Database Index

CSCI 435

Slides

- https://github.com/lxie21/csci435

Find 10 “NULL”s in Your Text Book

- Start from the first page, word by word.
- Look at “Index” at the end of the book.

- In the relational database, INDEX is used to locate rows in a table without the need to inspect every row in the table.
- Indexes are stored in a special TABLE, which contains information:
  - Column(s) used to locate rows in the table
  - Where rows are physically located

Search with an index

- ***Single-level index*** = auxiliary file to make search for a record in a file more efficient
- Specified on one or more fields
- ***Access path on a field*** = index ordered by field value of the form <**field value, pointer to record>**
- Index file usually occupies considerably fewer blocks than file because its entries are much smaller
- Binary search on index yields a pointer to record
- On a large file, biggest delays are block accesses, so retrieval cost is measured in terms of them

How big is an index search?

- Given data file EMPLOYEE(NAME, SSN, ADDRESS, JOB, SAL, ...) with record size R = 100 bytes, block size B = 1024 bytes, r = 30000 records
- **Blocking factor** bfr = $\lfloor B/R \rfloor = \lfloor 1024/100 \rfloor$ = 10 records/block
- Number of file blocks b = (r/bfr) = $\lceil 30000/10 \rceil$ = 3000 blocks
- For dense index on SSN, field size VSSN = 9 bytes, record pointer size PR = 6 bytes
- Index entry size RI=(VSSN+ PR) = (9 + 6) = 15 bytes
- **Index blocking factor** BfrI = $\lfloor B / RI \rfloor = \lfloor 1024/ 15 \rfloor$ = 68 entries/block
- Number of index blocks bi = (r / BfrI)= $\lceil 30000/68 \rceil$= 45 blocks
- Average search cost in block accesses
  - Linear (no index): (r/2) = 30000/2 = 15000 block accesses
  - Binary (no index): $\log_2 b = \log_2 30000$= 12 block accesses
  - Index and binary search: $\log_2 bi + 1 = \log_2 45 + 1$= 7 block accesses

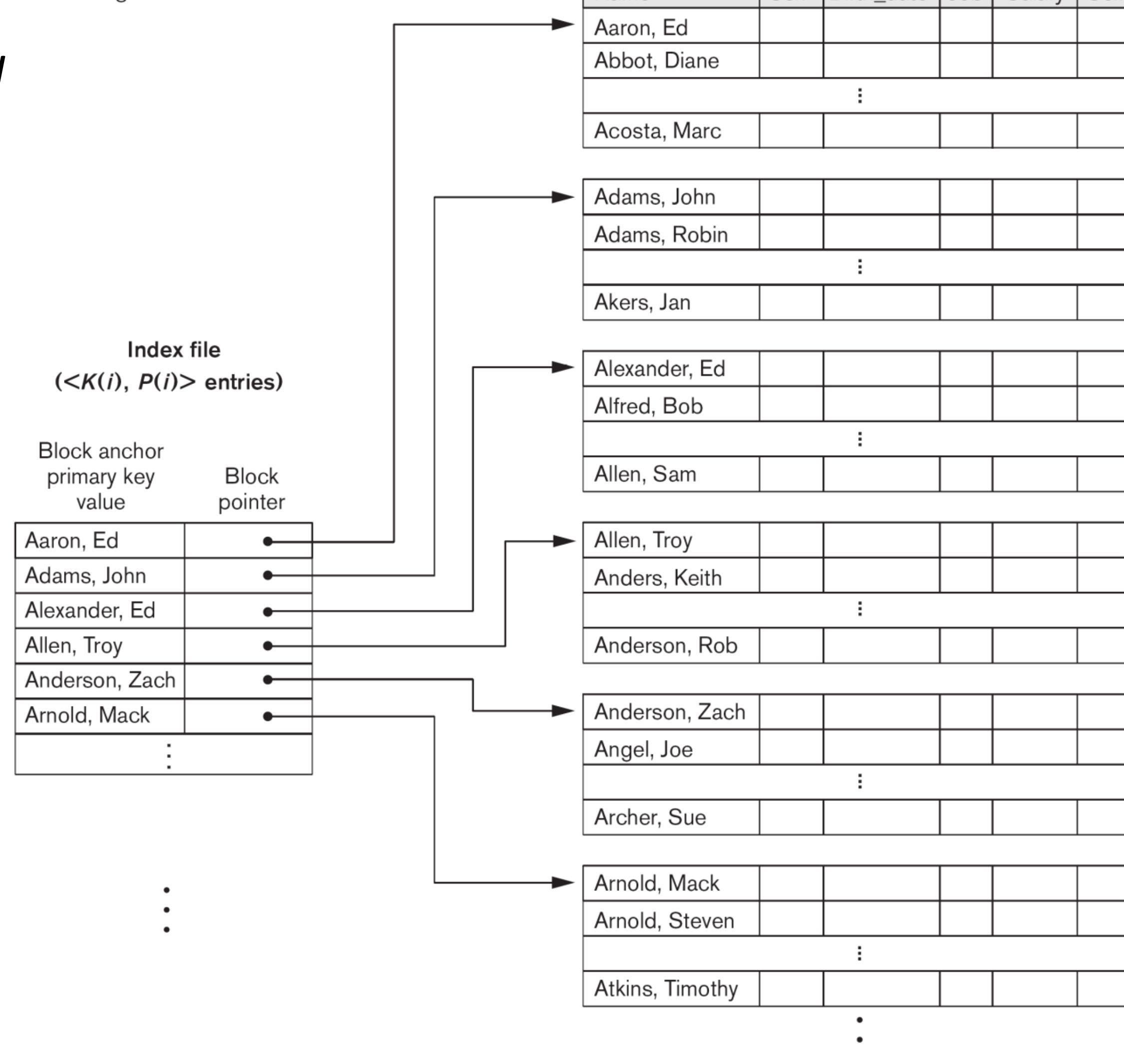
Important distinctions

- ***Dense index*** has an entry for *every* search key value (and hence every record) in the file while…
- ***Sparse index*** has entries only for some search values
- ***Primary index*** uses a specified key for a given file while…
- ***Secondary index*** uses a non-ordering field for the same file and incurs additional overhead

Primary index

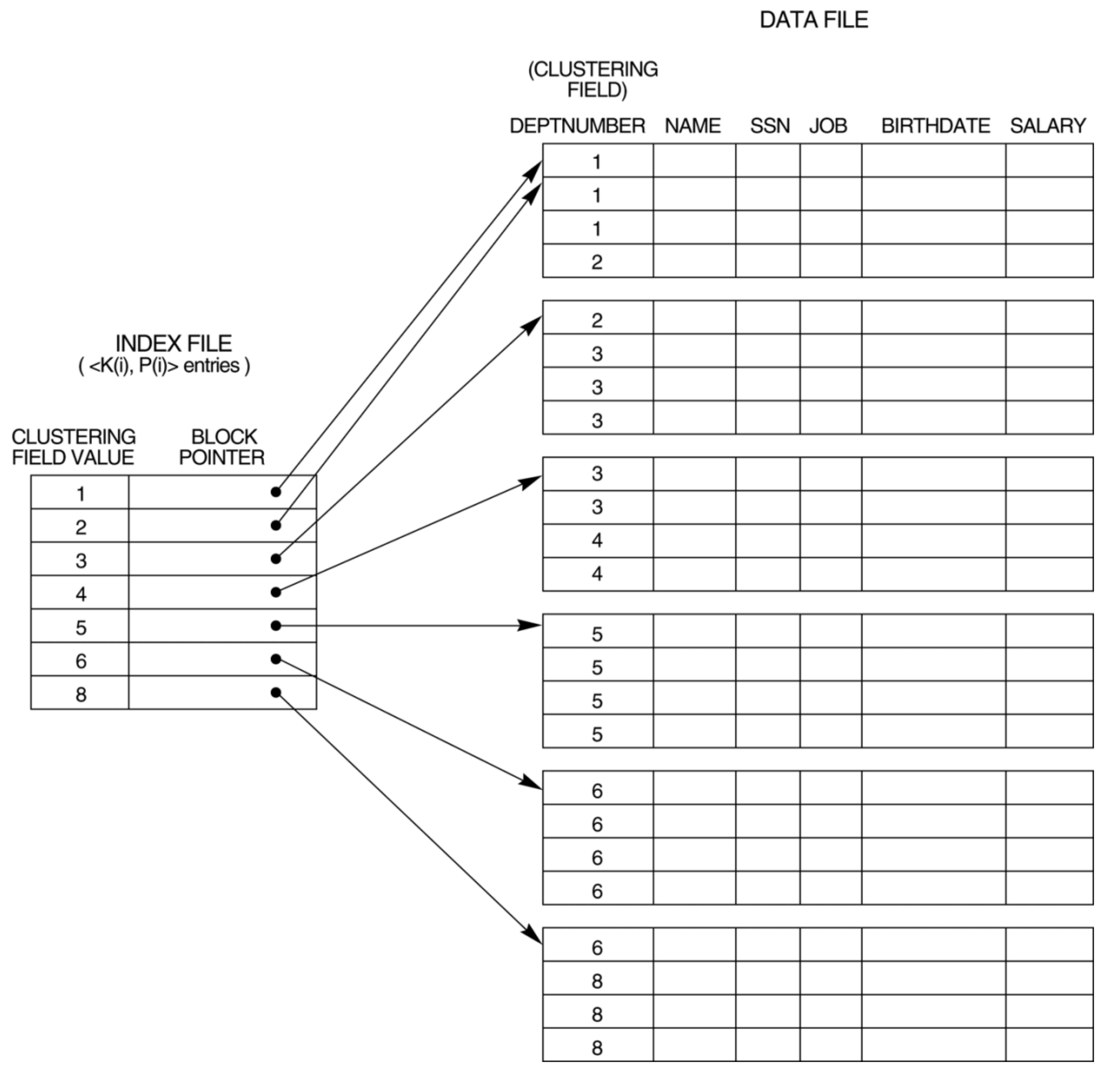
- Defined on data file *ordered* on some **key field(s)**
- Includes one entry *for each block* in data file
- Entry has key field value for the ***block anchor*** = first record in the block
- Can also use the *last record* in a block
- ***Primary index*** is a sparse index, since it includes an entry for each disk *block* of the file and the keys of its anchor record rather than for every search value

Figure 18.1
Primary index on the ordering key field of the file shown in Figure 17.7.



Clustering index

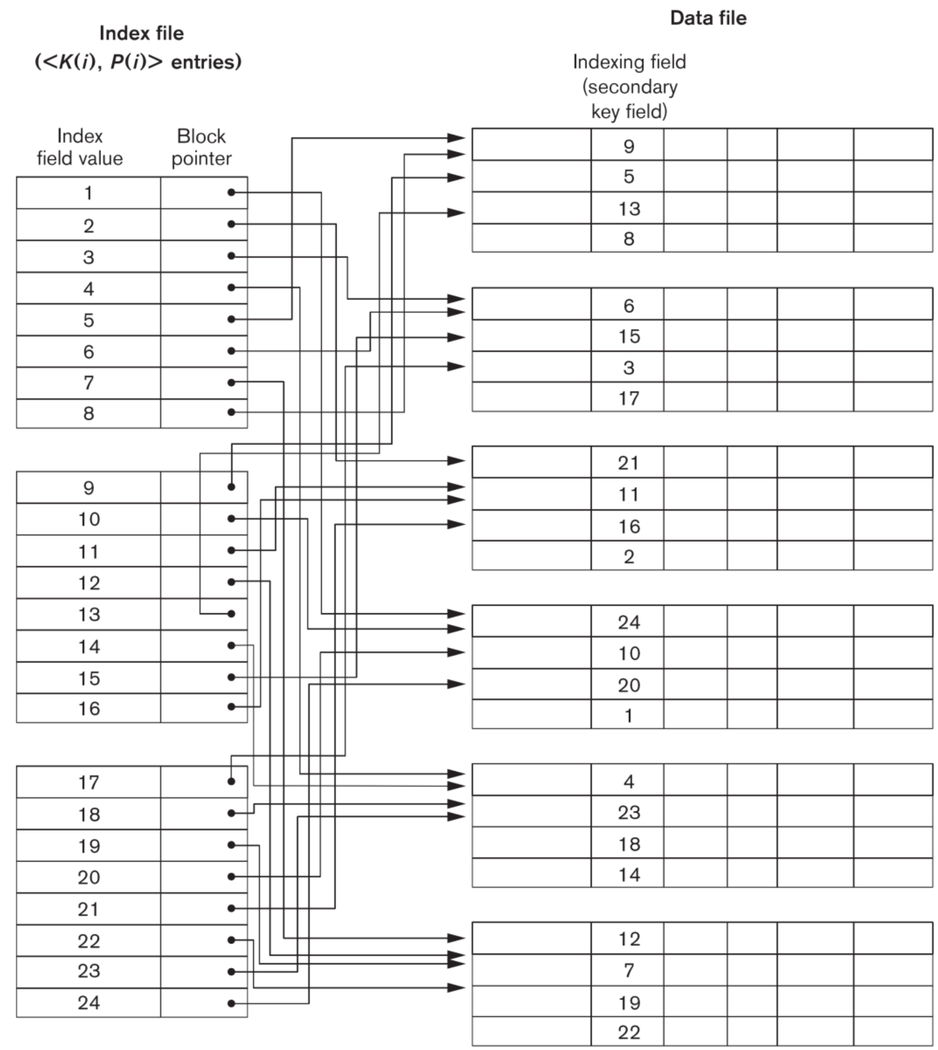
- Defined on data file ordered on a ***non-key field*** (need not be distinct)
- Includes one index entry *for each distinct value* of the field
- Index entry points to the first data block that contains records with that field value
- Sparse
- INSERT and DELETE are relatively straightforward



Secondary index

- Another way to access file for which *some primary access already exists*
- On candidate key or on non-key
- Ordered file with two fields
- Same data type as some **non-ordering field** of the file that is an indexing field
- *Either* a block pointer or a record pointer
- There can be ***many*** secondary indexes (and hence, indexing fields) for the same file
- Dense: one entry *for each record*

Figure 18.4
A dense secondary index (with block pointers) on a nonordering key field of a file.



Another example of a secondary index

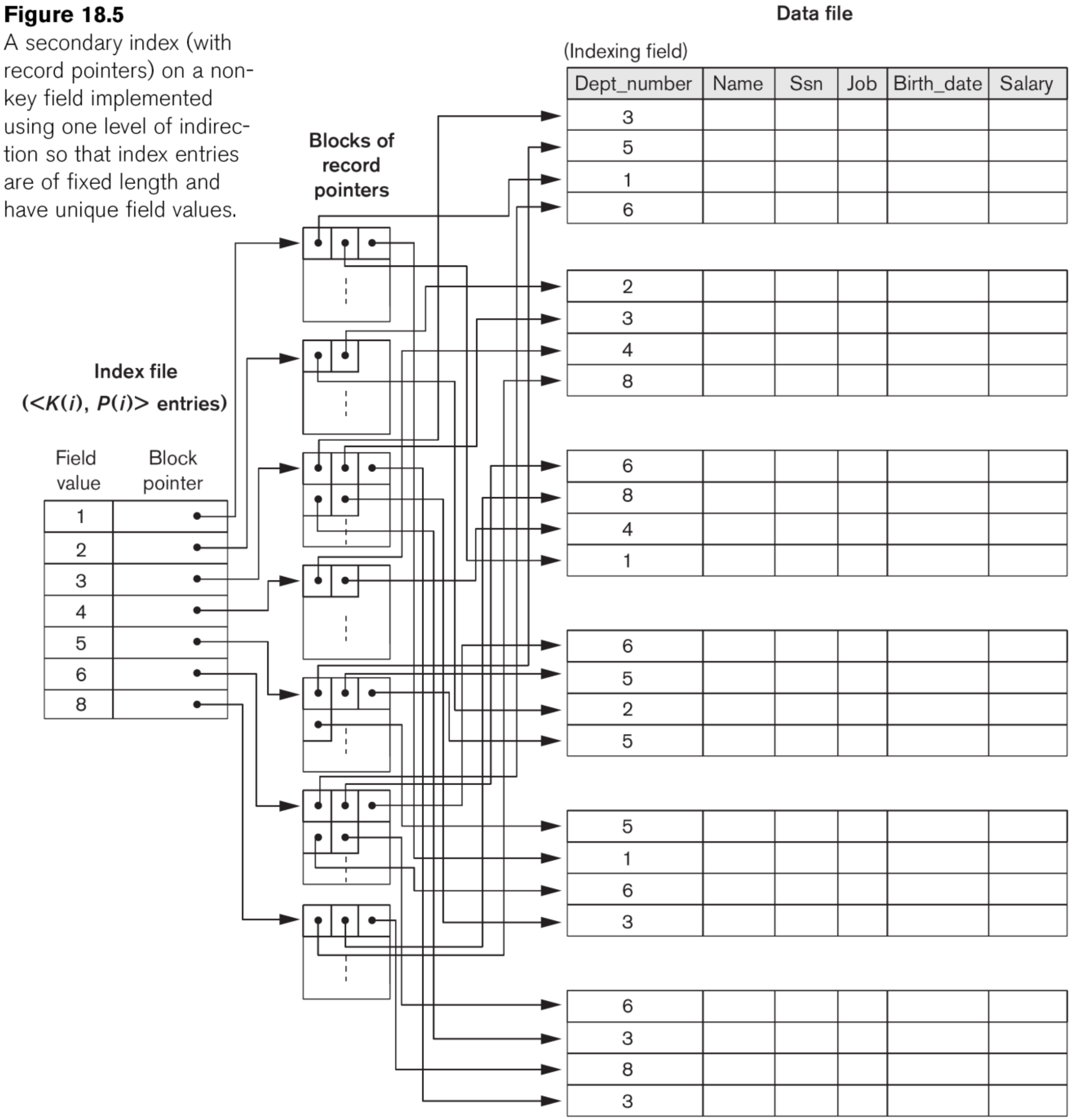


Figure 18.5
A secondary index (with record pointers) on a non-key field implemented using one level of indirection so that index entries are of fixed length and have unique field values.

Properties of index types

Table 18.1 Types of Indexes Based on the Properties of the Indexing Field

	Index Field Used for Physical Ordering of the File	**Index Field Not Used for Physical Ordering of the File**
Indexing field is key	Primary index	Secondary index (Key)
Indexing field is nonkey	Clustering index	Secondary index (NonKey)

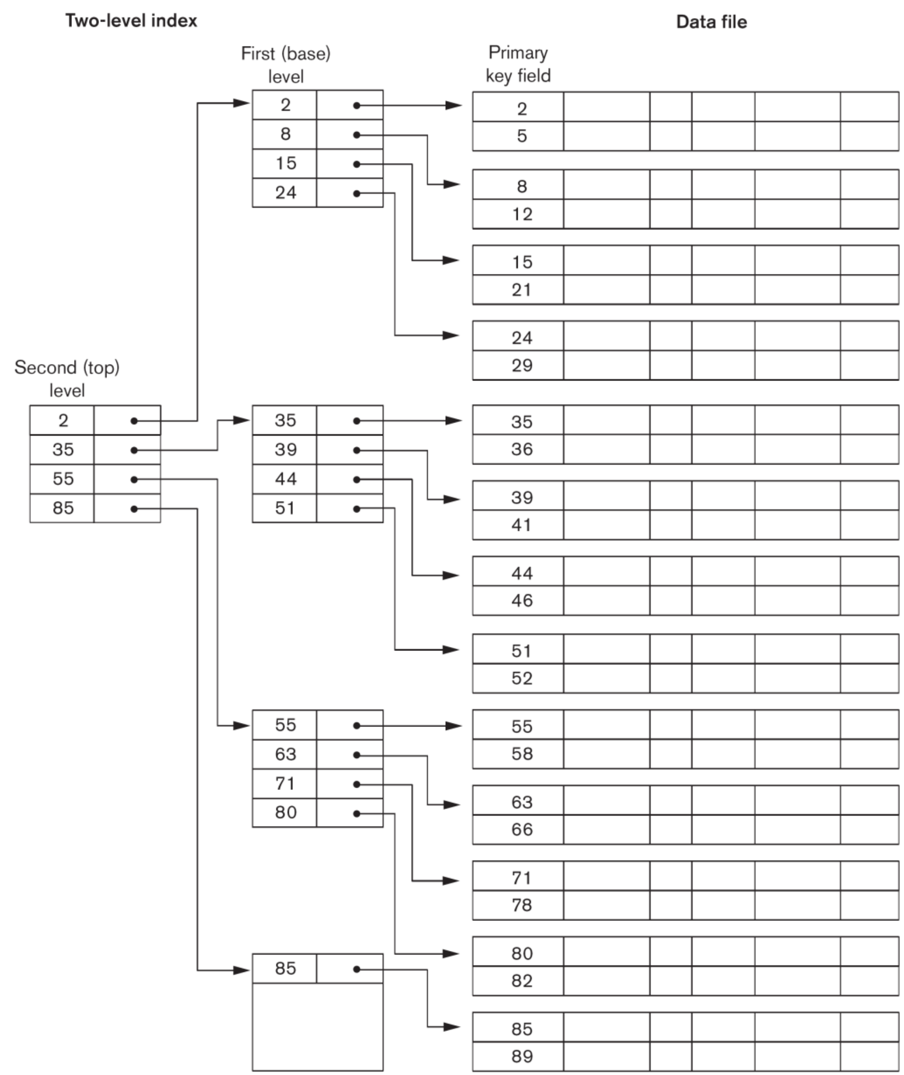
Table 18.2 Properties of Index Types

Type of Index	**Number of (First-level) Index Entries**	**Dense or Nondense (Sparse)**	**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

Multilevel indices

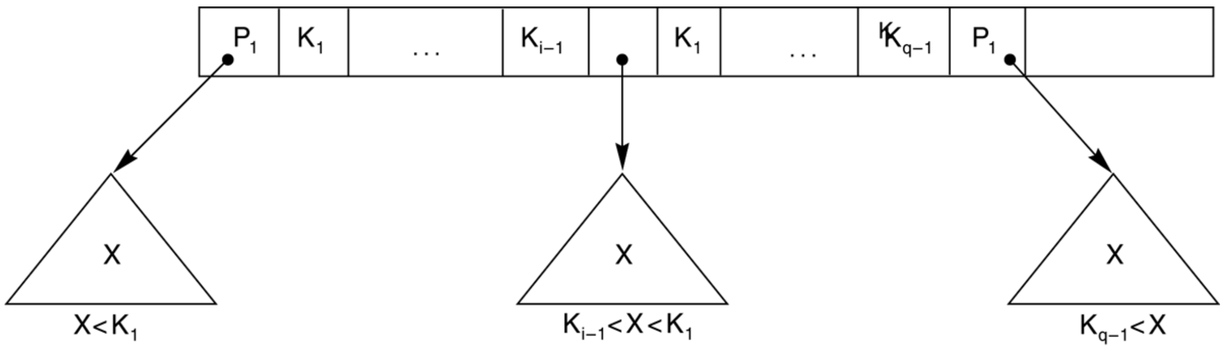
- Create ***primary index to the index itself***
- ***First-level index*** = original index file
- ***Second-level index*** = index to the index
- Can repeat the process: create third, fourth, ..., top level until all top level entries ***fit in one disk block***
- Multi-level index can be created for *any* type of first-level index (primary, secondary, clustering) as long as the first level index consists of *more than one* disk block
- Kind of ***search tree***, but INSERT and DELETE are severe problems because every level of the index is an *ordered file*

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

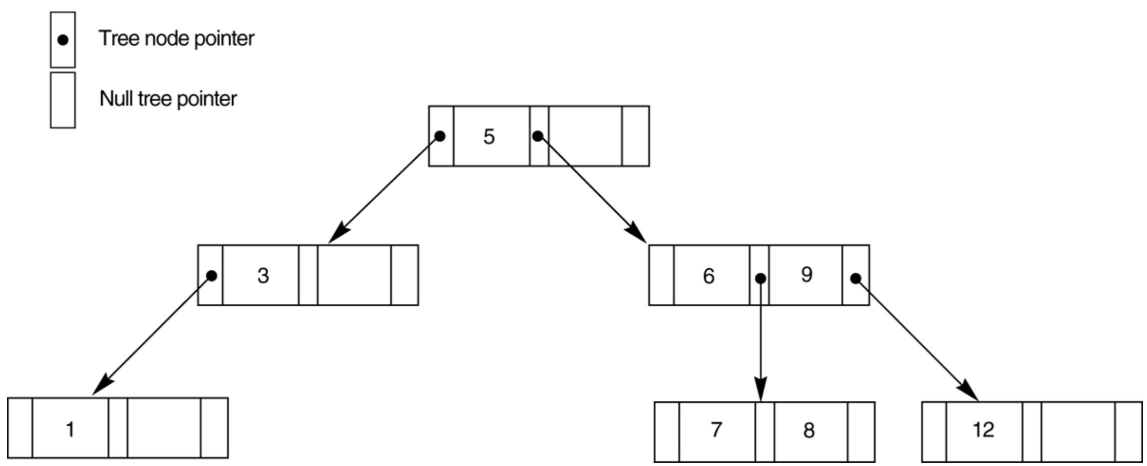


Search tree nodes

- Point to subtrees beneath them



- Here's one of order 3



Dynamic multilevel indexes

- Most multi-level indexes use B-trees or B+-trees because of the insertion and deletion problem
- They leave space in each tree node (disk block) for new index entries
- Each node corresponds to a disk block
- Each node is kept between half-full and completely full
- INSERT into a node that is ***not*** full is quite efficient
- INSERT into a full node causes a ***split*** into two nodes
- Splitting may propagate to other tree levels
- DELETE is quite efficient if node does not become less than half full
- DELETE that causes a node to become less than half full causes a merger with neighboring nodes

B-tree Structures

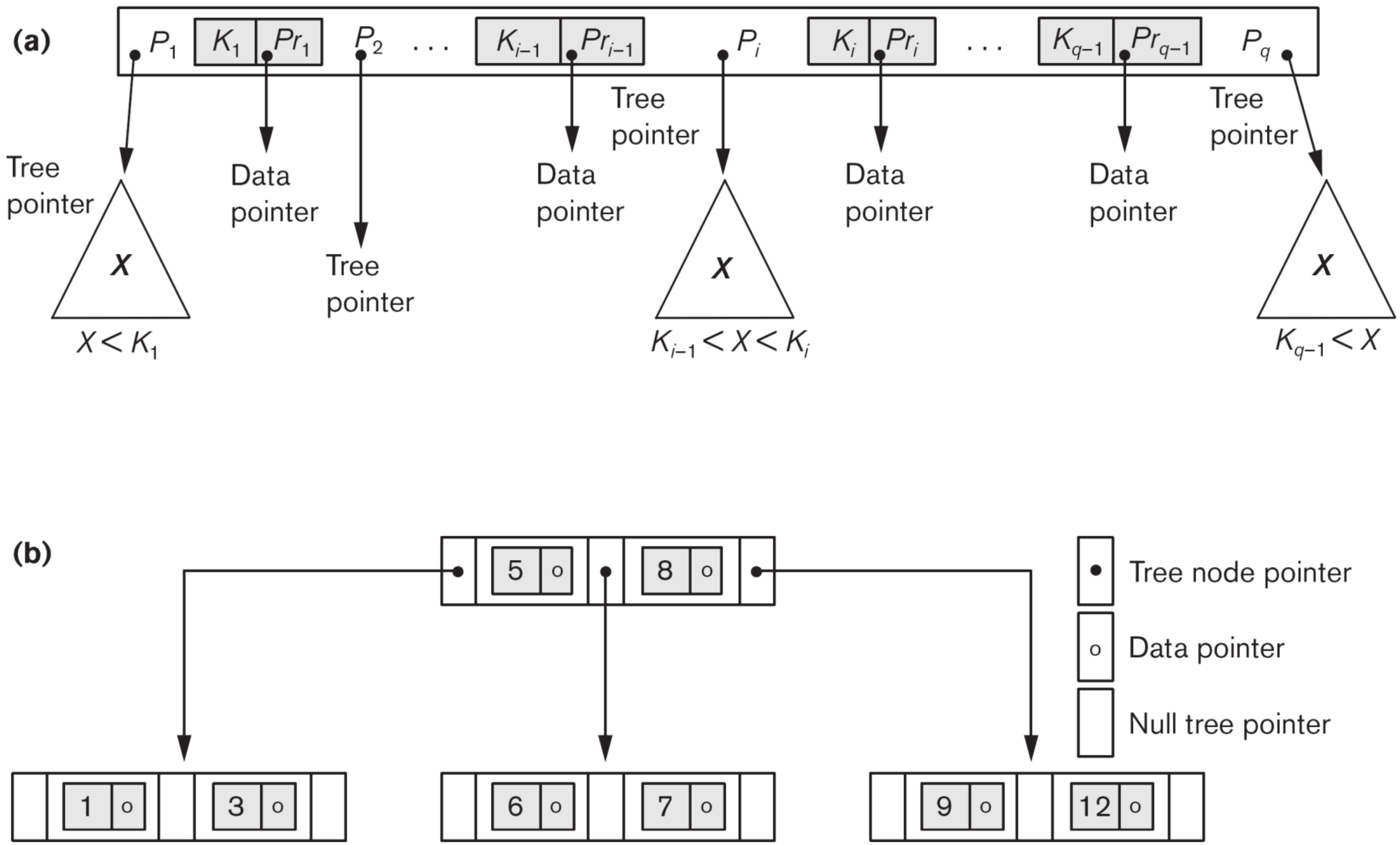


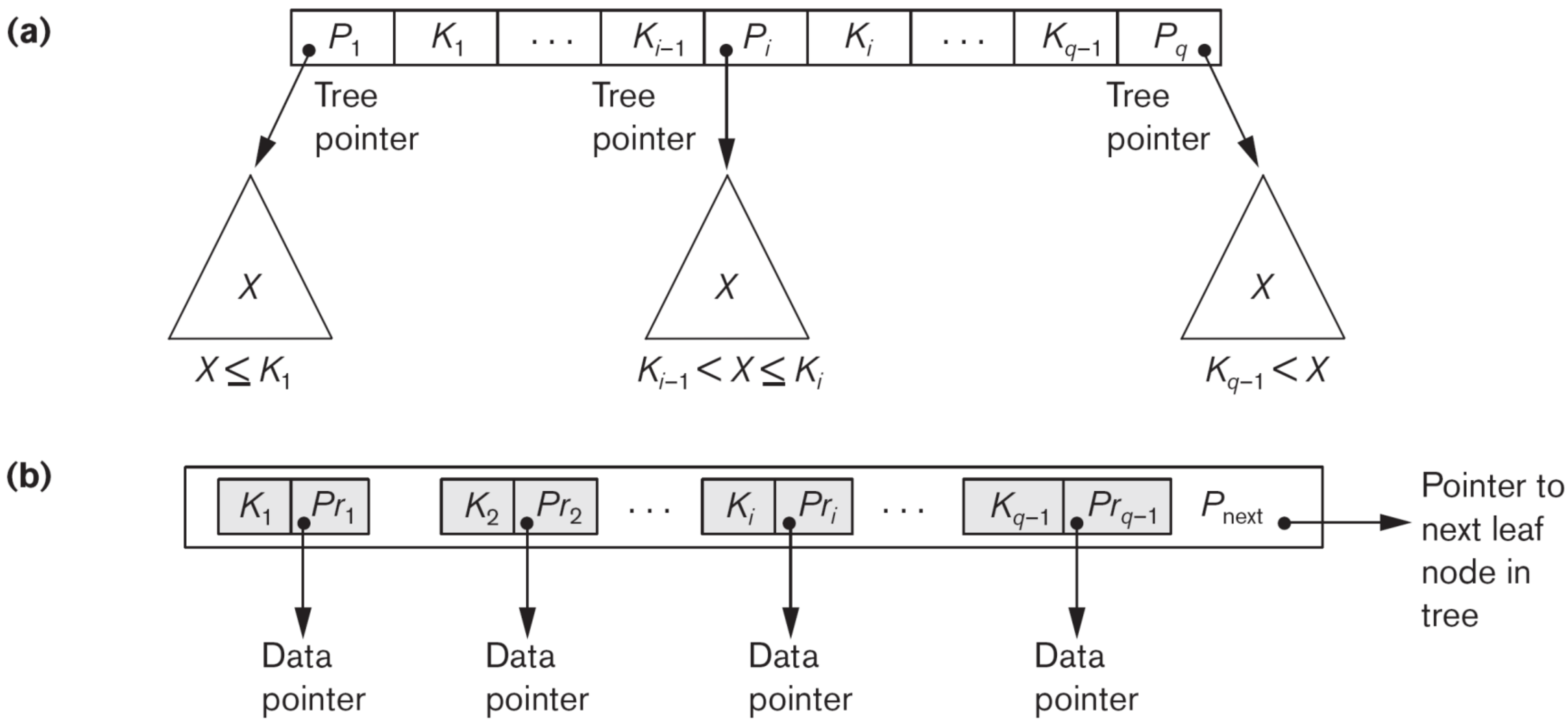
Figure 18.10
B-tree structures. (a) A node in a B-tree with $q - 1$ search values. (b) A B-tree of order $p = 3$.The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.

Differences between B-trees and B+-trees

- B-tree: pointers to data records exist at ***all*** levels of the tree
- B+-tree: ***all*** pointers to data records exists at ***leaves only***
- B+-tree can have fewer levels (or hold more search values) than its corresponding B-tree

Nodes in a B+-tree

Figure 18.11
The nodes of a B$^+$-tree. (a) Internal node of a B$^+$-tree with $q - 1$ search values.
(b) Leaf node of a B$^+$-tree with $q - 1$ search values and $q - 1$ data pointers.



INSERT on B+-tree

- *p = 3*
- *pleaf = 2*
- Insert 8,5,1,7,3,12,9,5

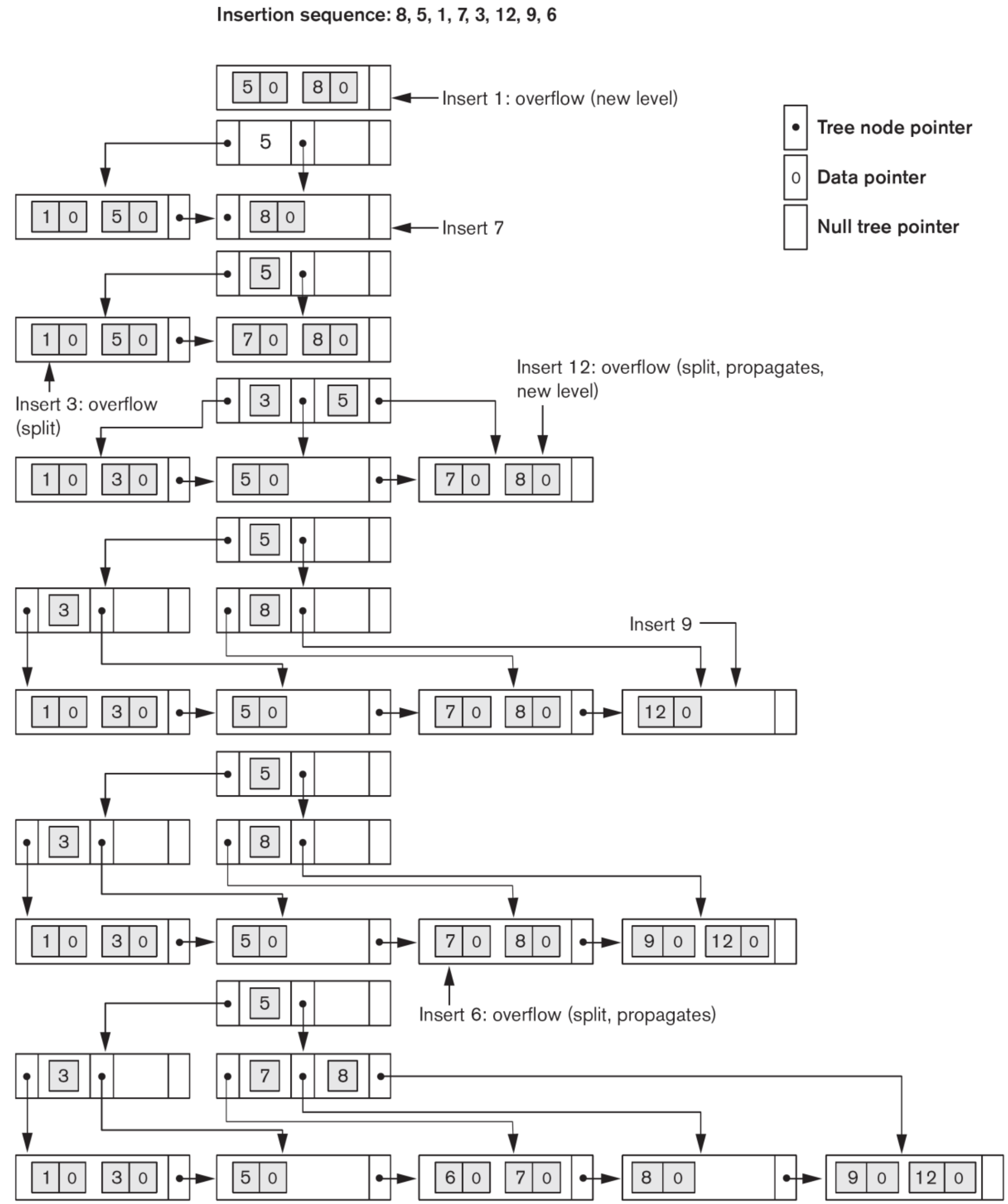


Figure 18.12
An example of insertion in a B^+-tree with $p = 3$ and $p_{leaf} = 2$.

DELETE on B+ tree

- 5, 12, 9

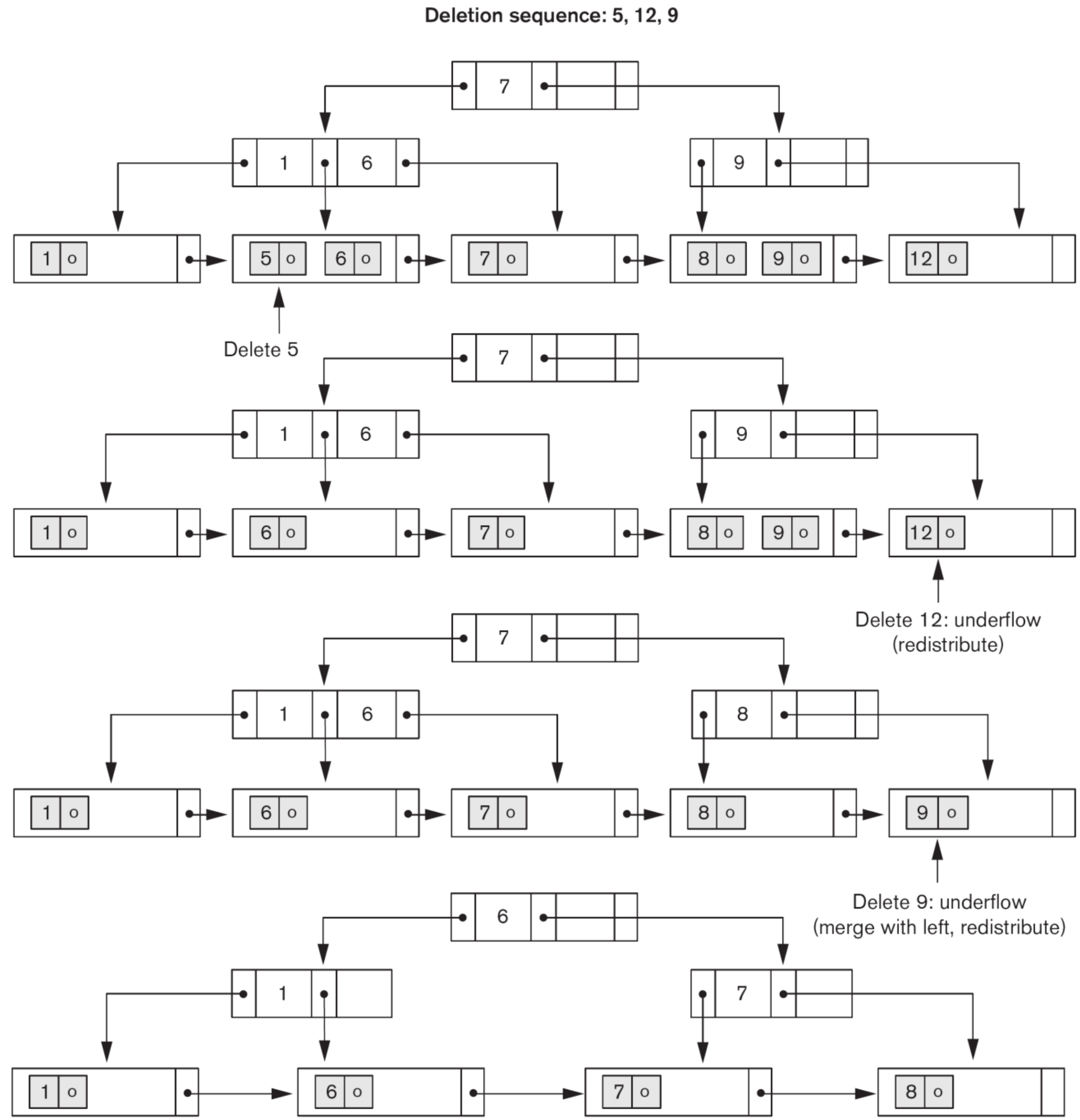


Figure 18.13
An example of deletion from a B^{+}-tree.

Indices on multiple keys

- Any index method can incorporate multiple attributes, using each successive field in ascending order
- ***Partitioned hashing*** = bucket key has parts that reference each field
  - Cannot handle range queries
  - Example: zip code ⇒ 01011 and salesperson ⇒ 010 so bucket number is 01011 010
- ***Grid file*** = array of pointers to buckets
  - One array for each attribute
  - Partition values of each field so number of records with that value are **uniformly distributed** (***linear scale***)
  - Create array where indices indicate item, and entry holds pointer
  - Space overhead

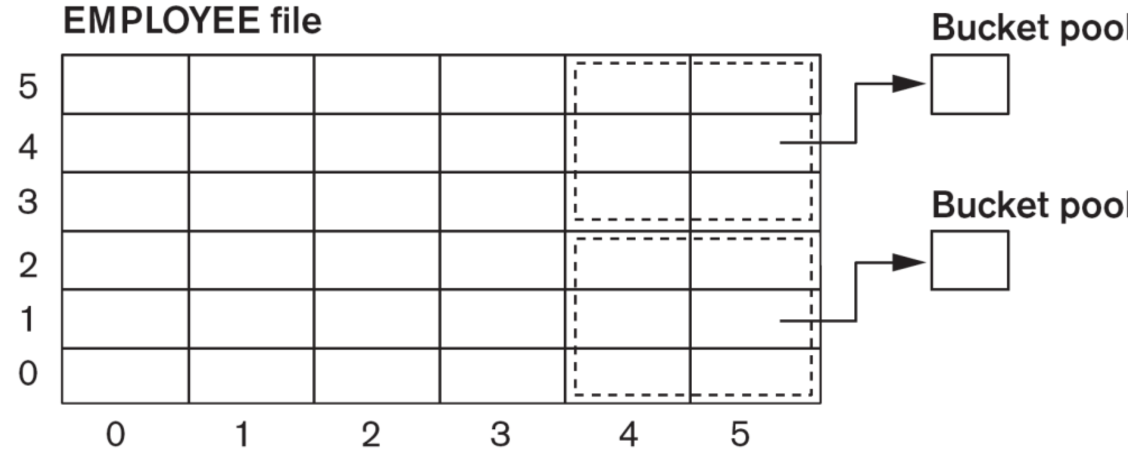
Grid file example

Figure 18.14
Example of a grid array on Dno and Age attributes.

Dno

0	1, 2
1	3, 4
2	5
3	6, 7
4	8
5	9, 10

Linear scale for Dno



Linear Scale for Age

0	1	2	3	4	5
< 20	21–25	26–30	31–40	41–50	> 50

Advantages and disadvantages of indices

Summary

- Single-level ordered indices
  - Primary
  - Clustering
  - Secondary
- Multilevel indices
- Dynamic multilevel indices
- Indices on more than 1 key