# Database file organization and Indexing

Aims:

At the end of this group of two lectures, you should be able to understand the physical level of databases.

Reading:

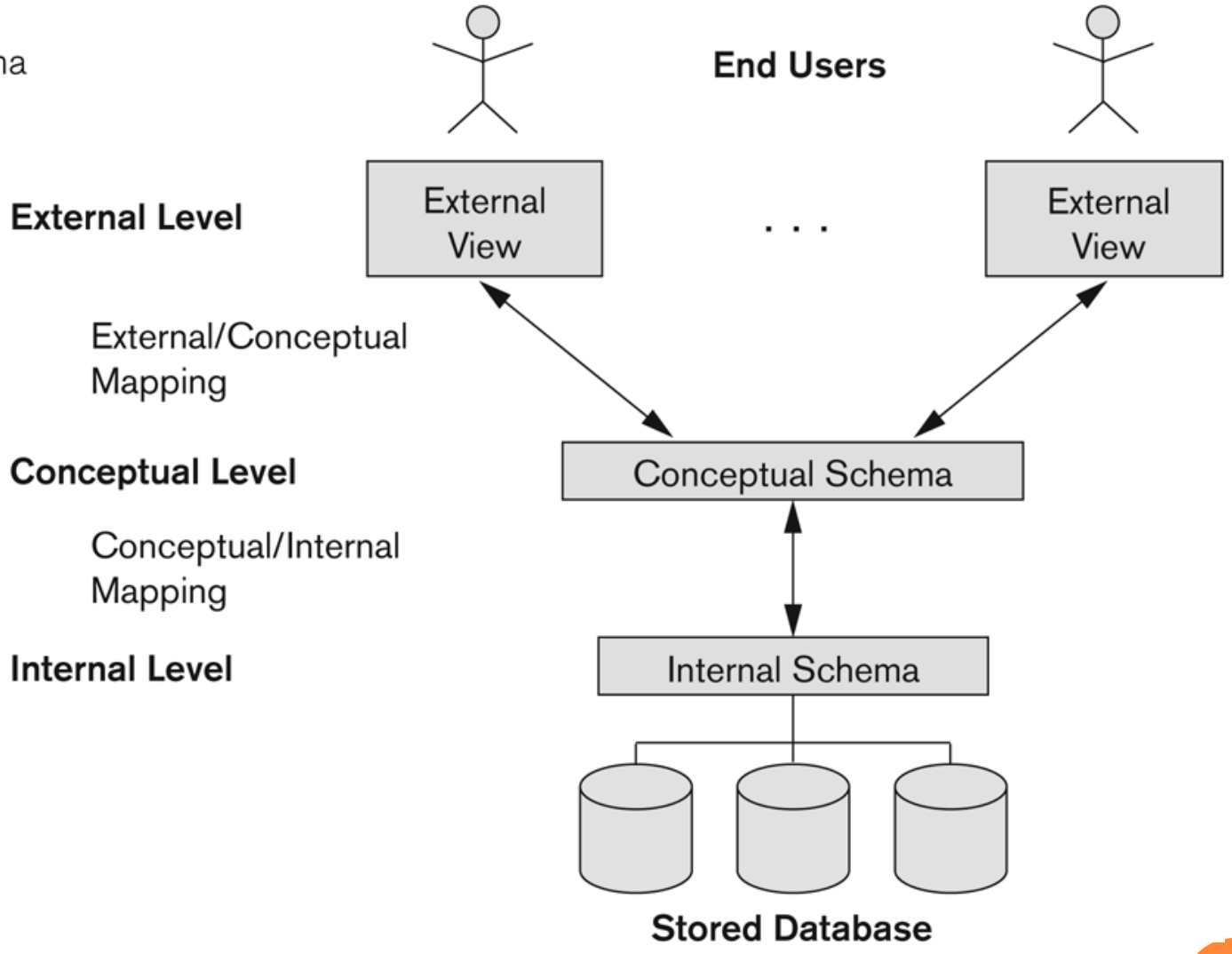
Elmasri & Navathe, Chapters 17 & 18 (6th ed.)

or Chapters 16 & 17 (7th ed.)

# Overview

1. Internal schema
2. Memory hierarchy
3. Disk space management
4. Record organization
5. File organization
6. RAID
7. Indexing
8. Physical level in Oracle

**Figure 2.2**
The three-schema architecture.



# Internal schema

- Efficient storage and index structures are very important
- Heavily depend on the chosen DBMS (no standards)
- Physical design:
  - How to store tables in persistent storage
  - How to speed up access to data

# Database performance

- Secondary storage
- Memory management: buffering, cache
- Indexing strategy
- Query optimization
- Network factors
- Concurrency
- …

# Factors we can influence

- Data types
- Degree of normalization
- Query formulation
- Overhead for constraints, triggers, …
- Indexing strategy
- Media
- …

# Memory hierarchy

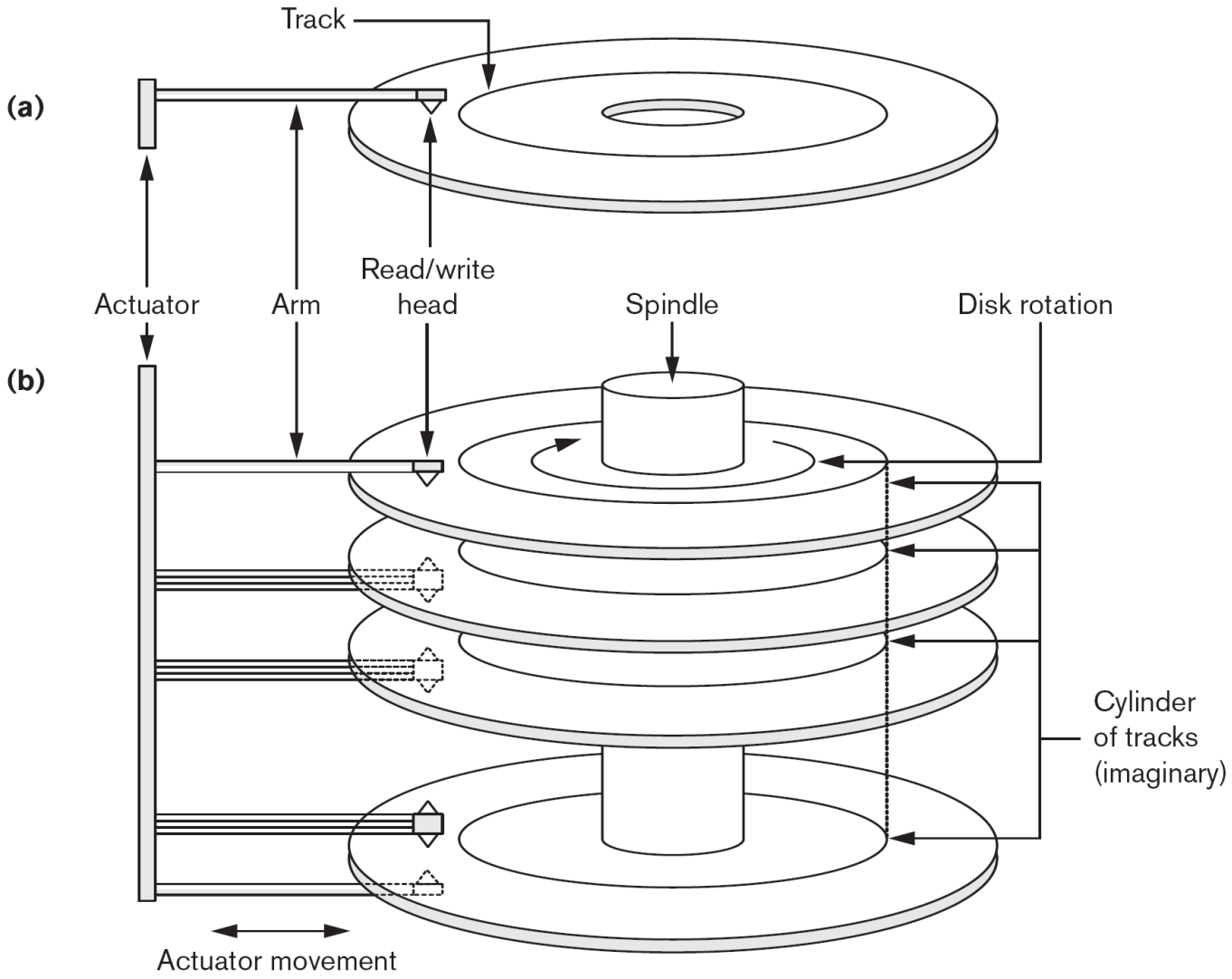
- Primary (cache and main memory)
  - Fast access
  - Limited capacity
  - Volatile
- Secondary and tertiary (magnetic disks, optical disks, tapes)
  - High capacity
  - Low cost
  - Non-volatile

# Disk storage devices

- Data stored as magnetized areas on disk surfaces
- A **disk pack** contains several magnetic disks connected to a rotating spindle.
- Disks are divided into concentric circular **tracks** on each disk **surface**.
- Track capacities vary typically from 4-50 kB or more
- A track is divided into smaller **blocks** or **sectors** (hard-coded)
  - The block size B is fixed for each system
  - Typical block sizes range from 512B to 4kB
  - Whole blocks are transferred between disk and main memory for processing.

# Disk Storage Devices (cont.)

**Figure 17.1**
(a) A single-sided disk with read/write hardware.
(b) A disk pack with read/write hardware.



# Disk Storage Devices (cont.)

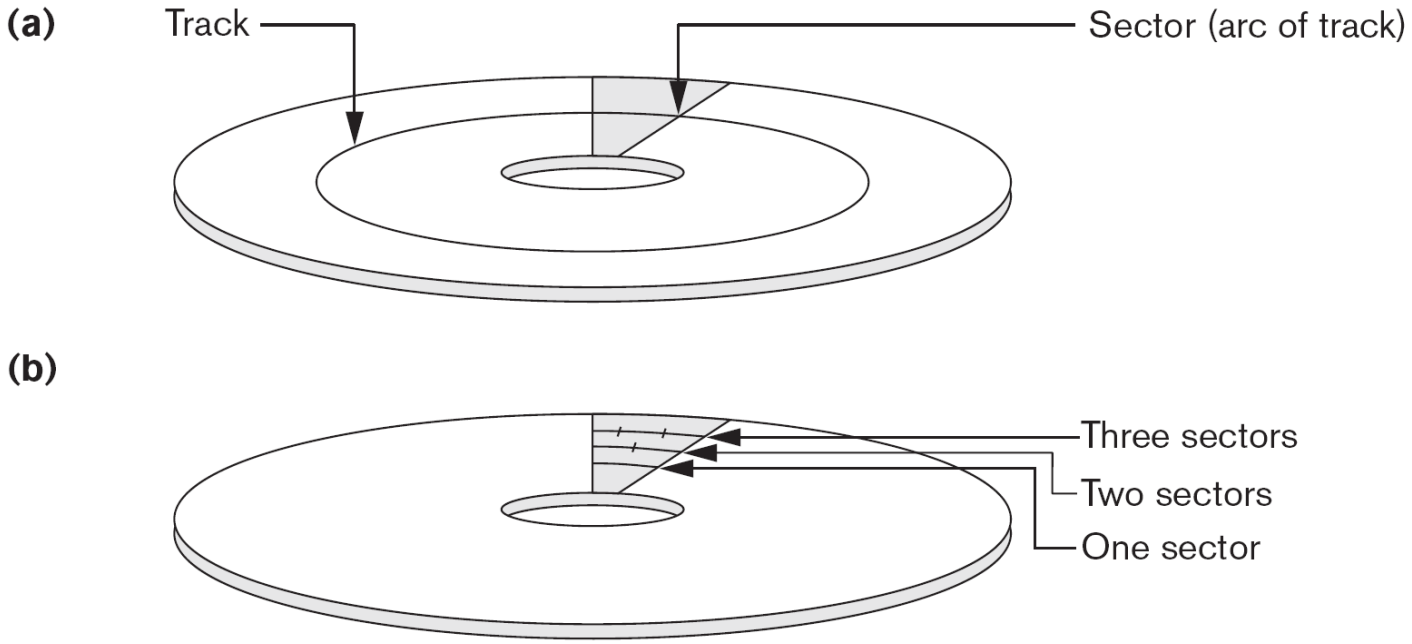



**Figure 17.2**
Different sector organizations on disk. (a) Sectors subtending a fixed angle. (b) Sectors maintaining a uniform recording density.

# Disk Storage Devices (cont.)

- 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 s and rotational delay (latency) **rd**.
- Double buffering can be used to speed up the transfer of contiguous disk blocks.

# Typical Disk Parameters

**Table 17.1** Specifications of Typical High-End Cheetah Disks from Seagate

| Description | Cheetah 15K.6 | Cheetah NS 10K |
|---|---|---|
| Model Number | ST3450856SS/FC | ST3400755FC |
| Height | 25.4 mm | 26.11 mm |
| Width | 101.6 mm | 101.85 mm |
| Length | 146.05 mm | 147 mm |
| Weight | 0.709 kg | 0.771 kg |
| **Capacity** | | |
| Formatted Capacity | 450 Gbytes | 400 Gbytes |
| **Configuration** | | |
| Number of disks (physical) | 4 | 4 |
| Number of heads (physical) | 8 | 8 |
| **Performance** | | |
| **Transfer Rates** | | |
| Internal Transfer Rate (min) | 1051 Mb/sec | |
| Internal Transfer Rate (max) | 2225 Mb/sec | 1211 Mb/sec |
| Mean Time Between Failure (MTBF) | | 1.4 M hours |
| **Seek Times** | | |
| Avg. Seek Time (Read) | 3.4 ms (typical) | 3.9 ms (typical) |
| Avg. Seek Time (Write) | 3.9 ms (typical) | 4.2 ms (typical) |
| Track-to-track, Seek, Read | 0.2 ms (typical) | 0.35 ms (typical) |
| Track-to-track, Seek, Write | 0.4 ms (typical) | 0.35 ms (typical) |
| Average Latency | 2 ms | 2.98 msec |

Courtesy Seagate Technology

# Solid-State Drives

- No actual disks, motors or heads
- Uses integrated circuit assemblies as persistent memory
- Still supports block data organization
- Advantages:
  - More resistant to physical shock
  - Run silently
  - Lower access time
  - Less latency
- Disadvantages
  - Expensive (6-7x traditional disk drives)

# RECORD ORGANIZATION

- A **file** is a *sequence* of records, where each record is a collection of values
- Records contain fields which have values of a particular type (e.g., amount, date, time, age)
- A file can have **fixed-length** records or **variable-length** records.
- Fields themselves may be fixed length or variable length
- Variable length fields can be mixed into one record:
  - Separator characters or length fields are needed

# BLOCKING

- **Blocking**: storing a number of records in one block
- 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.
- File records can be **unspanned** or **spanned**
  - **Unspanned**: no record can span two blocks
  - **Spanned**: a record can be stored in more than one block

# FILE ORGANIZATION

- DBMS interacts with OS & file system to read/write physical blocks (pages) between disc and memory.
- All DB data are stored on one or more disks, which are managed by DBMS.
- DBMSs may have their own file system.

Question:Why do DBMSs generally not store all data in (primary) memory?

# In-Memory Database Systems

- A database system that relies (primarily) on main memory for data storage
- Advantages – faster access
- Disadvantage – loss of persistence if system crashes
- Evolution – use of non-volatile random access memory (NVRAM) such as flash memory used on solid-state drives
- Still, issues remain: larger block writing requirements not always supported, lower longevity for such devices, performance limitations, cost
- Example product: MemSQL (http://memsql.com)

# FILE ORGANIZATION

- 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.
- The physical disk blocks that are allocated to hold the records of a file can be *contiguous, linked, or indexed*.
- In a file of fixed-length records, all records have the same format. Usually, unspanned blocking is used with such files.
- Files of variable-length records require additional information to be stored in each record, such as **separator characters** and **field types**.

# Operation on Files

- **OPEN**: Readies the file, a pointer refers to a *current* record
- **FIND**: Searches for the first record that satisfies a certain condition, and makes it the current record.
- **FINDNEXT**: Searches for the next record (from the current record) that satisfies a certain condition, and makes it the current record.
- **READ**: Reads the current file record into a program variable.
- **INSERT**: Inserts a new record into the file & makes it the current file record.
- **DELETE**: Removes the current file record, usually by marking the record to indicate that it is no longer valid.
- **MODIFY**: Changes the values of some fields of the current 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.

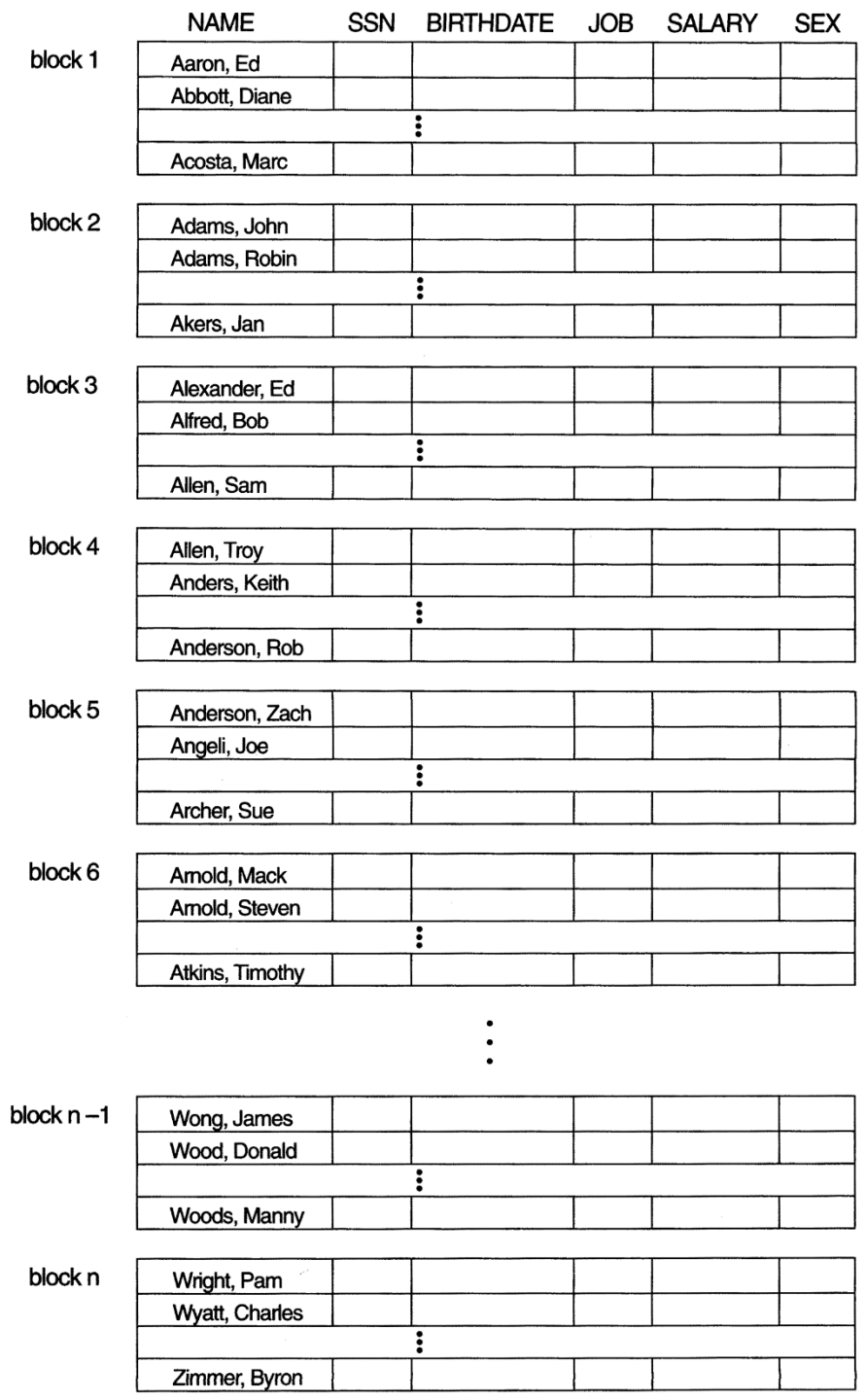
# Unordered Files

- Also called a **heap** or a **pile** file
- Efficient insertion: new records are inserted at the end of the file
- However, a **linear search** through the file records is necessary to search for a record
  - This requires reading and searching half the file blocks on the average, and is hence quite expensive.
- Reading the records in order of a particular field requires sorting the file records

# Ordered Files

- Records are sorted by an *ordering field*
- Expensive update: records must be inserted in the correct order.
  - It is common to keep a separate unordered *overflow* (or *transaction*) file for new records to improve insertion efficiency; this is periodically merged with the main ordered file.
- A **binary search** can be used to search for a record on its *ordering field* value.
  - This requires reading and searching $\log_2$ of the file blocks on the average, an improvement over linear search.
- Reading the records in order of the ordering field is quite efficient.

# Ordered Files (cont.)



# Average Access Times

- The following table shows the average access time to access a specific record for a given type of file

**Table 17.2** Average Access Times for a File of *b* Blocks under Basic File Organizations

| Type of Organization | Access/Search Method | Average Blocks to Access a Specific Record |
|---|---|---|
| Heap (unordered) | Sequential scan (linear search) | $b/2$ |
| Ordered | Sequential scan | $b/2$ |
| Ordered | Binary search | $\log_2 b$ |

# Hashed Files

- Hashing for disk files - **External Hashing**
- The file blocks are divided into M equal-sized **buckets**
  - Typically, a bucket corresponds to one (or a fixed number of) disk block(s).
- H**ash key** is computed for each file record
- The record with hash key value K is stored in bucket i, where i=h(K), and h is the **hashing function**.
- Search is very efficient on the hash key.
- Collisions occur when a new record hashes to a bucket that is already full.
  - An overflow file is kept for storing such records.
  - Overflow records that hash to each bucket can be linked together.

# Hashed Files (cont.)

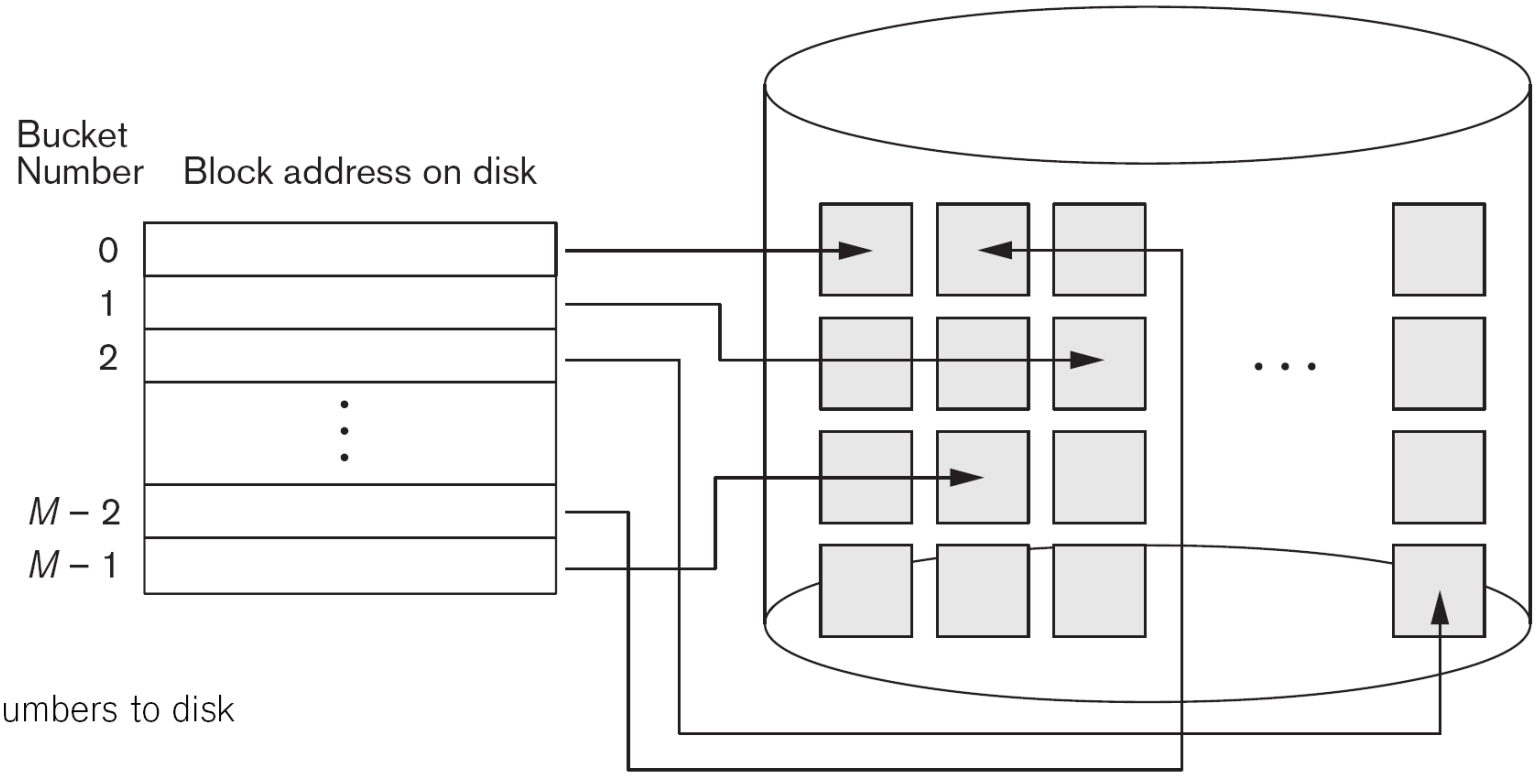



**Figure 17.9**
Matching bucket numbers to disk block addresses.

# Hashed Files (cont.)

- Methods for collision resolution:
  - **Open addressing**: Proceeding from the occupied position specified by the hash address, the program checks the subsequent positions in order until an unused (empty) position is found.
  - **Chaining**: For this method, various overflow locations are kept, usually by extending the array with a number of overflow positions. In addition, a pointer field is added to each record location. A collision is resolved by placing the new record in an unused overflow location and setting the pointer of the occupied hash address location to the address of that overflow location.
  - **Multiple hashing**: The program applies a second hash function if the first results in a collision. If another collision results, the program uses open addressing or applies a third hash function and then uses open addressing if necessary.

# Hashed Files (cont.)

- To reduce overflow records, a hash file is typically kept 70-80% full.
- The hash function h should distribute the records uniformly among the buckets
  - Otherwise, search time will be increased because many overflow records will exist.
- Main disadvantages of static external hashing:
  - Fixed number of buckets M is a problem if the number of records in the file grows or shrinks.
  - Ordered access on the hash key is quite inefficient (requires sorting the records).

# Hashed Files - Overflow Handling

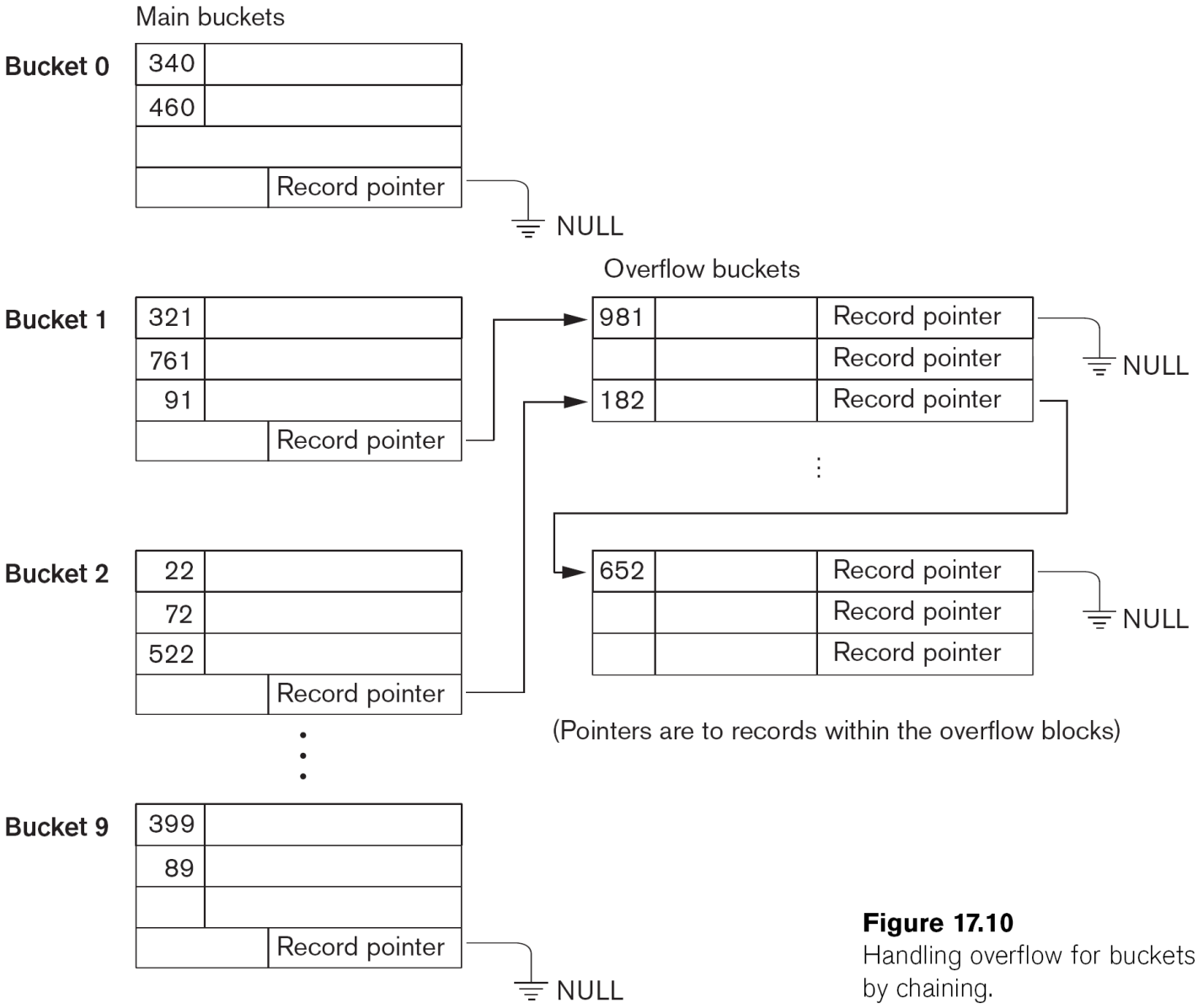


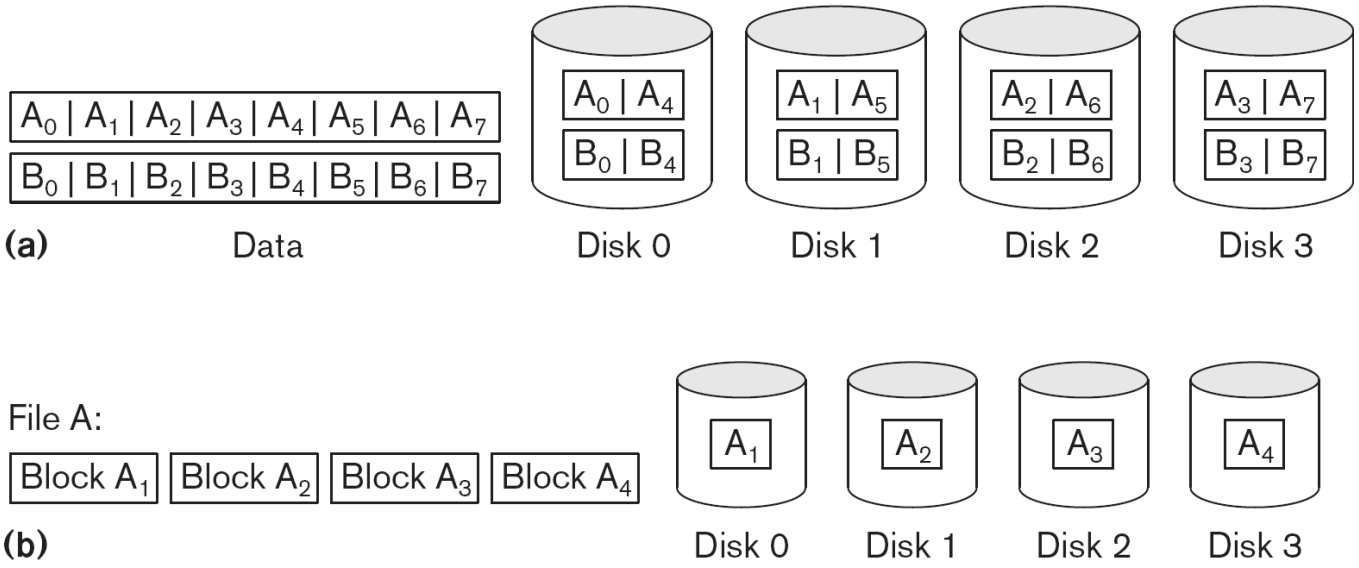

**Figure 17.10**
Handling overflow for buckets by chaining.

# Parallelizing Disk Access using RAID

- **Redundant Arrays of Independent Disks**
- A disk array with data distributed over all disks, appearing to the user as a single large disk
- The main goal of RAID is to even out the widely different rates of performance improvement of disks against those in memory and microprocessors.
- A concept called **data striping** is used, which utilizes parallelism to improve disk performance
- Stripping unit: bit, byte or block

# RAID Technology (cont.)

**Figure 17.13**
Striping of data across multiple disks. (a) Bit-level striping across four disks. (b) Block-level striping across four disks.



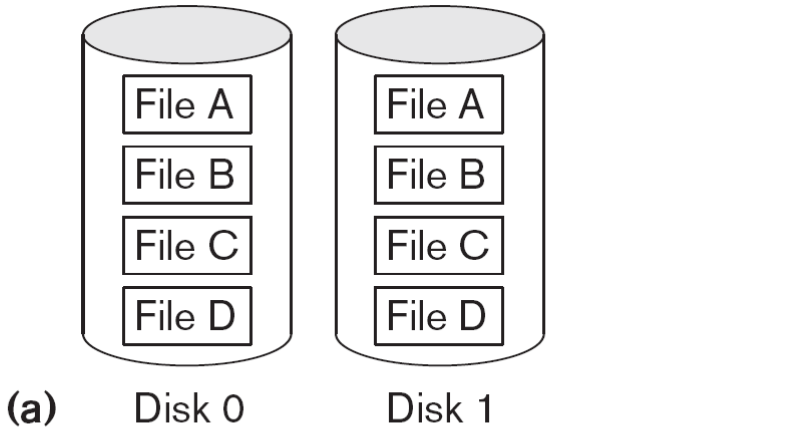
# RAID levels

- Level 0: striping, but no redundant data – the best write performance at the risk of data loss
- Level 1: mirrored disks (space utilization 50%)
- Level 2: 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
- Level 3: single parity disk relying on the disk controller to figure out which disk has failed
- Levels 4 and 5: block-level data striping, with level 5 distributing data and parity information across all disks
- Level 6: P + Q redundancy scheme using Reed-Solomon codes to protect against up to two disk failures by using just two redundant disks.
- Also combinations: e.g. RAID 10 – combination of RAID 0 and RAID 1

# Use of RAID Technology (cont.)

- Different raid organizations are being used under different situations
  - RAID 1 (mirrored disks) is the easiest for rebuild of a disk from other disks; used for critical applications like logs
  - RAID 2 includes both error detection and correction, but as of 2014 no longer used commercially
  - RAID 3 (single parity disks relying on the disk controller to figure out which disk has failed) and RAID 6 (block-level data striping with distributed parity) are preferred for large volume storage, with level 3 giving higher transfer rates but not commonly used in practice.
- Most popular uses of the RAID technology currently are:
  - RAID 0 (with striping), RAID 1 (with mirroring), RAID 6 with an extra drive for parity, and RAID 10
- Design Decisions for RAID include:
  - Level of RAID, number of disks, choice of parity schemes, and grouping of disks for block-level striping.

# Use of RAID Technology (cont.)



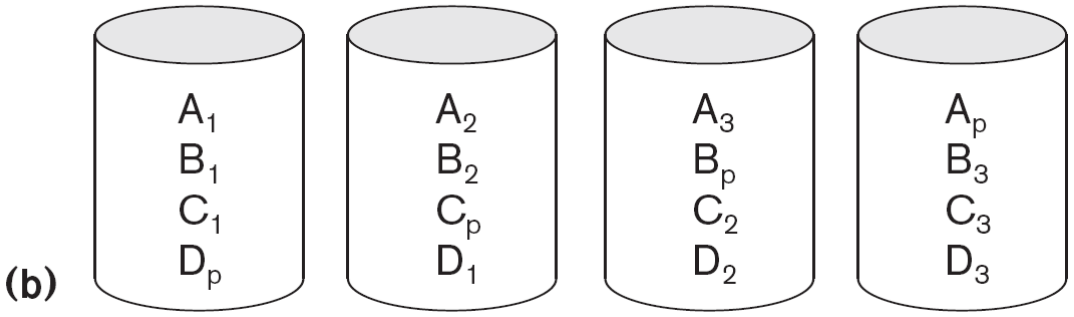
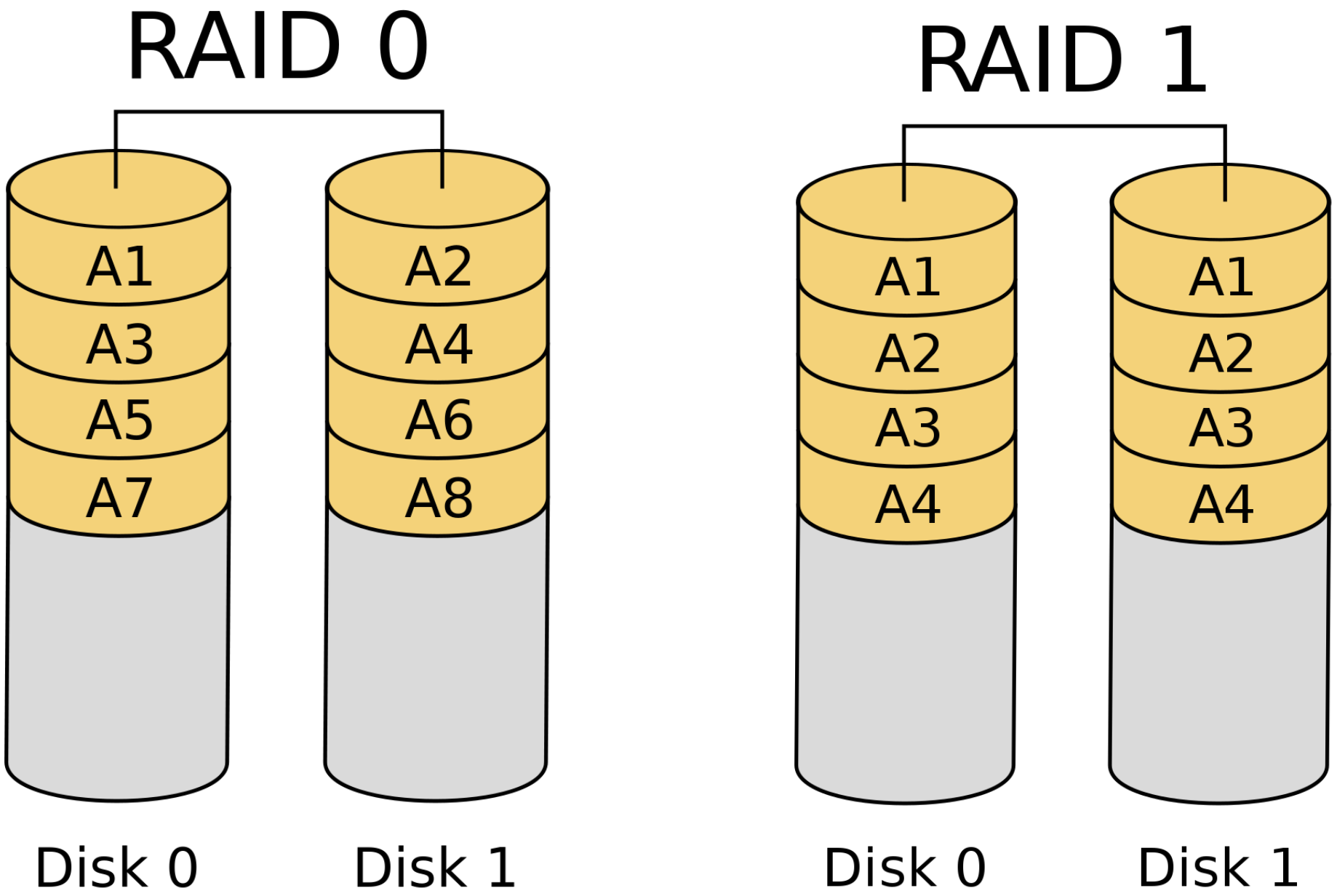
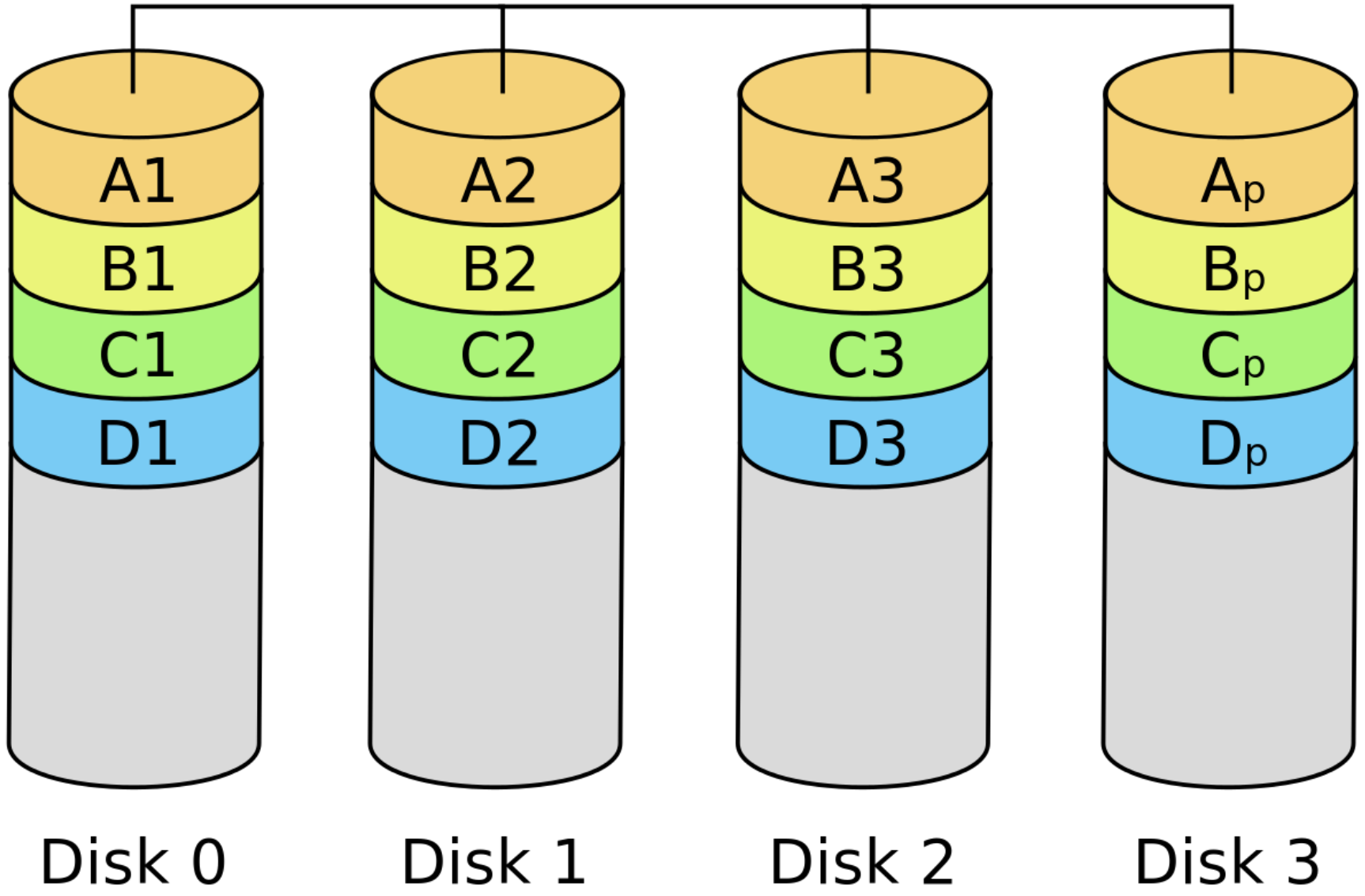



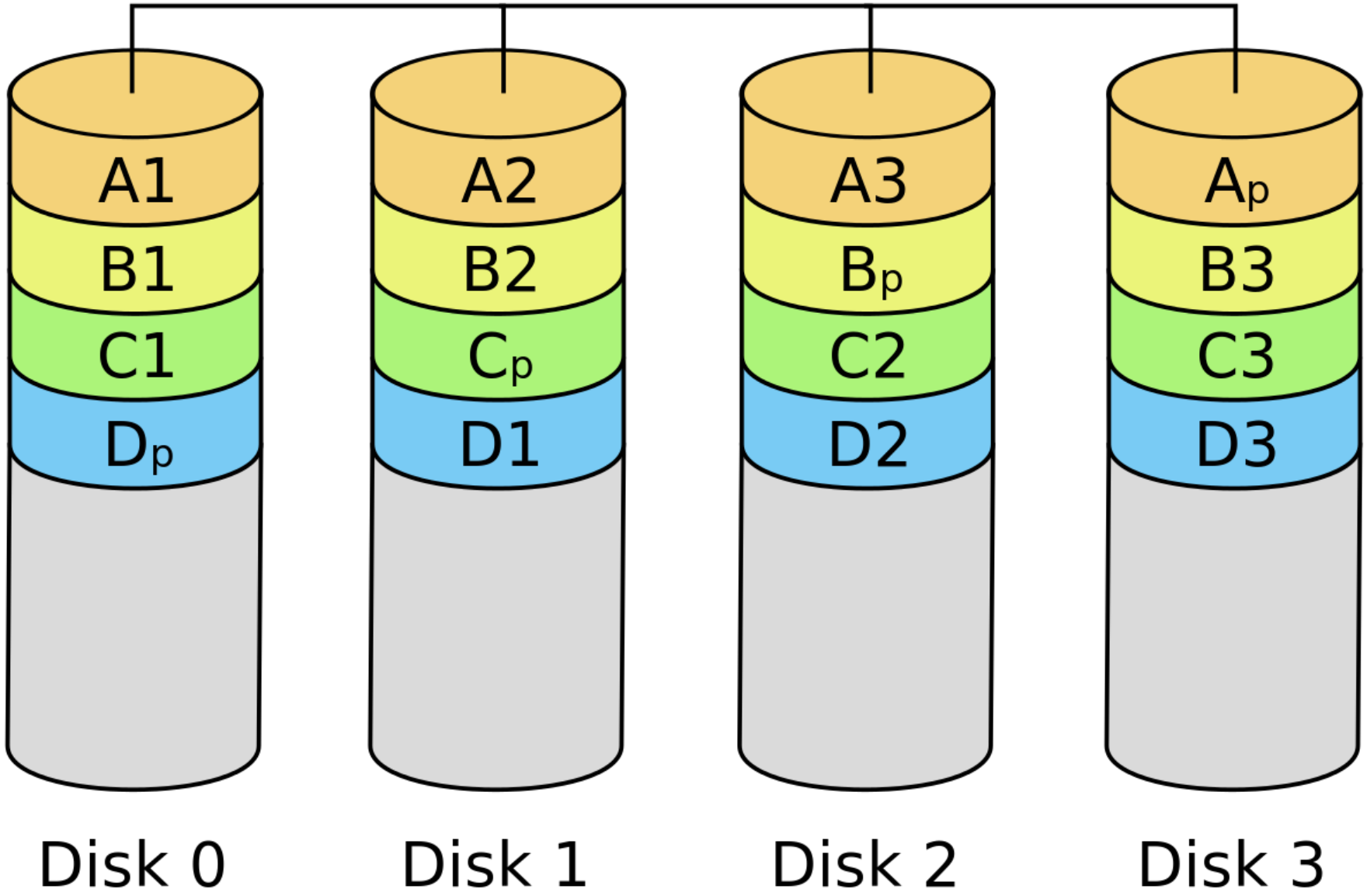

**Figure 17.14**
Some popular levels of RAID. (a) RAID level 1: Mirroring of data on two disks. (b) RAID level 5: Striping of data with distributed parity across four disks.



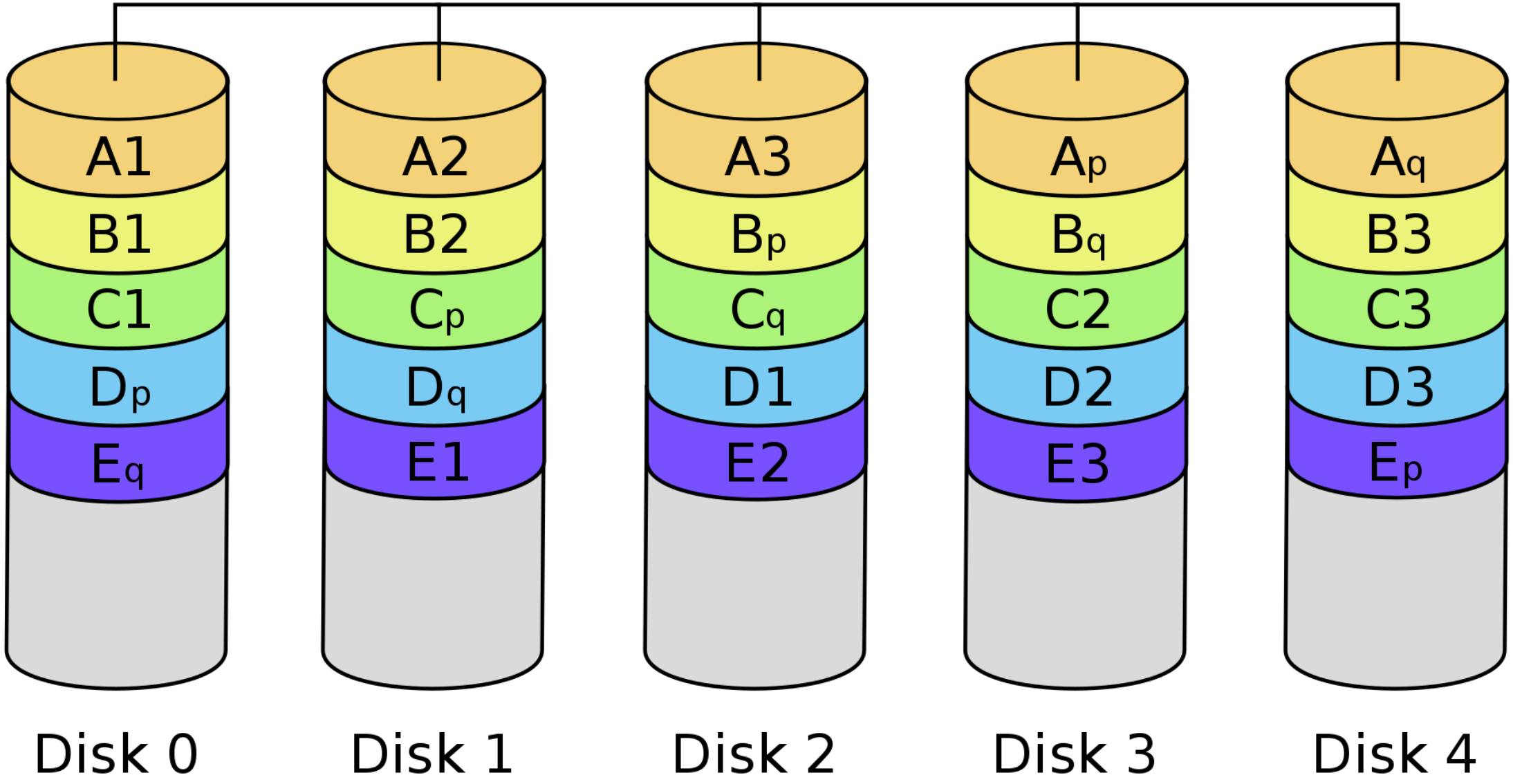
# RAID 4



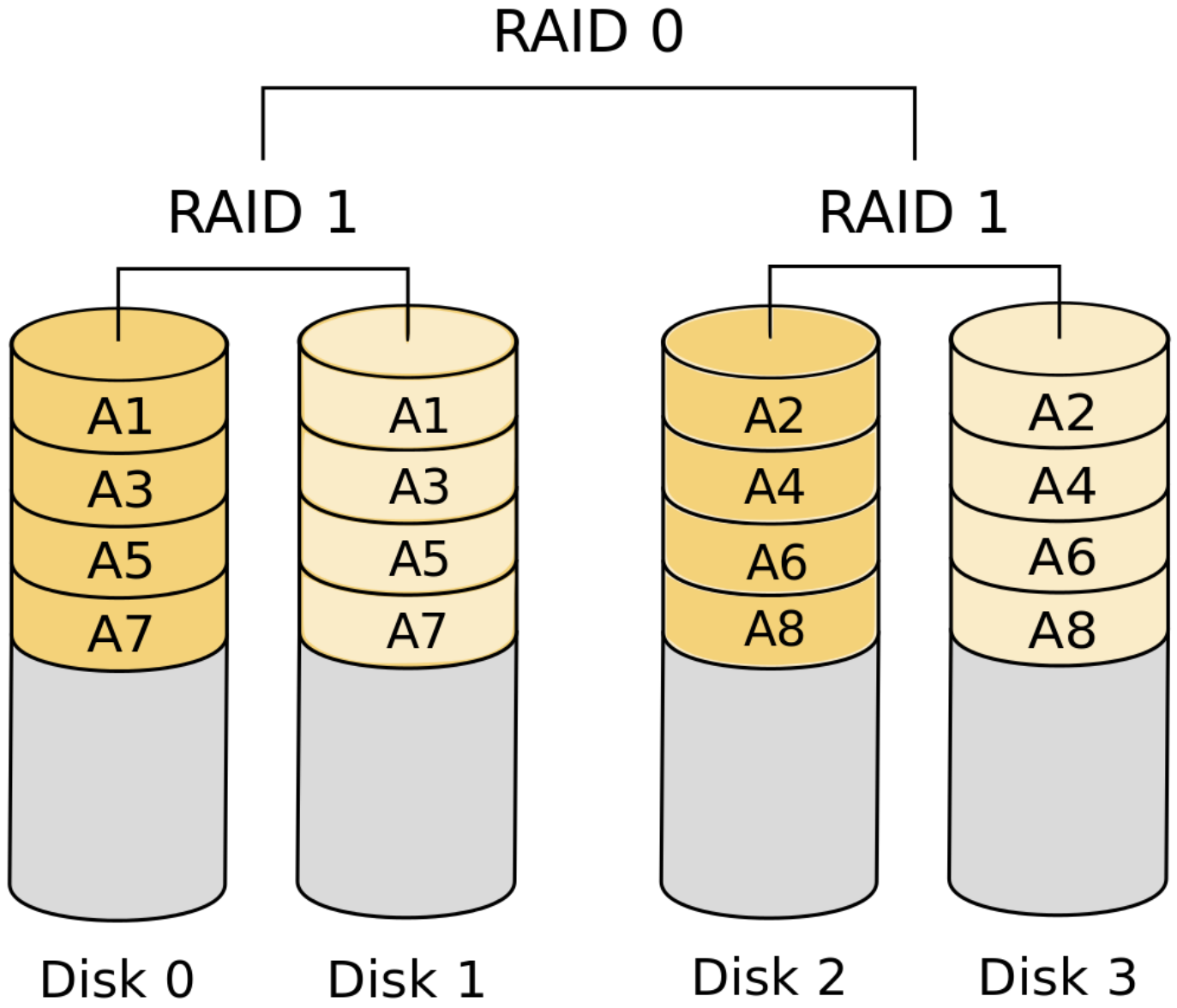
# RAID 5



# RAID 6



# RAID 1+0




By Wheart, based on image File:RAID 0.svg by Cburnett

# Index Structures - Overview

- Introduction
- Types of indexes
- Implementation of indexes
- CREATE INDEX statement

# Index Structures

- Physical order of records affects efficiency of individual queries requiring file traversal
- Physical records can only be in one order at a time
- Need an efficient way to access records in different orders without expensive operations like physical sorting

# Index Structures (Cont.)

- Index (access path): provides faster access to data
- Can be compared to the index of a book
- Index maps values of indexing field/attribute to corresponding records
- Index entries ordered by indexing field value – thus binary search can be used
- Can have several indexes per table
- Dense and sparse indexes
  - D**ense index:** an entry for every search key value/record
  - S**parse (or nondense) index:** index entries for only some of the search values

# Classifying Indexes

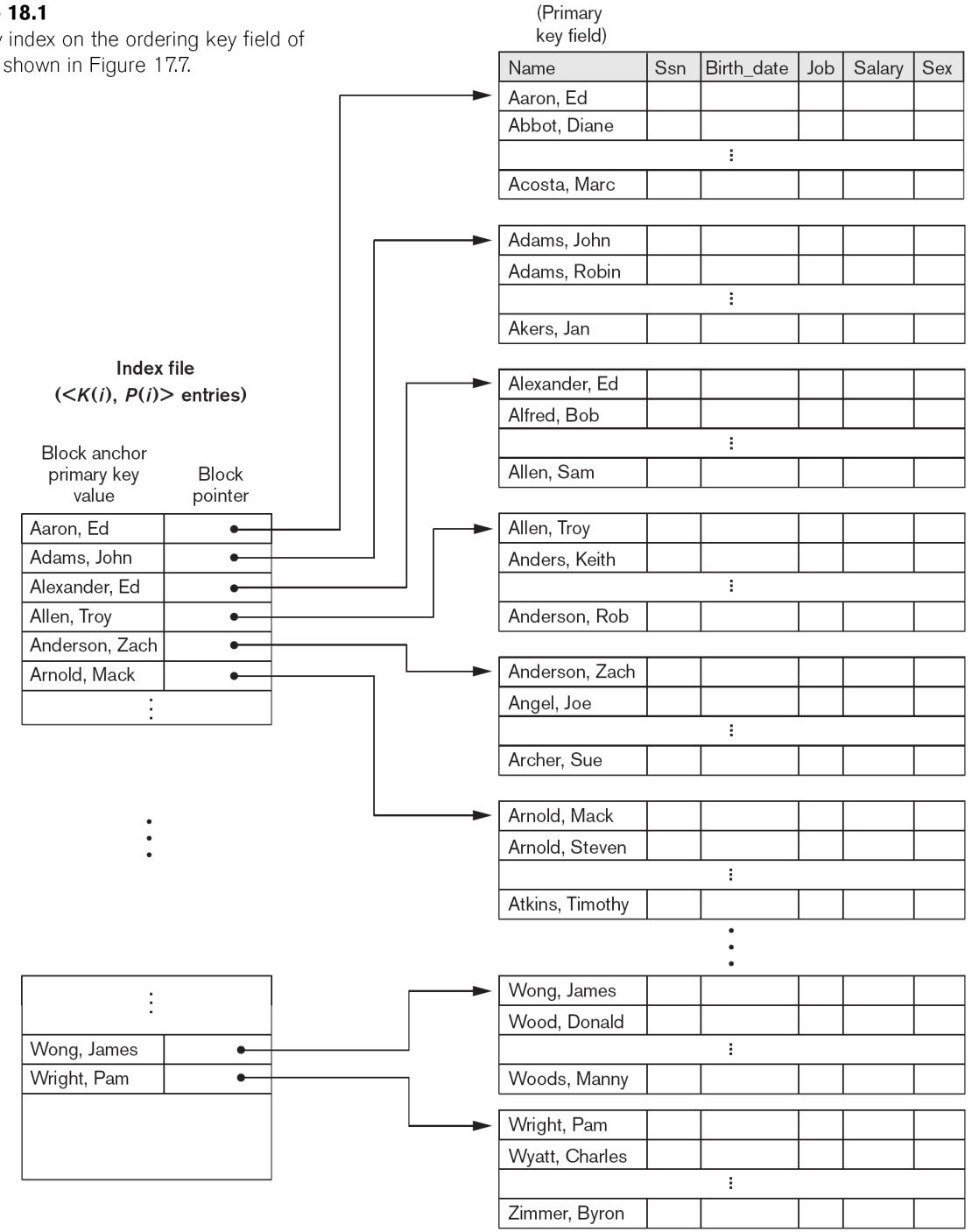
- Single-level indexes
  - Primary index:  specified on the ordering key field of an ordered file
  - Clustering index: specified on the ordering non-key field of an ordered file
  - Secondary  index: can be specified on any non-ordering field of a file
  - A file can have either a primary or a clustering index (not both!)
- Multi-level indexes
- Dynamic Multilevel Indexes Using B-Trees and B+-Trees
- Indexes on Multiple Keys

# Primary Index

- Defined on an ordered data file
- The data file is ordered on a **key field**
- Includes one index entry *for each block* in the data file; the index entry has the key field value for the *first record* in the block, which is called the *block anchor*
- A similar scheme can use the *last record* in a block.
- A primary index is a nondense (sparse) index, since it includes an entry for each disk block of the data file and the keys of its anchor record rather than for every search value.

# Primary Index

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



# Clustering Index

- Defined on an ordered data file
- The data file is ordered on a *non-key field*
- Includes one 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.
- Another example of *nondense* index
- Insertion and Deletion is relatively straightforward with a clustering index.

# Clustering Index

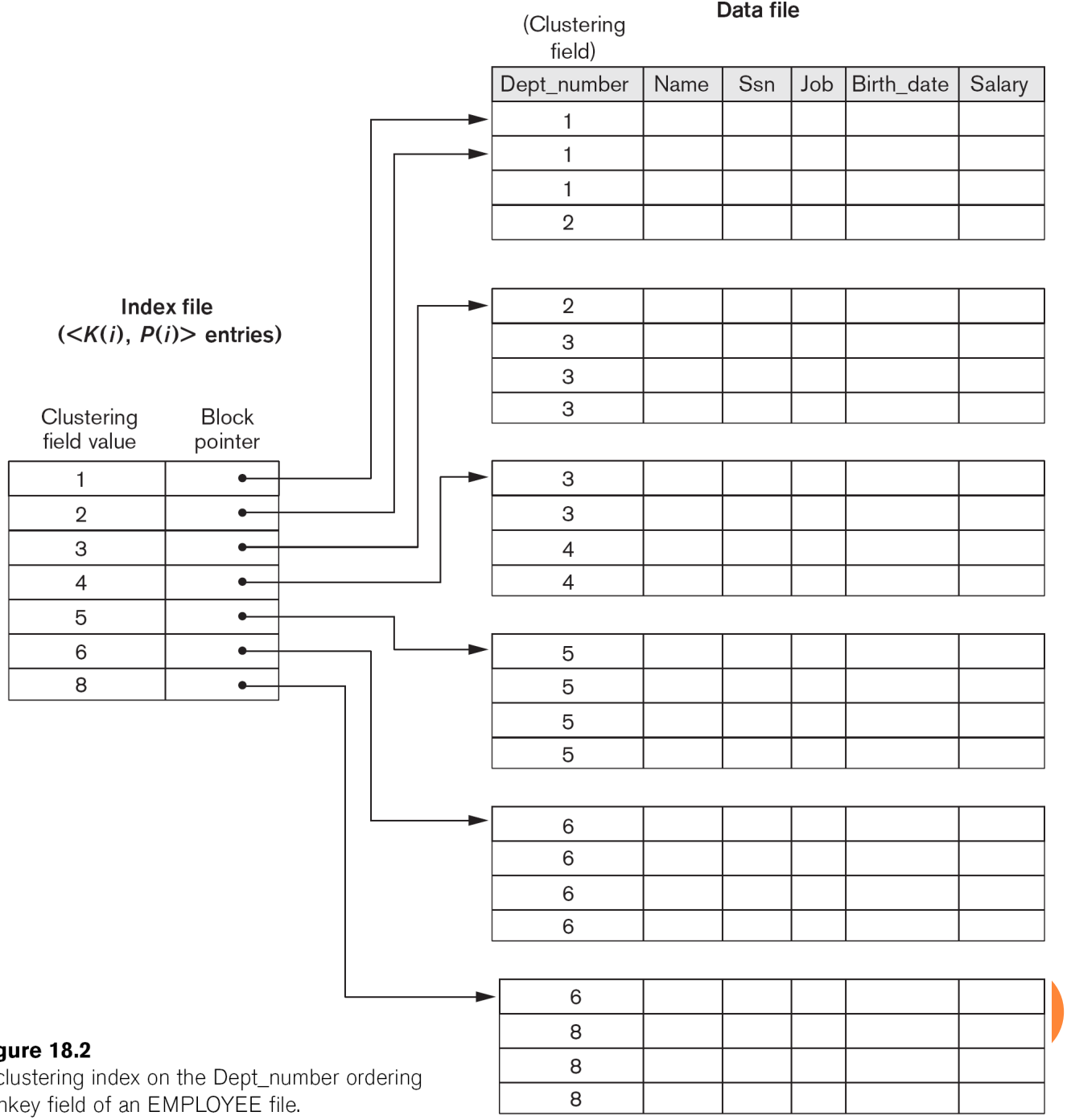



**Figure 18.2**
A clustering index on the Dept_number ordering nonkey field of an EMPLOYEE file.

# Another Clustering Index Example

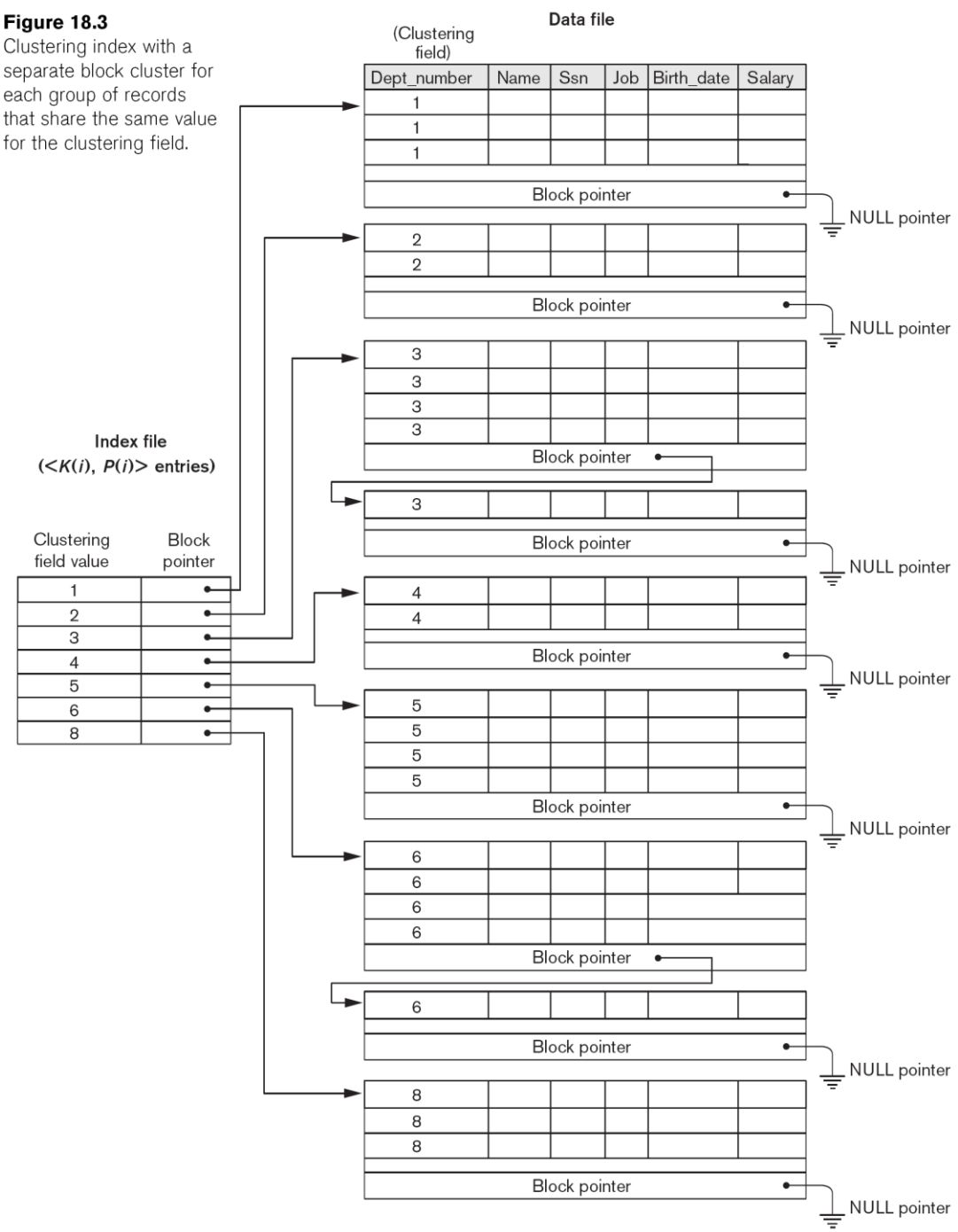


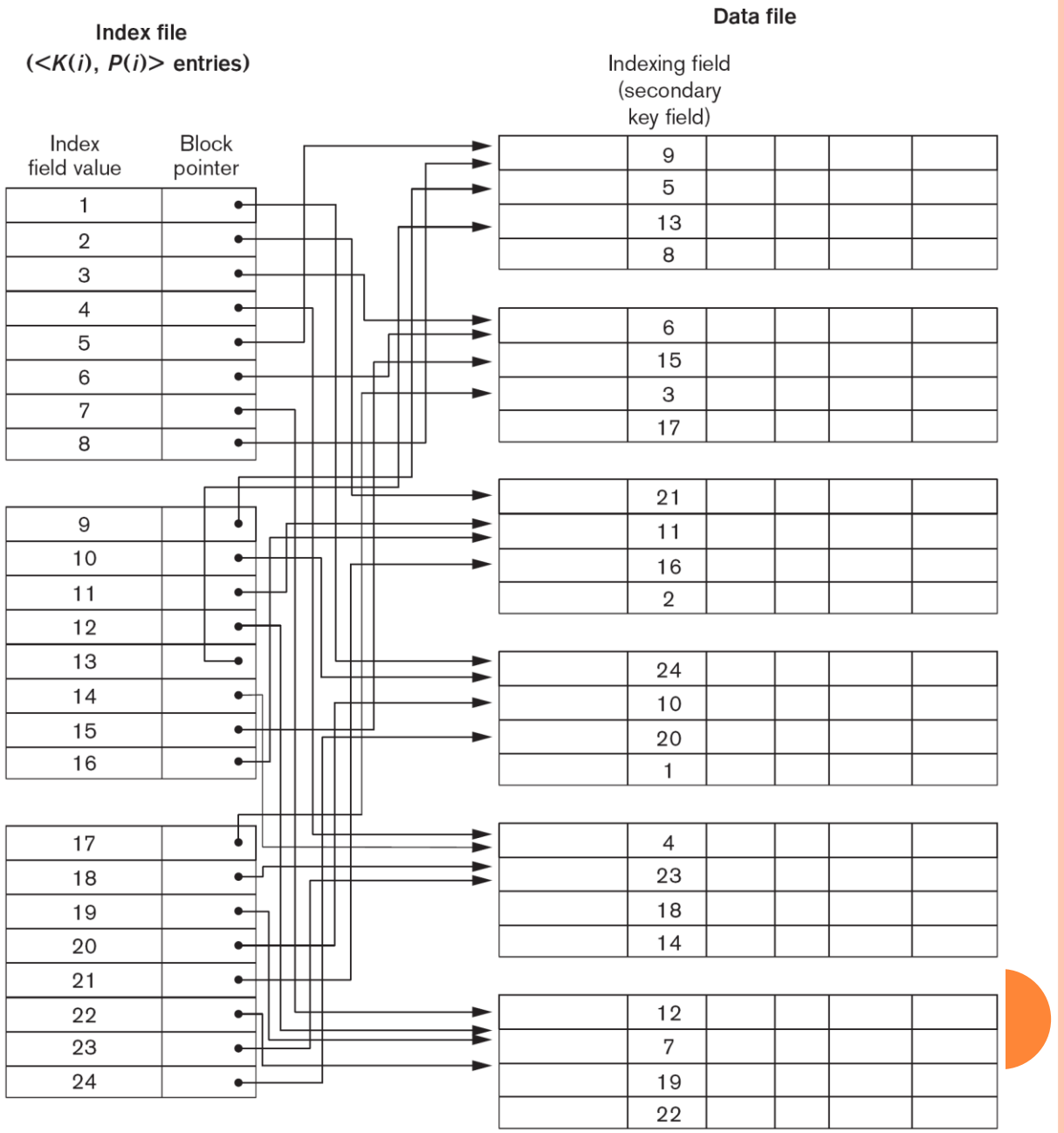

**Figure 18.3**
Clustering index with a separate block cluster for each group of records that share the same value for the clustering field.

# Secondary Indexes

- Provides secondary means of accessing a file for which some primary access already exists
- Secondary index may be on a field which is a candidate key and has a unique value in every record, or a non-key with duplicate values
- The index is an ordered file with two fields:
  - The first field is of the same data type as some **non-ordering field** of the data file that is an indexing field.
  - The second field is either a **block** pointer or a record pointer.
- Includes one entry *for each record* in the data file; hence, it is a *dense index*
- There can be *many* secondary indexes (and hence, indexing fields) for the same file.

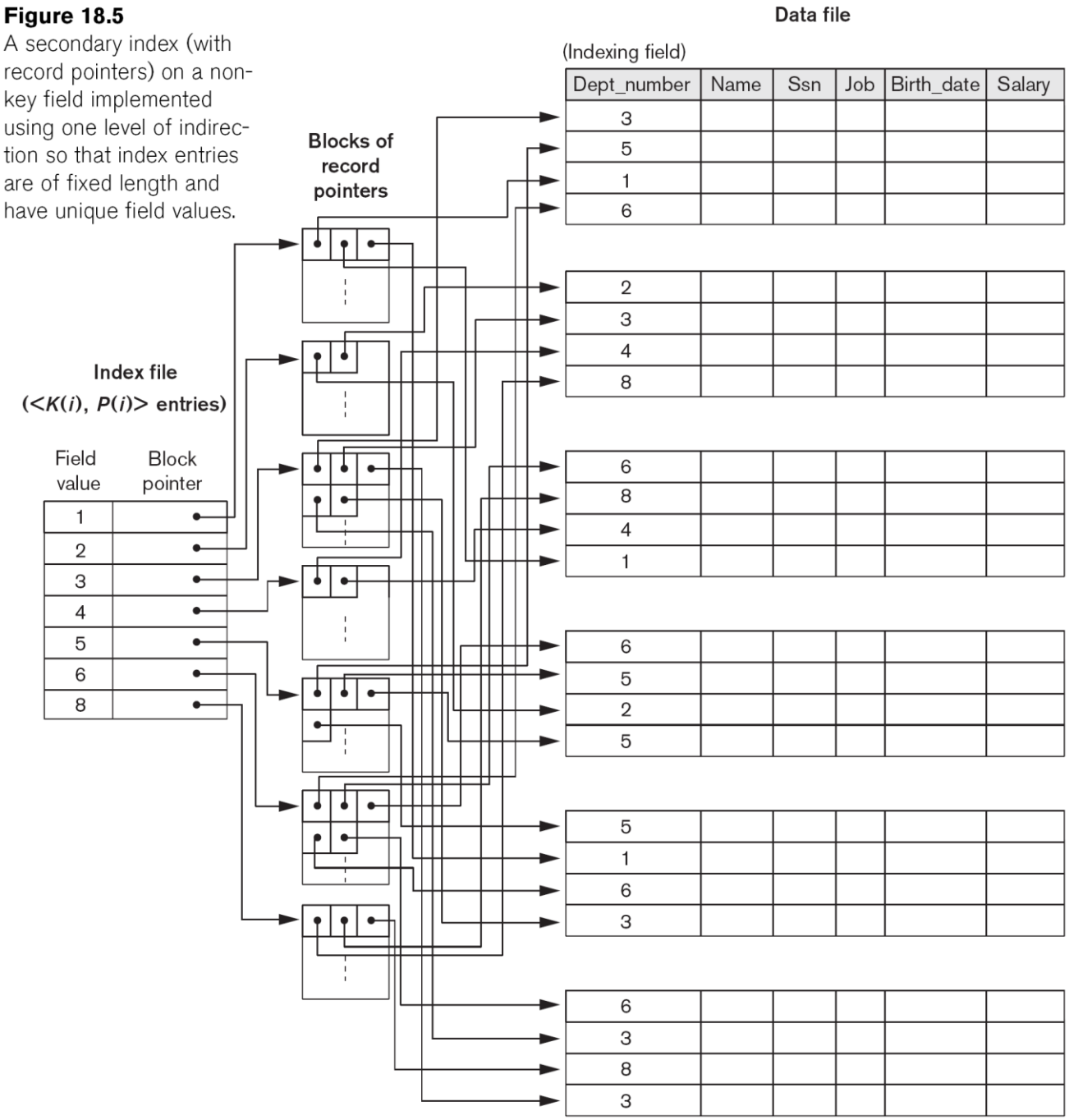
# Example of a Dense Secondary Index

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



# Example of a Secondary Index

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



# Example 1

Compare the speed of retrieval for a file with r=30,000 records of fixed length R=100B, given block size of 1024B and V=9B, P=6B, in case of

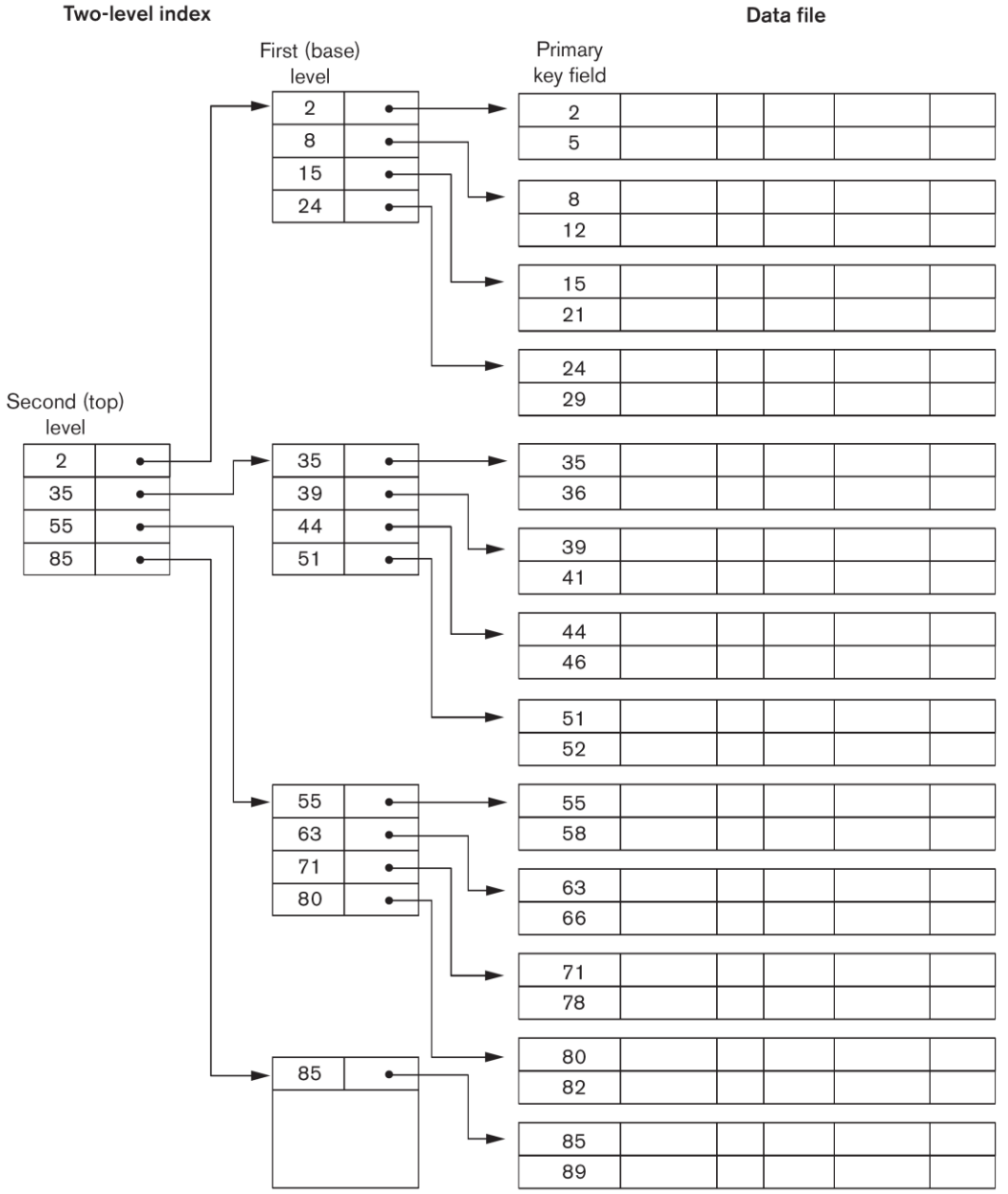
- an ordered file
- a file with a primary index
- a file with a secondary index

# Multi-Level Indexes

- Because a single-level index is an ordered file, we can create a 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 a 3rd, 4th, ..., top level until all entries fit in one disk block
- Can be created for any type of first-level index
- *Search tree*
- *Fan-out (fo)*
- Insertion and deletion of new entries is a severe problem because every level of the index is an *ordered file*.

# A Two-Level Primary Index

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



# Example 2

- How many levels of indexing would be needed for the file from example 1?

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

- Most multi-level indexes use B-tree or B+-tree data structures because of the insertion and deletion problem
- In B-Tree and B+-Tree data structures, each node corresponds to a disk block
- Generalizations of binary search trees
- Each node is kept between 50% and 100% full

# B-TREES

- Node structure in a B-tree of order *p*:

  $<P_1, <K_1, Pr_1>, P_2, <K_2, Pr_2>, P_3, \ldots, <K_{q-1}, Pr_{q-1}>, P_q>$

  $P_i$ is a tree pointer

  $Pr_i$ is a data pointer

  $K_i$ is a key value
- Each node has at most *p* tree pointers
- Keys are ordered: $K_1 < K_2 < \ldots < K_{q-1}$
- All leaf nodes are at the same level

# B-tree Structures

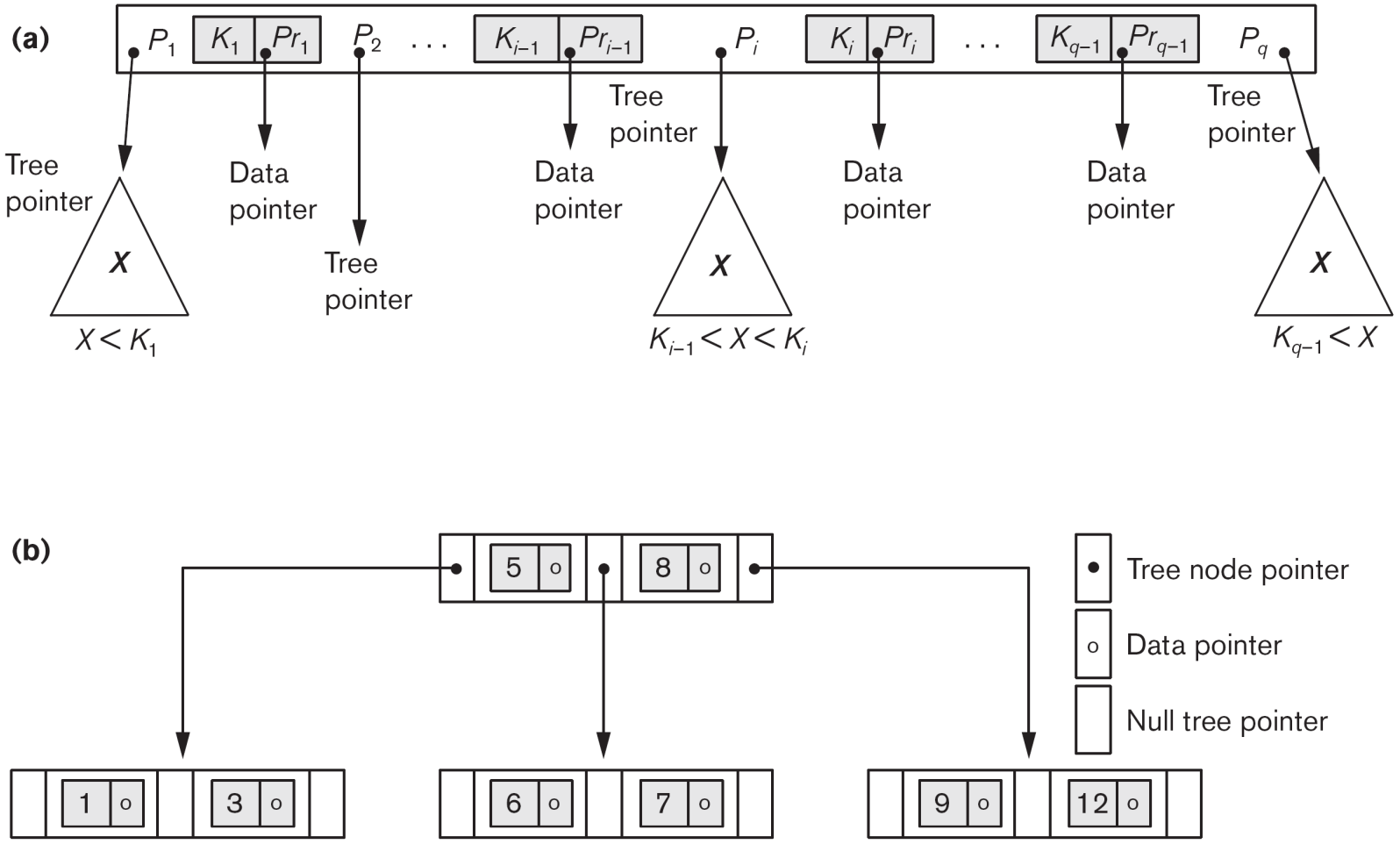



**Figure 18.10**
B-tree structures. (a) A node in a B-tree with $q - 1$ search values. (b) A B-tree of order $p = 3$.The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.

# Operations with B-trees

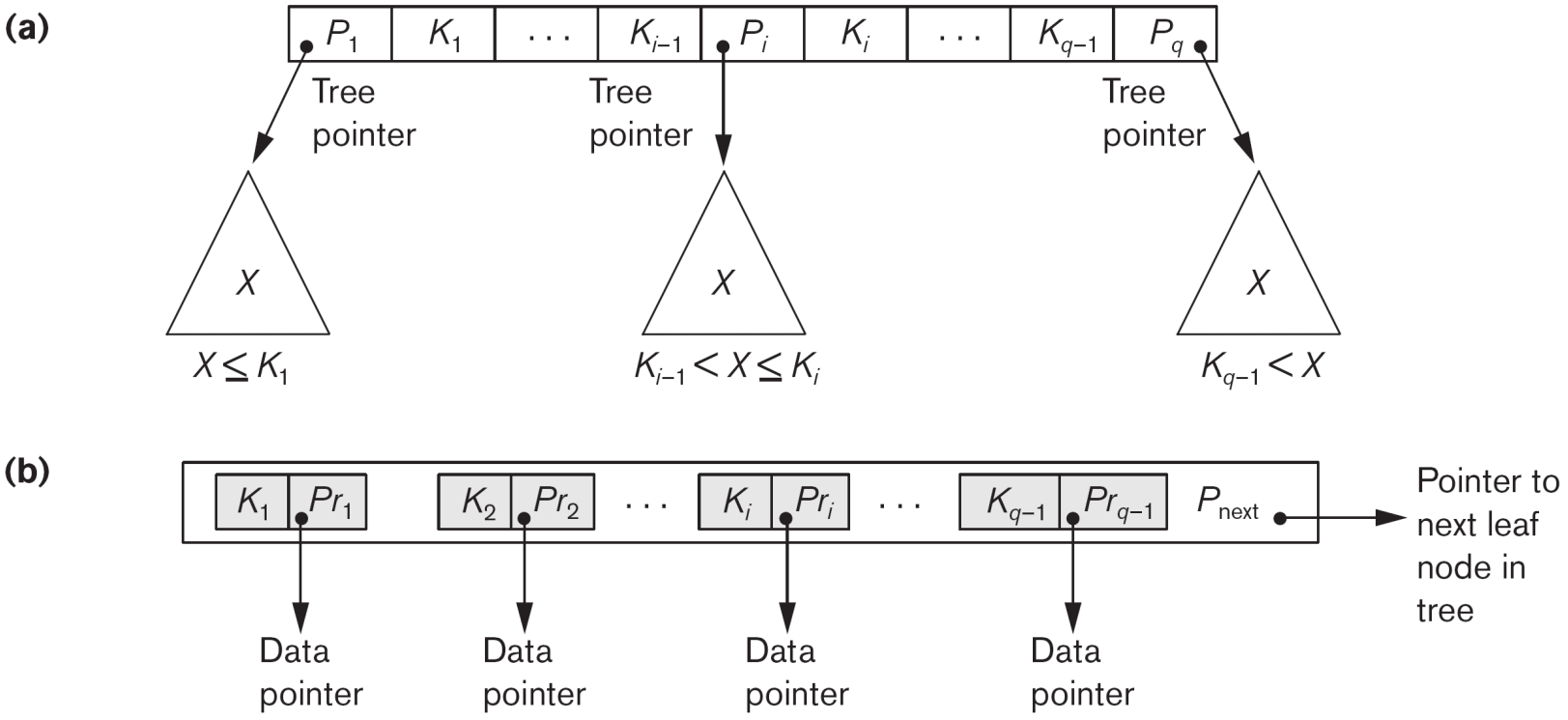
- Search
- Insertion
  - Very efficient
  - If a node fills up, it is split into two nodes at the next lower level
  - Splitting may propagate to other tree levels
- Deletion
  - quite efficient if a node does not become less than half full
  - If node shrinks below 50%, it is merged. Tree might need to be re-balanced.

# Difference between B-tree and B+-tree

- In a B-tree, pointers to data records exist at all levels
- In a B+-tree, all pointers to data records exists at the leaf-level nodes
- A B+-tree can have less levels (or higher capacity of search values) than the corresponding B-tree
- Internal nodes $<P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q>$
- Each internal node has at most p tree pointers
- Leaf nodes $<K_1, Pr_1>, <K_2, Pr_2>, \ldots, <K_{q-1}, Pr_{q-1}>, P_{next}>$

# The Nodes of a B+-tree

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

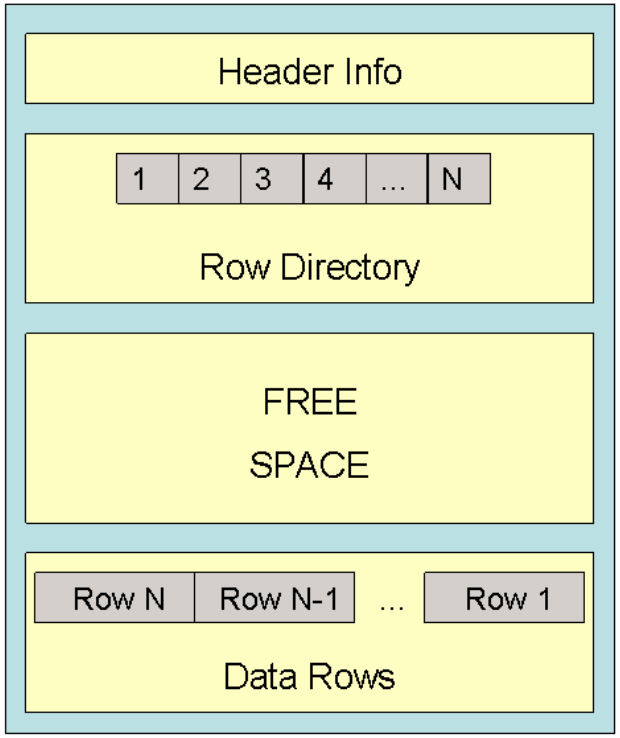


# Example 3

- Calculate the order p of a B+ tree if B=512, V=9B, Pr=7B and P=6B.

# Physical level in Oracle

- Block size 4-64kB (typically 4 or 8kB)
- If different from the OS block size, should be a multiple of it

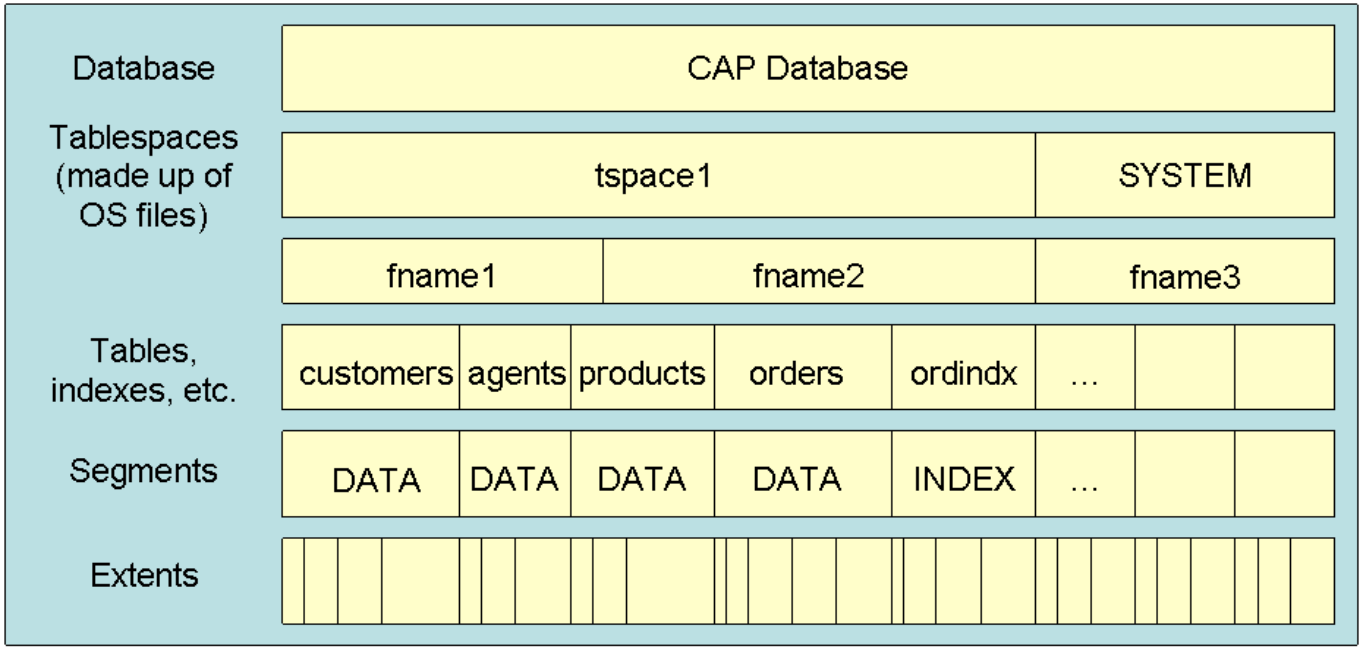


# Physical and Logical Units

- The smallest physical unit is page.
- Several *contiguous* blocks build an extent (fixed or dynamic size.)
- Several extents build a segment (logical unit!)
- Several segments build a *tablespace*.

  An Oracle database consists of at least two tablespaces, which collectively store all of the database's data (SYSTEM, SYSAUX).
- Each tablespace in an Oracle database consists of one or more files called datafiles, which are physical structures that conform to the operating system in which the Oracle database is running.

# Oracle: Physical and Logical Units



# Creation of Tablespaces in Oracle

```
CREATE TABLESPACE tspace2
   DATAFILE fname3 SIZE 200M,
    DATAFILE fname4 SIZE 400M;
```

*A tablespace* contains different database objects. The CREATE object statement is used to assign an object to the tablespace.

# Specifying physical level

- CREATE TABLE table_name (attribute_list),
  [TABLESPACE tblspc]
  [ORGANIZATION {INDEX param | HEAP param}
  | [CLUSTER] cluster (column [, column ]...) ]
  [physical_attributes];

- CREATE INDEX index_name
  ON tab_name(col_list)
  [physical_attributes]);

# Physical attributes

- PCTFREE int
  - Percentage of free space in a block (0-99, default 10)
- PCTUSED int
  - Minimum % of used space in a block (default 40)
- INITRANS int
  - Initial number of concurrent transactions allocated within a block (1-255, default 1)

```
CREATE TABLE my_table (column_def),
   TABLESPACE tspace2
    PCTFREE 20
   PCTUSED 60
   ORGANIZATION HEAP
```