Primary Index

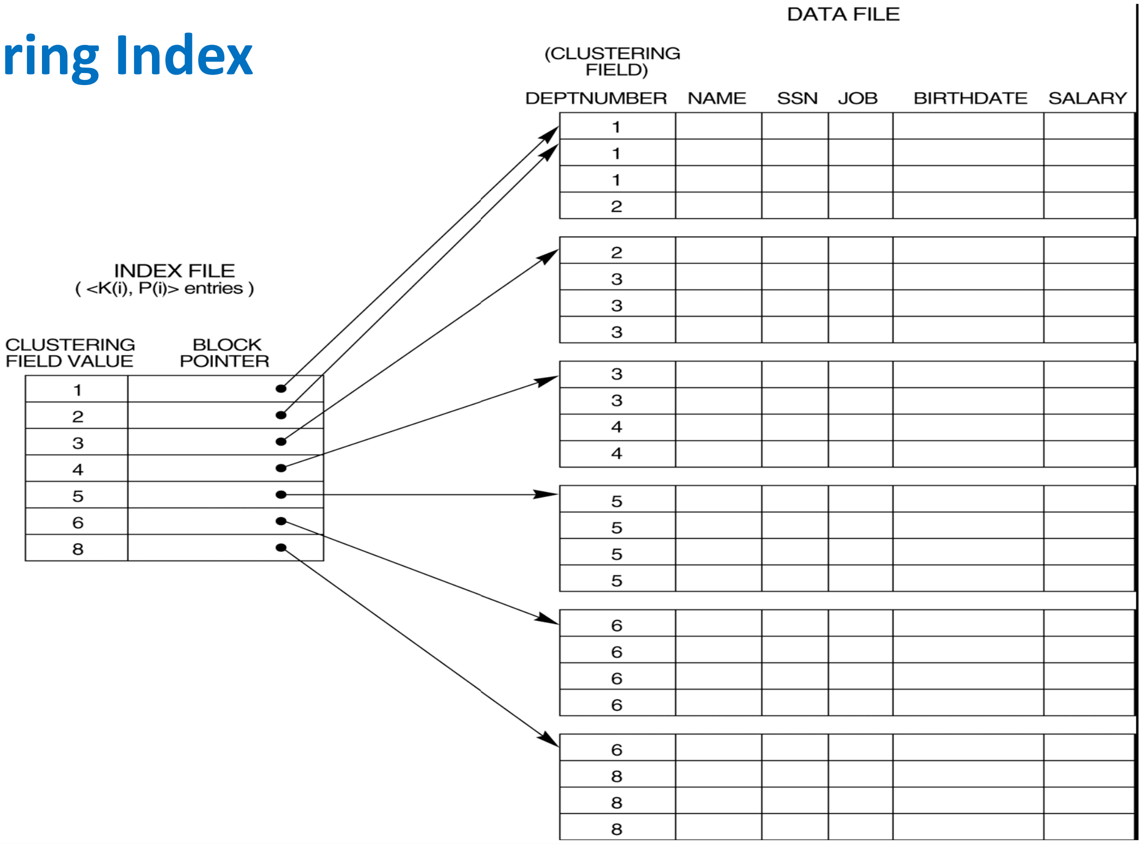
Problems:

- Inserting and deleting records in the main file must move other records, since it's ordered.
- Some insertions/deletions must also change index entries, if the anchor records change.
- There can be only one primary index on a file.

Clustering Index

- Defined on an ordered data file.
- The data file is ordered on a *non-key field* unlike primary index, which requires that the ordering field of the data file have a distinct value for each record.
- Includes one index entry *for each distinct value* of the field; the index entry points to the first data block that contains records with that field value.
- It is another example of *non-dense (sparse)* index because it has an entry for every *distinct value* of the indexing field which is a non-key by definition.

Clustering Index



Clustering Index With a Separate Block/Cluster

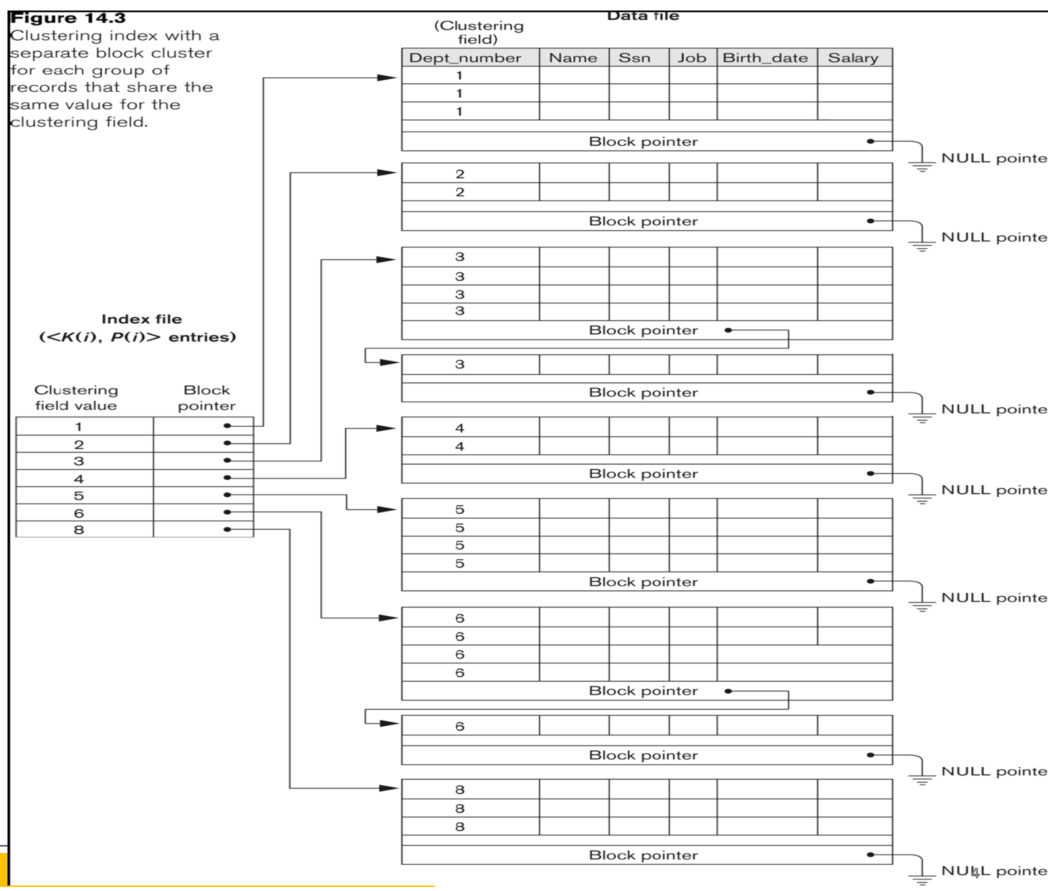


Figure 14.3
Clustering index with a separate block cluster for each group of records that share the same value for the clustering field.

Secondary Index

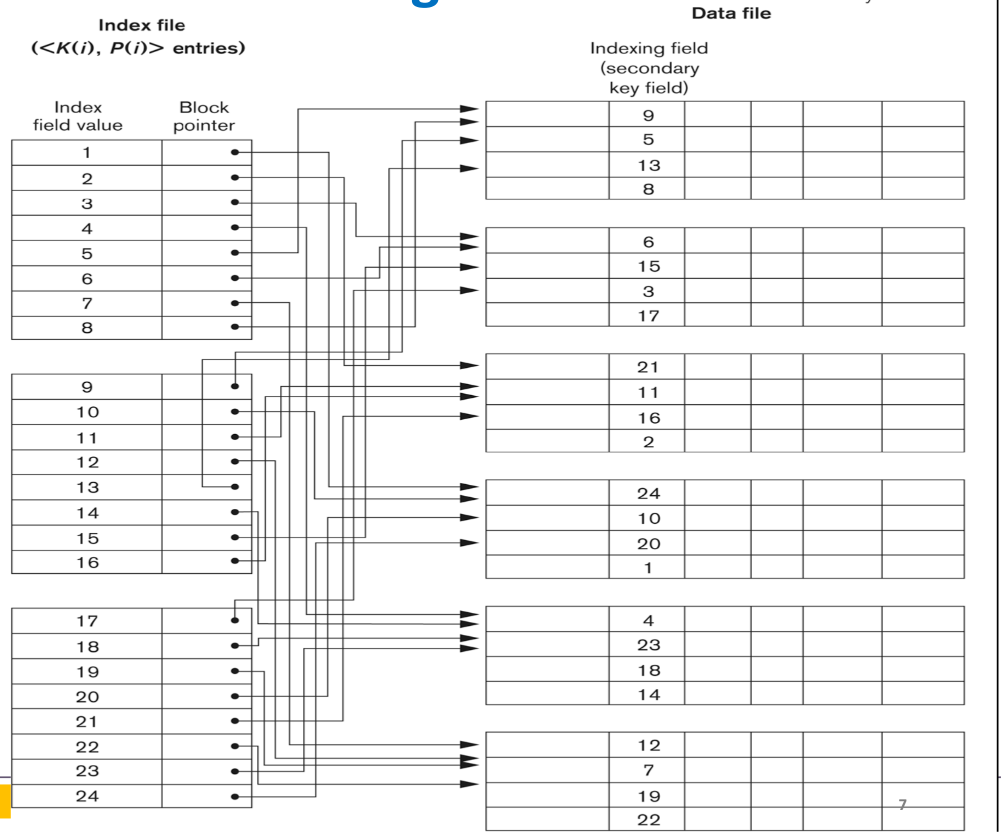
- A secondary index provides a secondary means of accessing a file for which some primary access already exists.
- The secondary index may be on a field which is a candidate key and has a unique value in every record, or a non-key with duplicate values.
- The index is an ordered file with two fields.
- The first field is of the same data type as some **non-ordering field** of the data file that is an indexing field.
- The second field is either a **block** pointer or a **record** pointer.

Secondary Index

- There can be *many* secondary indexes (and hence, indexing fields) for the same file.
- A secondary index access structure on a key field that has a distinct value for every record. Such a field is sometimes called a secondary key.
- In this case there is one index entry for each record in the data file, which 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. Hence, such an index is dense.

Dense Secondary Index on Non-ordering Key Field

Figure 14.4
A dense secondary index (with block pointers) on a nonordering key field of a file.



Secondary Index

- Consider a file with r = 30,000 fixed-length records of size R = 100 bytes stored on a disk with block size B = 1024 bytes.

  Number of block accesses without an index structure
- Bfr => L(1024/100)J = 10 records per block.
- The file has b = 3000 blocks (i.e. [(30,000/10)]). To do a linear search on the file, we would require b/2 = 3000/2 = 1500 block accesses on the average.

Secondary Index

- Construct a secondary index on a non-ordering key field of the file that is V = 9 bytes long. Block pointer is P = 6 bytes long. Number of block accesses with an index structure?
- Each index entry is R, = (9 + 6) = 15 bytes, and the blocking factor for the index is $bfr_i = \lfloor(B/R_i)\rfloor = \lfloor(1024/15)\rfloor = 68$ entries per block.
- In a dense secondary index such as this, the total number of index entries r_i is equal to the number of records in the data file, which is 30,000. The number of blocks needed for the index is hence

$$b_i = \lceil(r/bfr_i)\rceil = \lceil(30000/68)\rceil = 442 \text{ blocks.}$$

Secondary Index

- A binary search on this secondary index needs $\lceil\log_2 b\rceil = \lceil\log_2 442\rceil$ = 9 blocks accesses.
- To search for a record using the index, we need an additional block access to the data file for a total of 9 + 1 = 10 block accesses.
- A vast improvement over the 1500 block accesses needed on the average for a linear search, but slightly worse than the 07 block accesses required for the primary index.

Secondary Index

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

i) Include several index entries with the same K(i) value-one for each record. The data pointers now must be record pointers, not block pointers. Simple but costly. This would be a dense index.

ii) The index contains only one entry for each index field value. However, the entry is variable-length and has a list of however many record pointers it needs to enumerate all the data records.

iii) The first index has one entry for each index field value, and a *block* pointer to a block in the second index. That block has a list of just *record* pointers, one for each data record with that index field value.

Secondary Index Option III

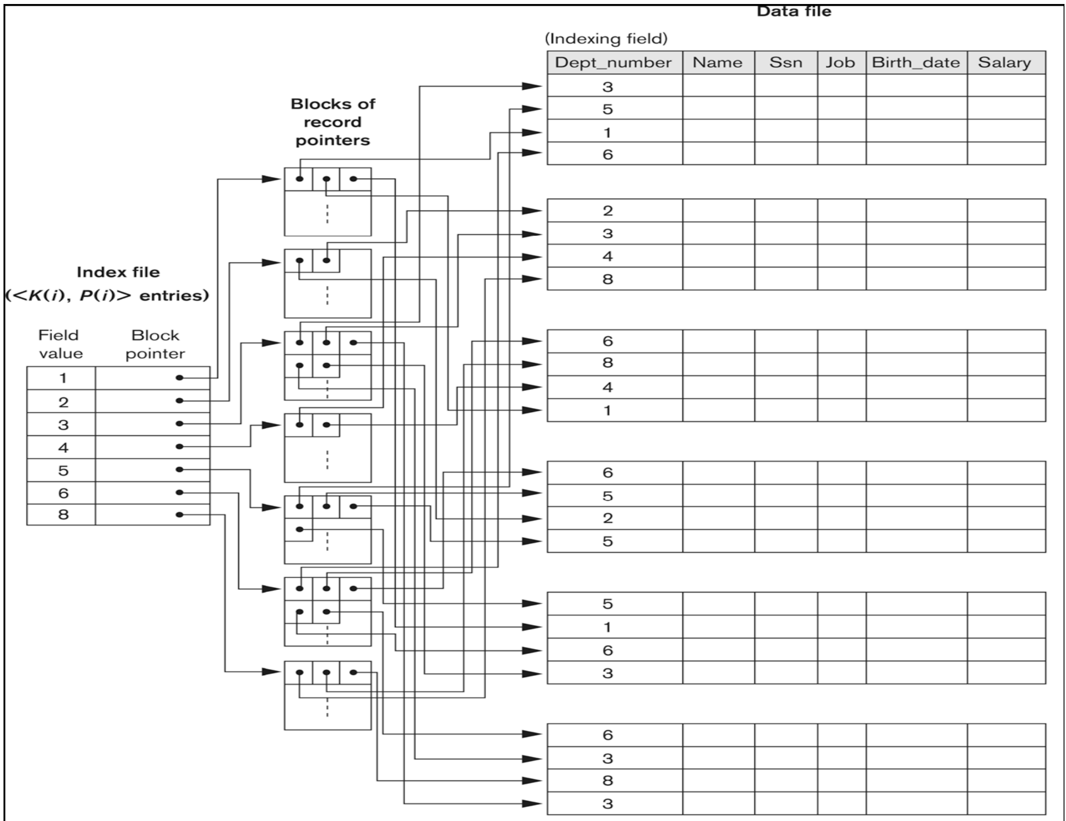


Figure 14.5
A secondary index (with record pointers) on a nonkey field implemented using one level of indirection so that index entries are of fixed length and have unique field values.

TABLE 14.2 PROPERTIES OF INDEX TYPES

TYPE OF INDEX	NUMBER OF (FIRST-LEVEL) INDEX ENTRIES	DENSE OR NONDENSE	BLOCK ANCHORING ON THE DATA FILE
Primary	Number of blocks in data file	Nondense	Yes
Clustering	Number of distinct index field values	Nondense	Yes/no[a]
Secondary (key)	Number of records in data file	Dense	No
Secondary (nonkey)	Number of records[b] or Number of distinct index field values[c]	Dense or Nondense	No

[a]Yes if every distinct value of the ordering field starts a new block; no otherwise.

[b]For option 1.

[c]For options 2 and 3.

Hash File Organization/Hash Index

- A bucket in hash file organization contains one or more records and is typically a disk block.
- In a hash file organization it is possible to find the bucket of a record directly from its search-key value using a hash function.
- Records with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate a record.

Example of Hash File Organization

Example: Hash file organization of Employee file, using dept_name as key

- The hash file has 10 buckets,
- Assume that based upon a selected hash function the following hash values are returned E.g. h(Music) = 1 h(Physics) = 3 h(History) = 2 h(Elec. Eng.) = 3

Ref: ©Silberschatz, Korth and Sudarshan

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000

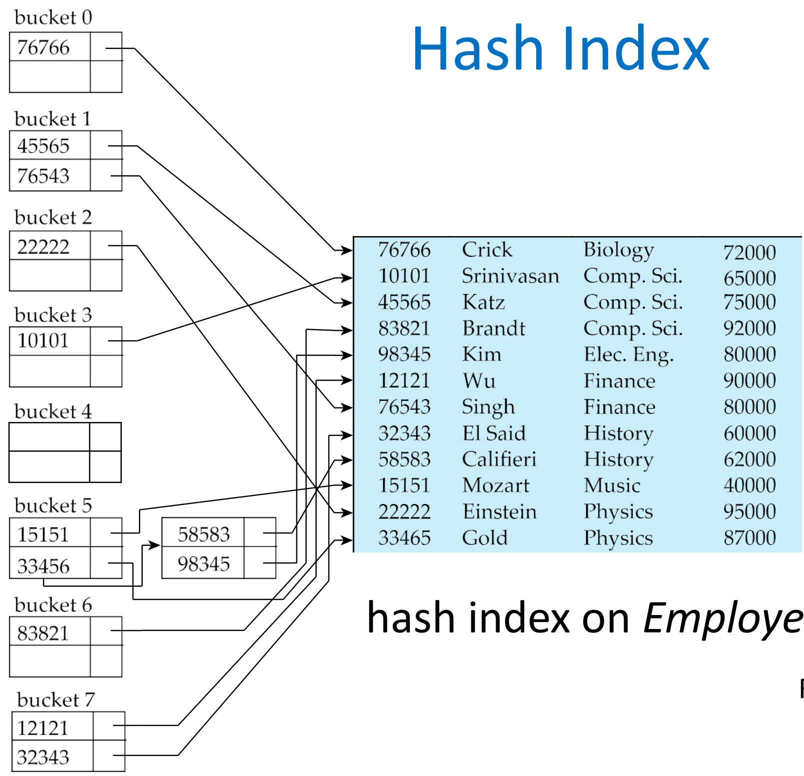
bucket 7

Hash file organization of *Employee* file, using *dept_name* as key

Hash Index

- In addition to file organization hashing can also be used for index-structure creation.
- A **hash index** is arranged based on the search keys, with their associated record pointers into the file structure.
- Hash indices are secondary indices and if the file itself is organized using hashing, a separate primary hash index on it with the same search-key is unnecessary.
- However, the term hash index is used to refer to both secondary index structures and hash organized files.

Hash Index



hash index on *Employee,* on attribute *Empid*

Ref: ©Silberschatz, Korth and Sudarshan

Multi-Level Indexes

- Since a single-level index is an ordered file, we can create a primary index *to the index itself*;
- In this case, the original index file is called the *first-level index* and the index to the index is called the *second-level index*.
- We can repeat the process, creating a third, fourth, ..., top level until all entries of the *top-level* fit in one disk block.
- A multi-level index can be created for any type of first level index (primary, secondary, clustering) as long as the first-level index consists of *more than one* disk block.

Multi-Level Indexes

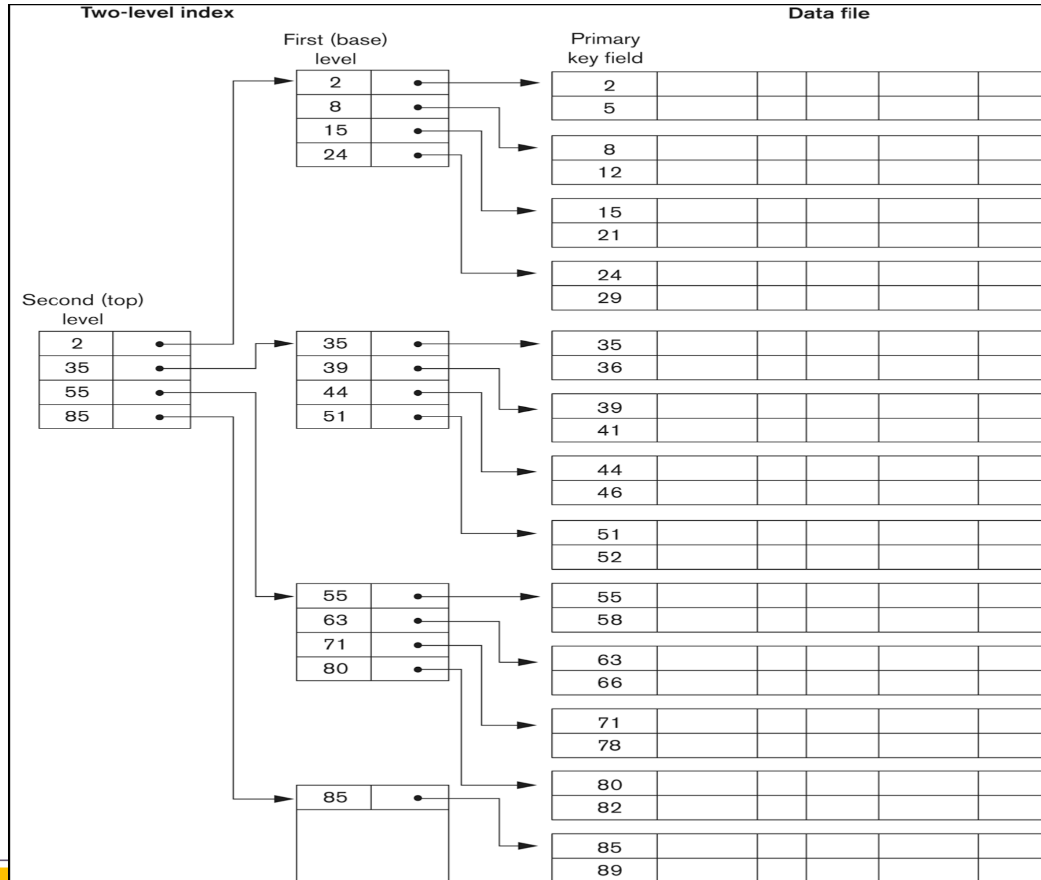


Figure 14.6
A two-level primary index resembling ISAM (Index Sequential Access Method) organization.

Multi-Level Indexes

- Such a multi-level index is a form of *search tree.*
- However, insertion and deletion of new index entries is a severe problem because every level of the index is an *ordered file*.
- Since the first-level index is ordered and key on the index value, the remaining levels can all be non-dense and block-pointing.

Multi-Level Indexes

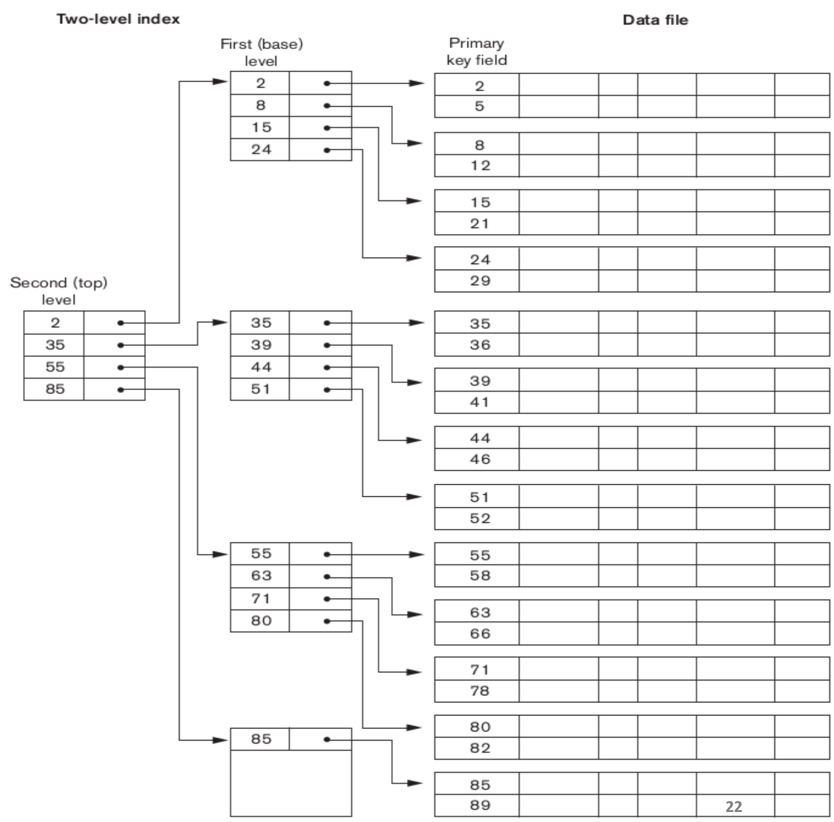
- The blocking factor of the index entries, bfr_i, is called the *fan-out* f_o.
- If the first level has r_1 index entries, it needs $\lceil r_1/f_o \rceil$ blocks, which is also r_2, the number of index entries in the next level up. That second level needs $\lceil r_2/f_o \rceil$ blocks, which is also r_3 and so on.
- Eventually at some level $r_t = 1$, and t is the top level where $t = \lceil \log_{fo}(r_1) \rceil$.
- Hence, t disk blocks are accessed for an index search and t is the number of index levels.

Multi-Level Indexes

A common file organization used in business data processing is an ordered file with a multilevel primary index on its ordering key field. Such an organization is called an indexed sequential file and was used in a large number of early IBM systems.

Insertion is handled by some form of overflow file that is merged periodically with the data file. The index is re-created during file reorganization.

Figure 18.6
A two-level primary index resembling ISAM (Indexed Sequential Access Method) organization.



Multi-Level Indexes

- Suppose that the example given in dense secondary index of is converted into a multilevel index.
  Calculated index blocking factor bfr, = 68 index entries per block, which is also f_o for the multilevel index;
- The number of first-level blocks b1 = 442 blocks.
- The number of second-level blocks will be b2 = $\lceil b_1/f_o \rceil = \lceil (442/68) \rceil$ = 7 blocks, and
- The number of third-level blocks will be b3 = $\lceil b_2/f_o \rceil = \lceil (7/68) \rceil$ = 1 block.

Hence, the third level is the top level of the index, and t =3.
To access a record by searching the multilevel index, we must access one block at each level plus one block from the data file, so we need t +1 = 3 + 1= 4 block accesses.

Dynamic Multilevel Indexes Using BTrees and B+-Trees

- Most multi-level indexes use B-tree or B+-tree data structures because of the insertion and deletion problem.
- This leaves space in each tree node (disk block) to allow for new index entries.
- These data structures are variations of search trees that allow efficient insertion and deletion of new search values.
- In B-Tree and B+-Tree data structures, each node corresponds to a disk block.
- Each node is kept between half-full and completely full.

Dynamic Multilevel Indexes Using BTrees and B+-Trees

- An insertion into a node that is not full is quite efficient.
- If a node is full the insertion causes a split into two nodes.
- Splitting may propagate to other tree levels.

Dynamic Multilevel Indexes Using BTrees and B+-Trees

- A deletion is quite efficient if a node does not become less than half full.
- If a deletion causes a node to become less than half full, it must be merged with neighboring nodes.

B - tree

A B-tree of order *p* can be defined as follows:

- Each internal node is of the form

  $<P_1,<K_1,Pr_1>,P_2,<K_2,Pr_2>,\ldots..,<K_{q-1},Pr_{q-1}>,P_q>$

  where $q \leq p$. Each P_i is a tree pointer and each Pr_i is a data pointer.
- Within each node $K_1 < K_2 < \ldots.<K_{q-1}$
- Each node has at most *p* tree pointers.
- Each node with q tree pointers, $q \leq p$, has q-1 search key field values.

Figure 14.10
B-Tree structures. (a) A node in a B-tree with $q - 1$ search values. (b) A B-tree of order $p = 3$.The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.