Faculty of Computer Science and Engineering
Ho Chi Minh City University of Technology

Chapter 6: Physical Storage and Data Management

Database Systems (CO2013)

Computer Science Program

Assoc. Prof. Dr. Võ Thị Ngọc Châu

(chauvtn@hcmut.edu.vn)

Semester 1 – 2022-2023

Content

Chapter 6: Physical Storage and Data Management

- 6.1. Physical Data Storage
- 6.2. Indexing
- 6.3. Complex Data Management Approaches (Semi-structured and Unstructured Data)
- 6.4. Massive Data Management Approaches
- 6.5. Quality Issues: Reliability, Scalability, Effectiveness, Efficiency

Main References

Text:

- [1] R. Elmasri, S. R. Navathe, Fundamentals of Database Systems- 6th Edition, Pearson- Addison Wesley, 2011.
  - ***R. Elmasri, S. R. Navathe, Fundamentals of Database Systems- 7th Edition, Pearson, 2016.***

References:

[1] S. Chittayasothorn, *Relational Database Systems: Language, Conceptual Modeling and Design for Engineers*, Nutcha Printing Co. Ltd, 2017.

[3] A. Silberschatz, H. F. Korth, S. Sudarshan, Database System Concepts – 7th Edition, McGraw-Hill, 2020.

[4] H. G. Molina, J. D. Ullman, J. Widom, *Database Systems: The Complete Book - 2nd Edition,* Prentice-Hall, 2009.

[5] R. Ramakrishnan, J. Gehrke, *Database Management Systems – 4th Edition,* McGraw-Hill, 2018.

[6] M. P. Papazoglou, S. Spaccapietra, Z. Tari, *Advances in Object-Oriented Data Modeling*, MIT Press, 2000.

[7]. G. Simsion, *Data Modeling: Theory and Practice*, Technics Publications, LLC, 2007.

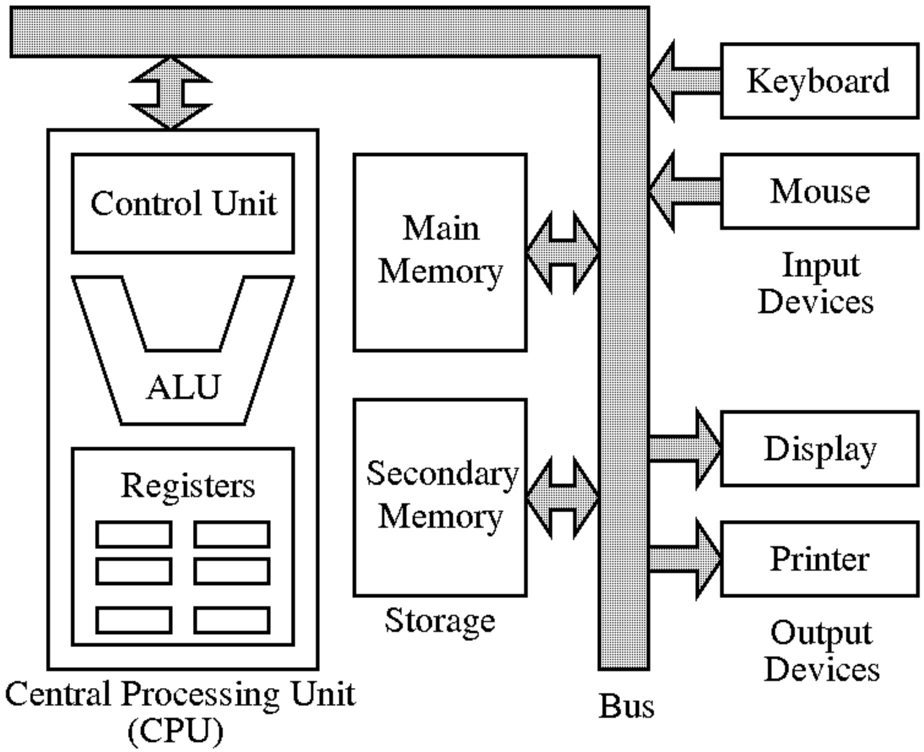
6.1. Physical Data Storage

- Database
  - A collection of data and their relationships
  - Computerized

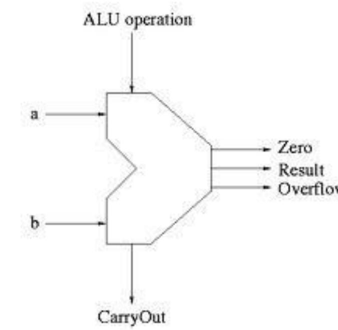
Database	Database
↓	↓
Base Tables (Relations)	Files
↓	↓
Pages (Blocks)	*Pages (Blocks)*
↓	↓
Rows (Tuples)	Records
⇣	⇣
Column values (Attribute values)	*Field values*
Logical Perspectives	**Physical Perspectives**

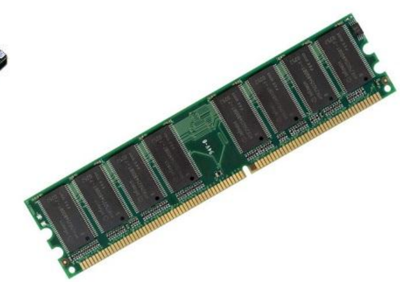
Stored physically on computer storage media

Computer Organization - Hardware



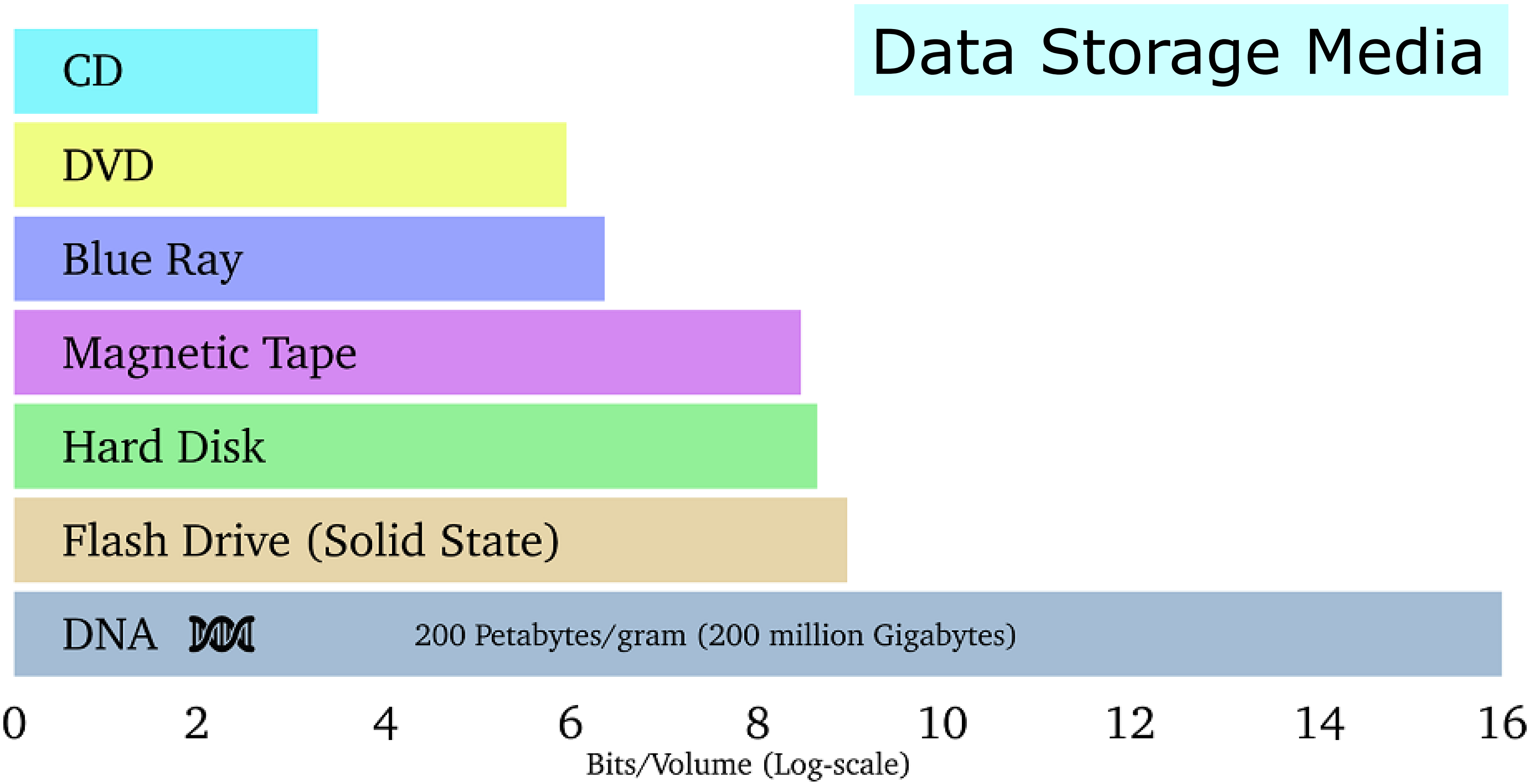
Computer Architecture





ALU = Arithmetic/logic gate unit: performing arithmetic and logic operations on data

Today’s Storage Technologies



Density in storage media

Source: Potomac Institute for Policy Studies, The Future of DNA Data Storage, Report, 2018.

Today's Storage Technologies

SSD

Specifications	Nytro 1351 SATA SSD — Light Endurance				
Capacity	3.84TB	1.92TB	960GB	480GB	240GB
Standard Model	XA3840LE10063	XA1920LE10063	XA960LE10063	XA480LE10063	XA240LE10003
Seagate Secure™ SED Model (TCG Enterprise)	XA3840LE10083	XA1920LE10083	XA960LE10083	XA480LE10083	XA240LE10023
Seagate Secure SED Model (TCG OPAL)[1]	XA3840LE10103	XA1920LE10103	XA960LE10103	XA480LE10103	XA240LE10043
Features					
Interface	SATA 6Gb/s	SATA 6Gb/s	SATA 6Gb/s	SATA 6Gb/s	SATA 6Gb/s
NAND Flash Type	3D TLC	3D TLC	3D TLC	3D TLC	3D TLC
Form Factor	2.5 in x 7mm	2.5 in x 7mm	2.5 in x 7mm	2.5 in x 7mm	2.5 in x 7 mm, 2.5 in x 7mm
Performance					
Sequential Read (MB/s) Sustained, 128KB QD32[2,3,4]	564	564	564	564	564
Sequential Write (MB/s) Sustained, 128KB QD32[2,3,4]	536	536	536	488	232
Random Read (IOPS) Sustained, 4KB QD32[2,3,4]	90,000	94,000	93,000	80,000	55,000
Random R70R (IOPS) Sustained, 4KB QD32[2,3,4]	46,000	47,000	40,000	29,000	18,000
Random Write (IOPS) Sustained, 4KB QD32[2,3,4]	22,000	23,000	22,000	16,000	8,000
Average Read Latency (µs), 4KB QD1[2,3,4]	172	154	153	154	154
Average Write Latency (µs), 4KB QD1[2,3,4]	58	58	58	65	124
Endurance/Reliability					
Lifetime Endurance (Drive Writes per Day)	1	1	1	1	1
Total Bytes Written to Flash (TB)	12,300	6,140	3,070	1,540	768
Non-recoverable Read Errors per Bits Read	1 per 10E17	1 per 10E17	1 per 10E17	1 per 10E17	1 per 10E17
Mean Time Between Failures (MTBF, hours)	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
Warranty, Limited (years)[5]	5	5	5	5	5
Power Management					
+5/+12V Overall Average Active Power (W)[6]	3.5	3.4	3.2	2.7	2.3
Average Idling Power (W)	1.2	1.2	1.2	1.1	1.1
Environmental					
Temperature, Operating Internal (°C)	0°C – 70°C	0°C – 70°C	0°C – 70°C	0°C – 70°C	0°C – 70°C
Temperature, Non-operating (°C)	-40°C – 85°C	-40°C – 85°C	-40°C – 85°C	-40°C – 85°C	-40°C – 85°C
Temperature Change Rate/Hr, Max (°C)	20	20	20	20	20
Shock, 0.5 ms (Gs)	1,000	1,000	1,000	1,000	1,000

<u>Source</u>:

seagate.com, 2019

Today’s Storage Technologies

- Direct-attached storage (DAS)
  - JBOD (just-a-bunch-of-disks)
  - RAID (Redundant Arrays of Inexpensive/ Independent Disks)
- Network-attached storage (NAS)
- Storage area network (SAN)
- Cloud storage and virtualization

6.1. Disk Storage

- Memory hierarchy and storage devices
  - The highest-speed memory is the most expensive and is therefore available with the least capacity.
  - The lowest-speed memory is offline tape storage, which is essentially available in indefinite (*without clear limits*) storage capacity.

Primary storage level	Secondary and tertiary storage level
- Register	- <u>Magnetic disk</u>
- Cache (static RAM)	- Mass storage (CD-ROM, DVD)
- <u>DRAM (dynamic RAM)</u>	- Tape

6.1. Disk Storage

- Types of storage with capacity, access time, max bandwidth (transfer speed), and commodity cost

Type	Capacity*	Access Time	Max Bandwidth	Commodity Prices (2014)**
Main Memory- RAM	4GB–1TB	30ns	35GB/sec	$100–$20K
Flash Memory- SSD	64 GB–1TB	50μs	750MB/sec	$50–$600
Flash Memory- USB stick	4GB–512GB	100μs	50MB/sec	$2–$200
Magnetic Disk	400 GB–8TB	10ms	200MB/sec	$70–$500
Optical Storage	50GB–100GB	180ms	72MB/sec	$100
Magnetic Tape	2.5TB–8.5TB	10s–80s	40–250MB/sec	$2.5K–$30K
Tape jukebox	25TB–2,100,000TB	10s–80s	250MB/sec–1.2PB/sec	$3K–$1M+

*Capacities are based on commercially available popular units in 2014.

**Costs are based on commodity online marketplaces.

Table 16.1, pp. 545

[1] ***R. Elmasri, S. R. Navathe, Fundamentals of Database Systems- 7th Edition, Pearson, 2016.***

6.1. Disk Storage

- Storage organization of databases
  - Databases typically store large amounts of data that must persist over long periods of time.

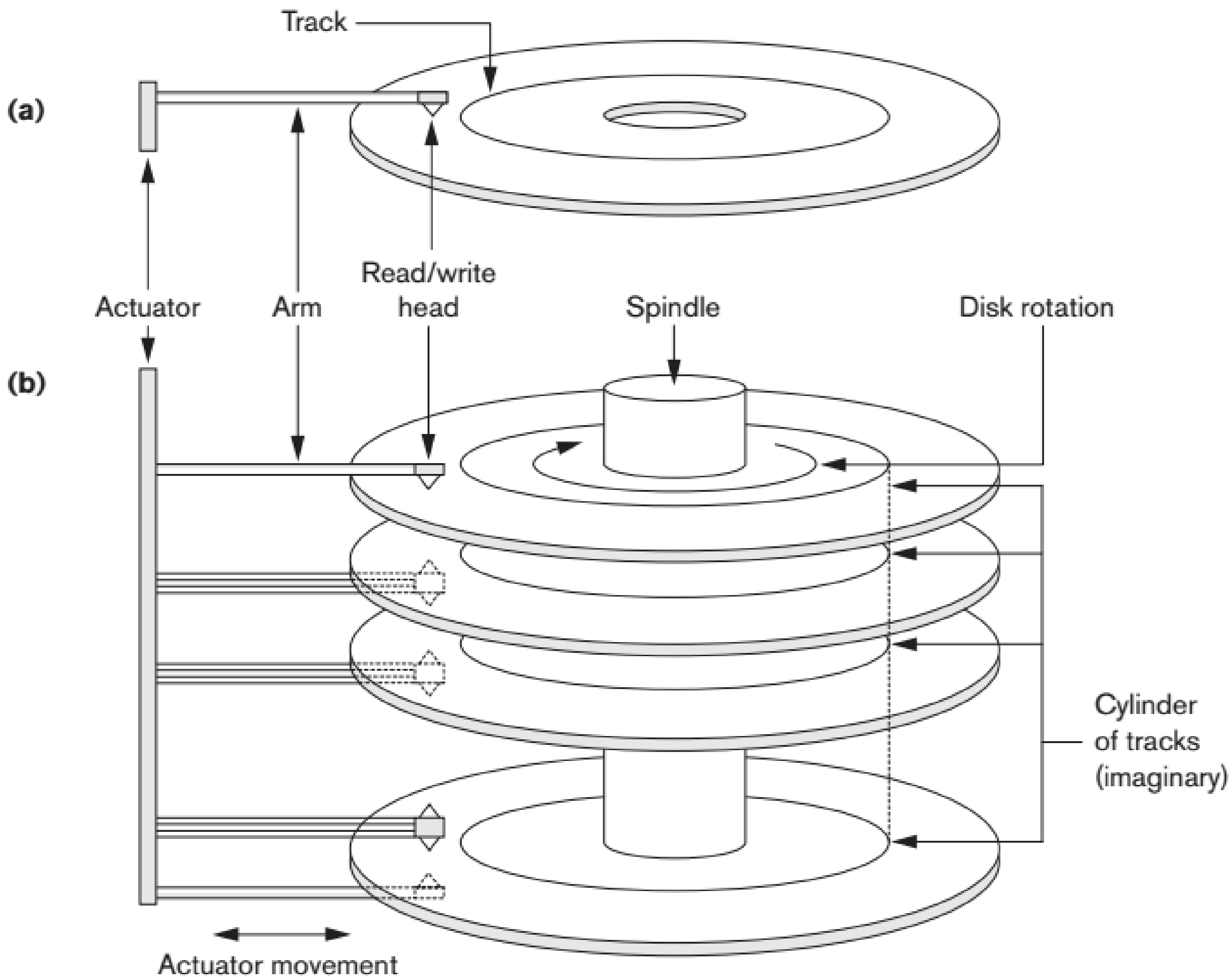
  → ***Persistent data*** (not *transient data* which persists for only a limited time during program execution)

  - Most databases are stored permanently (or *persistently*) on *magnetic disk* secondary storage.
    - Database size
    - No permanent loss of stored data with nonvolatile storage
    - Storage cost

6.1. Disk Storage

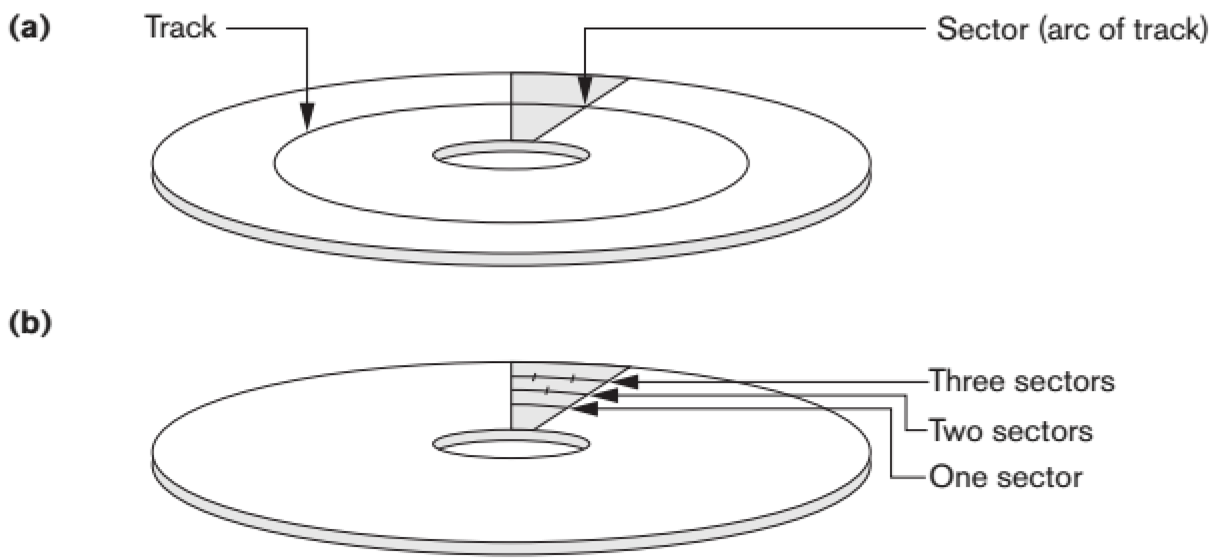
- Magnetic disks
  - Disks are covered with **magnetic** material.
  - The most basic unit of data on the disk is a single **bit** of information.
  - By magnetizing an area on a disk in certain ways, one can make that area represent a bit value of either 0 (zero) or 1 (one).
  - To code information, bits are grouped into **bytes** (or **characters**): 1 byte = 8 bits, normally.
  - The **capacity** of a disk is the number of bytes it can store.
  - Whatever their capacity, all disks are made of magnetic material shaped as a thin circular disk.

6.1. Disk Storage



(a) A single-sided disk with read/write hardware. (b) A disk pack with read/write hardware. Figure 16.1, pp. 548, [1]

6.1. Disk Storage



Different sector organizations on disk.

(a) Sectors subtending a fixed angle.

(b) Sectors maintaining a uniform recording density.

Figure 16.2, pp. 548, [1]

6.1. Disk Storage

- Magnetic disks
  - A disk is **single-sided** if it stores information on one of its surfaces only and **double-sided** if both surfaces are used.
  - To increase storage capacity, disks are assembled into a **disk pack**.
  - Information is stored on a disk surface in concentric circles of *small width,* each having a distinct diameter. Each circle is called a **track**.
  - In disk packs, tracks with the same diameter on the various surfaces are called a **cylinder**.

6.1. Disk Storage

- Magnetic disks
  - A track is divided into smaller blocks or sectors.
  - The division of a track into **sectors** is *hard-coded* on the disk surface and cannot be changed.
    - One type of sector organization calls a portion of a track that subtends a fixed angle at the center a sector.
  - The division of a track into equal-sized **disk blocks** (or **pages**) is set by the operating system during disk **formatting** (or **initialization**).
    - Block size is fixed during initialization and cannot be changed dynamically: from 512 bytes to 8,192 bytes.

6.1. Disk Storage

- Magnetic disks
  - A disk with hard-coded sectors often has the sectors subdivided or combined into blocks during initialization.
  - Not all disks have their tracks divided into sectors.
  - Blocks are separated by fixed-size **interblock gaps**, which include specially coded control information written during disk initialization.
    - This information is used to determine which block on the track follows each interblock gap.

6.1. Disk Storage

- Magnetic disks
  - Transfer of data between main memory and disk takes place in units of disk blocks.
  - A disk is a *random access* addressable device.
  - The **hardware address** of a block = a combination of a *cylinder number*, *track number* (surface number within the cylinder on which the track is located), and *block number* (within the track)
  - For a **read** command, the disk block is copied into the buffer; whereas for a **write** command, the contents of the buffer are copied into the disk block.

6.1. Disk Storage

- Magnetic disks
  - The device that holds the disks is referred to as a **hard disk drive**.
  - A disk or disk pack is mounted in the disk drive, which includes a motor that rotates the disks.
  - Disk packs with multiple surfaces are controlled by several **read/write heads**—one for each surface.
    - Disk units with an actuator are called **movable-head disks**.
    - Disk units have **fixed read/write heads**, with as many heads as there are tracks.
  - A **read/write head** includes an electronic component attached to a **mechanical arm**.
  - All arms are connected to an **actuator** attached to another electrical motor, which moves the read/write heads together and positions them precisely over the cylinder of tracks specified in a block address.
  - Once the read/write head is positioned on the right track and the block specified in the block address moves under the read/write head, the electronic component of the read/write head is activated to transfer the data.

6.1. Disk Storage

- Magnetic disks
  - A **disk controller**, typically embedded in the disk drive, controls the disk drive and interfaces it to the computer system.
  - The controller accepts high-level I/O commands and takes appropriate action to position the arm and causes the read/write action to take place.
  - Locating data on disk is a *major bottleneck* in database applications.
  - *Minimizing the number of block transfers* is needed to locate and transfer the required data from disk to main memory.

6.1. Physical Data Storage

- Storage devices
  - Magnetic disks for large amounts of data
    - Disk pack → cylinder → track → sector → bit
    - Bit → byte → block → track → cylinder
    - **Block**: data transfer unit between disks and main memory
    - Block address = block *pointer*:

cylinder number + track number + block number

$\underbrace{\text{cylinder number}}_{\text{Move read/ write heads}} + \underbrace{\text{track number}}_{\text{Activate the corresponding read/ write head}} + \underbrace{\text{block number}}_{\text{Rotate the disk pack}}$

6.1. Physical Data Storage

- Placing file records on disk
  - Data in a database is regarded as a set of records organized into a set of files.
  - Data is usually stored in the form of **records**.
    - Each record consists of a collection of related data **values** or **items**, where each value is formed of one or more bytes and corresponds to a **field** of the record.
    - A **data type**, associated with each field, specifies the types of values a field can take.
    - Records usually describe entities and their attributes.
  - A collection of field names and their corresponding data types constitutes a **record type**.
  - A **file** is a *sequence* of records.

6.1. Physical Data Storage

- Fixed-length records
  - The same record type, the same size
- Variable-length records
  - The same type, variable-length field(s)
    - Special **separator** characters (such as ? or % or $)—which do not appear in any field value—to terminate variable-length fields
  - The same type, repeating field(s)
  - The same type, optional field(s)
  - Different record types with different sizes

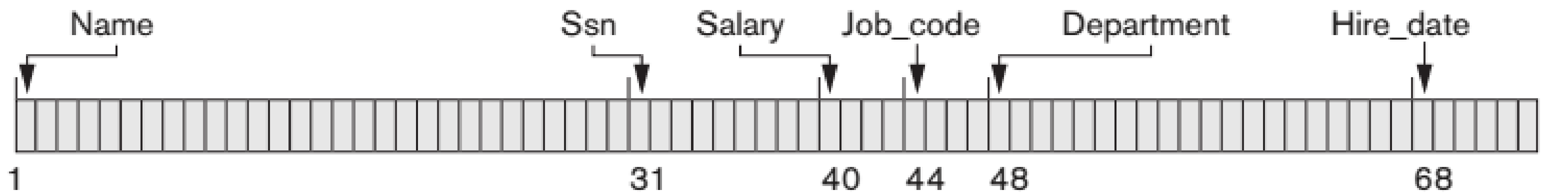
6.1. Physical Data Storage

```
struct employee {
        char    name[30];           //30 bytes
        char    ssn[9];             //9 bytes
        int     salary;             //4 bytes
        int     job_code;           //4 bytes
        char    department[20];     //20 bytes
};
```

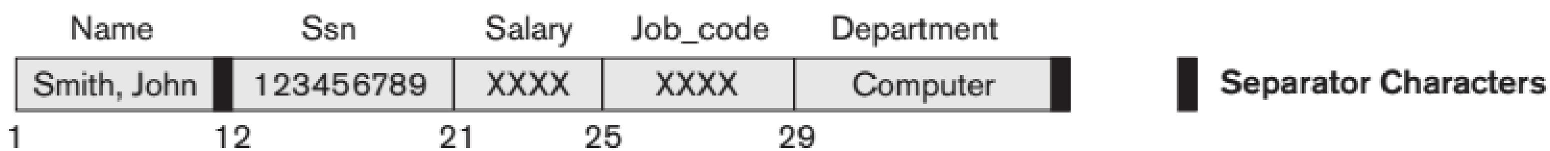
```
CREATE TABLE employee (
        name          VARCHAR2(30),    //30 bytes
        ssn           VARCHAR2(9),     //9 bytes
        salary        NUMBER,          //22 bytes
        job_code      NUMBER,          //22 bytes
        department    VARCHAR2(20)     //20 bytes
);
```

6.1. Physical Data Storage

(a) A fixed-length record with six fields and size of 71 bytes.



(b) A record with two variable-length fields and three fixed-length fields.



(c) A variable-field record with three types of separator characters.

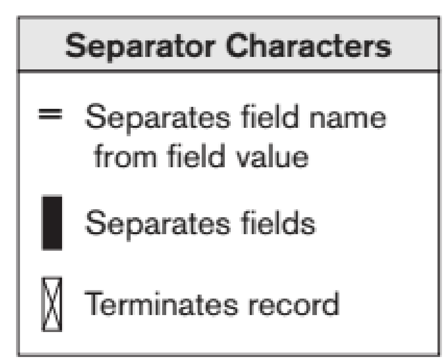


Figure 16.5, pp. 562, [1]

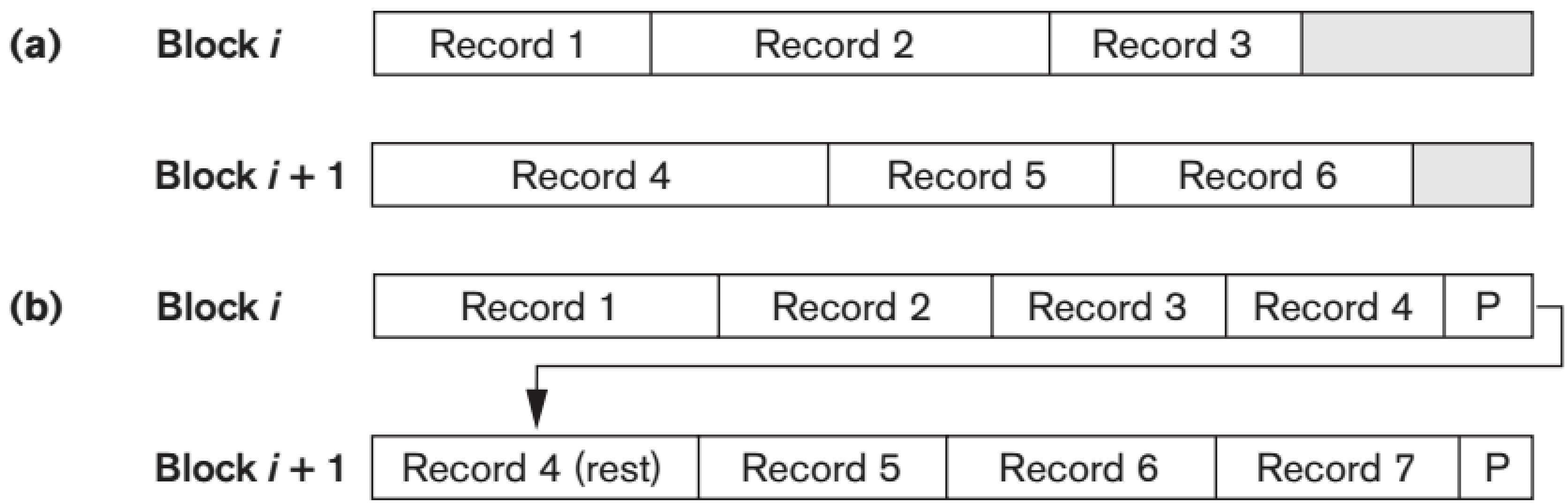
6.1. Physical Data Storage

- Unspanned records
  - Records are not allowed to cross block boundaries.
    - Fixed-length records
    - $R \leq B$ where R: record size, B: block size
- Spanned records
  - Records can span more than one block.
    - Help reducing the lost space in each block

→ How many records can be stored in a block?

→ Blocking factor

6.1. Physical Data Storage



Types of record organization.

(a) Unspanned.

(b) Spanned.

Figure 16.6, pp. 563, [1]

6.1. Physical Data Storage

- Blocking factor (*bfr*)
  - The average number of records per block for a file
  - Fixed-length records of size *R* bytes, with *B*≥*R*, using unspanned organization

$$bfr = \lfloor B/R \rfloor \textbf{ records/block}$$

    - Unused space in each block = *B* – *bfr***R* bytes
  - Variable-length records using (un)spanned organization

$$bfr = \left\lfloor \frac{The\ total\ number\ r\ of\ records}{The\ number\ b\ of\ blocks} \right\rfloor \text{ records/block}$$

→ use *bfr* to calculate the number of blocks *b* needed for a file of *r* records $b = \left\lceil \frac{r}{bfr} \right\rceil$ blocks

6.1. Physical Data Storage

- Placing file blocks on disk
  - In **contiguous allocation**, the file blocks are allocated to consecutive disk blocks.
    - Double buffering
  - In **linked allocation**, each file block contains a pointer to the next file block.
  - A combination of the two allocates **clusters** (**file segments** or **extents**) of consecutive disk blocks, and the clusters are linked.
  - In **indexed allocation**, one or more **index blocks** contain pointers to the actual file blocks.

6.1. Physical Data Storage

- Actual operations for locating and accessing file records vary from system to system.
- Representative operations
  - Open
  - Close
  - Reset
  - Record-at-a-time operations
    - Find (or Locate), Read (or Get), FindNext, Delete, Modify, Insert
  - Set-at-a-time operations
    - FindAll, Find (or Locate) *n*, FindOrdered, Reorganize

6.1. Physical Data Storage

- A **file organization**: the organization of the data of a file into records, blocks, and access structures
  - The way that records and blocks are placed on the storage medium and interlinked
  - The *goal* of a good file organization is to avoid linear search or full scan of the file and to locate the block that contains a desired record with a *minimal number of block transfers*.
- An **access method** provides a group of operations that can be applied to a file.
- **Static** files vs. **Dynamic** files
  - How frequently is a file updated?

6.1. Physical Data Storage

- Data File of a Database
  - Bit → byte → field → record → file => disk blocks
    - Blocking factor: the number of records/block
  - Primary file organization for records of one type
    - Unordered (heap) files
    - Ordered (sequential) files
    - Hash files

Unordered Files

- Unordered files = Heap files = Pile files
  - Records are placed in the file in the order in which they are inserted.
  - New records are inserted at the end of the file.
  - ***Searching*** for a record using any search condition involves a **linear search** through the file block by block—an *expensive* procedure.
    - For a file of b blocks, this requires searching (b/2) blocks, on average.
    - If *no* records or several records satisfy the search condition, the program must read and search all b blocks in the file.

Unordered Files

Employee's data

ID	Name	Salary	Department	Experiences	Deletion_marker
12	Peter	2000	D1	E5	0
35	Jane	1000	D3	E2	0
9	Mary	2500	D2	E1	0
10	Smith	1500	D1	E3	0
24	John	1800	D5	E0	0
15	Ann	2000	D2	E2	0
8	Daisy	1900	D4	E3	0
1	Alice	2100	D3	E5	1
7	Brown	2500	D3	E5	0
53	White	2300	D2	E5	0
28	Mike	1600	D1	E1	1
3	Beth	2000	D5	E0	0
36	Lily	2300	D4	E2	0

Unordered Files

Employee's data

ID	Name	Salary	Department	Experiences	Deletion_marker
12	Peter	2000	D1	E5	0
35	Jane	1000	D3	E2	0
9	Mary	2500	D2	E1	0
10	Smith	1500	D1	E3	0
24	John	1800	D5	E0	0
15	Ann	2000	D2	E2	0
8	Daisy	1900	D4	E3	0
1	Alice	2100	D3	E5	1
7	Brown	2500	D3	E5	0
53	White	2300	D2	E5	0
28	Mike	1600	D1	E1	1
3	Beth	2000	D5	E0	0
36	Lily	2300	D4	E2	0

?

Get details of Ann?

15	Ann	2000	D2	E2	0

Unordered Files

Employee's data

ID	Name	Salary	Department	Experiences	Deletion_marker
12	Peter	2000	D1	E5	0
35	Jane	1000	D3	E2	0
9	Mary	2500	D2	E1	0
10	Smith	1500	D1	E3	0
24	John	1800	D5	E0	0
15	Ann	2000	D2	E2	0
8	Daisy	1900	D4	E3	0
1	Alice	2100	D3	E5	1
7	Brown	2500	D3	E5	0
53	White	2300	D2	E5	0
28	Mike	1600	D1	E1	1
3	Beth	2000	D5	E0	0
36	Lily	2300	D4	E2	0

?

New employee

13	David	3000	D2	E5

Unordered Files

Employee's data

ID	Name	Salary	Department	Experiences	Deletion_marker
12	Peter	2000	D1	E5	0
35	Jane	1000	D3	E2	0
9	Mary	2500	D2	E1	0
10	Smith	1500	D1	E3	0
24	John	1800	D5	E0	0
15	Ann	2000	D2	E2	0
8	Daisy	1900	D4	E3	0
1	Alice	2100	D3	E5	1
7	Brown	2500	D3	E5	0
53	White	2300	D2	E5	0
28	Mike	1600	D1	E1	1
3	Beth	2000	D5	E0	0
36	Lily	2300	D4	E2	0
13	*David*	*3000*	*D2*	*E5*	*0*

?

New employee	13	David	3000	D2	E5

Unordered Files

Employee's data

ID	Name	Salary	Department	Experiences	Deletion_marker
12	Peter	2000	D1	E5	0
35	Jane	1000	D3	E2	0
9	Mary	2500	D2	E1	0
10	Smith	1500	D1	E3	0
24	John	1800	D5	E0	0
15	Ann	2000	D2	E2	0
8	Daisy	1900	D4	E3	0
1	Alice	2100	D3	E5	1
7	Brown	2500	D3	E5	0
53	White	2300	D2	E5	0
28	Mike	1600	D1	E1	1
3	Beth	2000	D5	E0	0
36	Lily	2300	D4	E2	0

? ?

Remove details of Ann?

15	Ann	2000	D2	E2	1

Unordered Files

Employee's data

ID	Name	Salary	Department	Experiences	Deletion_marker
12	Peter	2000	D1	E5	0
35	Jane	1000	D3	E2	0
9	Mary	2500	D2	E1	0
10	Smith	1500	D1	E3	0
24	John	1800	D5	E0	0
15	Ann	2000	D2	E2	0
8	Daisy	1900	D4	E3	0
1	Alice	2100	D3	E5	1
7	Brown	2500	D3	E5	0
53	White	2300	D2	E5	0
28	Mike	1600	D1	E1	1
3	Beth	2000	D5	E0	0
36	Lily	2300	D4	E2	0

? ?

Update details of Daisy?

8	Daisy	1900	D4	E4	0

Unordered Files

- Unordered files = Heap files = Pile files
  - ***Inserting*** a new record is *very efficient.*
    - The last disk block of the file is copied into a buffer, the new record is added, and the block is then rewritten back to disk.
  - To ***delete*** a record, a program must first find its block, copy the block into a buffer, delete the record from the buffer, and finally rewrite the block back to the disk.=> reorganization
  - For ***deletion***, an extra byte or bit, a **deletion marker**, is stored with each record. A record is deleted by setting the deletion marker to a certain value. => reorganization
  - For ***modifying*** a fixed-length record, a program must first find its block, copy the block into a buffer, modify the record from the buffer, and finally rewrite the block back to the disk.
  - ***Modifying*** a variable-length record may require deleting the old record and inserting a modified record because the modified record may not fit in its old space on disk.

Ordered Files

- Ordered files = Sorted files = Sequential files
  - The records of a file are physically ordered on disk based on the values of one of their fields—called the **ordering field**.
  - If the ordering field is also a **key field** of the file, the field is called the **ordering key** for the file.
    - A *key field* is a field guaranteed to have a unique value in each record.
  - Reading the records in order of the ordering field values is efficient because no sorting is required.
  - Ordered files are blocked and stored on contiguous cylinders to minimize the seek time.

Ordered Files

Employee's data

ID	Name	Salary	Department	Experiences	Deletion_marker
2	Peter	2000	D1	E5	0
5	Jane	1000	D3	E2	0
9	Mary	2500	D2	E1	0
10	Smith	1500	D1	E3	0
14	John	1800	D5	E0	0
15	Ann	2000	D2	E2	0
17	Daisy	1900	D4	E3	0
19	Alice	2100	D3	E5	1
21	Brown	2500	D3	E5	0
23	White	2300	D2	E5	0
28	Mike	1600	D1	E1	1
30	Beth	2000	D5	E0	0
36	Lily	2300	D4	E2	0

Ordered by column ID => ordering field

Ordered Files

- Ordered files = Sorted files = Sequential files
  - ***Search*** with a search criterion involving the conditions >, <, ≥, and ≤ on the *ordering* field is efficient using *binary search*.
    - At least $\log_2(b)$ block accesses
  - ***Search*** with a search criterion on other *non-ordering* fields or *other* search criteria is done with a *linear search* for random access.
    - At least b/2 block accesses

Ordered Files

Employee's data

ID	Name	Salary	Department	Experiences	Deletion_marker
2	Peter	2000	D1	E5	0
5	Jane	1000	D3	E2	0
9	Mary	2500	D2	E1	0
10	Smith	1500	D1	E3	0
14	John	1800	D5	E0	0
15	Ann	2000	D2	E2	0
17	Daisy	1900	D4	E3	0
19	Alice	2100	D3	E5	1
21	Brown	2500	D3	E5	0
23	White	2300	D2	E5	0
28	Mike	1600	D1	E1	1
30	Beth	2000	D5	E0	0
36	Lily	2300	D4	E2	0

?

New employee	13	David	3000	D2	E5

Ordered Files

- Ordered files = Sorted files = Sequential files
  - Inserting and deleting records are expensive because the records must remain physically ordered.
  - One frequently used ***insertion*** method
    - The actual ordered file (called *main* or *master* file) for binary search on ordering field values
    - A temporary unordered file (called *overflow* or *transaction* file) for inserting new records at the end
    - File reorganization for periodically sorting and merging the overflow file with the master file
  - For record ***deletion***, deletion markers and periodic reorganization are used.

Ordered Files

- ***Modifying*** a field value of a record depends on two factors: the search condition to locate the record and the field to be modified.
- If the search condition involves the ordering key field, we can locate the record using a binary search; otherwise we must do a linear search.
- A non-ordering field can be modified by changing the record and rewriting it in the same physical location on disk—assuming fixed-length records.
- Modifying the ordering field means that the record can change its position in the file.
  - This requires deletion of the old record followed by insertion of the modified record.

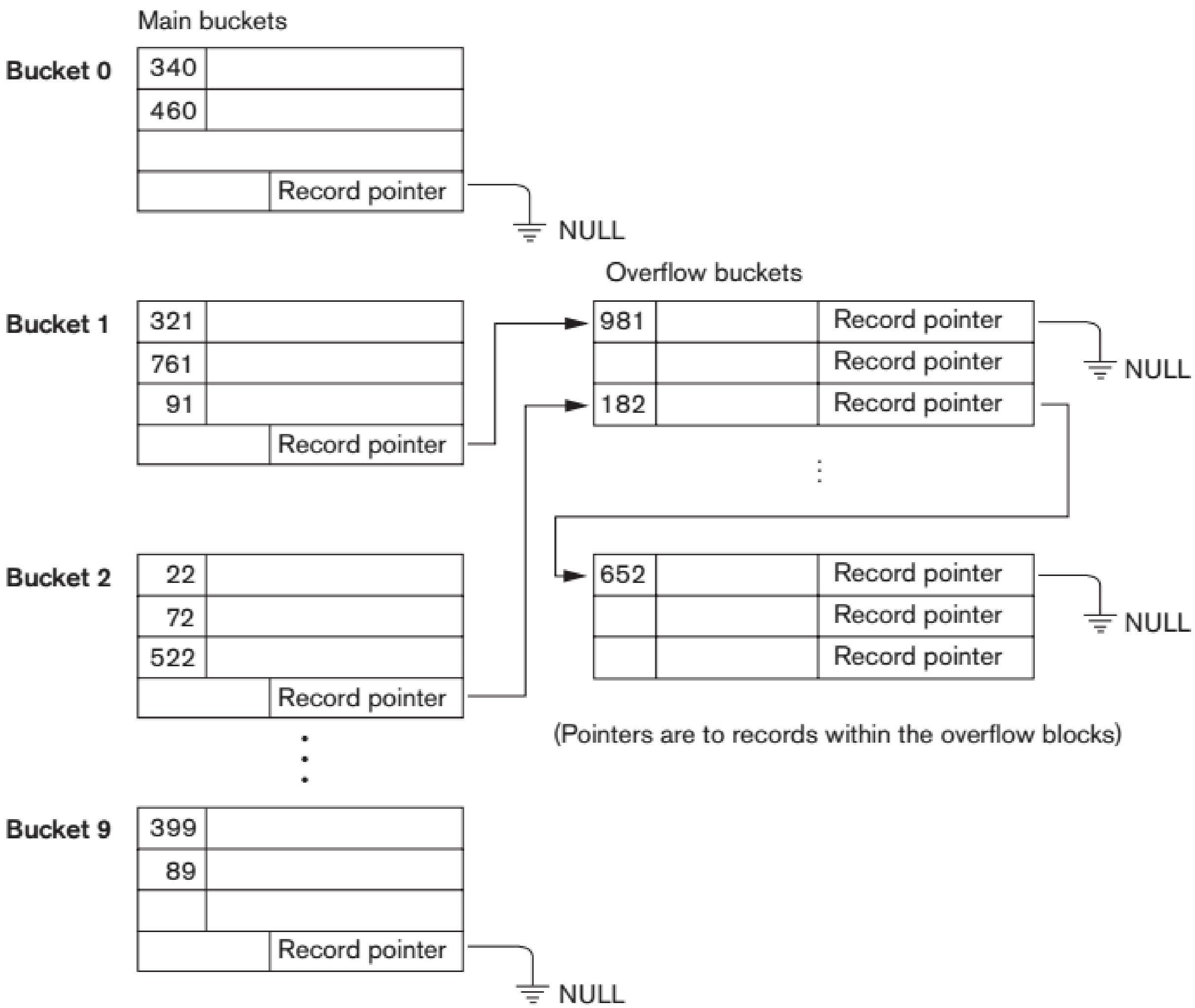
Hash Files

- Hashing for files is called *external hashing*.
- The target address space is made of buckets.
  - A bucket is either one disk block or a cluster of contiguous disk blocks.
  - A bucket holds multiple records.
- The hash function maps a key into a *relative bucket number* rather than assigning an absolute block address to the bucket.
- Files stored on disk by hashing are *hash files*.
- A field used in a hash function is called the *hash field*. If a key field, it is called the *hash key*.

Hash Files

- The collision problem: a variation of chaining in which a pointer is maintained in each bucket to a linked list of overflow records for the bucket
  - ***Record pointer*** = a block address + a relative record position within the block
- A choice of a good hash function
  - To distribute the records uniformly over the address space to minimize collisions, thus making it possible to locate a record with a given key in a single access
  - To achieve the buckets fully, thus not leaving many unused locations: 70%-90% full

Hash Files



Handling overflow for buckets by chaining

Figure 16.10, pp. 576, [1]

Address space:

M buckets (=10)

Hash function:

$h(K) = K \bmod M$

How to store the following records in this hash file: 32, 179?

Are the following records: 81, 652 in this file?

Hash Files

- ***Search*** with the *equality* condition on the *hash field* is efficient by using the hash function.
  - At least one block access !!!
- ***Search*** with other search conditions on the hash field or with any search conditions on other non-hash fields are not efficient by using a *linear search*.
  - At least $b/2$ block accesses
- ***Insertion*** can be done efficiently by using the hash function.
  - *Collision* must be handled.

Hash Files

- Record ***deletion*** can be implemented by removing the record from its bucket.
  - If the bucket has an overflow chain, we can move one of the overflow records into the bucket to replace the deleted record.
  - If the record to be deleted is already in overflow, we simply remove it from the linked list.
  - A linked list of unused overflow locations in overflow is maintained to track empty positions in overflow.

Hash Files

- ***Modifying*** a specific record's field value depends on two factors: the search condition to locate that record and the field to be modified.
  - If the search condition is an equality comparison on the hash field, we can locate the record efficiently by using the hash function; otherwise, we must do a linear search.
  - A non-hash field can be modified by changing the record and rewriting it in the same bucket.
  - Modifying the hash field may require the record to be moved to another bucket.
    - delete the old record and then, insert the modified record

Hash Files

- The hashing scheme with a fixed number of allocated buckets *M* is called **static hashing**.
  - Not suitable for dynamic files
  - *Solution (?)*: change the number of buckets *M* (smaller, larger) and redistribute the records with a new hash function based on the new value of *M*
- → Newer dynamic file organizations based on hashing allow the number of buckets to vary *dynamically* with only localized reorganization.
  - → Extendible hashing: a directory (access structure)
  - → Dynamic hashing: a binary tree (access structure)
  - → Linear hashing: a sequence of hash functions

Primary File Organization

Average Access Times for Basic File Organizations

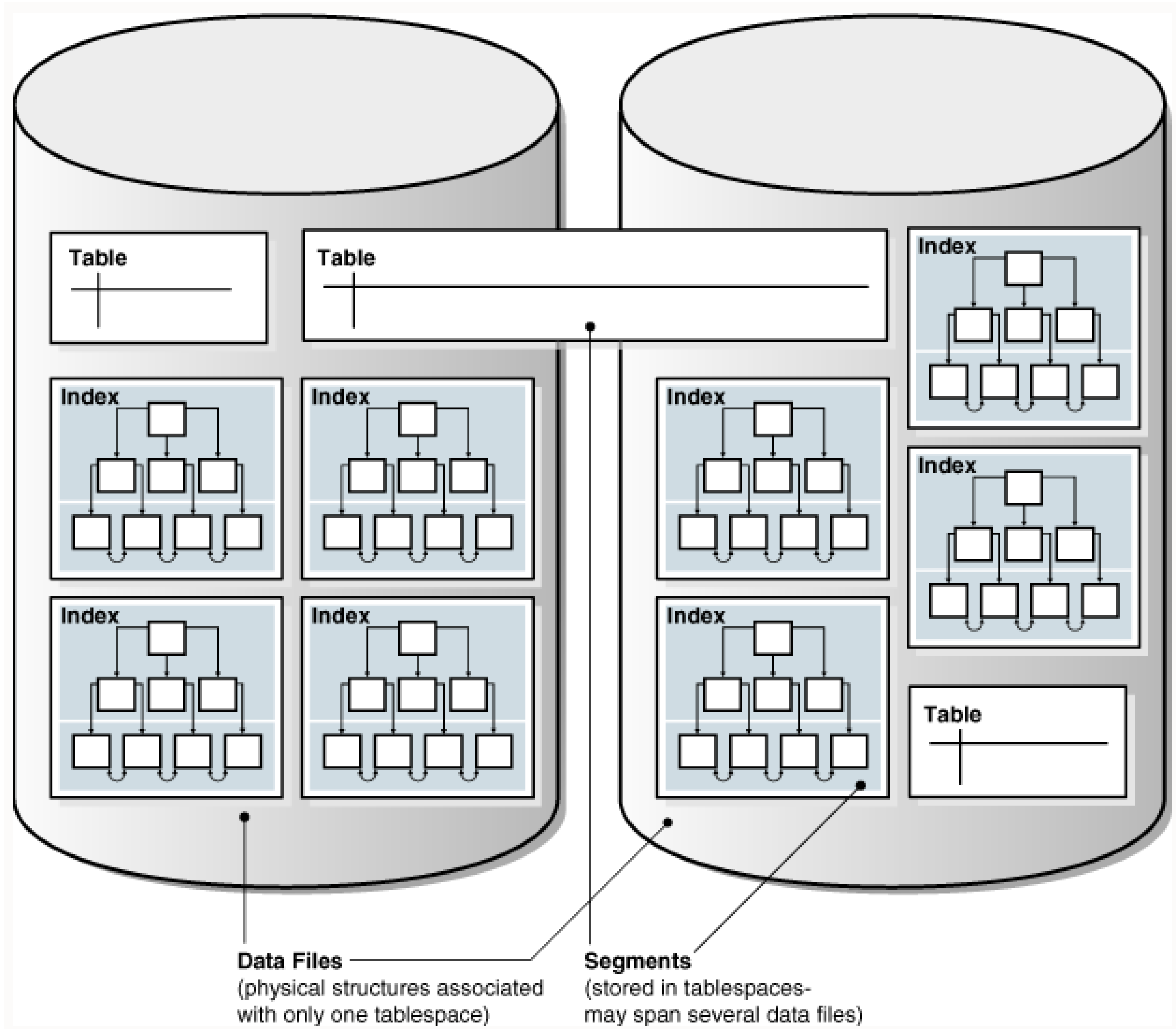
Type of organization	Access/Search method	Average time to access one specific record with "=" condition
Heap (unordered)	Sequential scan (linear search)	$b/2$
Ordered	Sequential scan	$b/2$
Ordered	Binary search	$\log_2 b$
Hash	Static hashing	~ 1
Hash	Linear (dynamic) hashing	1
Hash	Extendible (dynamic) hashing	1-2

Note: a file with *b* blocks

Physical Storage in Today's DBMSs

- Oracle 19c
  - Schema Object Storage
    - Storage structure = segment: data segment, index segment
    - Tablespace: a database storage unit that contains objects from different schemas. One tablespace is associated with one or many physical data files.
      - One segment cannot span multiple tablespaces.
      - One data segment for one table spans two data files, which are both part of the same tablespace.

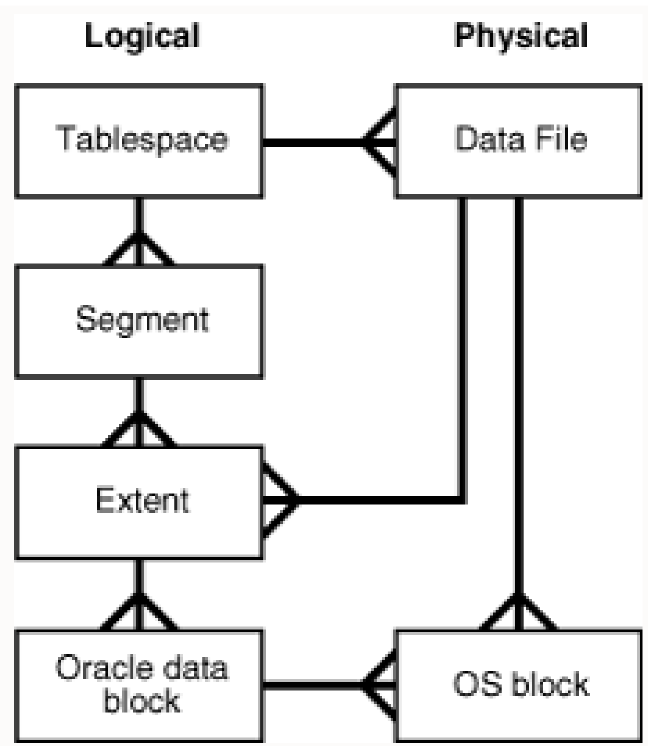
Physical Storage in Today's DBMSs

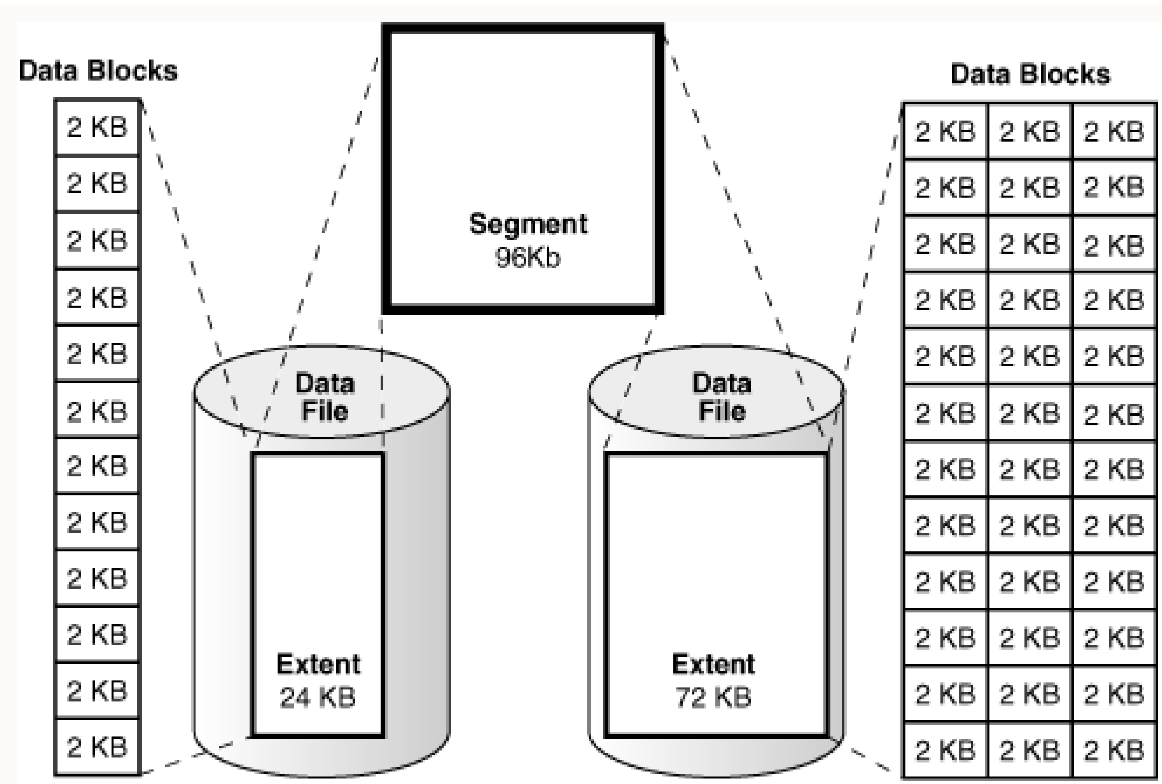


Tablespace, Segments, Data Files in Oracle 19c

Physical Storage in Today's DBMSs

- Tablespace → Segment → Extent → Oracle Data Block





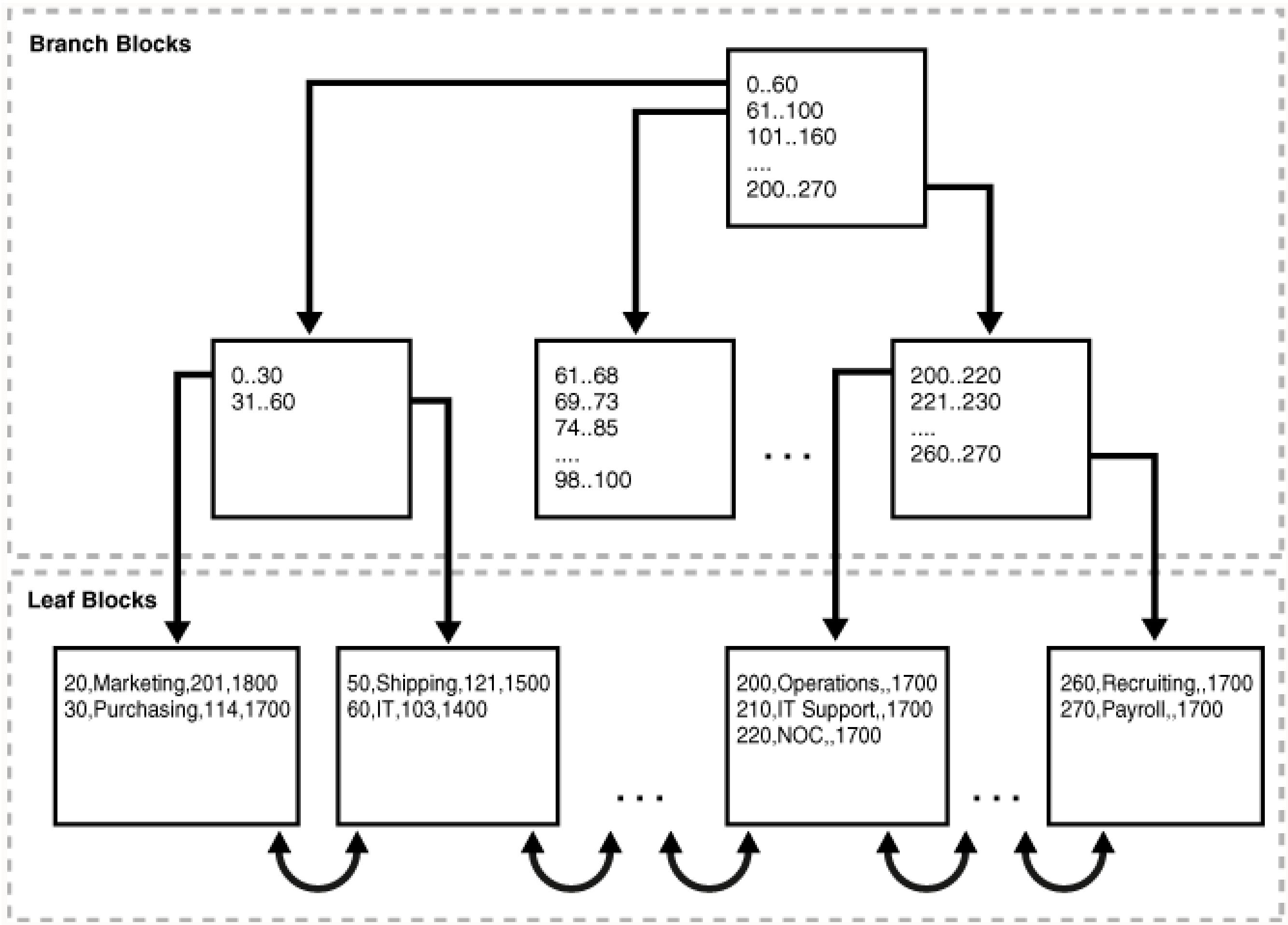
Storage in Oracle 19c

An extent is a set of logically *contiguous* data blocks allocated for storing a specific type of information

Physical Storage in Today's DBMSs

- Oracle Database tables in Oracle 19c:
  - Relational tables
    - Relational tables have simple columns and are the most common table type.
    - A *heap-organized table* does not store rows in any particular order, created by default.
    - An *index-organized table* orders rows according to the primary key values by means of B-tree indexes.
    - An *external table* is a read-only table whose metadata is stored in the database but whose data is stored outside the database.
  - Object tables
    - The columns correspond to the top-level attributes of an object type.

Physical Storage in Today's DBMSs



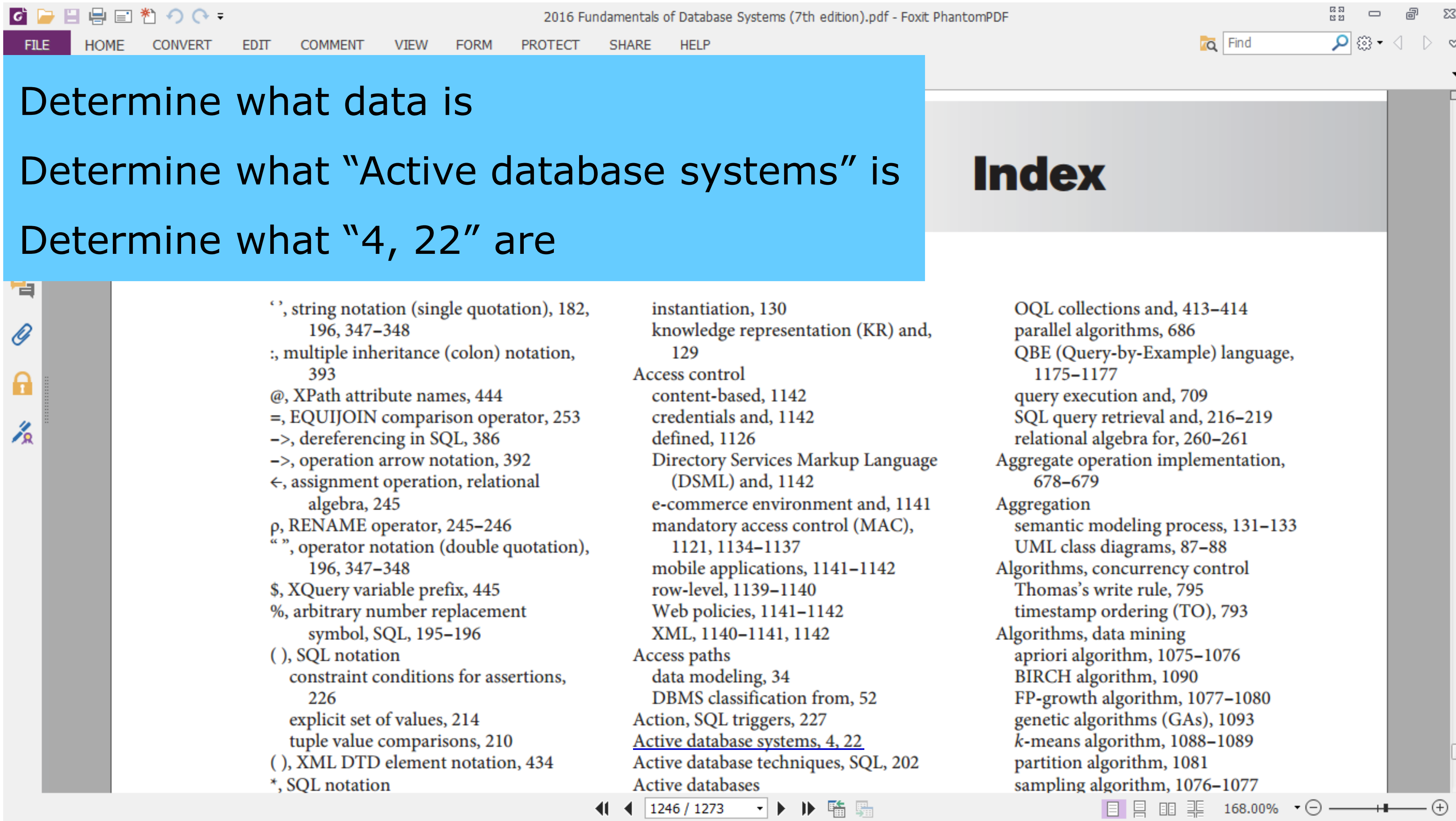
An index-organized table in Oracle 19c

6.2. Indexing …

Determine what data is

Determine what "Active database systems" is

Determine what "4, 22" are



Index

The index section in [1]

Have you ever used this section in any book?

6.2. Indexing …

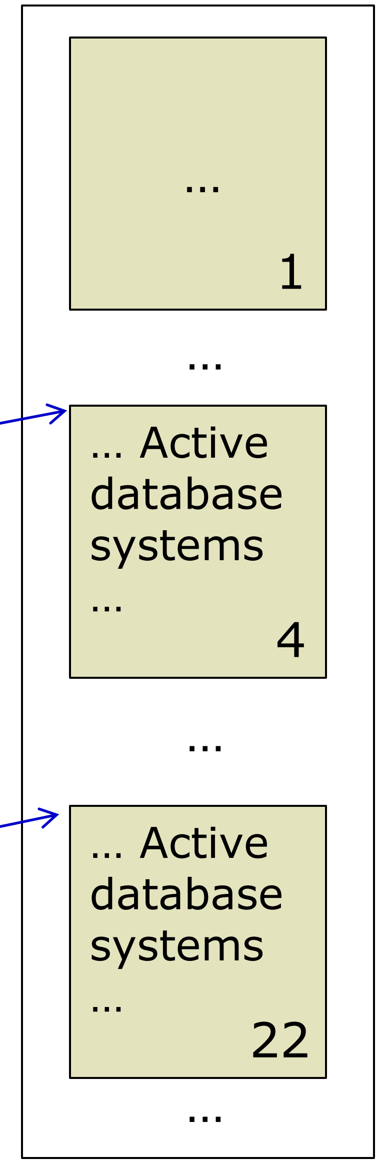
Index

Value	Page No.
‘’, string notation, …	182, 196, 347-348
…	…
Action, SQL triggers	227
Active database systems	4, 22
Active database techniques, SQL	202
…	…

Ordered indexed values

Linking values

(addressable)



Book content

6.2. Indexing …

- Assumption: a data file exists with some primary organization such as the unordered, ordered, or hashed organization.
- Indexes are *additional* auxiliary access structures of a data file.
  - Role: ***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
  - Purpose: speed up the retrieval of records in response to *certain search conditions*
  - Management: *additional* ordered files on disk

Types of Single-level Ordered Indexes

- A single-level ordered index is an access structure defined on a field of a file (or multiple fields of a file).
  - This index is a file including many entries. Each entry is <**Field value**, **Pointer**(s)>.
    - Field values: able to be ordered.
    - Pointer(s): record pointers or block pointers to the data file.
  - The field is called an **indexing field**.
  - The index file is **ordered** with the field values.
  - **Binary search** is applied on the index file with the conditions =, >, <, ≥, ≤, *between* on the indexing field.

Types of Single-level Ordered Indexes

Single-level ordered index file
(bfr_i = 4 records/block)

Field Value	(Block) Pointer
2	•
10	•
18	•
23	•
35	•

Data file of employee records
(*bfr* = 3 records/block)

SSN	Name	...	Dept
2	F	...	2
5	AA	...	5
8	U	...	4
10	C	...	2
11	L	...	2
15	EG	...	1
18	B	...	4
19	FL	...	5
21	N	...	5
23	Y	...	5
28	D	...	1
34	O	...	2
35	NM	...	3

SSN: indexing field

Types of Single-level Ordered Indexes

- The index file usually occupies considerably less disk blocks than the data file because the number of the entries in the index file is much smaller.
- A binary search on the index file yields a pointer to the file record.
- Indexes are characterized as dense or sparse.
  - A **dense index** has an index entry for *every field value* (and hence every record) in the data file.
  - A **sparse** (or **non-dense**) **index** has index entries for only some field values.
    - The previous example is a *non-dense* index.

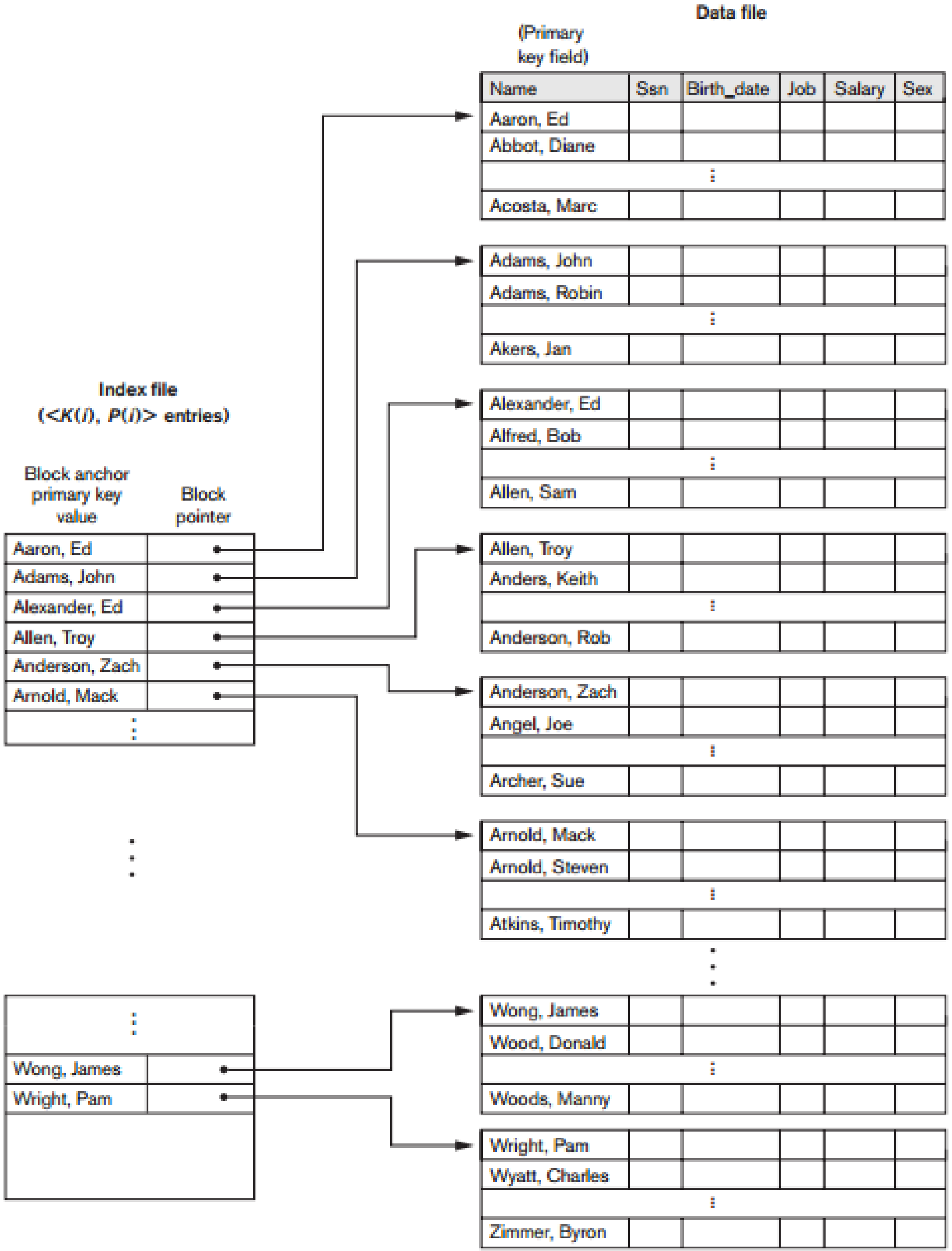
Types of Single-level Ordered Indexes

- Types of ordered indexes
  - A **primary index** is specified on the *ordering key field* of an **ordered file** of records.
  - A **clustering index** is specified on the *ordering non-key field* of an **ordered file** of records.
  - A **secondary index** is specified on any *non-ordering field* of a file of records.
  - → 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 data file can have *several* secondary indexes in addition to its primary access method.

Types of Single-level Ordered Indexes

- Primary indexes
  - 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.
  - 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: $<K(i), P(i)>$.
  - The first record in each block of the data file is called the **anchor record** of the block, or simply the **block anchor**.

Primary Indexes



- Non-dense index
- Block anchor

Types of Single-level Ordered Indexes

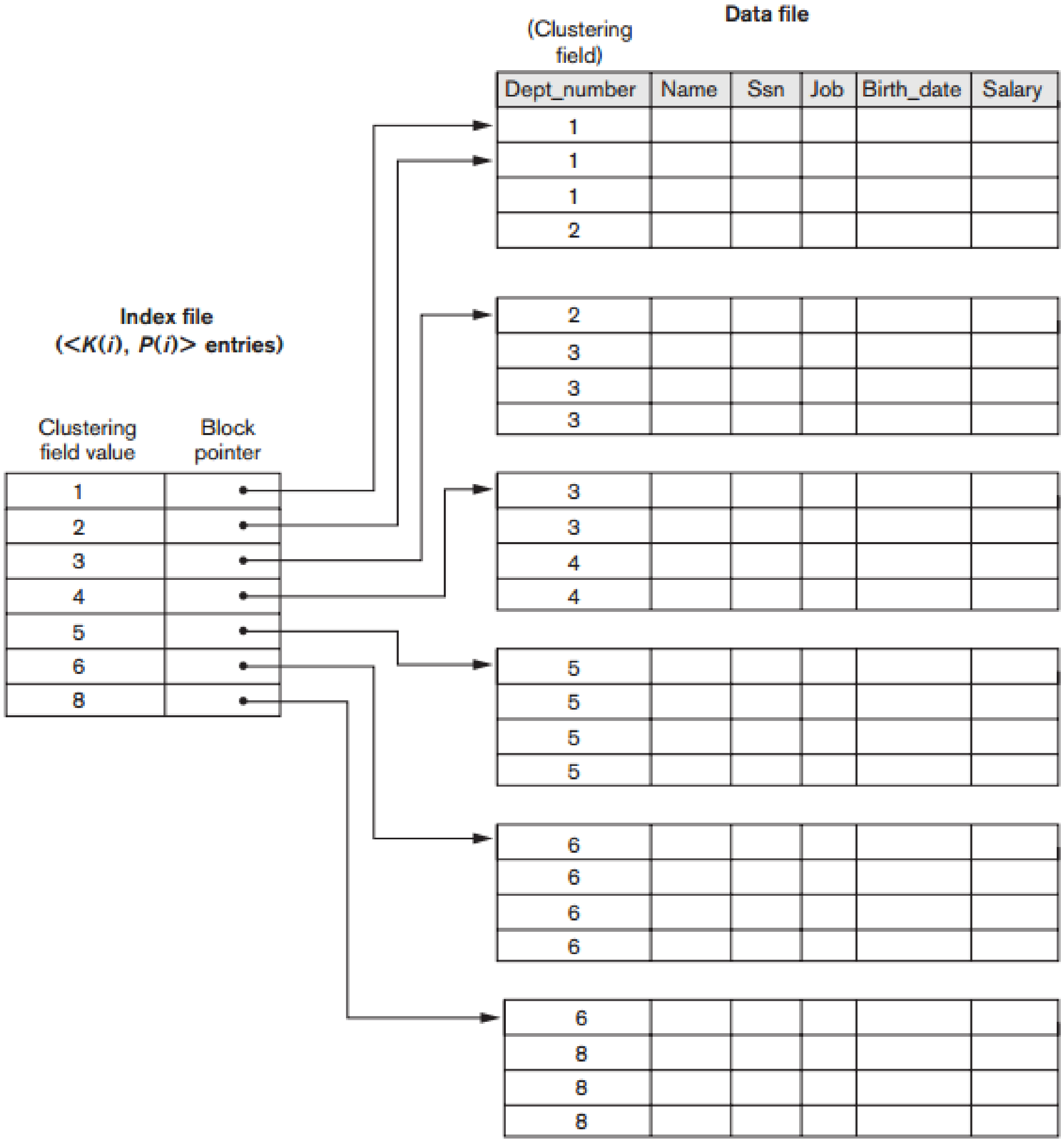
- Primary indexes
  - A primary index is a nondense (sparse) index.
    - Why?
  - The index file for a primary index occupies a much smaller space than does the data file.
    - Why?
  - Given the value *K* of its primary key field, a binary search is used on the index file to find the appropriate index entry *i*, and then retrieve the data file block whose address is *P*(*i*).
  - A major problem with a primary index is insertion and deletion of records.
    - Why and how to solve it?

Types of Single-level Ordered Indexes

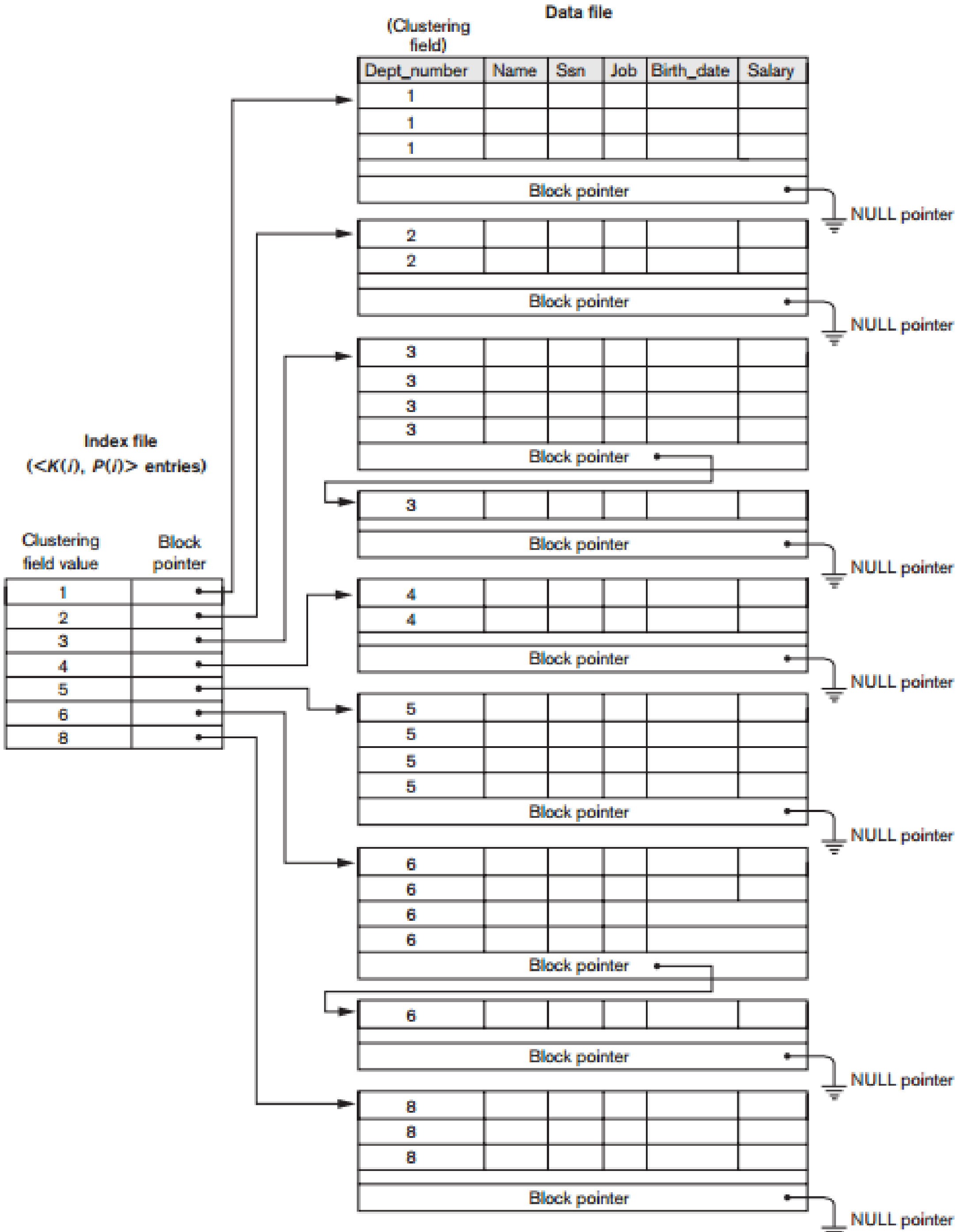
- Clustering indexes
  - Also defined on an ordered data file with an ordering *non-key* field.
  - Each index entry *for each distinct value* of the field
    - The index entry points to the first data block that contains records with that field value.
  - It is another example of *non-dense* index.
  - Record insertion and deletion cause problems.
    - Why?
  - To alleviate the problem of insertion, it is common to reserve a whole block (or a cluster of contiguous blocks) for *each value* of the clustering field.
    - All records with that value are placed in the block (or block cluster).

Clustering indexes

- Non-dense
- *No block anchor*



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



Clustering indexes

- Non-dense
- *Block anchor*

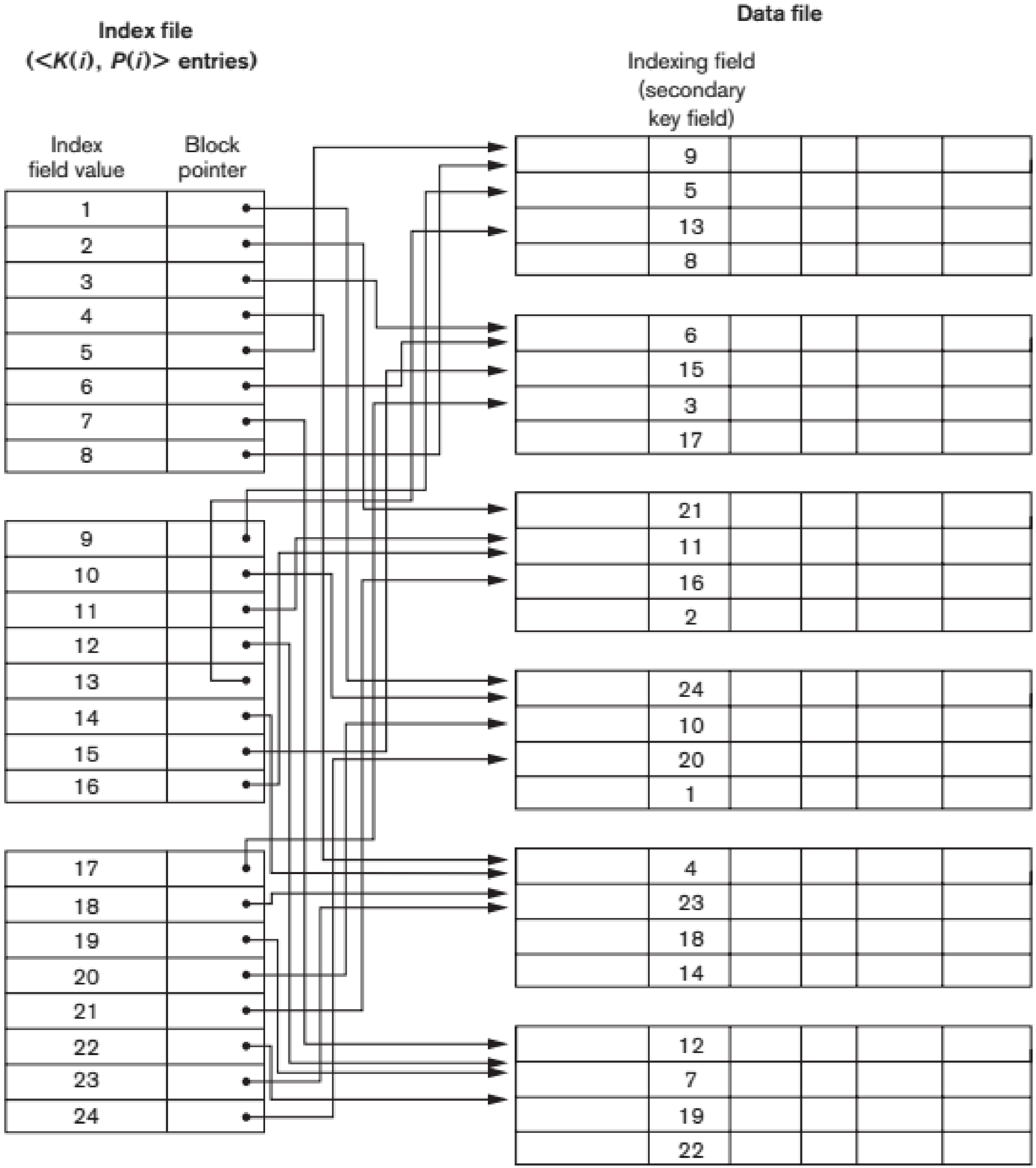
A clustering index on the Dept_number *ordering nonkey* field of an EMPLOYEE file:

a separate block cluster for each group of records that share the same value for the clustering field

Types of Single-level Ordered Indexes

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

Secondary indexes



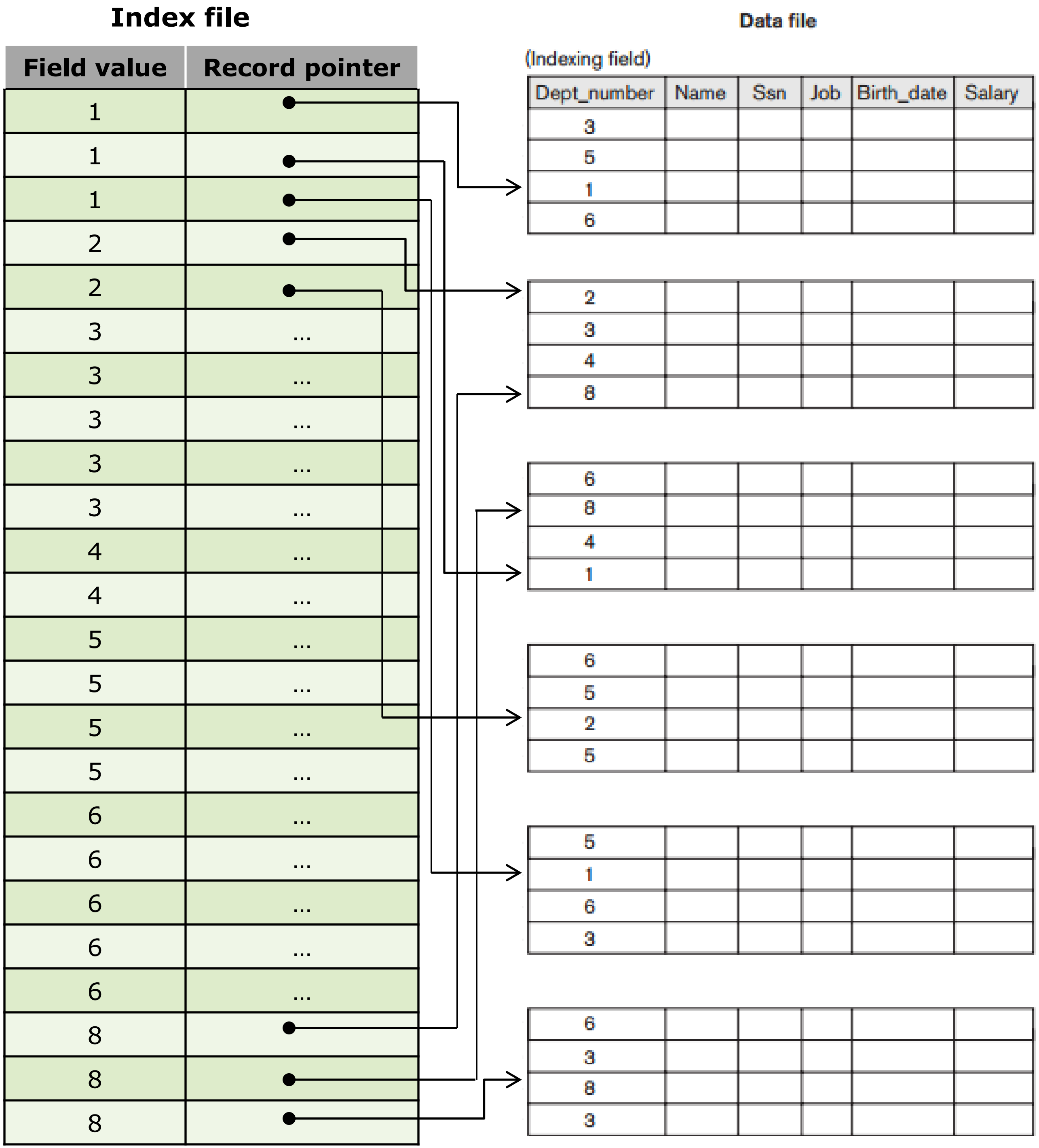
A dense secondary index (with block pointers) on a *nonordering key field* of a file

→ The secondary index provides a ***logical ordering*** of the data file.

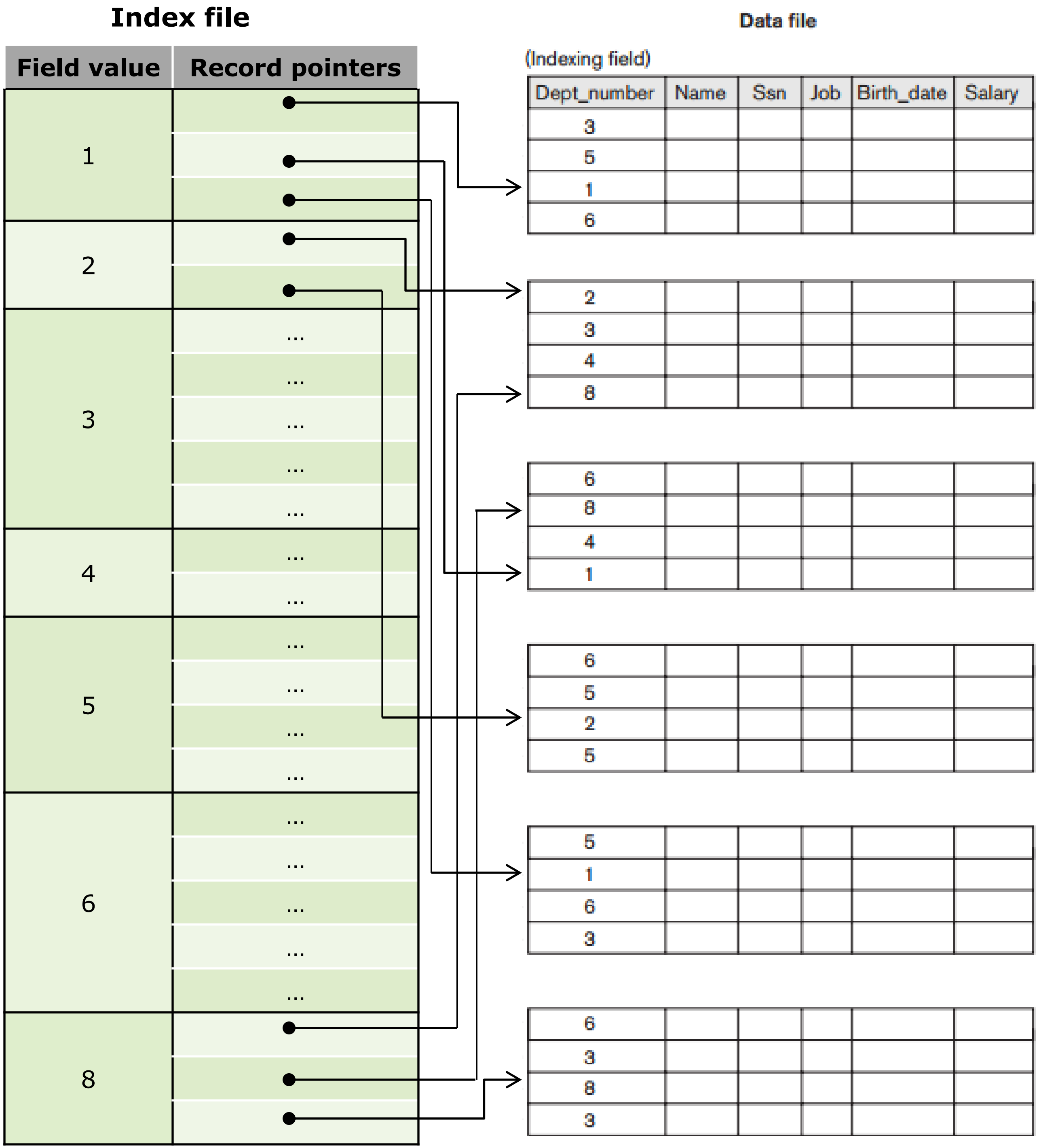
Types of Single-level Ordered Indexes

- Secondary indexes
  - A secondary index defined on a *non-ordering key* field has the number of index entries equal to the number of records in the data file.
  - A secondary index defined on a *non-ordering non-key* field can be implemented in three ways:
    - (1). Include duplicate index entries with the same $K(i)$ value —one for each record
    - (2). Use variable-length records for the index entries, with a repeating field for the pointer
      - A list of pointers <$P(i, 1), \ldots, P(i, k)$> in the index entry for $K(i)$—one pointer to each block that contains a record whose indexing field value equals $K(i)$
    - (3). Keep the index entries at a fixed length and have a single entry for each *index field value*, but to create *an extra level of indirection* to handle the multiple pointers

Secondary indexes



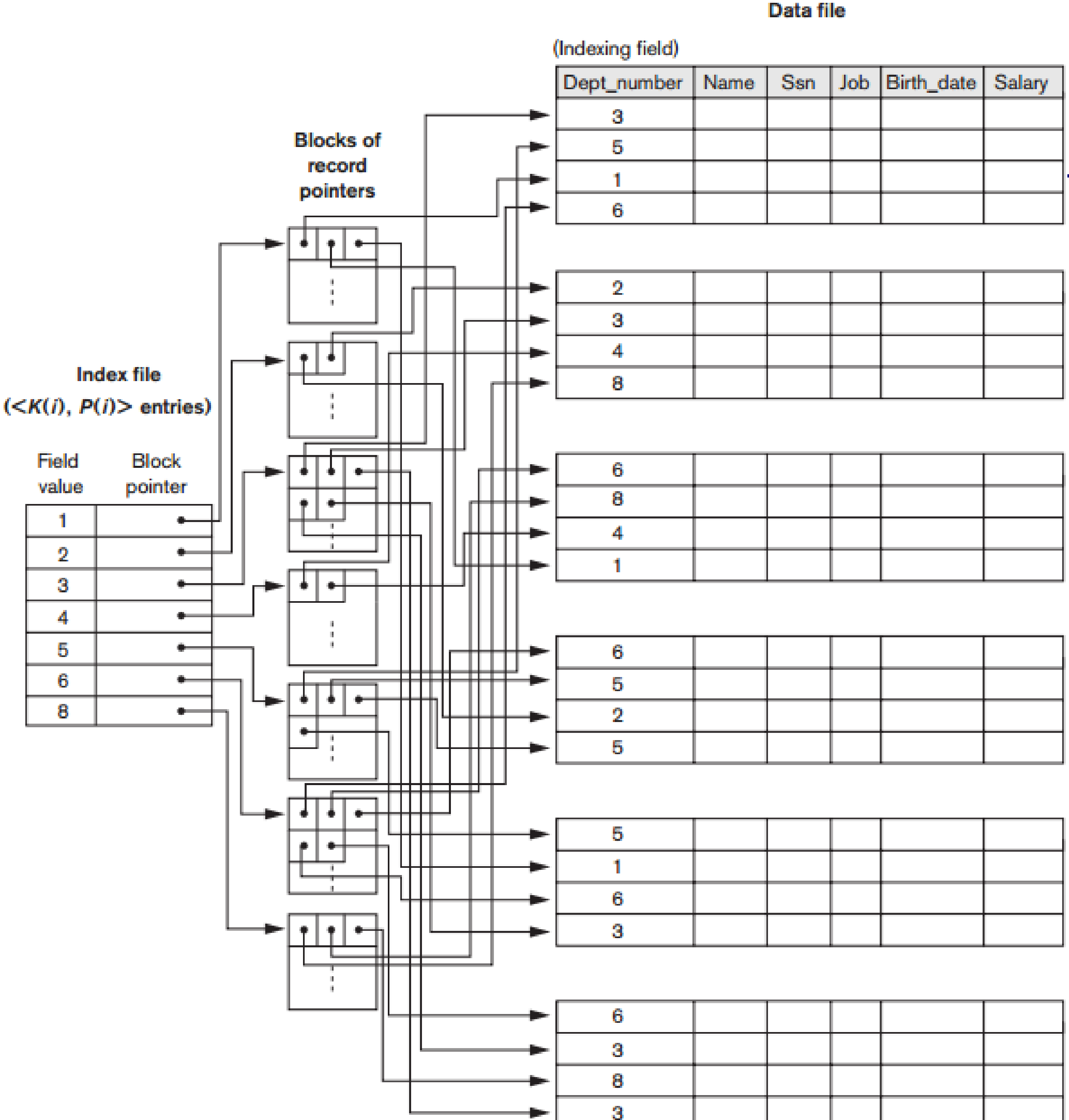
A secondary index (with record pointers) on a *nonordering nonkey field* implemented using ***option (1)***



Secondary indexes

A secondary index (with record pointers) on a ***nonordering nonkey field*** implemented using ***option (2)***

Secondary indexes



A secondary index (with record pointers) on a ***nonordering nonkey field*** implemented using one level of indirection in ***option (3)*** so that index entries are of fixed length and have unique field values.

Types of Single-level Ordered Indexes

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)

Types of Single-level Ordered Indexes

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

[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.

Multilevel Indexes

- Because a single-level index is an *ordered* file, we can create another *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 the third, fourth, ..., top level until all entries of the *top level* fit in one disk block.
- A multilevel index can be created for any type of the first-level index (primary, secondary, clustering) as long as the first-level index consists of *more than one* disk block.

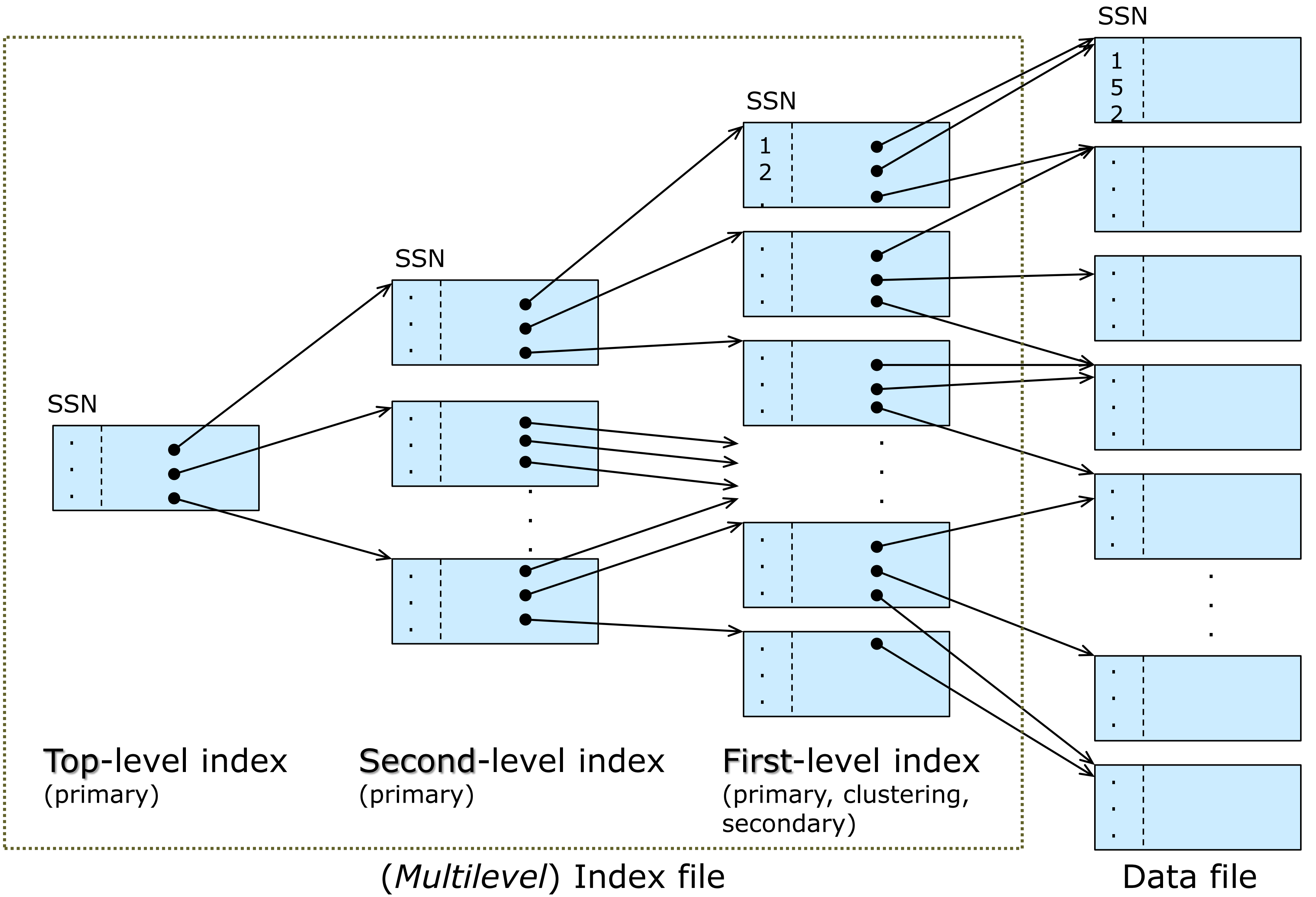
Multilevel Indexes

- A binary search is applied to *the single-level ordered index file* to locate pointers to a disk block or to a record (or records) in the file having a specific index field value.
  - A binary search requires approximately ($\log_2 b_i$) block accesses for an index with b_i blocks because each step reduces the part of the index file that we continue to search by a factor of 2.
- For much faster search, a **multilevel index** reduces the part of the index that we continue to search by bfr_i, the blocking factor for the index, called the **fan-out** *fo*.

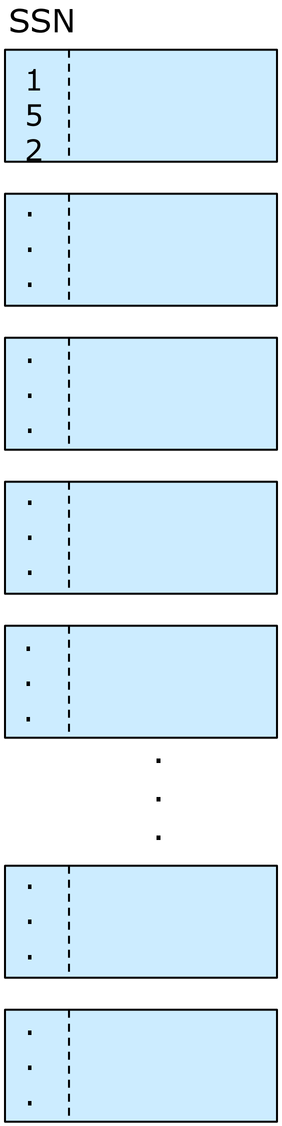
Multilevel Indexes

- We divide the *record search space* into two halves at each step during a *binary search*.
- We divide the record search space *n*-ways (where *n* = the *fan-out*) at each search step using the *multilevel index*.
- Searching a multilevel index requires approximately ($\log_{fo} b_i$) block accesses, which is a substantially smaller number than for a binary search if the fan-out is larger than 2.
  - In most cases, the fan-out is much larger than 2.

Multilevel Indexes

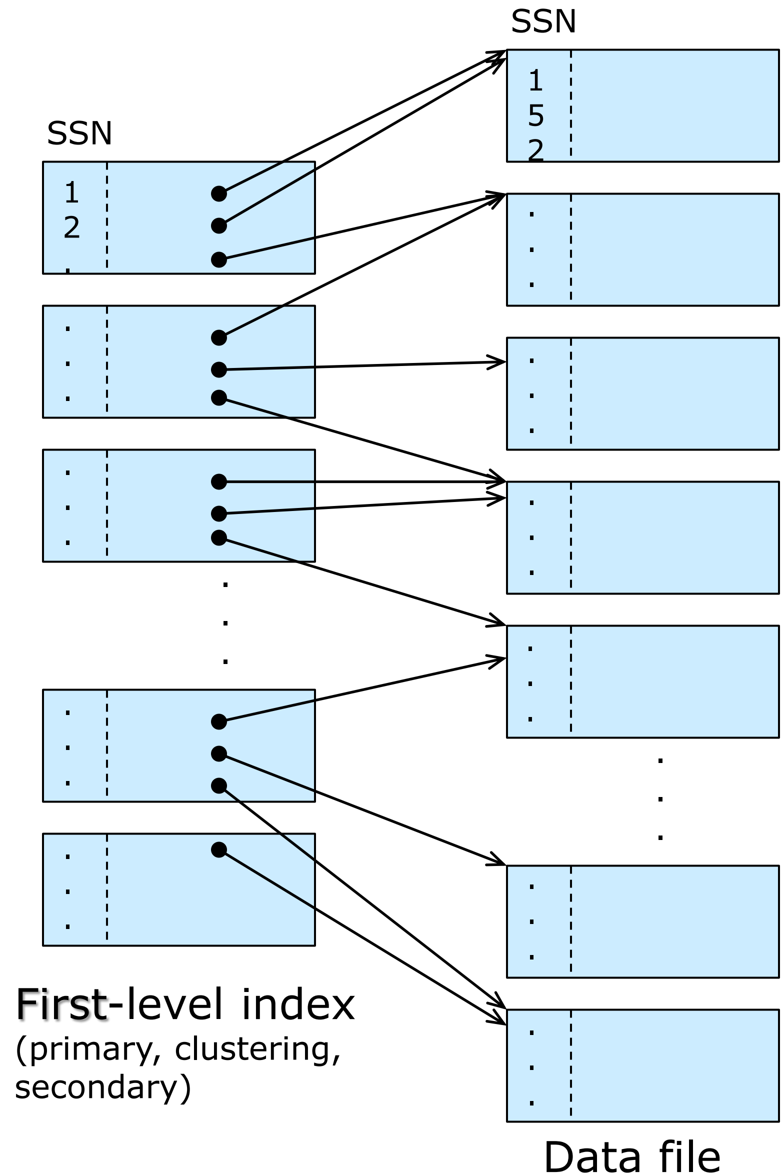


Multilevel Indexes

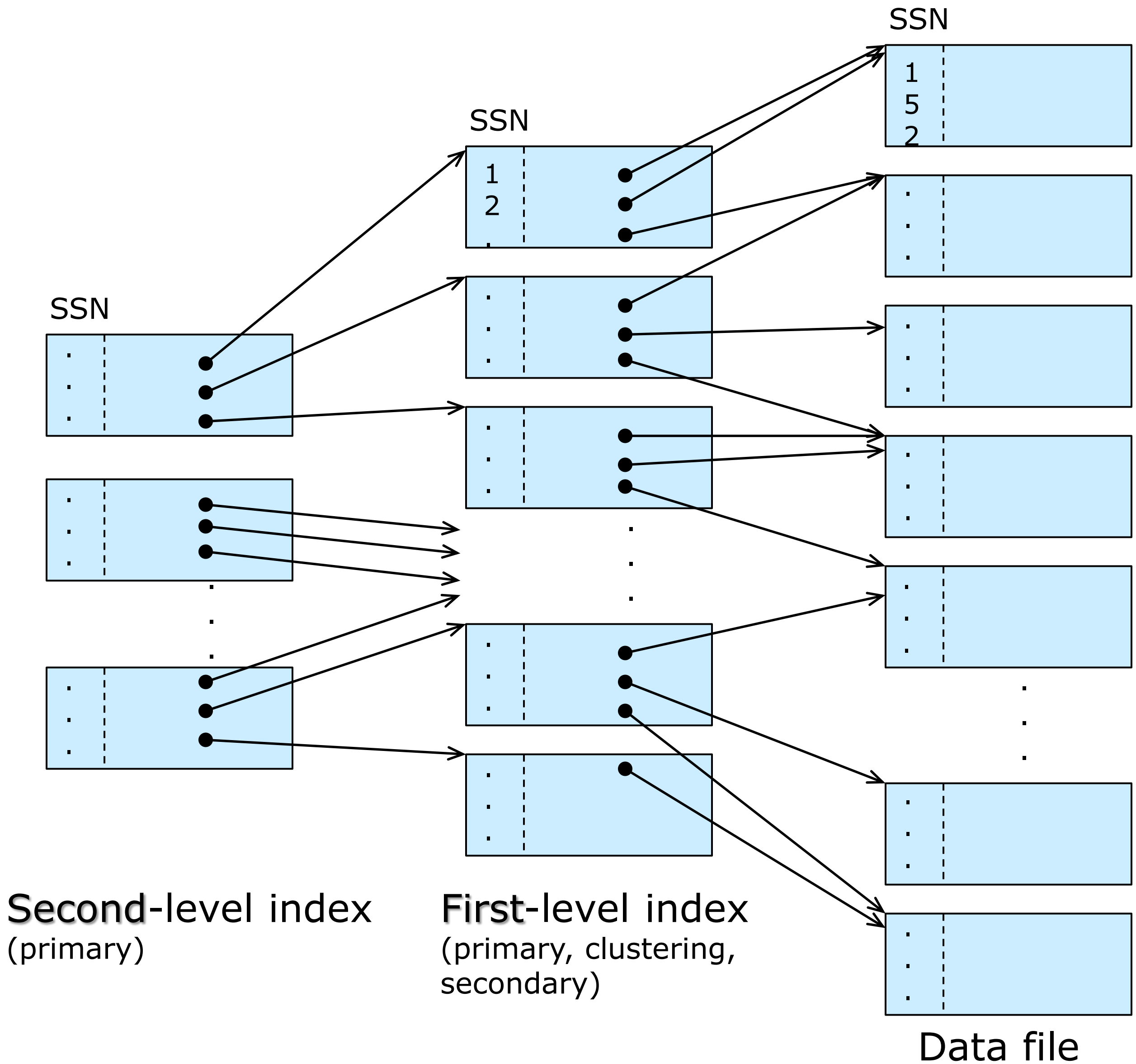


Data file

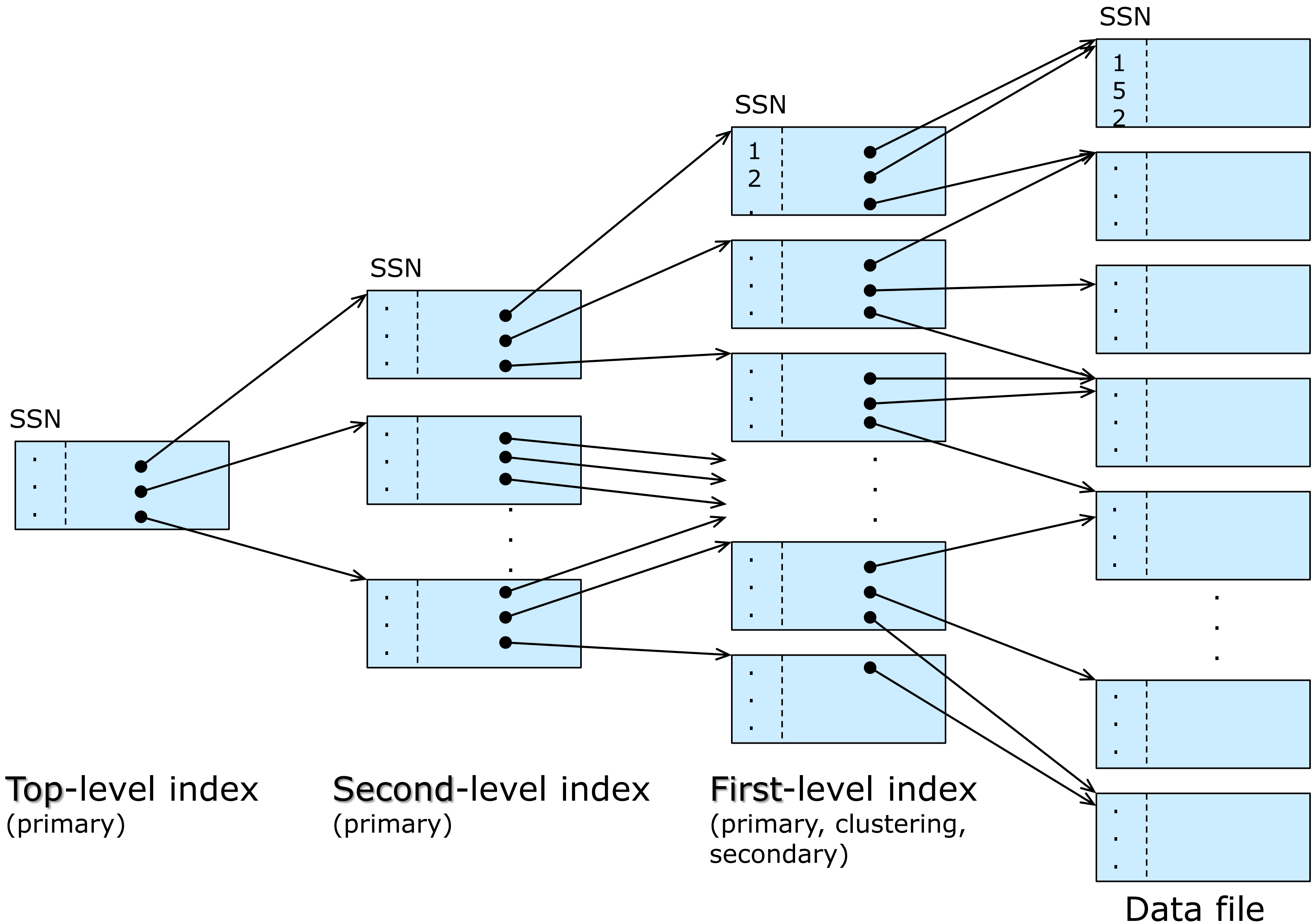
Multilevel Indexes



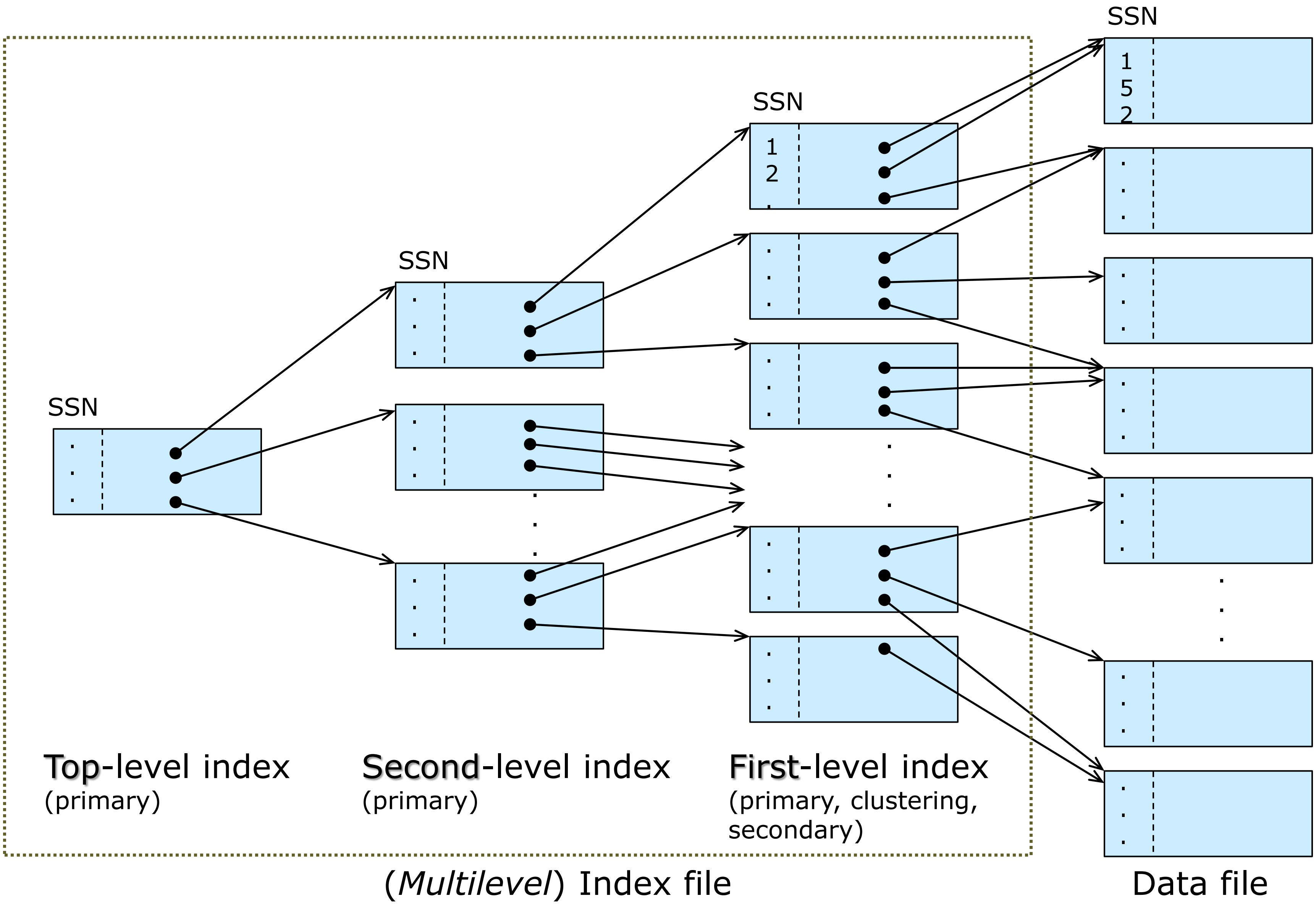
Multilevel Indexes



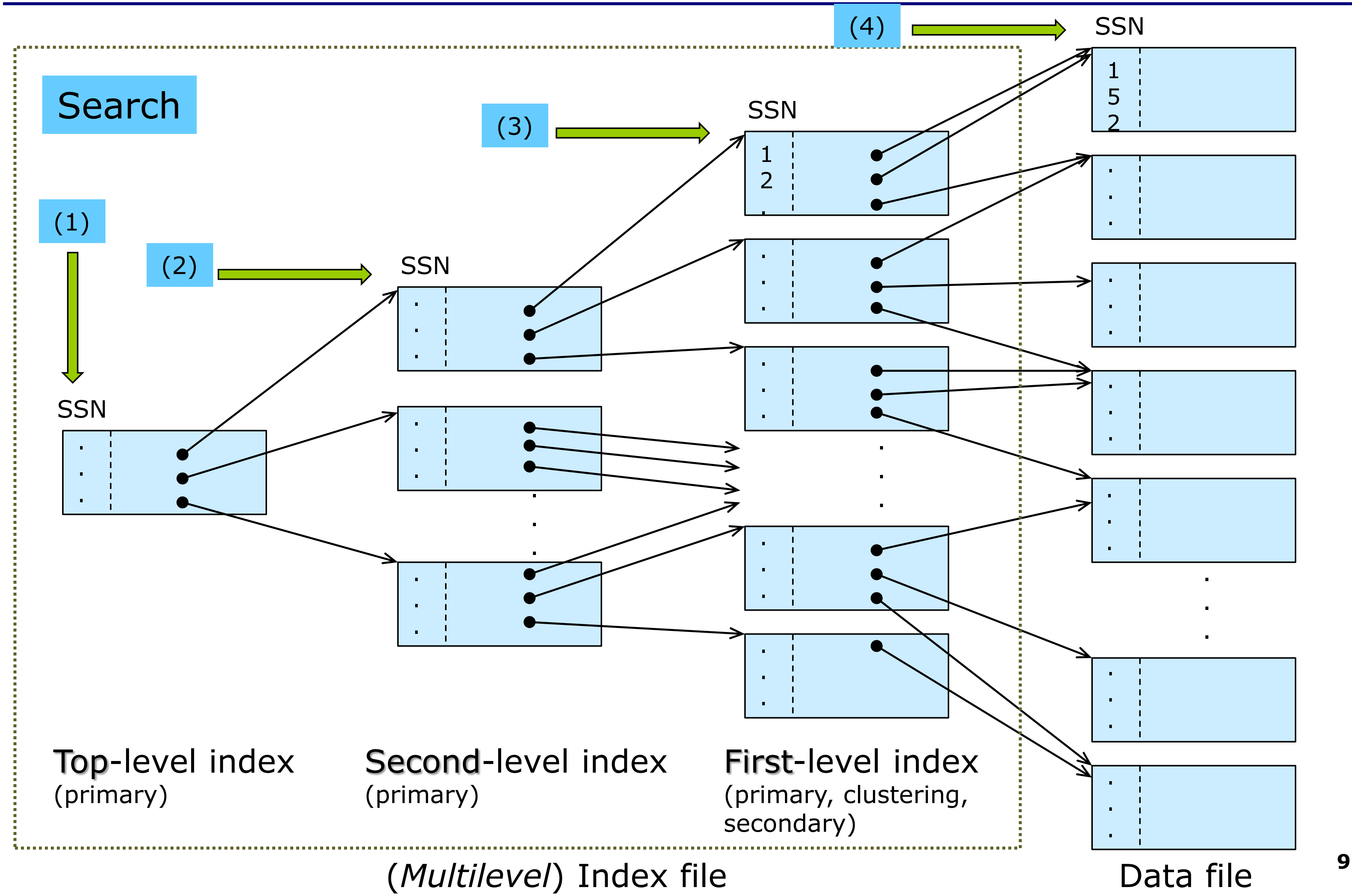
Multilevel Indexes



Multilevel Indexes

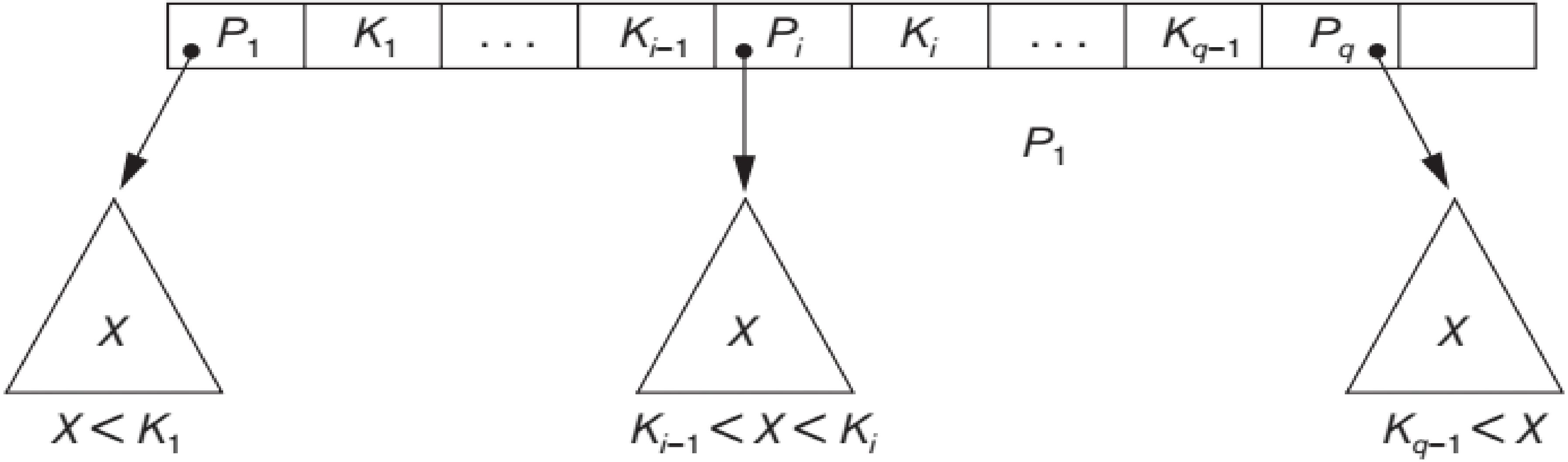


Multilevel Indexes



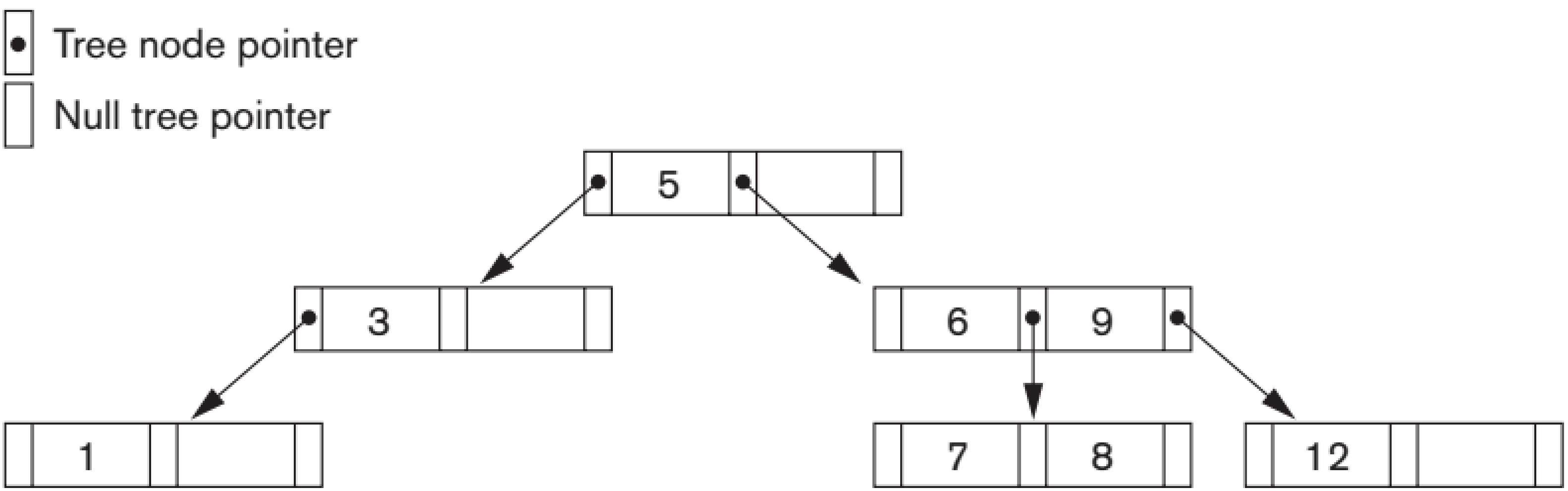
Multilevel Indexes

- A multilevel index is a form of a *search tree*; however, insertion and deletion of new index entries is a severe problem because every level of the index is an *ordered file.*
- To retain the benefits of using multilevel indexing while reducing index insertion and deletion problems, designers adopted a multilevel index called a **dynamic multilevel index** that *leaves some space in each of its blocks* for inserting new entries and uses appropriate insertion/deletion algorithms for creating and deleting new index blocks *when the data file grows and shrinks*.



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

q ≤ p where p is the tree order.



A search tree of order p = 3

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

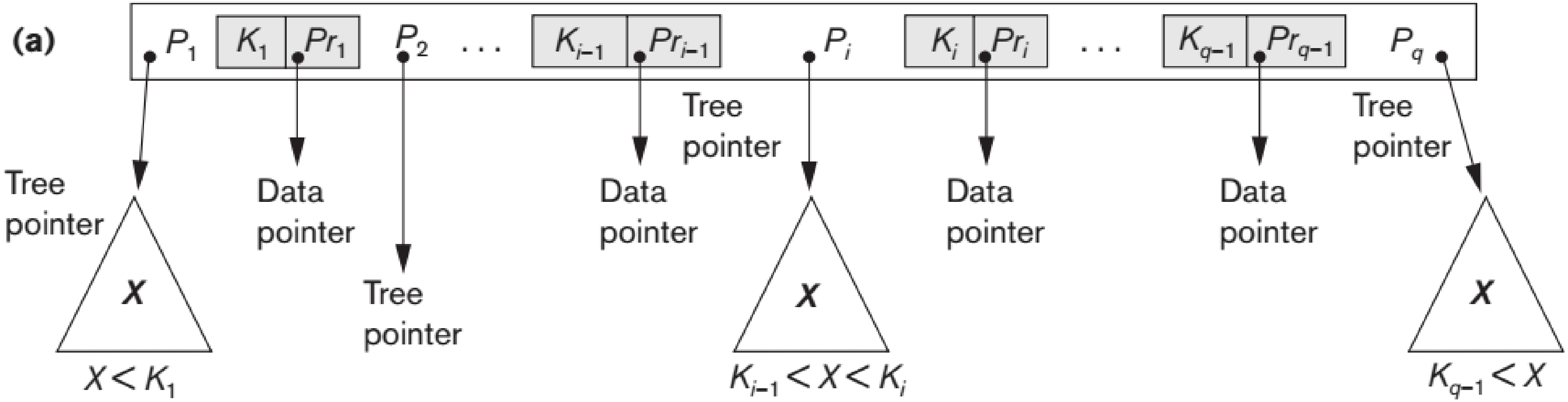
- B-trees and B+-trees are special cases of the **balanced** search tree structure.
- A **tree** is formed of **nodes**.
  - Each node in the tree, except for a special node called the **root**, has one **parent** node and zero or more **child** nodes. The root node has no parent.
  - A node that does not have any child nodes is called a **leaf** node; a nonleaf node is called an **internal** node.
  - The **level** of a node is always one more than the level of its parent, with the level of the root node being *zero*.

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

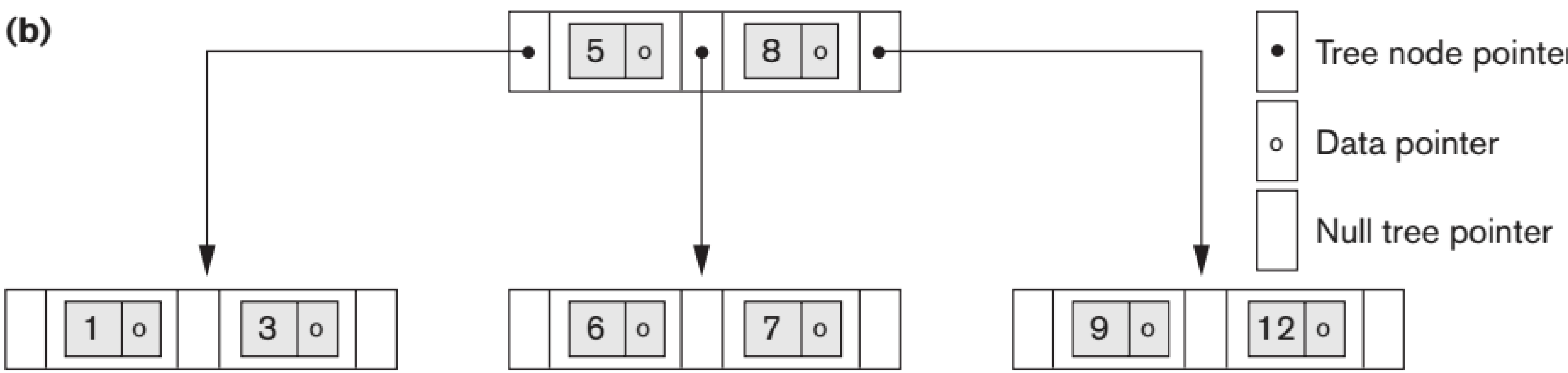
- B-Tree and B+-Tree
  - Each node is stored in a disk block.
  - Each node is kept between 50 and 100 percent full.
  - All the leaf nodes are at the same level.
- In B-tree, pointers to the data blocks are stored in both internal nodes and leaf nodes of the B-tree structure.
- B+-tree is a variation of B-tree.
  - Pointers to the data blocks are stored only in leaf nodes.
  - The leaf nodes are usually linked to each other.

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

- All the nodes in B-tree are of the same structure.
- In B+-tree, the structure of internal nodes is different from that of leaf nodes.
  - For the same data file, B+-tree has fewer levels.
  - For the same levels, B+-tree is a higher-capacity index.
- B+-tree is a common structure used for indexing in current DBMSs.



(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-tree structures

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

- A **B-tree of order** *p*, when used as an access structure on a *key field* to search for records in a data file, can be defined:

  1. Each internal node in the B-tree 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**—a pointer to another node in the B-tree. Each Pr_i is a **data pointer**—a pointer to the record whose search key field value is equal to K_i (or to the data file block containing that record).

  2. Within each node, $K_1 < K_2 < \ldots < K_{q-1}$.

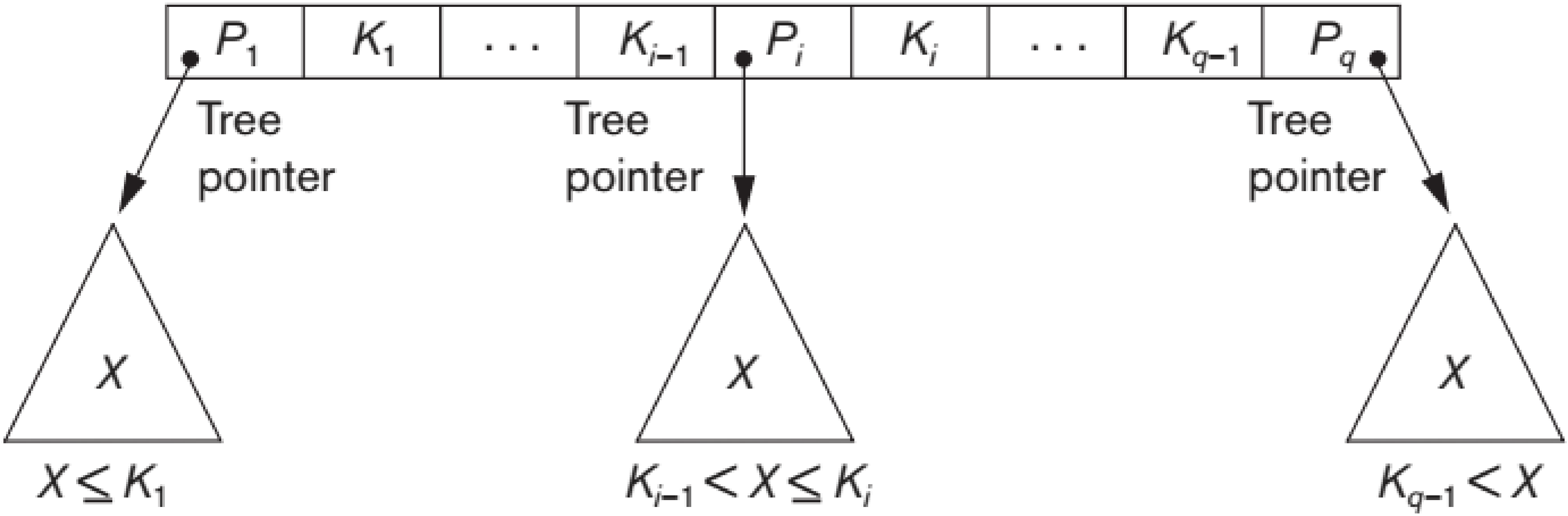
  3. For all search key field values X in the subtree pointed at by P_i, we have: $K_{i-1} < X < K_i$ for $1 < i < q$; $X < K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$

  4. Each node has at most *p* tree pointers.

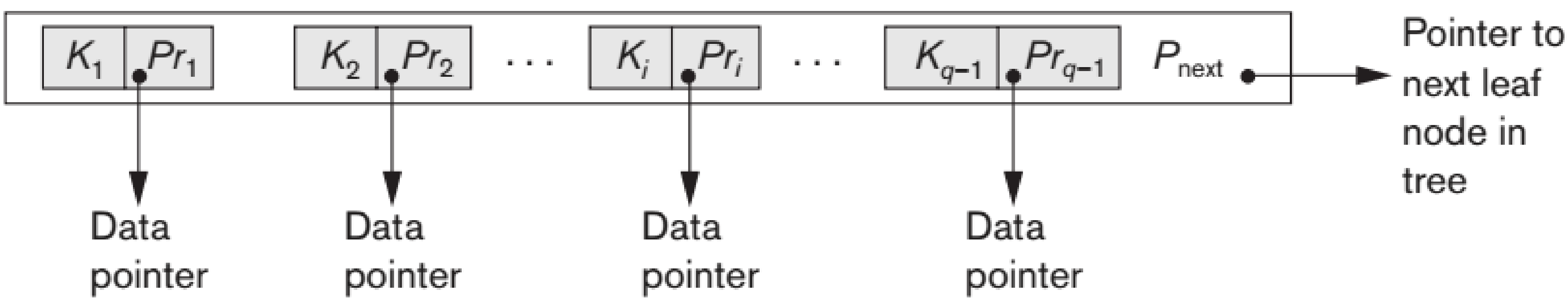
  5. Each node, except the root and leaf nodes, has at least $\lceil p/2 \rceil$ tree pointers. The root node has at least two tree pointers unless it is the only node in the tree.

  6. A node with q tree pointers, $q \leq p$, has $q - 1$ search key field values (and hence has $q - 1$ data pointers).

  7. All leaf nodes are at the same level. Leaf nodes have the same structure as internal nodes except that all of their *tree pointers* P_i are NULL.



(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.

The nodes of a B+-tree

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

- The structure of the *internal nodes* of a B+-tree of order p:

  1. Each internal node is of the form
  $<P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q>$
  where $q \leq p$ and each P_i is a **tree pointer**.

  2. Within each internal node, $K_1 < K_2 < \ldots < K_{q-1}$.

  3. For all search values X in the subtree pointed at by P_i, we have $K_{i-1} < X \leq K_i$ for $1 < i < q$; $X \leq K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$.

  4. Each internal node has at most p tree pointers.

  5. Each internal node, except the root, has at least $\lceil p/2 \rceil$ tree pointers. The root node has at least two tree pointers if it is an internal node.

  6. An internal node with q pointers, $q \leq p$, has $q - 1$ search field values.

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

- The structure of the *leaf nodes* of a B+-tree of order p_{leaf}:

  1. Each leaf node is of the form
  $<<K_1, Pr_1>, <K_2, Pr_2>, \ldots, <K_{q-1}, Pr_{q-1}>, P_{next}>$
  where $q \leq p_{leaf}$, each Pr_i is a data pointer, and P_{next} points to the next *leaf node*.

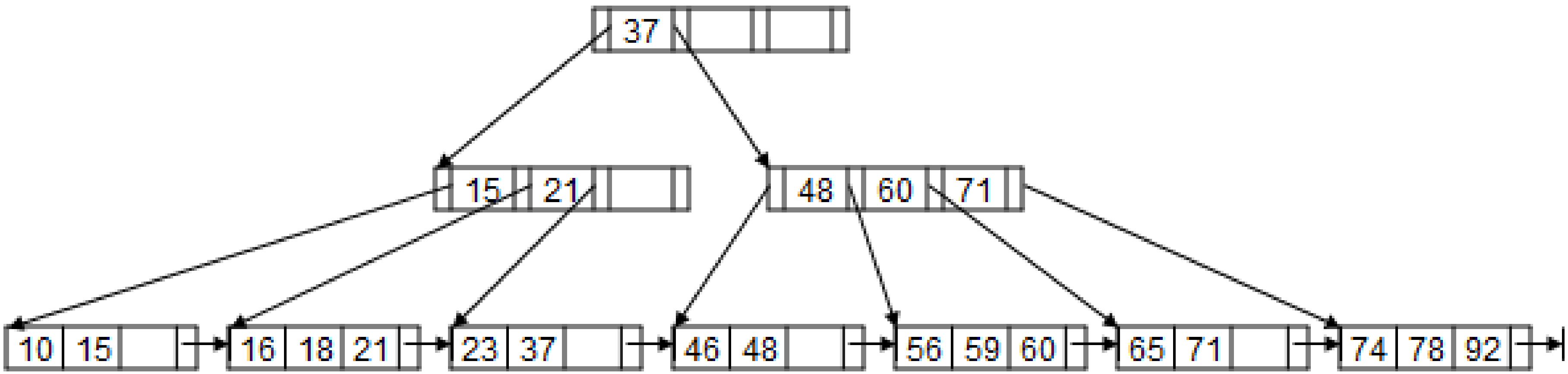
  2. Within each leaf node, $K_1 \leq K_2 \ldots, K_{q-1}, q \leq p_{leaf}$.

  3. Each Pr_i is a **data pointer** that points to the record whose search field value is K_i or to a file block containing the record (or to a block of record pointers that point to records whose search field value is K_i if the search field is not a key).

  4. Each leaf node has at least $\lceil p_{leaf}/2 \rceil$ values.

  5. All leaf nodes are at the same level.

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



A B+-tree with orders: p = 4 and p_{leaf} = 3, *half* full

For simplicity, data pointers are not included in the leaf nodes.

Insertion sequence:

23, 65, 37, 60, 46, 92, 48, 71, 56, 59, 18, 21, 10, 74, 78, 15, 16

Indexes on Multiple Keys

- In many retrieval and update requests, multiple attributes are involved.
- If a certain combination of attributes is used frequently, it is advantageous to set up an access structure to provide efficient access by a key value that is a combination of those attributes.
- if an index is created on attributes $<A_1, A_2, \ldots , A_n>$, the search key values are tuples with n values: $<v_1, v_2, \ldots , v_n>$.
- A lexicographic ordering of these tuple values establishes an order on this composite search key.
- An index on a composite key of n attributes works similarly to any index discussed so far.

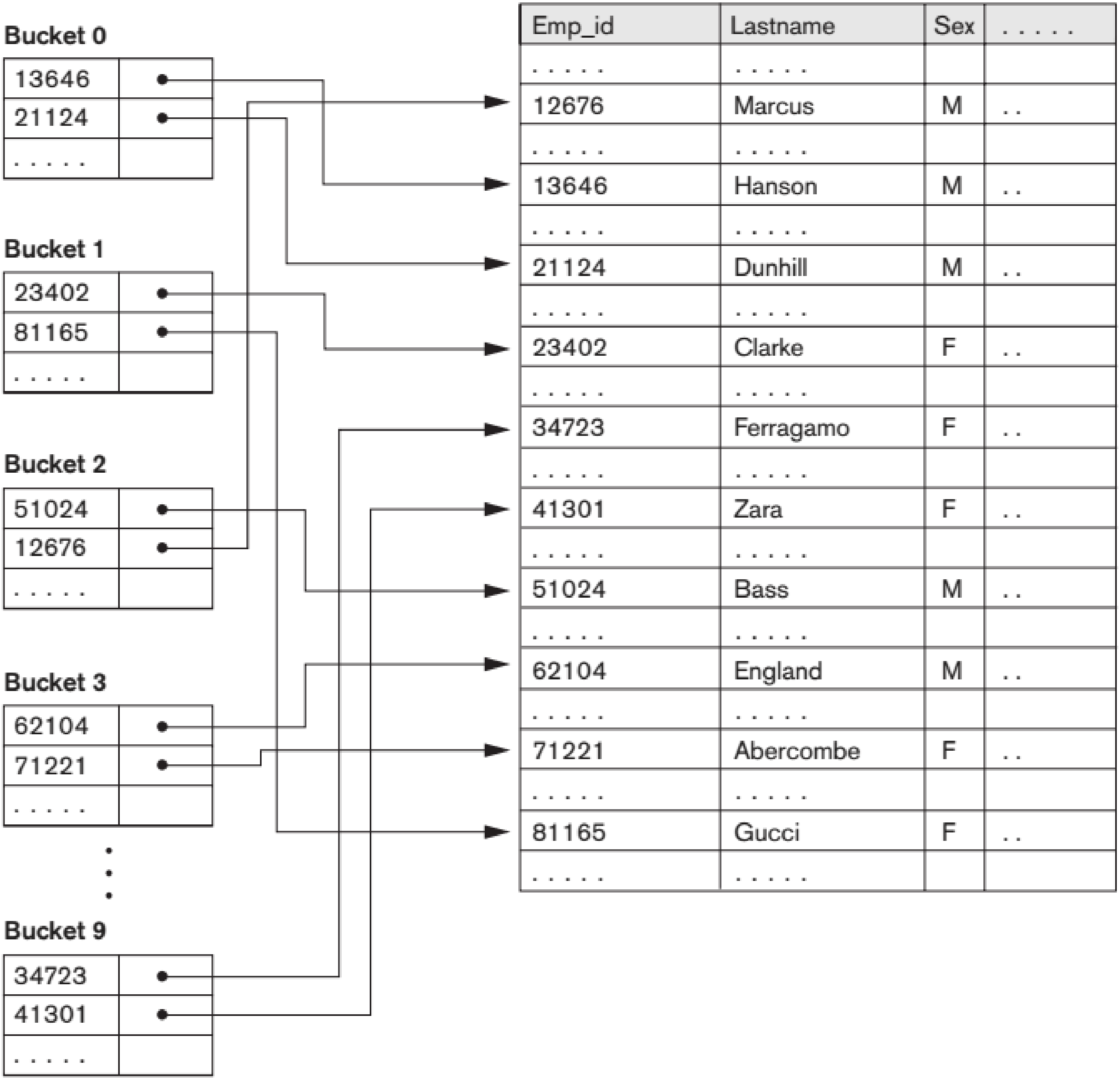
Other File Indexes

- Hash indexes
  - The **hash index** is a secondary structure to access the file by using hashing on a search key other than the one used for the primary data file organization.
- Bitmap indexes
  - A bitmap index is built on **one particular value** of a field (the column in a table) with respect to all the rows (records) and is an array of bits.
- Function-based indexes
  - In Oracle, an index such that the value that results from applying a function (expression) on a field or some fields becomes the key to the index

Other File Indexes

- Hash indexes
  - The **hash index** is a secondary structure to access the file by using hashing on a search key other than the one used for the primary data file organization.
    - access structures similar to indexes, based on *hashing*
  - Support for *equality searches on the hash field*

Hash-based indexing



a *hashing function*: the sum of the digits of Emp_id modulo 10

Other File Indexes

- Bitmap indexes
  - A bitmap index is built on **one particular value** of a field (the column in a table) with respect to all the rows (records) and is an array of bits.
    - Each bit in the bitmap corresponds to a row. If the bit is set, then the row contains the key value.
  - In a bitmap index, each indexing field value is associated with pointers to multiple rows.
  - Bitmap indexes are primarily designed for data warehousing or environments in which queries reference many columns in an ad hoc fashion.
    - The number of distinct values of the indexed field is small compared to the number of rows.
    - The indexed table is either read-only or not subject to significant modification by DML statements.

Other File Indexes

- Bitmap indexes – Adapted examples in Oracle - E25789-01

customers Table

cust_id	cust_last_name	cust_marital_status	cust_gender
1	Kessel		M
2	Koch		F
3	Emmerson		M
4	Hardy		M
5	Gowen		M
6	Charles	single	F
7	Ingram	single	F

Sample bitmaps

Value	Row 1	Row 2	Row 3	Row 4	Row 5	Row 6	Row 7
M	1	0	1	1	1	0	0
F	0	1	0	0	0	1	1
single	0	0	0	0	0	1	1
divorced	0	0	0	0	0	0	0

Other File Indexes

- Bitmap indexes – Adapted examples in Oracle - E25789-01

SELECT COUNT(*)
FROM customers
WHERE cust_gender = 'F'
  AND cust_marital_status **IN** ('single', 'divorced');

The resulting bitmap to access the table

Value	Row 1	Row 2	Row 3	Row 4	Row 5	Row 6	Row 7
M	1	0	1	1	1	0	0
F	0	1	0	0	0	1	1
single	0	0	0	0	0	1	1
divorced	0	0	0	0	0	0	0
single **or** divorced, **and** F	0	0	0	0	0	1	1

Other File Indexes

- Function-based indexes
  - The use of any function on a column prevents the index defined on that column from being used.
    - *Indexes are only used with some specific search conditions on indexed columns.*
  - In Oracle, a function-based index is an index such that the value that results from applying some function (expression) on a field or a collection of fields becomes the key to the index.
    - A function-based index can be either a B-tree or a bitmap index.

Other File Indexes

- Function-based indexes

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

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


CREATE INDEX income_ix
ON Employee(Salary + (Salary*Commission_pct));

SELECT First_name, Lname
FROM Employee
WHERE ((Salary*Commission_pct) + Salary ) > 15000;


CREATE UNIQUE INDEX promo_ix ON Orders
(CASE WHEN Promotion_id = 2 THEN Customer_id ELSE NULL END,
CASE WHEN Promotion_id = 2 THEN Promotion_id ELSE NULL END);
```

Other File Indexes

- Common statements for index creation

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

```
CREATE INDEX DnoIndex
ON EMPLOYEE (Dno)
CLUSTER ;
```

- UNIQUE is used to guarantee that no two rows of a table have duplicate values in the key column or column.
- CLUSTER is used when the index to be created should also sort the data file records on the indexing attribute.
- Specifying CLUSTER on a key (unique) attribute would create some variation of a primary index, whereas specifying CLUSTER on a nonkey (nonunique) attribute would create some variation of a clustering index.

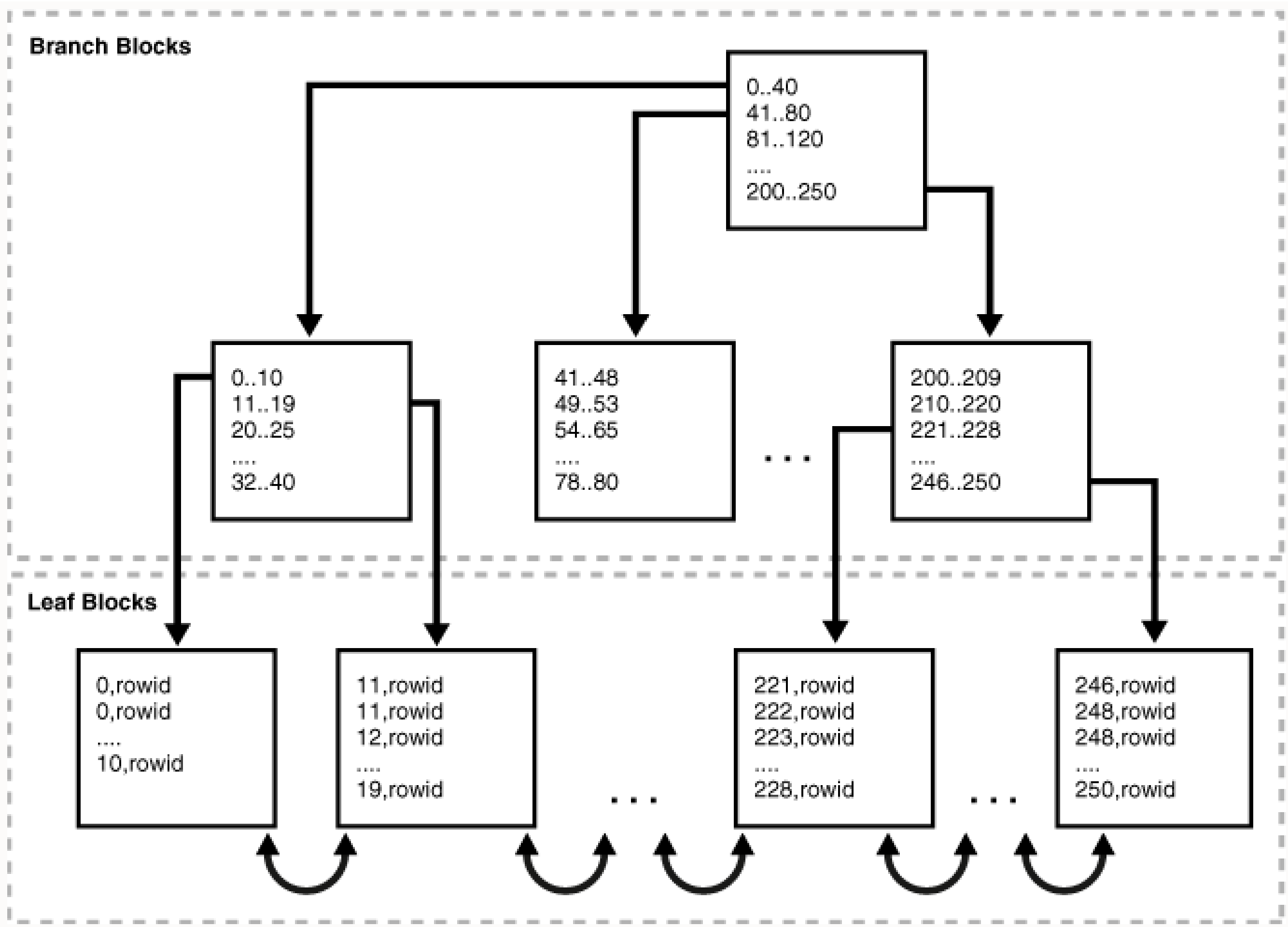
Indexes in Today's DBMSs

Features of InnoDB storage engine in MySQL 8.0

B-tree for index structures

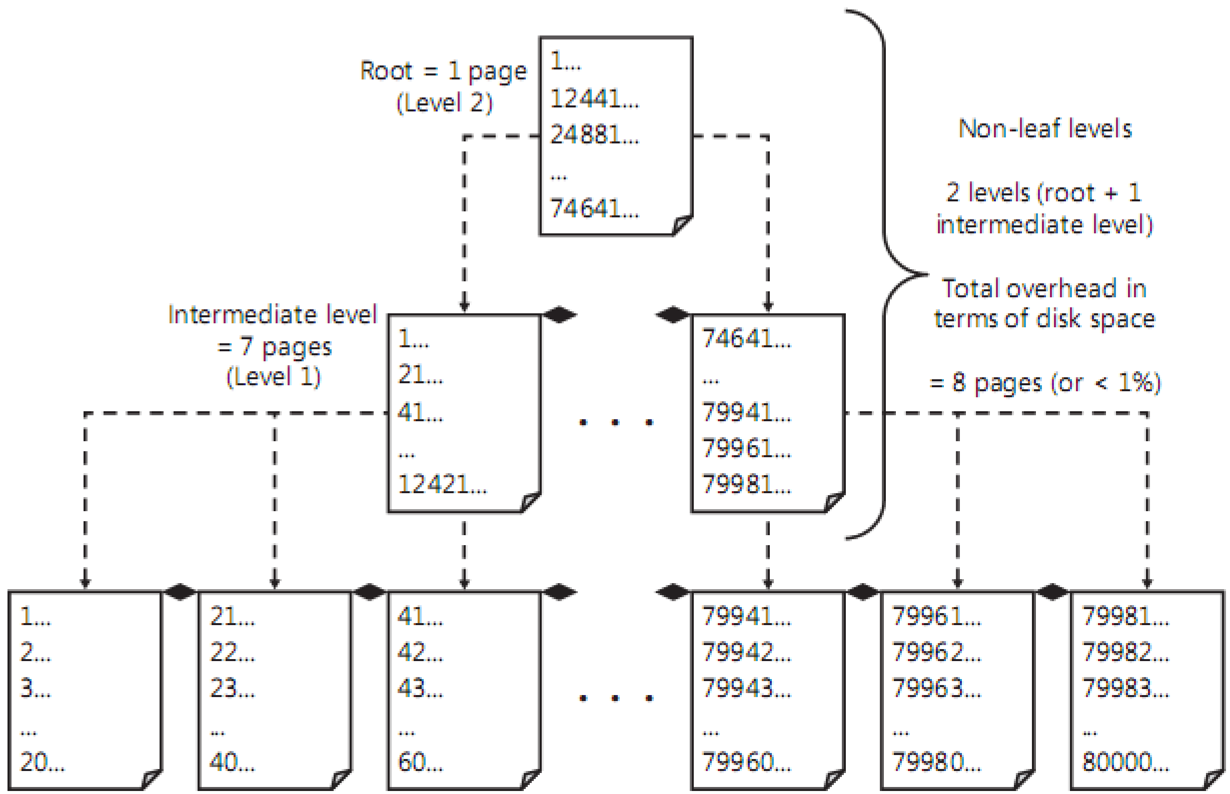
Feature	Support
B-tree indexes	Yes
Backup/point-in-time recovery (Implemented in the server, rather than in the storage engine.)	Yes
Cluster database support	No
Clustered indexes	Yes
Compressed data	Yes
Data caches	Yes
Encrypted data	Yes (Implemented in the server via encryption functions; In MySQL 5.7 and later, data-at-rest tablespace encryption is supported.)
Foreign key support	Yes
Full-text search indexes	Yes (InnoDB support for FULLTEXT indexes is available in MySQL 5.6 and later.)
Geospatial data type support	Yes
Geospatial indexing support	Yes (InnoDB support for geospatial indexing is available in MySQL 5.7 and later.)
Hash indexes	No (InnoDB utilizes hash indexes internally for its Adaptive Hash Index feature.)
Index caches	Yes
Locking granularity	Row
MVCC	Yes
Replication support (Implemented in the server, rather than in the storage engine.)	Yes
Storage limits	64TB
T-tree indexes	No
Transactions	Yes
Update statistics for data dictionary	Yes

Indexes in Today's DBMSs



Internal structure of a B-tree index in Oracle 19c

Indexes in Today's DBMSs



An adapted **B-tree** for a *clustered index* in MS SQL Server

How about *Columnstore* Index in today's MS SQL Server?

6.2. Indexing - Summary

- Indexes: additional access structures for *efficiency*
  - Created on one or many fields of a data file, called indexing fields
    - Ordering key field => Primary indexes
    - Ordering non-key field => Clustering indexes
    - Non-ordering field => Secondary indexes
  - Also stored in index files on disk
  - Single-level vs. Multilevel indexes
    - Dynamic multilevel index structures: B-tree, B+-tree
  - Support certain search conditions
    - =, >, >=, <, <=, and “between” on indexing fields

6.3. Complex Data Management Approaches (Semi-structured and Unstructured Data)

- *Structured data*: data stored in relational databases which are represented in a *strict* format.
  - For example: data in the COMPANY database defined with the relational data model.
- *Semistructured data*: data that may have a certain structure, but not all the data collected will have the identical structure. Some attributes may be shared among the various entities, but other attributes may exist only in a few entities. Moreover, additional attributes can be introduced in some of the newer data items at any time, and there is no fixed predefined schema. Semistructured data are sometimes referred to as *self-describing* data.
  - For example: data in the COMPANY database defined with XML (Extensible Markup Language) or JSON (Javascript Object Notation) format.
- *Unstructured data*: data for which there is very limited indication of the type of data.
  - For example: textual data on webpages, images, audio files, videos, …

6.3. Complex Data Management Approaches (Semi-structured and Unstructured Data)

- General-purpose DBMSs that support complex data
  - (Extended) Relational
  - XML
  - Object
  - Object relational
  - NoSQL
  - NewSQL
- Specialized DBMSs that support complex data
  - Multimedia DBMSs

6.4. Massive Data Management Approaches

- How big is big?
  - 1 exabyte (EB) = 10^{18} bytes = 1000 petabytes = 1 million terabytes = 1 billion gigabytes = 1 trillion megabytes
  - 1 zettabyte (ZB) = 1000 exabytes = 10^{21} bytes
  - Global Datasphere = 175 zettabytes of digital data by 2025

Source:
Wikipedia for units.
IDC Data Age 2025: the Digitization of the World from Edge to Core, November 2018.
https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf

6.4. Massive Data Management Approaches

- Massive Data
  - Data sets that traditional data architectures are unable to handle efficiently
    - Those in organizations such as Google, Amazon, Facebook, and Twitter and in applications such as social media, Web links, user profiles, marketing and sales, posts and tweets, road maps and spatial data, and e-mail
    - Those with the following characteristics
      - **V**olume (lượng): too big
      - **V**elocity (tốc độ): arrives too fast
      - **V**ariability (sự biến thiên): changes too fast
      - **V**eracity (độ chính xác): contains too much noise
      - **V**ariety (độ đa dạng): too diverse

6.4. Massive Data Management Approaches

- Massive data management systems
  - SQL systems give an emphasis on immediate data consistency, powerful query languages, and structured data storage. For example: Oracle, PostgreSQL, DB2, MS SQL Server, MySQL, Informix, …
  - Most NoSQL systems are distributed database or distributed storage systems, with a focus on semistructured data storage, high performance, availability, data replication, and scalability. For example: MongoDB, Hbase, Cassandra, Neo4J, …
  - NewSQL systems are modern relational DBMSs that seek to provide the same scalable performance of NoSQL for OLTP read-write workloads while still maintaining ACID guarantees for transactions. For example: VoltDB, ...
    - Andrew Pavlo and Mathew Aslett. What’s really new with NewSQL? SIGMOD Record, vol. 45, no. 2, pp. 45-55, June 2016.

6.4. Massive Data Management Approaches - NoSQL

- *Data representation*: most schemaless for self-describing data.
- *Constraint enforcement*: most at the application programs.
- *Data manipulation*: CRUD or SCRUD operations for search, create, read, update, and delete with programming APIs. A few query languages like SQL: Cypher (Neo4J), CQL (Cassandra), N1QL (Couchbase), ...
- *Versioning*: storage of multiple versions of the data items, with the timestamps of when the data version was created.
- *Data architecture*: distributed systems.
  - *Scalability*: often horizontal by adding more nodes for data storage and processing as the volume of data grows
  - *Availability*, *Replication* and *Eventual Consistency*: no concurrency control
  - *Replication*: Master-Slave (write at master), Master-Master (reconciliation)
  - *Sharding* of Files: horizontal partitioning
  - *High-Performance Data Access*: hashing, range partitioning, and indexing on object keys

6.4. Massive Data Management Approaches - NoSQL

- Three desirable properties of distributed systems with replicated data: CAP theorem (*principle*)
  - **C**: *consistency* among replicated copies
    - The nodes will have the same copies of a replicated data item visible for various transactions.
    - A form of *consistency* known as *eventual consistency* is often adopted in NoSQL systems, different from that in ACID of SQL ones.
  - **A**: *availability* of the system for read and write operations
    - Each read or write request for a data item will either be processed successfully or will receive a message that the operation cannot be completed.
  - **P**: *partition tolerance* in the face of the nodes in the system being partitioned by a network fault
    - The system can continue operating if the network connecting the nodes has a fault that results in two or more partitions, where the nodes in each partition can only communicate among each other.

6.4. Massive Data Management Approaches - NoSQL

- Document-based NoSQL systems: MongoDB, CouchDB, …
  - These systems store data in the form of documents using well-known formats, such as JSON (JavaScript Object Notation). Documents are accessible via their document id, but can also be accessed rapidly using other indexes.
- NoSQL key-value stores: DynamoDB, Cassandra, …
  - These systems have a simple data model based on fast access by the key to the value associated with the key; the value can be a record or an object or a document or even have a more complex data structure.
- Column-based or wide column NoSQL systems : HBase, BigTable, …
  - These systems partition a table by column into column families, where each column family is stored in its own files. They also allow versioning of data values.
- Graph-based NoSQL systems: Neo4J, GraphBase, …
  - Data is represented as graphs, and related nodes can be found by traversing the edges using path expressions.
- Multimodel systems: Cassandra (key-value, column), OrientDB (document, key-value, graph), CouchBase (document, key-value), …

6.4. Massive Data Management Approaches - NoSQL

- Document-based NoSQL systems – MongoDB
  - https://www.mongodb.com
  - MongoDB documents are stored in BSON (*Binary JSON*) format, which is a variation of JSON with some additional data types and is more efficient for storage than JSON. Individual ***documents*** (≈ rows, records in relational databases) are stored in a ***collection*** (≈ table, relation).

db.createCollection("project", { capped : true, size : 1310720, max : 500 })

Collection name

Collection options:
- capped: true > storage option
- size: 1310720 > upper limits on storage space
- max: 500 > number of documents

db.collection.createIndex({}): create an index using B-tree

CRUD: create, insert, find (for read), update, remove (for delete)

6.4. Massive Data Management Approaches - NoSQL

- Document-based NoSQL systems – MongoDB
  - https://www.mongodb.com
  - MongoDB documents are stored in BSON (*Binary JSON*) format, which is a variation of JSON with some additional data types and is more efficient for storage than JSON. Individual ***documents*** (≈ rows, records in relational databases) are stored in a ***collection*** (≈ table, relation).

```
db.createCollection("project", { capped : true, size : 1310720, max : 500 } )

db.createCollection("worker", { capped : true, size : 5242880, max : 2000 } )

db.project.insert( { _id: "P1", Pname: "ProductX", Plocation: "Bellaire" } )
db.worker.insert( [ { _id: "W1", Ename: "John Smith", ProjectId: "P1", Hours: 32.5 },
                    { _id: "W2", Ename: "Joyce English", ProjectId: "P1",
                      Hours: 20.0 } ] )
```

```
db.<collection_name>.insert(<document(s)>)

db.<collection_name>.remove(<condition>)

db.<collection_name>.find(<condition>)
```

6.4. Massive Data Management Approaches - NoSQL

- Document-based NoSQL systems – MongoDB

project document with an array of embedded workers:

```
{
    _id:            "P1",
    Pname:          "ProductX",
    Plocation:      "Bellaire",
    Workers: [
                { Ename: "John Smith",
                  Hours: 32.5
                },
                { Ename: "Joyce English",
                  Hours: 20.0
                }
            ]
}
```

project document with an embedded array of worker ids:

```
{
    _id:            "P1",
    Pname:          "ProductX",
    Plocation:      "Bellaire",
    WorkerIds:      [ "W1", "W2" ]
}
    { _id:          "W1",
    Ename:          "John Smith",
    Hours:          32.5
}
    { _id:          "W2",
    Ename:          "Joyce English",
    Hours:          20.0
}
```

6.4. Massive Data Management Approaches - NoSQL

- Document-based NoSQL systems – MongoDB
  - Transaction support mainly for atomicity (all-or-nothing):
    - An operation on a single document is atomic.
    - The two-phase commit method is used to ensure atomicity and consistency of multidocument transactions.
  - Replication: master-slave: primary, secondaries
    - primary: read/write > MongoDB can ensure that every read request gets the latest document value.
    - primary: write, secondaries: read > A read at a secondary is not guaranteed to get the latest version of a document.
  - Sharding (horizontal partitioning)
  - Horizontal scaling: add more nodes as needed

6.5. Quality Issues: Reliability, Scalability, Effectiveness, Efficiency

- Data quality issues
  - Accuracy
  - Timeliness
  - Completeness
  - Consistency
  - Interpretability
  - Accessibility
  - Usability
  - Trustworthiness
- Data management quality issues
  - Reliability
  - Scalability
  - Effectiveness
  - Efficiency

6.5. Data Quality Issues

- Chất lượng (quality of data/information)
  - Phù hợp với đặc tả (specifications), yêu cầu từ người dùng (user requirements), ngữ cảnh sử dụng (context of use), …
  - "A comprehensive list of commonly agreed quality dimensions is still not available."
  - Phân loại chiều chất lượng (quality dimensions)
    - Schema quality dimensions → structure
    - Data quality dimensions → instance

C. Batini, B. Pernici. Data Quality Management and Evolution of Information Systems. In IFIP International Federation for Information Processing, volume 214, The Past and Future of Information Systems: 1976-2006 and Beyond, eds. D. Avison, S. Elliot, J. Krogstie, J. Pries-Heje, Boston: Springer, 2006, pp. 51-62.

6.5. Data Quality Issues

- Data quality dimensions
  - *Accuracy*: “inaccuracy implies that the information system represents a real world state different from the one that should have been represented.”
  - *Timeliness*: refers to “the delay between a change of the real-world state and the resulting modification of the information system state.”
  - *Completeness*: is “the ability of an information to represent every meaningful state of the represented real world system”

C. Batini, B. Pernici. Data Quality Management and Evolution of Information Systems. In IFIP International Federation for Information Processing, volume 214, The Past and Future of Information Systems: 1976-2006 and Beyond, eds. D. Avison, S. Elliot, J. Krogstie, J. Pries-Heje, Boston: Springer, 2006, pp. 51-62.

6.5. Data Quality Issues

- Data quality dimensions
  - *Consistency*: consistency of data values occurs whether or not there is more than one state of the information system matching a state of the real world system; therefore, "inconsistency would mean that the representation mapping is one-to-many."
  - *Interpretability*: concerns the documentation and metadata that are available to interpret correctly the meaning and properties of data sources

C. Batini, B. Pernici. Data Quality Management and Evolution of Information Systems. In IFIP International Federation for Information Processing, volume 214, The Past and Future of Information Systems: 1976-2006 and Beyond, eds. D. Avison, S. Elliot, J. Krogstie, J. Pries-Heje, Boston: Springer, 2006, pp. 51-62.

6.5. Data Quality Issues

- Data quality dimensions
  - *Accessibility*: measures the ability of the user to access the data as from his/her own culture, physical status/functions and technologies available
  - *Usability*: measures the effectiveness, efficiency, satisfaction with which specified users perceive and make use of data
  - *Trustworthiness*: measures how reliable the organization is in providing data sources

C. Batini, B. Pernici. Data Quality Management and Evolution of Information Systems. In IFIP International Federation for Information Processing, volume 214, The Past and Future of Information Systems: 1976-2006 and Beyond, eds. D. Avison, S. Elliot, J. Krogstie, J. Pries-Heje, Boston: Springer, 2006, pp. 51-62.

6.5. Data Management Quality Issues: Reliability, Scalability, Effectiveness, Efficiency

- Data Management Quality Issues
  - *Reliability*: broadly defined as the probability that a system is running (not down) at a certain time point.
  - *Scalability*: the extent to which the system can expand its capacity (i.e. data volumes, users, connections) while continuing to operate without interruption.
  - *Effectiveness*: how correctly a task has been resolved.
  - *Efficiency*: how well resources have been used.

6.5. Data Management Quality Issues: Reliability, Scalability, Effectiveness, Efficiency

- Data Management Quality Issues
  - *Reliability*
    - One common approach stresses fault tolerance; it recognizes that faults will occur, and it designs mechanisms that can detect and remove faults before they can result in a system failure.
    - Another more stringent (severe, very serious) approach attempts to ensure that the final system does not contain any faults. This is done through an exhaustive design process followed by extensive quality control and testing.
    - For example: improving *reliability* with RAID systems.
  - *Scalability*
    - Support for large databases. For example:
      - Scalability with storage technologies: NAS.
      - Scalability support from existing DBMSs
        - MySQL: mentioned with 200,000 tables, 5,000,000,000 rows, up to 64 indexes per table, each index of 1 to 16 columns or parts of columns.
        - Oracle 19c: unlimited for the max number of tables, the max number of rows per table, the max number of constraints per column, and the max number of indexes per table; restrictions for index-organized tables: the max number of columns: 1000, the max number of columns in a primary key: 32, the max number of columns in the index portion of a row: 255
  - *Effectiveness*
  - *Efficiency*

6.5. Data Management Quality Issues: Reliability, Scalability, Effectiveness, Efficiency

- Data Management Quality Issues
  - *Reliability*
  - *Scalability*
  - *Effectiveness*
    - An assumption called the *closed world assumption* states that the only true facts in the universe are those present within the extension (state) of the relation(s).
    - DBMSs must ensure any true fact can be retrieved from or updated to the database consistently.
      - Query processing
      - Data manipulation with constraint enforcement
  - *Efficiency*
    - Space and time in connection, data processing, concurrency control, recovery, and backup

Summary

- Storage devices
  - Magnetic disks for large amounts of data
    - Disk pack → cylinder → track → sector → bit
    - Bit → byte → block → track → cylinder
    - **Block**: data transfer unit between disks and main memory
    - Block address = block *pointer*
- Data File of a Database
  - Bit → byte → field → record → file => disk blocks
    - Blocking factor: the number of records/block
  - Primary file organization for records of one type
    - Unordered (heap) files
    - Ordered (sequential) files
    - Hashed files

Summary

- Indexes: additional access structures for *efficiency*
  - Created on one or many fields of a data file, called indexing fields
    - Ordering key field => Primary indexes
    - Ordering non-key field => Clustering indexes
    - Non-ordering field => Secondary indexes
  - Also stored in index files on disk
  - Single-level vs. Multilevel indexes
    - Dynamic multilevel index structures: B-tree, B+-tree
  - Support certain search conditions
    - =, >, >=, <, <=, and “between” on indexing fields

Summary

- Complex data management approaches
  - Structured data
  - Semi-structured data
  - Unstructured data

Simple data

Complex data

→ Logical data representation

→ Database management systems: XML, Object, Object Relational, NoSQL, NewSQL

Summary

- Massive data management approaches
  - SQL vs. NoSQL vs. NewSQL
  - NoSQL
    - Representation: schemaless
      - Document, key-value, column, graph, …
    - Constraint enforcement mostly at the application side
    - Data manipulation: CRUD, SCRUD
    - Versioning with timestamps
    - Architecture: distributed with the CAP theorem
      - Eventual consistency

Summary

- Data quality issues
  - Accuracy
  - Timeliness
  - Completeness
  - Consistency
  - Interpretability
  - Accessibility
  - Usability
  - Trustworthiness
- Data management quality issues
  - Reliability
  - Scalability
  - Effectiveness
  - Efficiency

Chapter 6: Physical Storage and Data Management

Review

- 1. Describe the memory hierarchy for data storage.
- 2. Distinguish between persistent data and transient data.
- 3. Describe disk parameters when magnetic disks are used for storing large amounts of data.
- 4. Describe the read/write commands with magnetic disks.

Review

- 5. Distinguish between fixed-length records and variable-length records.
- 6. Distinguish between spanned records and unspanned records.
- 7. What is blocking factor? How to compute it?
- 8. What is file organization? What is its goal?
- 9. What is access method? How is it related to file organization?

Review

- 10. Distinguish between static files and dynamic files.
- 11. Compare unordered files, ordered files, and hash files.
- 12. Which operations are more efficient for each file organization: unordered, ordered, hash? Why?
- 13. Distinguish between static hashing and dynamic hashing.

Review

- 14. What are indexes? Give at least three examples.
- 15. What are primary, secondary, and clustering indexes? Give at least one example for each.
- 16. Compare primary, secondary, and clustering indexes with each other. Which are dense and which are not? Explain the characteristics in their corresponding data file that make them dense or sparse.

Review

- 17. Why can at most one primary or clustering index created on a data file, but zero or many secondary indexes? Give an example to demonstrate your answer.
- 18. Distinguish between single-level indexes and multilevel indexes. Give an example to demonstrate your answer.
- 19. Describe B-tree and B+-tree when they are used as secondary access structures for a data file. Distinguish between B-tree and B+-tree. Give an example for each structure.

Review

- 20. For which types of applications were NoSQL systems developed?
- 21. What are the main categories of NoSQL systems? List a few of the NoSQL systems in each category.
- 22. What are the main characteristics of NoSQL systems in the areas related to data models and query languages?
- 23. What are the main characteristics of NoSQL systems in the areas related to distributed systems and distributed databases?
- 24. What is the CAP theorem? Which of the three properties (consistency, availability, partition tolerance) are most important in NoSQL systems?
- 25. What are the similarities and differences between using consistency in CAP versus using consistency in ACID?

Next
Chapter 7: Database security

- 7.1. An overview of database security
- 7.2. Discretionary access control based on granting and revoking privileges
- 7.3. Mandatory access control and role-based access control for multilevel security
- 7.4. Inference control and flow control
- 7.5. Security in new DBMSs