# Chapter – 3

# File Organization and Indexing

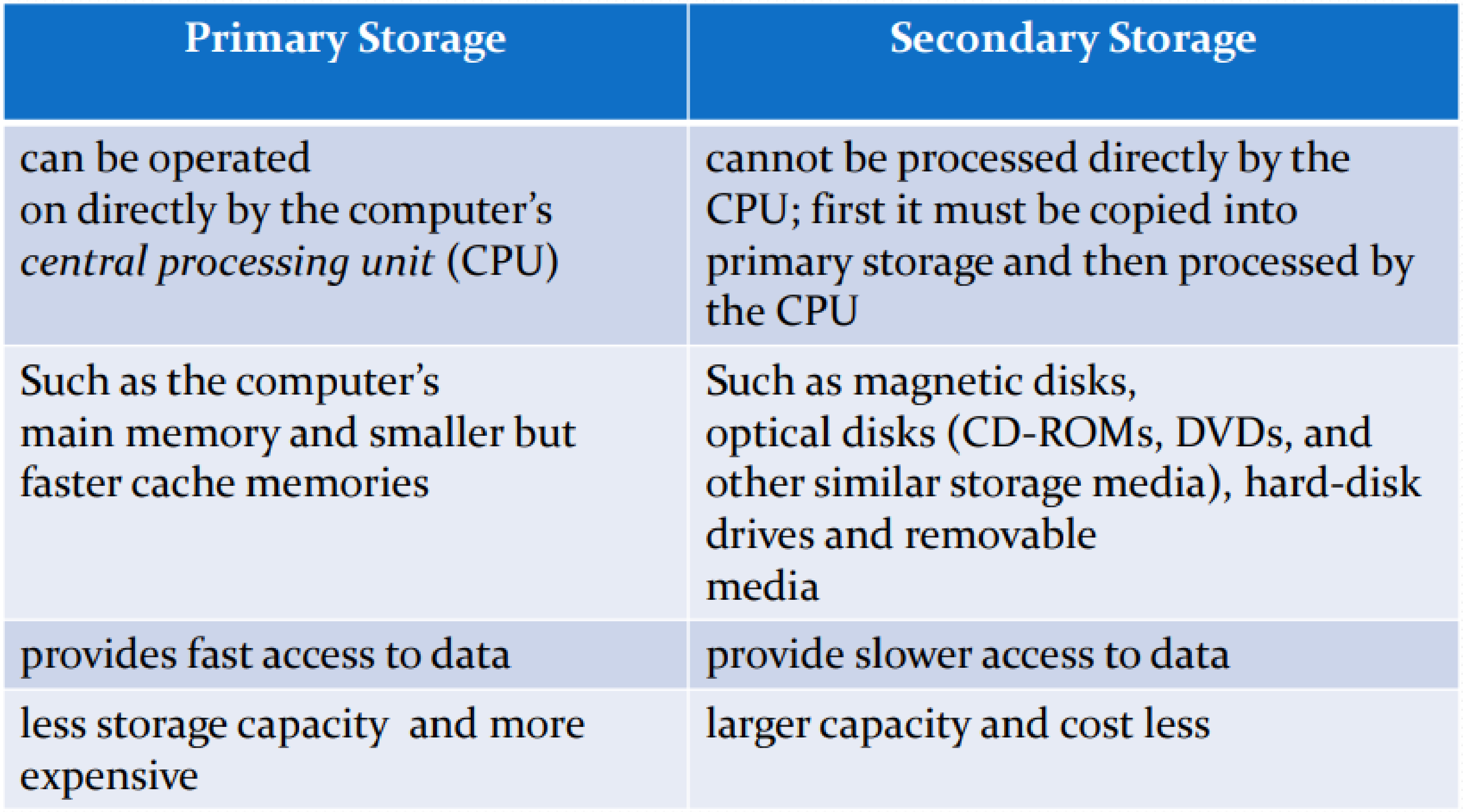
## Outline

- Disk Storage Devices
- Files of Records
- Operations on Files
- Unordered Files
- Ordered Files
- Hashed Files
  - Dynamic, Extendible and linear Hashing Techniques
- RAID Technology

# Introduction

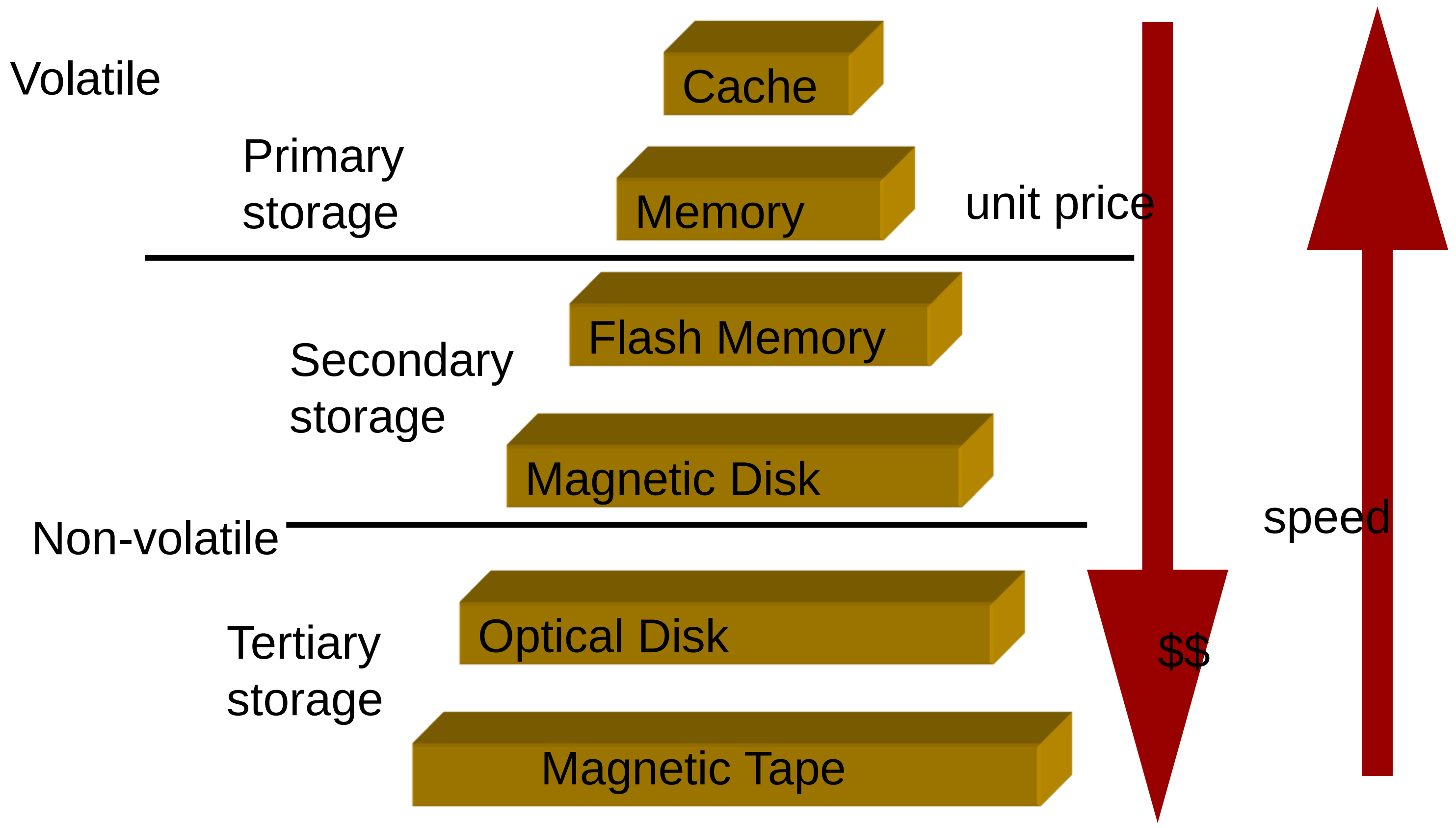
- Databases are stored physically as files of records, which are typically stored on magnetic disks
- The collection of data that makes up a computerized database must be stored physically on some computer **storage medium**
- DBMS software can then retrieve, update, and process this data as needed
- Two main categories of storage medium
  - **Primary storage**
  - **Secondary and tertiary storage**

# Primary vs Secondary Storage



| Primary Storage | Secondary Storage |
|---|---|
| can be operated on directly by the computer's *central processing unit* (CPU) | cannot be processed directly by the CPU; first it must be copied into primary storage and then processed by the CPU |
| Such as the computer's main memory and smaller but faster cache memories | Such as magnetic disks, optical disks (CD-ROMs, DVDs, and other similar storage media), hard-disk drives and removable media |
| provides fast access to data | provide slower access to data |
| less storage capacity  and more expensive | larger capacity and cost less |

# Storage Hierarchy



## Storage Hierarchy

- At the primary storage level
- **Cache memory**
  - Cache memory is typically used by the CPU to speed up execution of program instructions using techniques such as prefetching and pipelining
  - It stores the segments of program that are frequently accessed by the processor
- **Main memory**
  - Provides the main work area for the CPU for keeping program instructions and data
  - It is less expensive than cache memory and therefore larger in size
  - The drawback is its **volatility** and **lower speed** compared with cache memory

# Storage Hierarchy

- At the secondary and tertiary storage level
- The hierarchy includes magnetic disks, as well as **mass storage** in the form of **CD-ROM** (Compact Disk–Read-Only Memory) and **DVD** (Digital Video Disk or Digital Versatile Disk) devices, and finally **tapes** at the least expensive end of the hierarchy
- The **storage capacity** is measured in **kilobytes** (Kbyte or 1000 bytes), **megabytes** (MB or 1 million bytes), **gigabytes** (GB or 1 billion bytes), and even **terabytes** (1000 GB)
- The word petabyte is now becoming relevant in the context of very large repositories of data
- **Magnetic tapes** are used for archiving and backup storage of data

# Storage of Databases

- DBMS stores information on ('hard') disks
- This has major implications for DBMS design!
  – **READ**: transfer data from disk to main memory (RAM)
  – **WRITE**: transfer data from RAM to disk
  – Both are **high-cost operations**, relative to in memory operations, so must be planned carefully!
- Why not store everything in main memory?
  - Costs too much
  - Main memory is volatile. We want data to be saved between runs
- Typical storage hierarchy
  - Main memory (RAM) for currently used data
  - Disk for the main database
  - Tapes for archiving older versions of the data

## Secondary Storage Devices

- Magnetic disks are used for **storing large amounts of data**
- The most basic unit of data on the disk is a single **bit** of information. To code information, bits are grouped into **bytes**
- **Capacity** of a disk is the **number of bytes it can store**, which is usually very large

- Preferred secondary storage device for high storage capacity and low cost.
- Data stored as magnetized areas on magnetic 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 to 50 Kbytes or more

## Disk Storage Devices

- A track is divided into smaller **blocks** or **sectors**
  - because it usually contains a large amount of information
- 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 as a sector

- A track is divided into **blocks**
  - The **block size B is fixed for each system**
    - Typical block sizes range from **B=512 bytes to B=4096 bytes**
  - Whole blocks are transferred between disk and main memory for processing

# Disk Storage Devices

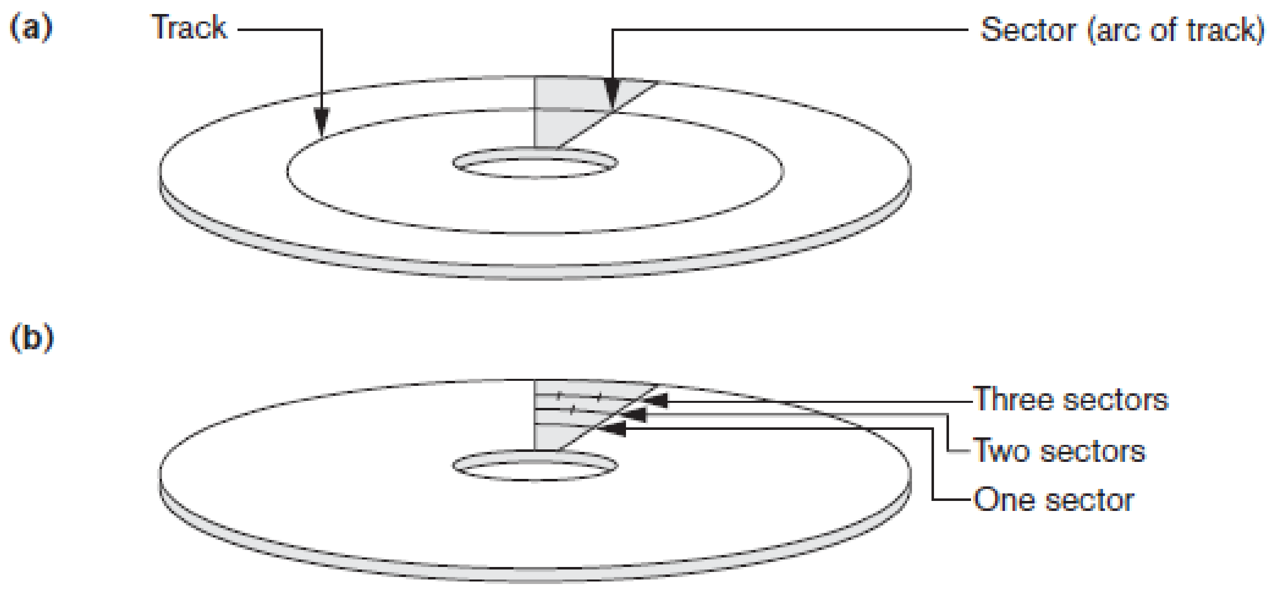


**Figure**
Different sector organizations on disk. (a) Sectors subtending a fixed angle. (b) Sectors maintaining a uniform recording density.

# Disk Storage 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)**

- Time to access (read/write) a disk block
  - Seek time (moving arms to position disk head on track)
  - Rotational delay or latency  (waiting for block to rotate under head)
  - Transfer time (actually moving data to/from disk surface)
- Locating data on a disk is a major bottleneck – need efficient techniques to do this

# Disk Storage Devices

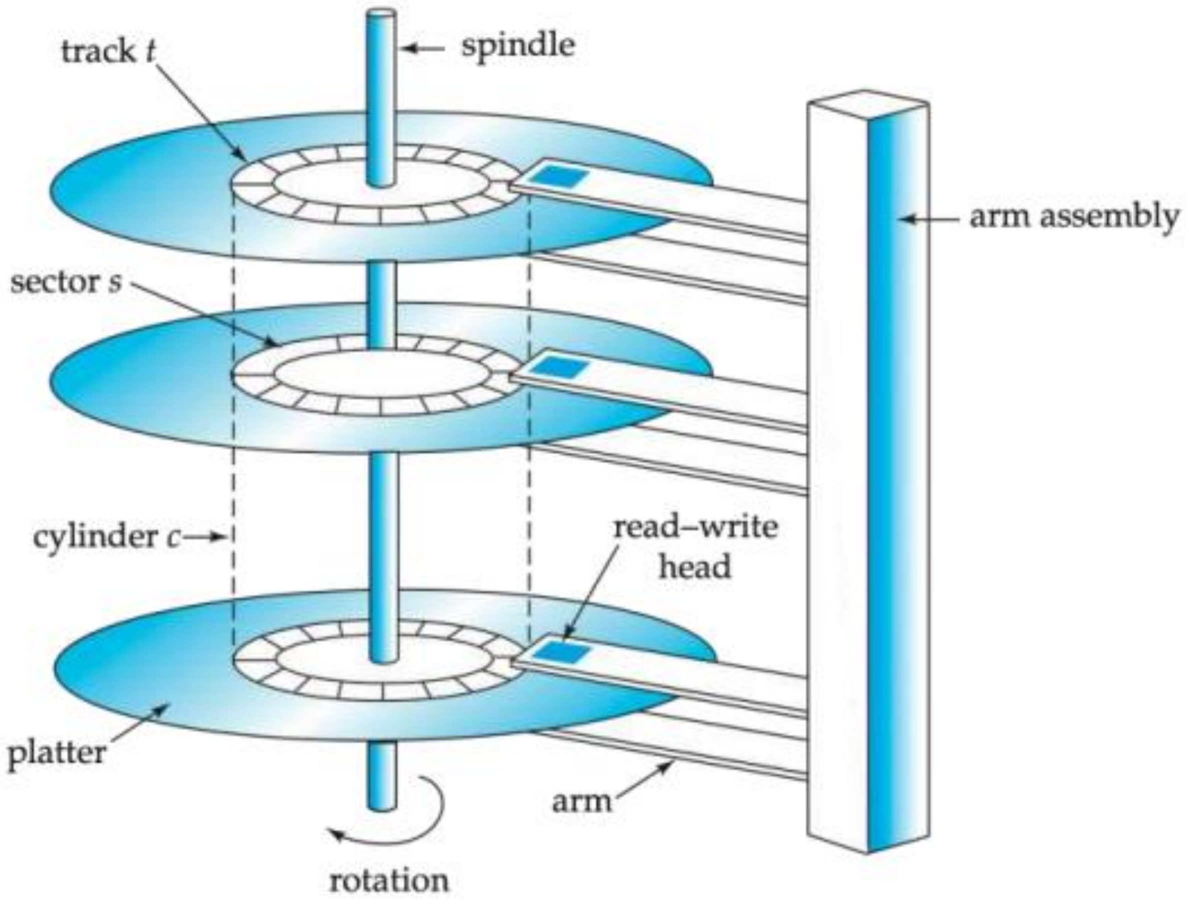


**Figure**
(a) A single-sided disk with read/write hardware
(b) A disk pack with read/write hardware

## Components of a Disk

- The **platters** spin (say, 90rps)
- The **arm assembly** is moved in or out to position a head on a desired track
- **Read-write** head
  - Positioned very close to the **platter surface** (almost touching it)
  - Reads or writes magnetically encoded information
  - Only one head reads/writes at any one time
- Surface of **platter** divided into circular **tracks**

## Physical Characteristics of Disks

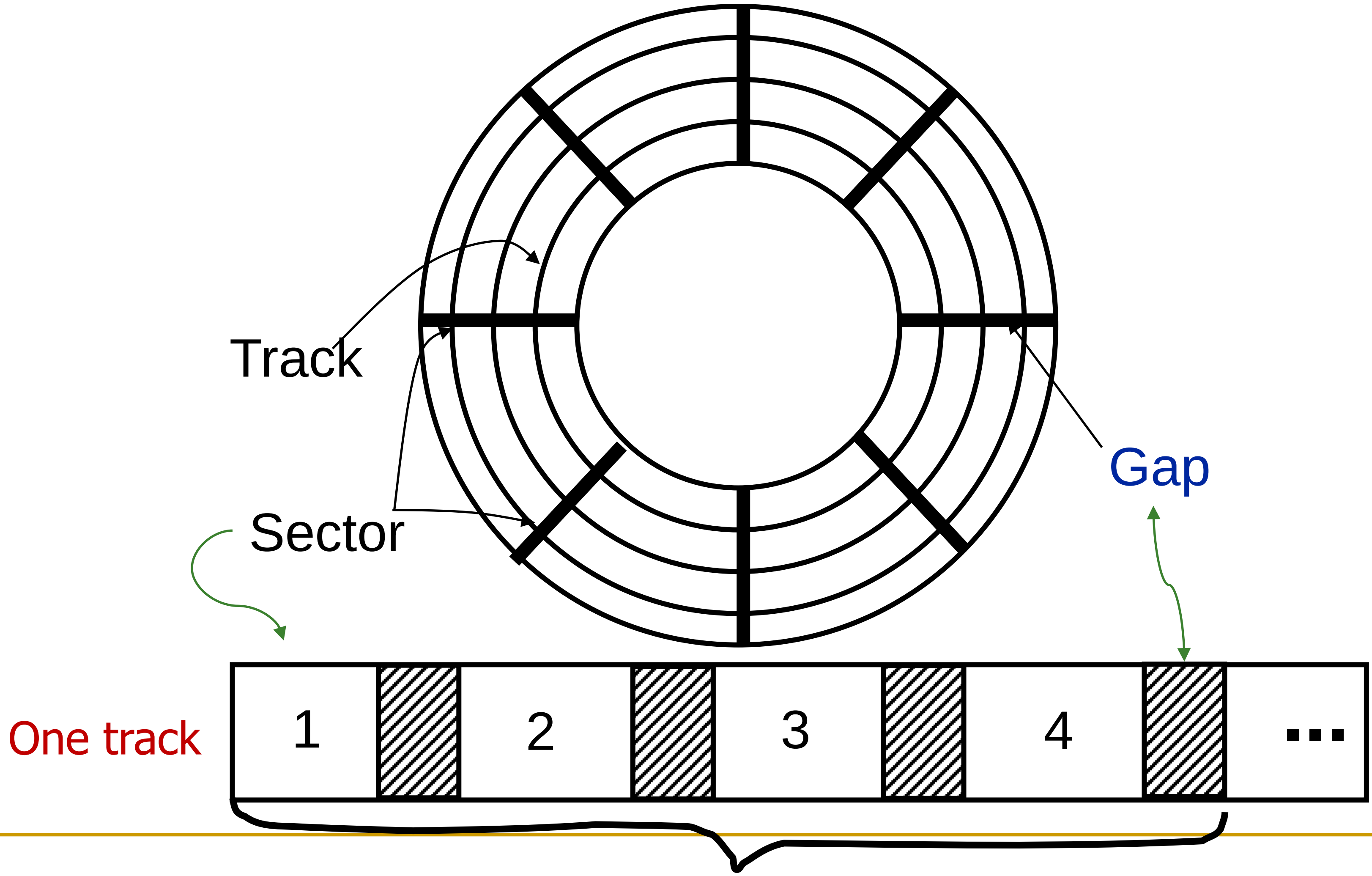
- **Track**
  - an information storage circle on the surface of a disk.
  - Over 16,000 tracks per platter
  - each track can store between 4KB and 50KB of data.
  - Each track is divided into **sectors**.
  - Tracks under heads make a cylinder (imaginary!)
- **Cylinder**
  - the tracks with the same diameter on all surfaces of a disk pack.
  - Cylinder *i* consists of *i*-th track of all the platters
- **Sector**
  - a part of a track with fixed size
  - separated by fixed-size interblock **gap**s
  - Typical sectors per track
    - 200 (on inner tracks) to 400 (on outer tracks)

## Pages and Blocks

- Data files decomposed into **pages** (**blocks**)
  - fixed size piece of contiguous information in the file
  - sizes range from 512 bytes to several kilobytes
- Block is the smallest unit for transferring data between the main memory and the disk
- Address of a page (block)

  (cylinder#, track# (within cylinder), sector# (within track)

# Pages and Blocks



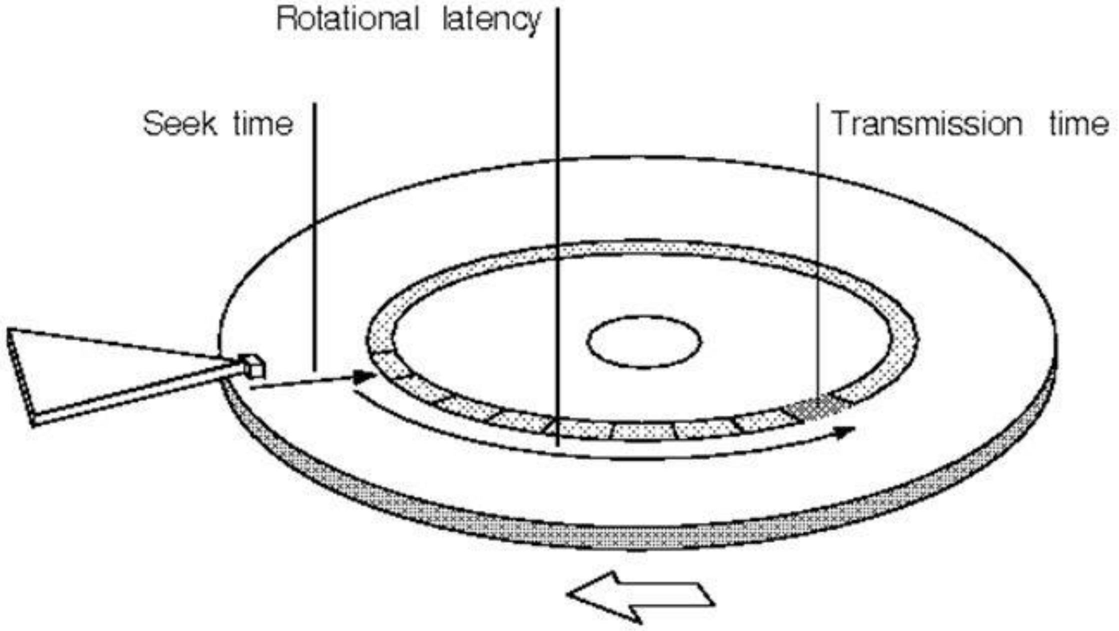
# Page I/O

- Page I/O --- one page I/O is the cost (or time needed) to transfer one page of data between the memory and the disk.
- The cost of a (random) page I/O =
  - seek time + rotational delay + block transfer time

- Seek time
  - time needed to position read/write head on correct track.
- Rotational delay (latency)
  - time needed to rotate the beginning of page under read/write head
- Block transfer time
  - time needed to transfer data in the page/block

## Data access time in Hard Disks

seek time + rotational latency + transmission time



## Magnetic Tape Storage Devices

- Disks are **random access** secondary storage devices because an **arbitrary disk block may be accessed *at random*** once we specify its address
- Magnetic tapes are **sequential access** devices; to access the *n*th block on tape, first we must scan the preceding *n*–1 blocks
- Data is stored on reels of high-capacity magnetic tape, somewhat similar to audiotapes or videotapes
- A read/write head is used to read or write data on tape.
- Data records on tape are also stored in blocks—although the blocks may be substantially larger than those for disks
- Tapes serve a very important function-**backing up** the database One reason for backup is to keep copies of disk files in case the data is lost due to a disk crash

## Buffering of Blocks

- When s**everal blocks need to be transferred from disk to main memory** and all the block addresses are known, several buffers can be reserved in main memory to speed up the transfer

- While one buffer is being read or written, the CPU can process data in the other buffer because an independent **disk I/O processor** (controller) exists that, once started, can proceed to **transfer a data block between memory and disk independent of and in parallel to CPU processing**

- Double buffering can be used to speed up the transfer of contiguous disk blocks

# Buffering of Blocks

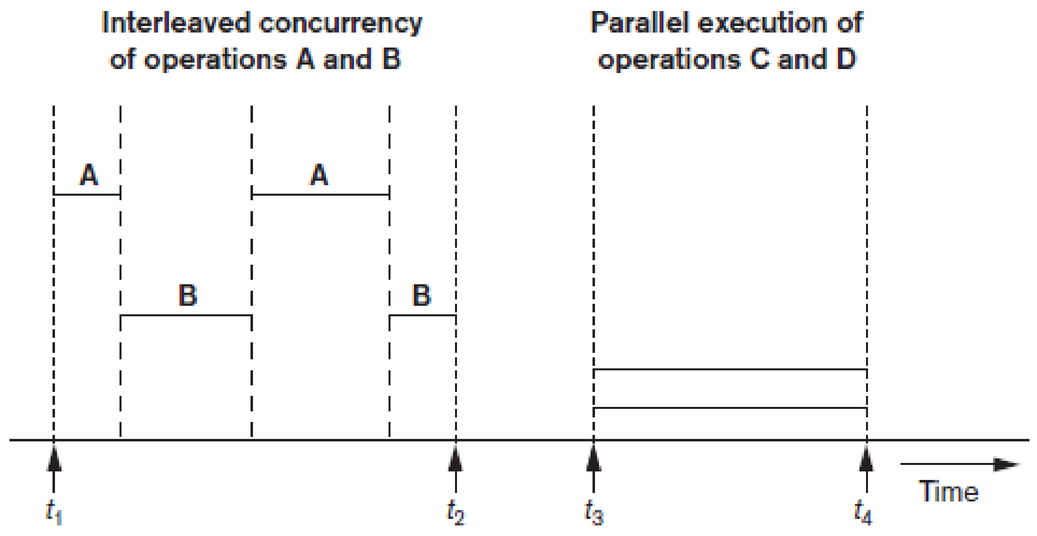

Interleaved concurrency of operations A and B
Parallel execution of operations C and D
A
A
B
B
$t_1$
$t_2$
$t_3$
$t_4$
Time

**Figure 17.3**
Interleaved concurrency versus parallel execution.

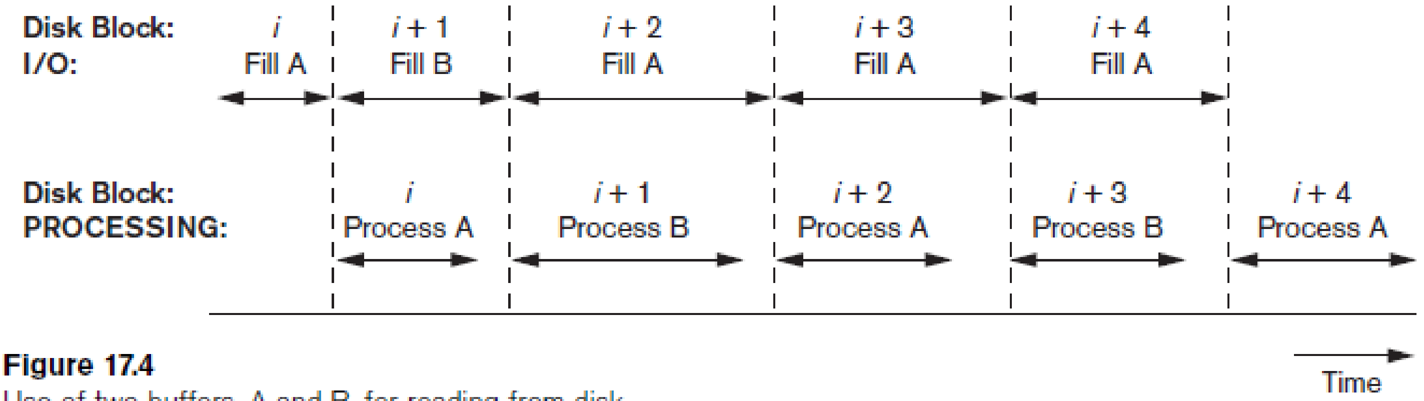


**Figure 17.4**
Use of two buffers, A and B, for reading from disk.

# Placing File Records on Disk

- **Records**
- 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
  - Records usually describe entities and their attributes
  - For example
    - An **EMPLOYEE record** represents an employee entity, and each field value in the record specifies some attribute of that employee, such as Name, Birth_date, Salary, or Supervisor

## Placing File Records on Disk

- **Record Types**
- 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
- For example, an **EMPLOYEE record type** may be defined—using the C programming language notation-as the following structure:

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

# Placing File Records on Disk

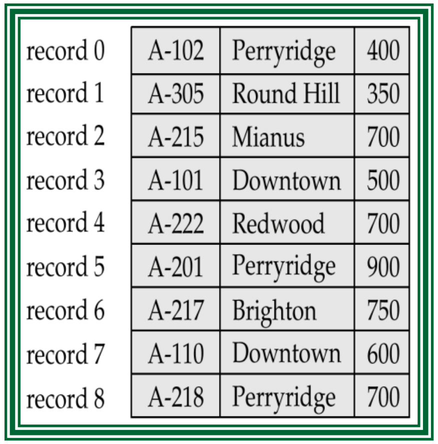
- **Files, Fixed-Length Records, and Variable-Length Records**
- File - *sequence* of records
- Fixed-Length Records - If every record in the file has exactly the same size (in bytes)
- Variable-length records - If different records in the file have different sizes

- Reasons for having variable-length records
  - The file records are of the same record type, but one or more of the fields are of varying size (**variable-length fields**). For example, the Name field of EMPLOYEE can be a variable-length field
  - The file records are of the same record type, but one or more of the fields are **optional**; that is, they may have values for some but not all of the file records (**optional fields**).

## File Organization

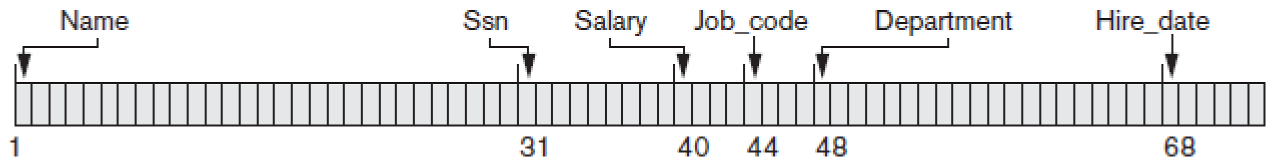
- The **database** is stored as a collection of ***files***
- Each **file** is a sequence of ***records***
- A **record** is a sequence of ***field***s
- Records are stored on disk **block**s
- A **file** can have **fixed-length** records or **variable-length** records

# Placing File Records on Disk



| | | | |
|---|---|---|---|
| record 0 | A-102 | Perryridge | 400 |
| record 1 | A-305 | Round Hill | 350 |
| record 2 | A-215 | Mianus | 700 |
| record 3 | A-101 | Downtown | 500 |
| record 4 | A-222 | Redwood | 700 |
| record 5 | A-201 | Perryridge | 900 |
| record 6 | A-217 | Brighton | 750 |
| record 7 | A-110 | Downtown | 600 |
| record 8 | A-218 | Perryridge | 700 |

- Fixed Length Records
- The fixed-length EMPLOYEE records in Figure have a record size of 71 bytes

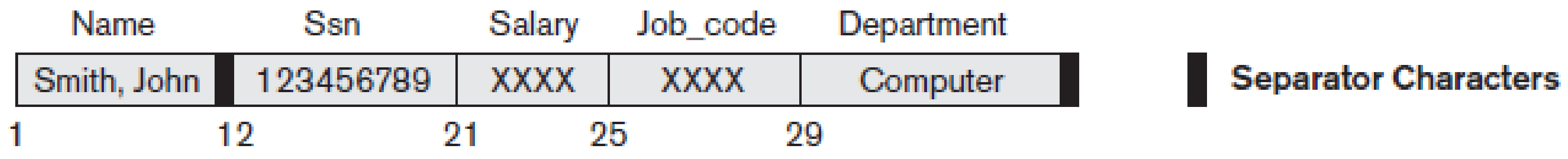


- Space is wasted when certain records do not have values for all the physical spaces provided in each record

# Placing File Records on Disk

- Variable-Length Records

- For *variable-length fields,* each record has a value for each field, but we do not know the exact length of some field values.
- To determine the bytes within a particular record that represent each field, we can use special **separator** characters (such as ? Or % or $) - which do not appear in any field value—to terminate variable-length fields

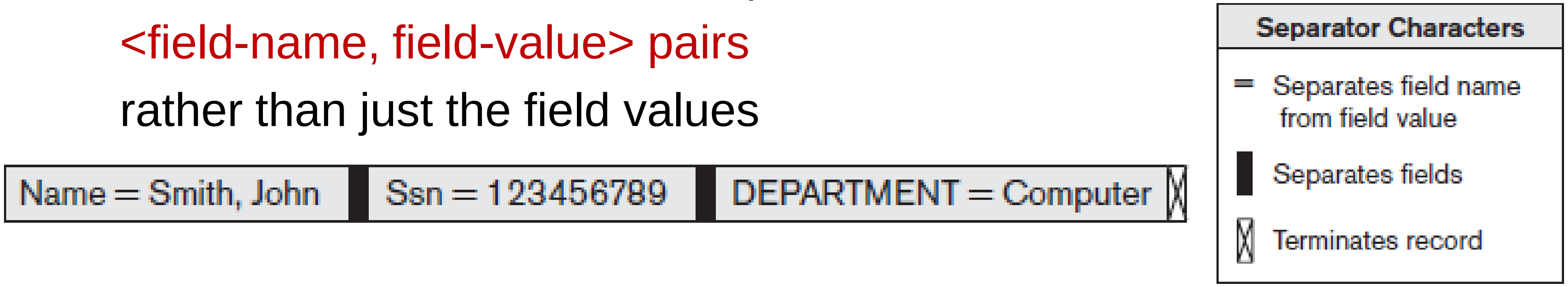


# Placing File Records on Disk

- Variable-Length Records
- A file of records with *optional fields* can be formatted in different ways.
- If the total number of fields for the record type is large, but the number of fields that actually appear in a typical record is small, we can include in each record a sequence of

  <field-name, field-value> pairs

  rather than just the field values



- A more practical option – to assign a short **field type** code—say, an integer number-to each field and include in each record a sequence of <field-type, field-value> pairs rather than <field-name, field-value> pairs

# Record Blocking and Spanned versus Unspanned Records

- The records of a file must be allocated to disk blocks because a block is the *unit of data transfer* between disk and memory
- When the block size > the record size, each block will contain numerous records, although some files may have unusually large records that cannot fit in one block

- Suppose that the block size is *B* bytes
- For a file of fixed-length records of size *R* bytes, with $B \geq R$, we can fit $bfr = \lfloor B/R \rfloor$ records per block, where the $\lfloor (x) \rfloor$ (*floor function*) *rounds down* the number *x* to an integer
- *Bfr* - called the **blocking factor – number of records per block**
- In general, *R* may not divide *B* exactly, so we have some unused space in each block equal to $B - (bfr * R)$ bytes

# Blocking Factor

- **Blocking Factor** (*bfr*) - the number of records that can fit into a single block.
  - *bfr* = ⌊*B*/*R*⌋
    - *B* : Block size in bytes
    - *R*: Record size in bytes

- Example:
  - Record size *R* = 100 bytes
  - Block Size *B* = 2,000 bytes
  - Thus the blocking factor *bfr* = floor(2000/100) = 20

- The number of blocks *b* needed to store a file of *r* records:
  - *b* = floor(*r* / *bfr)*blocks

## Record Blocking and Spanned versus Unspanned Records

- Spanned organization of records
  - To utilize this unused space, we can store part of a record on one block and the rest on another.
  - A **pointer** at the end of the first block points to the block containing the remainder of the record in case it is not the next consecutive block on disk
  - Spanned - records can span more than one block
  - Whenever a record is larger than a block - use spanned organization

- Unspanned organization of records
  - Records are not allowed to cross block boundaries
  - This is used with fixed-length records having $B > R$

- *Note:* **For variable-length records, either a spanned or an unspanned organization can be used**

# Record Blocking and Spanned versus Unspanned Records

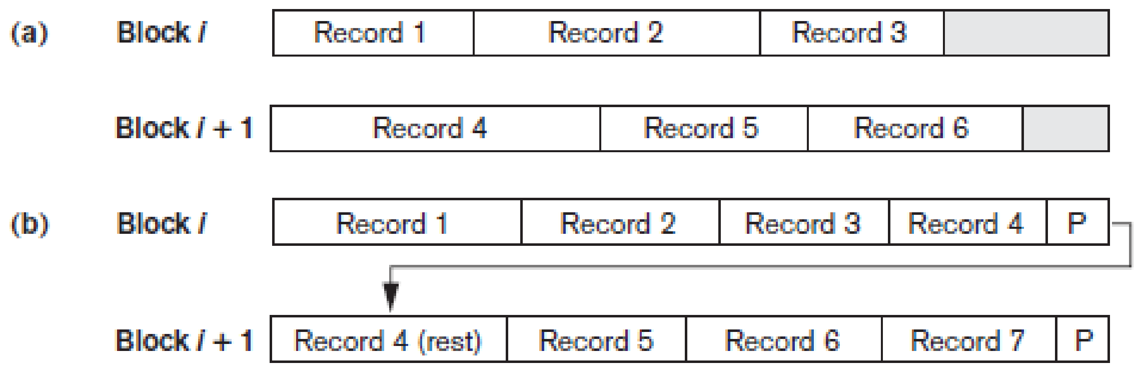


**Figure :** Types of record organization.
(a) Unspanned.
(b) Spanned

- For **variable-length records using spanned organization**, each block may store a different number of records.
- In this case, the blocking factor *bfr* represents the *average* number of records per block for the file
- *bfr* to calculate the number of blocks *b* needed for a file of *r* records

  $b = \lceil (r/bfr) \rceil$ blocks

  where the $\lceil (x) \rceil$ (*ceiling function*) rounds the value *x* up to the next integer

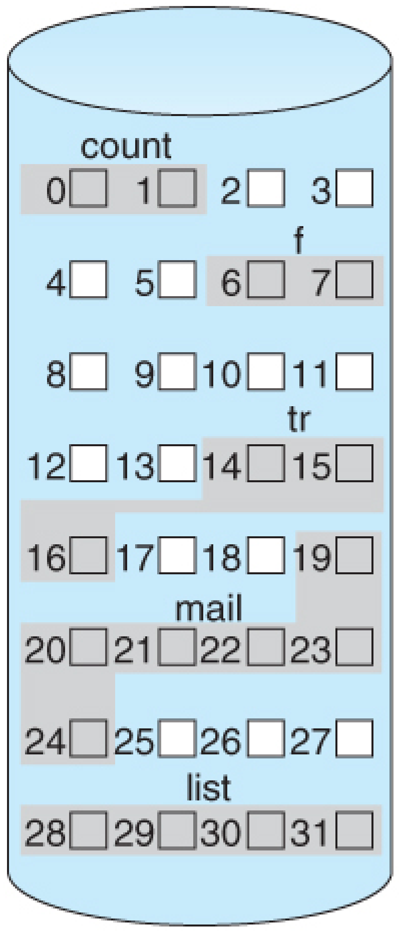
## Allocating File Blocks on Disk

- Several standard techniques for allocating the blocks of a file on disk
  - Contiguous allocation
  - linked allocation
  - Indexed allocation

- **Contiguous Allocation -** requires that all blocks of a file be kept together contiguously
- Performance is very fast, because reading successive blocks of the same file generally requires no movement of the disk heads, or at most one small step to the next adjacent cylinder
- Problems can arise when files grow, or if the exact size of a file is unknown at creation time

# Allocating File Blocks on Disk

- Contiguous Allocation

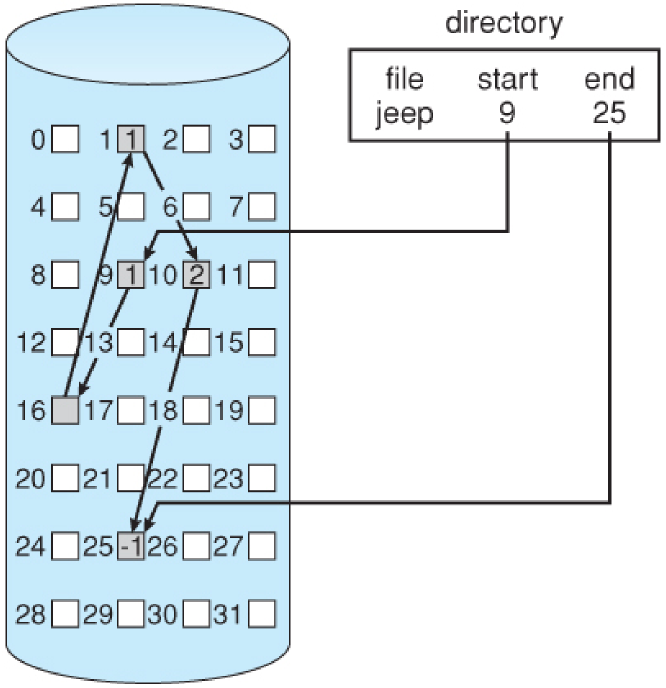


directory

| file | start | length |
|---|---|---|
| count | 0 | 2 |
| tr | 14 | 3 |
| mail | 19 | 6 |
| list | 28 | 4 |
| f | 6 | 2 |

# Allocating File Blocks on Disk

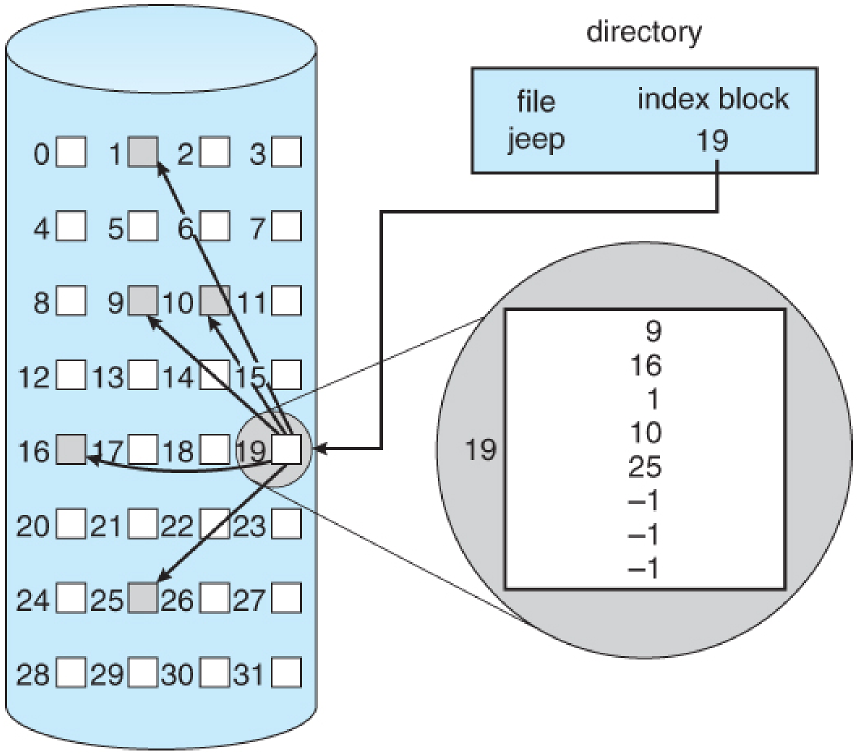
- Linked Allocation
  - Disk files can be stored as linked lists, with the expense of the storage space consumed by each link
  - Linked allocation involves no wastage of space, does not require pre-known file sizes, and allows files to grow dynamically at any time
  - A large number of seeks are needed to access every block individually



# Allocating File Blocks on Disk

- Indexed Allocation
  - One or more **index blocks** contain pointers to the actual file blocks
  - Supports direct access to the blocks occupied by the file and therefore provides fast access to the file blocks

- The indexed allocation would keep one entire block (index block) for the pointers (even for small files) which is inefficient in terms of memory utilization



## Operations on Files

- DBMS software programs, access records by using the following commands
- **OPEN** - Prepares the file for reading or writing. Allocates appropriate buffers (typically at least two) to hold file blocks from disk, and retrieves the file header. Sets the file pointer to the beginning of the file
- **Reset** - Sets the file pointer of an open file to the beginning of the file
- **Find (or Locate)-** Searches for the first record that satisfies a search condition. Transfers the block containing that record into a main memory buffer (if it is not already there). The file pointer points to the record in the buffer and it becomes the *current record*

## Operations on Files

- **Read (or Get)** - Copies the current record from the buffer to a program variable in the user program. This command may also advance the current record pointer to the next record in the file, which may necessitate reading the next file block from disk
- **FindNext** - Searches for the Searches for the next record in the file that satisfies the search condition. Transfers the block containing that record into a main memory buffer (if it is not already there)
- **Delete** -Deletes the current record and (eventually) updates the file on disk to reflect the deletion
- **Modify**-Modifies some field values for the current record and (eventually) updates the file on disk to reflect the modification
- **Insert** - Inserts a new record in the file by locating the block where the record is to be inserted, transferring that block into a main memory buffer (if it is not already there), writing the record into the buffer, and (eventually) writing the buffer to disk to reflect the insertion
- **Close** - Completes the file access by releasing the buffers and performing any other needed cleanup operations

# Operations on Files

- The preceding (except for Open and Close) are called **record-at-a-time** operations because each operation applies to a single record

- In database systems, additional **set-at-a-time** higher-level operations may be applied to a file
- **FindAll.** Locates *all* the records in the file that satisfy a search condition
- **Find (or Locate) *n*.** Searches for the first record that satisfies a search condition and then continues to locate the next *n* – 1 records satisfying the same condition. Transfers the blocks containing the *n* records to the main memory buffer (if not already there)
- **FindOrdered.** Retrieves all the records in the file in some specified order.
- **Reorganize.** Starts the reorganization process. Some file organizations require periodic reorganization. An example is to reorder the file records by sorting them on a specified field

# File Organization Vs Access Method

- **File organization**
  - Refers to the organization of the data of a file into records, blocks, and access structures; this includes the way records and blocks are placed on the storage medium and interlinked
- **Access method**
  - On the other hand, provides a group of operations that can be applied to a file

## Methods for Organizing Records of a File on Disk

- Heap file
- Sorted file
- Hash file
- RAID

# Heap Files

- Files of Unordered Records (Heap Files)
  - Also called a heap or a pile file
  - New records are inserted at the end of the file
    - Record insertion is quite efficient
  - 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
    - **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
  - Reading the records in order of a particular field requires sorting the file records
  - This organization is often used with additional access paths, such as the secondary indexes

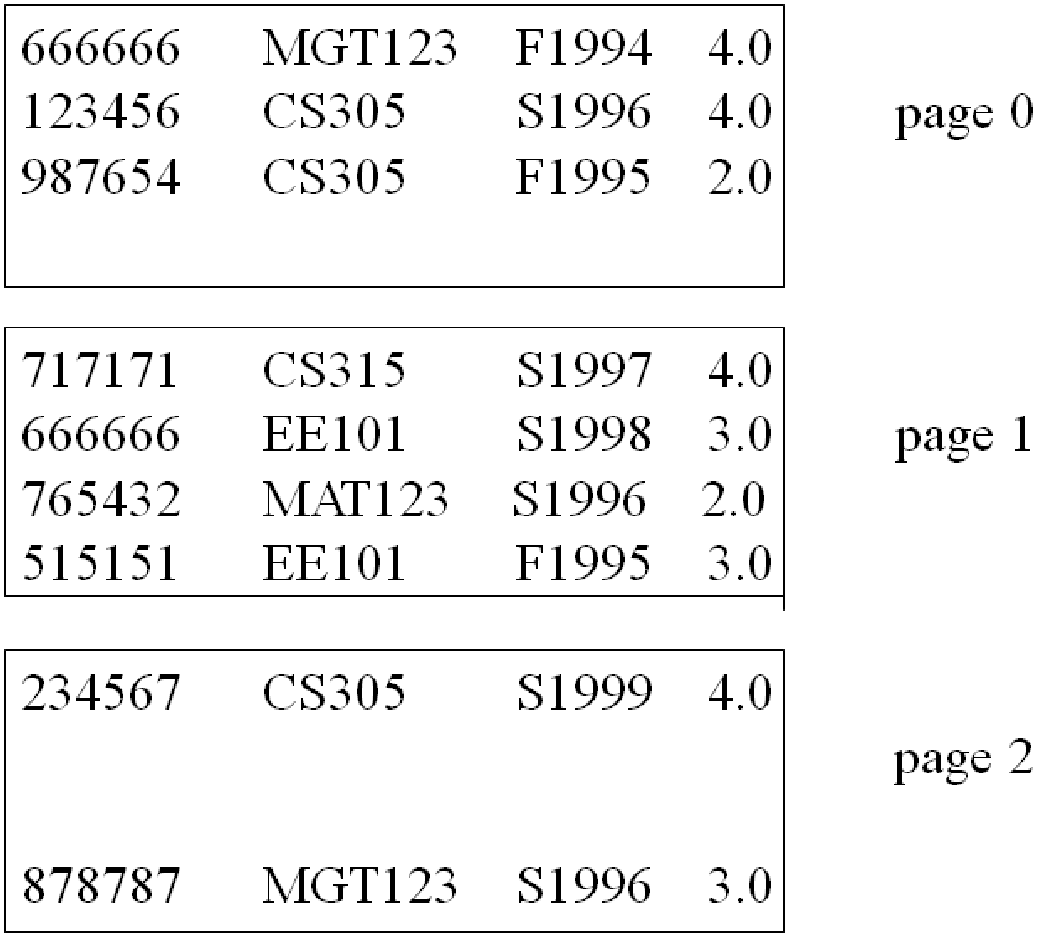
# Heap Files

- 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. This leaves unused space in the disk block.
- Deleting a large number of records in this way results in wasted storage space.
- Another technique used for record deletion is to have an extra byte or bit, called a **deletion marker**, stored with each record
- spanned or unspanned organization can be use and it may be used with either fixed-length or variable-length records
- 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

## Heap File Organization

- Records are placed in the file in the order in which they are inserted. Such an organization is called a **heap** file
  - Insertion is at the end
    - takes **constant time** *O*(1) (very efficient)
  - Searching
    - requires a **linear** search (expensive)
  - Deleting
    - requires a search, then delete
  - Select, Update and Delete
    - take *b*/2 time (**linear time**) in average
    - *b* is the number of blocks

# File Stored as a Heap File

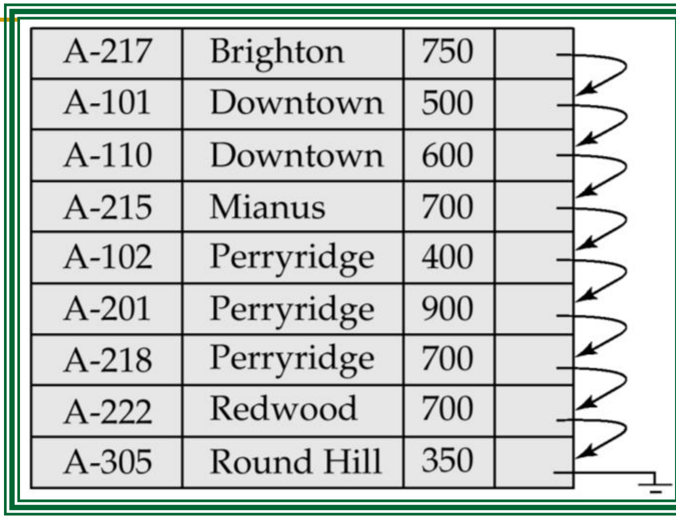


| | | | | |
|---|---|---|---|---|
| 666666 | MGT123 | F1994 | 4.0 | page 0 |
| 123456 | CS305 | S1996 | 4.0 | |
| 987654 | CS305 | F1995 | 2.0 | |
| 717171 | CS315 | S1997 | 4.0 | page 1 |
| 666666 | EE101 | S1998 | 3.0 | |
| 765432 | MAT123 | S1996 | 2.0 | |
| 515151 | EE101 | F1995 | 3.0 | |
| 234567 | CS305 | S1999 | 4.0 | page 2 |
| 878787 | MGT123 | S1996 | 3.0 | |

## Sorted Files

- **Files of Ordered Records (Sorted Files)**
  - We can physically order the records of a file on disk based on the values of one of their fields—called the **ordering field**
  - This leads to an **ordered** or **sequential** file
  - If the ordering field is also a **key field** of the file—a field guaranteed to have a unique value in each record
- Some advantages ordered files over unordered files
  - Reading the records in order of the ordering key values becomes extremely efficient because no sorting is required
  - Finding the next record from the current one in order of the ordering key usually requires no additional block accesses because the next record is in the same block as the current o
  - Using a search condition based on the value of an ordering key field results in faster access when the binary search technique is used

# Sorted Files

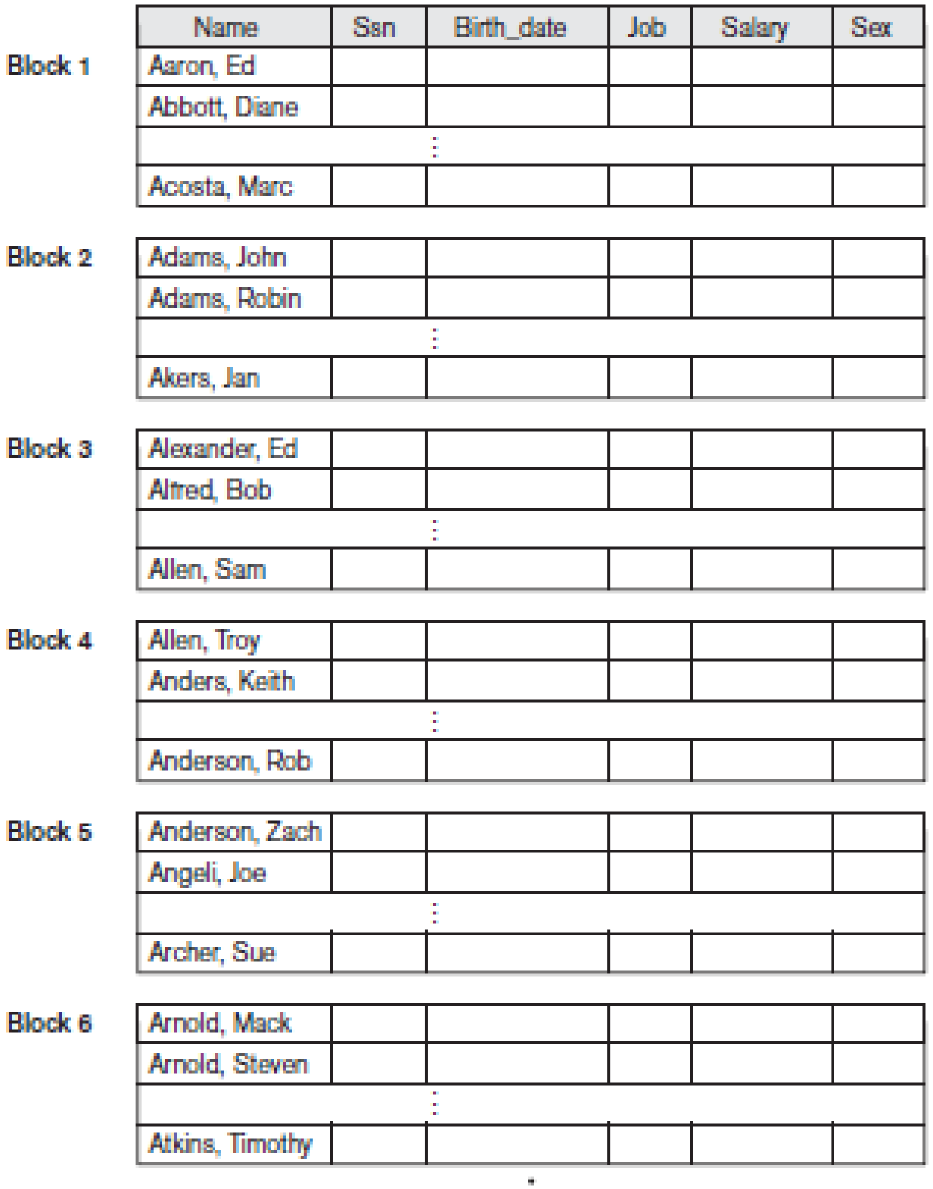


| | | |
|---|---|---|
| A-217 | Brighton | 750 |
| A-101 | Downtown | 500 |
| A-110 | Downtown | 600 |
| A-215 | Mianus | 700 |
| A-102 | Perryridge | 400 |
| A-201 | Perryridge | 900 |
| A-218 | Perryridge | 700 |
| A-222 | Redwood | 700 |
| A-305 | Round Hill | 350 |

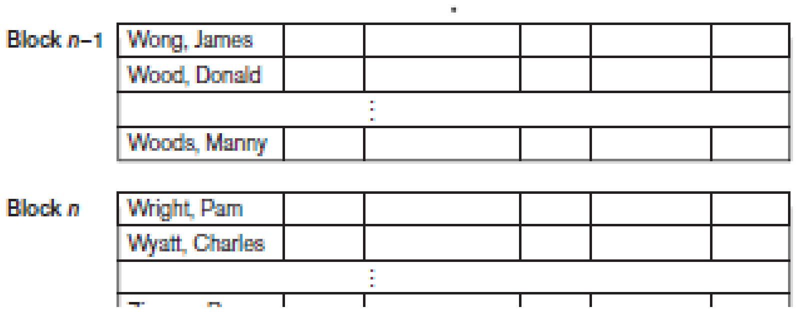
- Suppose that the file has *b* blocks numbered 1, 2, ..., *b*
- The records are ordered by ascending value of their ordering key field
- Searching for a record whose ordering key field value is *K*

- Binary search usually accesses $\log_2(b)$ blocks, whether the record is found or not
  - An improvement over linear searches, where, on the average, (*b*/2) blocks are accessed when the record is found and *b* blocks are accessed when the record is not found

# Sorted Files



| | Name | Ssn | Birth_date | Job | Salary | Sex |
|---|---|---|---|---|---|---|
| Block 1 | Aaron, Ed | | | | | |
| | Abbott, Diane | | | | | |
| | ⋮ | | | | | |
| | Acosta, Marc | | | | | |
| Block 2 | Adams, John | | | | | |
| | Adams, Robin | | | | | |
| | ⋮ | | | | | |
| | Akers, Jan | | | | | |
| Block 3 | Alexander, Ed | | | | | |
| | Alfred, Bob | | | | | |
| | ⋮ | | | | | |
| | Allen, Sam | | | | | |
| Block 4 | Allen, Troy | | | | | |
| | Anders, Keith | | | | | |
| | ⋮ | | | | | |
| | Anderson, Rob | | | | | |
| Block 5 | Anderson, Zach | | | | | |
| | Angeli, Joe | | | | | |
| | ⋮ | | | | | |
| | Archer, Sue | | | | | |
| Block 6 | Arnold, Mack | | | | | |
| | Arnold, Steven | | | | | |
| | ⋮ | | | | | |
| | Atkins, Timothy | | | | | |



| | | | | | | |
|---|---|---|---|---|---|---|
| Block $n-1$ | Wong, James | | | | | |
| | Wood, Donald | | | | | |
| | ⋮ | | | | | |
| | Woods, Manny | | | | | |
| Block $n$ | Wright, Pam | | | | | |
| | Wyatt, Charles | | | | | |
| | ⋮ | | | | | |

**Figure**
Some blocks of an ordered (sequential) file of EMPLOYEE records with Name as the ordering key field

# Sorted Files

| | | | |
|---|---|---|---|
| 111111 | MGT123 | F1994 | 4.0 |
| 111111 | CS305 | S1996 | 4.0 |
| 123456 | CS305 | F1995 | 2.0 |

page 0

| | | | |
|---|---|---|---|
| 123456 | CS315 | S1997 | 4.0 |
| 123456 | EE101 | S1998 | 3.0 |
| 232323 | MAT123 | S1996 | 2.0 |
| 234567 | EE101 | F1995 | 3.0 |

page 1

| | | | |
|---|---|---|---|
| 234567 | CS305 | S1999 | 4.0 |
| 313131 | MGT123 | S1996 | 3.0 |

page 2

## Sorted Files

- **Inserting and deleting records are expensive operations for an ordered file because the records must remain physically ordered**
- Insert
    - To insert a record, we must find its correct position in the file, based on its ordering field value, and then make space in the file to insert the record in that position.
    - For a large file this can be very time consuming because, on the average, half the records of the file must be moved to make space for the new record
- Delete
    - For record deletion, the problem is less severe if deletion markers and periodic reorganization are used

## Sorted Files

- Modification
- Modifying a field value of a record depends on two factors: the search condition to locate the record and the field to be modified
  - Search Condition
  - 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
  - Field to be modified
  - 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

## Sequential File Organization

- Insertion is expensive
  - records must be inserted in the correct order
    - locate the position where the record is to be inserted
    - if there is free space insert there
    - if no free space insert the record in **an overflow block**
    - In either case, pointer chain must be updated
  - Insert takes $\mathbf{lg_2(\mathit{b})}$ plus the time to re-organize records
  - *b* is the number of blocks
- Deletion
  - use pointer chains
- Searching
  - very efficient (Binary search)
  - This requires $\mathbf{lg_2(\mathit{b})}$ on the average

## Sorted Files

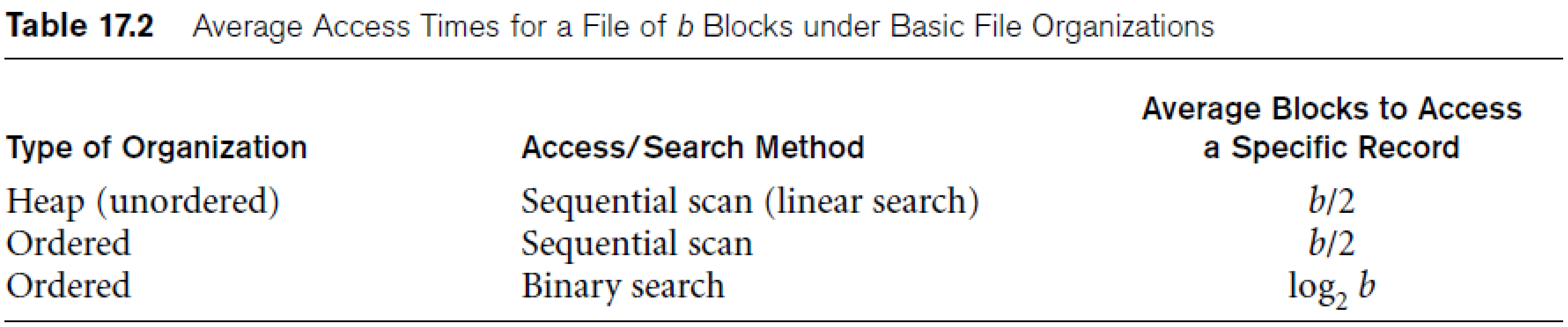


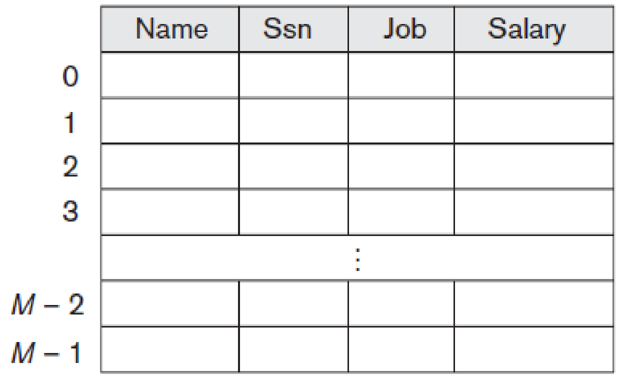
**Table 17.2** Average Access Times for a File of *b* Blocks under Basic File Organizations

| **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 | $\log_2 b$ |

# Hash Files

- Another type of primary file organization is based on hashing - provides **very fast access to records under certain search conditions**
- The search condition must be an equality condition on a single field, called the **hash field**
- In most cases, the hash field is also a key field of the file, in which case it is called the **hash key**
- Idea - to provide a function *h*, called a **hash function** or **randomizing function**, which is applied to the hash field value of a record and yields the *address* of the disk block in which the record is stored
- A search for the record within the block can be carried out in a main memory buffer. For most records, we need only a single-block access to retrieve that record

# Hash Files



| | Name | Ssn | Job | Salary |
|---|---|---|---|---|
| 0 | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| | ⋮ | | | |
| $M-2$ | | | | |
| $M-1$ | | | | |

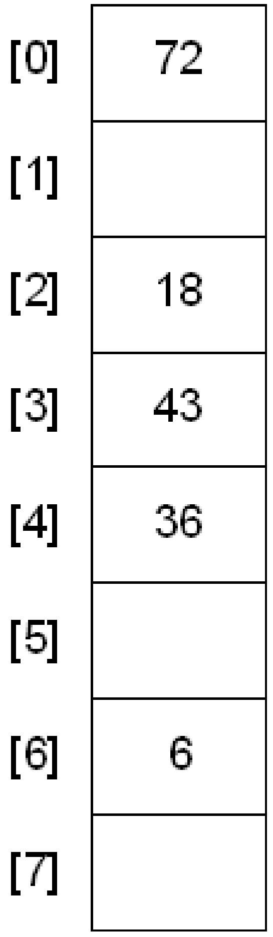
- **Internal Hashing**
  - For internal files, hashing is typically implemented as a **hash table** through the use of an array of records. Suppose that the array index range is from 0 to $M-1$
  - we have $M$ **slots** whose addresses correspond to the array indexes.
  - Choose a hash function that transforms the hash field value into an integer between 0 and $M-1$.
  - One common hash function is the $h(K) = K$ **mod** $\boldsymbol{M}$ function - which returns the remainder of an integer hash field value $K$ after division by $M$; this value is then used for the record address

## Hash Files

Assume a table with 8 slots:

Hash key = key % table size

|  |  |  |
|---|---|---|
| 4 | = | 36 % 8 |
| 2 | = | 18 % 8 |
| 0 | = | 72 % 8 |
| 3 | = | 43 % 8 |
| 6 | = | 6 % 8 |



# Hash Files

- Other hashing functions can be used
  - Folding
    - Involves applying an arithmetic function such as *addition* or a logical function such as *exclusive or* to different portions of the hash field value to calculate the hash address
    - For example, with an address space from 0 to 999 to store 1,000 keys, a 6-digit key 235469 may be folded and stored at the address: (235+964) mod 1000 = 199)
- Another technique involves picking some digits of the hash field value
  - For instance, the third, fifth, and eighth digits—to form the hash address 301-67-8923 a hash value of 172 by this hash function

## Hash Files

- Most hashing functions is that they do not guarantee that distinct values will hash to distinct addresses

- **Hash collision**
  - Occurs when the hash field value of a record that is being inserted hashes to an address that already contains a different record
  - In this situation, we must insert the new record in some other position, since its hash address is occupied
  - The process of finding another position is called **collision resolution**

# Hash Files

- 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. Additionally, 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
  - **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

## Hash Files

- **Hash Collision - Open addressing**

| | |
|---|---|
| [0] | 72 |
| [1] | |
| [2] | 18 |
| [3] | 43 |
| [4] | 36 |
| [5] | |
| [6] | 6 |
| [7] | |

Add the keys 10, 5, and 15 to the previous table .

Hash key = key % table size

2 = 10 % 8

5 = 5 % 8

7 = 15 % 8

| | |
|---|---|
| [0] | 72 |
| [1] | 15 |
| [2] | 18 |
| [3] | 43 |
| [4] | 36 |
| [5] | 10 |
| [6] | 6 |
| [7] | 5 |

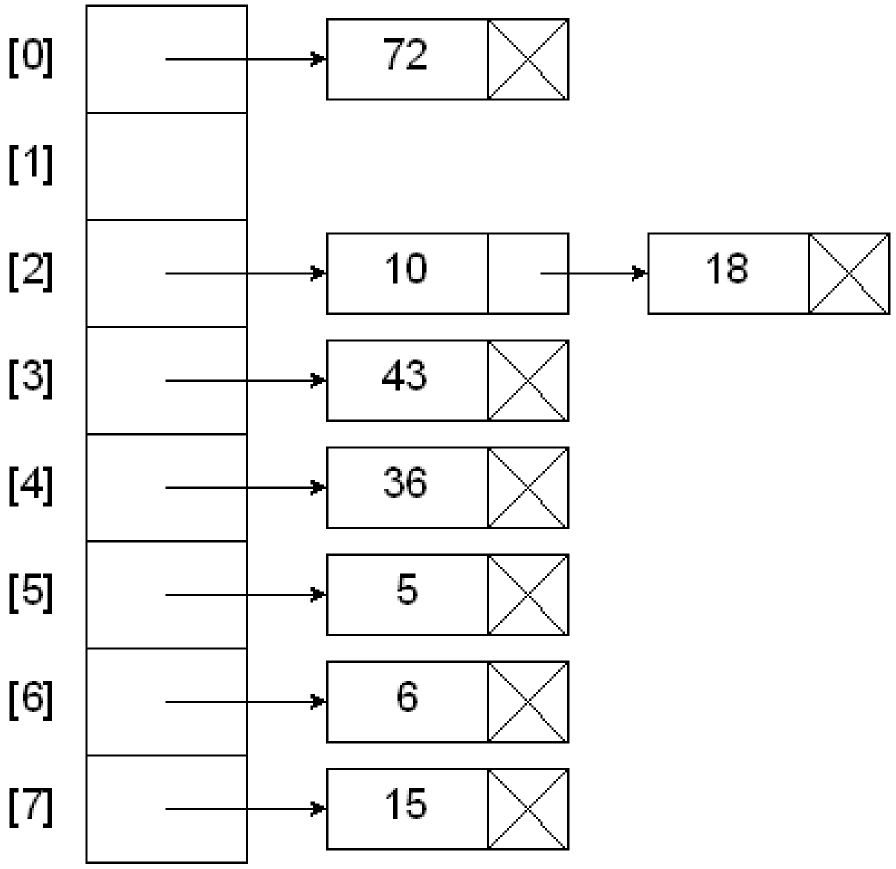
# Hash Files

**Hashing with Chains**

When a collision occurs, elements with the same hash key will be **chained** together. A **chain** is simply a linked list of all the elements with the same hash key.

Hash key = key % table size

$4 = 36 \% 8$
$2 = 18 \% 8$
$0 = 72 \% 8$
$3 = 43 \% 8$
$6 = 6 \% 8$
$2 = 10 \% 8$
$5 = 5 \% 8$
$7 = 15 \% 8$



# Hash Files

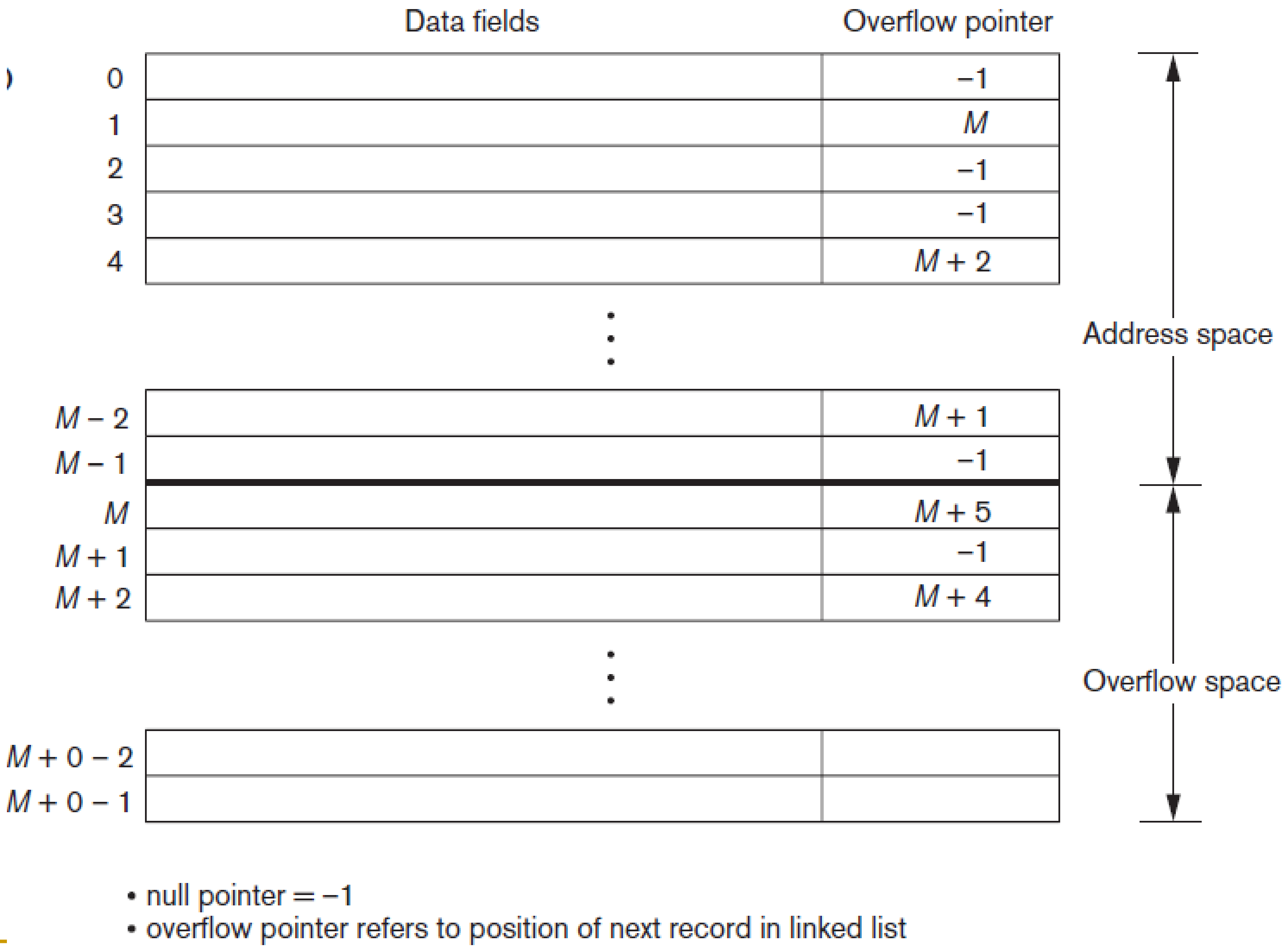


**Figure : Collision resolution by chaining records**

## Hash Files

- **External Hashing for Disk Files**
  - Hashing for disk files is called **external hashing**
  - The target address space is made of **buckets**, each of which holds multiple records.
  - A bucket is either one disk block or a cluster of contiguous disk blocks.
  - The hashing function maps a key into a relative bucket number, rather than assigning an absolute block address to the bucket.
  - A table maintained in the file header converts the bucket number into the corresponding disk block address

# Hash Files

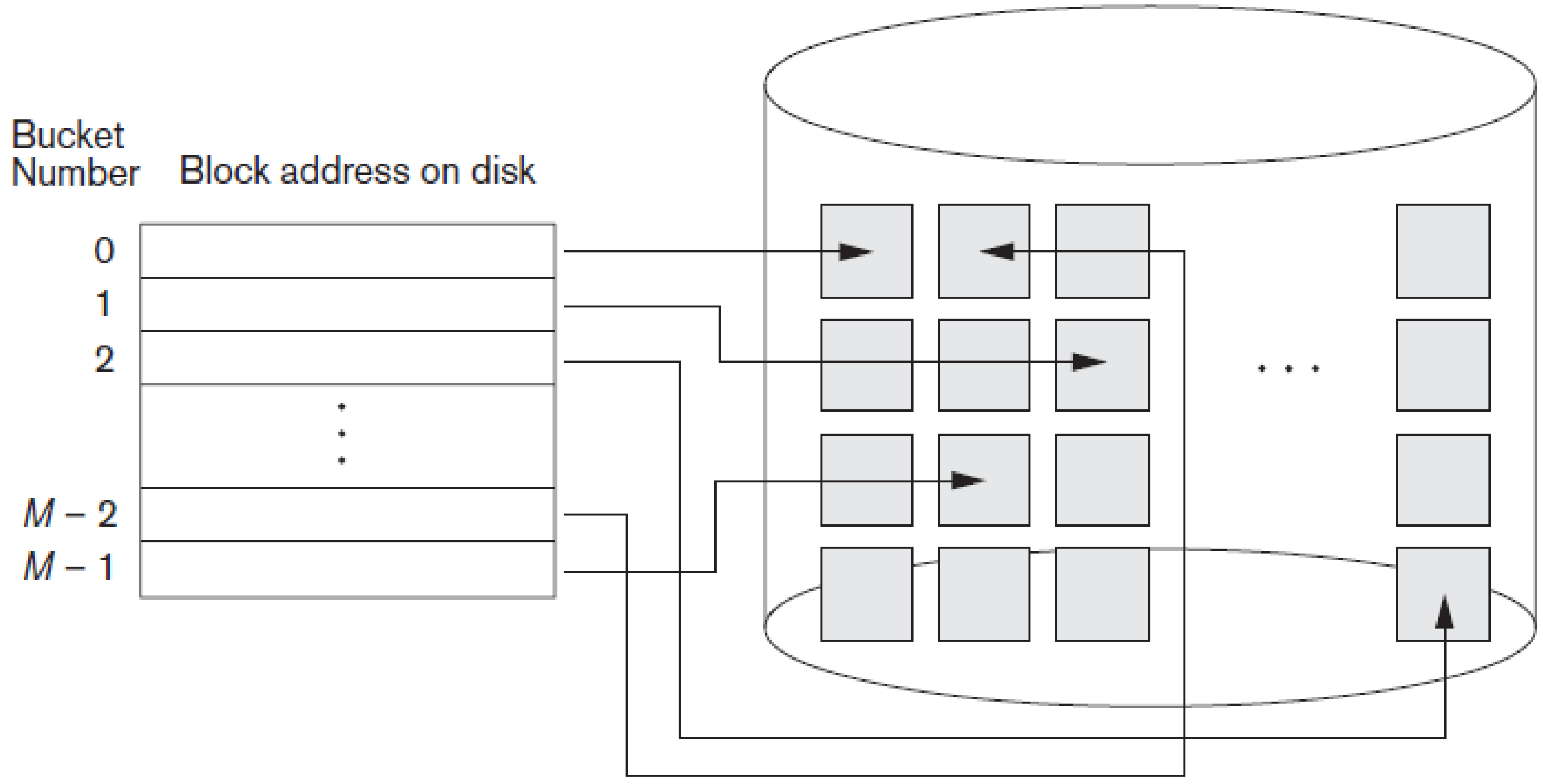


**Figure:** Matching bucket numbers to disk block addresses

# Hash Files

- Collision problem is less severe with buckets - as many records as will fit in a bucket can hash to the same bucket without causing problems
- A variation of chaining in which a pointer is maintained in each bucket to a linked list of overflow records for the bucket
- The pointers in the linked list should be **record pointers**, which include both a block address and a relative record position within the block

- The hashing scheme described so far is called **static hashing** because a fixed number of buckets *M* is allocated.

# Hash Files

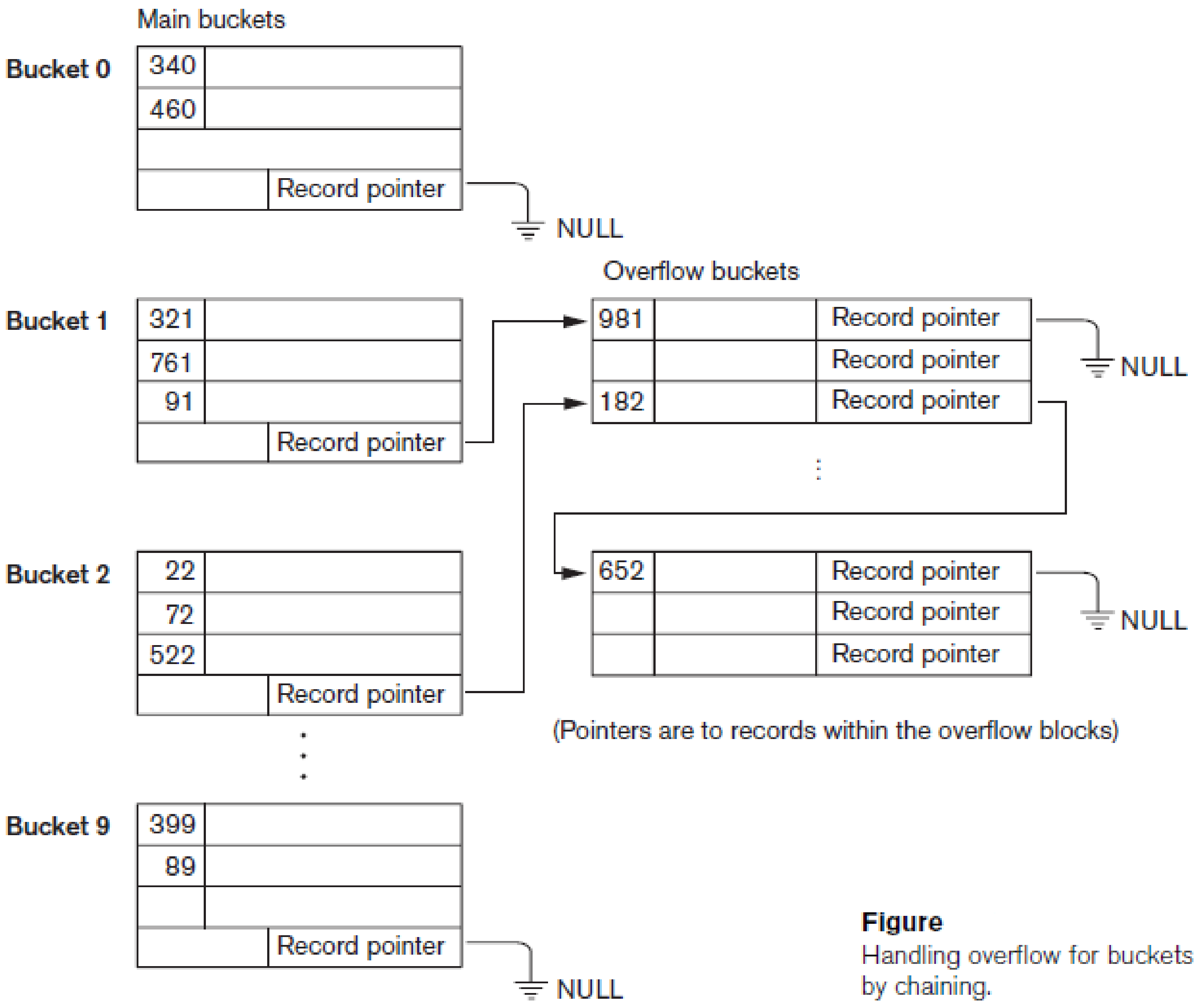


**Figure**
Handling overflow for buckets by chaining.

# Hashing Techniques

- The hashing scheme is called **static hashing** if a fixed number of buckets is allocated
- Main disadvantage of **static** external hashing:
  - The number of buckets must be chosen large enough that can handle large files. That is, it is difficult to expand or shrink the file dynamically.
- Solutions to the above problem
  - Dynamic hashing
  - Extendible hashing
  - Linear hashing

## Hashing for Dynamic File Organization

- Hashing for Dynamic File Organization
- **Dynamic Files**
  - Files where record insertions and deletion take place frequently
  - The file keeps growing and also shrinking

- Hashing for dynamic file organization
  - Bucket numbers are integers
  - The binary representation of bucket numbers
    - Exploited cleverly to devise dynamic hashing schemes

# Dynamic And Extendible Hashed Files

- Dynamic and Extendible Hashing Techniques
  - Hashing techniques are adapted to allow the dynamic growth and shrinking of the number of file records
  - These techniques include the following: **dynamic hashing, extendible hashing**, and **linear hashing**
- Both dynamic and extendible hashing use the **binary representation** of the hash value h(K) in order to access a **directory**
  - In dynamic hashing, the directory is a binary tree
  - In extendible hashing the directory is an array of size $2^d$ where d is called the **global depth**
    - The value of *d* can be increased or decreased by one at a time, thus doubling or halving the number of entries in the directory array
    - **Doubling is needed if a bucket, whose local depth *d* is equal to the global depth *d*, overflows**

## Dynamic And Extendible Hashed Files

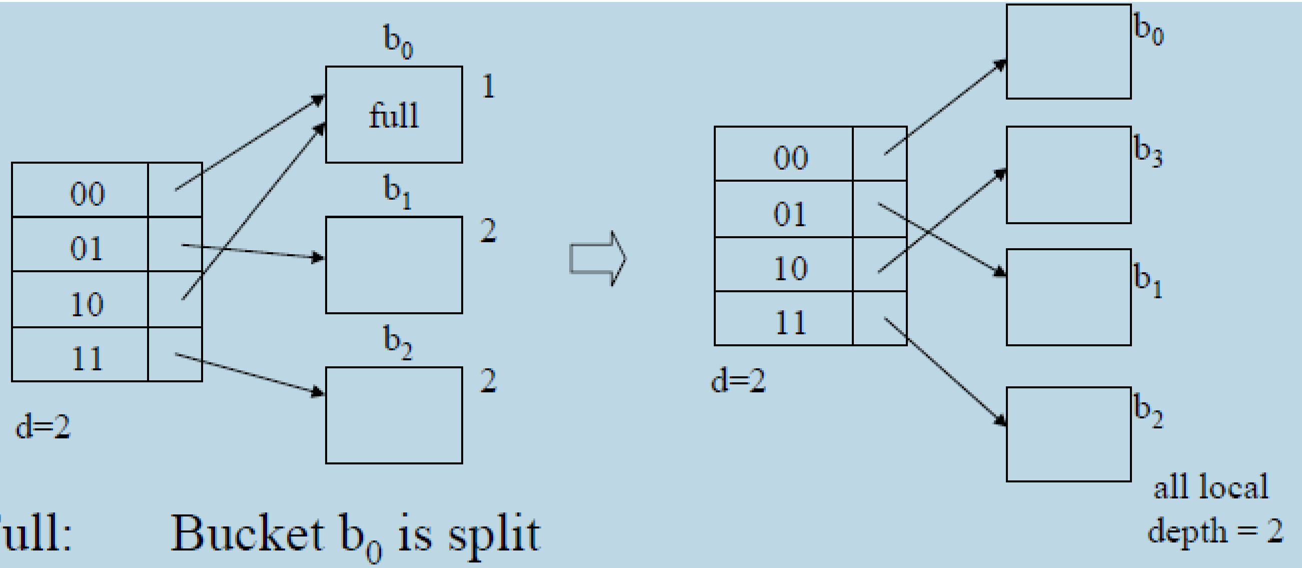
- The directories can be stored on disk, and they expand or shrink dynamically
  - Directory entries point to the disk blocks that contain the stored records

- An insertion in a disk block that is full causes the block to split into two blocks and the records are redistributed among the two blocks
  - The directory is updated appropriately
- Dynamic and extendible hashing do not require an overflow area.

- Linear hashing does require an overflow area but does not use a directory
  - Blocks are split in *linear order* as the file expands

# Insertion in Extendible Hashing Scheme

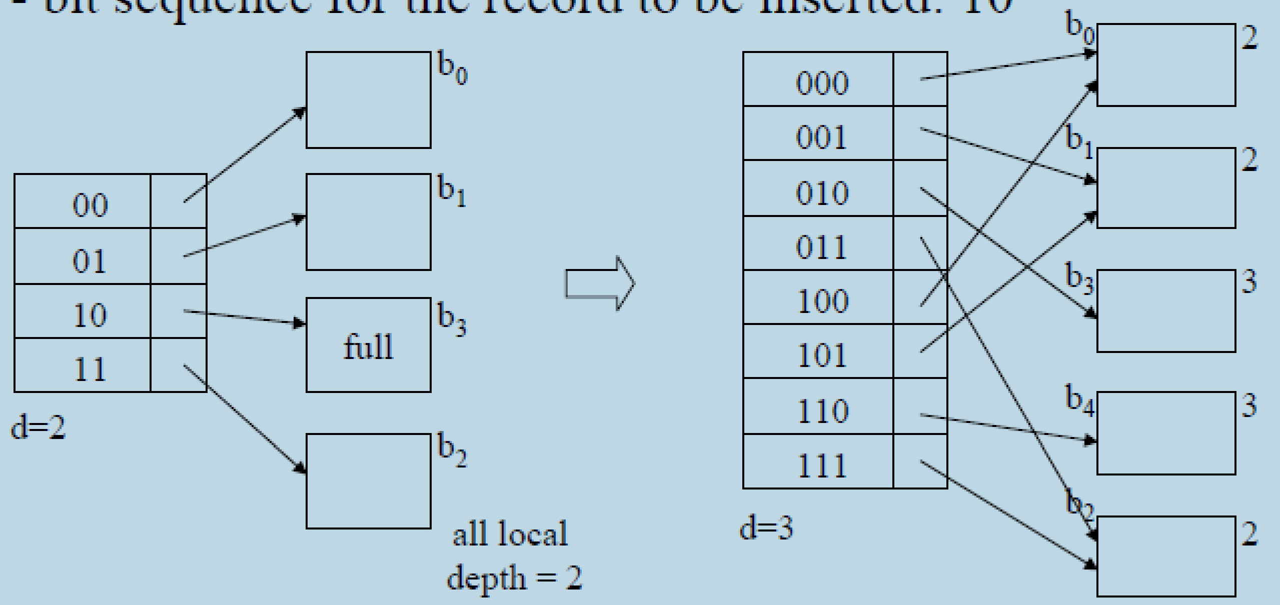
- 2 -bit sequence for the record to be inserted



$b_0$ Full: Bucket $b_0$ is split
All records whose 2-bit sequence is ‘10’ are sent to a new bucket $b_3$. Others are retained in $b_0$
Directory is modified.
$b_0$ Not full: New record is placed in $b_0$. No changes in the directory.

# Insertion in Extendible Hashing Scheme

2 - bit sequence for the record to be inserted: 10

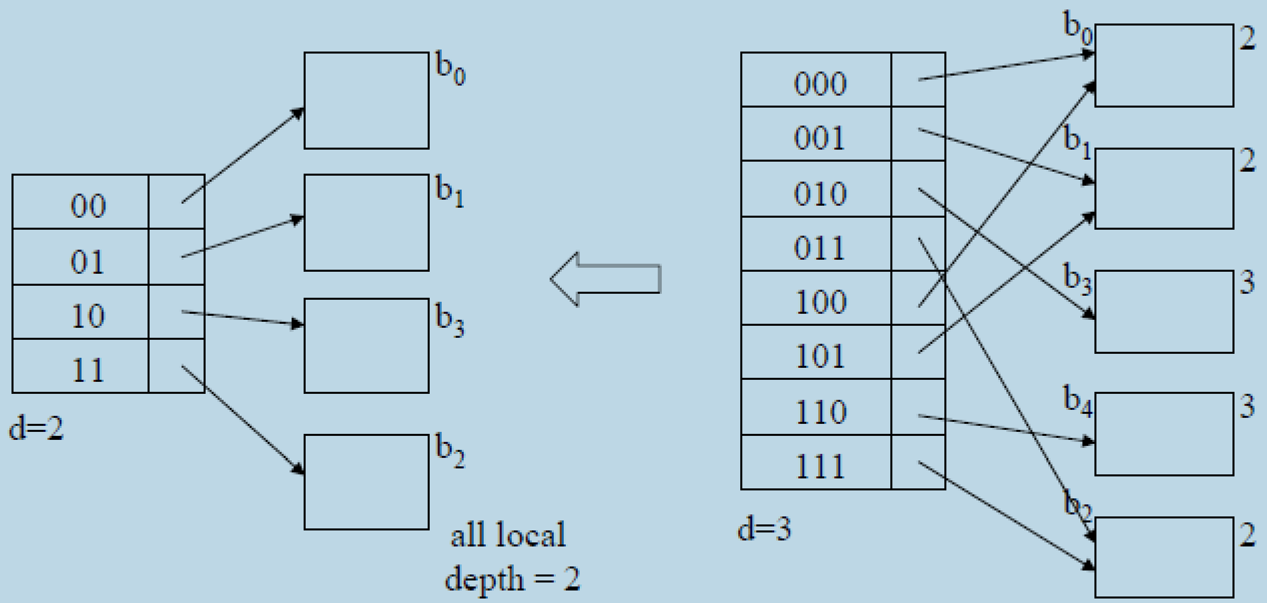


$b_3$ not full: new record placed in $b_3$. No changes.

$b_3$ full : $b_3$ is split, directory is doubled, all records with 3-bit sequence 110 sent to $b_4$. Others in $b_3$.

In general, if the local depth of the bucket to be split is equal to the global depth, directory is doubled

# Deletion in Extendible Hashing Scheme



Matching pair of data buckets:

k-bit sequences have a common k-1 bit suffix, e.g, $b_3$ & $b_4$

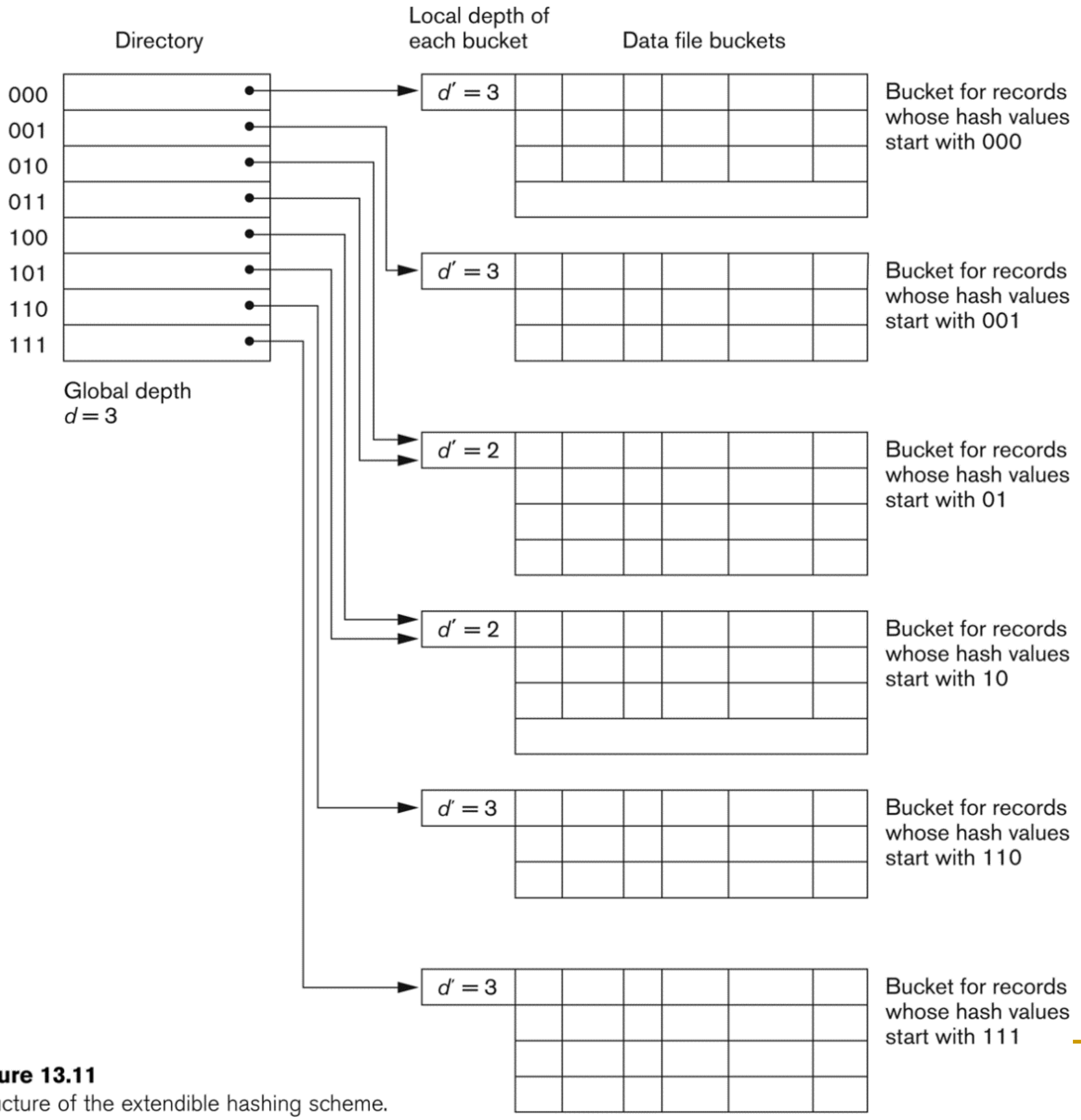
Due to deletions, if a pair of matching data buckets

-- become less than half full – *try* to merge them into one bucket

If the local depth of all buckets is one less than the global depth

-- reduce the directory to half its size

# Extendible Hashing



Directory
Local depth of each bucket
Data file buckets
000
001
010
011
100
101
110
111
Global depth
d = 3
d′ = 3
Bucket for records whose hash values start with 000
d′ = 3
Bucket for records whose hash values start with 001
d′ = 2
Bucket for records whose hash values start with 01
d′ = 2
Bucket for records whose hash values start with 10
d′ = 3
Bucket for records whose hash values start with 110
d′ = 3
Bucket for records whose hash values start with 111

**Figure 13.11**
Structure of the extendible hashing scheme.

# Dynamic Hashing

- A precursor to extendible hashing was **dynamic hashing**
- The storage of records in buckets for dynamic hashing is somewhat similar to extendible hashing.
- The major difference is in the organization of the directory
- Dynamic hashing maintains a tree-structured directory with two types of nodes:
  - Internal nodes that have two pointers—the left pointer corresponding to the 0 bit (in the hashed address) and a right pointer corresponding to the 1 bit
  - Leaf nodes—these hold a pointer to the actual bucket with records

# Dynamic Hashing

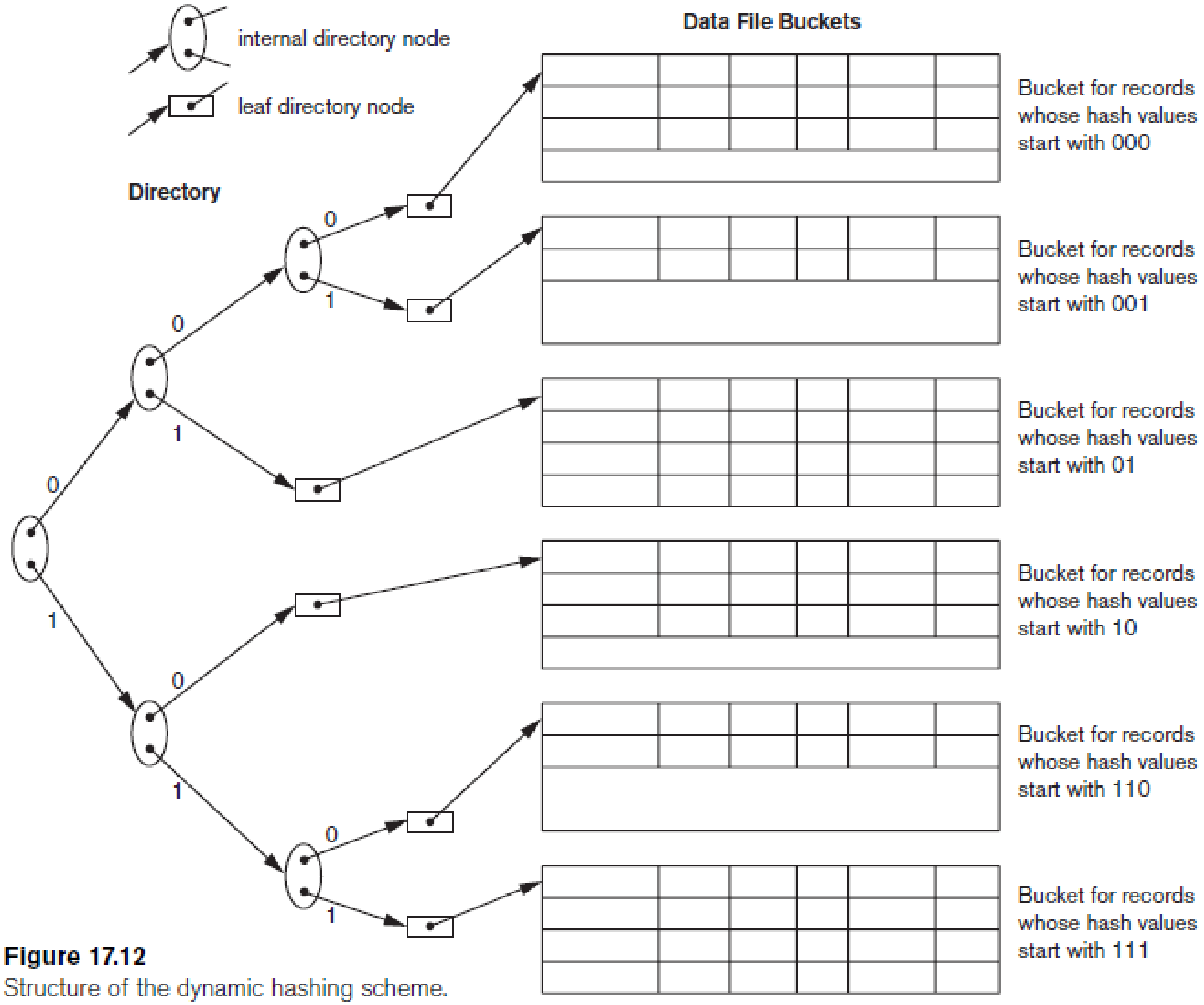

internal directory node
leaf directory node
Directory
Data File Buckets
0
1
Bucket for records whose hash values start with 000
Bucket for records whose hash values start with 001
Bucket for records whose hash values start with 01
Bucket for records whose hash values start with 10
Bucket for records whose hash values start with 110
Bucket for records whose hash values start with 111

**Figure 17.12**
Structure of the dynamic hashing scheme.

# Linear Hashing

- Idea - is to allow a hash file to expand and shrink its number of **buckets dynamically without needing a directory**

- Starts with $M$ buckets numbered 0, 1, ..., $M - 1$ and uses the mod hash function $h(K) = K \bmod M$; this hash function is called the **initial hash function** $h_i$.
- Overflow because of collisions is still needed and can be handled by maintaining individual overflow chains for each bucket
- When a collision leads to an overflow record in *any* file bucket, the *first* bucket in the file - bucket 0-is split into two buckets:
  - The original bucket 0 and a new bucket $M$ at the end of the file.
  - The records originally in bucket 0 are distributed between the two buckets based on a different hashing function $h_i$+1($K$) = $K \bmod 2M$

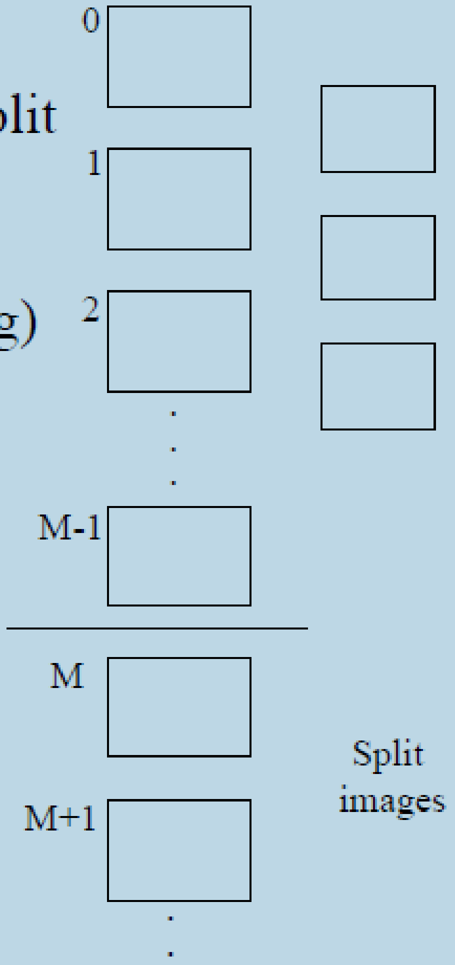
# Linear Hashing

- A key property of the two hash functions $h_i$ and $h_{i+1}$ is that any records that hashed to bucket 0 based on *hi* will hash to either bucket 0 or bucket *M* based on $h_{i+1}$; this is necessary for linear hashing to work
- As further collisions lead to overflow records, additional buckets are split in the *linear* order 1, 2, 3, .... If enough overflows occur, all the original file buckets 0, 1, ..., $M-1$ will have been split, so the file now has 2*M* instead of *M* buckets, and all buckets use the hash function $h_{i+1}$.
- Hence, the records in overflow are eventually redistributed into regular buckets, using the function $h_{i+1}$ via a *delayed split* of their buckets

# Insertion

On first overflow,
 irrespective of where it occurs, bucket 0 is split
On subsequent overflows
 buckets 1, 2, 3, … are split in that order
 (This why the scheme is called linear hashing)
N: the next bucket to be split
After M overflows,
 all the original M buckets are split.
 We switch to hash functions $h_1$, $h_2$
  and set N = 0.

$$\begin{matrix} h_o \\ h_1 \end{matrix} \longrightarrow \begin{matrix} h_1 \\ h_2 \end{matrix} \longrightarrow \cdots \begin{matrix} h_i \\ h_{i+1} \end{matrix} \longrightarrow \cdots$$



## Linear Hashing

- **Advantages**
  - Directory is not needed
  - Simple to implement

- **Reference - Example for Linear hashing**

  http://queper.in/drupal/blogs/dbsys/linear_hashing

## Parallelizing Disk Access Using RAID Technology

- Secondary storage technology must take steps to keep up in performance and reliability with processor technology
- A major advance in secondary storage technology is represented by the development of **RAID**, which originally stood for **Redundant Arrays of Inexpensive Disks**
- The main goal of RAID is to even out the widely different rates of performance improvement of disks against those in memory and microprocessors

## RAID Technology

- A natural solution is a large array of small independent (inexpensive) disks acting as a single higher-performance logical disk
- A concept called **data striping** is used, which utilizes parallelism to improve disk performance
- Data striping distributes data transparently over multiple disks to make them appear as a single large, fast disk

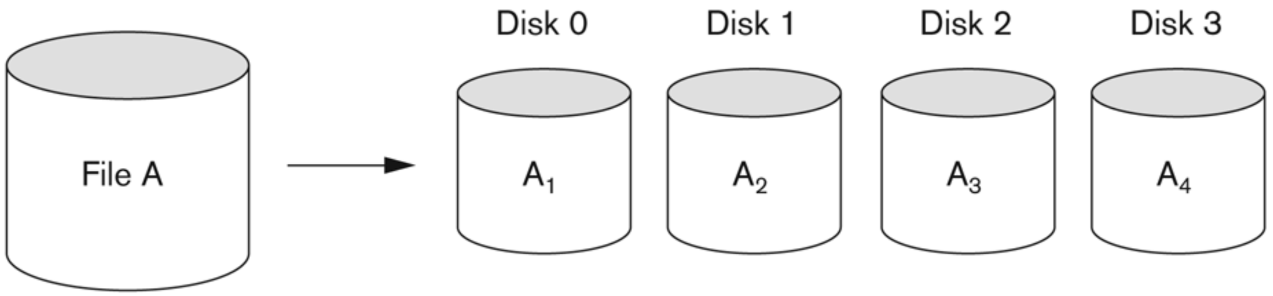


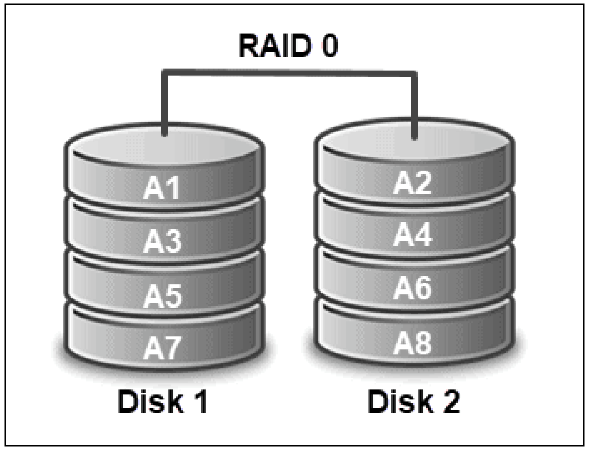
**Figure 13.12**
Data striping. File A is striped across four disks.

# RAID Technology

- Provides
  - Increased performance
  - Fault Tolerance
  - Redundancy

  - RAID Levels
    - **Level 0**
    - **Level 1**
    - **Level 2**
    - **Level 3**
    - **Level 4**
    - **Level 5**
    - **Level 6**
    - **Level 10 (1+0)**

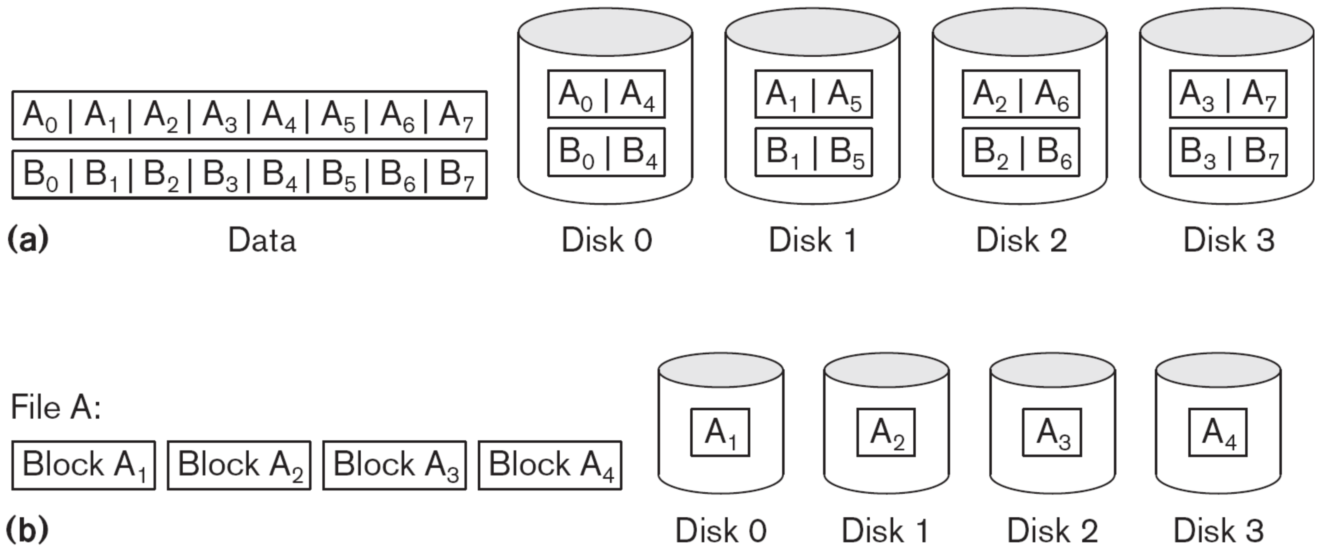
# RAID Technology



- RAID Level 0
  - Minimum number of drives required - 2
  - A RAID Level 0 system uses **data striping** **-** dividing data evenly across two or more storage devices
  - No redundant information is maintained
  - Purpose - **speed up performance** as organizing data in such a way allows faster reading and writing of files
  - **Not fault-tolerant** should not be used for critical data
  - Simple and easy to implement

# RAID Technology

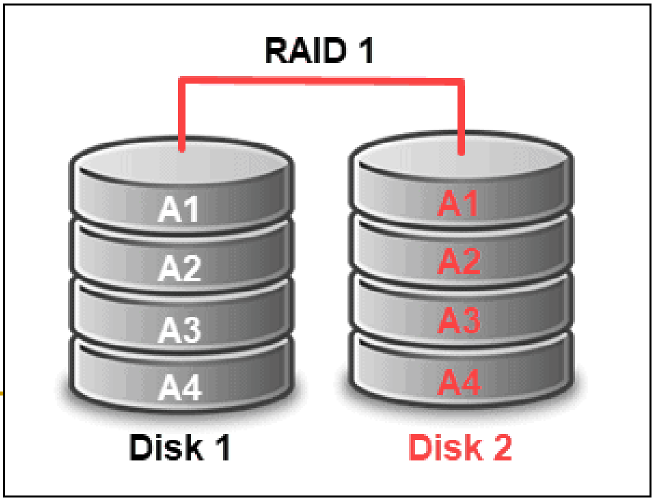
- Data striping means breaking up contiguous data that would normally go on a single disk
- The data is distributed to many disks, either by byte (a) or by block (b)



# RAID Technology

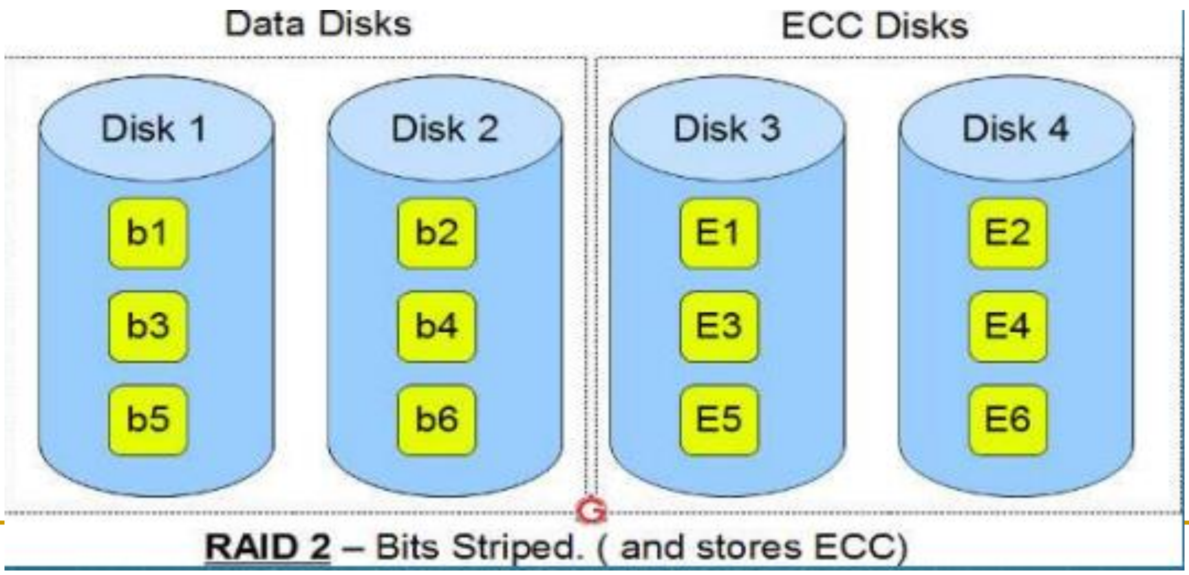
- RAID Level 1 – minimum no. of drives required - 2
- **Disk Mirroring -** is **fault-tolerant** as it duplicates data by simultaneously writing on two storage devices
- Therefore, each disk has an exact copy on another disk
- RAID 1 - ensures protection against data loss. If a problem arises with one disk, the copy provides the data needed
- Writing takes more time as it only uses the capacity of one disk and has to operate twice

- **Disadvantages**
- Uses only half of the storage capacity
- More expensive



# RAID Technology

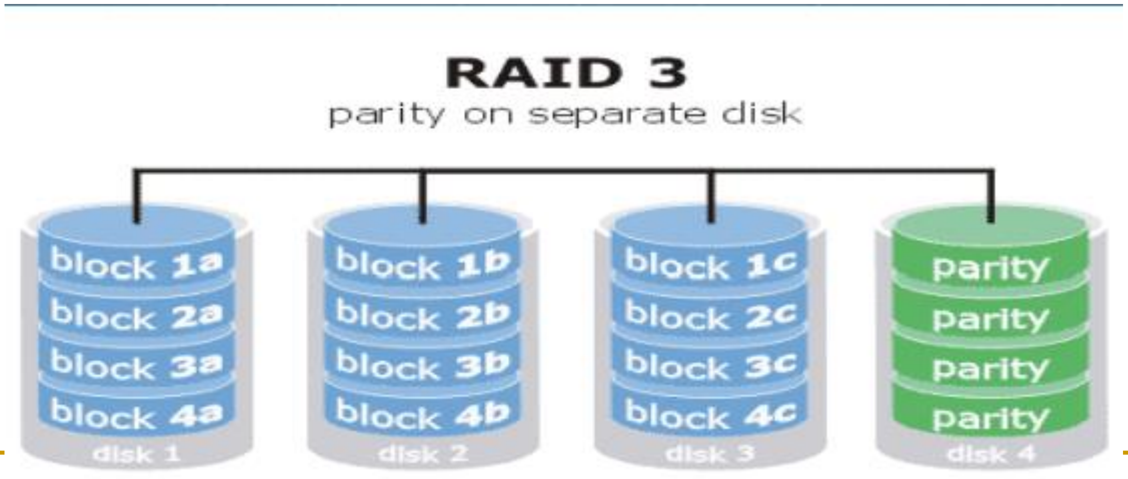
- RAID Level 2
- Bit-level striping means that the file is broken into “**bit-sized pieces**”.
- It uses a **Hamming code** for error correction
- Theoretical performance is very high, but it would be so expensive to implement



RAID 2 – Bits Striped. ( and stores ECC)

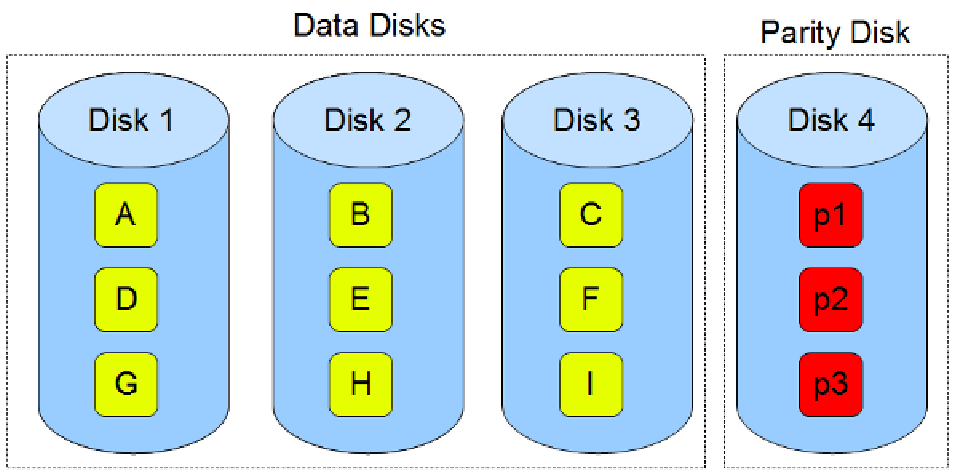
# RAID Technology

- RAID Level 3
- Requires a minimum of 3 drives to implement
- Byte-level striping means that the file is broken into "**byte sized pieces**“.
- Written in parallel on two or more drives
- An additional drive stores parity information



# RAID Technology

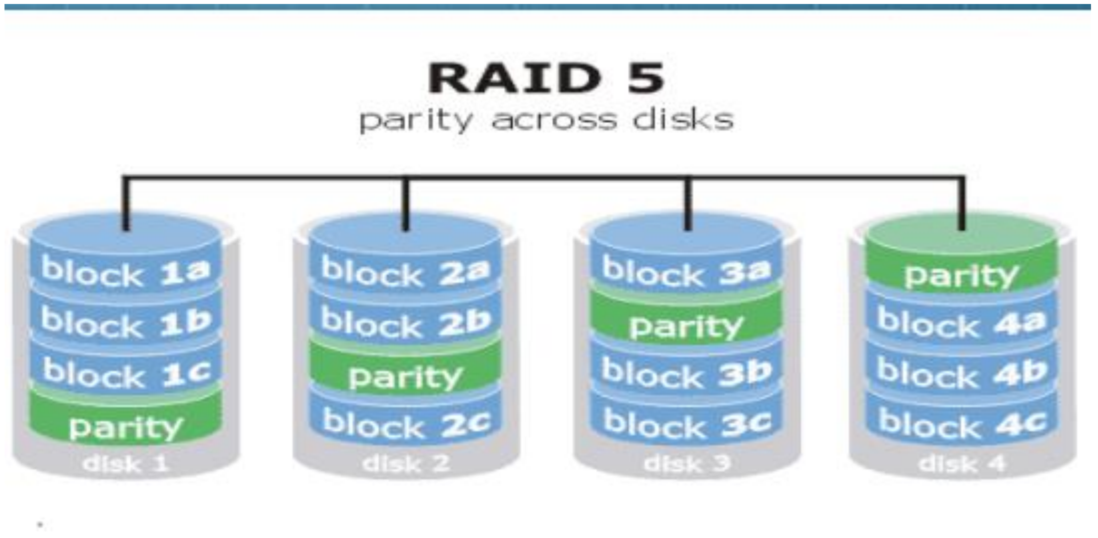
- RAID Level 4
- Minimum nos. of drives required : 3 (2 disks for data and 1 for parity)
- Level 4 provides **block-level striping** (like Level 0) with a parity disk
- If a data disk fails, the parity data is used to create a replacement disk



**RAID 4** – Blocks Striped. ( and Dedicated Parity Disk)

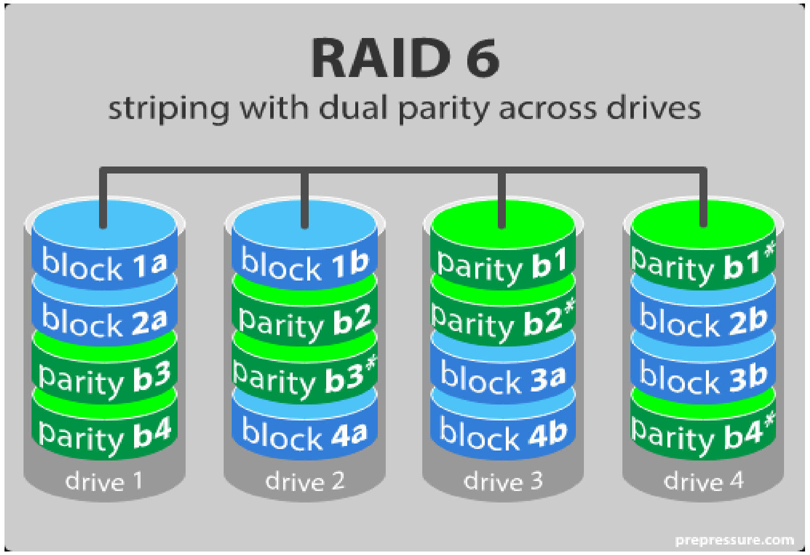
## RAID Technology

- RAID Level 5
  - Most common secure RAID level
  - Instead of a dedicated parity disk, parity information is spread across all the drives



# RAID Technology

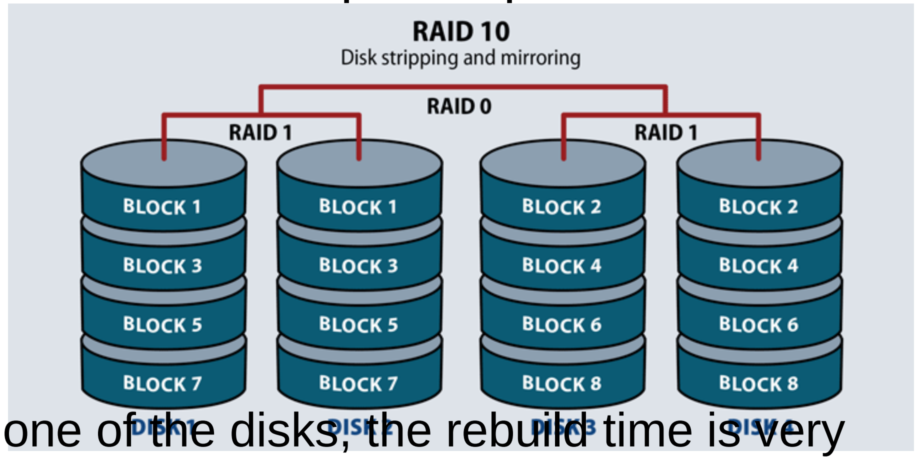
- RAID Level 6
- The parity data are written to two drives
- The chances that two drives break down at exactly the same moment are of course very small



- Advantages
  - Read data transactions are very fast
  - RAID 6 is more secure than RAID 5

## RAID Technology

- RAID level 10 – combining RAID 1 & RAID 0
- Combine the advantages of RAID 0 and RAID 1 in one single system
- Provides security by **mirroring** all data on secondary drives while using **striping** across each set of drives to speed up data transfers



- Advantage
  - If something goes wrong with one of the disks, the rebuild time is very fast since all that is needed is copying all the data from the surviving mirror to a new drive
- Disadvantage
  - Half of the storage capacity goes to mirroring. expensive way to have redundancy.