Indexing Structures for Files and Physical Database Design

Introduction

- Indexes used to speed up record retrieval in response to certain search conditions
- Index structures provide secondary access paths
- Any field can be used to create an index
  - Multiple indexes can be constructed
- Most indexes based on ordered files
  - Tree data structures organize the index

4.1 Types of Single-Level Ordered Indexes

- Ordered index similar to index in a textbook
- Indexing field (attribute)
  - Index stores each value of the index field with list of pointers to all disk blocks that contain records with that field value
- Values in index are ordered
- Primary index
  - Specified on the ordering key field of ordered file of records

Types of Single-Level Ordered Indexes (cont'd.)

- Clustering index
  - Used if numerous records can have the same value for the ordering field
- Secondary index
  - Can be specified on any nonordering field
  - Data file can have several secondary indexes

Primary Indexes

- Ordered file with two fields
  - Primary key, *K(i)*
  - Pointer to a disk block, *P(i)*
- One index entry in the index file for each block in the data file
- Indexes may be dense or sparse
  - Dense index has an index entry for every search key value in the data file
  - Sparse index has entries for only some search values

Primary Indexes (cont'd.)

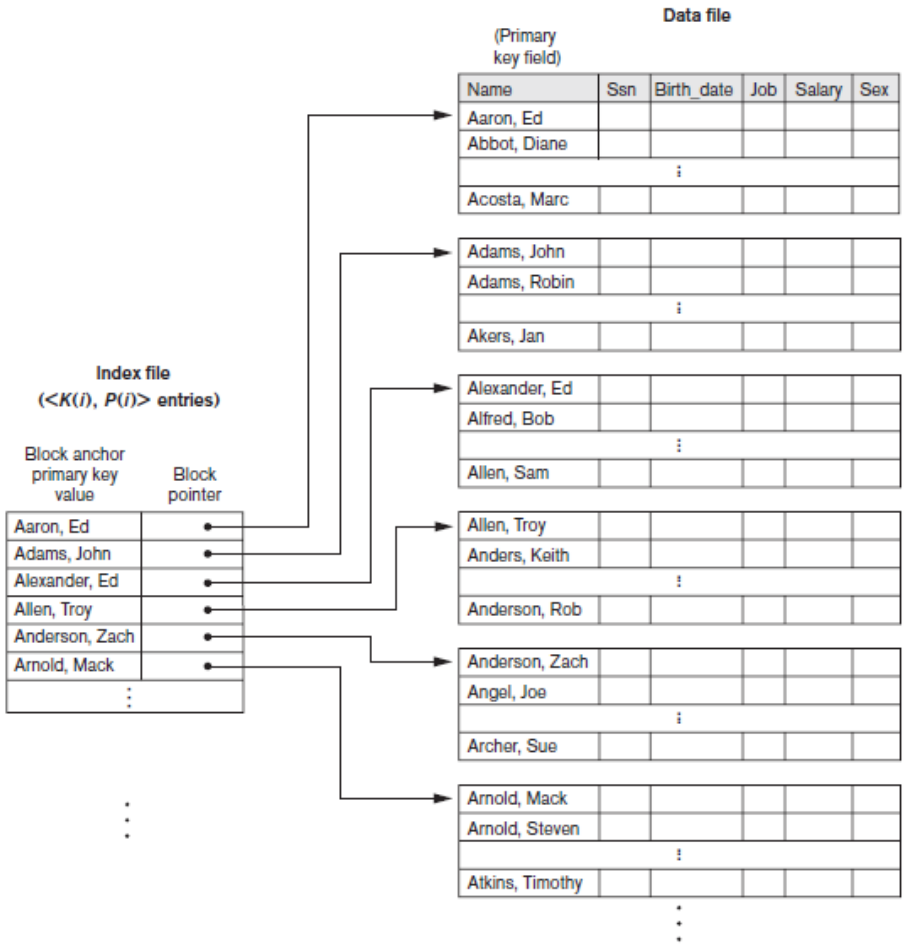


Figure 17.1 Primary index on the ordering key field of the file shown in Figure 16.7

Primary Indexes (cont'd.)

- Major problem: insertion and deletion of records
  - Move records around and change index values
  - Solutions
    - Use unordered overflow file
    - Use linked list of overflow records

Clustering Indexes

- Clustering field
  - File records are physically ordered on a nonkey field without a distinct value for each record
- Ordered file with two fields
  - Same type as clustering field
  - Disk block pointer

Clustering Indexes (cont'd.)

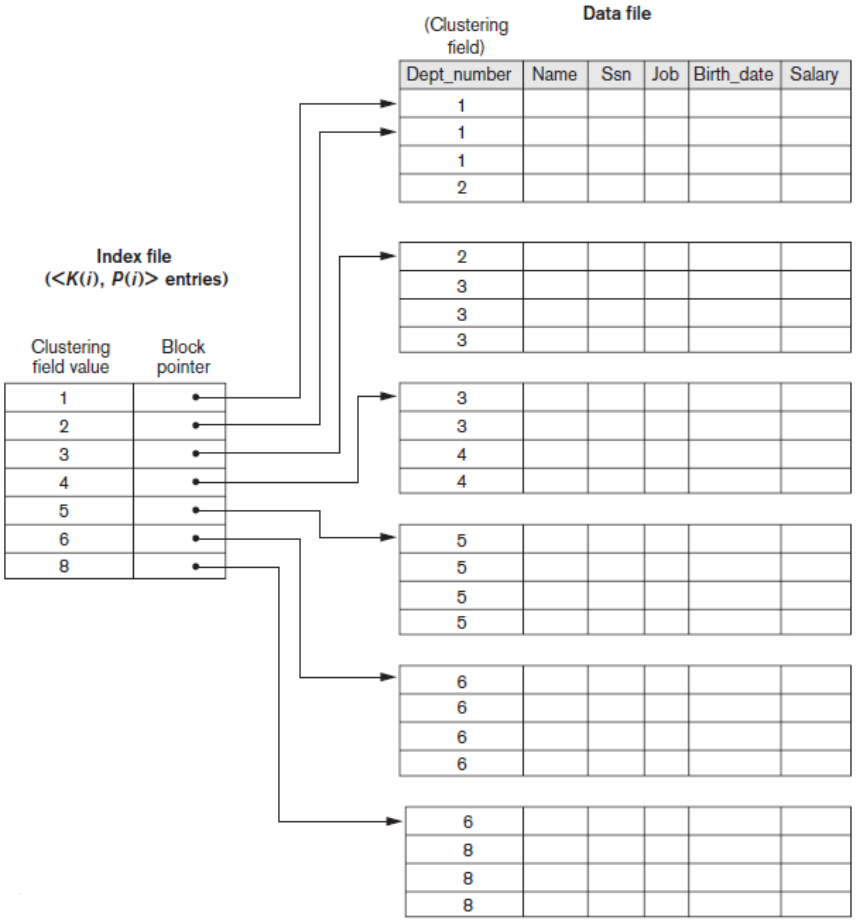


Figure 17.2 A clustering index on the Dept_number ordering nonkey field of an EMPLOYEE file

Secondary Indexes

- Provide secondary means of accessing a data file
  - Some primary access exists
- Ordered file with two fields
  - Indexing field, *K(i)*
  - Block pointer or record pointer, *P(i)*
- Usually need more storage space and longer search time than primary index
  - Improved search time for arbitrary record

Secondary Indexes (cont'd.)

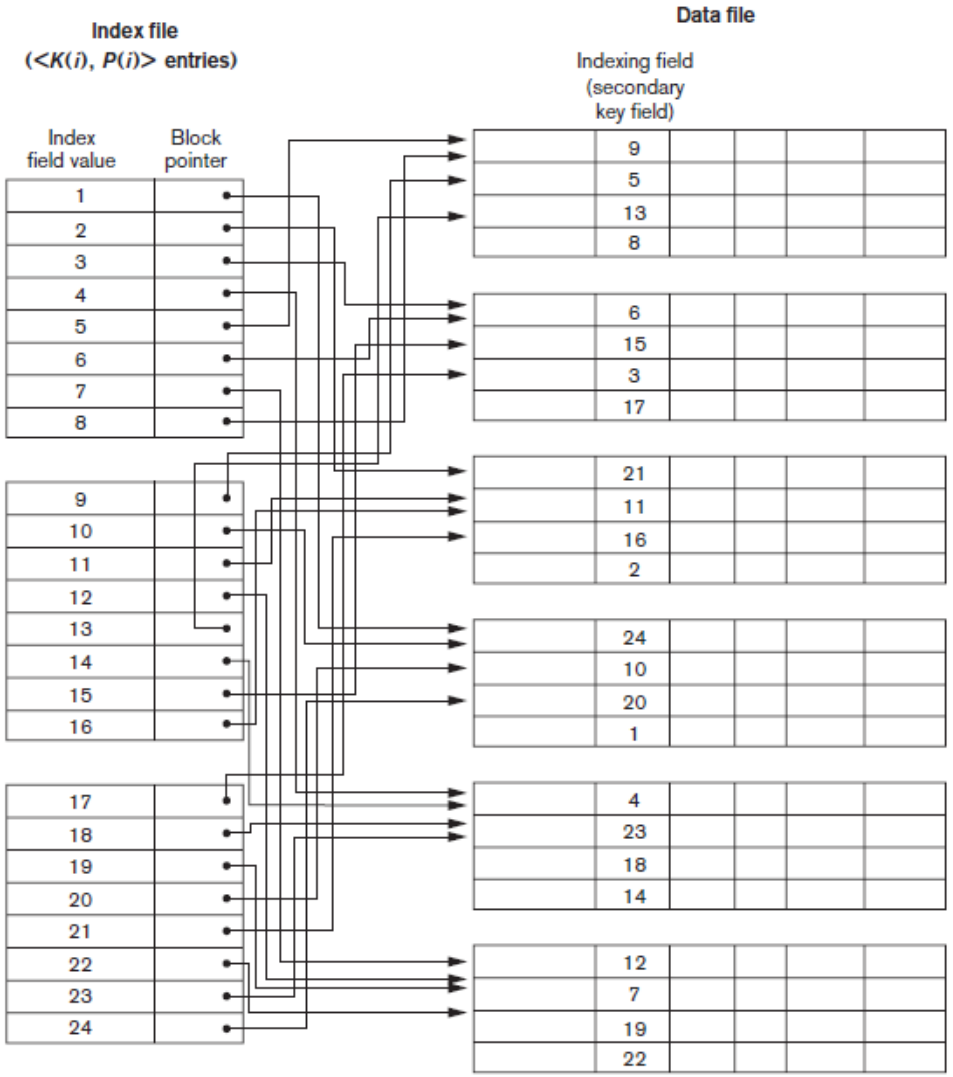


Figure 17.4 Dense secondary index (with block pointers) on a nonordering key field of a file.

Types of Single-Level Ordered Indexes (cont'd.)

	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 17.1 Types of indexes based on the properties of the indexing field

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

[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.

Table 17.2 Properties of index types

17.2 Multilevel Indexes

- Designed to greatly reduce remaining search space as search is conducted
- Index file
  - Considered first (or base level) of a multilevel index
- Second level
  - Primary index to the first level
- Third level
  - Primary index to the second level

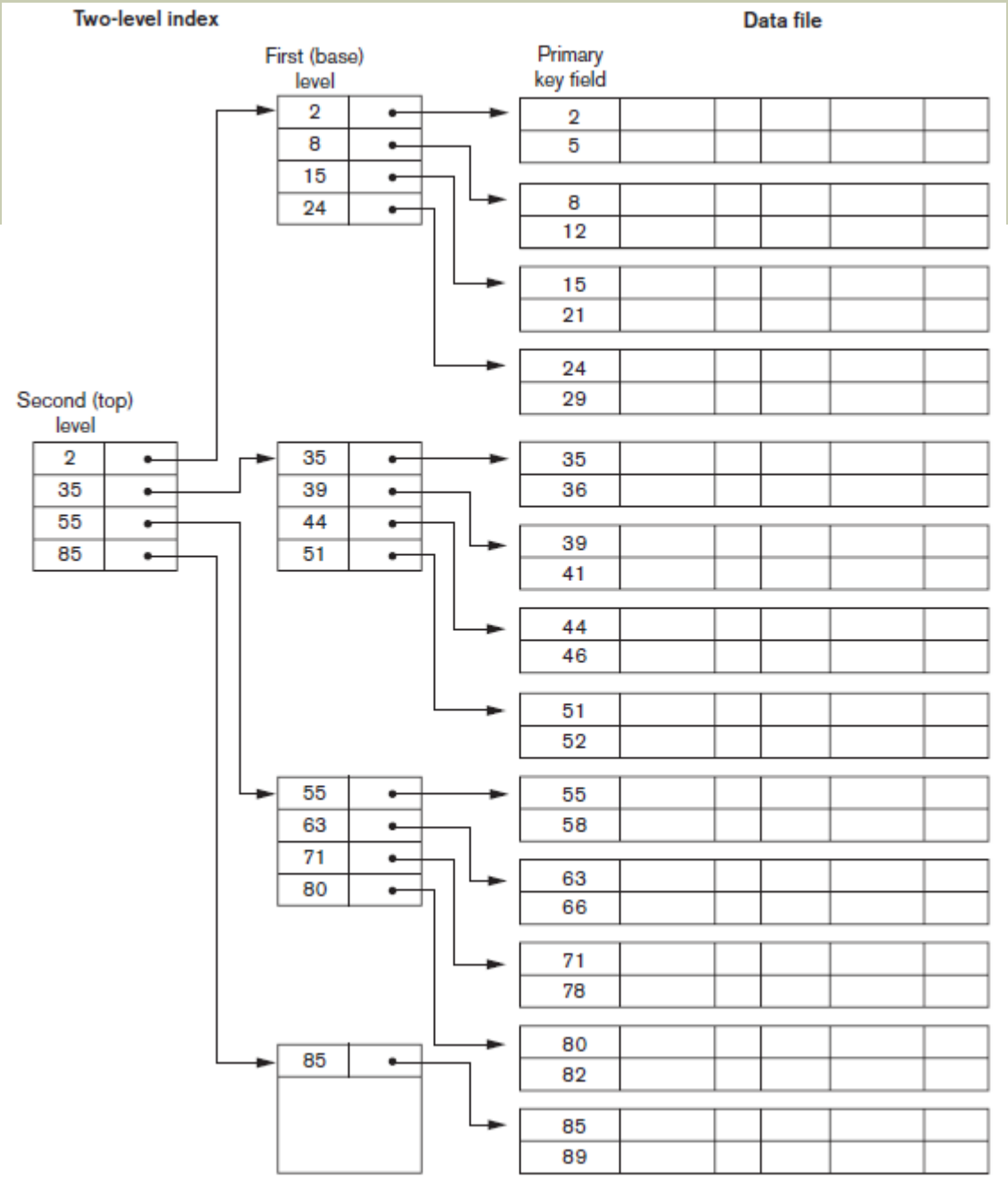


Figure 17.6 A two-level primary index resembling ISAM (indexed sequential access method) organization

17.3 Dynamic Multilevel Indexes Using B-Trees and B+ -Trees

- Tree data structure terminology
  - Tree is formed of nodes
  - Each node (except root) has one parent and zero or more child nodes
  - Leaf node has no child nodes
    - Unbalanced if leaf nodes occur at different levels
  - Nonleaf node called internal node
  - Subtree of node consists of node and all descendant nodes

Tree Data Structure

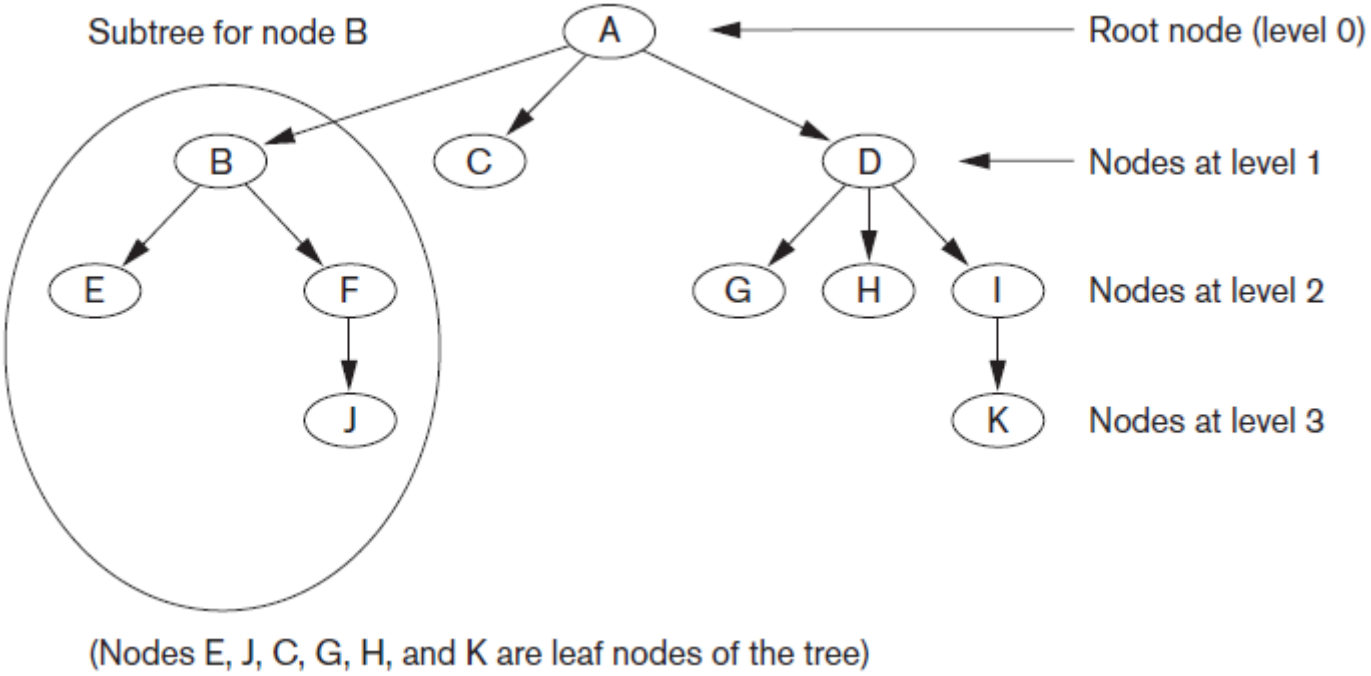
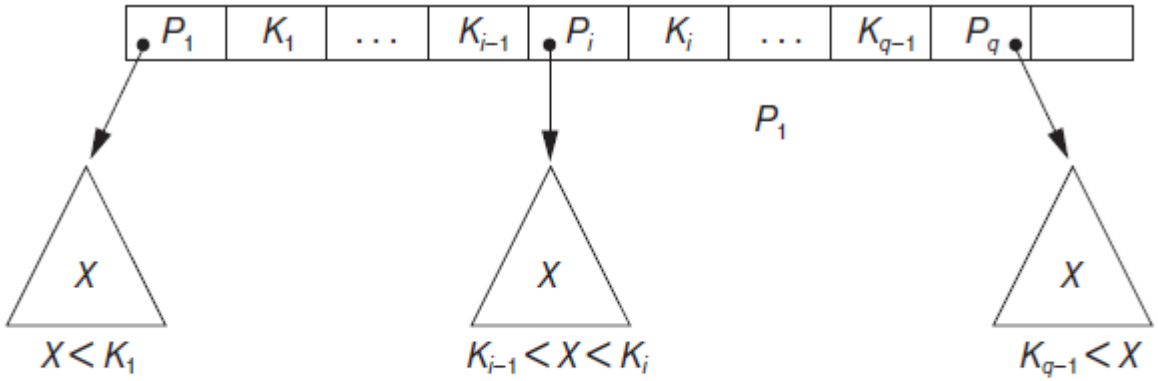


Figure 17.7 A tree data structure that shows an unbalanced tree

Search Trees and B-Trees

- Search tree used to guide search for a record
  - Given value of one of record's fields



[8]This restriction can be relaxed. If the index is on a nonkey field, duplicate search values may exist and the node structure and the navigation rules for the tree may be modified.

Figure 17.8 A node in a search tree with pointers to subtrees below it

Search Trees and B-Trees (cont'd.)

- Algorithms necessary for inserting and deleting search values into and from the tree

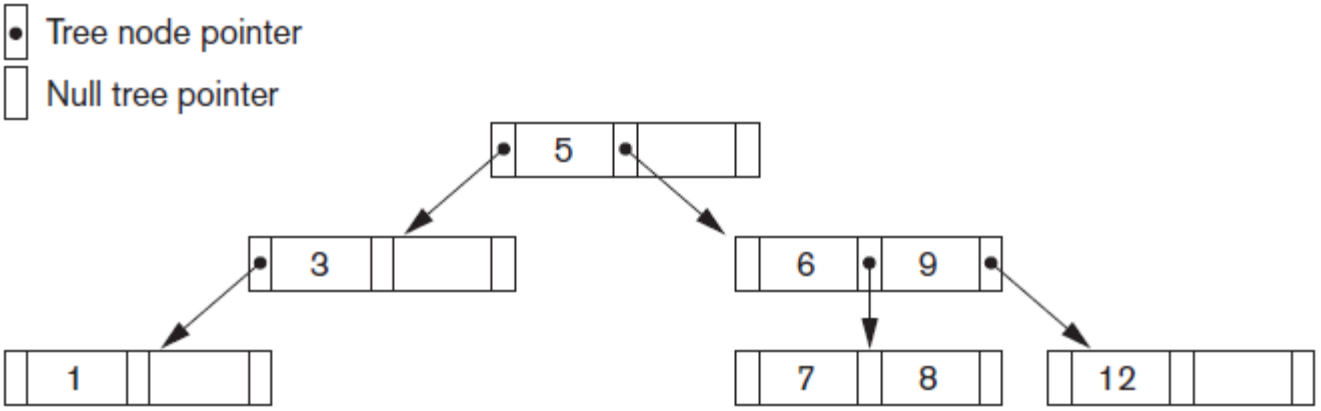


Figure 17.9 A search tree of order p = 3

B-Trees

- Provide multi-level access structure
- Tree is always balanced
- Space wasted by deletion never becomes excessive
  - Each node is at least half-full
- Each node in a B-tree of order *p* can have at most *p-1* search values

B-Tree Structures

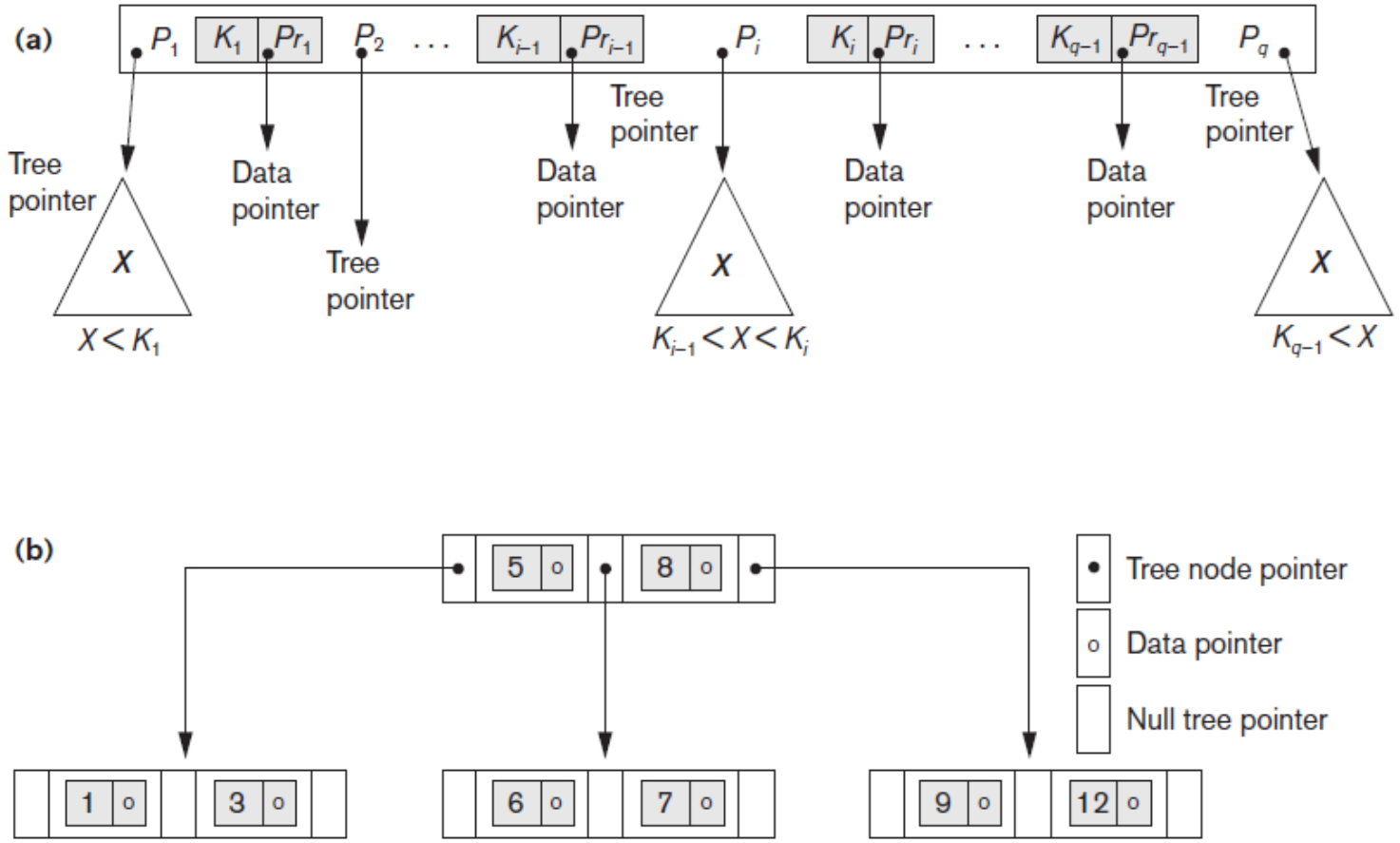


Figure 17.10 B-tree structures (a) A node in a B-tree with *q−1* search values (b) A B-tree of order *p=3.* The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6

B+ -Trees

- Data pointers stored only at the leaf nodes
  - Leaf nodes have an entry for every value of the search field, and a data pointer to the record if search field is a key field
  - For a nonkey search field, the pointer points to a block containing pointers to the data file records
- Internal nodes
  - Some search field values from the leaf nodes repeated to guide search

B+ -Trees (cont'd.)

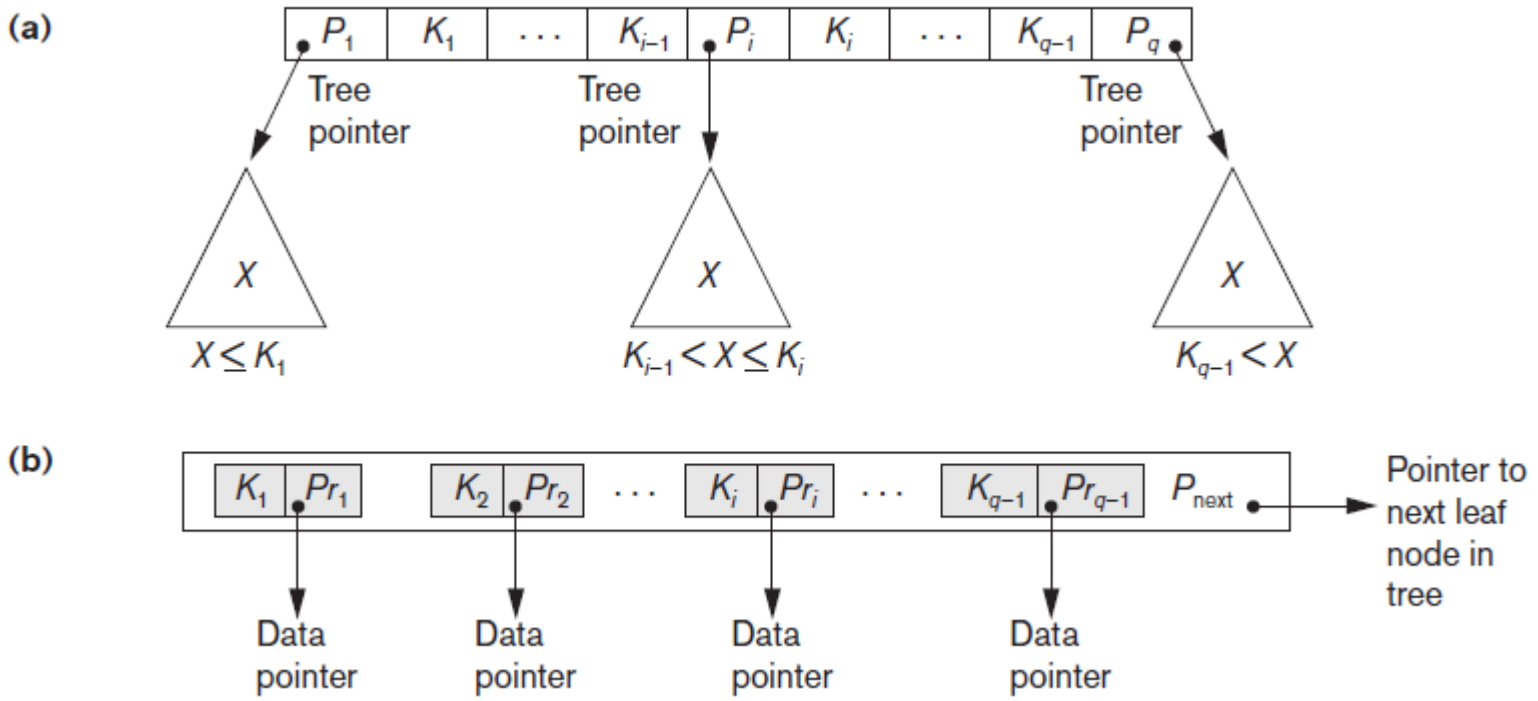


Figure 17.11 The nodes of a B+-tree (a) Internal node of a B+-tree with *q−1* search values (b) Leaf node of a B+-tree with *q−1* search values and *q−1* data pointers

Searching for a Record With Search Key Field Value K, Using a B+ -Tree

$n \leftarrow$ block containing root node of B^+-tree;
read block n;
while (n is not a leaf node of the B^+-tree) do
    begin
    $q \leftarrow$ number of tree pointers in node n;
    if $K \leq n.K_1$ (*$n.K_i$ refers to the ith search field value in node n*)
        then $n \leftarrow n.P_1$ (*$n.P_i$ refers to the ith tree pointer in node n*)
        else if $K > n.K_{q-1}$
            then $n \leftarrow n.P_q$
            else **begin**
                search node n for an entry i such that $n.K_{i-1} < K \leq n.K_i$;
                $n \leftarrow n.P_i$
                end;
    read block n
    end;
search block n for entry (K_i, Pr_i) with $K = K_i$; (* search leaf node *)
if found
    then read data file block with address Pr_i and retrieve record
    else the record with search field value K is not in the data file;

Algorithm 17.2 Searching for a record with search key field value K, using a B+ -Tree

17.4 Indexes on Multiple Keys

- Multiple attributes involved in many retrieval and update requests
- Composite keys
  - Access structure using key value that combines attributes eg: (Dno, Age)
  - Lexicographic ordering on tuples
  - <3,n>……. <4,m>
  - Eg :<2,40> <2,45>,<3,5>

- Partitioned hashing

For example, consider the composite search key <DNO, AGE>. If DNO and AGE are hashed into a 3-bit and 5-bit address respectively, we get an 8-bit bucket address. Suppose that DNO = 4 has a hash address "100" and AGE = 59 has hash address "10101". Then to search for the combined search value, DNO = 4 and AGE = 59, one goes to bucket address 100 10101; just to search for all employees with AGE = 59, all buckets (eight of them) will be searched whose addresses are "000 10101", "001 10101", . . . etc. An advantage of partitioned hashing is that it can be easily extended to any number of attributes. The bucket addresses can be designed so that high order bits in the addresses correspond to more frequently accessed attributes. Additionally, no separate access structure needs to be maintained for the individual attributes. The main drawback of partitioned hashing is that it cannot handle range queries on any of the component attributes.

For example, consider the composite search key <DNO, AGE>. If DNO and AGE are hashed into a 3-bit and 5-bit address respectively, we get an 8-bit bucket address. Suppose that DNO = 4 has a hash address "100" and AGE = 59 has hash address "10101". Then to search for the combined search value, DNO = 4 and AGE = 59, one goes to bucket address 100 10101; just to search for all employees with AGE = 59, all buckets (eight of them) will be searched whose addresses are "000 10101", "001 10101", . . . etc. An advantage of

partitioned hashing is that it can be easily extended to any number of attributes. The bucket addresses can be designed so that high order bits in the addresses correspond to more frequently accessed attributes. Additionally, no separate access structure needs to be maintained for the individual attributes. The main drawback of partitioned hashing is that it cannot handle range queries on any of the component attributes.

Indexes on Multiple Keys (cont'd.)

- Grid files
  - Array with one dimension for each search attribute

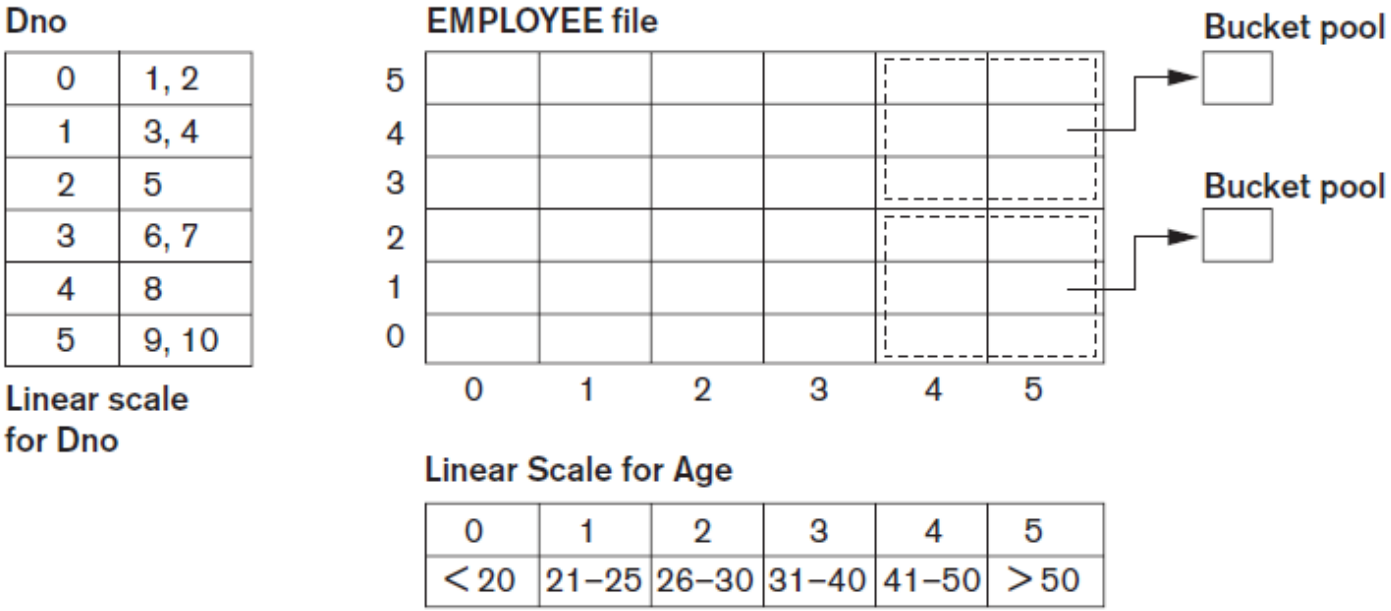


Figure 17.14 Example of a grid array on Dno and Age attributes

Buckets having pointers to employess of age >=41 in deprtment 1-5 and 6 to 7

17.5 Other Types of Indexes

- Hash indexes
  - Secondary structure for file access
  - Uses hashing on a search key other than the one used for the primary data file organization
  - Index entries of form (K, P_r) or (K, P)
    - P_r: pointer to the record containing the key
    - *P:* pointer to the block containing the record for that key

Hash Indexes (cont'd.)

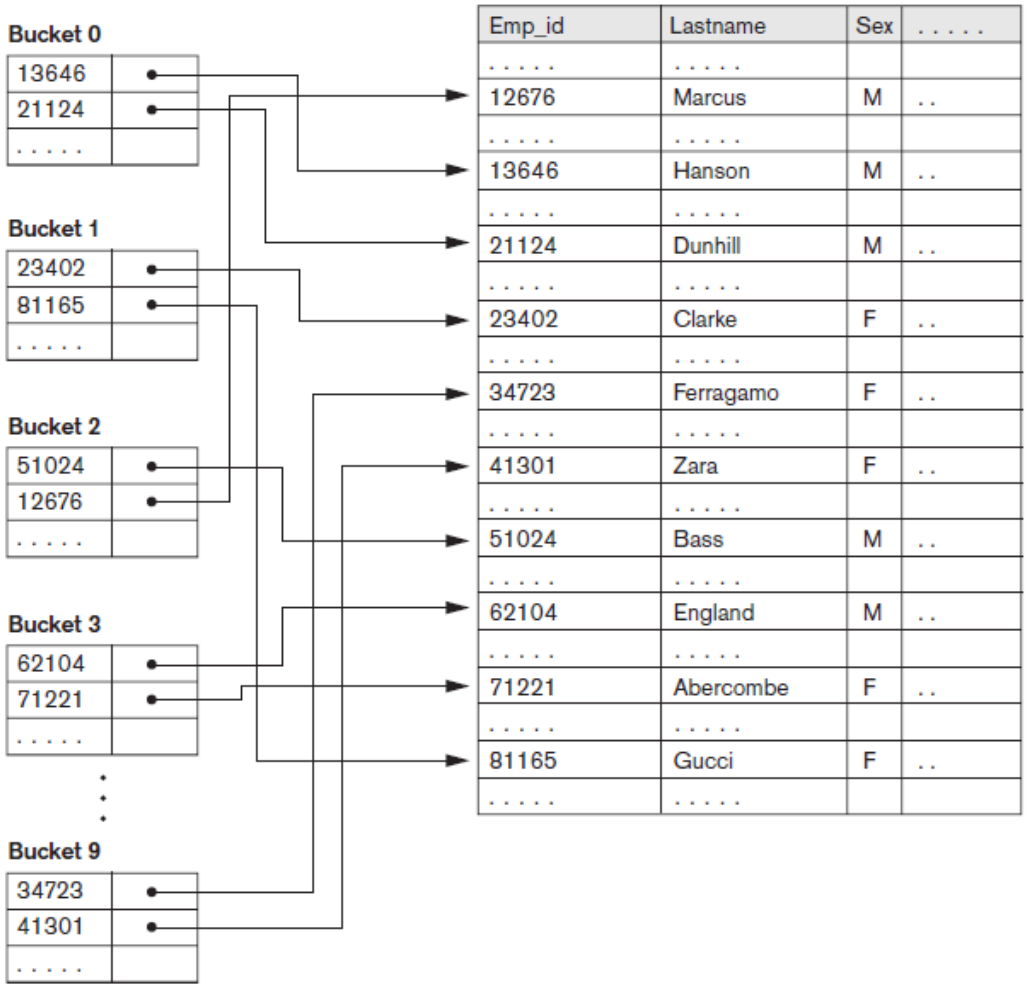


Figure 17.15 Hash-based indexing

Bitmap Indexes

- Used with a large number of rows
- Creates an index for one or more columns
  - Each value or value range in the column is indexed
- Built on one particular value of a particular field
  - Array of bits
- Existence bitmap
- Bitmaps for B+ -tree leaf nodes

Bitmap Indices

- Bitmap indices are a special type of index designed for efficient querying on multiple keys
- Very effective on attributes that take on a relatively small number of *distinct values*
  - E.g. gender, country, state, …
  - E.g. income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000- infinity)
- A bitmap is simply an array of bits
  - For each gender, we associate a bitmap, where each bit represents whether or not the corresponding record has that gender.

Bitmap Indices (Cont.)

- In its simplest form a bitmap index on an attribute has a bitmap for each value of the attribute
  - Bitmap has as many bits as records

record number	*name*	*gender*	*address*	*income_level*
0	John	m	Perryridge	L1
1	Diana	f	Brooklyn	L2
2	Mary	f	Jonestown	L1
3	Peter	m	Brooklyn	L4
4	Kathy	f	Perryridge	L3

Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income_level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

Bitmap Indices (Cont.)

- Bitmap indices are useful for queries on multiple attributes
  - not particularly useful for single attribute queries
- Queries are answered using bitmap operations
  - Intersection (and)
  - Union (or)
  - Complementation (not)
- Each operation takes two bitmaps of the same size and applies the operation on corresponding bits to get the result bitmap
  - E.g. 100110 AND 110011 = 100010
    100110 OR 110011 = 110111

Bitmap Indices (Cont.)

- Bitmap indices generally very small compared with relation size
  - E.g. if record is 100 bytes, space for a single bitmap is 1/800 of space used by relation.
    - If number of distinct attribute values is 8, bitmap is only 1% of relation size
- Deletion needs to be handled properly
  - Existence bitmap to note if there is a valid record at a record location
  - Needed for complementation
    - not(*A=v*): *(NOT bitmap-A-v) AND ExistenceBitmap*
- Should keep bitmaps for all values, even null

Index Definition in SQL

- Create a B-tree index (default in most databases)

  create index <index-name> **on** <relation-name>
  (<attribute-list>)

  -- create index *b-index* **on** *branch(branch_name)*

  -- create index *ba-index* **on** *branch(branch_name, account)* *-- concatenated index*

  -- create index *fa-index* **on** *branch(func(balance, amount)) – function index*

- Use **create unique index** to indirectly specify and enforce the condition that the search key is a candidate key.
- Hash indexes: not supported by every database (but implicitly in joins,…)
  - PostgresSQL has it but discourages due to performance
- Create a bitmap index

  create bitmap index <index-name> **on** <relation-name>
  (<attribute-list>)

  - For attributes with few distinct values
  - Mainly for decision-support(query) and not OLTP (do not support updates efficiently)
- To drop any index

  drop index <index-name>

Function-Based Indexing

- Value resulting from applying some function on a field (or fields) becomes the index key
- Introduced in Oracle relational DBMS
- Example
  - Function UPPER(Lname) returns uppercase representation

```
CREATE INDEX upper_ix ON Employee (UPPER(Lname));
```

  - Query

```
SELECT First_name, Lname
FROM Employee
WHERE UPPER(Lname)= "SMITH".
```

17.6 Some General Issues Concerning Indexing

- Physical index
  - Pointer specifies physical record address
  - Disadvantage: pointer must be changed if record is moved
- Logical index
  - Used when physical record addresses expected to change frequently
  - Entries of the form (K, K_p)

Index Creation

- General form of the command to create an index

```
CREATE [ UNIQUE ] INDEX <index name>
ON <table name> ( <column name> [ <order> ] { , <column name> [ <order> ] } )
[ CLUSTER ] ;
```

  - Unique and cluster keywords optional
  - Order can be ASC or DESC
- Secondary indexes can be created for any primary record organization
  - Complements other primary access methods

Indexing of Strings

- Strings can be variable length
- Strings may be too long, limiting the fan-out
- Prefix compression
  - Stores only the prefix of the search key adequate to distinguish the keys that are being separated and directed to the subtree

Tuning Indexes

- Tuning goals
  - Dynamically evaluate requirements
  - Reorganize indexes to yield best performance
- Reasons for revising initial index choice
  - Certain queries may take too long to run due to lack of an index
  - Certain indexes may not get utilized
  - Certain indexes may undergo too much updating if based on an attribute that undergoes frequent changes

Additional Issues Related to Storage of Relations and Indexes

- Enforcing a key constraint on an attribute
  - Reject insertion if new record has same key attribute as existing record
- Duplicates occur if index is created on a nonkey field
- Fully inverted file
  - Has secondary index on every field
- Indexing hints in queries
  - Suggestions used to expedite query execution

Additional Issues Related to Storage of Relations and Indexes (cont'd.)

- Column-based storage of relations
  - Alternative to traditional way of storing relations by row
  - Offers advantages for read-only queries
  - Offers additional freedom in index creation

17.7 Physical Database Design in Relational Databases

- Physical design goals
  - Create appropriate structure for data in storage
  - Guarantee good performance
- Must know job mix for particular set of database system applications
- Analyzing the database queries and transactions
  - Information about each retrieval query
  - Information about each update transaction

Physical Database Design in Relational Databases (cont'd.)

- Analyzing the expected frequency of invocation of queries and transactions
  - Expected frequency of using each attribute as a selection or join attribute
  - 80-20 rule: 80 percent of processing accounted for by only 20 percent of queries and transactions
- Analyzing the time constraints of queries and transactions
  - Selection attributes associated with time constraints are candidates for primary access structures

Physical Database Design in Relational Databases (cont'd.)

- Analyzing the expected frequency of update operations
  - Minimize number of access paths for a frequently-updated file
    - Updating the access paths themselves slows down update operations
- Analyzing the uniqueness constraints on attributes
  - Access paths should be specified on all *candidate key* attributes that are either the primary key of a file or unique attributes

Physical Database Design Decisions

- Design decisions about indexing
  - Whether to index an attribute
    - Attribute is a key or used by a query
  - What attribute(s) to index on
    - Single or multiple
  - Whether to set up a clustered index
    - One per table
  - Whether to use a hash index over a tree index
    - Hash indexes do not support range queries
  - Whether to use dynamic hashing
    - Appropriate for very volatile files

17.8 Summary

- Indexes are access structures that improve efficiency of record retrieval from a data file
- Ordered single-level index types
  - Primary, clustering, and secondary
- Multilevel indexes can be implemented as B-trees and B+ -trees
  - Dynamic structures
- Multiple key access methods
- Logical and physical indexes