GROUP 4

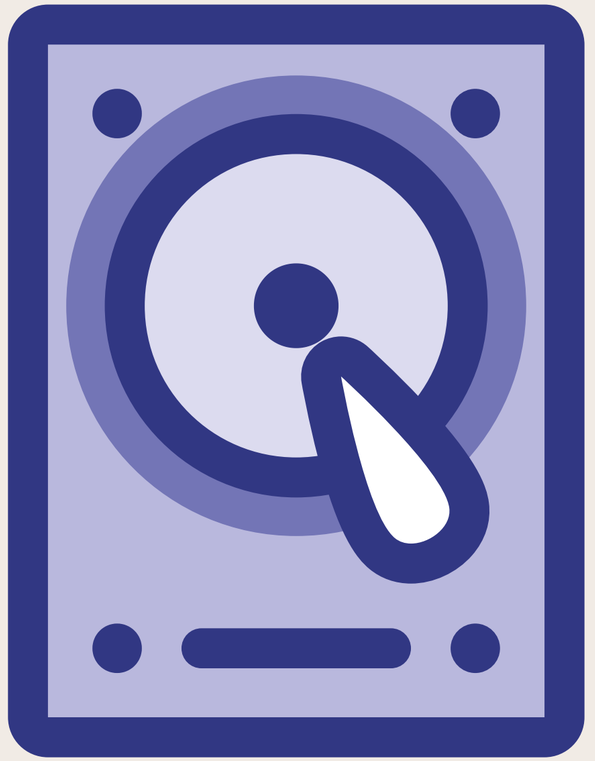
INTRODUCTION

DATA STORAGE AND INDEXING

PROF. TRAN MINH QUANG

DATA STORAGE

- DISK STORAGE
- FILES OF RECORDS
- RAID TECHNOLOGY



INTRODUCTION

Computer storage media form a storage hierarchy that includes two main categories:

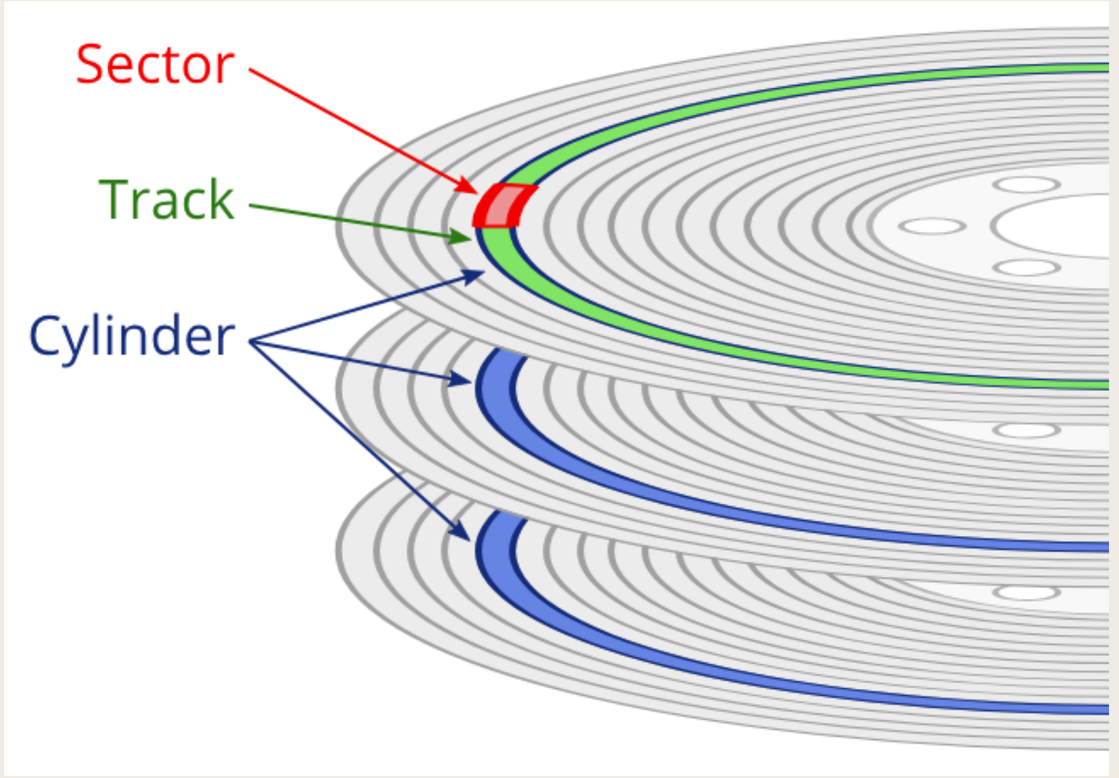
- **Primary storage:** This category includes storage media that can be operated on directly by the computer's central processing unit (CPU), such as the computer's main memory and smaller but faster cache memories.
- **Secondary storage:** The primary choice of storage medium for online storage of enterprise databases has been magnetic disks.
- **Tertiary storage:** Optical disks (CD-ROMs, DVDs, and other similar storage media) and tapes are removable media used in today's systems as offline storage for archiving databases and hence come under the category

=> Generally, databases are too large to fit entirely in main memory.

=> Mainly using **secondary and teriary storage.**

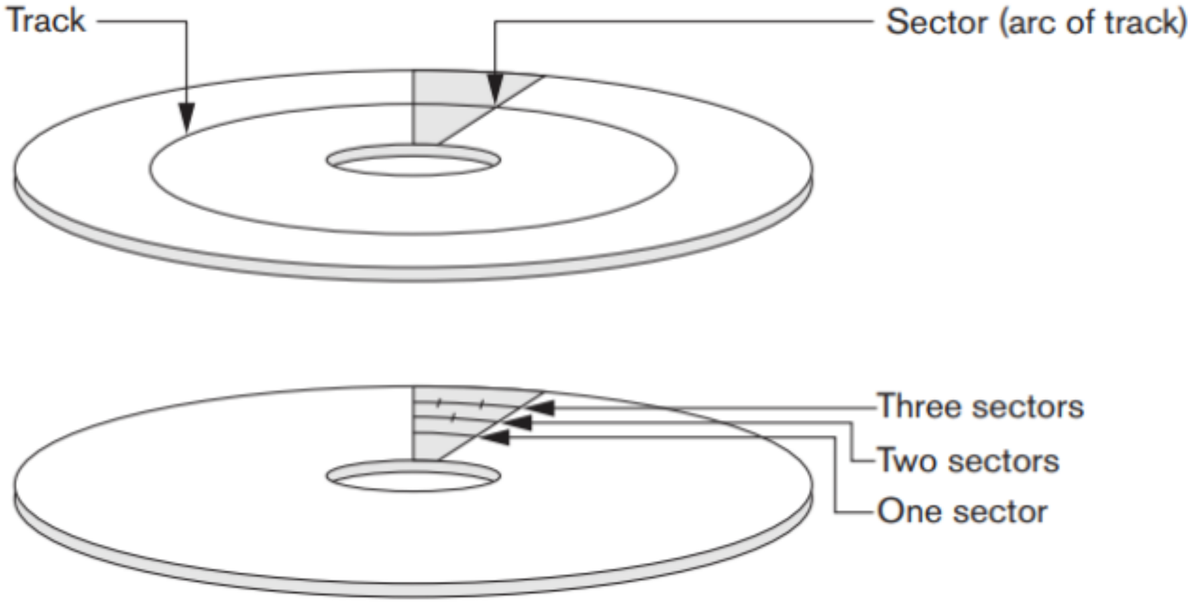
DESCRIPTION OF DISK

- Disks are assembled into a disk pack which may include many **magnetic disks** and many **surfaces.**
- Information is stored on a disk surface in concentric circles called **tracks.**
- A track is divided into smaller **sectors**.
- Tracks with the same diameter on the various surfaces are called a **cylinder.**



DESCRIPTION OF DISK

- The division of a track into equal-sized disk **blocks** (or pages) is set by the operating system during disk formatting (or initialization).
- The division of a track into **sectors** is hard-coded on the disk surface and cannot be changed.



DISK DEVICES

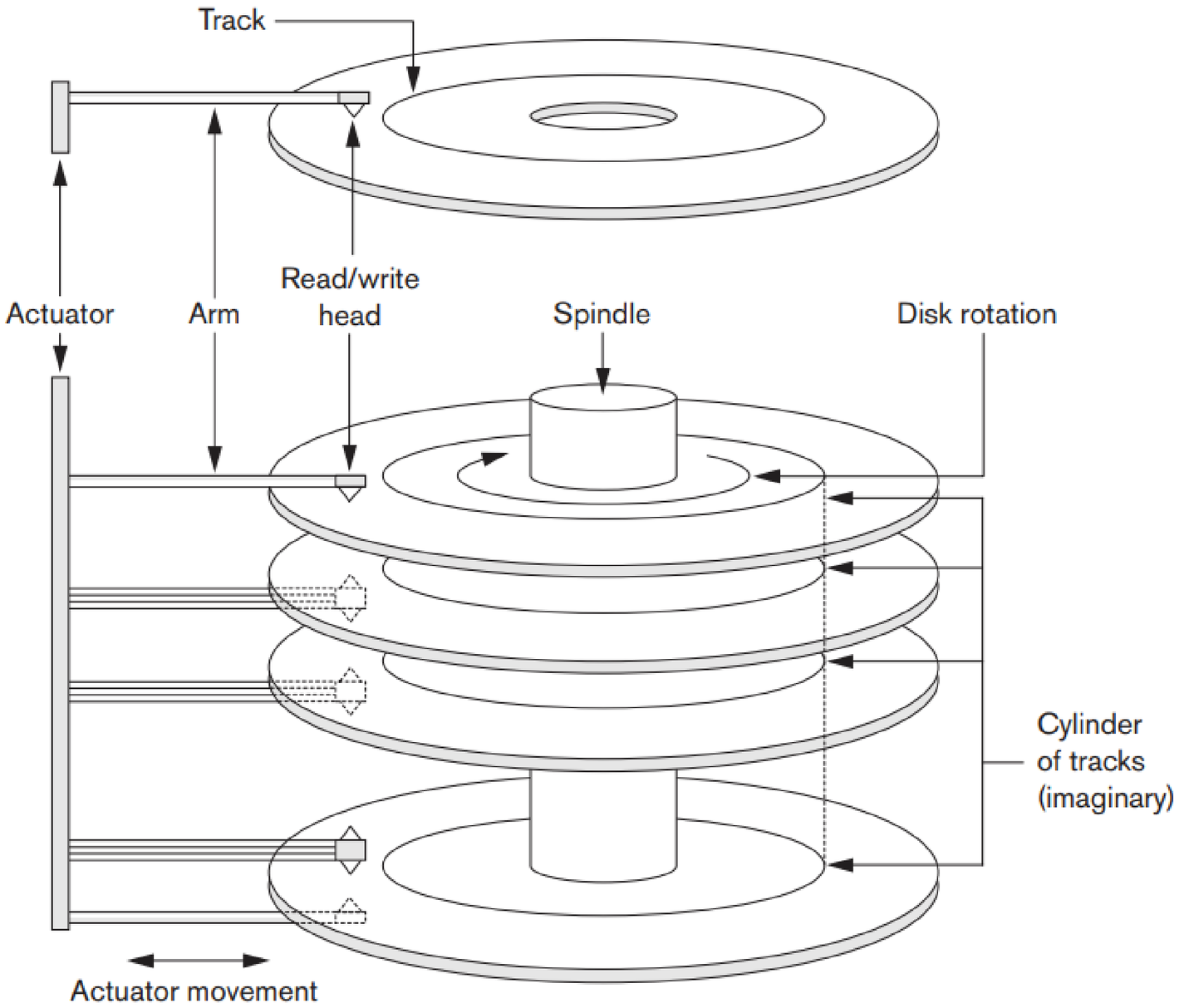
- A **read-write head** moves to the track that contains the block to be transferred. => Disk rotation moves the block under the read-write head for reading or writing.
- A physical disk block (hardware) address consists of:

  - A cylinder number (imaginary collection of tracks of same radius from all recorded surfaces)

  - The track number or surface number (within the cylinder)and block number (within track).

- Reading or writing a disk block is time consuming because of the seek time and rotational delay (latency).
- **Double buffering** can be used to speed up the transfer of contiguous disk blocks.

DISK DEVICES



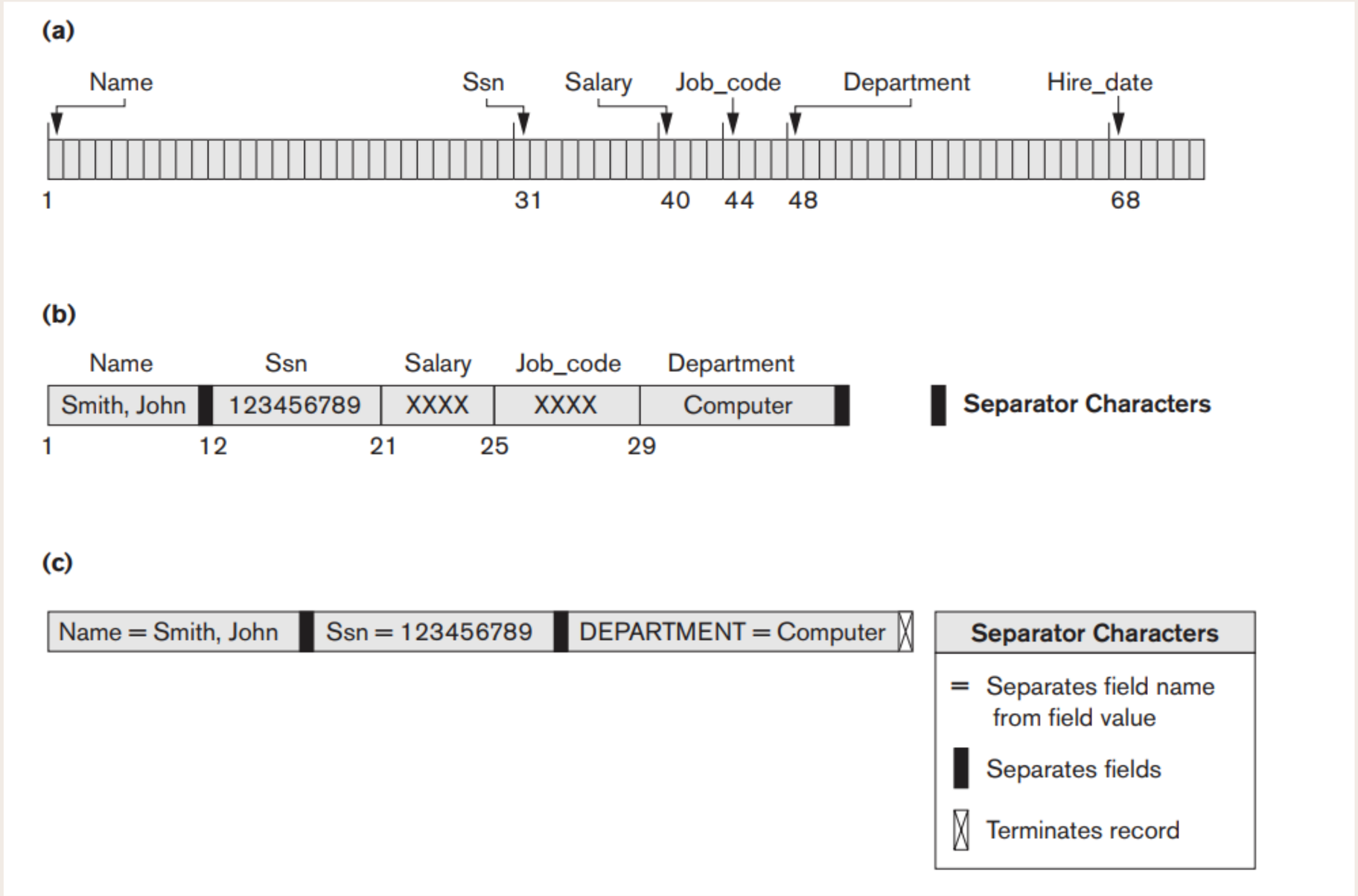
RECORDS AND RECORD TYPES

- 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 particular **field** of the record
- A collection of **field** names and their corresponding data types constitutes a record type or record format definition. A data type, associated with each field, specifies the types of values a field can take.
- The data type of a field is include **numeric** (integer, ...), **string** of characters, **Boolean** , and sometimes specially coded **date** and **time** data types

FILES

- A **file** is a sequence of records
- A **file descriptor** (or **file header**) includes information that describes the file, such as the field names and their data types, and the addresses of the file blocks on disk
- If every record in the file has exactly the same size (in bytes), the file is said to be made up of **fixed-length records**.
- If different records in the file have different sizes, the file is said to be made up of **variable-length records**.

FILES AND RECORDS EXAMPLE

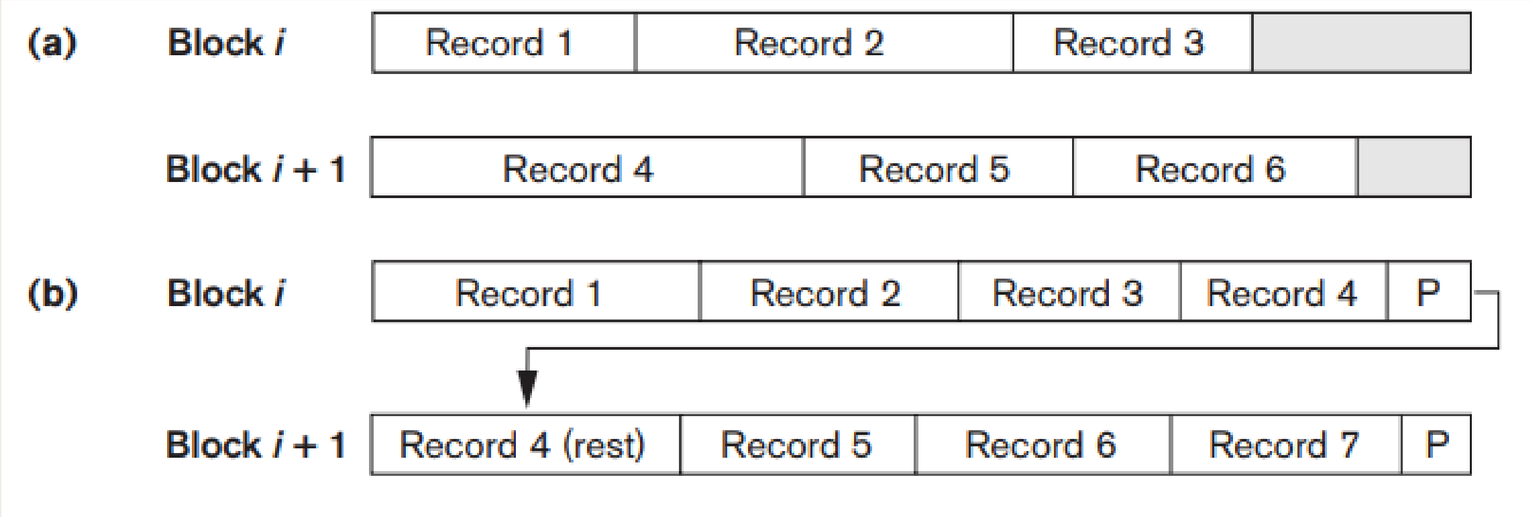


BLOCKING

- **Blocking**: refers to storing a number of records in one block on the disk.
- **Blocking factor** (bfr): refers to the number of records per block.

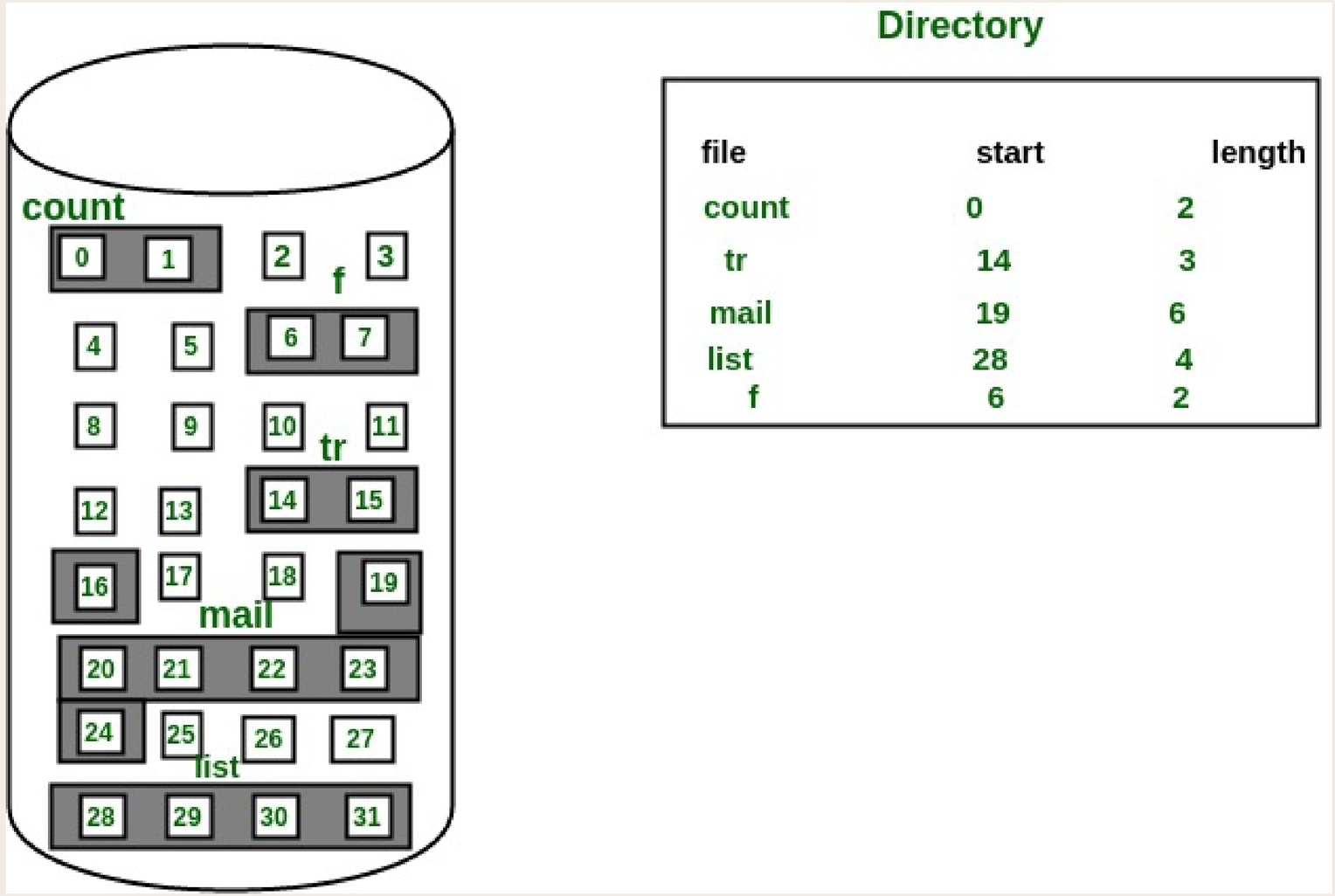
=> There may be empty space in a block if an integral number of records do not fit in one block.

- **Spanned Records:** refer to records that exceed the size of one or more blocks and hence span a number of blocks

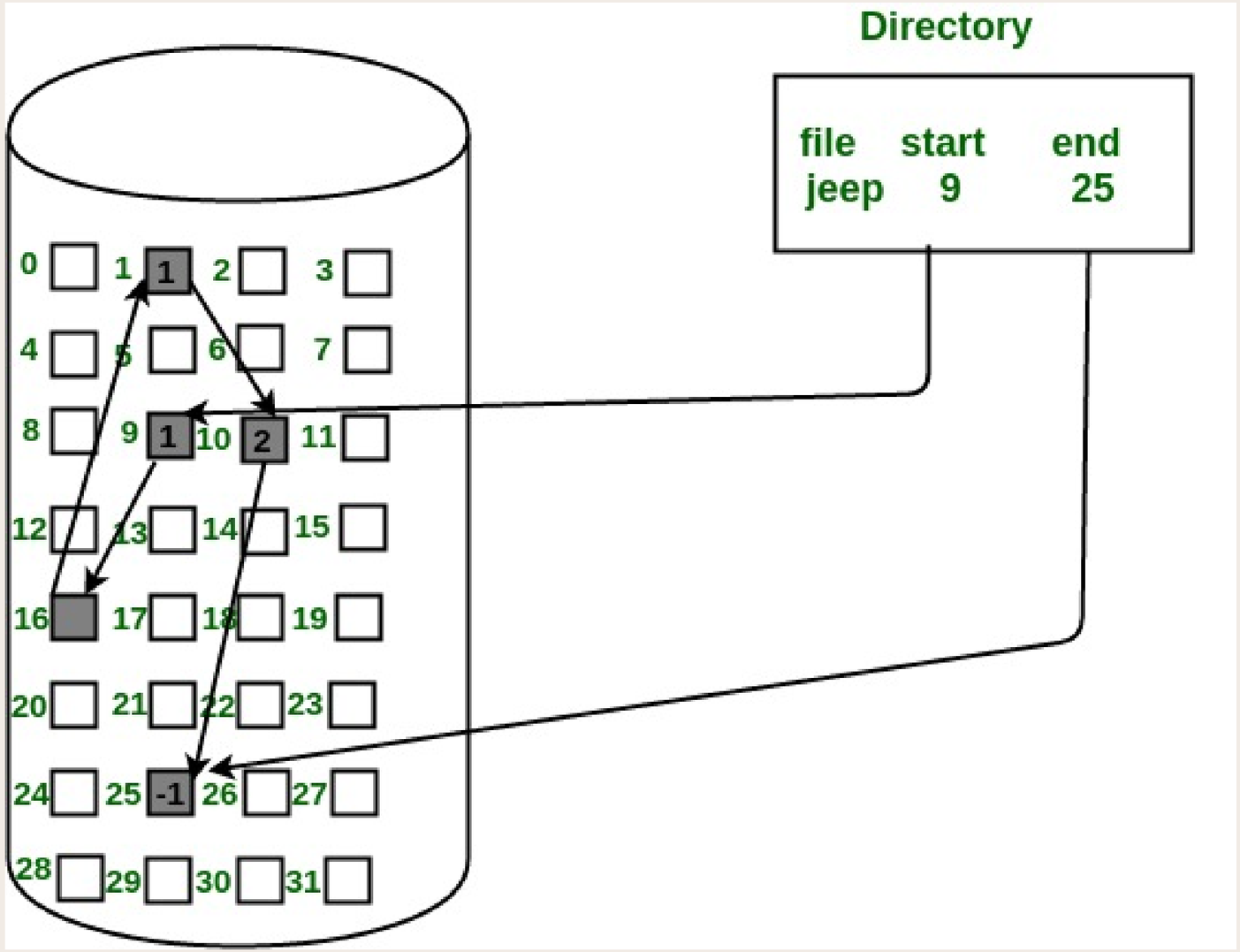


ALLOCATING FILE BLOCKS ON DISK

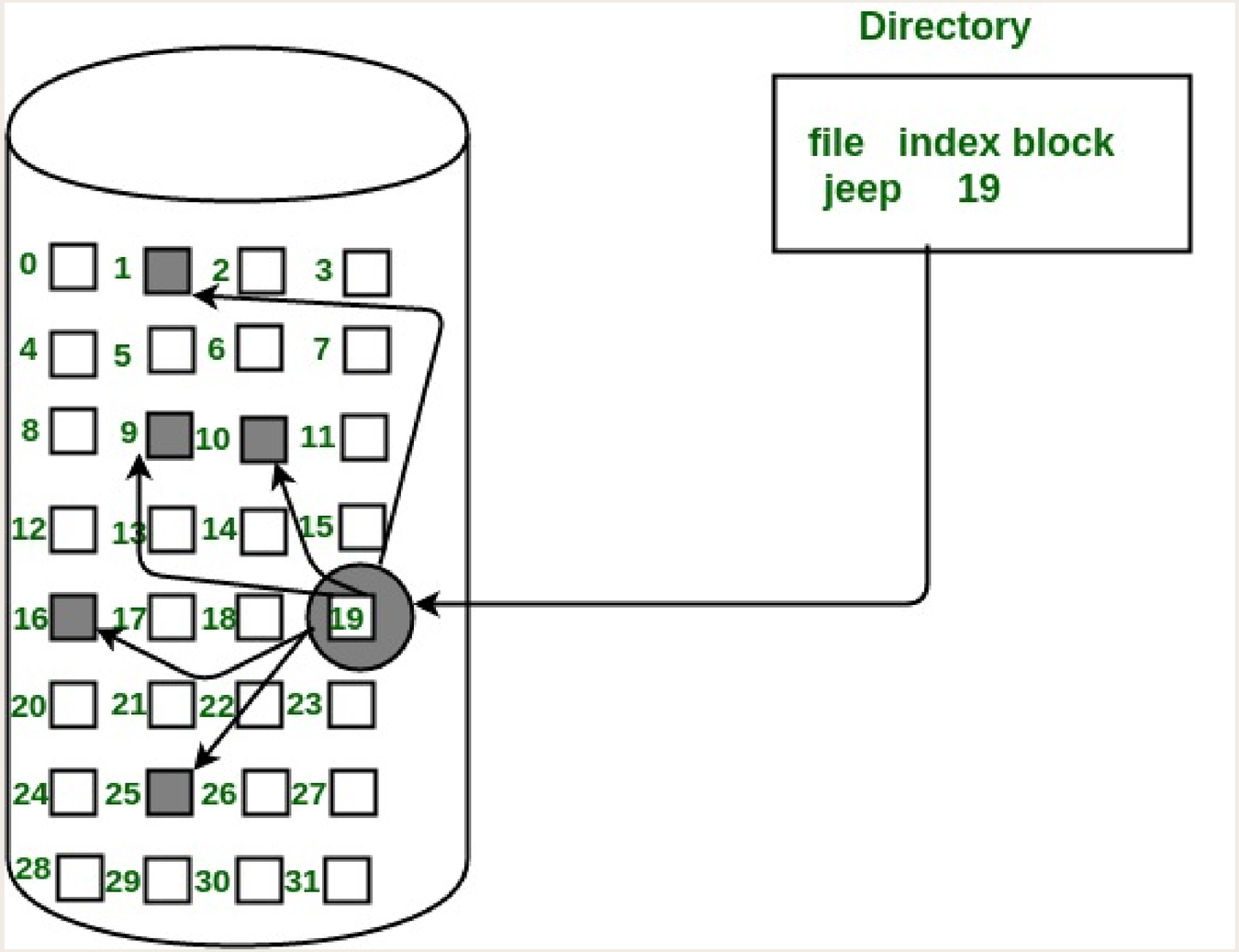
- **Contiguous allocation:** the file blocks are allocated to consecutive disk blocks
- **Linked allocation:** each file block contains a pointer to the next file block
- **Indexed allocation:** one or more index blocks contain pointers to the actual file blocks.
- It is also common to use combinations of these techniques.



Contiguos allocation



Linked allocation



Indexed allocation

OPERATIONS ON FILES

- **OPEN**: Reads the file for access, and associates a pointer that will refer to a current file record at each point in time.
- **FIND**: Searches for the first file record that satisfies a certain condition, and makes it the current file record.
- **FINDNEXT**: Searches for the next file record (from the current record) that satisfies a certain condition, and makes it the current file record.
- **READ**: Reads the current file record into a program variable.
- **INSERT**: Inserts a new record into the file, andmakes it the current file record.

OPERATIONS ON FILES

- **DELETE**: Removes the current file record from the file, usually by marking the record to indicate that it is no longer valid.

- **MODIFY**: Changes the values of some fields of the current file record.

- **CLOSE**: Terminates access to the file.

- **REORGANIZE**: Reorganizes the file records. For example, the records marked deleted are physically removed from the file or a new organization of the file records is created.

- **READ_ORDERED**: Read the file blocks in order of a specific field of the file.

FILES OF UNORDERED RECORDS (HEAP)

New records are inserted at the end of the file.

=> More effecient

Searching for a record requires a linear search through the file.

=> Average access time: b/2

FILES OF ORDERED RECORDS (SEQUENTIAL)

New records must be inserted in the correct order.

=> Less effecient

Searching is done using a binary search.

=> Average access time: log2(b)

Useful if pattern of access matches ordering of records on the file.

Used with an additional access path, called a primary index. (later)

HASHING

- Applies a **hash function** to a **hash field** and yields the address of the disk block in which the record is stored.
- Reduce time to search for data in a large collection.

PARALLELIZING DISK ACCESS USING RAID TECHNOLOGY

RAID (redundant array of independent disks) is a way of storing the same data in different places on multiple hard disks or solid-state drives (SSDs) for data redundancy, performance improvement, or both.

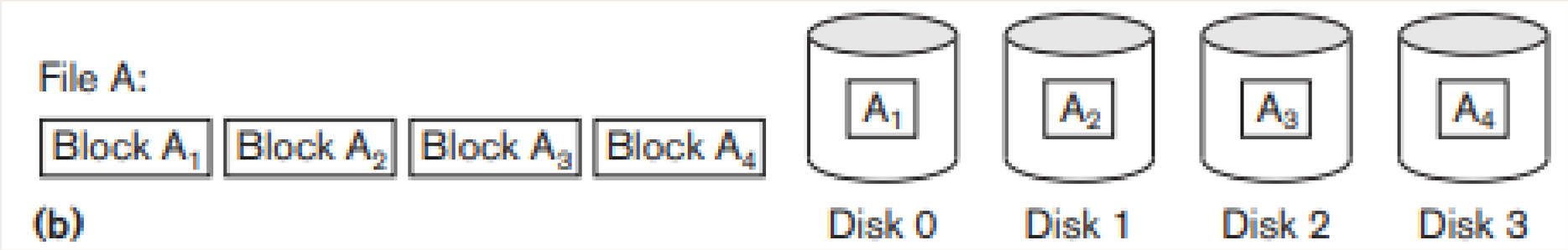
DIFFERENT RATES OF PERFROMANCE IMPROVEMENTS

RAM capacities	Processors	Disk access time	Disk transfer rate
Quadrupled every two to three years	Over 50% per year	Less than 10% per year.	Roughly 20% per year

=> Speed and access time of disk improvements are of a much smaller magnitude

DATA STRIPING

- Utilizes parallelism to improve disk performance.
- Distributes data transparently over multiple disks to make them appear as a single large, fast disk.
- With this striping, every disk participates in every read or write operation; the number of accesses per second would remain the same as on a single disk, but the amount of data read in a given time would increase n-fold.



IMPROVING RELIABILITY WITH RAID

- For an array of n disks, the likelihood of failure is n times as much as that for one disk. Keeping a single copy of data in such an array of disks will cause a significant loss of reliability.
  => Employ redundancy of data so that disk failures can be tolerated.

- Disadvantages are many: additional I/O operations for write, extra computation to maintain redundancy and to do recovery from errors, and additional disk capacity to store redundant information.

TECHNIQUES FOR INTRODUCING REDUNDANCY

Mirroring or shadowing:

- Data is written redundantly to two identical physical disks that are treated as one logical disk.
- If a disk fails, the other disk is used until the first is repaired.

Store extra information:

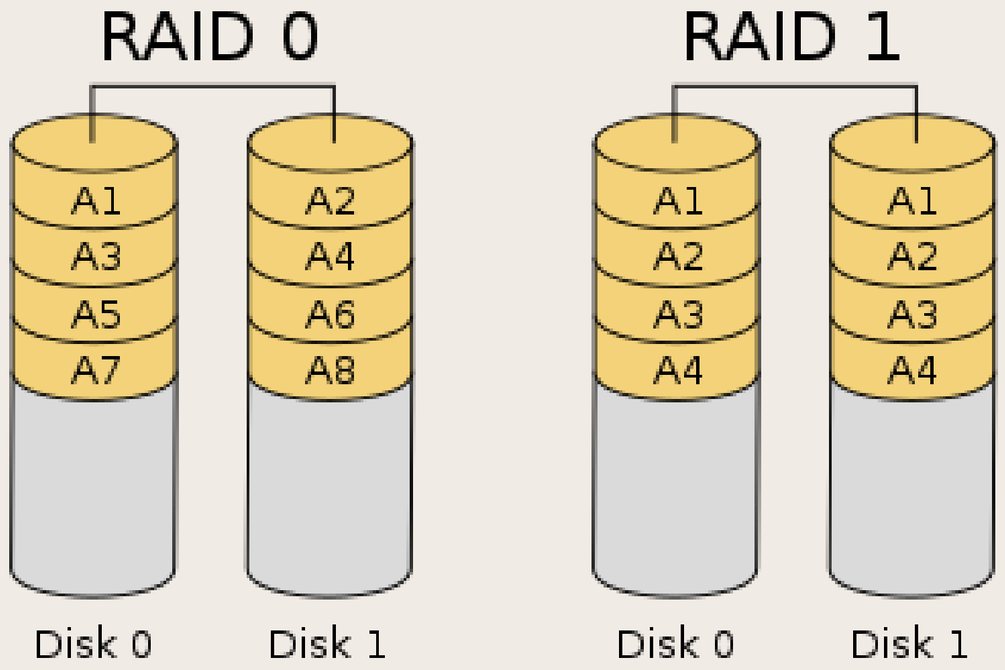
- A technique for computing the redundant information
- A method of distributing the redundant information across the disk array

Different levels of RAID choose a combination of these options.

RAID ORGANIZATIONS AND LEVELS

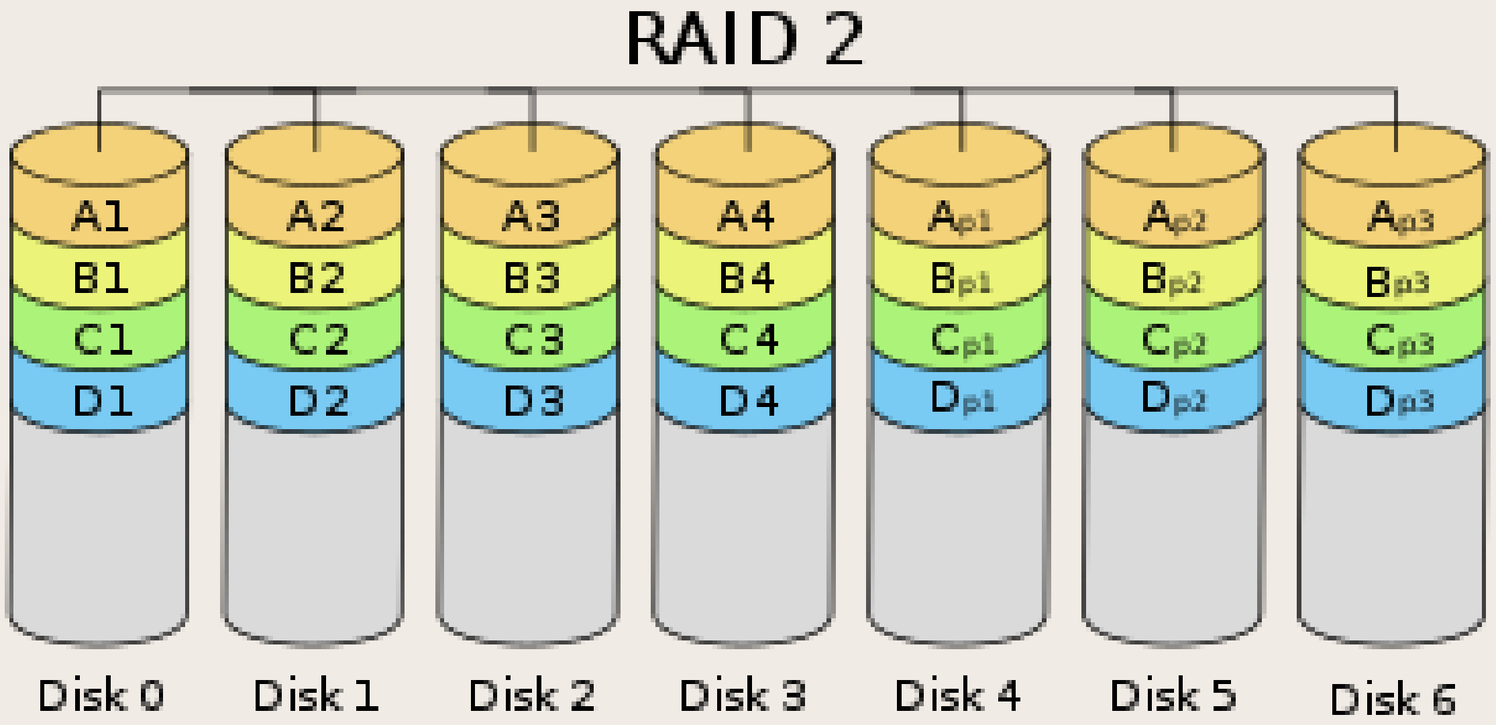
Different raid organizations were defined based on different combinations of the two factors of granularity of data interleaving (striping) and pattern used to compute redundant information.

- Raid level 0 has no redundant data and hence has the best write performance.
- Raid level 1 uses mirrored disks.



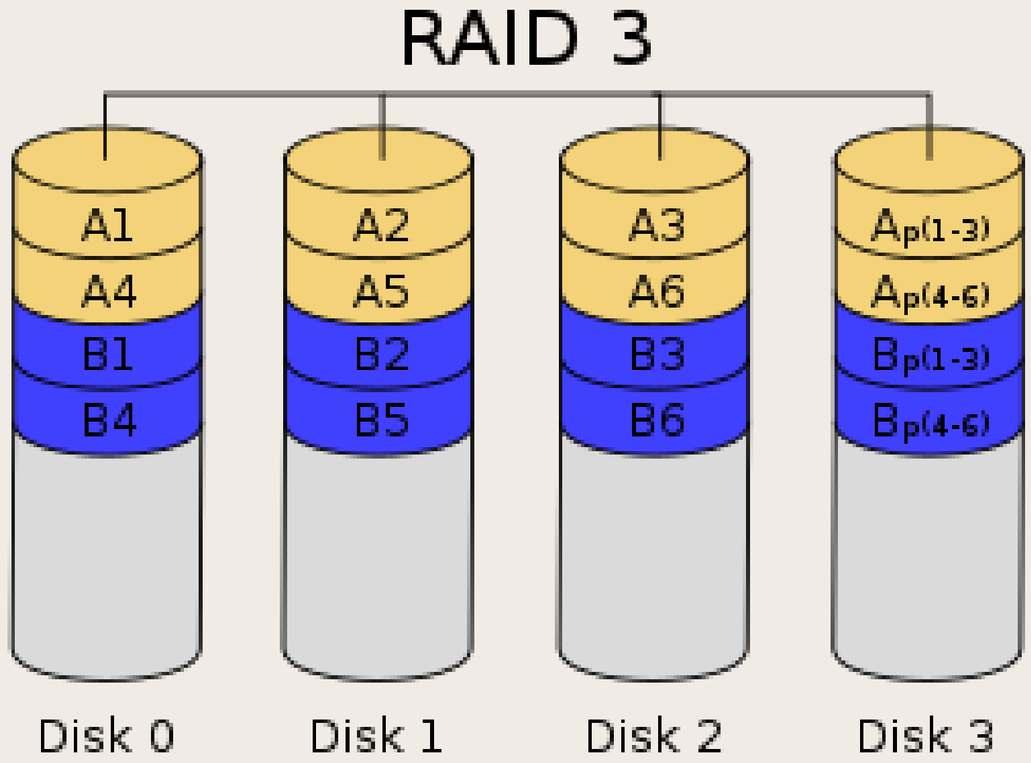
RAID ORGANIZATIONS AND LEVELS

Raid level 2 uses memory-style redundancy by using Hamming codes, which contain parity bits for distinct overlapping subsets of components. Level 2 includes both error detection and correction.



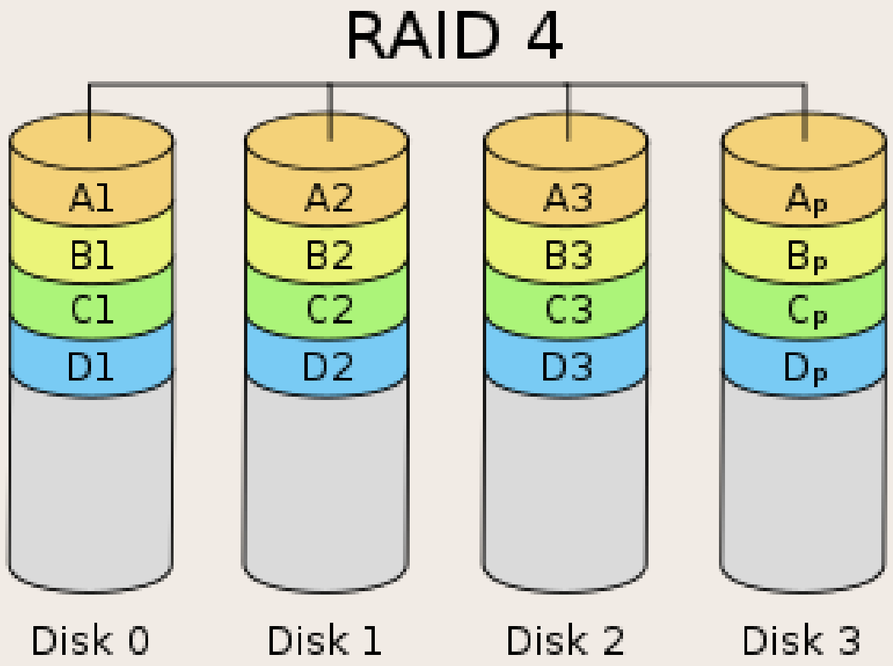
RAID ORGANIZATIONS AND LEVELS

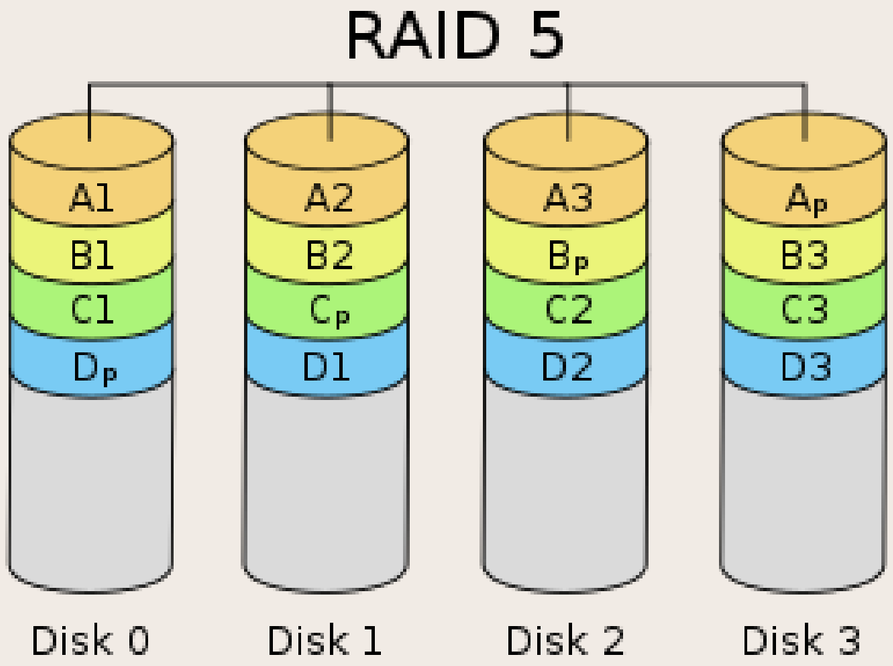
Raid level 3 uses a single parity disk relying on the disk controller to figure out which disk has failed.



RAID ORGANIZATIONS AND LEVELS

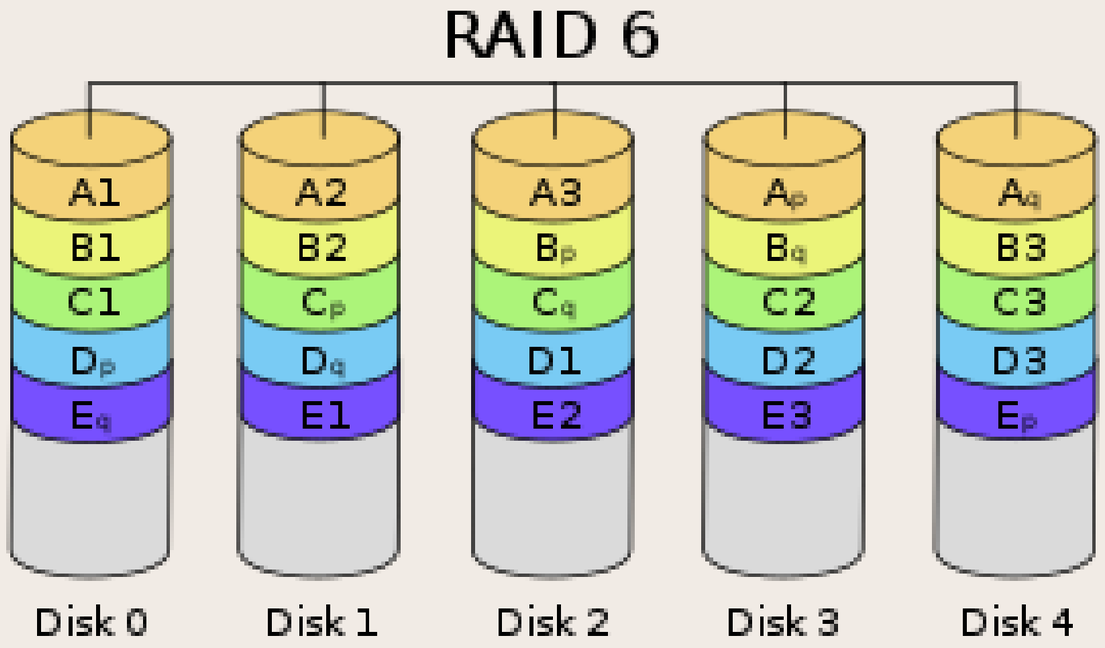
Raid levels 4 and 5 use block-level data striping, with level 5 distributing data and parity information across all disks.





RAID ORGANIZATIONS AND LEVELS

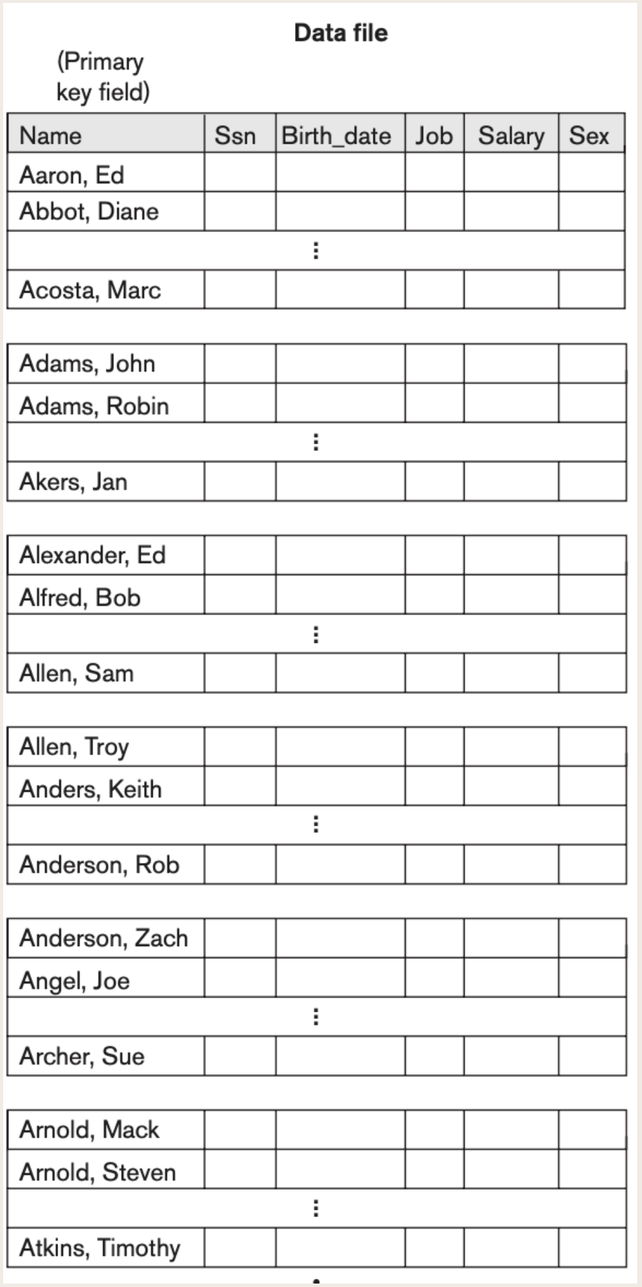
Raid level 6 applies the so-called P + Q redundancy scheme using Reed-Soloman codes to protect against up to two disk failures by using just two redundant disks.



INDEXING STRUCTURES FOR FILES

- SINGLE-LEVEL INDEXES
- MULTI-LEVEL INDEXES
- OTHER TYPE OF INDEXES

WITHOUT INDEX ACCESS



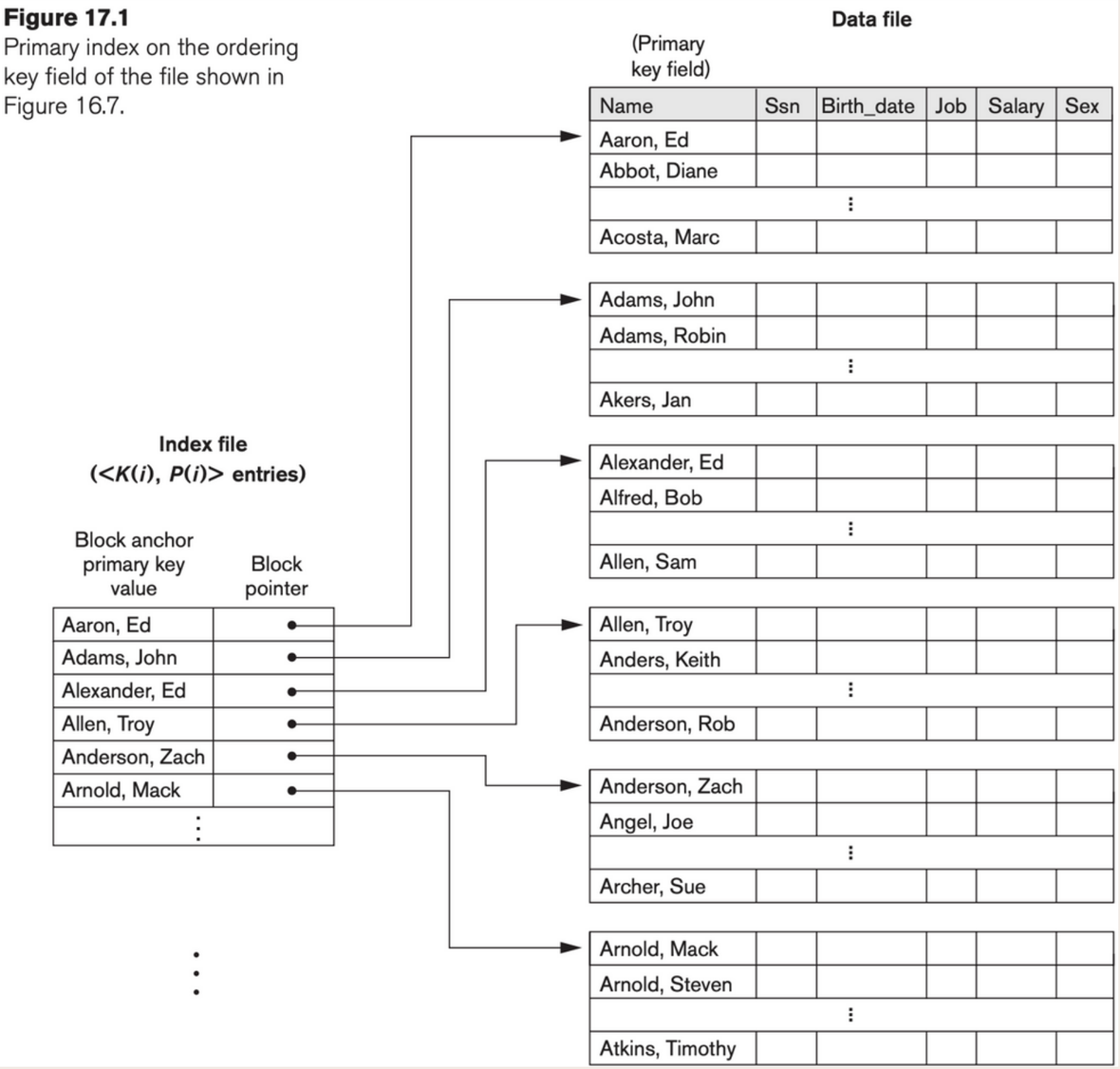
Data file

(Primary key field)

Name	Ssn	Birth_date	Job	Salary	Sex
Aaron, Ed					
Abbot, Diane					
⋮					
Acosta, Marc					
Adams, John					
Adams, Robin					
⋮					
Akers, Jan					
Alexander, Ed					
Alfred, Bob					
⋮					
Allen, Sam					
Allen, Troy					
Anders, Keith					
⋮					
Anderson, Rob					
Anderson, Zach					
Angel, Joe					
⋮					
Archer, Sue					
Arnold, Mack					
Arnold, Steven					
⋮					
Atkins, Timothy					

Search on entire table !

INDEX ACCESSING



Search on indexing file table !

- Index file usually occupies significantly less disk blocks than data files (its entries are much smaller)
- Binary search on index file yields location of block containing desired record
- Dense (1 index for each record) or Non-dense (index for only some records)

=> More efficient to search for a record in the data file

TYPES OF SINGLE-LEVEL ORDERED INDEXES

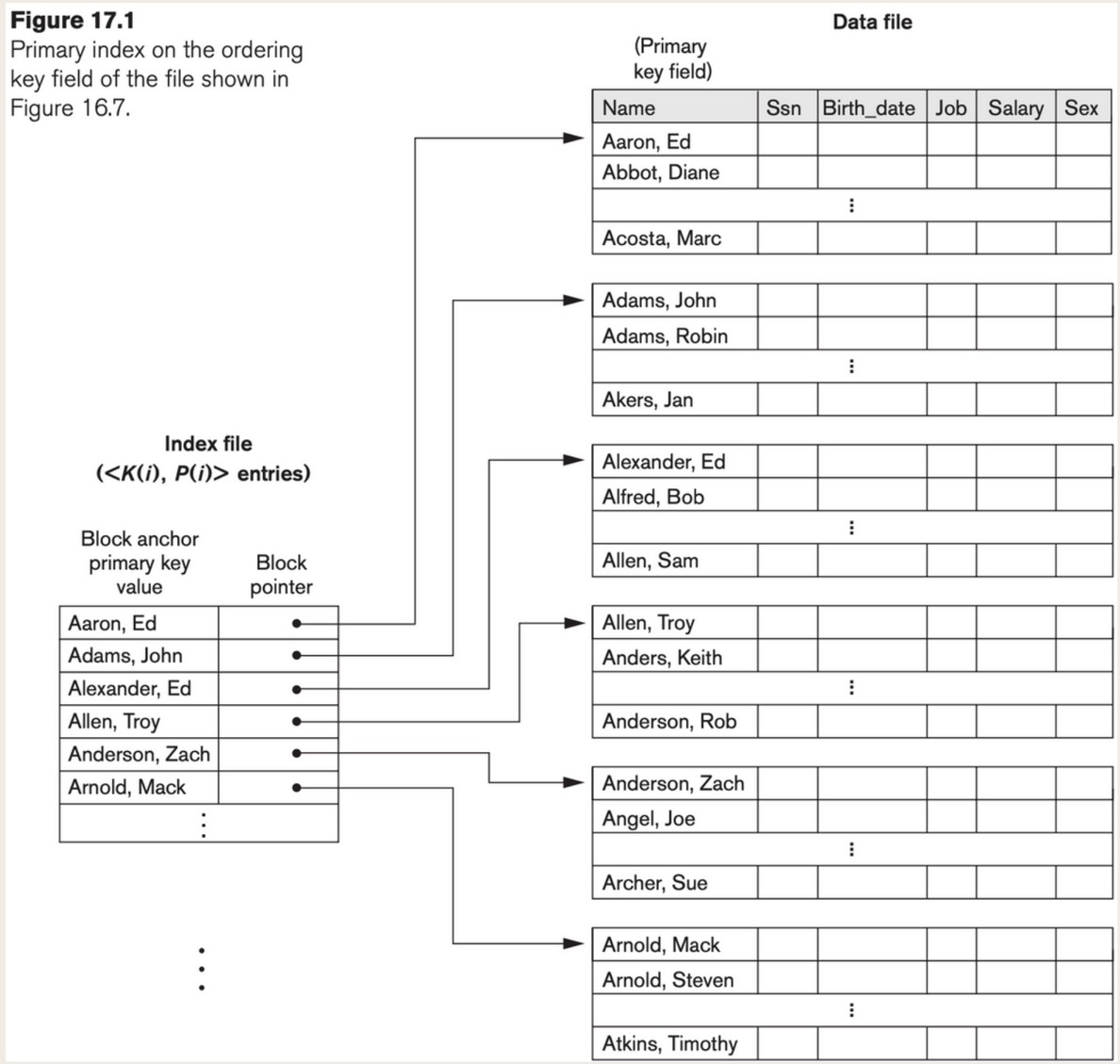
01. PRIMARY INDEXES

02. CLUSTERING INDEXES

03. SECONDARY INDEXES

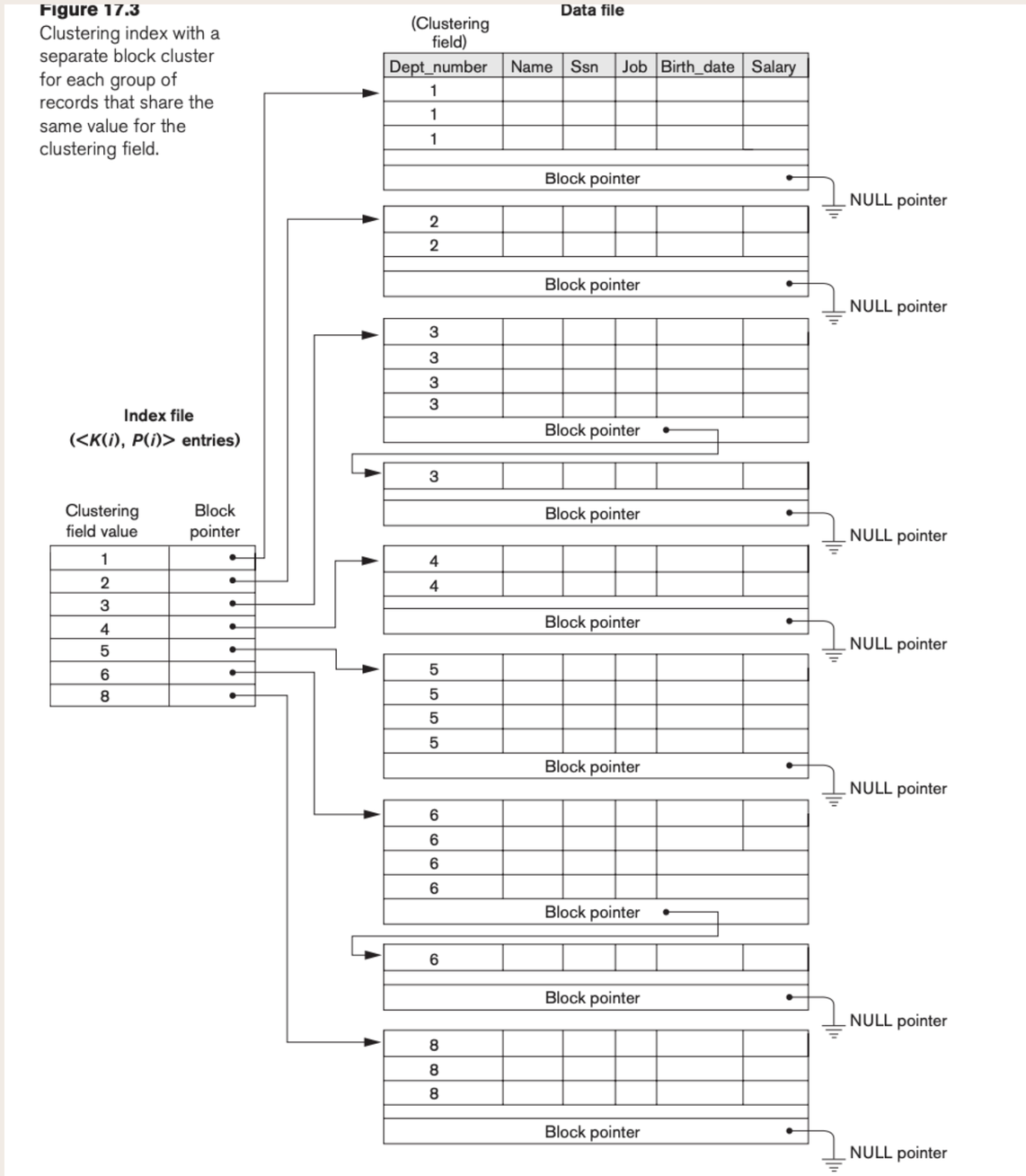
PRIMARY INDEXES

- Data file is ordered on key field
- Each entry correspond to exactly one block in the data file
- Entries contain pointer to first record in the block (block anchor)
    - Number of blocks = Number of index entries
- Non-dense



CLUSTERING INDEXES

- Data file is ordered on non-key field (duplicated values)
- Each entry correspond to distinct values of that field
- Entries contain pointer to first block that contains record with that field value
    - Number of distinct field value = Number of index entries
    - If 1 block cannot hold every record for that field value -> pointer to the next block
- Non-dense



SECONDARY INDEXES

ID	Name	DoB	Salary	Sex
1				
2				
3				
4				
6				
7				
8				
9				
10				
12				
13				
15				

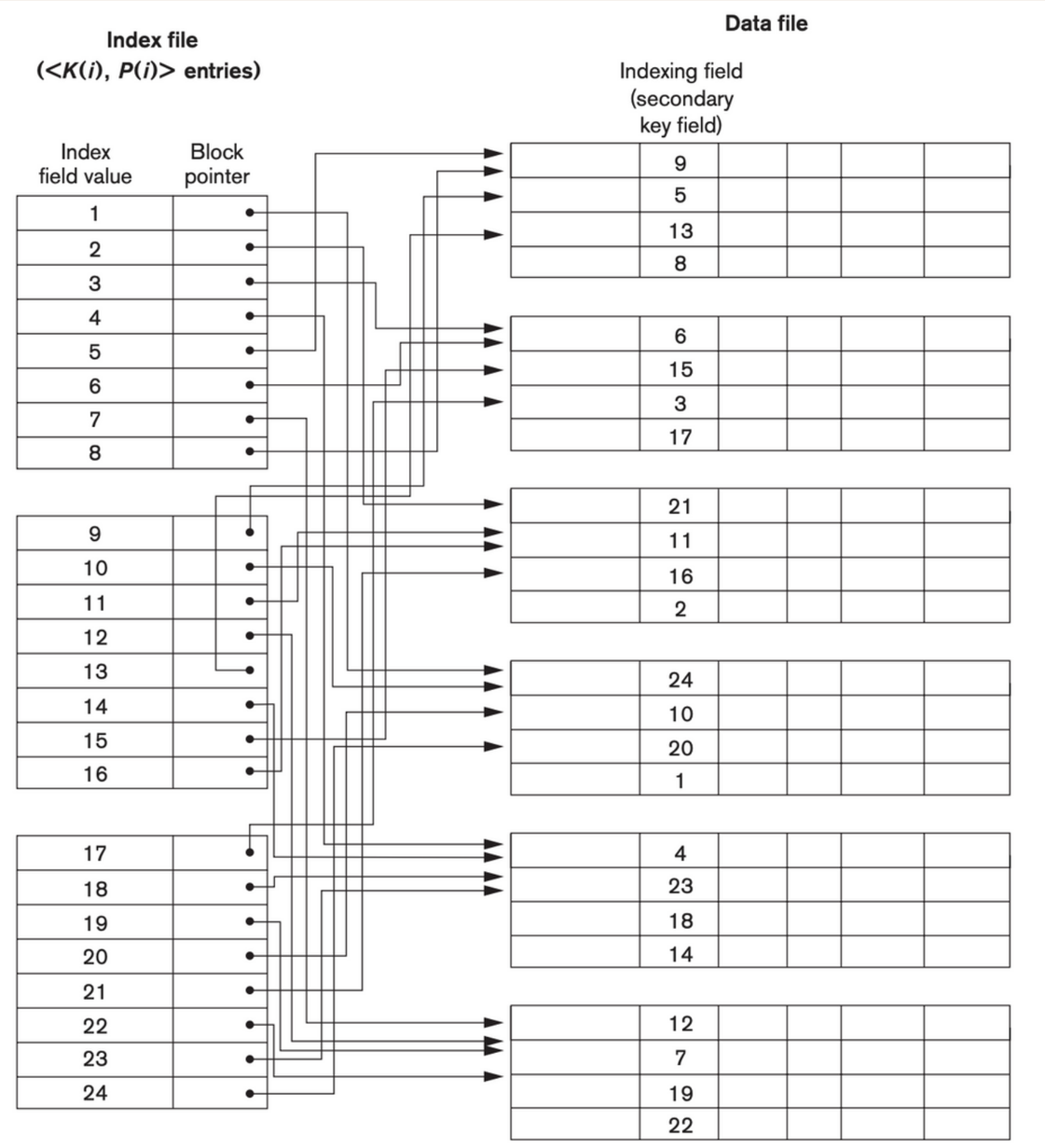
- Data file is sorted on ID and an index file supports searching based on ID
  - What about other fields, how to search for Name, Salary, Sex … ?

SECONDARY INDEXES

- Data file can be unordered on indexing field
- Indexing field can be:
  - Secondary key (unique values)
  - Non-key (duplicated values)
- Index file are ordered on indexing field
- Entries contain indexing field and pointer to block or record
- A data file can have many secondary indexes

SECONDARY INDEXES

On key field



- Data file is unsorted on secondary key field so one index entry is needed for each record => Dense

SECONDARY INDEXES

On non-key field

3 options for structure:

1. Include the duplicated index entries with the same non-key value for each record
2. Keep a list of pointer for each entry to store record with same value
3. Keep 1 entry for each distinct non-key value and **add an extra level of indirection** to handle multiple pointers of record with same value (most popular)

SECONDARY INDEXES

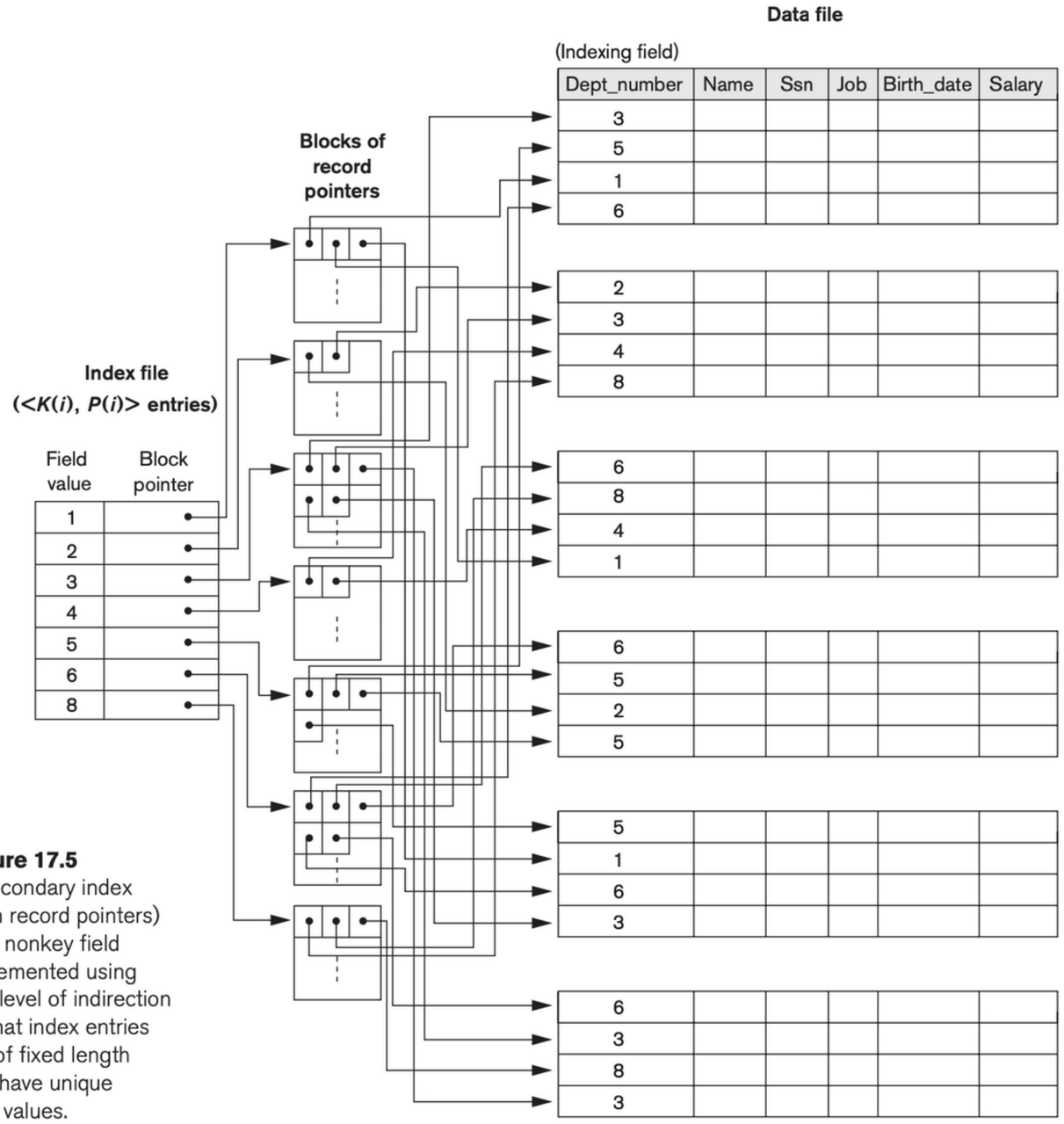


Figure 17.5
A secondary index (with record pointers) on a nonkey field implemented using one level of indirection so that index entries are of fixed length and have unique field values.

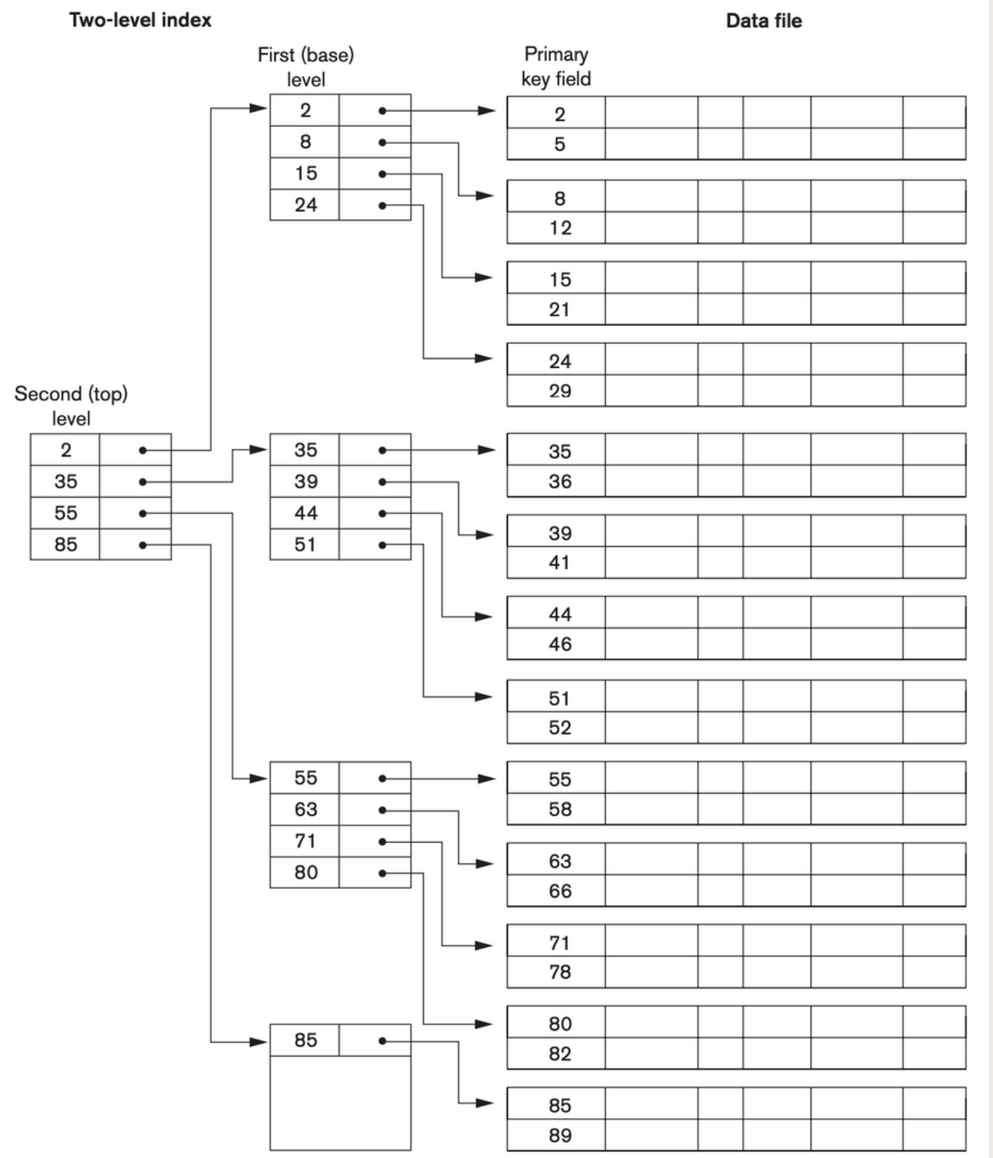
=> Secondary indexing for non-key field is non-dense for option 2 and 3

MULTI-LEVEL INDEXES

- We can create additional indexing files for current indexing files: An index to index file
  - The original index file to records/blocks are called first-level index and the index to the indexes of first-level index are called second-level index
- Create additional indexing files to current indexing files until all entries of highest-level fit in only 1 block
- Can be created with any type of first-level index (primary, clustering and secondary)

MULTI-LEVEL INDEXES

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



- Insertion/deletion causes big burden because each layer of indexing is ordered

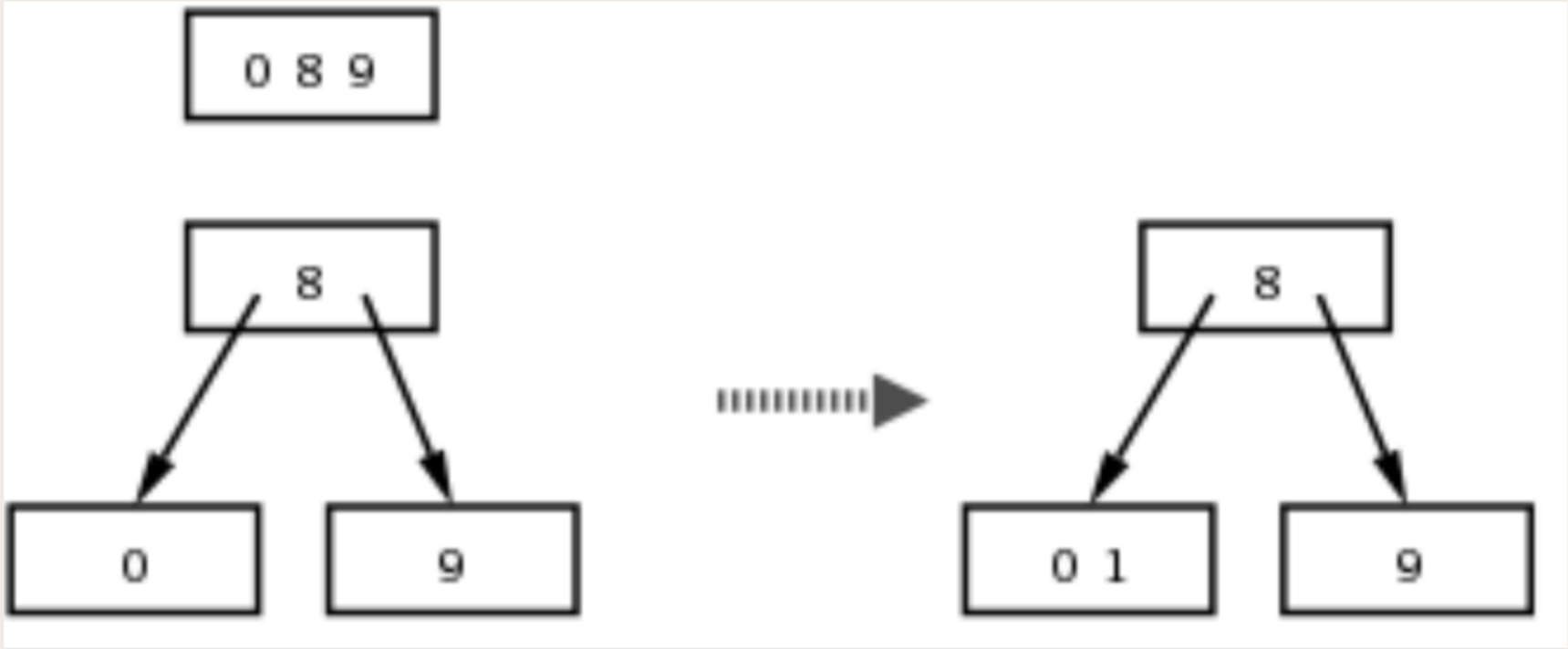
=> Dynamic multi-level indexes ease this problem

B/B+ - TREES OVERVIEW

- Most used in database management systems
- Self - balancing tree data structure
- Each node correspond to a disk block and contains pointers to child nodes
- Highly efficient for range queries and support insertion/deletion of records without complete restructuring of index
- Nodes are kept between full and half full

B/B+ - TREES OVERVIEW

Tree: 0,8,9 (full: 3) -> insert 1

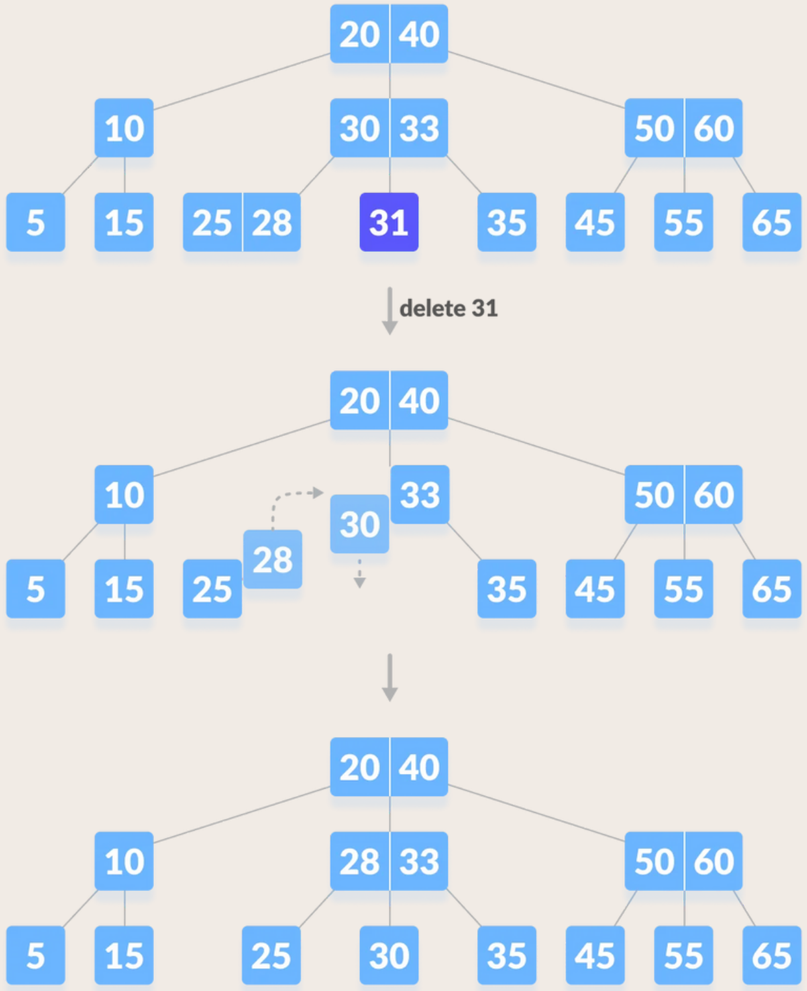


Insertion to half-full nodes are efficient

- If node is full, insertion causes a split into 2 nodes

B/B+ - TREES OVERVIEW

Deletion overview



If deletion causes a node to become half-full, it must be merged with neighboring nodes

OTHER TYPES OF INDEXES

Indexes on multiple keys

Treat a combination of search keys as a search key consists of many attributes - composite keys

- Very efficient for partial match queries
- Works well for ranged queries

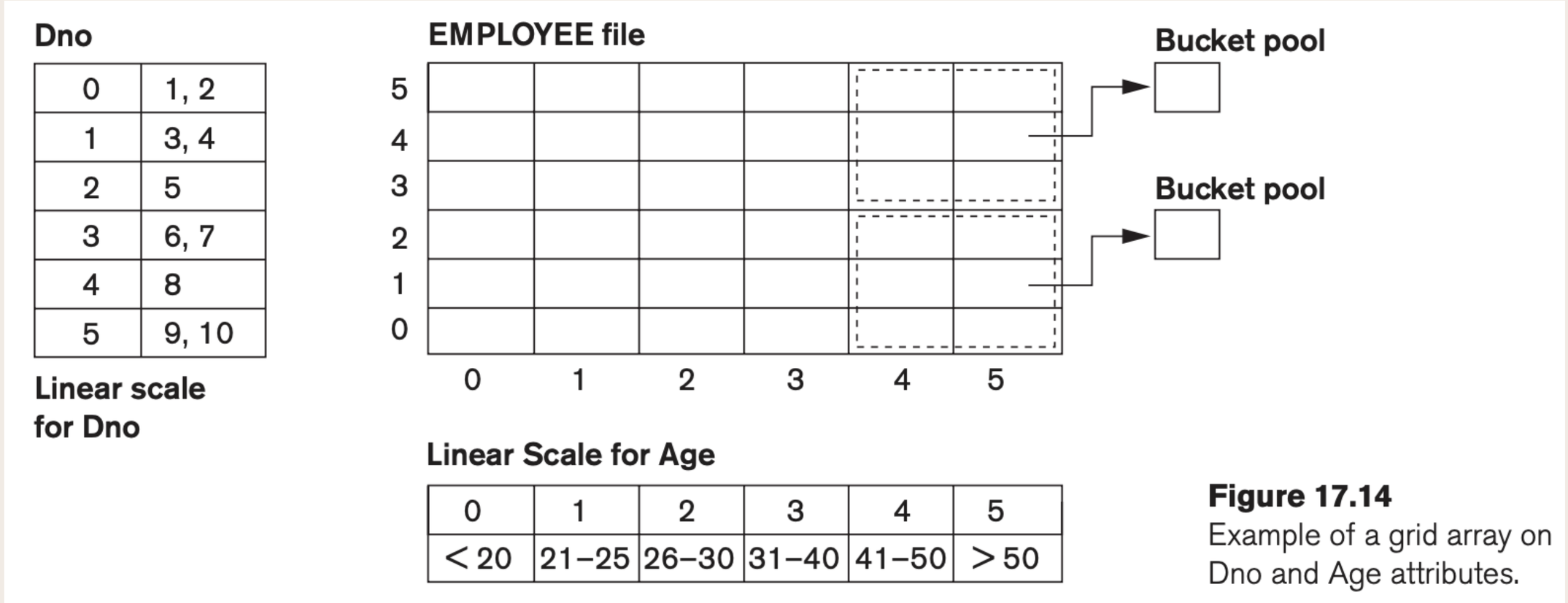
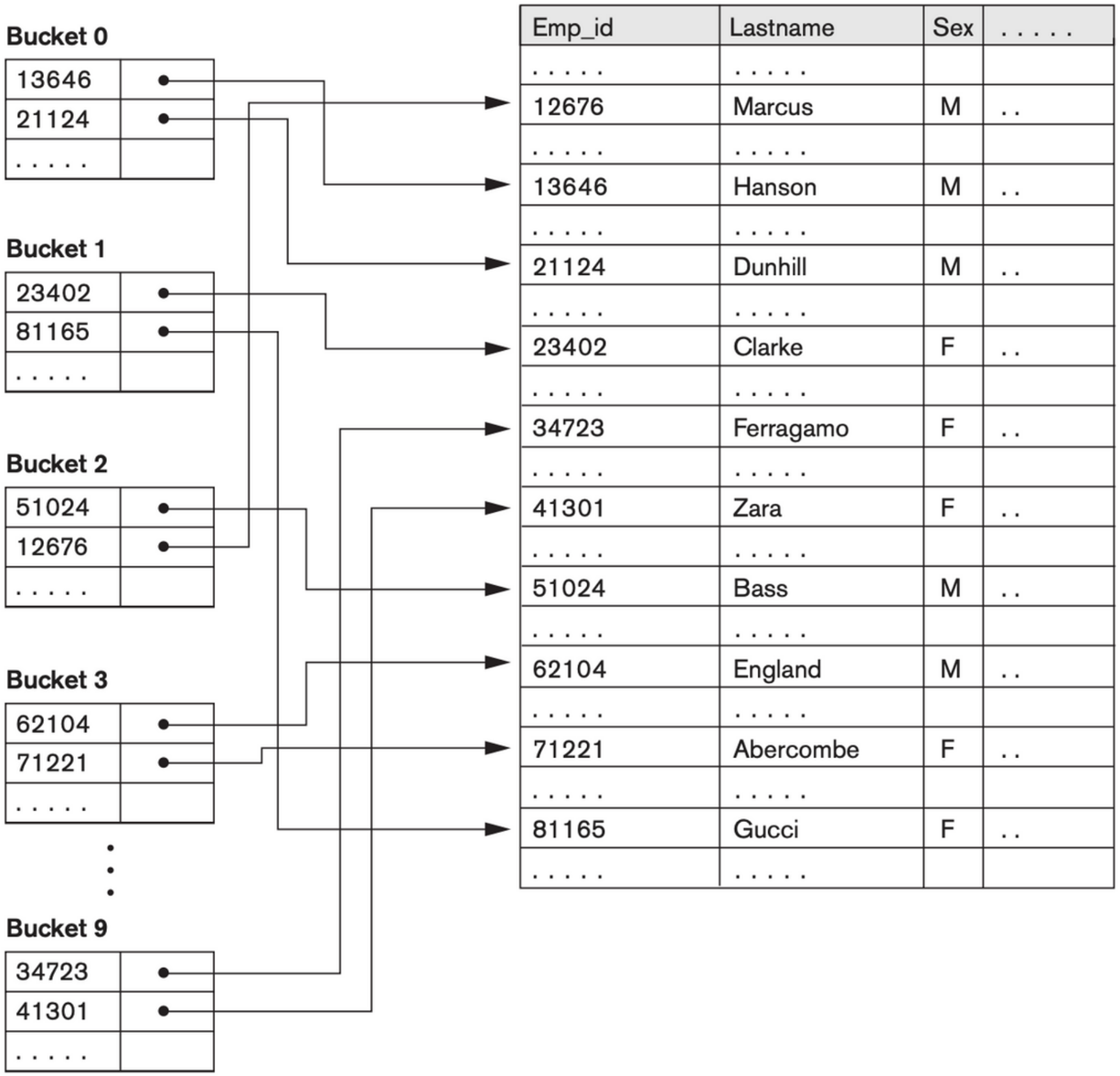


Figure 17.14
Example of a grid array on Dno and Age attributes.

OTHER TYPES OF INDEXES

Hash indexes

Figure 17.15
Hash-based indexing.



A secondary structure to access the file using hasing on a search key other than the one used for primary data file organization

- Very fast query speed (only equal '=' conditions are used)
- Does not support ranged queries

CALCULATING BLOCK ACCESS

An ordered file with r = 300,000 records stored on a disk with block size B = 4,096 bytes .File records are of fixed size and are unspanned, with record length R = 100 bytes.

a. How many block accesses would a binary search need to search for a record?

CALCULATING BLOCK ACCESS

The blocking factor for the file would be

bfr = ⌊(B/R)⌋ = ⌊(4,096/100)⌋ = 40

The number of blocks needed for the file is

b = ⌈(r/bfr)⌉ = ⌈(300,000/40)⌉ = 7,500 blocks.

A binary search on the data file would need approximately

⌈log2 (b)⌉= ⌈(log2 (7,500))⌉ = 13 block accesses

CALCULATING BLOCK ACCESS

b. Suppose that the ordering key field of the file is V = 9 bytes long, a block pointer is P = 6 bytes long. Using primary index, how many block accesses are needed?

CALCULATING BLOCK ACCESS

The size of each index entry is $R_i = (9 + 6) = 15$ bytes

The blocking factor for the index is

$bfr_i = \lfloor (B/R_i) \rfloor = \lfloor (4{,}096/15) \rfloor = 273$ entries per block

The total number of index entries r_i is equal to the number of blocks in the data file, which is 7,500.

The number of index blocks is hence

$b_i = \lceil (r_i/bfr_i) \rceil = \lceil (7{,}500/273) \rceil = 28$ blocks.

Binary search on the index file would need

$\lceil (\log_2 (b_i)) \rceil = \lceil (\log_2(28)) \rceil = 5$ block accesses.

We need one additional block access to the data file for a total of 5 + 1 = 6 block accesses

CALCULATING BLOCK ACCESS

c. Without the secondary index, to do a linear search for a record with a specific value for a nonordering key field would require b/2 = 7,500/2 = 3,750 block accesses on the average. Using primary index, how many block accesses are needed?

CALCULATING BLOCK ACCESS

bfr_i = 273, same as in exercise b, but the secondary index is dense so, the total number of index entries r_i is equal to the number of records in the data file, which is 300,000.

The number of blocks needed for the index is hence

$b_i = \lceil (r_i / bfr_i) \rceil = \lceil (300{,}000/273) \rceil = 1{,}099$ blocks.

A binary search on this secondary index needs

$\lceil (\log_2 (b_i)) \rceil = \lceil (\log_2(1{,}099)) \rceil = 11$ block accesses.

We need an additional block access to the data file for a total of

11 + 1 = 12 block accesses.

CALCULATING BLOCK ACCESS

d. Suppose that the dense secondary index of exercise c is converted into a multilevel index. How many block accesses are needed?

CALCULATING BLOCK ACCESS

bfr_i = 273, same as in exercise b and c, is also the fan-out fo for the multilevel index. The number of first-level blocks b1 = 1,099 blocks, as in exercise c.

The number of second-level blocks will be

b2 = ⌈(b1/fo)⌉ = ⌈(1,099/273)⌉ = 5 blocks

the number of third-level blocks will be

b3 = ⌈(b2/fo)⌉ = ⌈(5/273)⌉ = 1 block.

Hence, the third level is the top level of the index, and t = 3, so we need t + 1 = 3 + 1 = 4 block accesses.

THANK YOU VERY MUCH!