STORAGE AND INDEXING

Modified from

Raghu Ramakrishnan and Johannes Gehrke, Database Management Systems, 3rd Edition, McGraw-Hill &

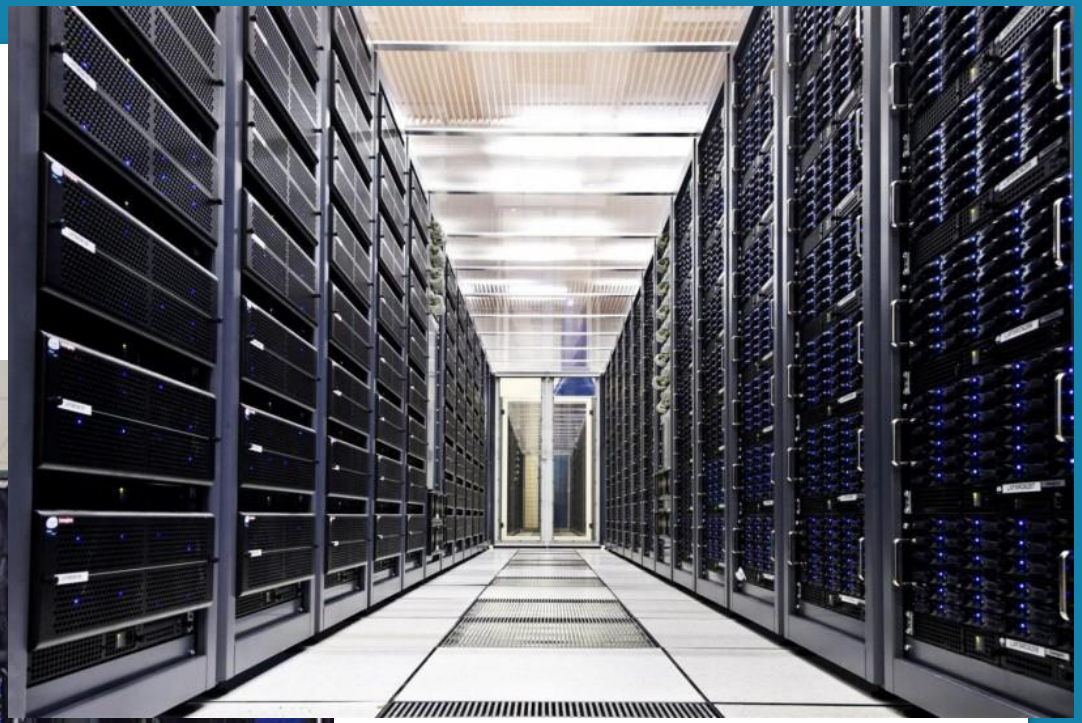
Ramez Elmasri and Shamkant B. Navathe, Fundamentals of Database Systems, 6th Edition, Pearson Education.

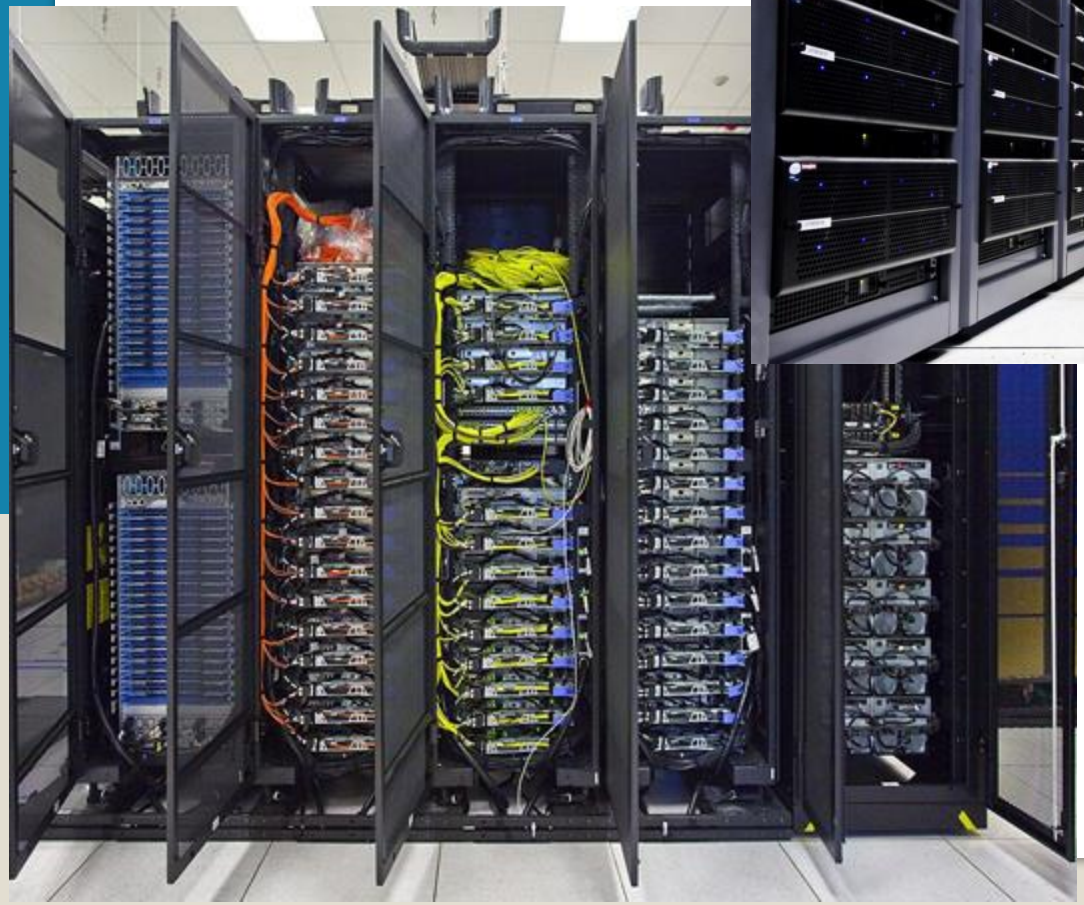
Overview

- How does a DBMS store and access persistent data?
- Why is I/O cost so important for database operation?
- How does a DBMS organize files of data records on disk to minimize I/O costs?
- What is an index, and why is it used?
- What are important properties of indexes?
- How does a hash-based index work, and when is it most effective?
- How does a tree-based index work, and when is it most effective?
- How can we use indexes to optimize performance for a given workload?

EXTERNAL STORAGE

- Access unit: page
  - The unit of information read from or written to disk
    - The size of a page is a DBMS parameter
  - Cost unit: Page I/O
    - Read a page into memory
    - Write a page to disk
  - Cost of database operations
    - Dominated by page I/Os
    - Our purpose: minimize page I/Os





https://en.wikipedia.org/wiki/Database_machine

http://belantic.net/portfolio-item/bp0018-erp-implementation/

EXTERNAL STORAGE

- Disks
  - Random access devices
  - Can retrieve any page at fixed cost
  - Reading consecutive pages is much cheaper than reading them in random order
- Tapes
  - Sequential access devices
  - Can only read pages in sequence
  - Cheaper than disks, used for archival storage

https://www.voltdb.com/files/whitepaper-scaling-memory-data-processing-samsung-advanced-dram-nandssd/

https://www.techradar.com/sg/news/heres-the-cheapest-way-to-store-a-huge-1000tb-of-data-online

FILES AND ACCESS METHODS LAYER

How is a relation stored?

- A relation is stored as a file of records
- Each record has a **record id** (rid) which is used to locate a record (page number)
- Implemented by the component: Files and access methods layer

Files and access methods layer

- Operations supported
  - creation, insertion, deletion, scan, …
- Keeps track of pages allocated to each file
- Tracks available space within pages allocated to the file

FILE ORGANIZATION

How are the records organized?	• File organization
What is it?	• A method of arranging the records in a file when the file is stored on disk • Sorted or unsorted
Target	• Minimize the cost of page I/O (input from disk to main memory and output from memory to disk)

Bette H. McLaughlin
Magnone Brothers
Stanley & Allen Magnone
Alexandra and
Sidney Sheldon
Helena Bessie Wing
"Rounded with a sleep."
The
Pacific Stock Exchange
Inc.
Pacific Bell
Awards Dinner
Sony Pictures
Entertainment
ARCO
Awards Dinner
288
289
290
291
301
302
303
304
314
315
316
317
327
328
329
330
340
341
342
343
353
354
355
356
366
367
368
369

Author

Title

```
F       Naske, Claus-M
905             Alaska, a pictorial history / by Claus-M.
N38     Naske and L.J. Rowinski.        Norfolk, Va. : Donning
1983    Co., c. 1983.
                271 p. : ill. (some col.) ; 31 cm.
```

Imprint: place of publication, publisher, date of publication

Number of Pages

Height

Has Illustrations

Physical description

```
                Includes index.
                Bibliography: p. 267-268.
                ISBN 0898654106 (pbk.)

                1. Alaska--History--Pictorial works.  2.
Alaska--Pictorical works.  I. Rowinski, L.
J. (Ludwig J.), 1929-  II. Title.
```

Subject access points

ALTERNATIVE FILE ORGANIZATIONS

Heap (random order) files

- Suitable when typical access is a file scan retrieving all records.

Sorted Files

- Best if records must be retrieved by some order, or a 'range' of records is needed
- Updates are expensive

Indexes / Indices

- Data structures that organize records via trees or hashes
- Like sorted files, they speed up searches for a subset of records, based on values in certain ("search key") fields
- Updates are much faster than in sorted files

INDEXES -- An intuitive overview

- Consider this **randomly ordered** table of names.
- If our objective is to minimize the number of rows that need to be examined in order to find a randomly chosen name, what search process could we use?
- What is the **average** search time if this process is repeated many times?
- What is the **maximum** search time?

Row Position	Last Name
1	Schluter
2	Mishra
3	Salehian
4	Vu
5	Wah
6	Al Rabeeah
7	Wong
8	Garcia
9	Johnson
10	Flores
11	Fung
12	Spievak
13	Doshi
14	Scruton
15	Winter
16	Beena
17	Ngo
18	Ly
19	Gani
20	Pham
21	Hu
22	Israr

INDEXES -- An intuitive overview (contd.)

- Consider this **alphabetically ordered** table of names.
- If we start at the top and examine one row at a time, how many rows will we need examine before we can fine a randomly selected name?
- What is the **average** search time if this process is repeated many times?
- What is the **maximum** search time?

Row Position	Last Name
6	Al Rabeeah
16	Beena
13	Doshi
10	Flores
11	Fung
19	Gani
8	Garcia
21	Hu
22	Israr
9	Johnson
18	Ly
2	Mishra
17	Ngo
20	Pham
3	Salehian
1	Schluter
14	Scruton
12	Spievak
4	Vu
5	Wah
15	Winter
7	Wong

INDEXES / INDICES

- Indexing mechanisms are used to speed up access to desired data.
  - e.g., author catalog in library
- An *index* on a file speeds up selections on the *search key fields* for the index.
  - **Search Key** - attribute (or set of attributes) is used to look up records in a file.
  - Any subset of the fields of a relation can be the search key for an index on the relation.
  - *Search key* is not the same as *key* (minimal set of fields that uniquely identify a record in a relation).

Basic Concepts

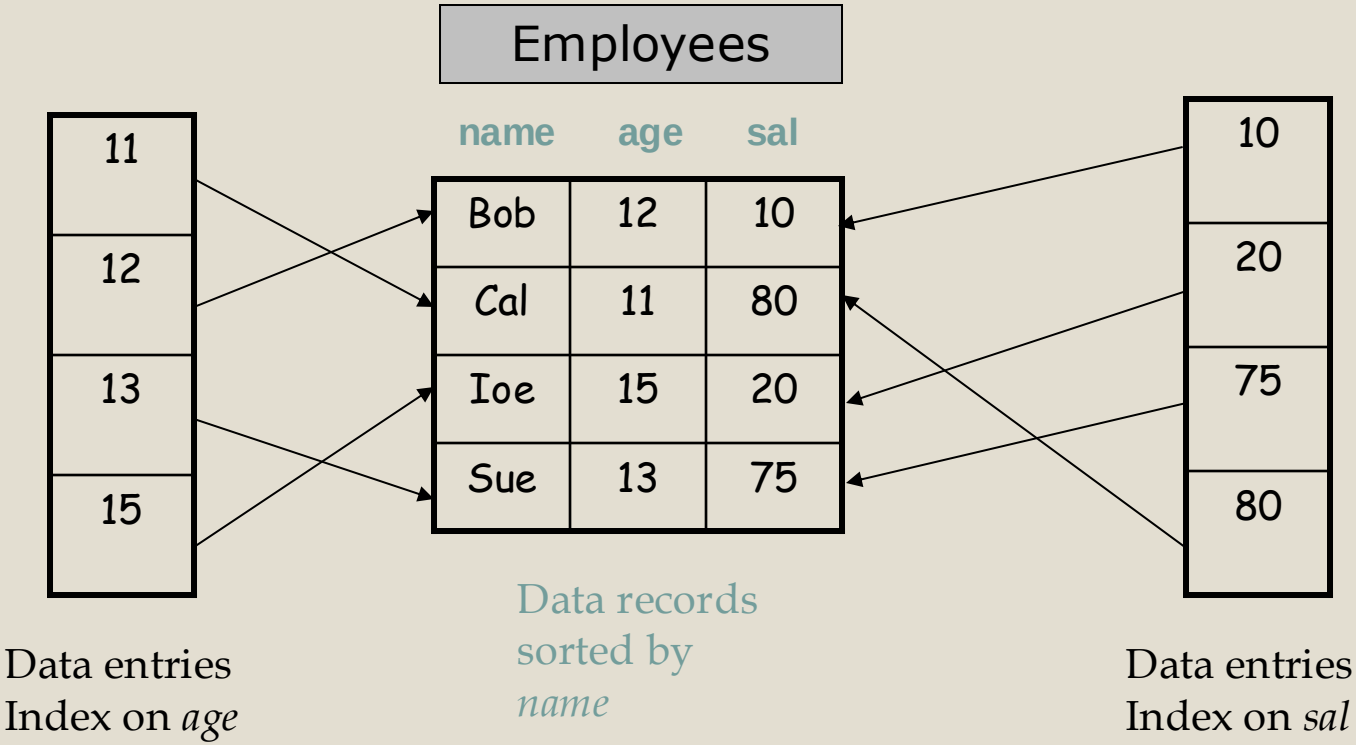
- An **index file** contains a collection of ***data entries* (or *index entries*)**, and supports efficient retrieval of all data entries **k*** with a given key value **k**.

search-key	pointer

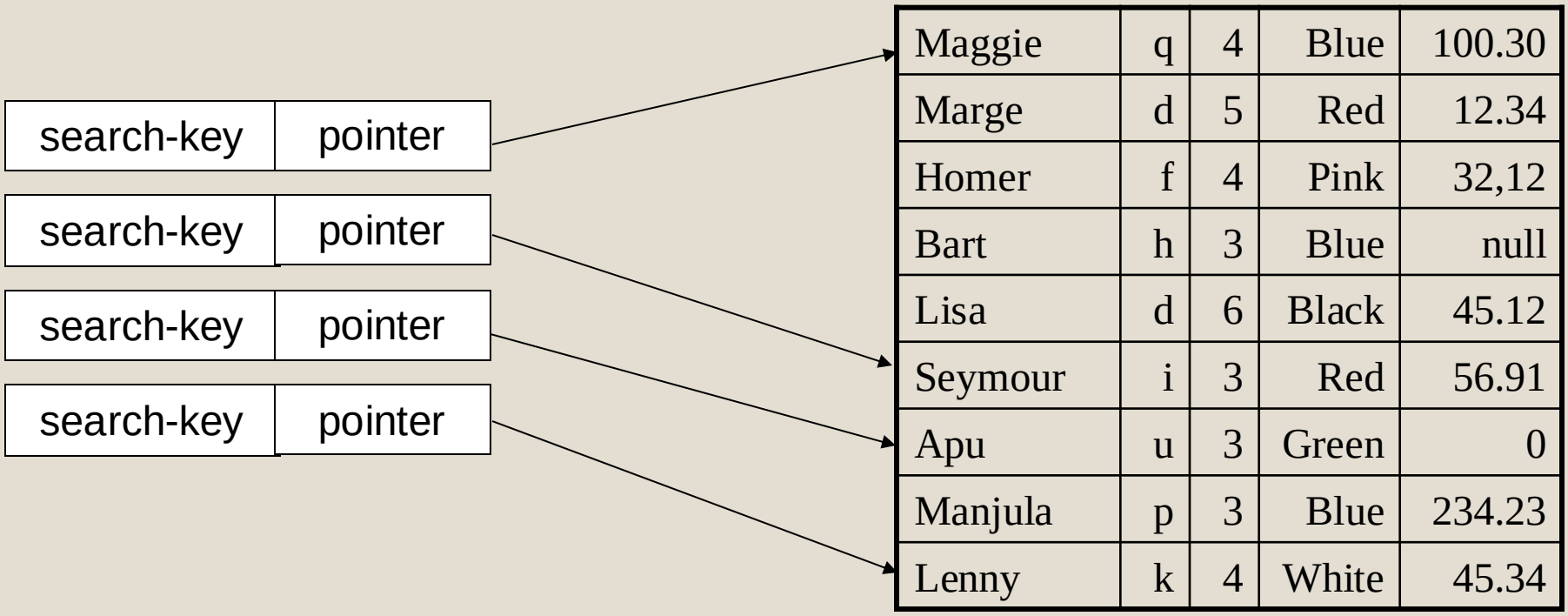
Index files are typically much smaller than the original files

- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order (i.e., tree based)
  - **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function".

DATA ENTRY


Employees
name
age
sal
Bob
12
10
Cal
11
80
Ioe
15
20
Sue
13
75
11
12
13
15
10
20
75
80
Data records sorted by *name*
Data entries Index on *age*
Data entries Index on *sal*

How do indices achieve speedup?



Maggie	q	4	Blue	100.30
Marge	d	5	Red	12.34
Homer	f	4	Pink	32,12
Bart	h	3	Blue	null
Lisa	d	6	Black	45.12
Seymour	i	3	Red	56.91
Apu	u	3	Green	0
Manjula	p	3	Blue	234.23
Lenny	k	4	White	45.34

1) The index is typically much smaller than a record
2) A data entry may point to several records

ALTERNATIVES FOR DATA ENTRIES

Alternative 1

A data entry k* is an actual data record (with search key value k)

- At most one index on a given collection of data records can use alternative 1
- If data records are very large, #of pages containing data entries is high.
- It's a special file organization, called indexed file organization.

ALTERNATIVES FOR DATA ENTRIES (CONTD.)

Alternative 2

a data entry is a <k, rid> pair

- rid is the record id of a data record with search key value k

Alternative 3

a data entry is <k, rid-list>

- Rid-list is a list of record ids of data records with search key value k

ALTERNATIVES FOR DATA ENTRIES (CONTD.)

Advantages of Alternative 2 & 3

- Contain data entries that point to data records
- Are independent of the file organization that is used for the index file
- Alternative 3 offers better space utilization than Alternative 2.

Contents

INDEX CLASSIFICATION

Primary vs. *Secondary*

- If search key contains primary key, then called primary index.
  Other indexes are called secondary indexes.
- unique index: Search key contains a candidate key.

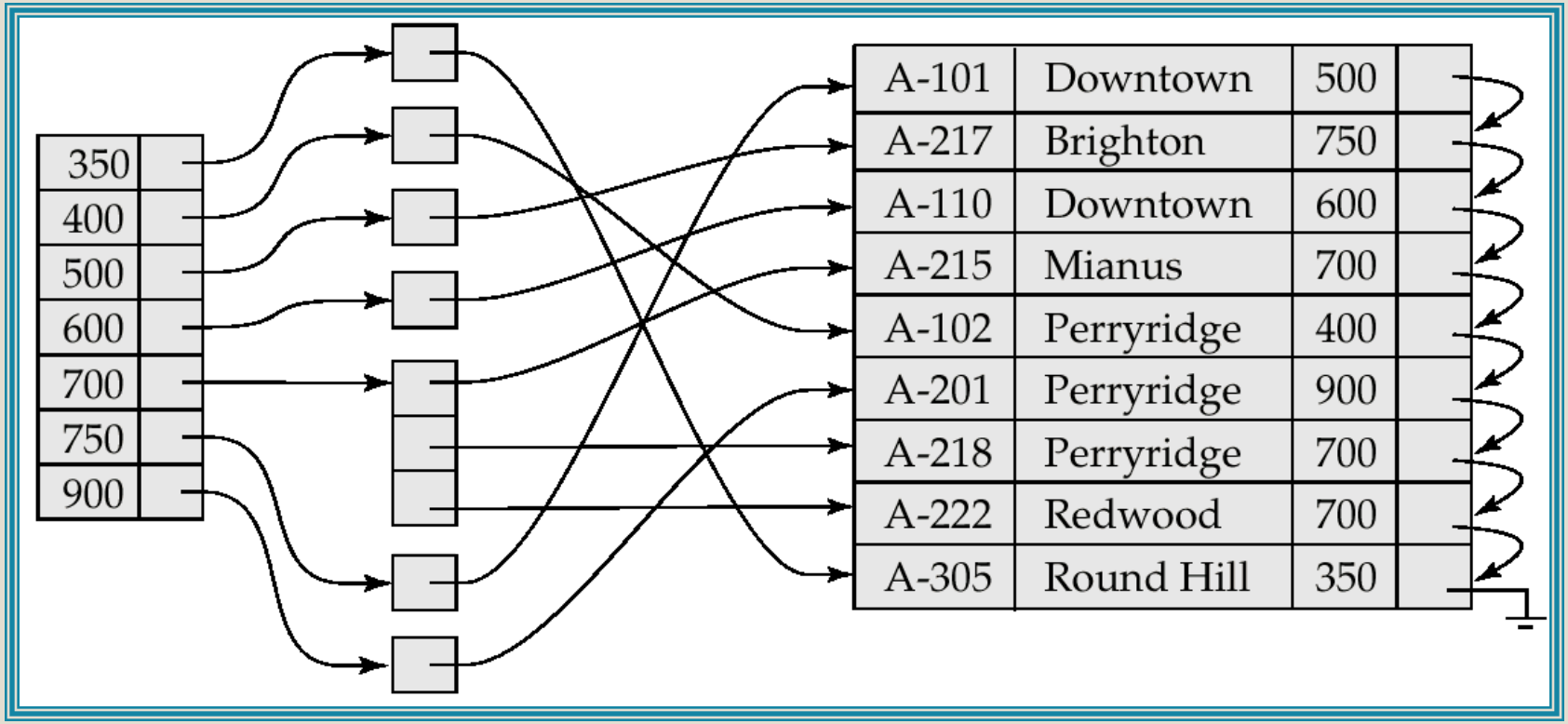
Clustered vs. *Unclustered*

- If order of data records is the same as, or 'close to' order of data entries, then called **clustered index**.
- Alternative 1 implies clustered; in practice, clustered also implies Alternative 1 (since sorted files are rare).
- A file can be clustered on **at most one** search key.
- Cost of retrieving data records through index varies greatly based on whether index is clustered or not!

Primary and Secondary Indices

- Secondary indices must be dense.
- Indices offer substantial benefits when searching for records.
- When a file is modified, every index on the file must be updated. Updating indices imposes overhead on database modification.
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive (probably worse than no index at all).
  - each record access may fetch a new block from disk

Secondary Index on *balance* field of ACCOUNT



CLUSTERED vs. UNCLUSTERED INDEX

Suppose that Alternative 2 is used for data entries, and that the data records are stored in a Heap file.

- To build clustered index, first sort the Heap file (with some free space on each page for future inserts).
- Overflow pages may be needed for inserts. (Thus, order of data records is 'close to', but not identical to, the sort order.)

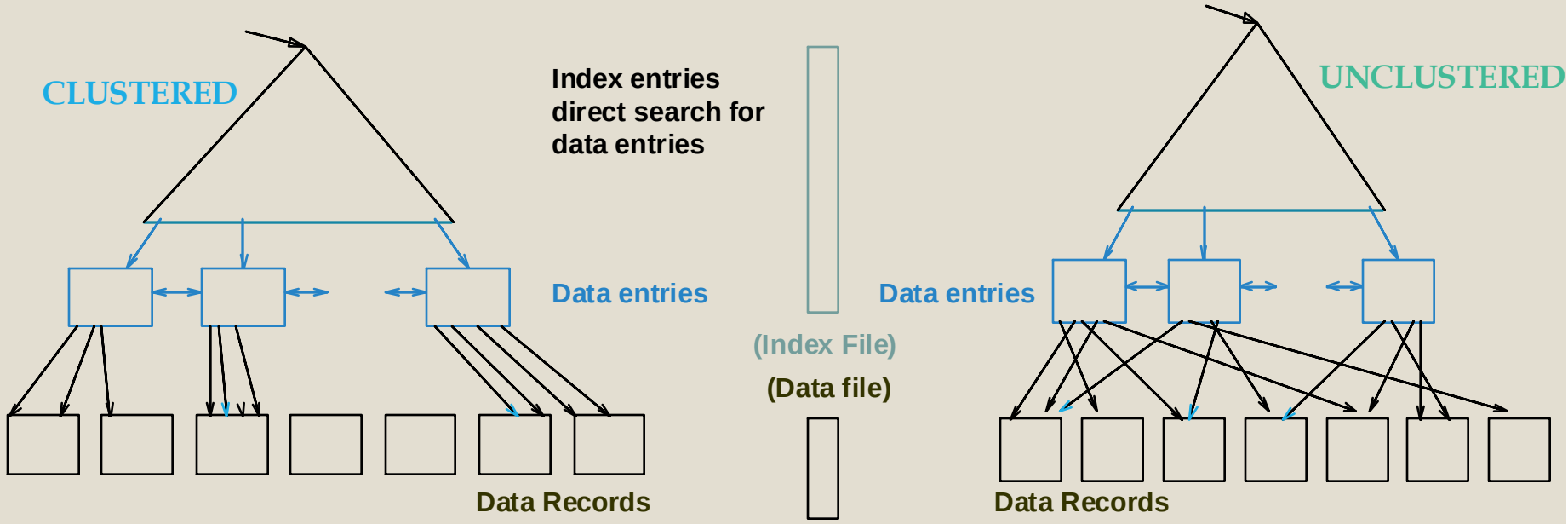
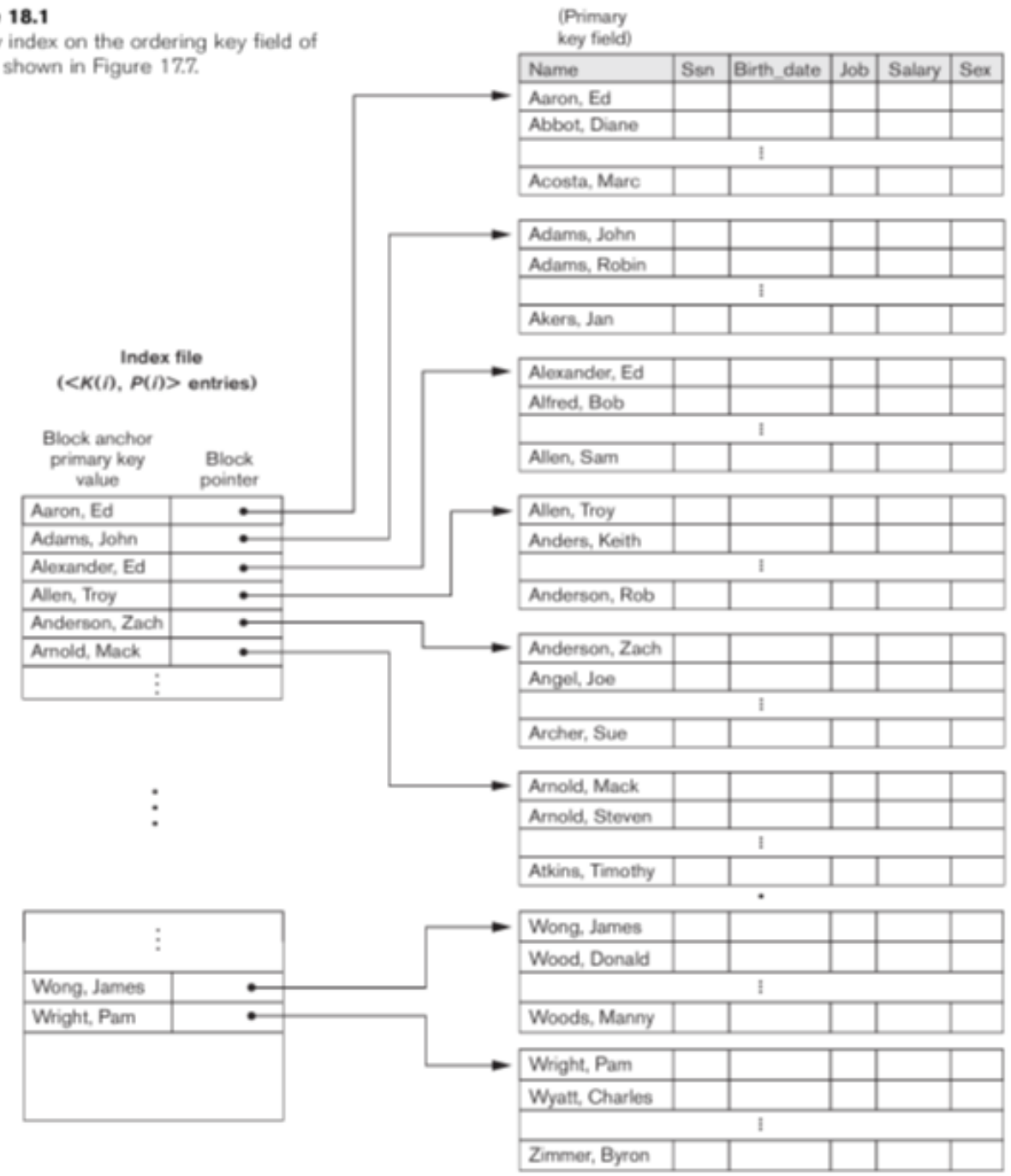


Figure 18.1
Primary index on the ordering key field of the file shown in Figure 17.7.



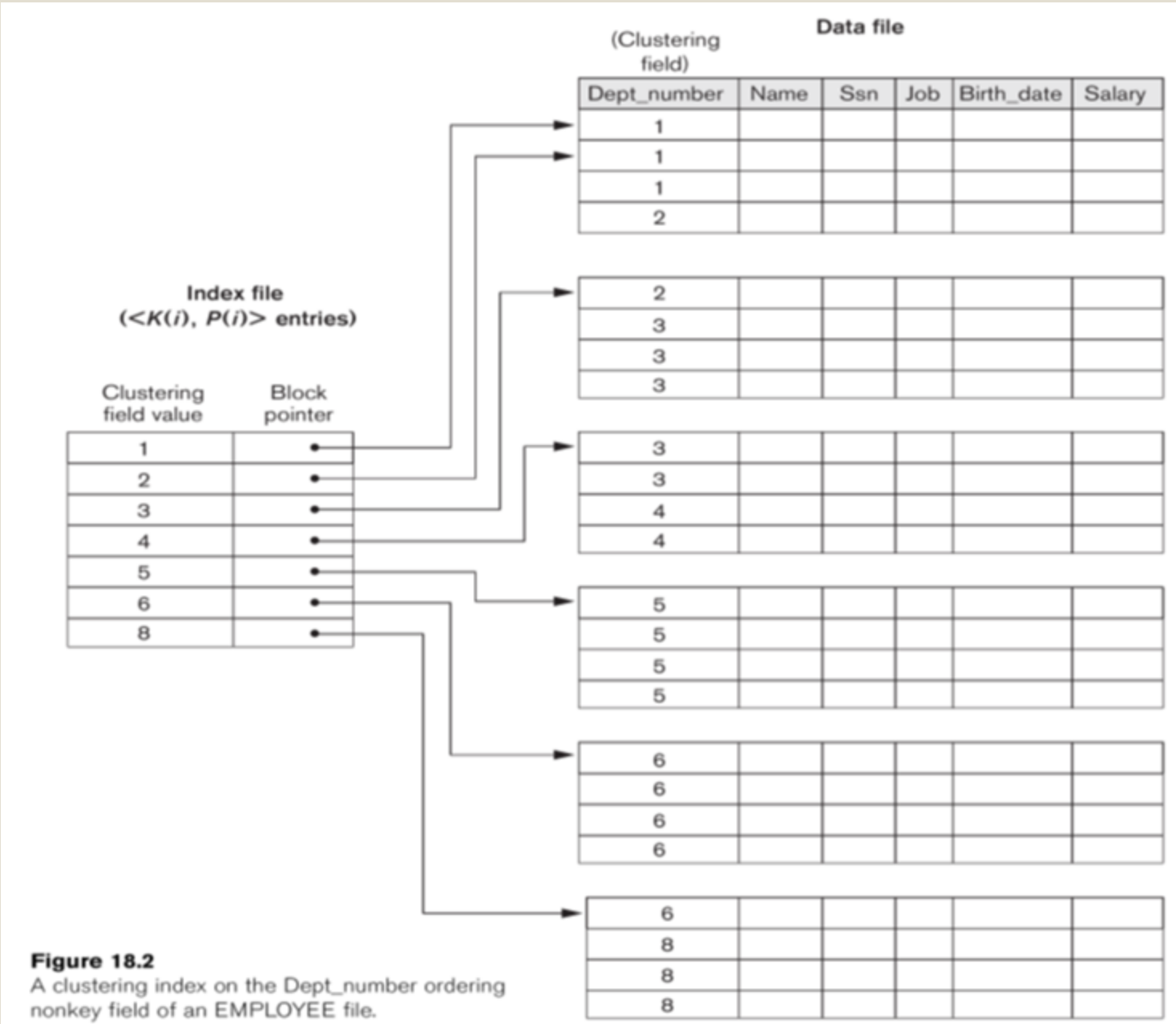
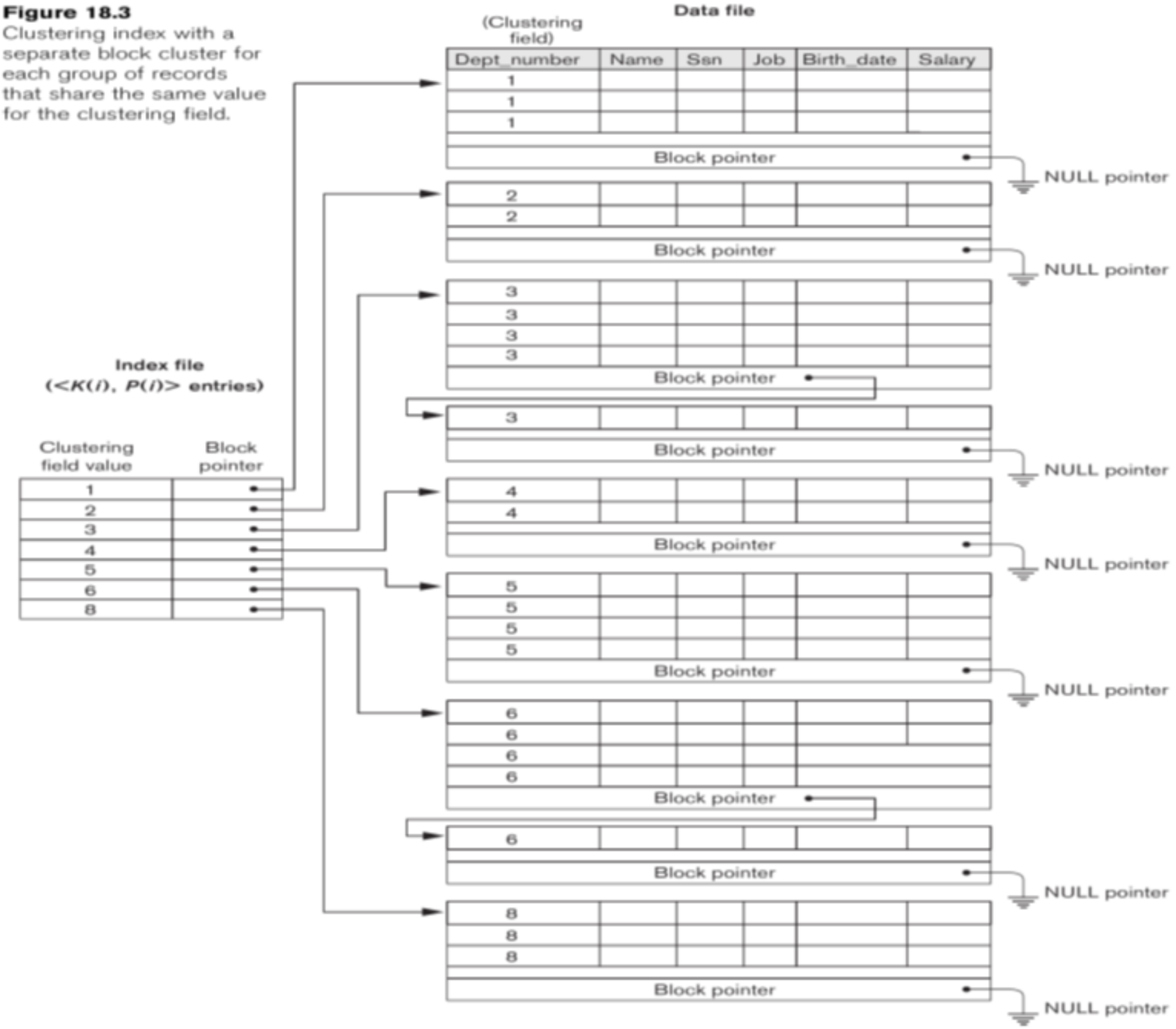


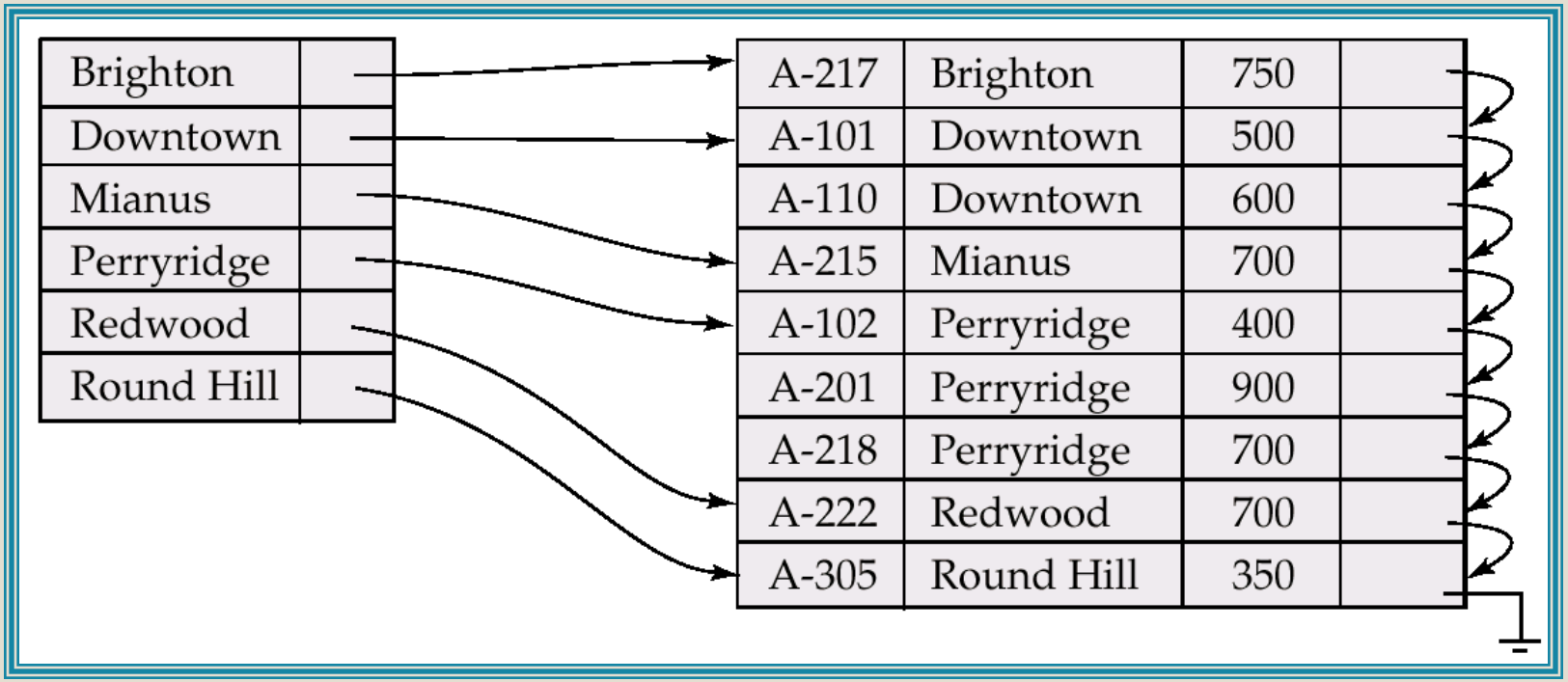
Figure 18.2
A clustering index on the Dept_number ordering nonkey field of an EMPLOYEE file.

Figure 18.3
Clustering index with a separate block cluster for each group of records that share the same value for the clustering field.



Dense Index Files

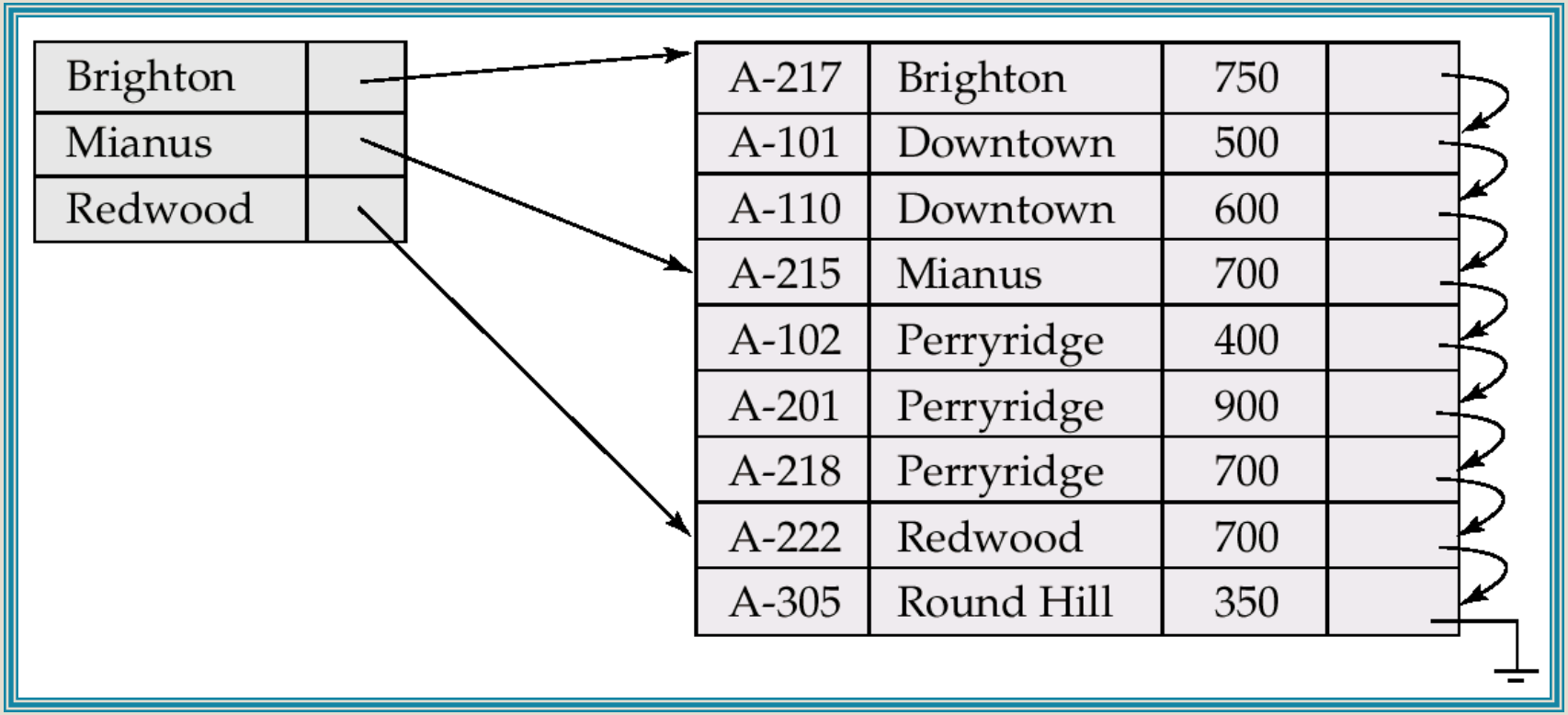
- Dense index — Index record appears for every search-key value in the file.



Sparse Index Files

- Sparse Index: contains index records for only some search-key values.
- Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value K we:
  - Find index record with largest search-key value < K
  - Search file sequentially starting at the record to which the index record points
- Less space and less maintenance overhead for insertions and deletions.
- Generally slower than dense index for locating records.
- Good tradeoff: sparse index with an index entry for every block in file, corresponding to the least search-key value in the block. (because bringing the block into memory is the greatest expense, and index size is still small).

Example of Sparse Index Files



INDEX DATA STRUCTURES

How to organize data entries in an index?

HASH INDEX

B+ TREE INDEX

HASH-BASED INDEXES

Hash data entries on the search key

Index is a collection of buckets.

- A bucket consists of a primary page and overflow pages linked in a chain

How to determine which bucket a record belongs to?

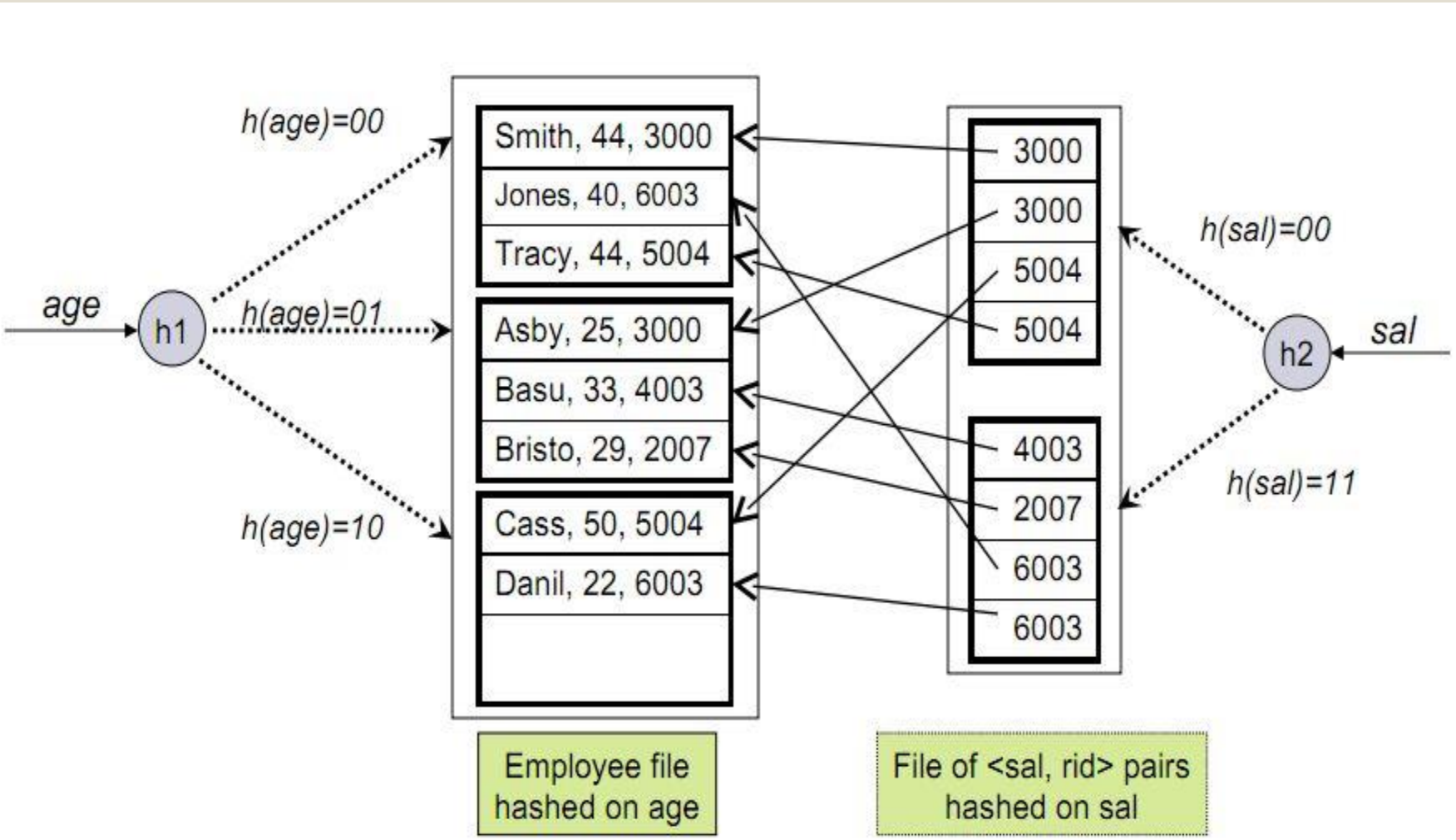
- Hashing function h is applied to the search key
- *Hashing function* **h**: **h**(*r*) = bucket in which record *r* belongs. **h** looks at the *search key* fields of *r.*

Good for equality selections

How to handle updates and search?

If Alternative 1 is used, the buckets contain the data records; otherwise, they contain <key, rid> or <key, rid-list> pairs.

INDEX-ORGANIZED FILE



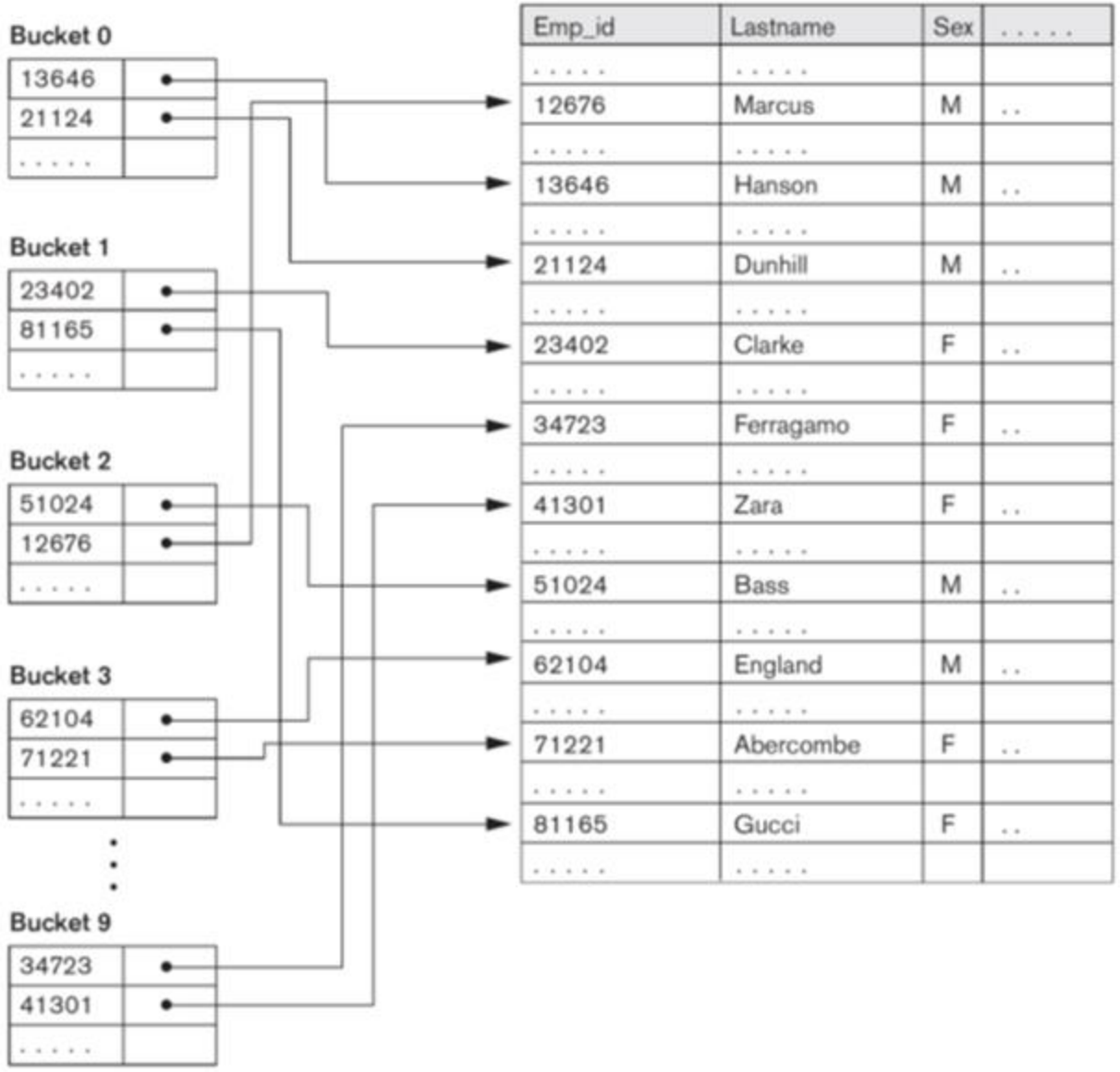
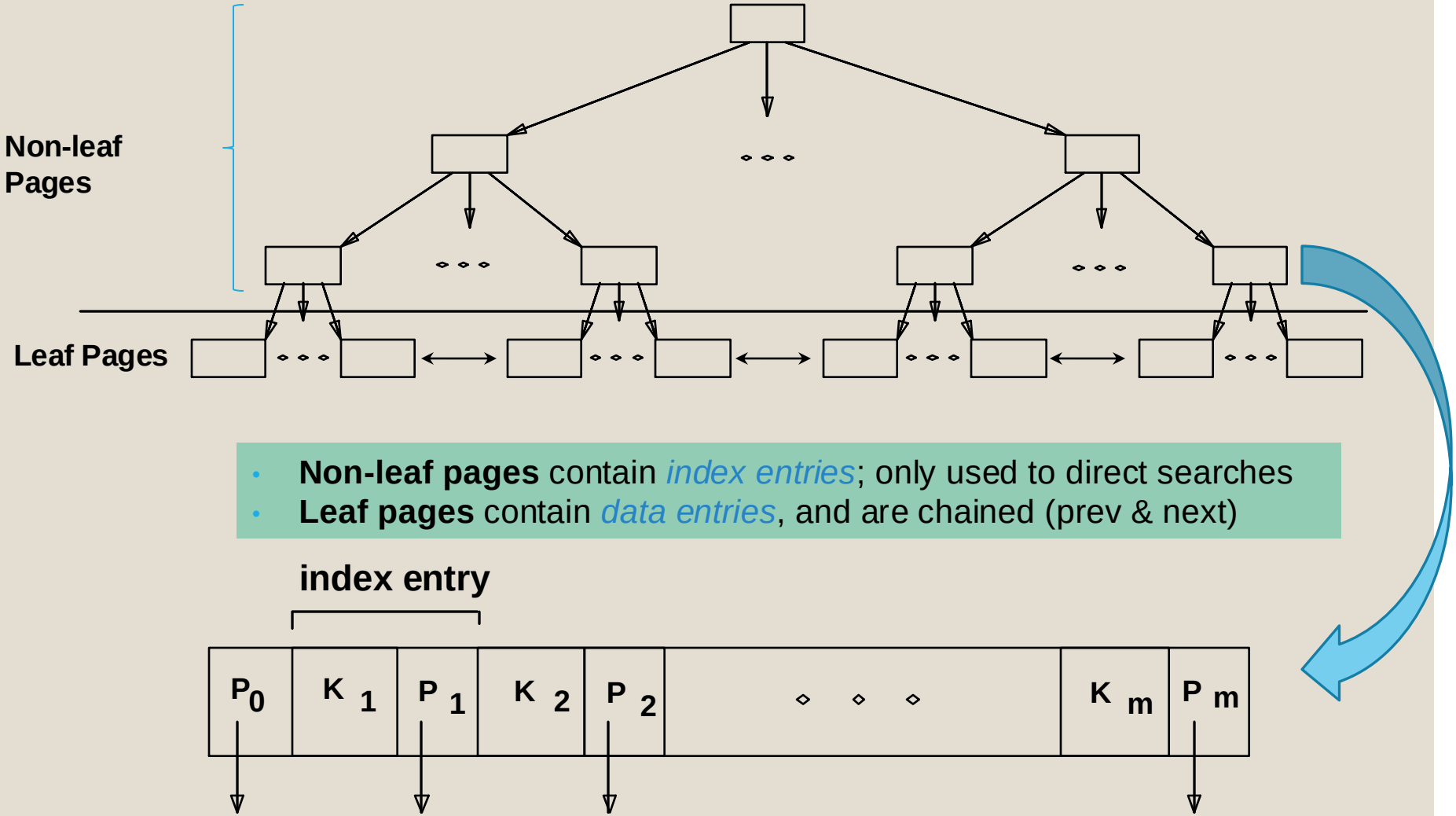


Figure 18.15
Hash-based indexing.

B+ TREE INDEX

- Data entries are arranged in sorted order by search key value
- A hierarchical search data structure is maintained that directs searches to the correct page of data entries
  - Allows to efficiently direct locate all data entries with search key values in a desired range

B+ TREE INDEXES



- **Non-leaf pages** contain *index entries*; only used to direct searches
- **Leaf pages** contain *data entries*, and are chained (prev & next)

B+ TREE INDEXES

Root

- Top-most node is where all searches begin

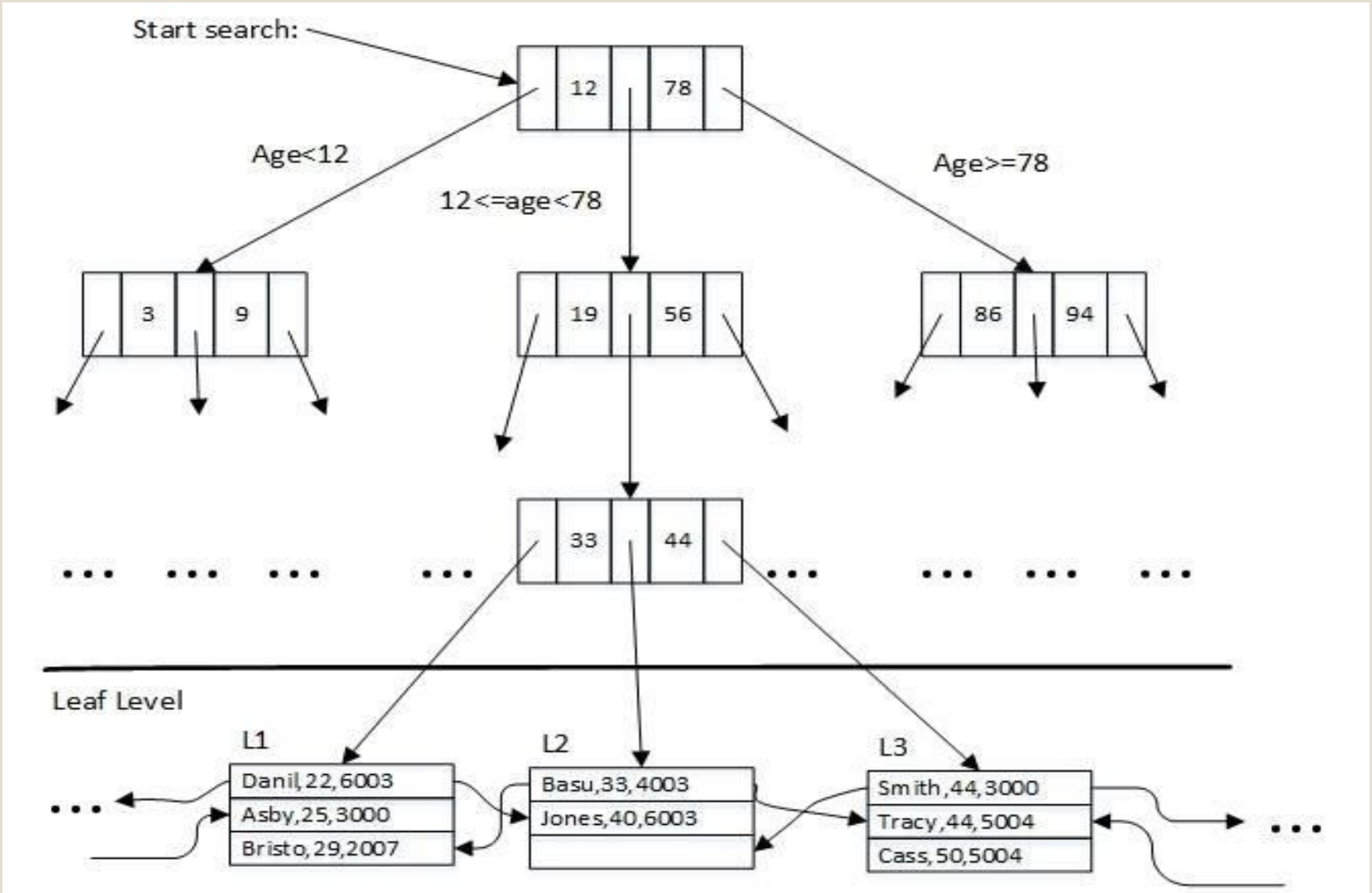
Non-leaf pages

- contain index entries
  - directed searches to the correct leaf page
- Non-leaf pages contain node pointers separated by search key values
- The node pointer to the left of a key value k points to a subtree that contains only data entries less than k
- The node pointer to the right of a key value k points to a subtree that contains only data entries greater than or equal to k

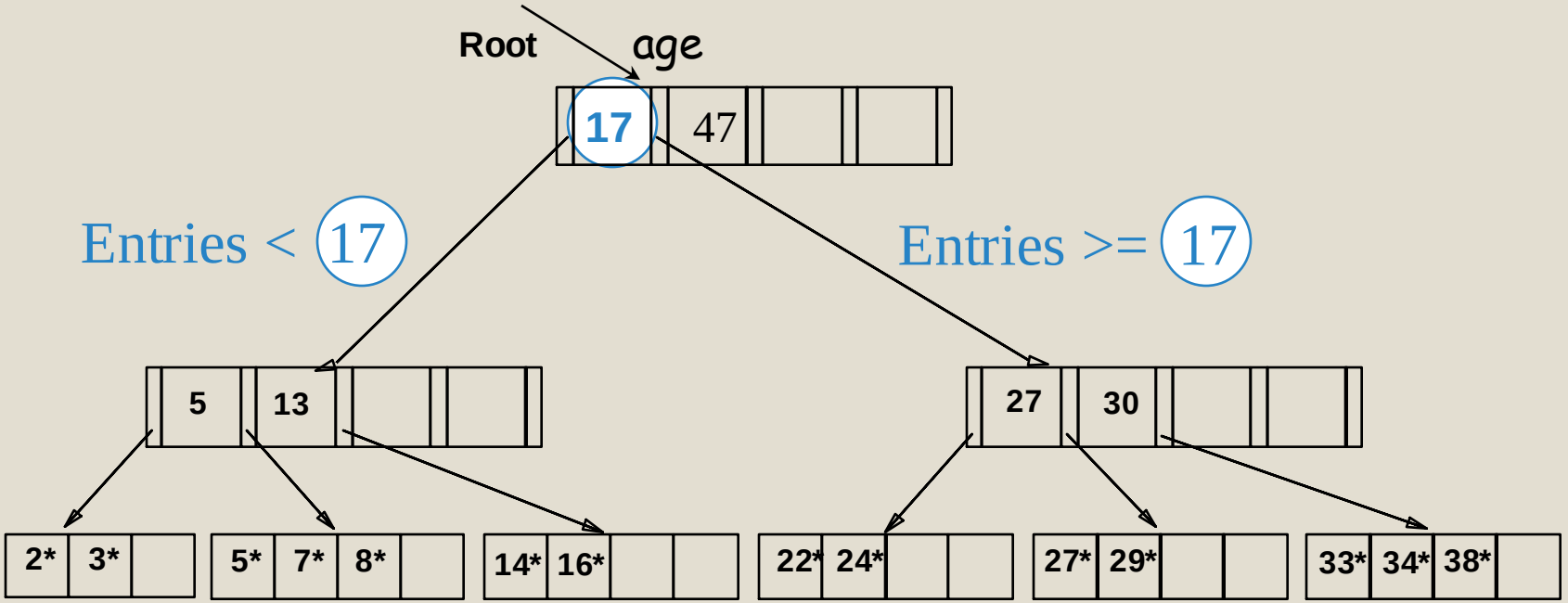
Leaf pages

- contain data entries and are chained

TREE-STRUCTURED INDEX



B+ TREE EXAMPLE



Find 28*, 29*, All > 15* and < 30*

Insert/delete:

- Find data entry in leaf, then change it.
- Need to adjust parent sometimes.
- And change sometimes bubbles up the tree.

MORE ON B+ TREE INDEX

The number of page I/Os of a search is equal to:

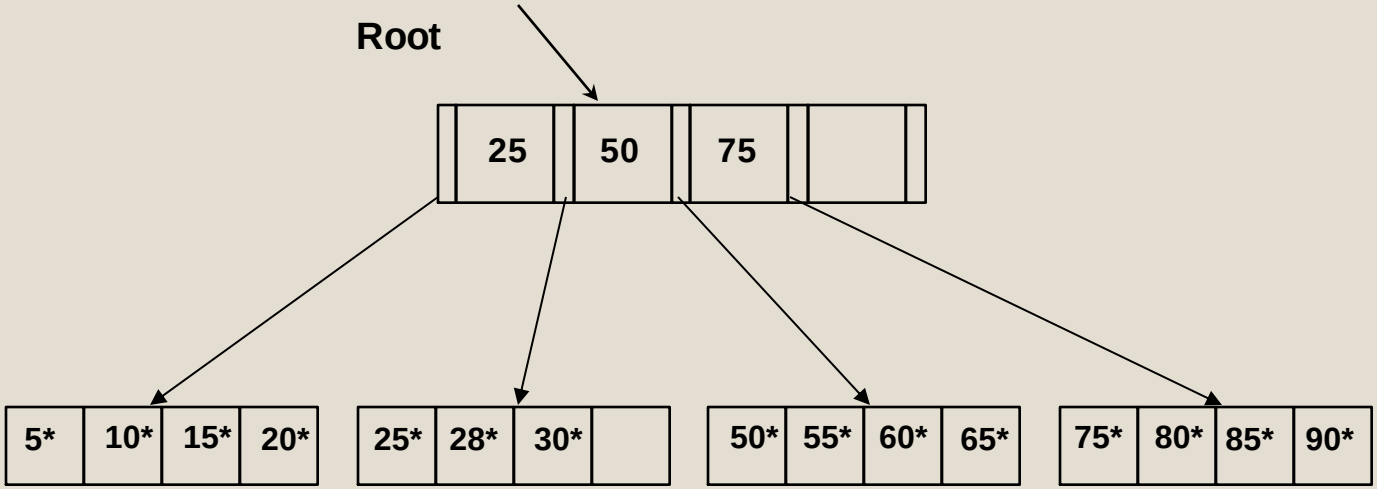
- the length of a path from the root to a leaf plus the number of leaf pages with qualifying data entries

B+ tree ensures that all paths from the root to a leaf in a given tree are of the same length.

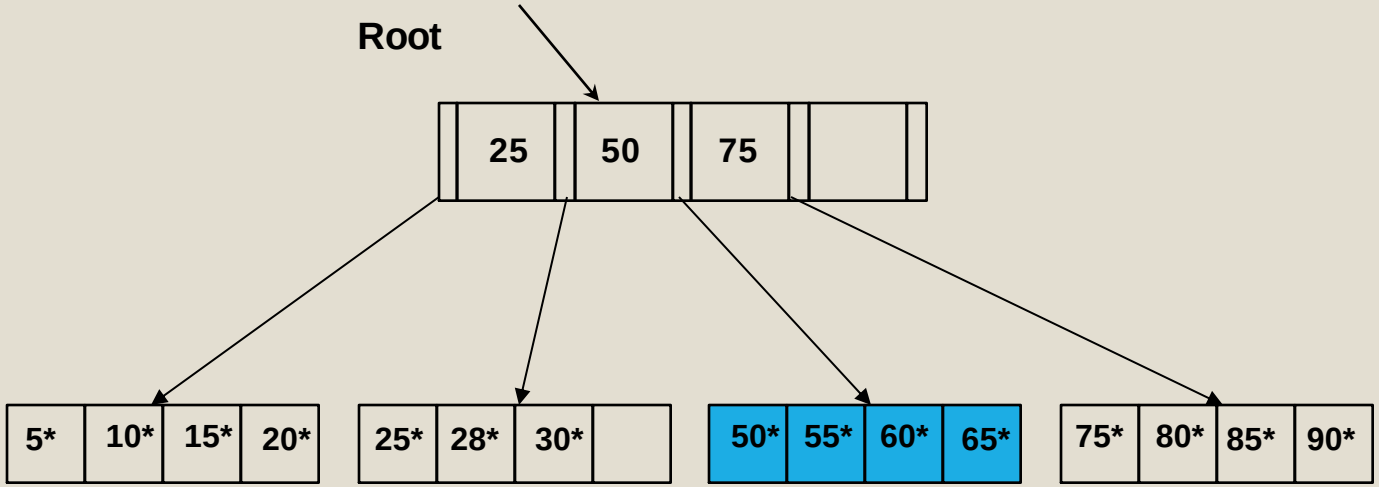
- The tree is always balanced in height
- The height of a balanced tree is the length of a path from root to leaf

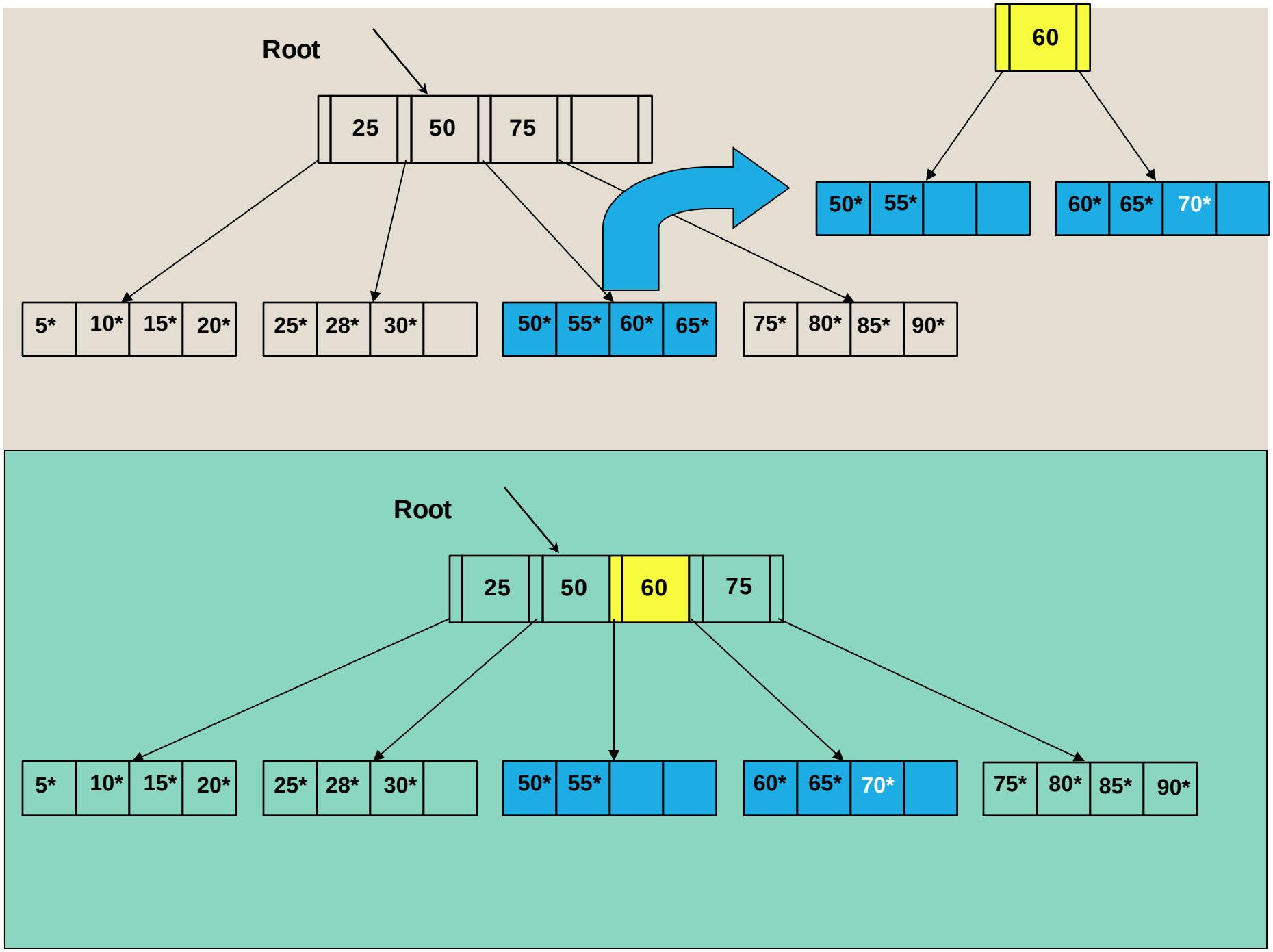
fan-out

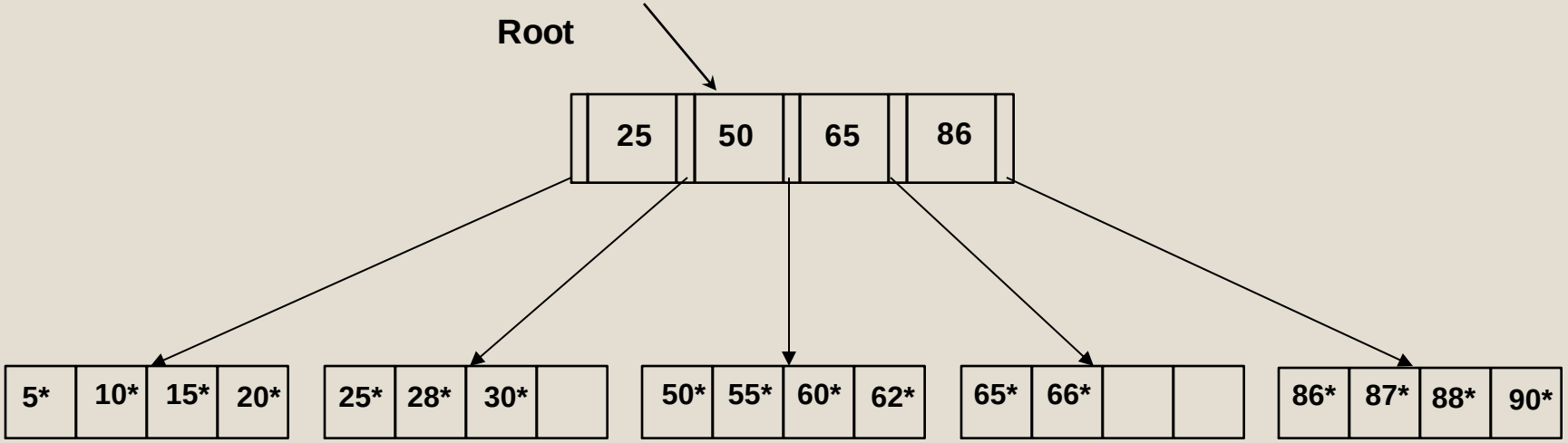
- The average number of children for a non-leaf node


Root
25
50
75
5*
10*
15*
20*
25*
28*
30*
50*
55*
60*
65*
75*
80*
85*
90*

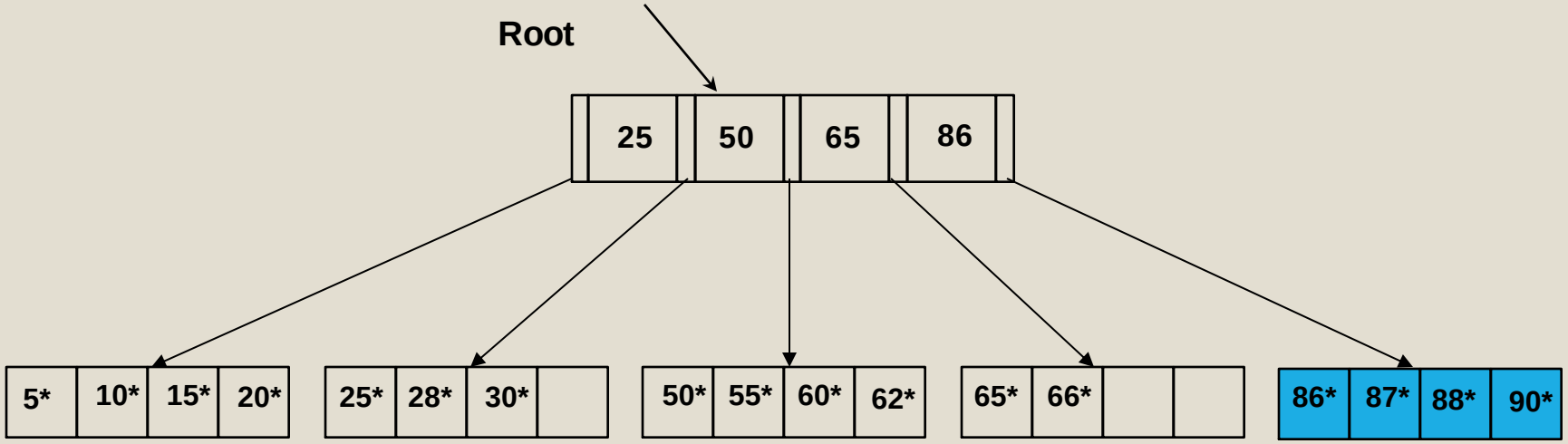
INSERT 70*

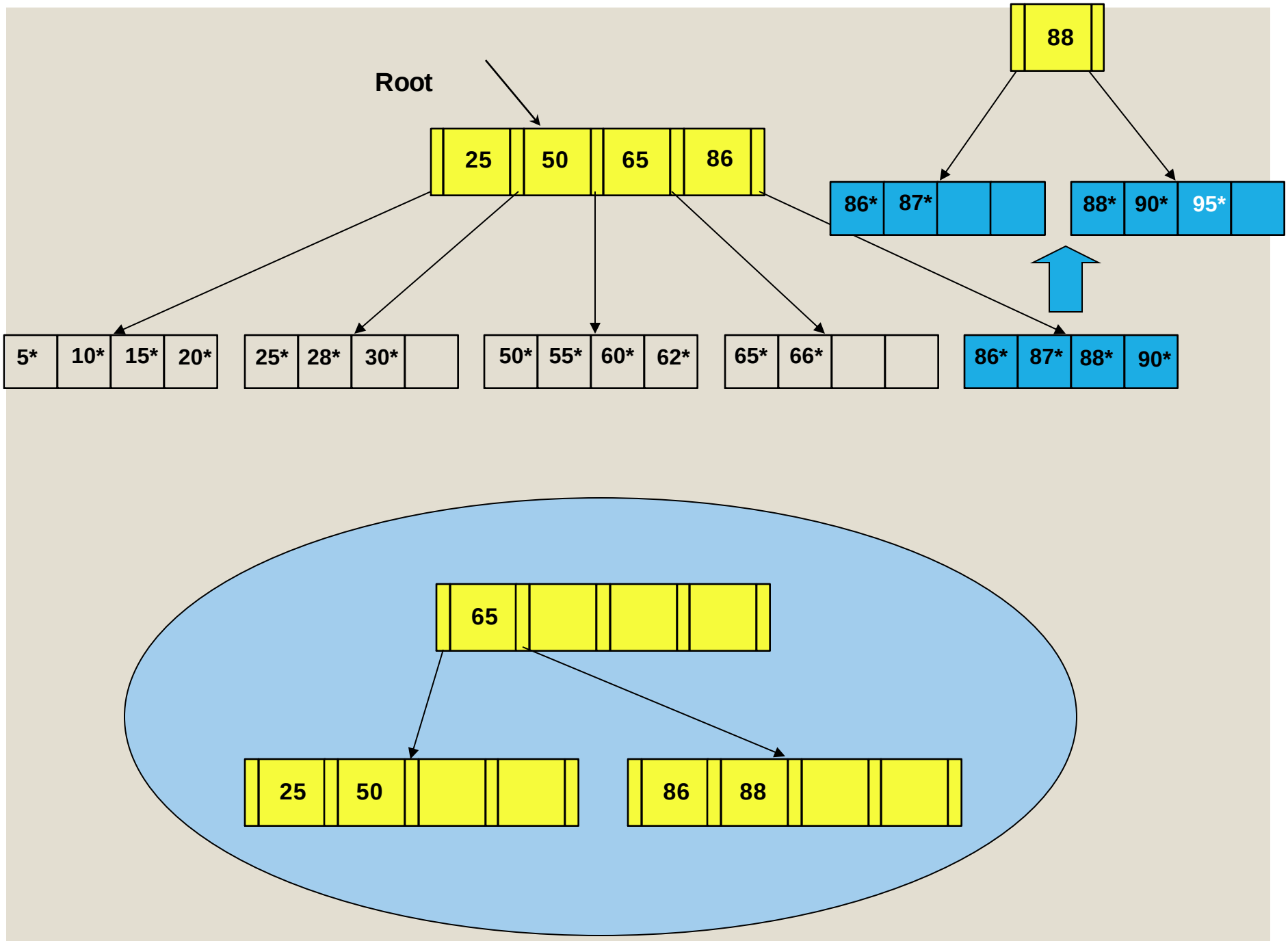

Root
25
50
75
5*
10*
15*
20*
25*
28*
30*
50*
55*
60*
65*
75*
80*
85*
90*

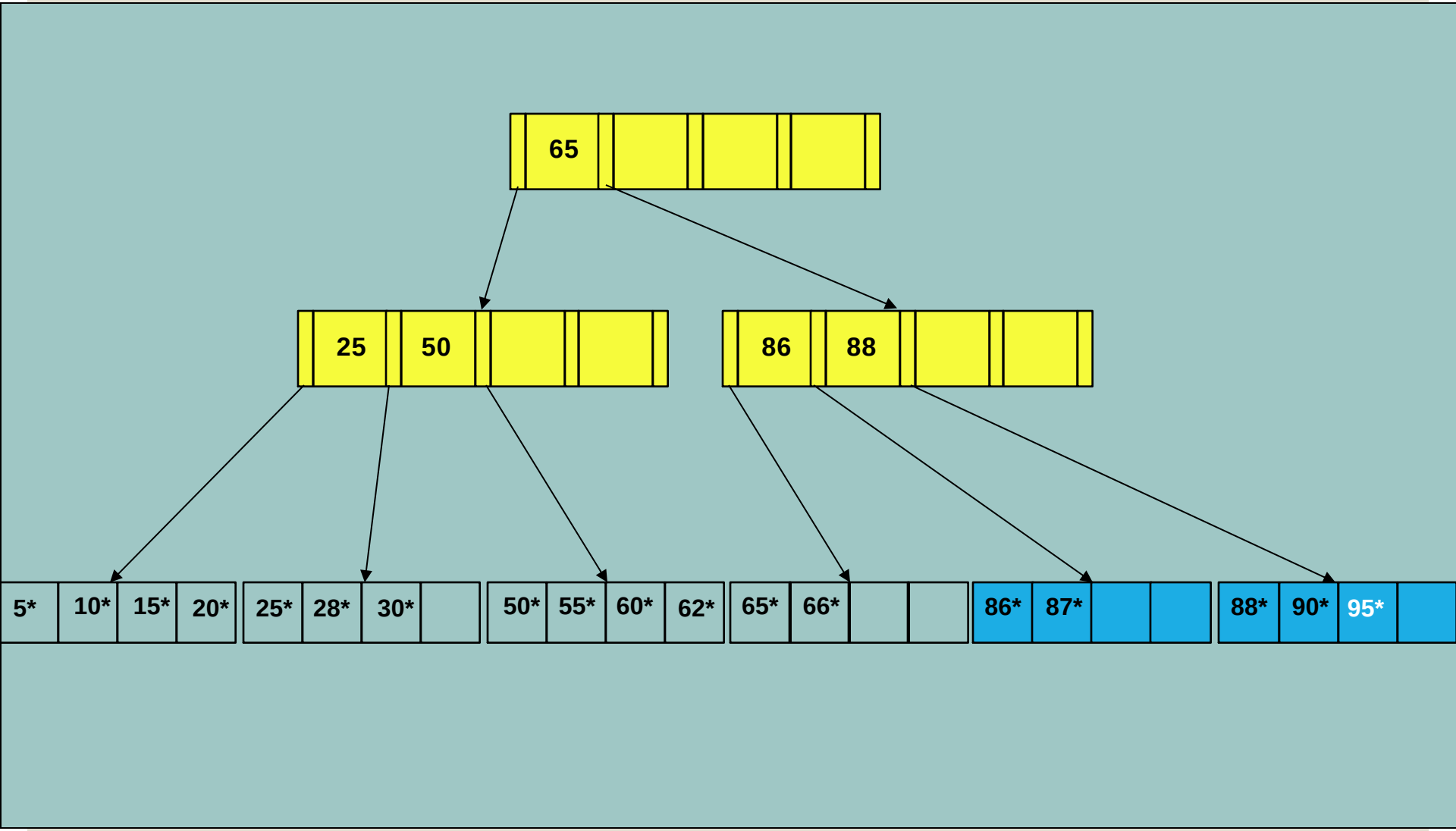

Root
25
50
75
60
50*
55*
60*
65*
70*
5*
10*
15*
20*
25*
28*
30*
50*
55*
60*
65*
75*
80*
85*
90*
Root
25
50
60
75
5*
10*
15*
20*
25*
28*
30*
50*
55*
60*
65*
70*
75*
80*
85*
90*


Root
25
50
65
86
5*
10*
15*
20*
25*
28*
30*
50*
55*
60*
62*
65*
66*
86*
87*
88*
90*

INSERT 95*


Root
25
50
65
86
5*
10*
15*
20*
25*
28*
30*
50*
55*
60*
62*
65*
66*
86*
87*
88*
90*


Root
88
25
50
65
86
86*
87*
88*
90*
95*
5*
10*
15*
20*
25*
28*
30*
50*
55*
60*
62*
65*
66*
86*
87*
88*
90*
65
25
50
86
88


65
25
50
86
88
5*
10*
15*
20*
25*
28*
30*
50*
55*
60*
62*
65*
66*
86*
87*
88*
90*
95*

COST MODEL FOR OUR ANALYSIS

We ignore CPU costs, for simplicity:

- **B:** The number of data pages
- **R:** Number of records per page
- **D:** (Average) time to read or write disk page
- Measuring number of page I/O's ignores gains of pre-fetching a sequence of pages; thus, even I/O cost is only approximated.
- **Average-case analysis**; based on several simplistic assumptions.

- *Good enough to show the overall trends!*

OPERATIONS TO COMPARE

SCAN

EQUALITY SEARCH

RANGE SELECTION

INSERT A RECORD

DELETE A RECORD

COMPARING FILE ORGANIZATIONS

- Heap files (random order; insert at eof)
- Sorted files, sorted on *a search key <age, sal>*
- Clustered B+ tree file on a search key, Alternative 1
- Heap file with unclustered B+ tree index on a search key
- Heap file with unclustered hash index on a search key

ASSUMPTIONS IN OUR ANALYSIS

Heap Files

- Equality selection on key
- exactly one match

Sorted Files

- Files compacted after deletions

Indexes

Alt 2 &3:

- data entry size = 10% size of record

Hash:

- No overflow buckets
- 80% page occupancy → File size = 1.25 data size

Tree:

- 67% occupancy (typical) → file size = 1.5 data size

COST OF OPERATIONS

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap					
(2) Sorted					
(3) Clustered					
(4) Unclustered Tree index					
(5) Unclustered Hash index					

- *Several assumptions underlie these (rough) estimates!*

COST OF OPERATIONS

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD				
(2) Sorted					
(3) Clustered					
(4) Unclustered Tree index					
(5) Unclustered Hash index					

- *Several assumptions underlie these (rough) estimates!*

COST OF OPERATIONS

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD			
(2) Sorted					
(3) Clustered					
(4) Unclustered Tree index					
(5) Unclustered Hash index					

- *Several assumptions underlie these (rough) estimates!*

COST OF OPERATIONS

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD		
(2) Sorted					
(3) Clustered					
(4) Unclustered Tree index					
(5) Unclustered Hash index					

- *Several assumptions underlie these (rough) estimates!*

COST OF OPERATIONS

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	
(2) Sorted					
(3) Clustered					
(4) Unclustered Tree index					
(5) Unclustered Hash index					

- *Several assumptions underlie these (rough) estimates!*

COST OF OPERATIONS

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted					
(3) Clustered					
(4) Unclustered Tree index					
(5) Unclustered Hash index					

- *Several assumptions underlie these (rough) estimates!*

COST OF OPERATIONS

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD				
(3) Clustered					
(4) Unclustered Tree index					
(5) Unclustered Hash index					

- *Several assumptions underlie these (rough) estimates!*

COST OF OPERATIONS

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	Dlog 2B			
(3) Clustered					
(4) Unclustered Tree index					
(5) Unclustered Hash index					

- *Several assumptions underlie these (rough) estimates!*

COST OF OPERATIONS

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	Dlog 2B	Dlog 2 B + #matches		
(3) Clustered					
(4) Unclustered Tree index					
(5) Unclustered Hash index					

- *Several assumptions underlie these (rough) estimates!*

COST OF OPERATIONS

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	Dlog 2B	Dlog 2 B + #matches	Search +BD	
(3) Clustered					
(4) Unclustered Tree index					
(5) Unclustered Hash index					

- *Several assumptions underlie these (rough) estimates!*

COST OF OPERATIONS

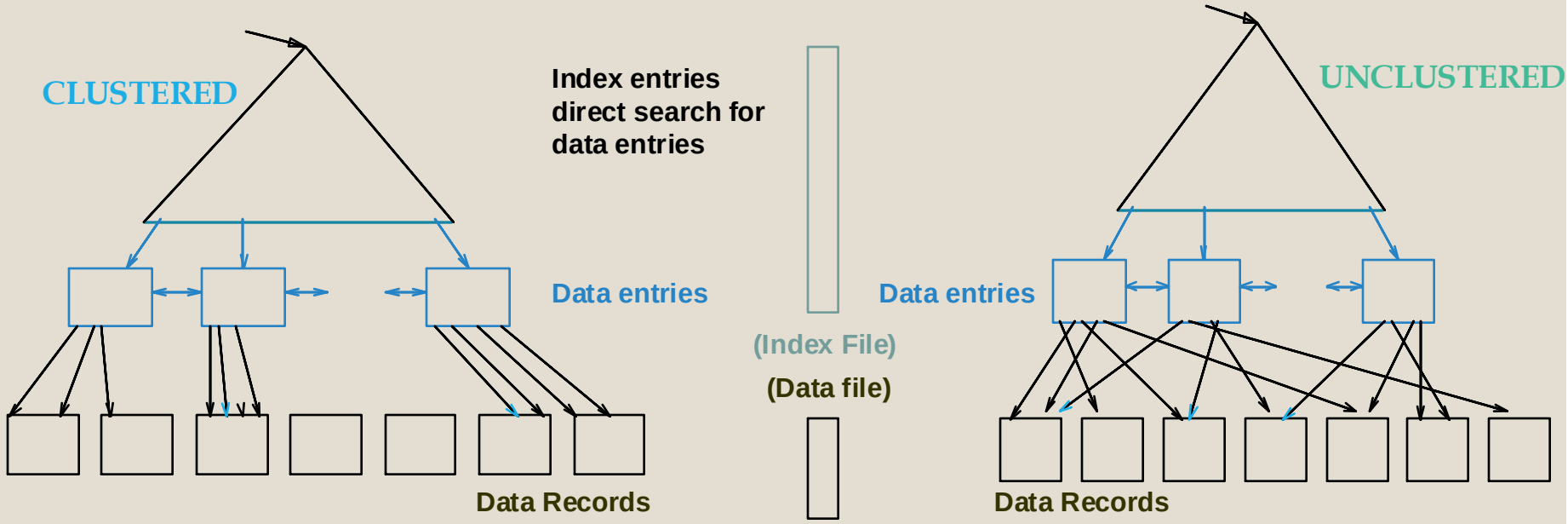
	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	Dlog 2B	Dlog 2 B + #matches	Search +BD	Search +BD
(3) Clustered					
(4) Unclustered Tree index					
(5) Unclustered Hash index					

- *Several assumptions underlie these (rough) estimates!*

CLUSTERED vs. UNCLUSTERED INDEX

Suppose that Alternative 2 is used for data entries, and that the data records are stored in a Heap file.

- To build clustered index, first sort the Heap file (with some free space on each page for future inserts).
- Overflow pages may be needed for inserts. (Thus, order of data records is 'close to', but not identical to, the sort order.)



COST OF OPERATIONS

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	Dlog 2B	Dlog 2 B + #matches	Search +BD	Search +BD
(3) Clustered	1.5BD				
(4) Unclustered Tree index					
(5) Unclustered Hash index					+2D

- *Several assumptions underlie these (rough) estimates!*

COST OF OPERATIONS

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	Dlog 2B	Dlog 2 B + #matches	Search +BD	Search +BD
(3) Clustered	1.5BD	Dlog F 1.5B			
(4) Unclustered Tree index					
(5) Unclustered Hash index					

- *Several assumptions underlie these (rough) estimates!*

COST OF OPERATIONS

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	Dlog 2B	Dlog 2 B + #matches	Search +BD	Search +BD
(3) Clustered	1.5BD	Dlog F 1.5B	Dlog F 1.5B +#matches		
(4) Unclustered Tree index					
(5) Unclustered Hash index					

- *Several assumptions underlie these (rough) estimates!*

COST OF OPERATIONS

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	Dlog 2B	Dlog 2 B + #matches	Search +BD	Search +BD
(3) Clustered	1.5BD	Dlog F 1.5B	Dlog F 1.5B +#matches	Search +D	
(4) Unclustered Tree index					
(5) Unclustered Hash index					

- *Several assumptions underlie these (rough) estimates!*

COST OF OPERATIONS

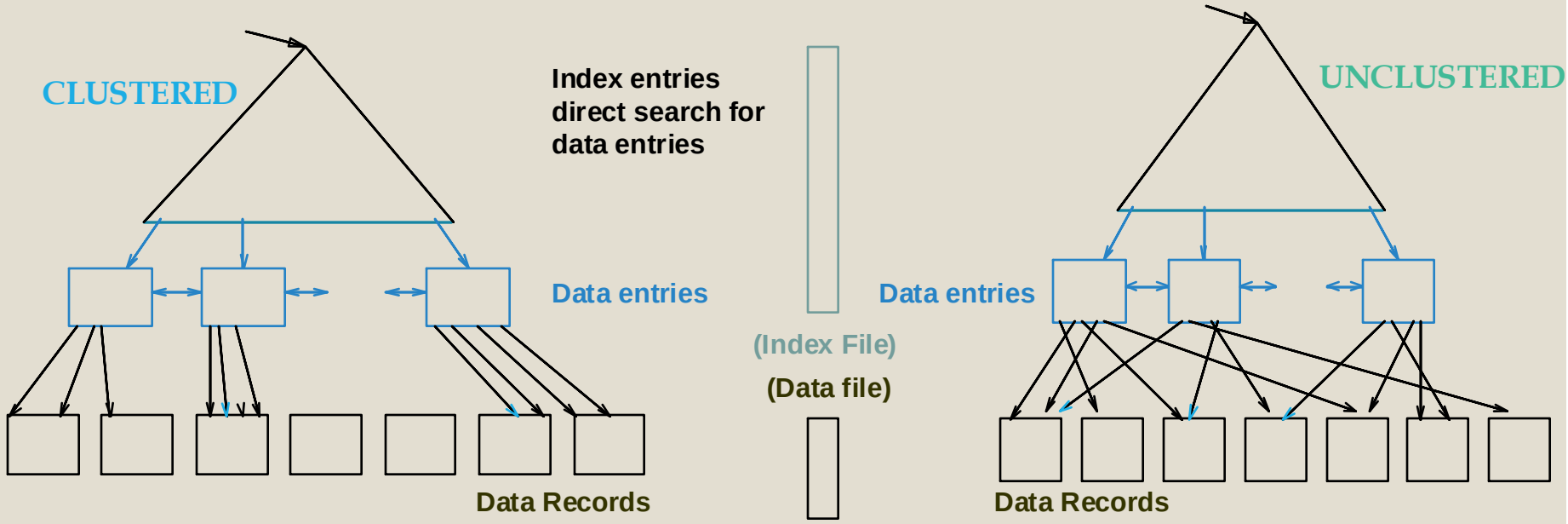
	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	Dlog 2B	Dlog 2 B + #matches	Search +BD	Search +BD
(3) Clustered	1.5BD	Dlog F 1.5B	Dlog F 1.5B +#matches	Search +D	Search +D
(4) Unclustered Tree index					
(5) Unclustered Hash index					

- *Several assumptions underlie these (rough) estimates!*

CLUSTERED vs. UNCLUSTERED INDEX

Suppose that Alternative 2 is used for data entries, and that the data records are stored in a Heap file.

- To build clustered index, first sort the Heap file (with some free space on each page for future inserts).
- Overflow pages may be needed for inserts. (Thus, order of data records is 'close to', but not identical to, the sort order.)



COST OF OPERATIONS

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	Dlog 2B	Dlog 2 B + #matches	Search +BD	Search +BD
(3) Clustered	1.5BD	Dlog F 1.5B	Dlog F 1.5B +#matches	Search +D	Search +D
(4) Unclustered Tree index					
(5) Unclustered Hash index					

- *Several assumptions underlie these (rough) estimates!*

COST OF OPERATIONS

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	Dlog 2B	Dlog 2 B + #matches	Search +BD	Search +BD
(3) Clustered	1.5BD	Dlog F 1.5B	Dlog F 1.5B +#matches	Search +D	Search +D
(4) Unclustered Tree index	BD(R+0.15)				
(5) Unclustered Hash index					

- *Several assumptions underlie these (rough) estimates!*

COST OF OPERATIONS

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	Dlog 2B	Dlog 2 B + #matches	Search +BD	Search +BD
(3) Clustered	1.5BD	Dlog F 1.5B	Dlog F 1.5B +#matches	Search +D	Search +D
(4) Unclustered Tree index	BD(R+0.15)	D(1 + log F 0.15B)			
(5) Unclustered Hash index					

- *Several assumptions underlie these (rough) estimates!*

COST OF OPERATIONS

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	Dlog 2B	Dlog 2 B + #matches	Search +BD	Search +BD
(3) Clustered	1.5BD	Dlog F 1.5B	Dlog F 1.5B +#matches	Search +D	Search +D
(4) Unclustered Tree index	BD(R+0.15)	D(1 + log F 0.15B)	Dlog F 0.15B +#matches		
(5) Unclustered Hash index					

- *Several assumptions underlie these (rough) estimates!*

COST OF OPERATIONS

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	Dlog 2B	Dlog 2 B + #matches	Search +BD	Search +BD
(3) Clustered	1.5BD	Dlog F 1.5B	Dlog F 1.5B +#matches	Search +D	Search +D
(4) Unclustered Tree index	BD(R+0.15)	D(1 + log F 0.15B)	Dlog F 0.15B +#matches	D(3 + log F 0.15B)	
(5) Unclustered Hash index					

- *Several assumptions underlie these (rough) estimates!*

COST OF OPERATIONS

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	Dlog 2B	Dlog 2 B + #matches	Search +BD	Search +BD
(3) Clustered	1.5BD	Dlog F 1.5B	Dlog F 1.5B +#matches	Search +D	Search +D
(4) Unclustered Tree index	BD(R+0.15)	D(1 + log F 0.15B)	Dlog F 0.15B +#matches	D(3 + log F 0.15B)	Search +2D
(5) Unclustered Hash index					

- *Several assumptions underlie these (rough) estimates!*

COST OF OPERATIONS

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	Dlog 2B	Dlog 2 B + #matches	Search +BD	Search +BD
(3) Clustered	1.5BD	Dlog F 1.5B	Dlog F 1.5B +#matches	Search +D	Search +D
(4) Unclustered Tree index	BD(R+0.15)	D(1 + log F 0.15B)	Dlog F 0.15B +#matches	D(3 + log F 0.15B)	Search +2D
(5) Unclustered Hash index	BD(R+0.125)				

- *Several assumptions underlie these (rough) estimates!*

COST OF OPERATIONS

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	Dlog 2B	Dlog 2 B + #matches	Search +BD	Search +BD
(3) Clustered	1.5BD	Dlog F 1.5B	Dlog F 1.5B +#matches	Search +D	Search +D
(4) Unclustered Tree index	BD(R+0.15)	D(1 + log F 0.15B)	Dlog F 0.15B +#matches	D(3 + log F 0.15B)	Search +2D
(5) Unclustered Hash index	BD(R+0.125)	2D			

- *Several assumptions underlie these (rough) estimates!*

COST OF OPERATIONS

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	Dlog 2B	Dlog 2 B + #matches	Search +BD	Search +BD
(3) Clustered	1.5BD	Dlog F 1.5B	Dlog F 1.5B +#matches	Search +D	Search +D
(4) Unclustered Tree index	BD(R+0.15)	D(1 + log F 0.15B)	Dlog F 0.15B +#matches	D(3 + log F 0.15B)	Search +2D
(5) Unclustered Hash index	BD(R+0.125)	2D	BD		

- *Several assumptions underlie these (rough) estimates!*

COST OF OPERATIONS

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	Dlog 2B	Dlog 2 B + #matches	Search +BD	Search +BD
(3) Clustered	1.5BD	Dlog F 1.5B	Dlog F 1.5B +#matches	Search +D	Search +D
(4) Unclustered Tree index	BD(R+0.15)	D(1 + log F 0.15B)	Dlog F 0.15B +#matches	D(3 + log F 0.15B)	Search +2D
(5) Unclustered Hash index	BD(R+0.125)	2D	BD	4D	

- *Several assumptions underlie these (rough) estimates!*

COST OF OPERATIONS

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	$D \log_2 B$	$D \log_2 B$ + #matches	Search +BD	Search +BD
(3) Clustered	1.5BD	$D \log_F 1.5B$	$D \log_F 1.5B$ +#matches	Search +D	Search +D
(4) Unclustered Tree index	BD(R+0.15)	$D(1 + \log_F 0.15B)$	$D \log_F 0.15B$ +#matches	$D(3 + \log_F 0.15B)$	Search +2D
(5) Unclustered Hash index	BD(R+0.125)	2D	BD	4D	Search +2D

- *Several assumptions underlie these (rough) estimates!*

UNDERSTANDING THE WORKLOAD

For each query in the workload:

- Which relations does it access?
- Which attributes are retrieved?
- Which attributes are involved in selection/join conditions?
  - How selective are these conditions likely to be?

For each update in the workload:

- Which attributes are involved in selection/join conditions?
  - How selective are these conditions likely to be?
- The type of update (INSERT/DELETE/UPDATE), and the attributes that are affected.

CHOICE OF INDEXES

- What indexes should we create?
  - Which relations should have indexes?
  - What field(s) should be the search key?
  - Should we build several indexes?
- For each index,
  - what kind of an index should it be?
    - Clustered?
    - Hash/Tree?

CHOICE OF INDEXES (CONTD.)

- One approach:
  Consider the most important **queries** in turn.
  Consider the best plan using the current indexes, and see if a better plan is possible with an additional index.
  If so, create it.
  - Obviously, this implies that we must understand how a DBMS evaluates queries and creates query evaluation plans.
  - For now, we discuss simple 1-table queries.
- Before creating an index, must also consider the impact on updates in the workload!
  - Trade-off: Indexes can make queries go faster, updates slower. Require disk space, too.
  - Do not over-index heavily updated tables.

INDEX SELECTION GUIDELINES

- Attributes in WHERE clause are candidates for index keys.
  - Exact match condition suggests hash index.
  - Range query suggests tree index.
    - Clustering is especially useful for range queries; can also help on equality queries if there are many duplicates.
- Multi-attribute search keys should be considered when a WHERE clause contains several conditions.
  - Order of attributes is important for range queries.
  - Such indexes can sometimes enable **index-only strategies** for important queries.
    - For index-only strategies, clustering is not important!
- Try to choose indexes that benefit as many queries as possible. Since only one index can be clustered per relation, choose it based on important queries that would benefit the most from clustering.

EXAMPLES OF CLUSTERED INDEXES

```
SELECT  E.dno
FROM  Emp E
WHERE  E.age>40
```

B+ tree index on *E.age* can be used to get qualifying tuples.

- How selective is the condition?
- Is the index clustered?

```
SELECT  E.dno,  COUNT (*)
FROM  Emp E
WHERE  E.age>10
GROUP BY E.dno
```

Consider the GROUP BY query.

- If many tuples have *E.age* > 10, using *E.age* index and sorting the retrieved tuples may be costly.
- Clustered *E.dno* index may be better!

```
SELECT  E.dno
FROM  Emp E
WHERE  E.hobby='Stamps'
```

Equality queries and duplicates:

- Clustering on *E.hobby* helps!

INDEXES WITH COMPOSITE SEARCH KEYS

Composite Search Keys:

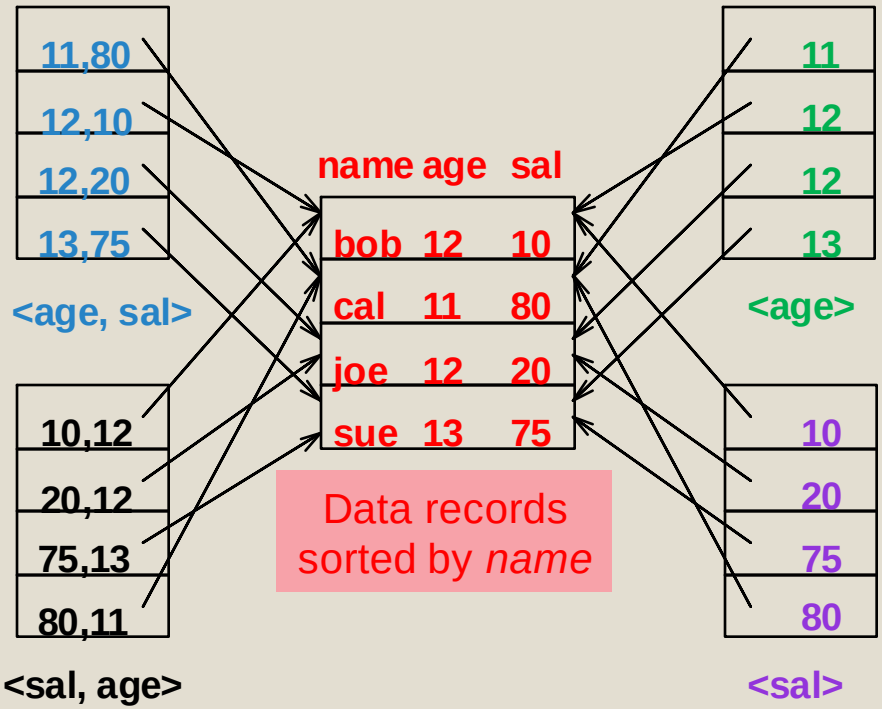
Search on a combination of fields.

- Equality query: Every field value is equal to a constant value, e.g., wrt <sal,age> index:
  - age=20 and sal =75
- Range query: Some field value is not a constant. E.g.:
  - age =20; or age=20 and sal > 10

Data entries in index sorted by search key to support range queries.

- Lexicographic order, or
- Spatial order.

Examples of composite key indexes using lexicographic order.



Data records sorted by *name*

Data entries sorted by <*sal,age*>

Data entries sorted by <*sal*>

COMPOSITE SEARCH KEYS

To retrieve Emp records with *age*=30 AND *sal*=4000, an index on <*age,sal*> would be better than an index on *age* or an index on *sal*.

- Choice of index key is orthogonal to clustering, etc.

If condition is: 20<*age*<30 AND 3000<*sal*<5000:

- Clustered tree index on <*age,sal*> or <*sal,age*> is best.

If condition is: *age*=30 AND 3000<*sal*<5000:

- Clustered <*age,sal*> index is much better than <*sal,age*> index!

Composite indexes are larger, updated more often.

INDEX-ONLY PLANS

A number of queries can be answered without retrieving any tuples from one or more of the relations involved if a suitable index is available.

```
SELECT  D.mgr
FROM  Dept D, Emp E
WHERE  D.dno=E.dno
```

<E.dno>

```
SELECT  D.mgr, E.eid
FROM  Dept D, Emp E
WHERE  D.dno=E.dno
```

<E.dno,E.eid>

Tree index

```
SELECT  E.dno, COUNT(*)
FROM  Emp E
GROUP BY  E.dno
```

<E.dno>

```
SELECT  E.dno, MIN(E.sal)
FROM  Emp E
GROUP BY  E.dno
```

<E.dno,E.sal>

Tree index

```
SELECT AVG(E.sal)
FROM  Emp E
WHERE  E.age=25 AND
  E.sal BETWEEN 3000 AND 5000
```

<E. age,E.sal>

or

<E.sal, E.age>

Tree index

INDEX-ONLY PLANS (CONTD.)

Index-only plans are possible if the key is

<dno,age>

or we have a tree index with the key

<age,dno>

- Which is better?
- What if we consider the second query?

```
SELECT  E.dno,  COUNT (*)
FROM  Emp E
WHERE  E.age=30
GROUP BY E.dno
```

```
SELECT  E.dno,  COUNT (*)
FROM  Emp E
WHERE  E.age>30
GROUP BY E.dno
```

SUMMARY

- Many alternative file organizations exist, each appropriate in some situation.
- If selection queries are frequent, sorting the file or building an *index* is important.
  - Hash-based indexes only good for equality search.
  - Sorted files and tree-based indexes best for range search; also good for equality search.
  - (Files rarely kept sorted in practice; B+ tree index is better.)
- Index is a collection of data entries plus a way to quickly find entries with given key values.

SUMMARY
(CONTD.)

- Data entries can be actual data records, <key, rid> pairs, or <key, rid-list> pairs.
  - Choice orthogonal to *indexing technique* used to locate data entries with a given key value.
- Can have several indexes on a given file of data records, each with a different search key.
- Indexes can be classified as clustered vs. unclustered, primary vs. secondary, and dense vs. sparse. Differences have important consequences for utility/performance.

SUMMARY
(CONTD.)

- Understanding the nature of the *workload* for the application, and the performance goals, is essential to developing a good design.
  - What are the important queries and updates? What attributes/relations are involved?
- Indexes must be chosen to speed up important queries (and perhaps some updates!).
  - Index maintenance overhead on updates to key fields.
  - Choose indexes that can help many queries, if possible.
  - Build indexes to support index-only strategies.
  - Clustering is an important decision; only one index on a given relation can be clustered!
  - Order of fields in composite index key can be important.

Exercises

1

Emp(eid: integer, ename: varchar, sal: integer, age: integer, did: integer)
Dept(did: integer, budget: integer, floor: integer, mgr eid: integer)

Salaries range from $10,000 to $100,000, ages vary from 20 to 80, each department has about five employees on average, there are 10 floors, and budgets vary from $10,000 to $1 million. You can assume uniform distributions of values.
For each of the following queries, which of the listed index choices would you choose to speed up the query?
Query: Print ename, age, and sal for all employees.

a. Clustered hash index on <ename, age, sal> fields of Emp.
b. Unclustered hash index on <ename, age, sal> fields of Emp.
c. Clustered B+ tree index on <ename, age, sal> fields of Emp.
d. Unclustered hash index on <eid, did> fields of Emp.
e. No index.

2

Emp(eid: integer, ename: varchar, sal: integer, age: integer, did: integer)
Dept(did: integer, budget: integer, floor: integer, mgr eid: integer)

Salaries range from $10,000 to $100,000, ages vary from 20 to 80, each department has about five employees on average, there are 10 floors, and budgets vary from $10,000 to $1 million. You can assume uniform distributions of values.
For each of the following queries, which of the listed index choices would you choose to speed up the query?

Query: Find the dids of departments that are on the 10^{th} floor and have a budget of less than $15,000.

a. Clustered hash index on the floor field of Dept.
b. Unclustered hash index on the floor field of Dept.
c. Clustered B+ tree index on <floor, budget> fields of Dept.
d. Clustered B+ tree index on the budget field of Dept.
e. No index.

3

From the SQL query below, which index structure will not be beneficial to the query execution?

```
Select * From Apply, College
Where Apply.cName = College.cName
And Apply.major = 'cs' and College.enrollment <5000
```

a. Tree-based index on Apply.cName
b. Hash-baed index on Apply.major
c. Hash-based index on College.enrollment
d. Hash-based index on College.cName
e. More than one choices are not beneficial

4

สำหรับฐานข้อมูลที่ใช้ Unclustered hash index [กำหนดให้ occupancy rate = 67% (ไม่มี overflow chains), Data entry size = 20% ของ data records, B = จำนวน Data page เมื่อมี record แพ็คอยู่เต็มโดยไม่มี slot ว่าง, R = จำนวน record ในแต่ละ page, D = เวลาเฉลี่ยในการอ่าน/เขียนแต่ละ disk page]

Leaf page แต่ละ page จะมีจำนวน Data entry เท่าใด

ถ้าต้องการ List data record ออกมาทั้งหมดโดยใช้ Hash index ต้องใช้เวลาทั้งหมดเท่าใด

5

ในกรณีที่ใช้ Clustered files index (B+ tree: Alternative (1)) กับฐานข้อมูลที่มีการเพิ่ม record ใหม่อยู่ตลอด จึงกำหนดให้ occupancy rate เป็น 25% ให้หาค่า cost ในการหา leaf page ที่เหมาะสมในการ Insert ข้อมูลใหม่ในแต่ละครั้ง (Average case) [กำหนดให้ F = Fanout ของ B+ tree, B = จำนวน Data page เมื่อมี record แพ็คอยู่เต็มโดยไม่มี slot ว่าง, R = จำนวน record ในแต่ละ page, D = เวลาเฉลี่ยในการอ่าน/เขียนแต่ละ disk page]