 1, 2, and 3. Thus, there are four entries in the clustering index. As can be seen from the figure, many different records have the same LEVEL number and can be stored in different blocks. Both LEVEL 2 and LEVEL 3 entries point to the 3rd block, because it stores the first record for LEVEL 2 as well as LEVEL 3 students. All other blocks following the 3rd block must contain LEVEL 3 records, because all the records are ordered by LEVEL.

Performance issues

Performance improvements can be obtained by using the index to locate a record. However, the record insertion and deletion still causes similar problems to those in primary indexes, because the data records are physically ordered.

To alleviate the problem of insertion, it is common to reserve a whole block for each distinct value of the clustering field; all records with that value are placed in the block. If more than one block is needed to store the records for a particular value, additional blocks are allocated and linked together. To link blocks, the last position in a block is reserved to hold a block pointer to the next block. If there is no following block, then the block pointer will have the null value.

Using this linked blocks structure, no records with different clustering field values can be stored in the same block. It also makes insertion and deletion more efficient than without the linked structure. More blocks will be needed to store records and some spaces may be wasted. That is the price to pay for improving insertion efficiency. Figure 4 explains the scheme.

A clustering index is another type of sparse index, because it has an entry for each distinct value of the clustering field rather than for every record in the file.

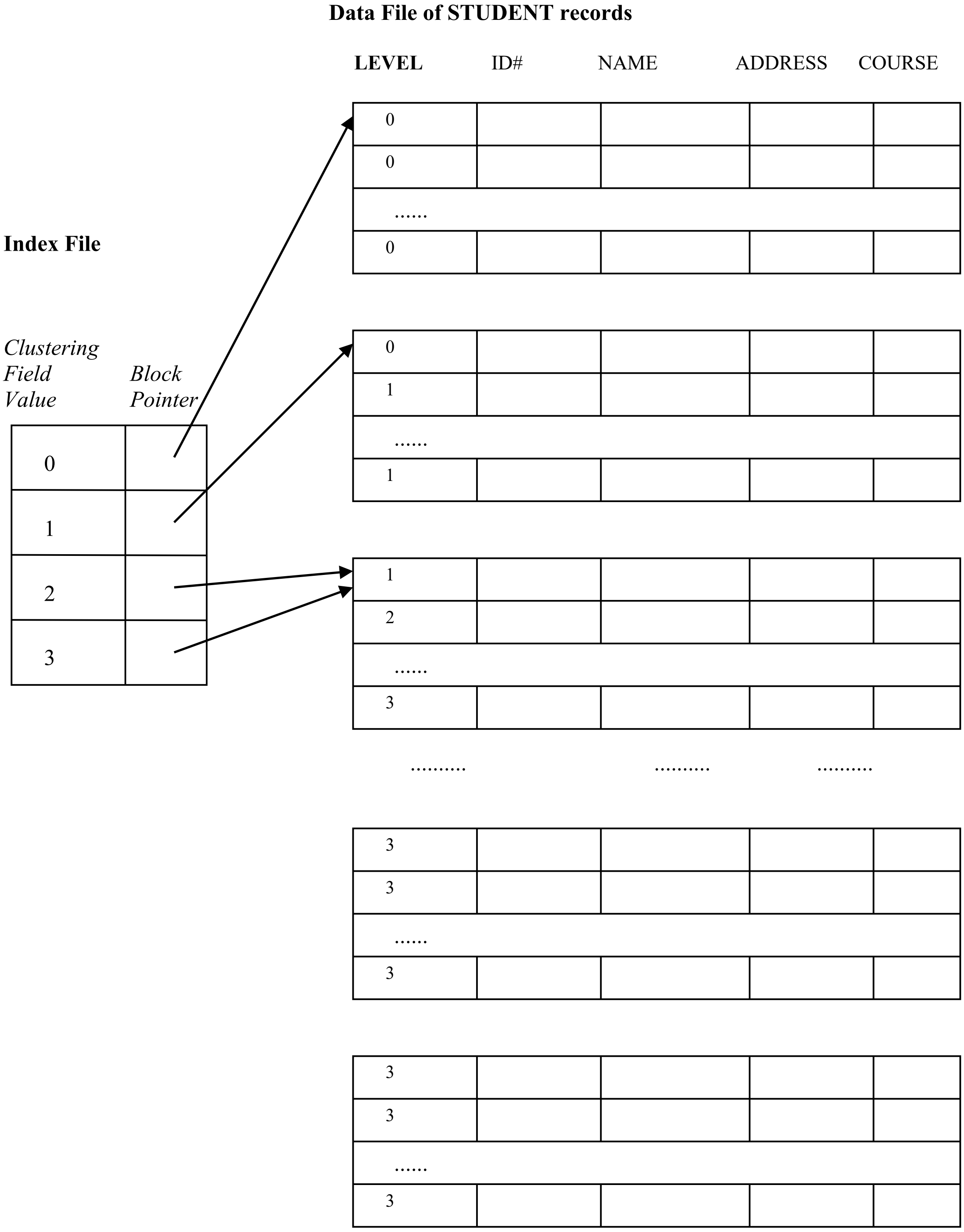


Figure 3 A clustering index on the STUDENT file sorted by LEVEL

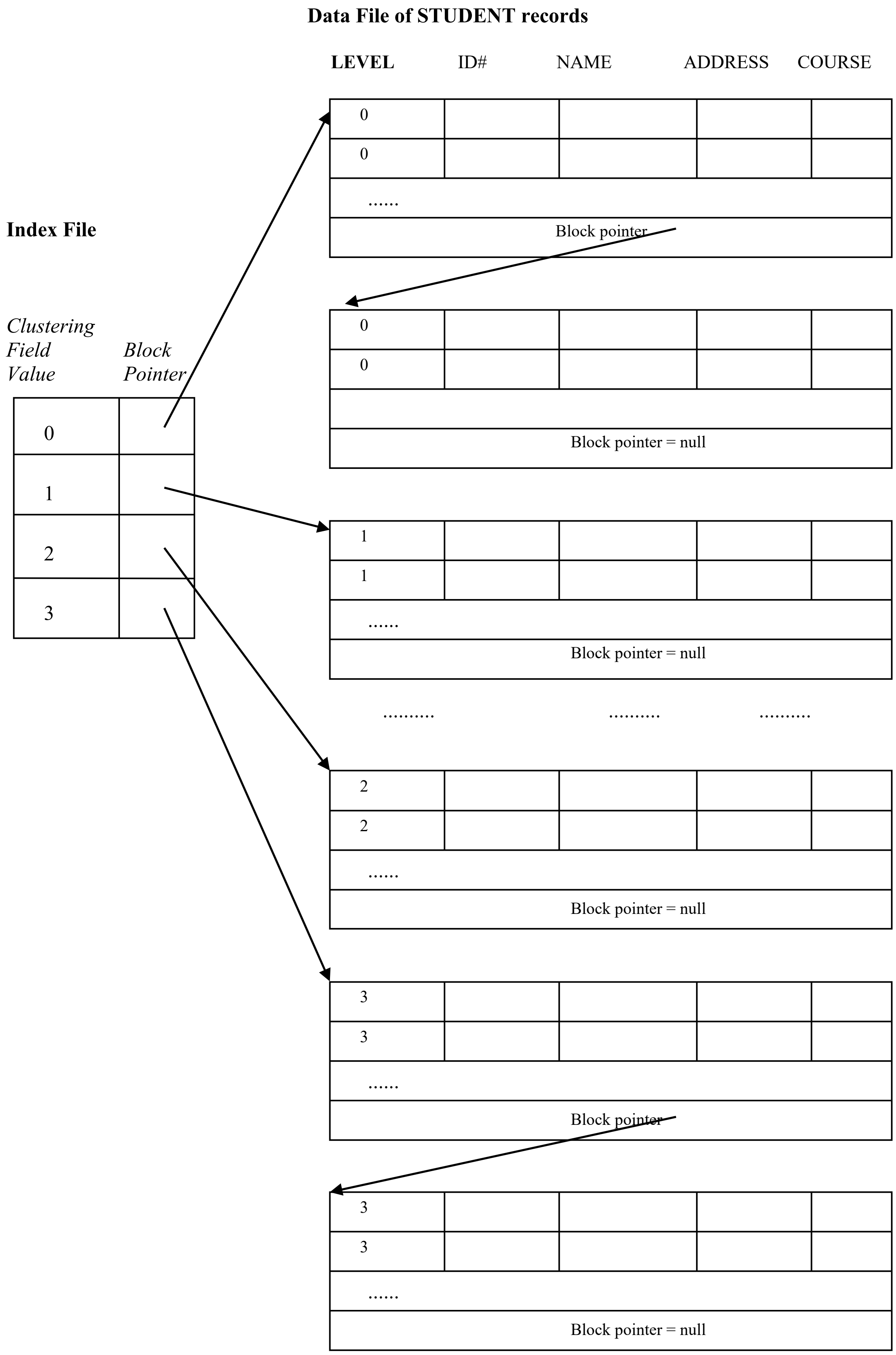

Data File of STUDENT records
LEVEL
ID#
NAME
ADDRESS
COURSE
0
0
……
Block pointer
0
0
Block pointer = null
1
1
……
Block pointer = null
……….
……….
……….
2
2
……
Block pointer = null
3
3
……
Block pointer
3
3
……
Block pointer = null
Index File
Clustering Field Value
Block Pointer
0
1
2
3

Figure 4 A clustering index with the linked blocks structure (separate blocks for each group of records that share the same clustering field value)

Secondary indexes

A *secondary index* is a sorted file of records (either fixed-length or variable-length) with two fields. The first field is of the same data type as an *indexing field* (i.e., a *non-ordering field* on which the index is built). The second field is either a *block pointer* or a *record pointer*. A file may have more than one secondary index.

In this section, we consider two cases of secondary indexes:

- ❑ The index access structure is constructed on a key field.
- ❑ The index access structure is constructed on a non-key field.

Index on a key field

Before we proceed, it must be emphasised that a key field is not necessarily an ordering field. In the case of a clustering index, the index is built on a non-key ordering field.

When the key field is not the ordering field, a secondary index can be constructed on it where the key field can also be called a *secondary key* (in contrast to a *primary key* where the key field is used to build a primary index). In such a secondary index, there is one index entry for each record in the data file, because the key field (i.e., the indexing field) has a distinct value for every record. Each entry contains the value of the secondary key for the record and a pointer either to the block in which the record is stored or to the record itself (a pointer to an individual record consists of the block address and the record's position in that block).

A secondary index on a key field is a dense index, because it includes one entry for every record in the data file.

Performance issues

We again use notation <K(i), P(i)> to represent an index entry i. All index entries are ordered by value of K(i), and therefore, a binary search can be performed on the index.

Because the data records are *not* physically ordered by values of the secondary key field, we cannot use block anchors as in primary indexes. That is why an index entry is created for each record in the data file, rather than for each block. P(i) is still a block pointer to the block containing the record with the key field value K(i). Once the appropriate block is transferred to main memory, a further search for the desired record within that block can be carried out.

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, much greater improvement in search time for an arbitrary record can be obtained by using the secondary index, because 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 data file, even if the index did not exist. Example 2 explains the improvement in terms of the number of blocks accessed when a secondary index is used to locate a record.

Example 2:

Consider the file in Example 1 with r = 40,000 records stored on a disk with block size B = 1024 bytes. File records are of fixed-length and are unspanned, with a record size R = 100 bytes. As calculated previously, this file has b = 4000 blocks. To do a linear search on the file, we would require b/2 = 4000/2 = 2000 block accesses on average to locate a record.

Now suppose that we construct a secondary index on a non-ordering key field of the file that is V = 11 bytes long. As in Example 1, a block pointer is P = 8 bytes long. Thus, the size of an index entry is $R_i = (11 + 8) = 19$ bytes, and the blocking factor for the index file is $bfr_i = \lfloor(B/R_i)\rfloor = \lfloor(1024/19)\rfloor = 53$ entries per block. In a dense secondary index like this, the total number of index entries r_i is equal to the number of records in the data file, which is 40000. The number of blocks needed for the index is hence $b_i = \lceil(r_i/bfr_i)\rceil = \lceil(40000/53)\rceil = 755$ blocks. Compare this to the 76 blocks needed by the sparse primary index in Example 1.

To perform a binary search on the index file would need $\lceil(\log_2 b_i)\rceil = \lceil(\log_2 755)\rceil = 10$ block accesses. To search for the actual data record using the index, one additional block access is needed. In total, we need 10 + 1 = 11 block accesses, which is a huge improvement over the 2000 block accesses needed on average for a linear search on the data file.

Index on a non-key field

Using the same principles, we can also build a secondary index on a non-key field of a file. In this case, many data records can have the same value for the indexing field. There are several options for implementing such an index:

Option 1:

We can create several entries in the index file with the same K(i) value – one for each record sharing the same K(i) value. The other field P(i) may have different block addresses, depending on where those records are stored. Such an index would be a dense index.

Option 2:

Alternatively, we can use variable-length records for the index entries, with a repeating field for the pointer. We maintain a list of pointers in the index entry for K(i) – one pointer to each block that contains a record whose indexing field value equals K(i). In other words, an index entry will look like this: <K(i), [P(i, 1), P(i, 2), P(i, 3), ...]>. In either option 1 or option 2, the binary search algorithm on the index must be modified appropriately.

Option 3:

This is the most commonly adopted approach. In this option, we keep the index entries themselves at a fixed-length and have a single entry for each indexing field value. Additionally, an extra level of indirection is created to handle the multiple pointers. Such an index is a sparse scheme, and the pointer P(i) in index entry <K(i), P(i)> points to a block of record pointers (this is the extra level); each record pointer in that block points to one of the data file blocks containing the record 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 block, a linked list of blocks is used. Figure 5 explains this option.

From figure 5, it can be seen that instead of using index entries like <K(i), [P(i, 1), P(i, 2), P(i, 3), ...]> and <K(j), [P(j, 1), P(j, 2), P(j, 3), ...]> as in option 2, an extra level of data structure is used to store the record pointers. Effectively, the repeating field in the index entries of option 2 is removed, which makes option 3 more appealing.

It should be noted that a secondary index provides a *logical ordering* on the data records by the indexing field. If we access the records in order of the entries in the secondary index, the records can be retrieved in order of the indexing field values.

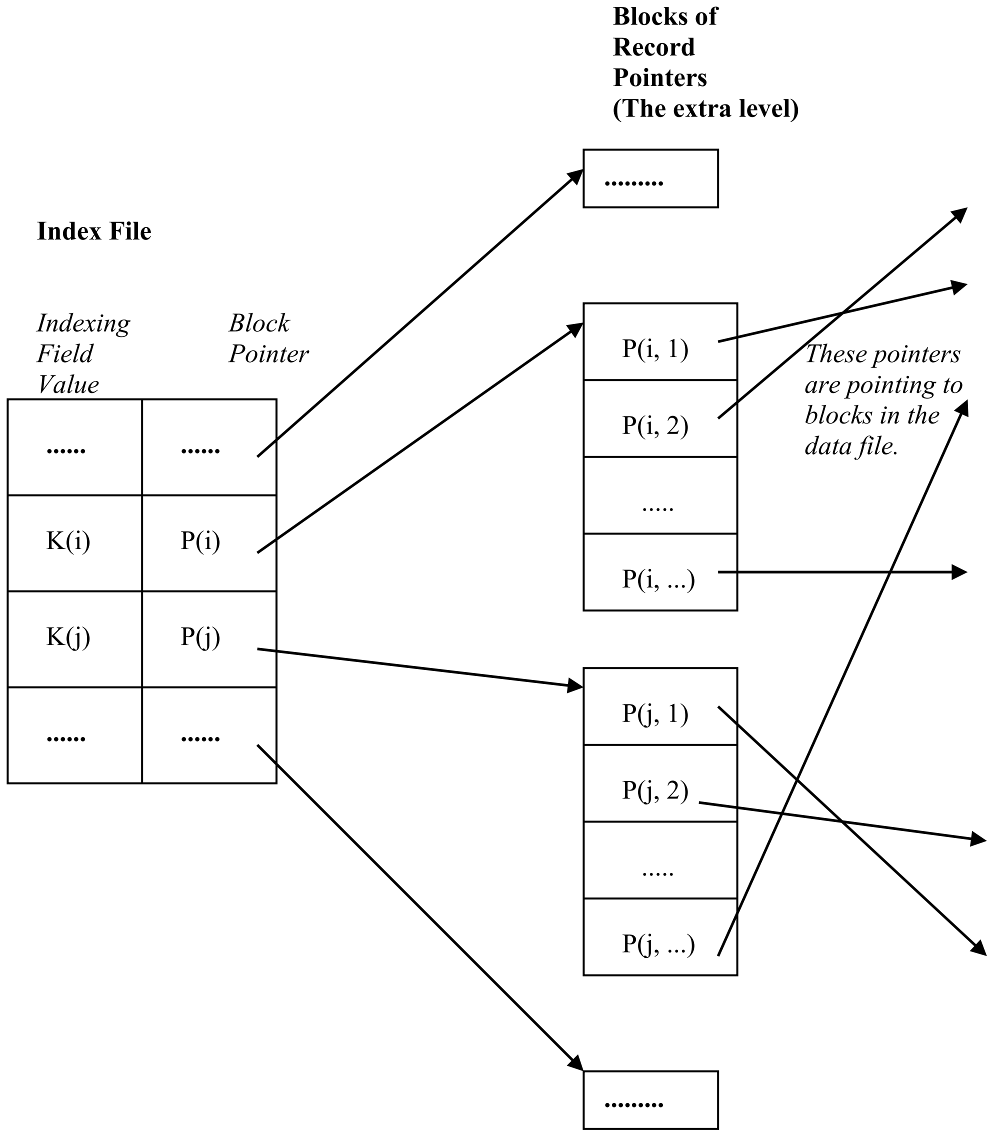


Figure 5 A secondary index on a non-key field implemented using one level of indirection so that index entries are of fixed-length and have distinct indexing field values

Summary of single-level ordered indexes

The following table summarises the properties of each type of indexes by comparing the number of index entries and specifying which indexes are dense or sparse and which use block anchors of the data file.

	Properties of Indexes		
Type of Index	**Number of Index Entries**	**Dense or Sparse**	**Using Block Anchor**
Primary	Equal to the number of blocks in the data file	Sparse	Yes
Clustering	Equal to the number of distinct indexing field values	Sparse	Yes if separate blocks are used for records with different indexing field values; No otherwise.
Secondary on a key field	Equal to the number of records in the data file	Dense	No
Secondary on a non-key field	Equal to the number of records for option 1; Equal to the number of distinct indexing field values for options 2 and 3	Dense for option 1; Sparse for options 2 and 3	No

Multilevel Indexes

The principle

The index structures that we have studied so far involve a sorted index file. A binary search is applied to the index to locate pointers to a block containing a record (or records) in the file with a specified indexing field value. A binary search requires $\lceil \log_2 b_i) \rceil$ block accesses for an index file with b_i 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 the log function to the base 2 is used.

The idea behind a *multilevel index* is to reduce the part of the index that we have to continue to search by bfr_i, the blocking factor for the index, which is almost always larger than 2. Thus, the search space can be reduced much faster. The value of bfr_i is also called the *fan-out* of the multilevel index, and we will refer it by the notation *fo*. By analogy to the binary search, searching a multilevel index requires $\lceil \log_{fo} b_i) \rceil$ block accesses, which is a smaller number than for binary search if the fan-out is bigger than 2.

The structure

A multilevel index considers the index file, which was discussed in the previous sections as single-level ordered indexes and will now be referred to as the *first* (or *base*) level of the multilevel structure, as a *sorted file* with a *distinct* value for each K(i). Remember we mentioned earlier that an index file is effectively a special type of data file with two fields. Thus, we can build a primary index for an index file itself (i.e., on top of the index at the first level). This new 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 index entries. The blocking factor bfr_i for the second level – and for all subsequent levels – is the same as that for the first-level index, because all index entries are of the same size; each has one field value and one block address. If the first level has r_1 entries, and the blocking factor – which is also the fan-out – for the index is $bfr_i = fo$, then the first level needs $\lceil (r_1/fo) \rceil$ blocks, which is therefore the number of entries r_2 needed at the second level of the index.

The above process can be repeated and a *third-level index* can be created on top of the second-level one. The third level, which is a primary index for the second level, has an entry for each second-level block. Thus, the number of third-level entries is $r_3 = \lceil (r_2/fo) \rceil$. (**Important note**: 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 continue the index-building process until all the entries of some index level *d* fit in a single block. This block at the d^{th} level is called the *top index level* (the first level is at the bottom and we work our way up). Each level reduces the number of entries at the previous level by a factor of fo – the multilevel index fan-out – so we can use the formula $1 \leq (r_1/((fo)^d))$ to calculate d. Hence, a multilevel index with r_1 first-level entries will need d levels, where $d = \lceil (\log_{fo}(r_1)) \rceil$.

Important: The multilevel structure can be used on *any type* of index, whether it is a primary, a clustering, or a secondary index, as long as the first-level index *has distinct values for K(i) and fixed-length entries*. Figure 6 depicts a multilevel index built on top of a primary index.

It can be seen from figure 6 that the data file is a sorted file on the key field. There is a primary index built on the data file. Because it has four blocks, a second-level index is created which fits in a single block. Thus, the second level is also the top level.

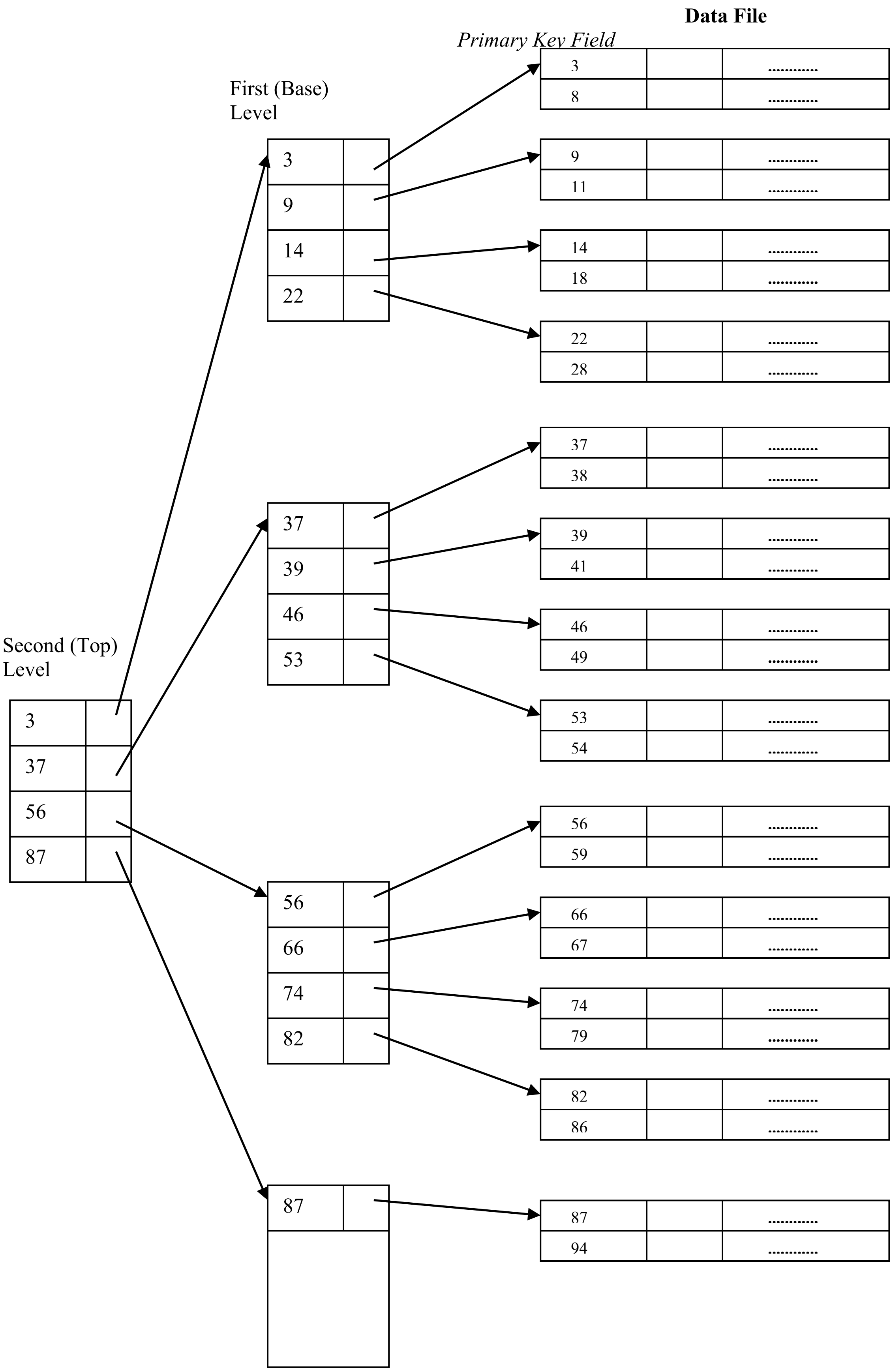

Data File
Primary Key Field
First (Base) Level
Second (Top) Level
3
37
56
87
3
9
14
22
37
39
46
53
56
66
74
82
87
3
8
9
11
14
18
22
28
37
38
39
41
46
49
53
54
56
59
66
67
74
79
82
86
87
94
....

Figure 6 A two-level primary index

Performance issues

Multilevel indexes are used to improve the performance in terms of the number of block accesses needed when searching for a record based on an indexing field value. The following example explains the process.

Example 3:

> Suppose that the dense secondary index of Example 2 is converted into a multilevel index. In Example 2, we have calculated that the index blocking factor bfr_i = 53 entries per block, which is also the fan-out for the multilevel index. We also knew that the number of first-level blocks b_1 = 755. Hence, the number of second-level blocks will be $b_2 = \lceil (b_1/fo) \rceil = \lceil (755/53) \rceil$ = 15 blocks, and the number of third-level blocks will be $b_3 = \lceil (b_2/fo) \rceil = \lceil (15/53) \rceil$ = 1 block. Now at the third level, because all index entries can be stored in a single block, it is also the top level and d = 3 (remember $d = \lceil (\log_{fo}(r_1)) \rceil = \lceil (\log_{53}(40000)) \rceil = 3$, where $r_1 = r = 40000$).
>
> To search for a record based on a non-ordering key value using the multilevel index, we must access one block at each level plus one block from the data file. Thus, we need d + 1 = 3 + 1 = 4 block accesses. Compare this to Example 2, where 11 block accesses were needed when a single-level index and binary search were used.

It should note that we could also have a multilevel primary index which could be sparse. In this case, we must access the data block from the file before we can determine whether the record being searched for is in the file. For a dense index, this can be determined by accessing the first-level index without having to access the data block, since there is an index entry for every record in the file.

As seen earlier, a multilevel index improves the performance of searching for a record based on a specified indexing field value. However, the problems with insertions and deletions are still there, because all index levels are physically ordered files. To retain the benefits of using multilevel indexing while reducing index insertion and deletion problems, database developers often adopt a multilevel structure that leaves some space in each of its blocks for inserting new entries. This is called *a dynamic multilevel index* and is often implemented by using data structures called *B-trees* and *B^+-trees*.

(there is a further handout giving details of tree file structures used to store indexes, and some detail of the algorithms used to traverse them. The information in that handout will not be examined, but is available on request for anyone who wishes to look further into the file structures frequently used for indexes).

Using Indexes in Practice

Indexes can be used to gain faster access to data in databases. Thus, all commercial database management systems (DBMS), such as Oracle, Postgres, Sybase etc., incorporate such access structures. In these DBMSs, however, the presence of an index is transparent to the ordinary users. Users will not have any control over what index structures should be used and what algorithms should be applied. These have all been determined by the DBMSs.

In Oracle, for example, indexes are built by using the B*-tree structure. Algorithms necessary for maintaining such indexes have all been implemented and tuned in Oracle to gain the maximum efficiency. After an index has been created for a table (i.e., a file of records), Oracle automatically maintains that index. Insertions, modifications, and deletions of rows in the table (i.e., records in a file) automatically update the related indexes.

Although, in practice, users do not have much control over the physical aspects of indexes, they can specify whether or not they want to create an index on a table and if so, what attribute or attributes should be the indexing field(s).

In Oracle, a table can have any number of indexes (at most one primary or clustering index, plus any number of secondary indexes). However, it is important to remember that more indexes there are, the more overhead is incurred during table modifications, insertions and deletions.

In order to study the practical use of indexes, we will use Oracle as an example throughout the rest of this document.

Types of indexes in Oracle

So far in this document, the indexes we have studied are all based on a single indexing field. In Oracle, another type of index is also allowed. That is the so-called *composite index*, which builds an access path based on several columns (fields) in a table. In other words, there is an index entry for a distinct combination of values in those columns. When creating such an index, the order of the columns in the definition is important, because the index is accessed based on that order.

Deciding what to index

An index should always be created with some knowledge of the database application and data-access patterns. If an index is created on a column that is not used to access data, the index is useless. In short, indexes should be created based on the values accessed in the application; the application should be developed to take advantages of these indexes. Having knowledge of and influence over these indexes can be very useful to the application developer.

What to index

An index is effective only when it is used. In order to minimise the overhead in maintaining indexes, we need to be selective.

What tables need indexes?

- ❑ Create indexes on tables when most queries select only a small number of rows from those tables. Queries that select a large number of rows defeat the purpose of indexing. Use indexes when queries access less than 5% of the rows in the table.
- ❑ Do not build indexes on tables that are frequently updated.
- ❑ Create indexes on tables that do not have duplicate values on the columns usually selected in WHERE clause (i.e., a large number of distinct values in the indexing field will help).

- ❑ Tables in which the selection is based on TRUE or FALSE values are not good candidates for indexing.
- ❑ Build indexes on tables that are queried with relatively simple WHERE clause. Complex WHERE clauses may not take advantages of indexes.

Which column should be used as the indexing field?

- ❑ Choose columns that are most frequently specified in WHERE clauses.
- ❑ Do not use columns that do not have many distinct values. Columns in which a large percentage of rows are duplicates cannot take advantages of indexing.
- ❑ Columns that have unique values are good candidates for indexing. *Oracle automatically indexes columns that are unique or are primary keys*.
- ❑ Columns that are commonly used to join tables are good candidates for indexing.
- ❑ Frequently updated columns should not be indexed.

When to use composite indexes?

- ❑ When two or more columns do not have unique values individually, but their value combinations are unique, a composite index using these columns is very effective.
- ❑ If a table has a composite key, a corresponding composite index could be created.
- ❑ If all values of a SELECT statement are in a composite index, Oracle does not query the actual table; the result is returned from the index.

Taking advantage of indexes

Once indexes have been created, it is necessary to include the indexed column or columns in the WHERE clause of the SQL statements. When the indexed column or columns are referenced in the WHERE clause, the Oracle *query optimiser* will immediately know to use an appropriate index to perform the search rather than carry out a complete table scan to access the requested data. If the indexed columns are not included, the index structure will be bypassed and it will cause a time-consuming table scan.

It takes the right application coding to take advantages of indexes. Co-ordination between the DBA (Database Administrator) and the application developers is required to achieve the objective.

Creating indexes in Oracle

In Oracle, indexes can be created either via the graphical utilities provided by the Oracle Enterprise Manager (Oracle Navigator if using Personal Edition of Oracle), or via the SQL CREATE INDEX command.

Creating indexes using SQL

Indexes can be created using standard SQL commands. To create an index on the LASTNAME field in an employee table, we execute the following SQL command in the Oracle SQL *Plus environment:

```
CREATE INDEX I_EMPLOYEE$EMPLOYEE_LAST_NAME
ON EMPLOYEE(LAST_NAME);
```

To remove it, we execute:

```
DROP INDEX IEMPLOYEE$EMPLOYEE_LAST_NAME;
```

Concatinated index

```
The following index makes sure that only unique supplier/part number
combinations may be entered in the SHIPMENTS table. This is called
CONCATENATED index.

CREATE UNIQUE INDEX ORDER
ON SHIPMENTS (S Num, P Num);
```