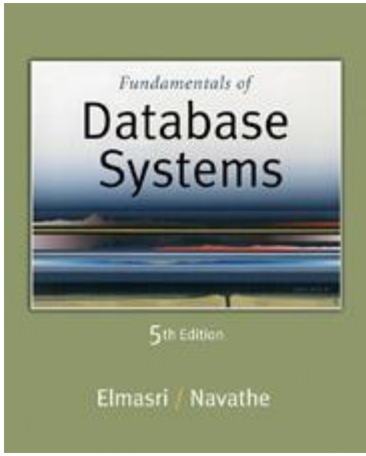
Fundamentals of Database Systems

5th Edition

Elmasri / Navathe

Chapter 14

Indexing Structures for Files



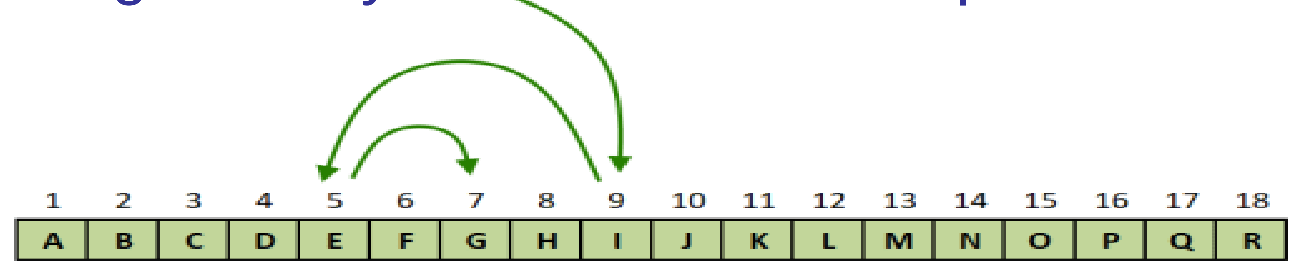
Example

- **Real Example**:
  - The idea behind an ordered index is similar to that behind the **index** used in a textbook, which **lists important terms** at the end of the book **in alphabetical order** along with a list of **page numbers** where the term appears in the book.
  - We can search the book index for a certain term in the textbook to find a list of addresses—page numbers in this case—and use these addresses to locate the specified pages first and then search for the term on each specified page.
  - The alternative, if **no other guidance is given**, would be to sift **slowly** through **the whole textbook word by word** to find the term we are interested in; this corresponds to doing a **linear search**, which **scans the whole file**.

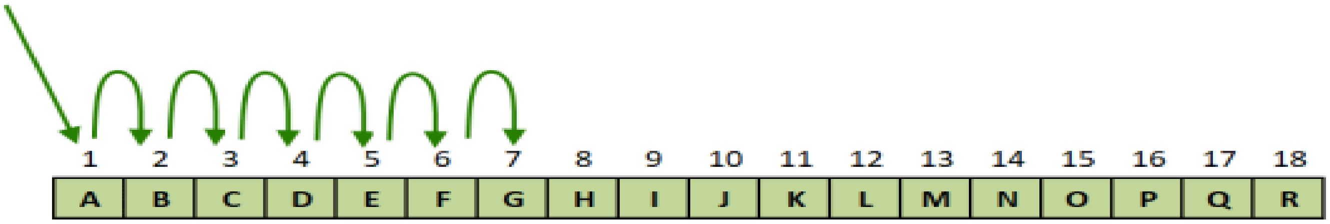
Indexes

- Indexes are additional auxiliary access structures, used to **speed up the retrieval of records** in response to certain **search conditions**.
- The index structures are **additional files** on disk that provide **secondary access paths**, which provide **alternative** ways to access the records **without affecting the physical placement of records in the primary data file on disk.**
- **Any field** of the file can be used to create an index, and multiple indexes on different fields—as well as indexes on multiple fields—can be constructed on the same file.
- One **form** of an index is a file of entries <**field value, pointer to record>**, which is **ordered by field value**

- The values in the index are **ordered** so that we can do a **binary search** on the index. If both the data file and the index file are ordered, and since **the index file is typically much smaller than the data file**, searching the index using a binary search is a better option.



Binary Search - Find 'G' in sorted list A-R



Linear Search - Find 'G' in sorted list A-R

Index Types

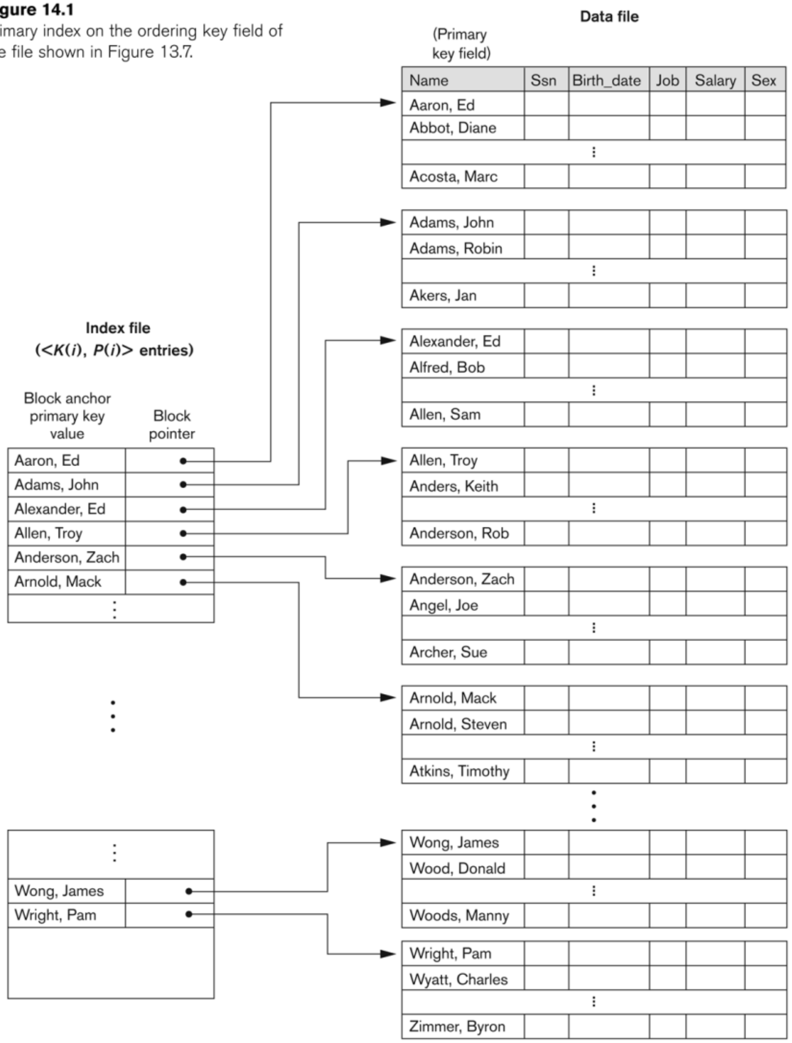
- Types of **Single-level Ordered Indexes**
  - **Primary** Indexes
  - **Clustering** Indexes
  - **Secondary** Indexes
- **Multilevel Indexes**

Primary Index

- A **primary index** is specified on the **ordering key field** of an ordered file of records.
- An ordering key field is used to **physically order the file records on disk**, and **every record has a unique value** for that field.
- A primary index is an **ordered** file whose records are of **fixed length with two fields**:
  - The first field is of the **same data type** as the ordering key field—called the primary key—of the data file
  - The second field is **a pointer to a disk block** (a block address). There is **one index entry (or index record) in the index file for each block** in the data file.
- Each index entry has the value of the primary key field **for the first record in a block and a pointer to that block as its two field values**. We will refer to the two field values of index entry i as <K(i), P(i)>.

Primary index on the ordering key field

Figure 14.1
Primary index on the ordering key field of the file shown in Figure 13.7.



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

- The index file for a primary index **occupies a much smaller space** than does the data file, for two reasons:
  - First, there are **fewer index entries** than there are records in the data file.
  - Second, **each index entry is typically smaller in size** than a data record because it has only two fields; consequently, more index entries than data records can fit in one block.
- Therefore, a **binary search on the index file requires fewer block accesses than a binary search on the data file**.

Example illustrates **the saving in block accesses** that is attainable when a primary index is used to search for a record.

- Suppose that we have an ordered file with **r = 30,000** records stored on a disk with **block size B = 1024 bytes**.
- File records are of fixed size and are unspanned, with **record length R = 100 bytes**.
- The **blocking factor** for the file would be
  $bfr = \lfloor(B/R)\rfloor = \lfloor(1024/100)\rfloor = 10$ records per block.
- The **number of blocks needed for the file** is
  $b = \lceil(r/bfr)\rceil = \lceil(30000/10)\rceil = 3000$ blocks.
- A **binary search on the data file** would need approximately
  $\lceil\log_2 b\rceil = \lceil(\log_2 3000)\rceil = 12$ block accesses.

- Now suppose that the ordering key field of the file is V = 9 bytes long, a block pointer is P = 6 bytes long, and we have constructed a primary index for the file.
- The **size of each index entry** is

  Ri = (9 + 6) = 15 bytes
- So the **blocking factor** for the index is

  bfri = $\lfloor$(B/Ri)$\rfloor$ = $\lfloor$(1024/15)$\rfloor$ = 68 entries per block.
- The **total number of index entries ri is equal to the number of blocks** in the data file, which is 3000.
- The **number of index blocks** is hence

  bi = $\lceil$(ri/bfri)$\rceil$ = $\lceil$(3000/68)$\rceil$ = 45 blocks.

- To perform a **binary search on the index file** would need $\lceil(\log_2 bi)\rceil = \lceil(\log_2 45)\rceil$ = 6 block accesses.
- To search for a record using the index, **we need one additional block access to the data file** for a total of 6 + 1 = 7 block accesses.
- An improvement over binary search on the data file, which required 12 disk block accesses.
- This is compared to an average linear search cost on data file of: (b/2)= 3000/2= 1500 block accesses.

Types of Single-Level Indexes

- Primary Index
  - Defined on an **ordered** data file
  - The data file is ordered on a **key field**
  - Includes one index entry *for* ***each block*** in the data file; the **index entry has the key field value for the *first record* in the block**, which is called the ***block anchor***
  - A similar scheme can use the *last record* in a block.

Indexes can also be characterized as **dense or sparse**

- Indexes can also be characterized as **dense or sparse**
  - A **dense index** has an index entry for every search key value (and hence **every record**) in the data file.
  - A **sparse (or nondense) index**, on the other hand, has index entries for **only some of the search values**

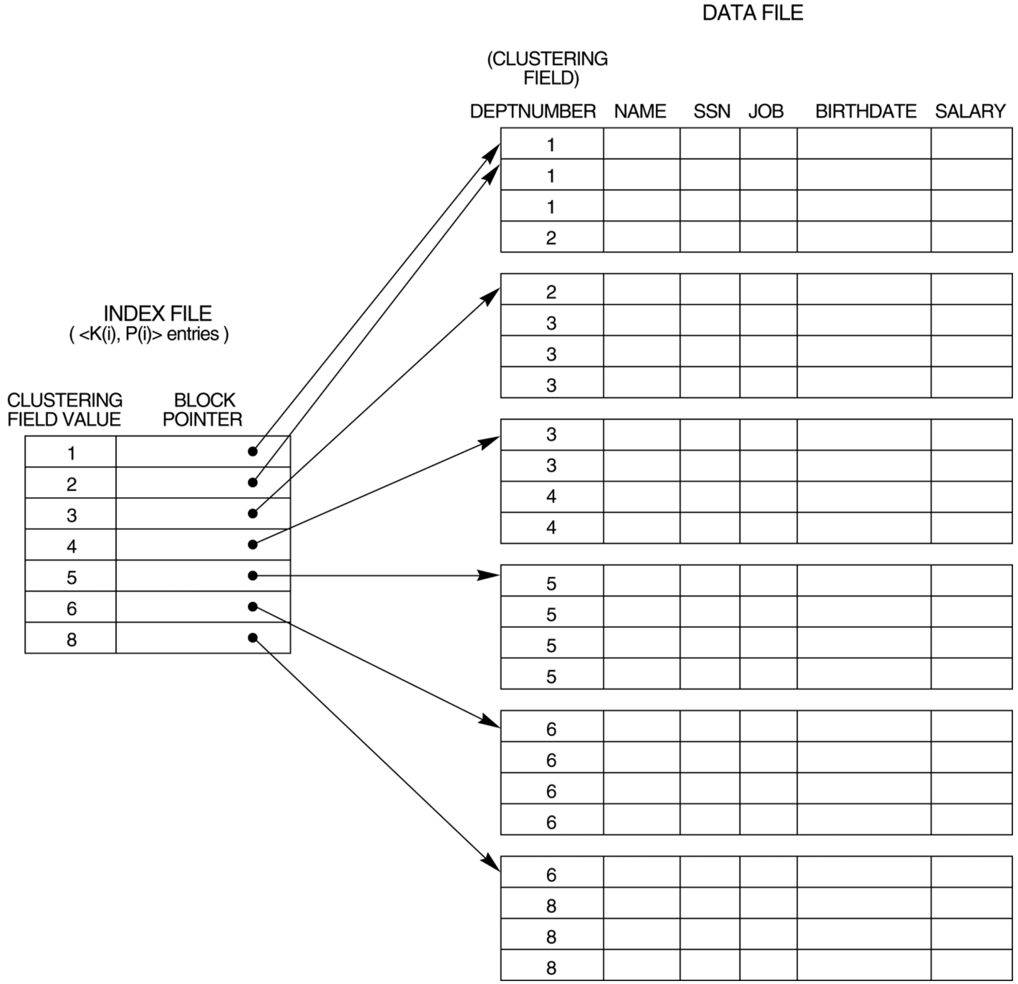
- Is primary index dense or sparse?
  - A sparse index has **fewer entries** than the number of records in the file.
  - Thus, **a primary index is a nondense (sparse) index**, since it includes an entry for each disk block of the data file and the keys of its anchor record rather than for every search value (or every record).

Clustering Index

- If file records are **physically ordered on a non key field** (**the ordering field is not a key field**)—which does not have a distinct value for each record—that field is called **the clustering field and the data file is called a clustered file.**
- Another type of index, called a **clustering index**, can be used, to **speed up retrieval of all the records that have the same value** for the clustering field.
- A clustering index is also **an ordered file with two fields**:
  - The first field is of the **same type as the clustering field of the data file.**
  - The second field is a **disk block pointer**.
- There is **one entry** in the clustering index **for each distinct value** of the clustering field.
- It contains the value and a pointer to **the first block in the data file that has a record with that value for its clustering field**.

A Clustering Index Example

- FIGURE 14.2
  A clustering index on the DEPTNUMBER ordering non-key field of an EMPLOYEE file.



Another Clustering Index Example

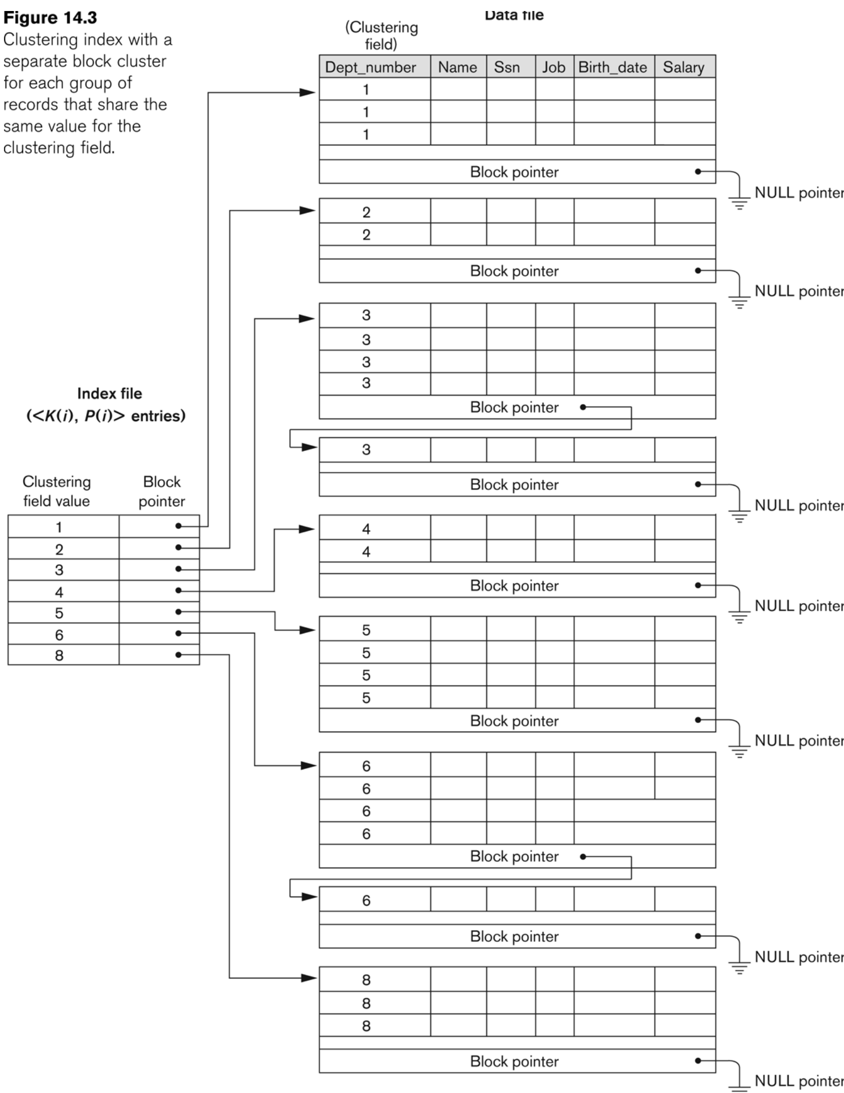


Figure 14.3
Clustering index with a separate block cluster for each group of records that share the same value for the clustering field.

Types of Single-Level Indexes

- **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.**
  - Dense or spars?
    - It is another example of ***nondense (sparse)*** index

Types of Single-Level Indexes

- **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**.
    - There can be ***many*** **secondary indexes** (and hence, indexing fields) for the same file.
  - Dense or sparse?
    - Includes **one entry *for each record*** in the data file; hence, it is a ***dense index***

How many indexes?

- Notice that a file can have **at most one physical ordering field**, so it can have **at most one primary index or one clustering index, but not both**.
- **A secondary index** can be specified on **any non-ordering field** of a file. **A data file can have several secondary indexes in addition to its primary access method.**
- The secondary index may be created on a field that is **a candidate key and has a unique value in every record**, or on a **non-key field with duplicate values**.

Secondary Index Access Structure on a Key (Unique) Field

- In this case there is one index entry for each record in the data file, which contains **the value of the field** for the record and **a pointer either to the block in which the record is stored or to the record itself**.
- Hence, such an index is **dense**.
- The entries are **ordered** so we can perform a **binary search.**

Example of a Dense Secondary Index

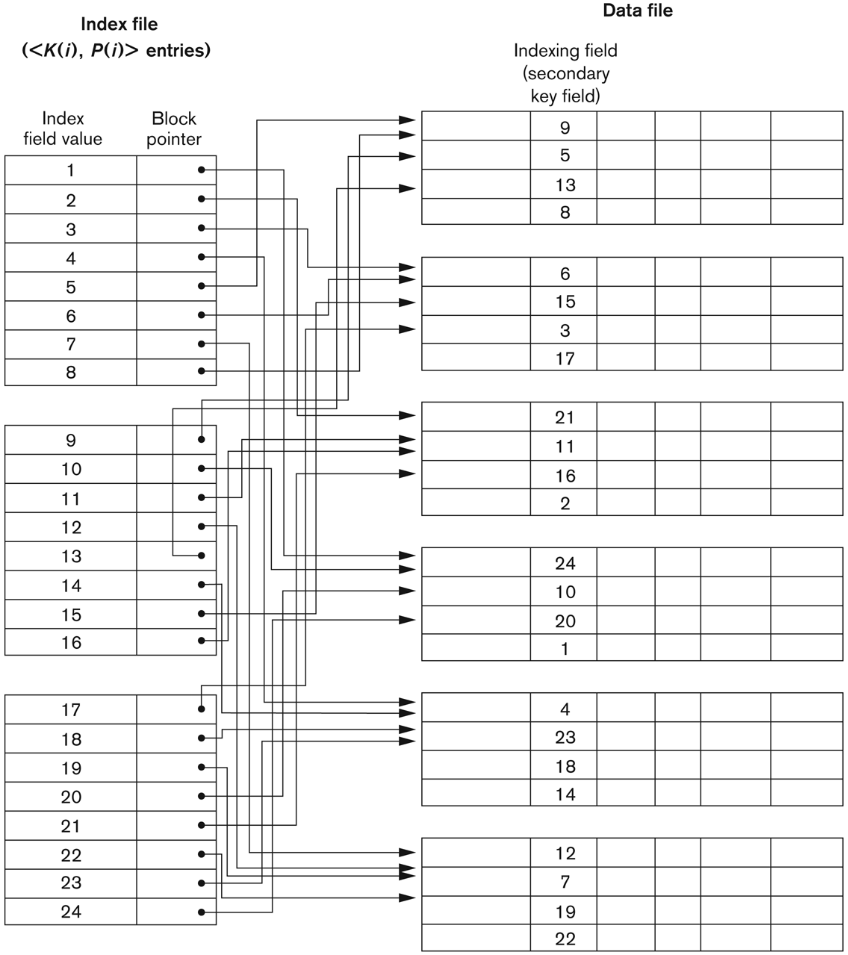


Figure 14.4
A dense secondary index (with block pointers) on a nonordering key field of a file.

- A secondary index usually needs **more storage space and longer search time** than does a primary index, because of its larger number of entries.

- However, the **improvement in search time** for an arbitrary record is **much greater for a secondary index than for a primary index**, since we would have to do a **linear search on the data file if the secondary index did not exist.** For a primary index, we could still use a binary search on the main file, even if the index did not exist

Example 2 illustrates **the improvement in number of blocks accessed**.

- Consider the file of Example 1 with **r = 30,000** fixed-length records of size **R = 100** bytes stored on a disk with block size **B = 1024** bytes. The file has **b = 3000** blocks, as calculated in Example 1.
- Suppose we want to search for a record with a specific value for the secondary key—a **non-ordering key field** of the file that is V = 9 bytes long.
- **Without the secondary index**, to do **a linear search** on the file would require

  b/2 = 3000/2 = 1500 block accesses on the average.

- Suppose that we construct a **secondary index** on that non-ordering key field of the file.
- As in Example 1, a block pointer is P = 6 bytes long, so each index entry is **Ri = (9 + 6) = 15 bytes**
- and the blocking factor for the index is

  $bfri = \lfloor (B/Ri) \rfloor = \lfloor (1024/15) \rfloor = 68$ entries per block.
- In a dense secondary index such as this, the **total number of index entries ri is equal to the number of records in the data file**, which is 30,000.
- The number of blocks needed for the index is hence

  $bi = \lceil (ri\ /bfri) \rceil = \lceil (30000/68) \rceil = 442$ blocks.

- A **binary search** on this secondary index needs
  - $\lceil(\log_2 b_i)\rceil = \lceil(\log_2 442)\rceil$ = 9 block accesses.
- To search for a record using the index, we need **an additional block access** to the data file for a total of 9 + 1 = 10 block accesses
- A **vast improvement over** the 1500 block accesses needed on the average for a linear search, **but slightly worse** than the 7 block accesses required for the primary index. This difference arose because the primary index was **non-dense and hence shorter**, with only 45 blocks in length.

Properties of Index Types

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.

Multi-Level Indexes

- Because 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**

A Two-level Primary Index

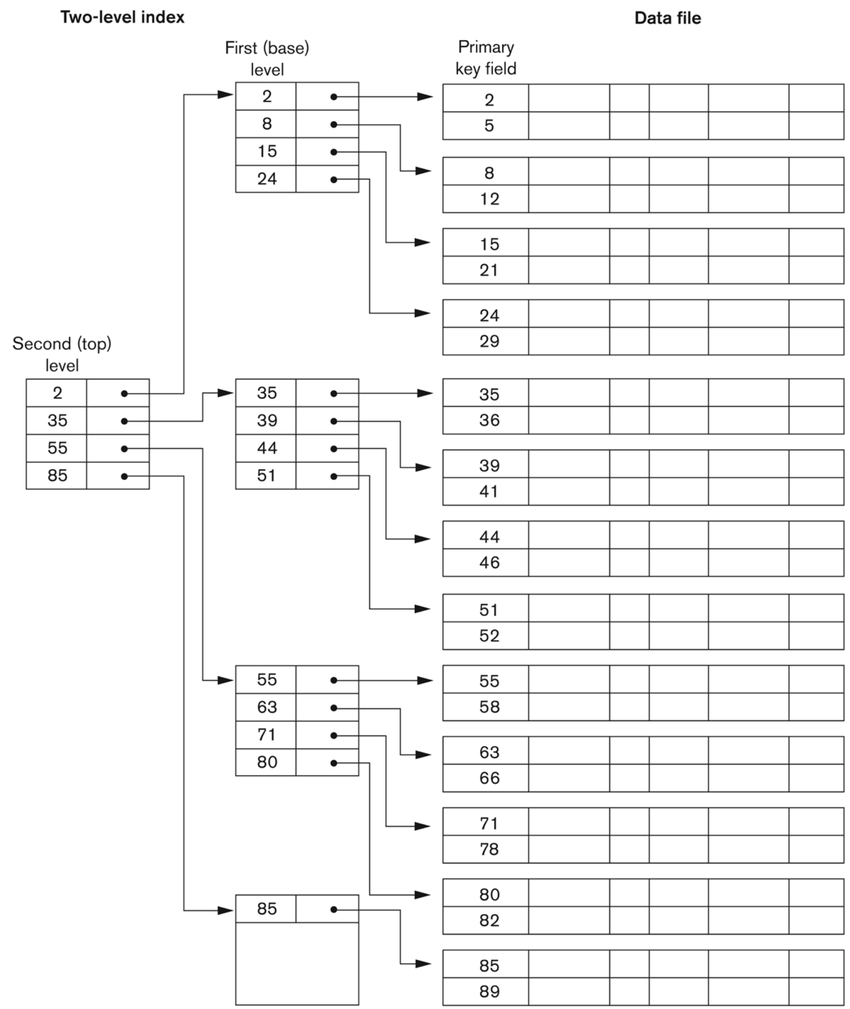


Figure 14.6
A two-level primary index resembling ISAM (Index Sequential Access Method) organization.

Example 3 illustrates the improvement in number of blocks accessed when a multilevel index is used to search for a record.

- Suppose that the **dense secondary index of Example 2** is converted into a multilevel index. We calculated the index **blocking factor bfri = 68 index entries per block**, which is also the **fan-out fo for the multilevel index**;
- The **number of first level blocks**
  b1 = $\lceil (ri/fo) \rceil = \lceil (3000/68) \rceil$ = 442 blocks was also calculated.
- The number of **second-level blocks** will be
  b2 = $\lceil (b1/fo) \rceil = \lceil (442/68) \rceil$ = 7 blocks,
- and the number of **third-level blocks** will be
  - b3 = $\lceil (b2/fo) \rceil = \lceil (7/68) \rceil$ = **1 block**.
- Hence, the **third level is the top level of the index**, and t = 3.

Example 3 illustrates the improvement in number of blocks accessed when a multilevel index is used to search for a record.

- 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 (no. of levels) + 1 = 4 block accesses. **Compare** this to Example 2, where 10 block accesses were needed when a single-level index and binary search were used.