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

Chapter 1: Disk Storage and Basic File Structures

Database Management Systems (CO3021)

Computer Science Program

Dr. Võ Thị Ngọc Châu

(chauvtn@hcmut.edu.vn)

Semester 1 – 2018-2019

Course outline

- Overall Introduction to Database Management Systems
- **Chapter 1. Disk Storage and Basic File Structures**
- Chapter 2. Indexing Structures for Files
- Chapter 3. Algorithms for Query Processing and Optimization
- Chapter 4. Introduction to Transaction Processing Concepts and Theory
- Chapter 5. Concurrency Control Techniques
- Chapter 6. Database Recovery Techniques

References

- [1] R. Elmasri, S. R. Navathe, Fundamentals of Database Systems- 6th Edition, Pearson- Addison Wesley, 2011.
  - ***R. Elmasri, S. R. Navathe, Fundamentals of Database Systems- 7th Edition, Pearson, 2016.***
- [2] H. G. Molina, J. D. Ullman, J. Widom, Database System Implementation, Prentice-Hall, 2000.
- [3] H. G. Molina, J. D. Ullman, J. Widom, Database Systems: The Complete Book, Prentice-Hall, 2002
- [4] A. Silberschatz, H. F. Korth, S. Sudarshan, Database System Concepts –3rd Edition, McGraw-Hill, 1999.
- [Internet] …

Content

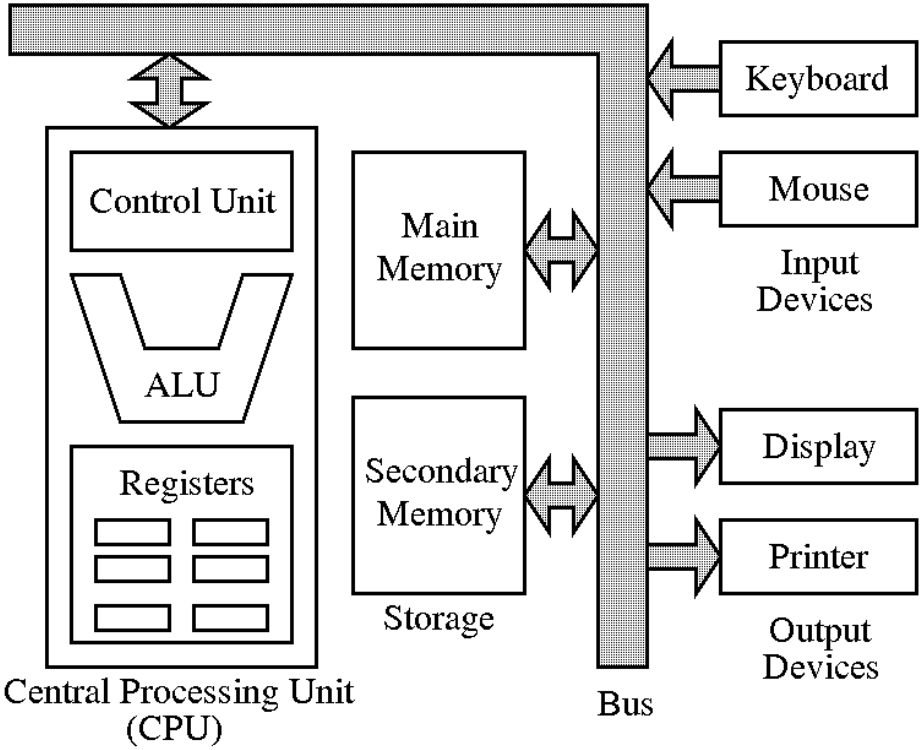
- 1.1. Disk Storage
- 1.2. File Operations
- 1.3. Unordered Files
- 1.4. Ordered Files
- 1.5. Hash Files
- 1.6. Other File Structures

1.1. Disk Storage

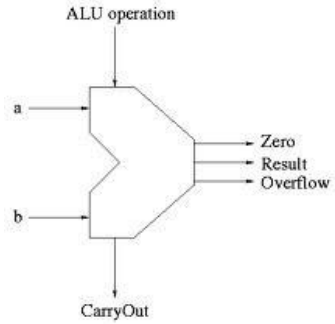
- Databases
  - A collection of data and their relationships
  - Computerized
  - Stored physically on computer storage media
    - Primary storage
    - Secondary storage
    - Tertiary storage (*Third-level* storage)

→ The DBMS software can then retrieve, update, and process the data as needed.

Computer Organization - Hardware



Computer Architecture



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

1.1. Disk Storage

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

Primary storage level

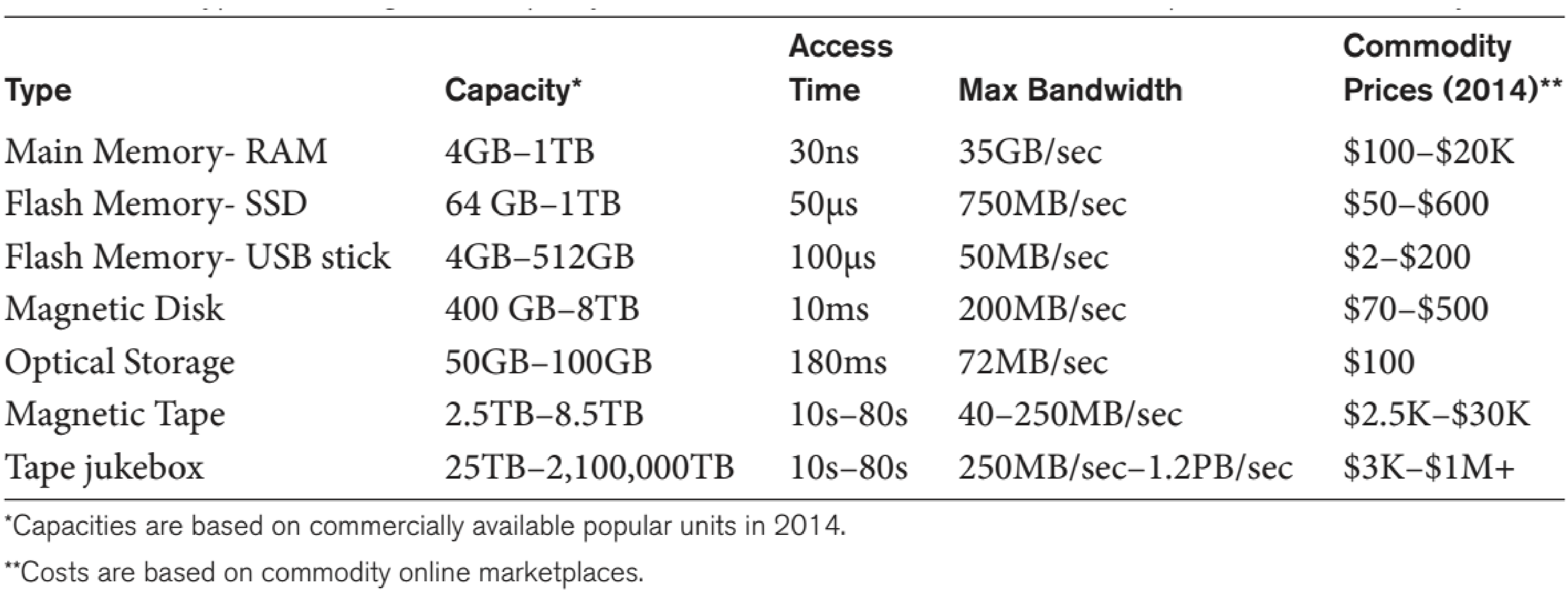
- Register
- Cache (static RAM)
- DRAM (dynamic RAM)

Secondary and tertiary storage level

- Magnetic disk
- Mass storage (CD-ROM, DVD)
- Tape

1.1. Disk Storage

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



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

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

**Costs are based on commodity online marketplaces.

Table 16.1, pp. 545

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

1.1. Disk Storage

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

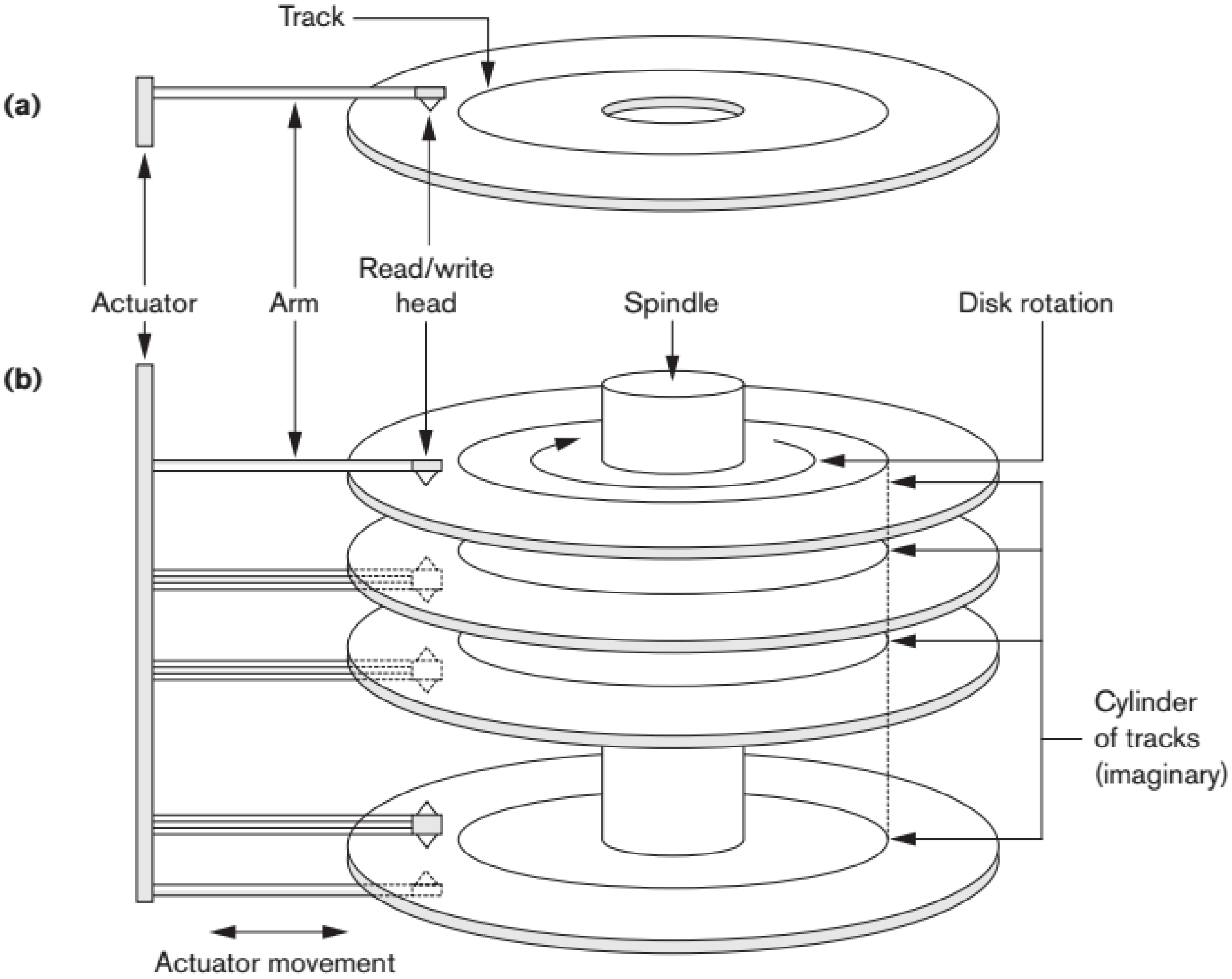
  → ***Persistent data*** (not *transient data* which persists for only a limited time during program execution)

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

1.1. Disk Storage

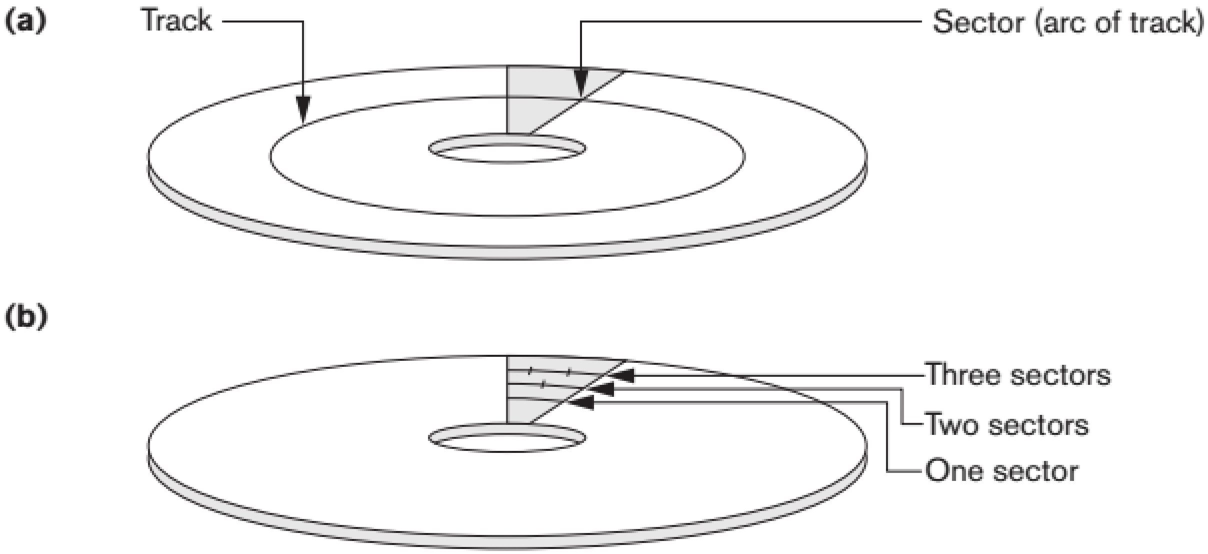
- Magnetic disks
  - Disks are covered with **magnetic** material.
  - The most basic unit of data on the disk is a single **bit** of information.
  - By magnetizing an area on a disk in certain ways, one can make that area represent a bit value of either 0 (zero) or 1 (one).
  - To code information,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

1.1. Disk Storage



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

1.1. Disk Storage



Different sector organizations on disk.

(a) Sectors subtending a fixed angle.

(b) Sectors maintaining a uniform recording density.

Figure 16.2, pp. 548, [1]

1.1. Disk Storage

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

1.1. Disk Storage

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

1.1. Disk Storage

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

1.1. Disk Storage

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

1.1. Disk Storage

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

1.1. Disk Storage

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

1.1. Disk Storage

- Disk parameters

Block size:	B bytes
Interblock gap size:	G bytes
Disk speed:	p rpm *(revolutions per minute)*
Seek time:	s msec
Rotational delay:	rd msec
Block transfer time:	btt msec
Rewrite time:	T_{rw} msec
Transfer rate:	tr bytes/msec
Bulk transfer rate:	btr bytes/msec

1.1. Disk Storage

- Disk parameters
  - *Rotational delay*: waiting time for the beginning of the required block to rotate into position under the read/write head once the read/write head is at the correct track

$$
\begin{aligned}
rd &= (1/2)*(1/p)\text{ min} \\
&= (60*1{,}000)*(1/2)*(1/p)\text{ msec} \\
&= 30{,}000/p\text{ msec}
\end{aligned}
$$

1.1. Disk Storage

- Disk parameters
  - *Block transfer time*: time to transfer the data in the block once the read/write head is at the beginning of the required block

    $btt = B/tr$ msec

  - If only *useful* bytes are considered, *block transfer time* is estimated with bulk transfer rate.

    $btt = B/btr$ msec

1.1. Disk Storage

- Disk parameters
  - *Rewrite time*: time for one disk revolution. This is useful in cases when we read a block from the disk into a main memory buffer, update the buffer, and then write the buffer back to the same disk block on which it was stored. In many cases, the time required to update the buffer in main memory is less than the time required for one disk revolution. If we know that the buffer is ready for rewriting, the system can keep the disk heads on the same track, and during the next disk revolution the updated buffer is rewritten back to the disk block.

T_{rw} = 2*rd msec = 60,000/p msec

1.1. Disk Storage

- Disk parameters
  - *Transfer rate*: the number of data bytes transferred in a time unit (msec)

$$tr = \frac{track\ size\ in\ byte}{time\ for\ one\ disk\ revolution\ in\ msec} \quad bytes/msec$$

1.1. Disk Storage

- Disk parameters
  - *Bulk transfer rate*: the rate of transferring *useful bytes* in the data blocks

$$btr = (B/(B+G))*tr \text{ bytes/msec}$$

1.1. Disk Storage

- The average time needed to find and transfer *one* block, given its address, is estimated by: (s + rd + btt) msec
- The average time needed to find and transfer any k blocks, given the address of each block, is: k*(s + rd + btt) msec
- The average time needed to find and transfer consecutively k *noncontiguous* blocks on the *same cylinder*, given the address of each block, is: (s + k*(rd + btt)) msec
- The average time needed to find and transfer consecutively k *contiguous* blocks on the *same track or cylinder*, given the address of the first block, is: (s + rd + k*btt) msec
- The estimated time to read *k contiguous* blocks consecutively stored on the *same cylinder*, when the *bulk transfer rate* is used to transfer the *useful* data, is: (s + rd + k*(B/btr)) msec

1.1. Disk Storage

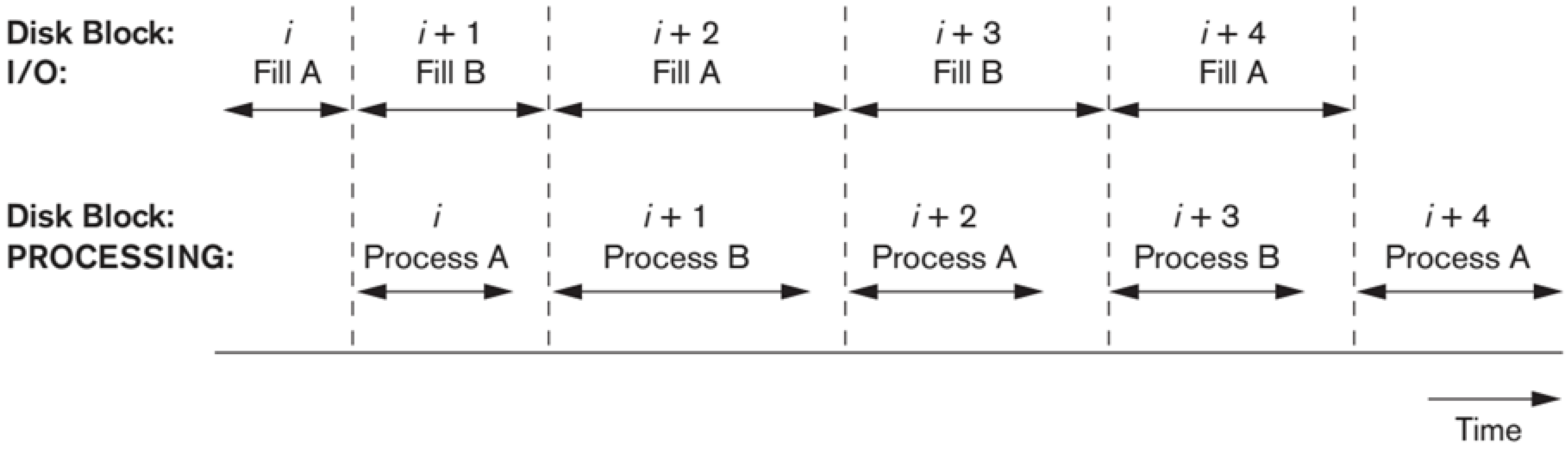
- Making data access more efficient on disk
  - Buffering of data
  - Proper organization of data on disk
  - Reading data ahead of request
  - Proper scheduling of I/O requests
  - Use of log disks to temporarily hold writes
  - Use of flash memory for recovery purposes

1.1. Disk storage

- Making data access more efficient on disk
  - Buffering of data
    - **Buffer**: a part of main memory that is available to receive blocks or pages of data from disk
    - **Buffer manager**: a software component of a DBMS that responds to requests for data and decides what buffer to use and what pages to replace in the buffer to accommodate the newly requested blocks
      - (1) to maximize the probability that the requested page is found in main memory
      - (2) in case of reading a new disk block from disk, to find a page to replace that will cause the least harm in the sense that it will not be required shortly again
    - **Double buffering**: a technique for more efficiency

1.1. Disk Storage

- Making data access more efficient on disk
  - **Double buffering**: a technique for more efficiency, using *two buffers*



Double buffering: use of two buffers, A and B, for reading from disk

Figure 16.4, pp. 557, [1]

1.1. Disk Storage

- Making data access more efficient on disk
  - **Double buffering**: a technique for more efficiency, using *two buffers*
    - performing the I/O operation between the disk and main memory into one buffer area
    - concurrently processing the data from another buffer
    - → continuous reading or writing of data on consecutive disk blocks, which eliminates the seek time and rotational delay for all but the first block transfer
    - → data is kept ready for processing, thus reducing the waiting time in the programs.

1.1. Disk storage

- Making data access more efficient on disk
  - Buffering of data
    - To enable its operation, the buffer manager keeps two types of information on hand about each block (page)
      - pin-count: the number of used times
      - dirty-bit: 1 if updated; otherwise, 0
  - Buffer replacement strategies
    - Least recently used (LRU)
    - Clock policy: a round-robin variant of the LRU policy
    - First-in-first-out (FIFO)
    - Most recently used (MRU)

1.2. File Operations

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

1.2. File Operations

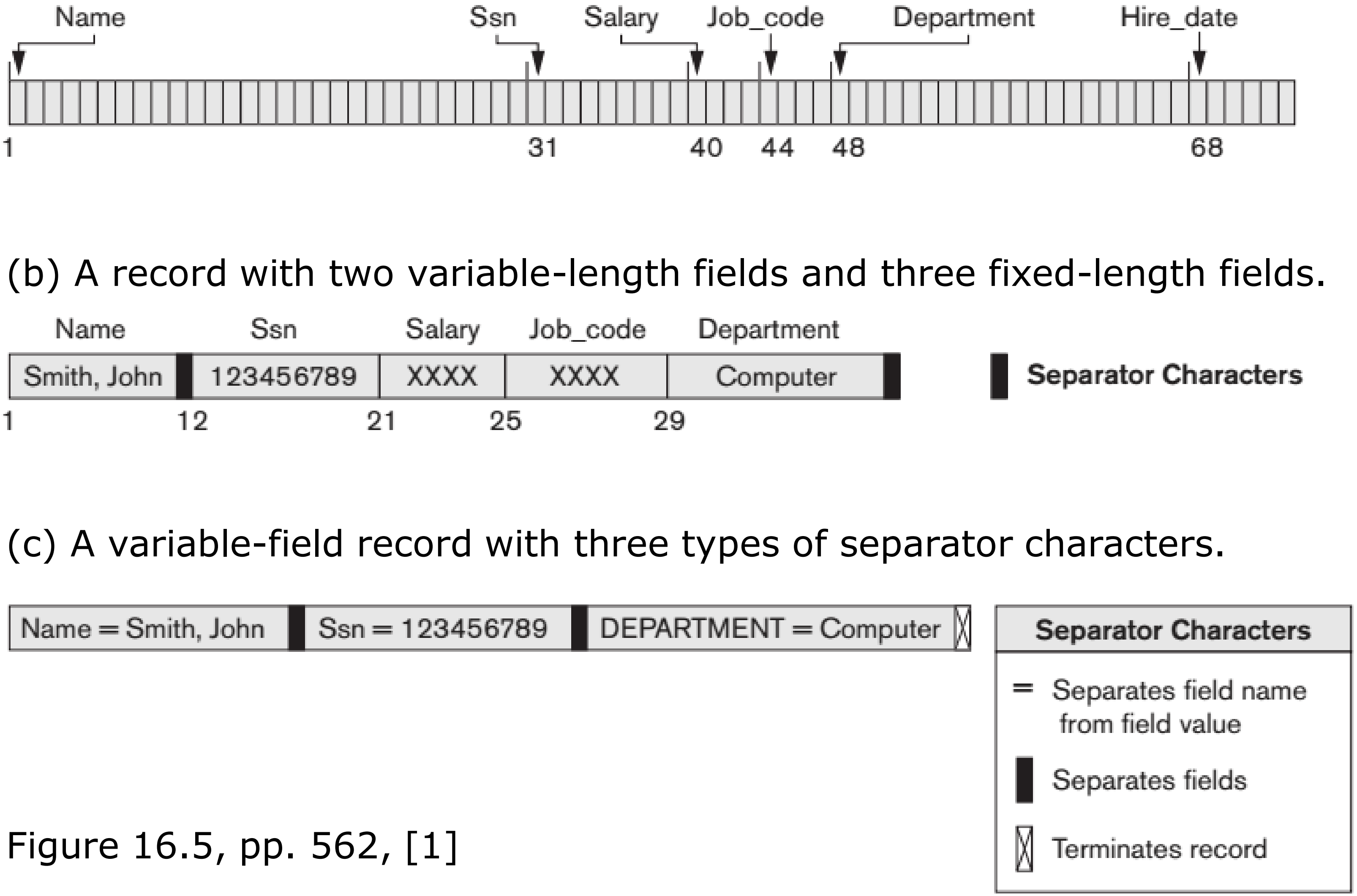
- Fixed-length records
    - The same record type, the same size
- Variable-length records
    - The same type, variable-length field(s)
        - Special **separator** characters (such as ? or % or $)—which do not appear in any field value—to terminate variable-length fields
    - The same type, repeating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

1.2. File Operations

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

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

1.2. File Operations



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

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

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

Figure 16.5, pp. 562, [1]

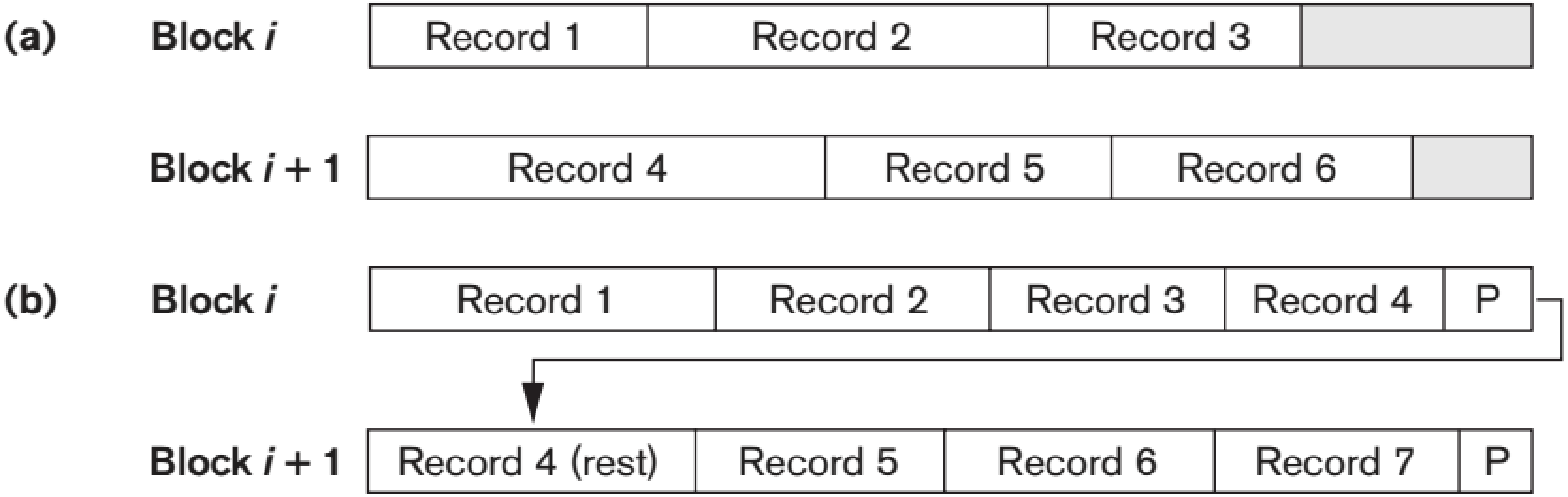
1.2. File Operations

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

→ How many records can be stored in a block?

→ Blocking factor

1.2. File Operations



Types of record organization.

(a) Unspanned.

(b) Spanned.

Figure 16.6, pp. 563, [1]

1.2. File Operations

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

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

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

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

→ use *bfr* to calculate the number of blocks *b* needed for a file of *r* records

$$b = \left\lceil \frac{r}{bfr} \right\rceil \text{blocks}$$

1.2. File Operations

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

1.2. File Operations

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

1.2. File Operations

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

1.3. Unordered Files

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

1.3. Unordered Files

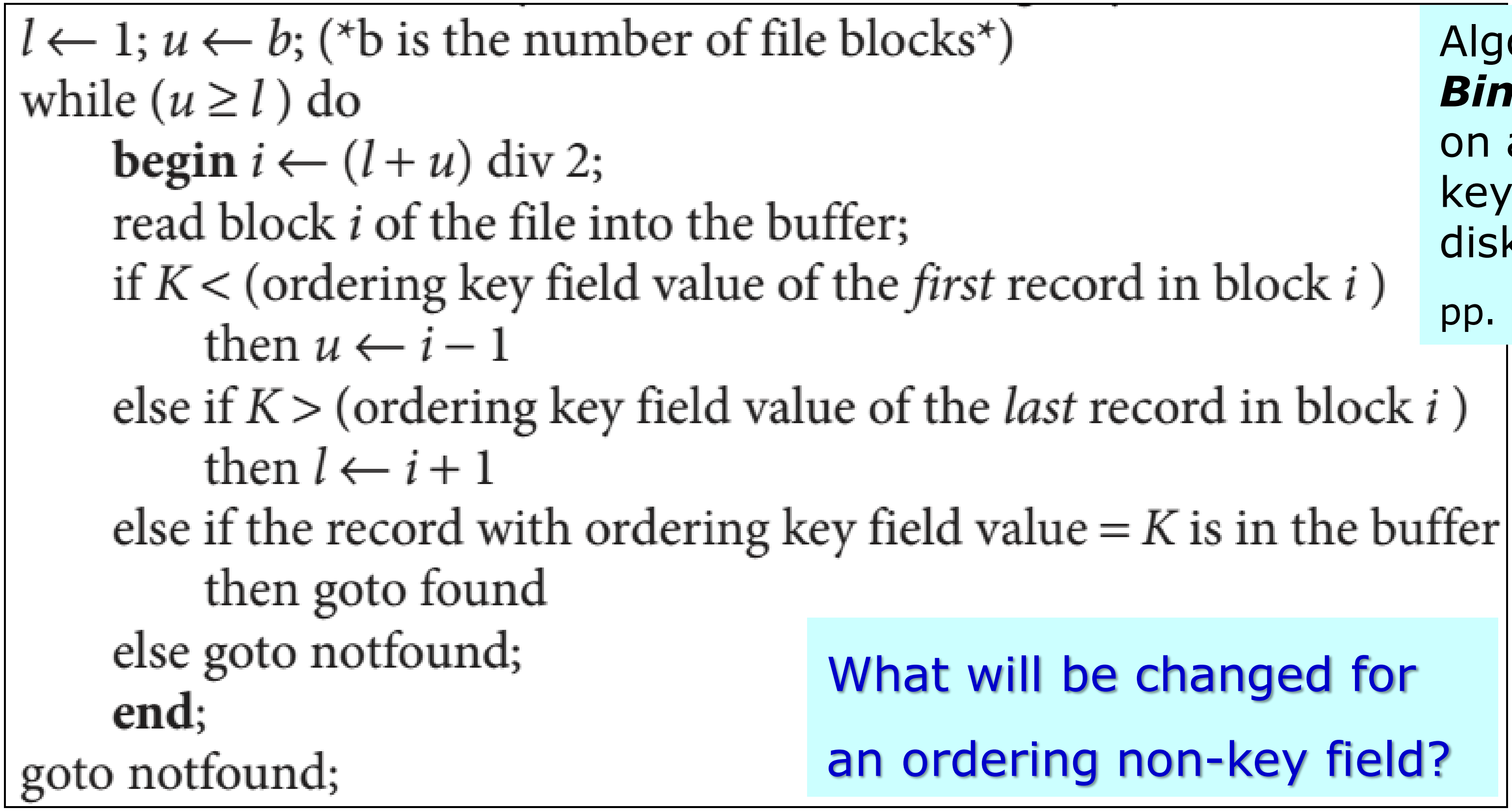
- Unordered files = Heap files = Pile files
  - ***Inserting*** a new record is *very efficient.*
    - The last disk block of the file is copied into a buffer, the new record is added, and the block is then rewritten back to disk.
  - To ***delete*** a record, a program must first find its block, copy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

1.4. Ordered Files

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

1.4. Ordered Files

- Ordered files = Sorted files = Sequential files
  - Searching for a record whose ordering key value is K, a ***binary search*** can be done on the blocks rather than on the records. => $\log_2(b)$ block accesses with b file blocks



```
$l \leftarrow 1$; $u \leftarrow b$; (*b is the number of file blocks*)
while ($u \geq l$) do
    begin $i \leftarrow (l + u)$ div 2;
    read block $i$ of the file into the buffer;
    if $K <$ (ordering key field value of the first record in block $i$)
        then $u \leftarrow i - 1$
    else if $K >$ (ordering key field value of the last record in block $i$)
        then $l \leftarrow i + 1$
    else if the record with ordering key field value $= K$ is in the buffer
        then goto found
    else goto notfound;
    end;
goto notfound;
```

Algorithm 16.1. ***Binary search*** on an ordering key field of a disk file

pp. 570, [1]

What will be changed for an ordering non-key field?

1.4. Ordered Files

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

1.4. Ordered Files

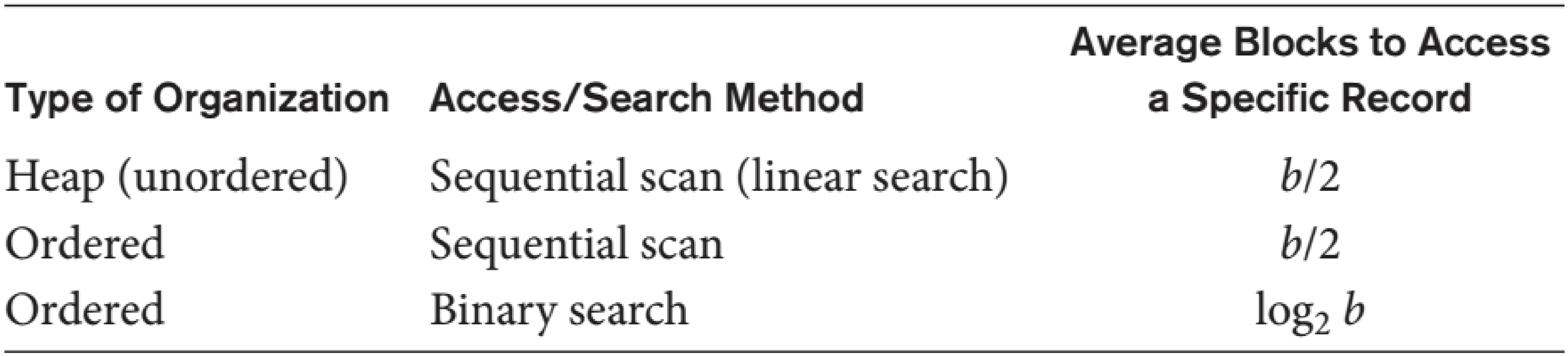
- Ordered files = Sorted files = Sequential files
  - Inserting and deleting records are expensive because the records must remain physically ordered.
  - One frequently used ***insertion*** method
    - The actual ordered file (called *main* or *master* file) for binary search on ordering field values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

1.4. Ordered Files

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

1.4. Ordered Files

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	$\log_2 b$

Table 16.3, pp. 572, [1]

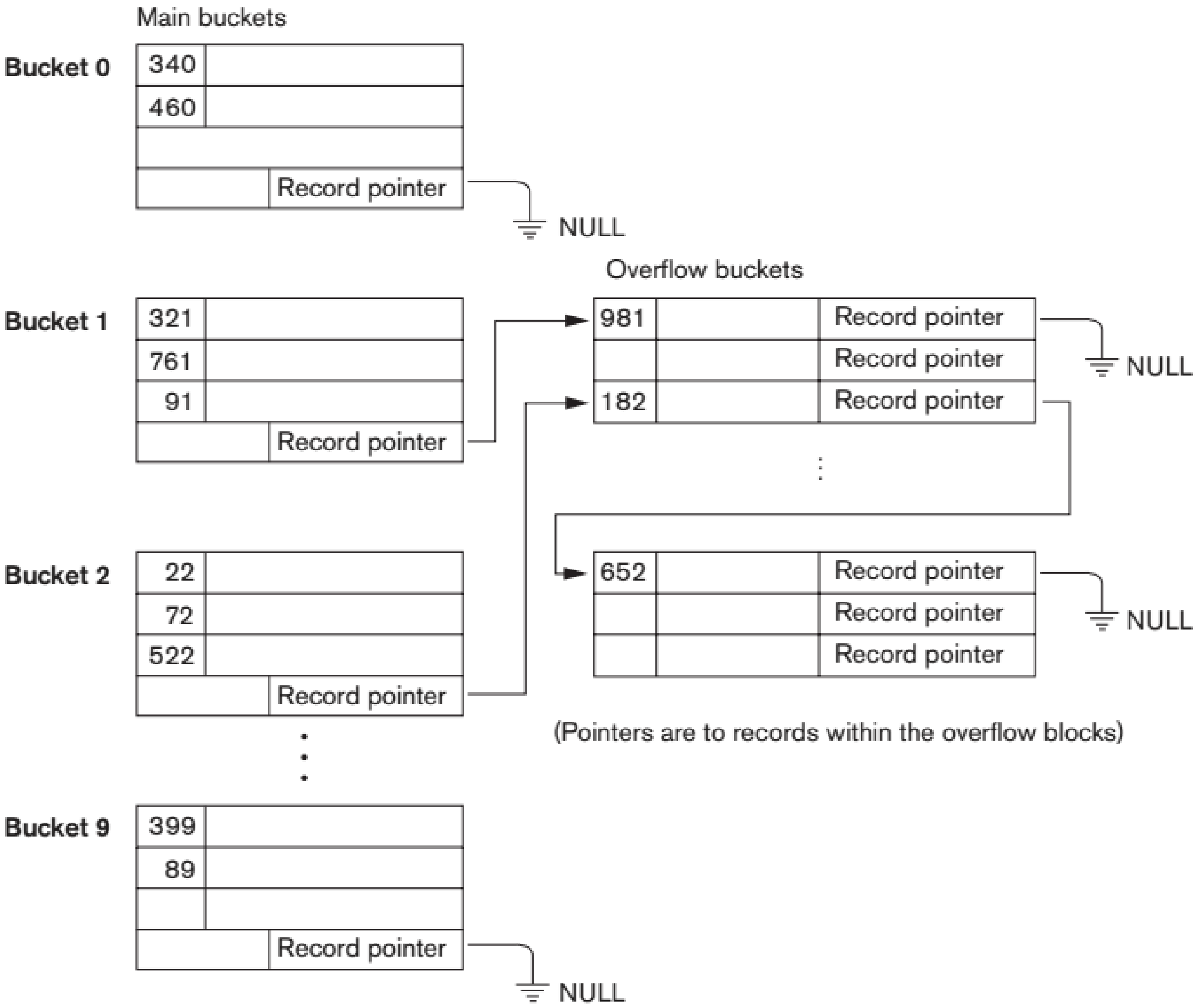
1.5. Hash Files

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

1.5. Hash Files

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

1.5. Hash Files



Handling overflow for buckets by chaining

Figure 16.10, pp. 576, [1]

Address space:

M buckets (=10)

Hash function:

$h(K) = K \bmod M$

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

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

1.5. Hash Files

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

1.5. Hash Files

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

1.5. Hash Files

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

1.5. Hash Files

- The hashing scheme with a fixed number of allocated buckets *M* is called **static hashing**.
  - Not suitable for dynamic files
  - *Solution (?)*: change the number of buckets *M* (smaller, larger) and redistribute the records with a new hash function based on the new value of *M*

→ Newer dynamic file organizations based on hashing allow the number of buckets to vary dynamically with only localized reorganization.

  → Extendible hashing: a directory (access structure)

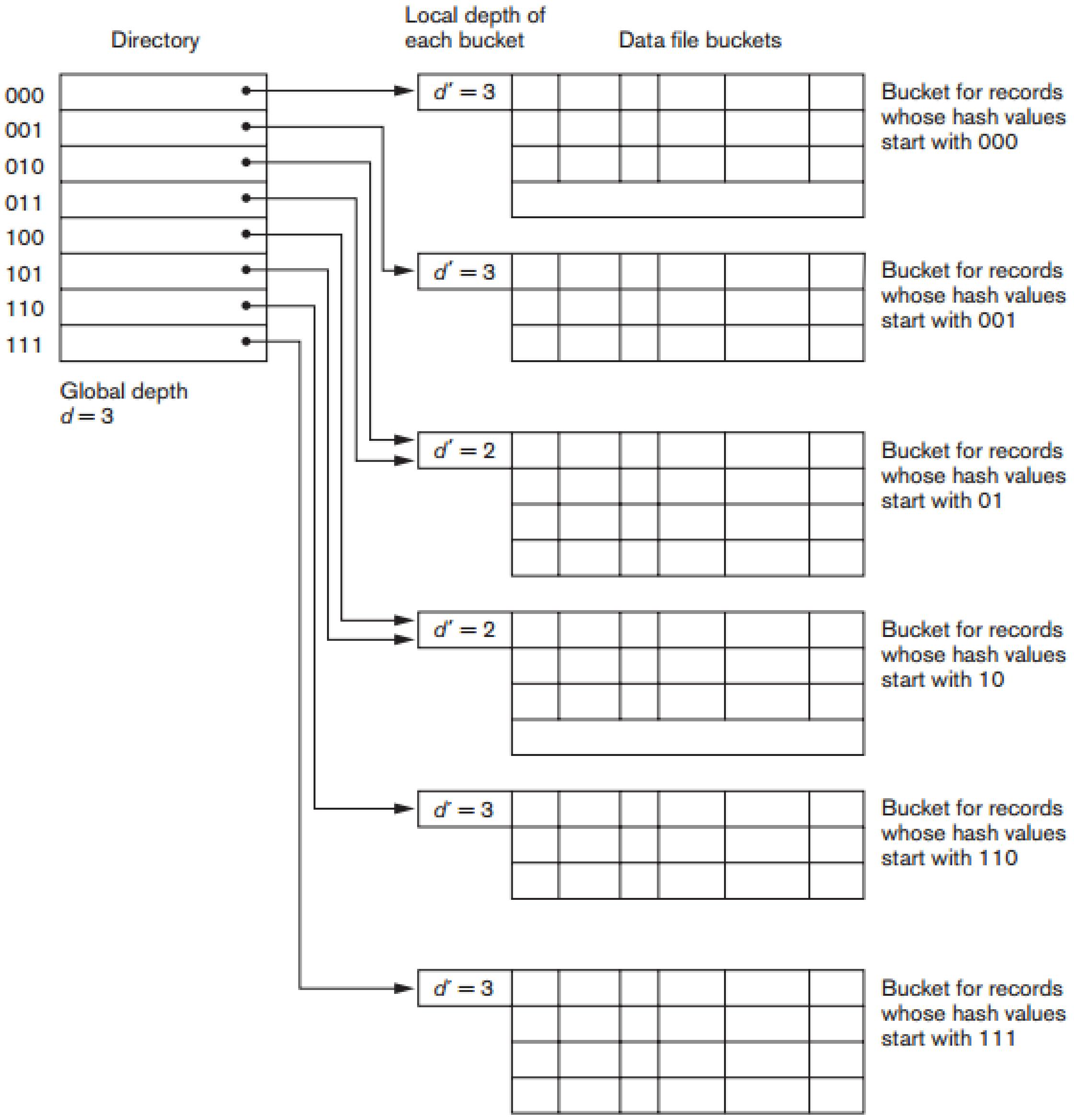
  → Dynamic hashing: a binary tree (access structure)

  → Linear hashing: a sequence of hash functions

1.5. Hash Files

- Extendible hashing
  - The access structure, *Directory*, is built on the **binary representation** of the hash function result, which is a string of **bits**, called the **hash value** of a record.
  - *Directory* contains the pointers to the buckets.
  - Records are distributed among buckets based on the values of the *leading bits* in their hash values.
  - The number of the leading bits used in record distribution is the global depth, *d*, and dynamically determines the maximum number 2^d of buckets.
  - *Two block accesses* for equality search

1.5. Hash Files



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

- Extendible hashing
  - Global depth, d: the number of the leading bits used for distributing all the records in at most 2^d buckets
  - Local depth of each bucket, d': the number of the leading bits used for reflecting the bucket contents

Structure of the extendible hashing scheme. Figure 16.11, pp. 579, [1]

1.5. Hash Files

K: hash field

h(*K*): hash function

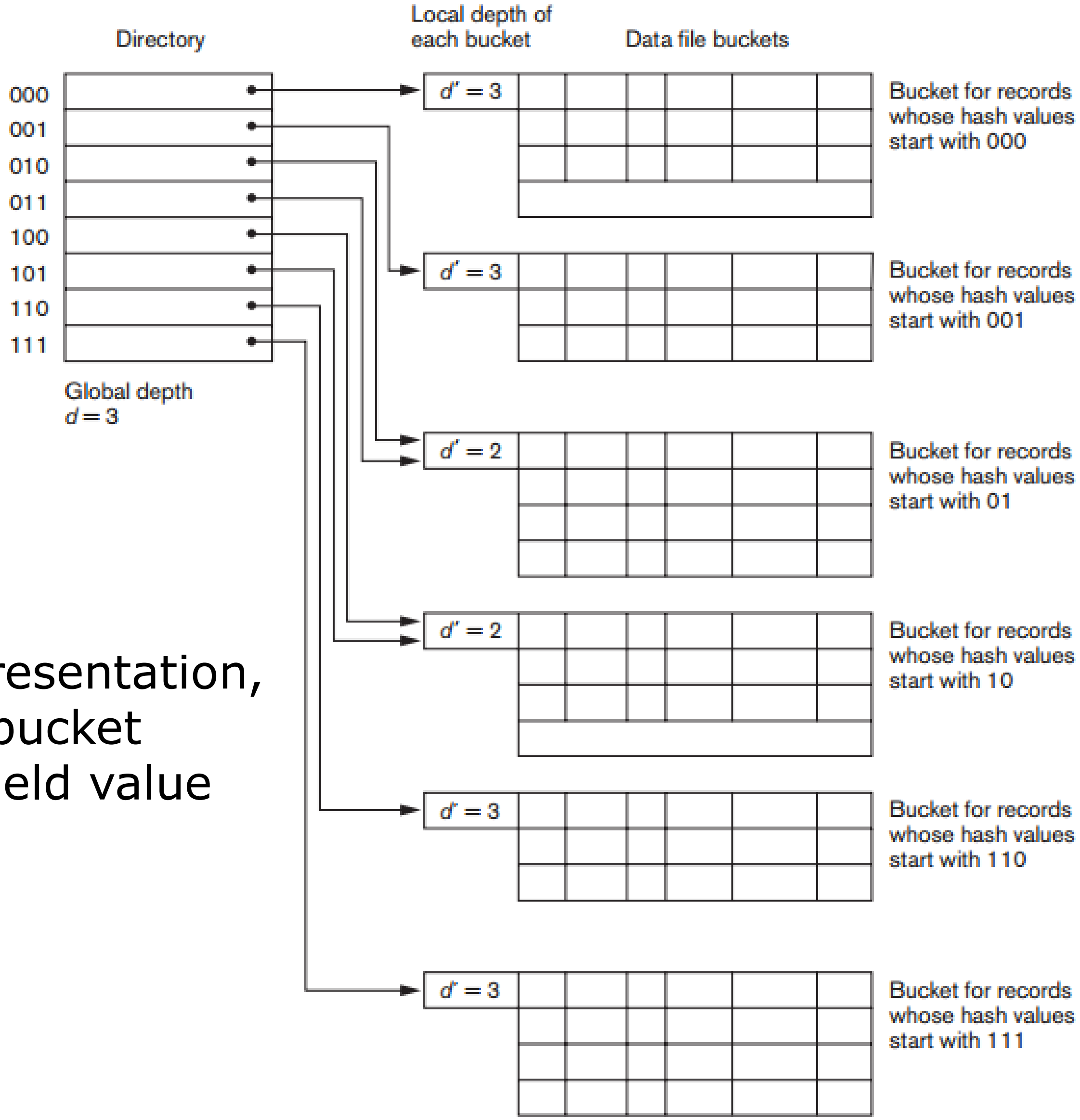
d: global depth

d': local depth

a = h(*v*) for *K*=*v*

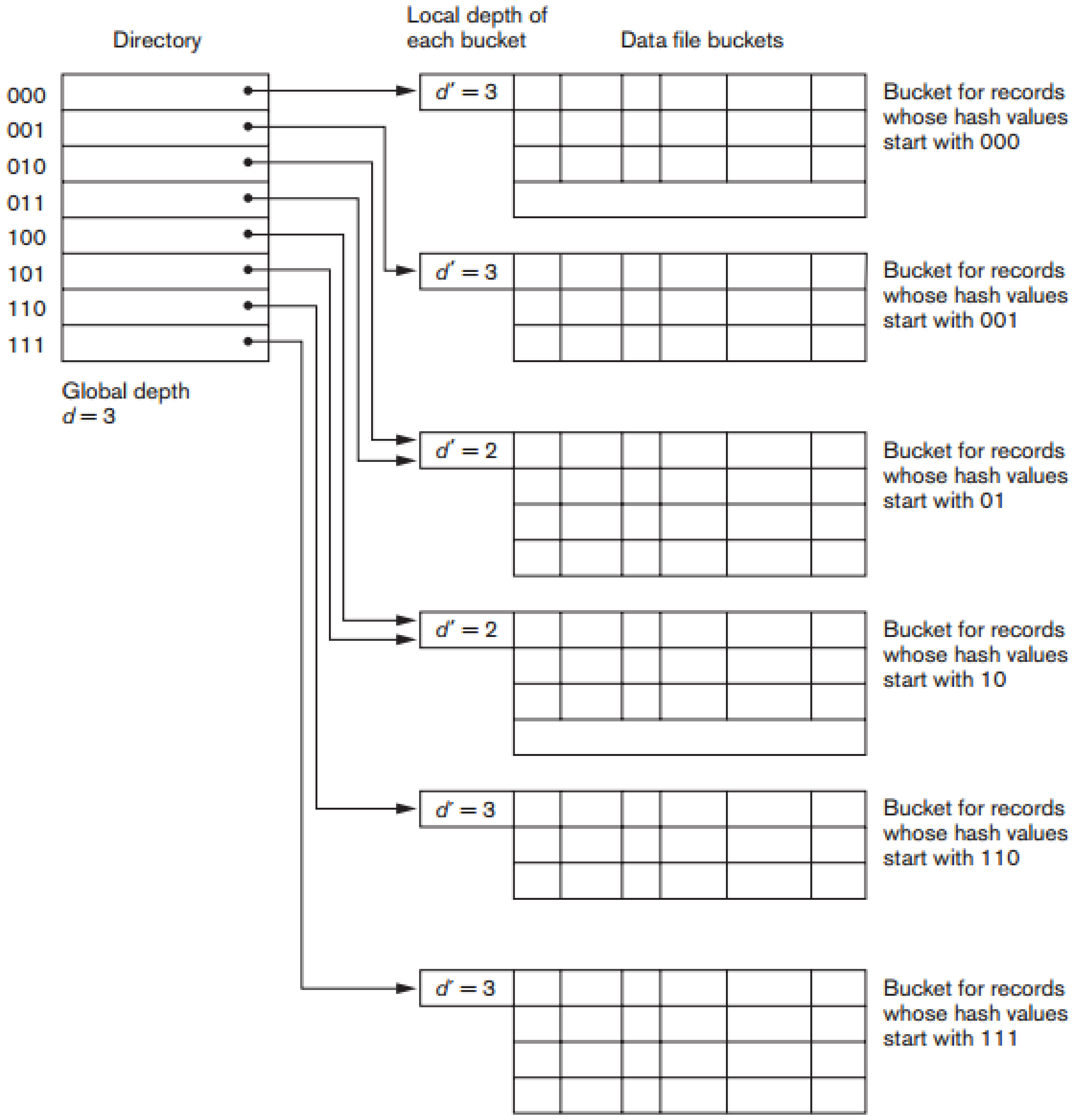
a: a value in a binary representation, showing the index of the bucket where a record whose *K* field value is *v* is going to be stored

NO overflow area!


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

1.5. Hash Files

- Extendible hashing
  - The value of *d* can be increased or decreased by one at a time: *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.
  - ***Halving*** occurs if *d* > *d′* for all the buckets after some deletions occur.



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

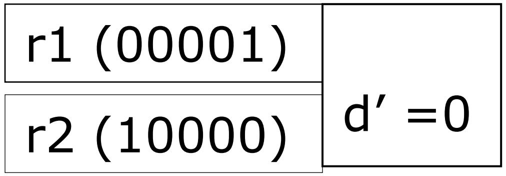
1.5. Hash Files

- Extendible hashing - Example

Record	K	h(K) = K mod 32	h(K)B
r1	2657	1	00001
r2	3760	16	10000
r3	4692	20	10100
r4	4871	7	00111
r5	5659	27	11011
r6	1821	29	11101
r7	1074	18	10010
r8	2123	11	01011
r9	1620	20	10100
r10	2428	28	11100
r11	3943	7	00111
r12	4750	14	01110
r13	6975	31	11111
r14	4983	23	10111
r15	9208	24	11000

Each bucket can store two records.

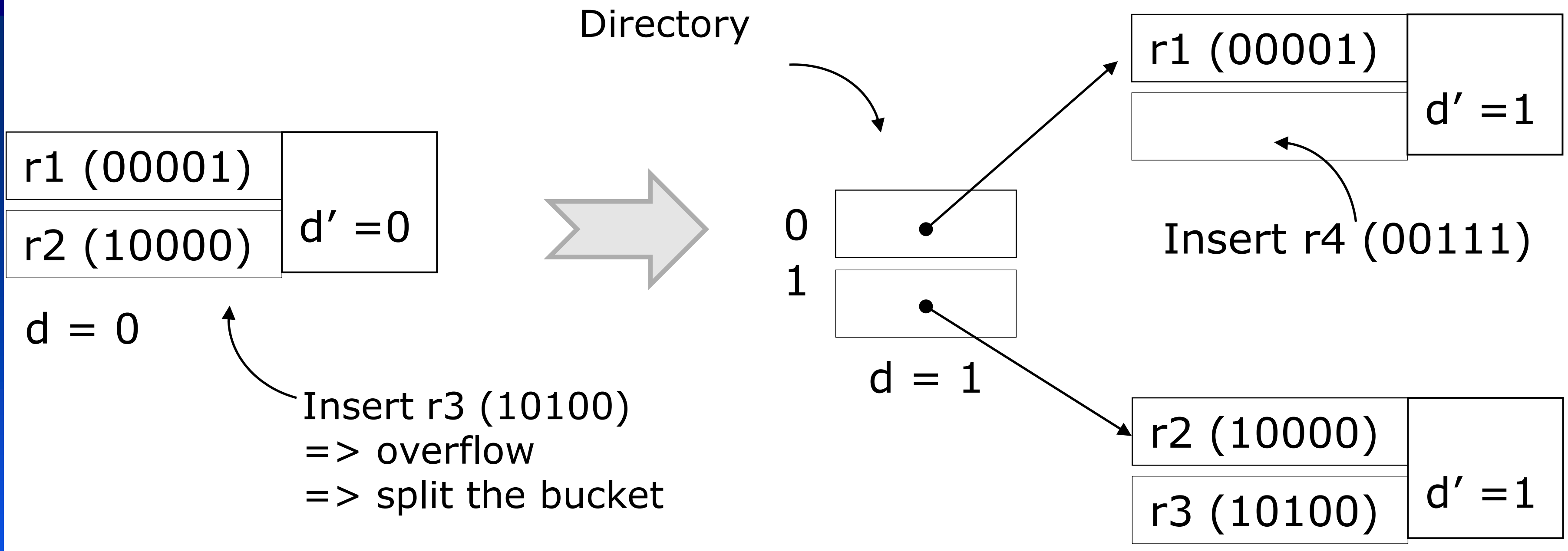
d′ = local depth
d = global depth



d = 0

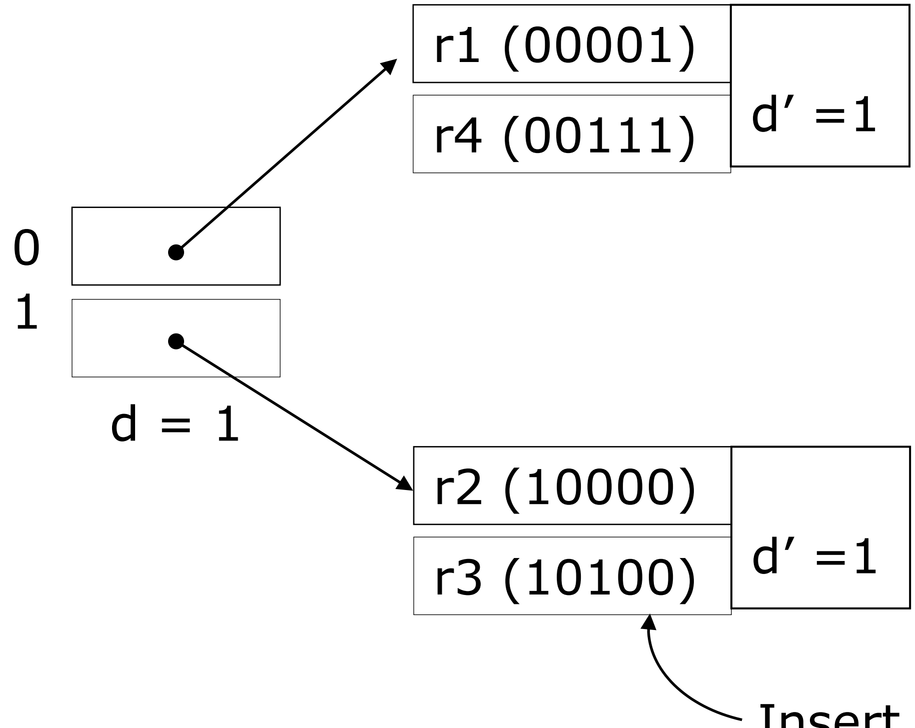
1.5. Hash Files

- Extendible hashing - Example



1.5. Hash Files

- Extendible hashing - Example



r1 (00001)

r4 (00111)

d’ =1

0

1

d = 1

r2 (10000)

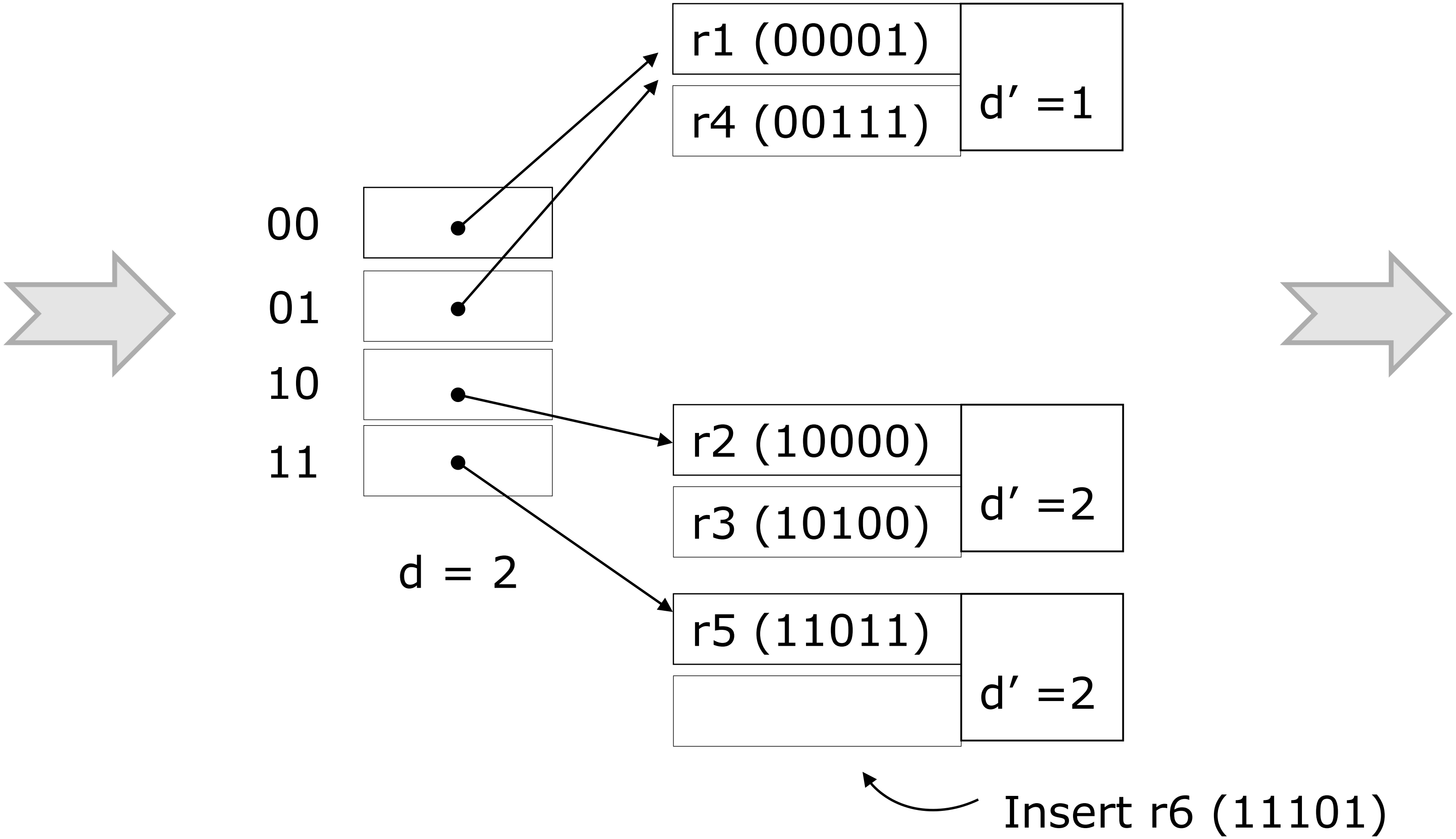
r3 (10100)

d’ =1

Insert r5 (11011)
⇒ Overflow
⇒ split the bucket
⇒ double the directory

1.5. Hash Files

- Extendible hashing - Example



r1 (00001)

r4 (00111)

d′ =1

00

01

10

11

d = 2

r2 (10000)

r3 (10100)

d′ =2

r5 (11011)

d′ =2

Insert r6 (11101)

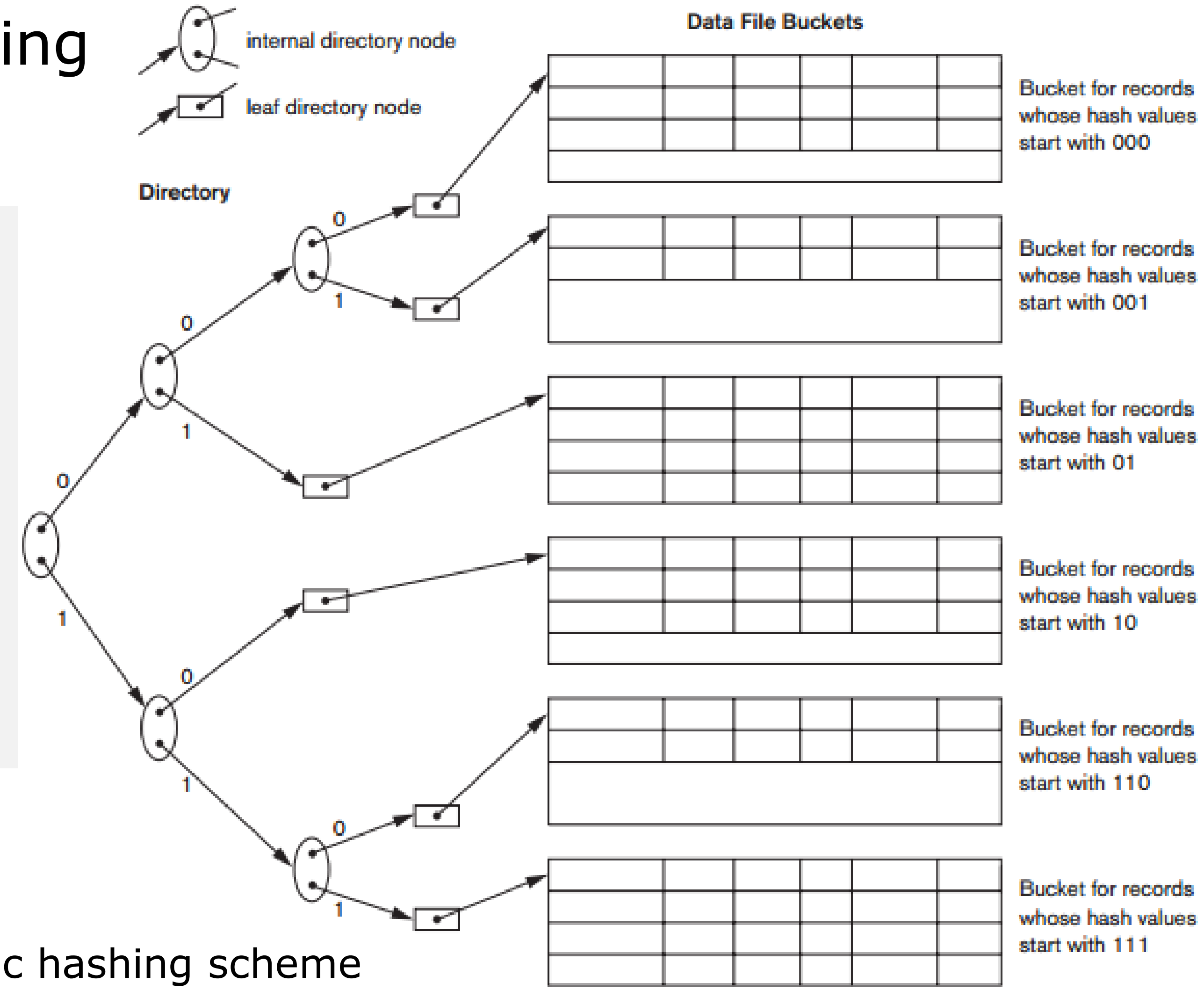
1.5. Hash Files

- Dynamic hashing
  - The storage of records in buckets for dynamic hashing is similar to extendible hashing.
  - The major difference is in the organization of the directory.
    - Extendible hashing uses the global depth (high-order *d* bits) for the flat directory and then combines adjacent collapsible buckets into a bucket of local depth *d*-1.
    - 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 the right pointer corresponding to the 1 bit.
      - Leaf nodes hold a pointer to the bucket with records.

1.5. Hash Files

- Dynamic hashing

How many block accesses are required for an *equality* search with this dynamic hashing scheme?


internal directory node
leaf directory node
Directory
Data File Buckets
0
1
0
1
0
1
0
1
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

Structure of the dynamic hashing scheme

Figure 16.12, pp. 581, [1]

1.5. Hash Files

- Dynamic hashing - Example

Record	K	h(K) = K mod 32	h(K)B
r1	2657	1	00001
r2	3760	16	10000
r3	4692	20	10100
r4	4871	7	00111
r5	5659	27	11011
r6	1821	29	11101
r7	1074	18	10010
r8	2123	11	01011
r9	1620	20	10100
r10	2428	28	11100
r11	3943	7	00111
r12	4750	14	01110
r13	6975	31	11111
r14	4983	23	10111
r15	9208	24	11000

Each bucket can store two records.

How to store these records with this dynamic hashing scheme?

1.5. Hash Files

- Linear hashing
  - No access structure like *Directory* in Extendible Hashing or the *tree*-based structure in Dynamic Hashing
  - An overflow chain of each bucket is used for collision at that bucket.
  - The number of buckets is dynamically decided by splitting one bucket *j* into two buckets *j* and *M* + *j* in sequence from *j*=0 to *j*=*M*-1 where *M* is the initial number of buckets.
    - A sequence of hash functions: $h_0(K) = K \bmod M$; $h_1(K) = K \bmod 2M$; $h_2(K) = k \bmod 4M$; …; $h_i(K) = K \bmod 2^i M$
    - *n*: an integer value that determines which bucket is split

1.5. Hash Files

- Linear hashing
  - *n* = 0: a hash function $h_i(K) = K \bmod 2^i M$ is applied to distribute the records over all the buckets.
    - If any bucket is overflowed, the bucket 0 is split into 0 and M; and its content and the overflow area's content are re-distributed by the hash function $h_{i+1}(K) = K \bmod 2^{i+1}M$. After that, *n* is increased by 1.
  - *n* > 0: the buckets with $h_i(K) \geq n$ are examined with $h_i(K)$ while the others with $h_{i+1}(K)$.
    - If any bucket is overflowed, the bucket *n* is split into *n* and $M + n$; and its content and the overflow area's content are re-distributed by the hash function $h_{i+1}(K) = K \bmod 2^{i+1}M$. After that, *n* is also increased by 1.
  - If $n=M$, *n* is set to 0 and $h_{i+1}(K) = K \bmod 2^{i+1}M$ is used as $h_i(K)$ and *M* is updated to be $2^{i+1}M$.

1.5. Hash Files

- Linear hashing - Example

Record	K	h(K) = K mod 4	h(K) = K mod 8
r1	2657	1	1
r2	3760	0	0
r3	4692	0	4
r4	4871	3	7
r5	5659	3	3
r6	1821	1	5
r7	1074	2	2
r8	2123	3	3
r9	1620	0	4
r10	2428	0	4
r11	3943	3	3
r12	4750	2	6
r13	6975	3	7
r14	4983	3	7
r15	9208	0	0

Each bucket can store two records.

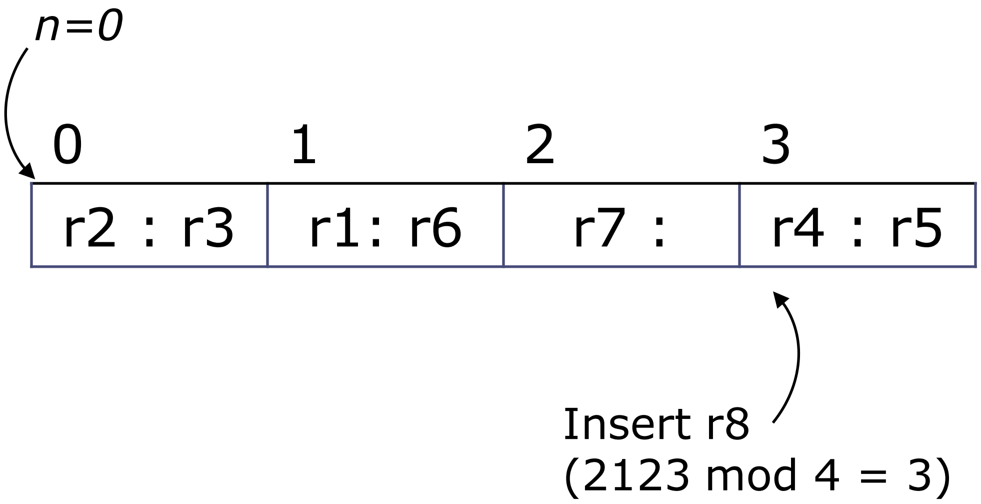
M = 4 for initialization

$h_0(K) = K \bmod M$

How to store these records with this linear hashing scheme?

1.5. Hash Files

- Linear hashing



n=0

0	1	2	3
r2 : r3	r1: r6	r7 :	r4 : r5

Insert r8
(2123 mod 4 = 3)

Overflow at bucket 3

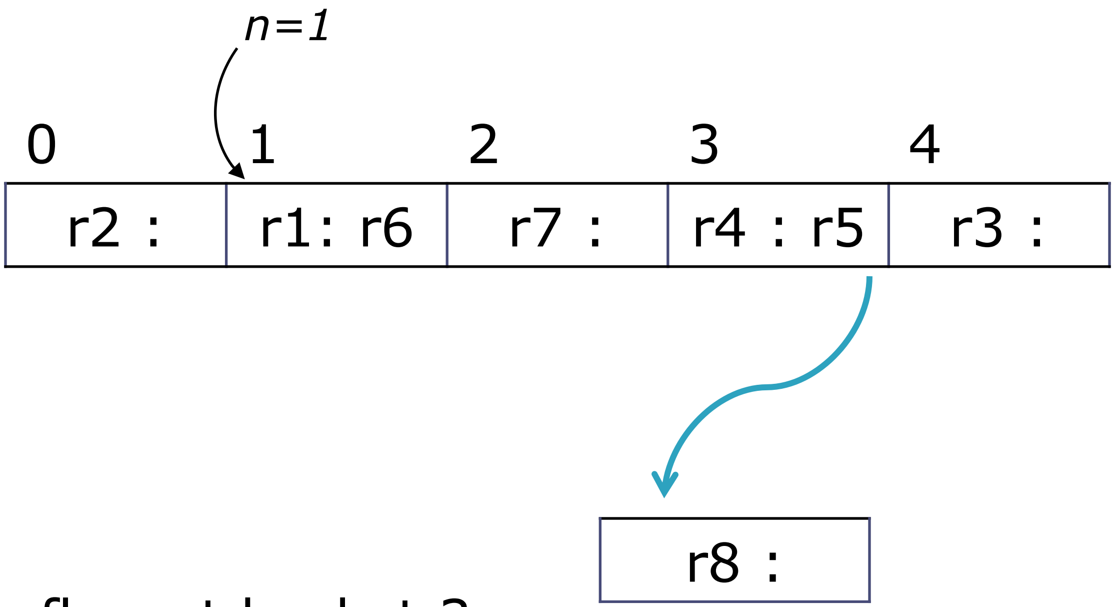
r8 is inserted in the overflow chain of bucket 3.

Split bucket 0 into bucket 0 and bucket 4

$h_1(K) = K \text{ mod } 2*M$

1.5. Hash Files

- Linear hashing



n=1

0	1	2	3	4
r2 :	r1: r6	r7 :	r4 : r5	r3 :

r8 :

Overflow at bucket 3

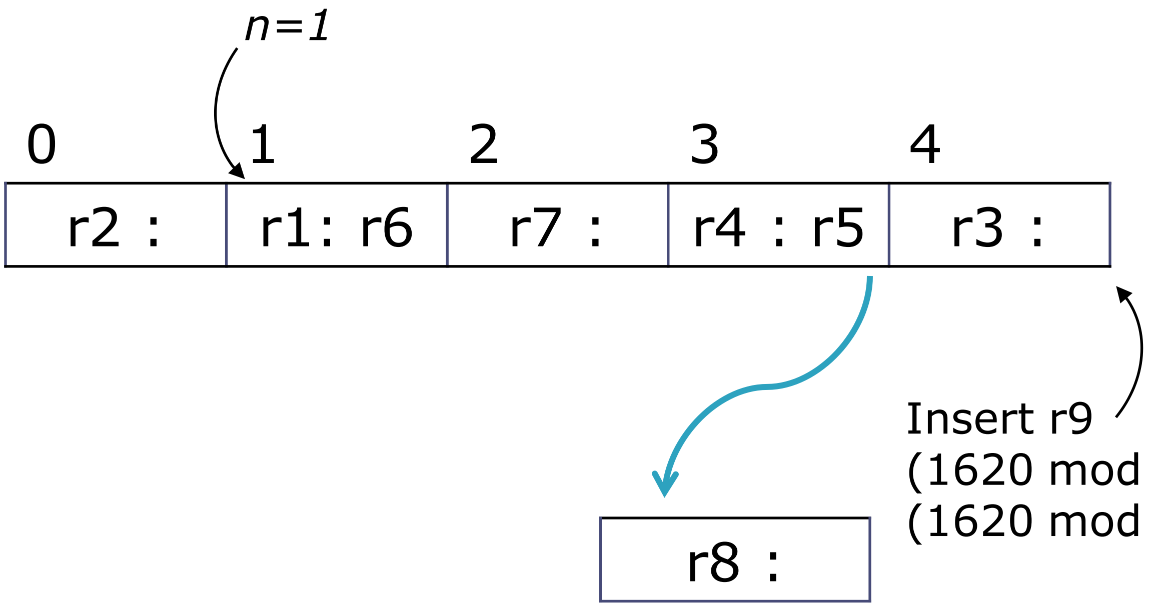
r8 is inserted in the overflow chain of bucket 3.

Split bucket 0 into bucket 0 and bucket 4

$h_1(K) = K \text{ mod } 2*M$

1.5. Hash Files

- Linear hashing



n=1

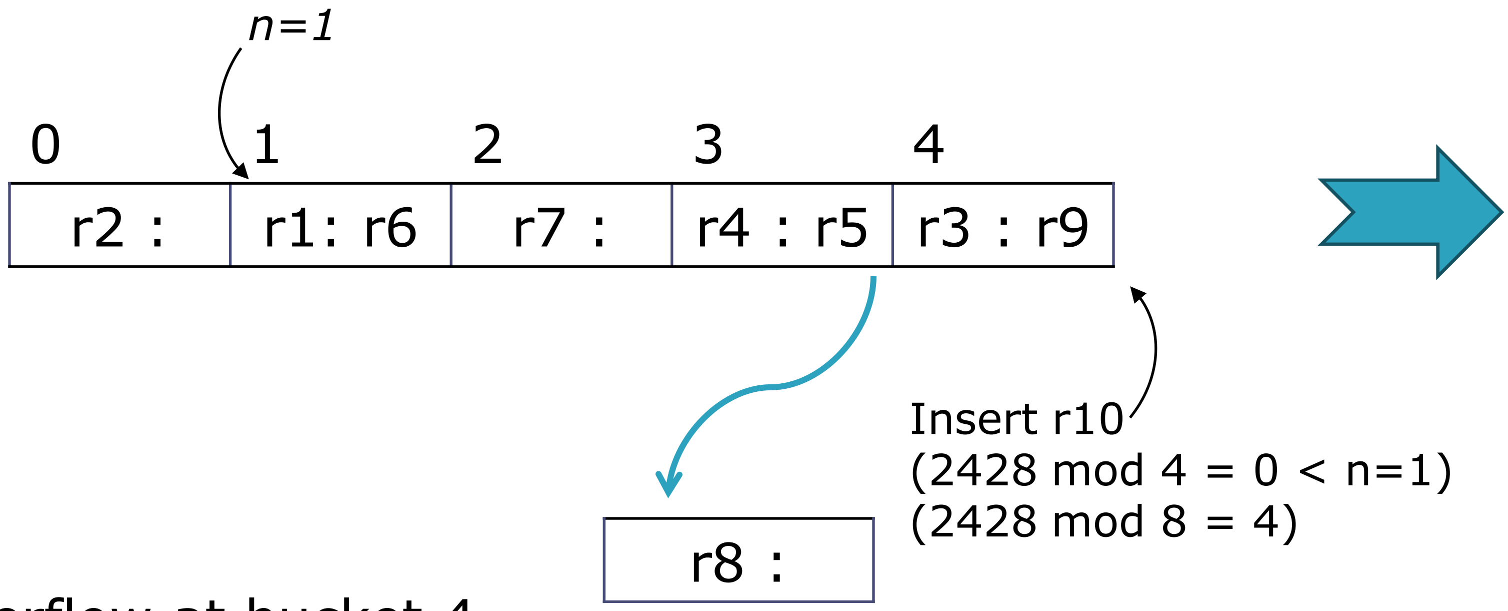
0	1	2	3	4
r2 :	r1: r6	r7 :	r4 : r5	r3 :

r8 :

Insert r9
(1620 mod 4 = 0 < n=1)
(1620 mod 8 = 4)

1.5. Hash Files

- Linear hashing



Insert r10
(2428 mod 4 = 0 < n=1)
(2428 mod 8 = 4)

Overflow at bucket 4

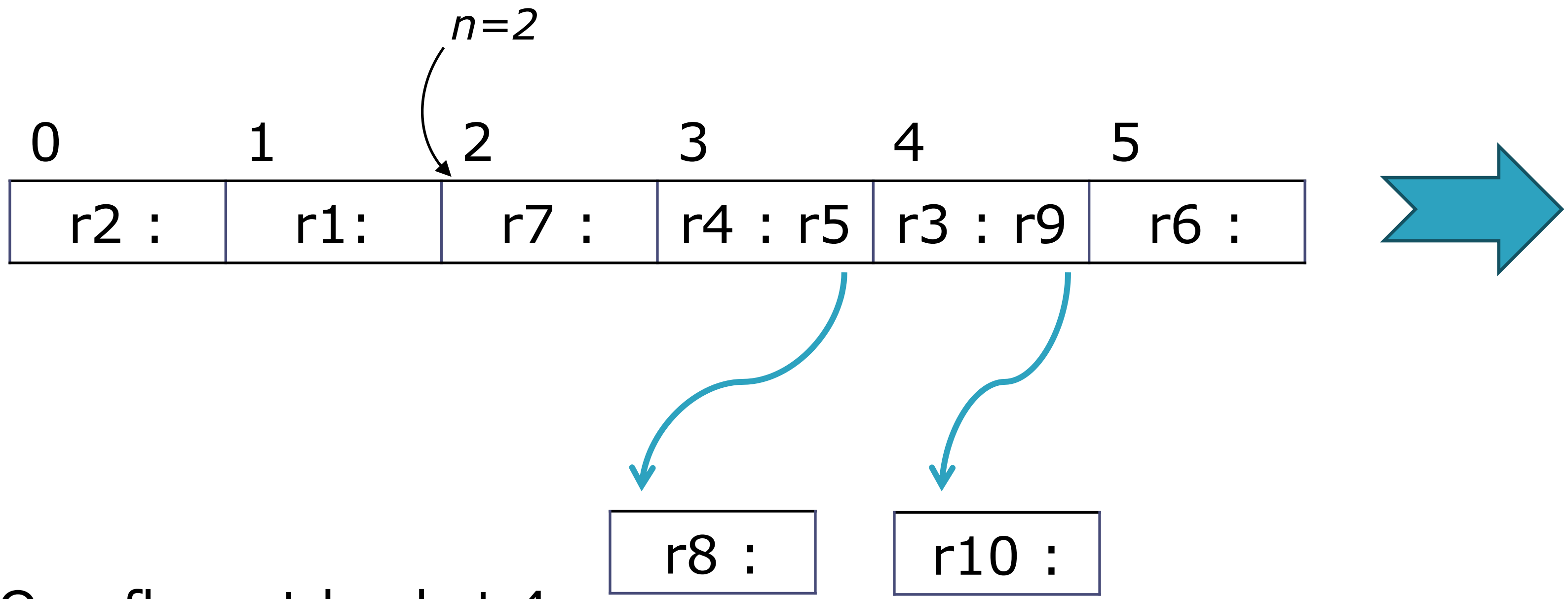
r10 is inserted in the overflow chain of bucket 4.

Split bucket 1 into bucket 1 and bucket 5

$h_1(K) = K \bmod 2*M$

1.5. Hash Files

- Linear hashing



n=2

0	1	2	3	4	5
r2 :	r1:	r7 :	r4 : r5	r3 : r9	r6 :

r8 :

r10 :

Overflow at bucket 4

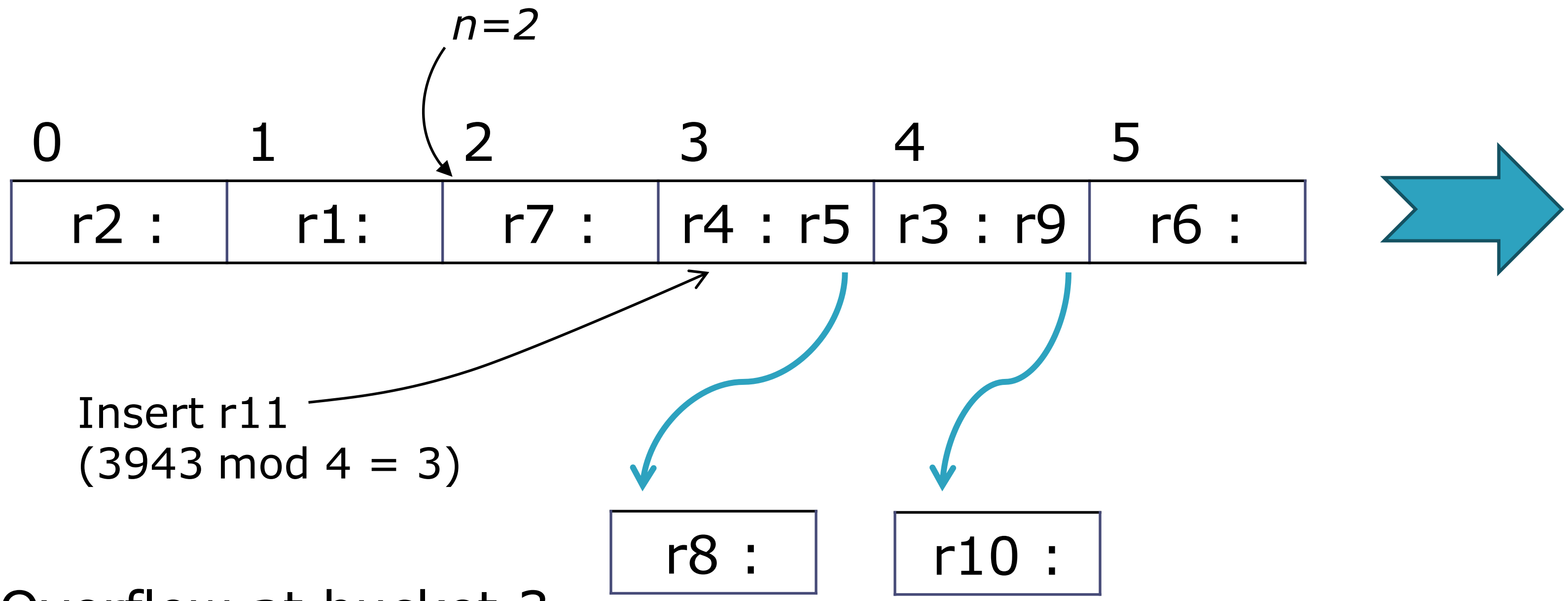
r10 is inserted in the overflow chain of bucket 4.

Split bucket 1 into bucket 1 and bucket 5

$h_1(K) = K \text{ mod } 2*M$

1.5. Hash Files

- Linear hashing



n=2

0	1	2	3	4	5
r2 :	r1:	r7 :	r4 : r5	r3 : r9	r6 :

r8 :

r10 :

Insert r11
(3943 mod 4 = 3)

Overflow at bucket 3

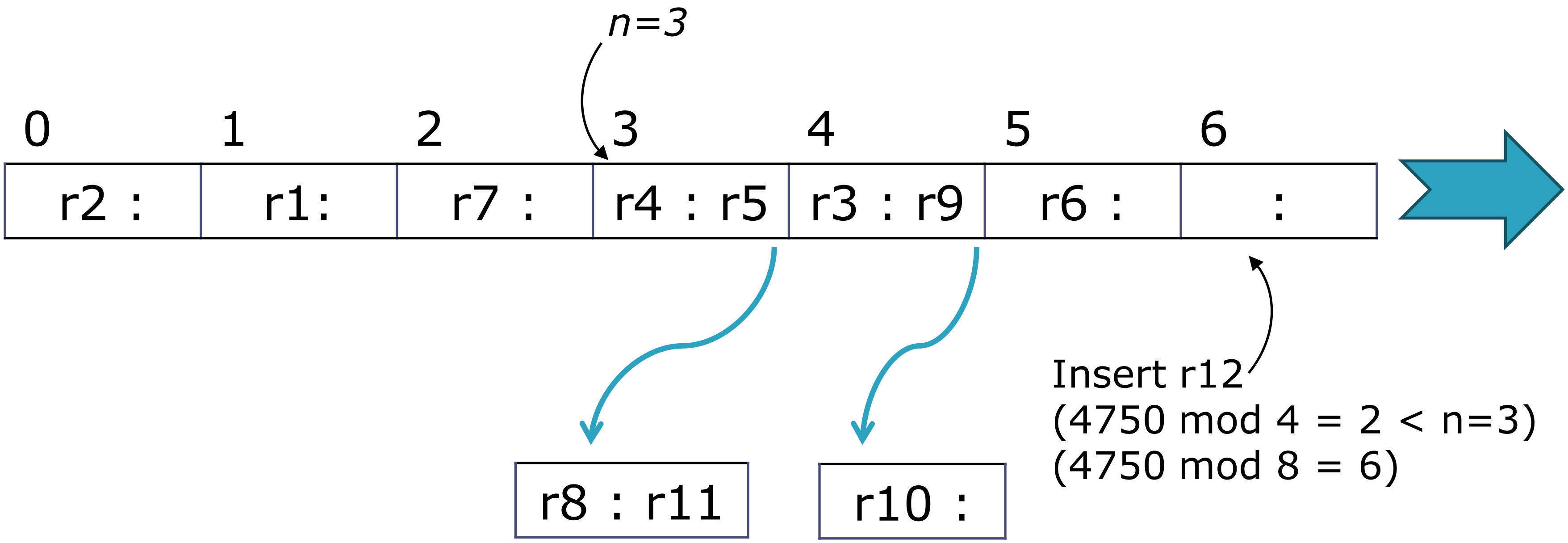
r11 is inserted in the overflow chain of bucket 3.

Split bucket 2 into bucket 2 and bucket 6

$h_1(K) = K \bmod 2*M$

1.5. Hash Files

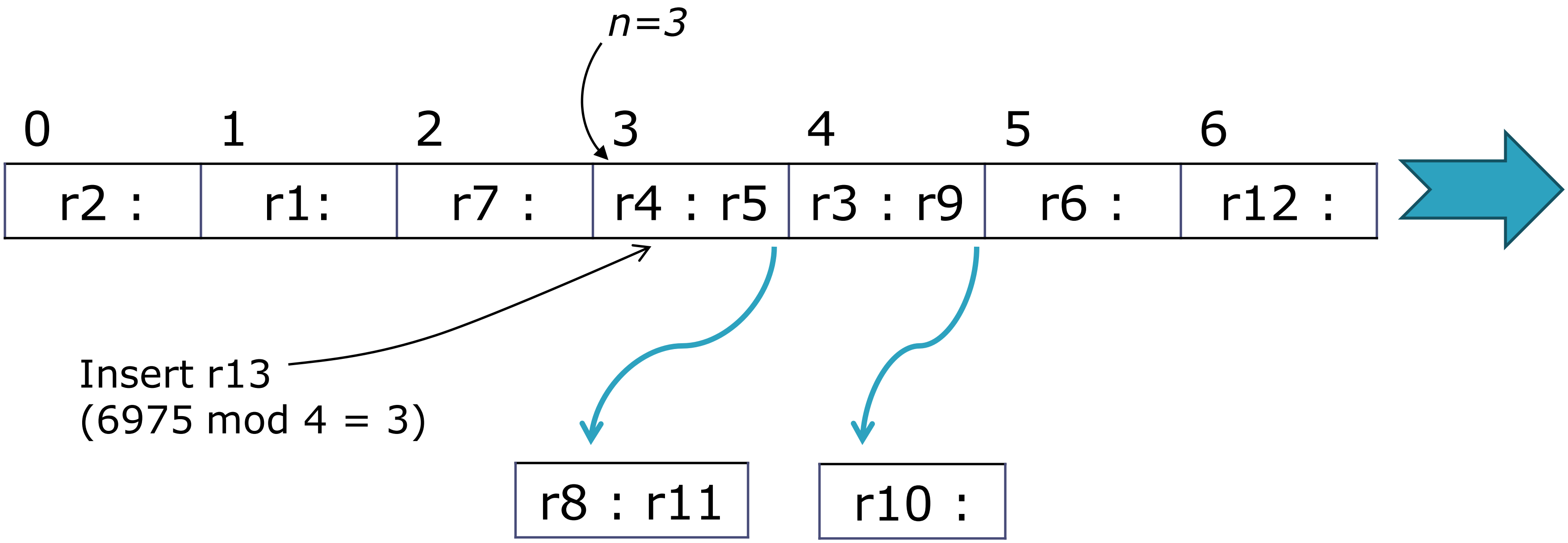
- Linear hashing



Insert r12
(4750 mod 4 = 2 < n=3)
(4750 mod 8 = 6)

1.5. Hash Files

- Linear hashing



Overflow at bucket 3

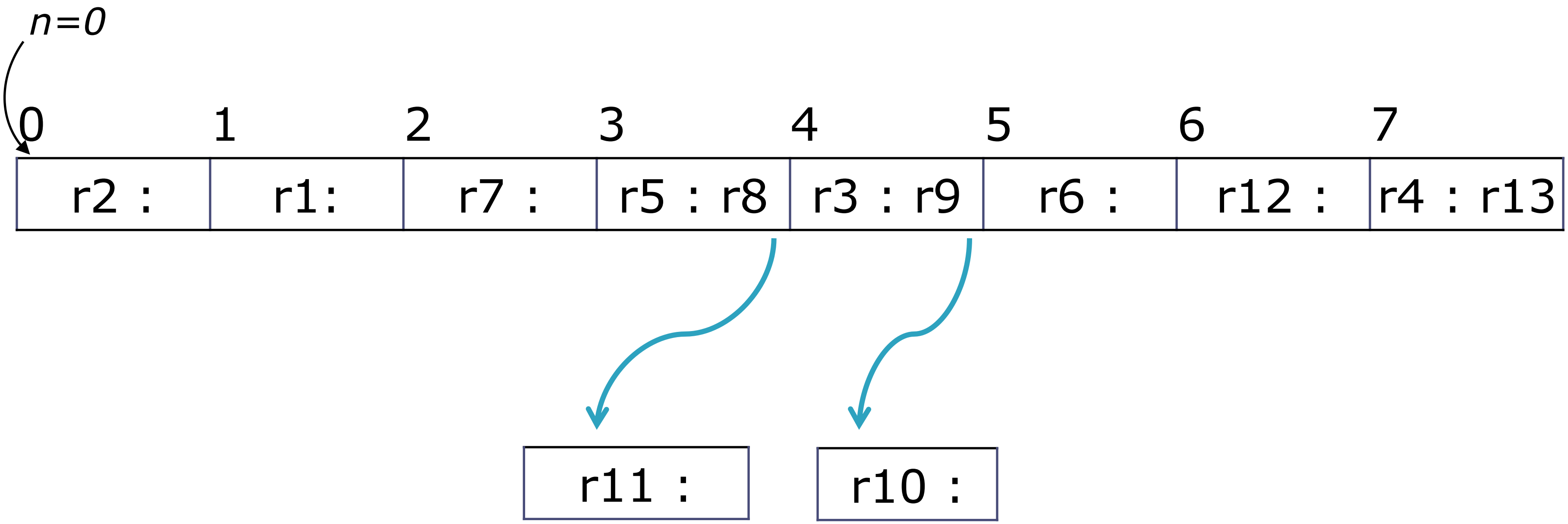
r13 is inserted in the overflow chain of bucket 3.

Split bucket 3 into bucket 3 and bucket 7 => n is reset to 0.

$h_1(K) = K \bmod 2*M$

1.5. Hash Files

- Linear hashing



n=0

0 1 2 3 4 5 6 7

r2 : | r1: | r7 : | r5 : r8 | r3 : r9 | r6 : | r12 : | r4 : r13

r11 :

r10 :

How about insertion of r14 and r15?

1.6. Other File Structures

- Organization for files of mixed records
  - A record type field is used to specify a record type.
  - Relationships among records (in various files)
    - Logical field references
    - Physical relationships by physical contiguity (or clustering) of related records or by physical pointers
- B-trees and other data structures as primary organization
  - Any data structure that can be adapted to the characteristics of disk devices can be used as a primary file organization for record placement on disk.

Summary

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

Summary

- Static files vs. Dynamic files
- File operations
- File organization vs. Access method
- Each primary file organization
  - Placing records in blocks and placing blocks on disks
  - Goal: minimize the number of block transfers
- Further readings
  - Parallelizing disk access using RAID technology
  - Modern storage architectures: storage area networks (SAN), network-attached storage (NAS), …

Chapter 1: Disk Storage and Basic File Structures

Check for understandings

- 1.1. Describe the memory hierarchy for data storage.
- 1.2. Distinguish between persistent data and transient data.
- 1.3. Describe disk parameters when magnetic disks are used for storing large amounts of data.
- 1.4. Describe double buffering. What kind of time can be saved with this technique?
- 1.5. Describe the read/write commands with magnetic disks.

Check for understandings

- 1.6. Distinguish between fixed-length records and variable-length records.
- 1.7. Distinguish between spanned records and unspanned records.
- 1.8. What is blocking factor? How to compute it?
- 1.9. What is file organization? What is its goal?
- 1.10. What is access method? How is it related to file organization?

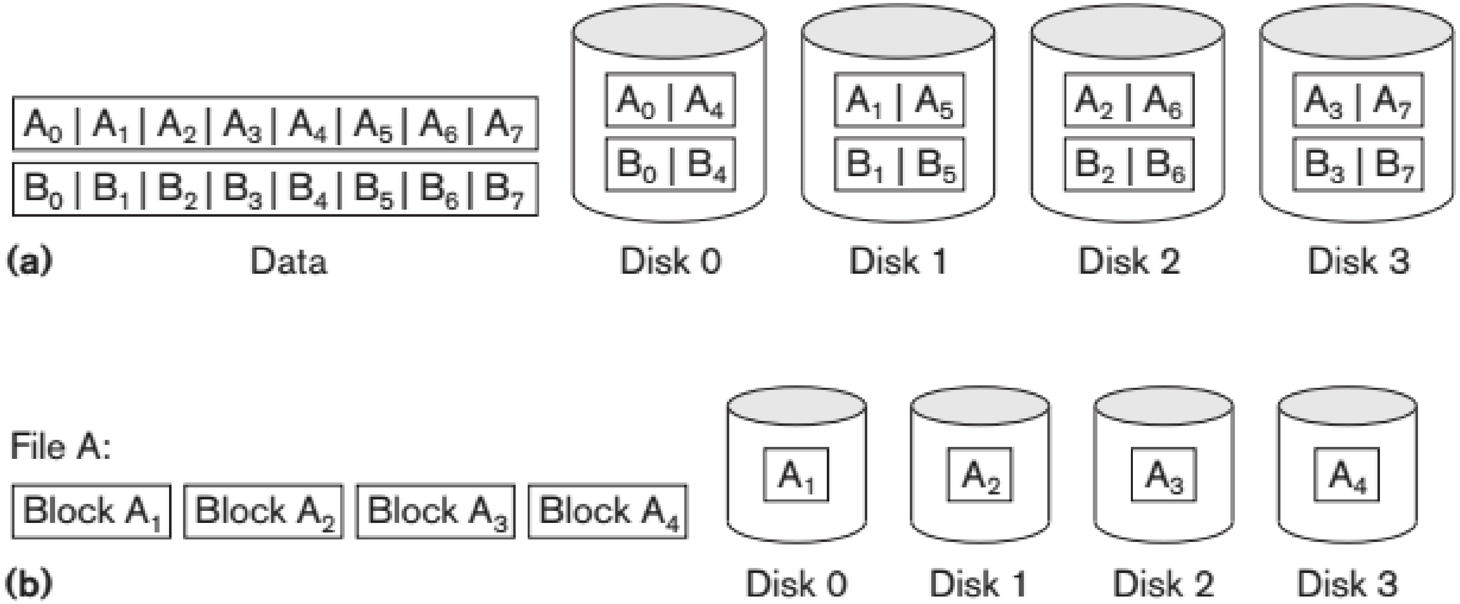
Check for understandings

- 1.11. Distinguish between static files and dynamic files.
- 1.12. Compare unordered files, ordered files, and hash files.
- 1.13. Which operations are more efficient for each file organization: unordered, ordered, hash? Why?
- 1.14. Distinguish between static hashing and dynamic hashing.
- 1.15. Compare three dynamic hashing techniques: Extendible Hashing, Dynamic Hashing, and Linear Hashing.

RAID - Redundant Arrays of Inexpensive/Independent Disks

- **Parallelizing Disk Access**
  - Even out the widely different rates of performance improvement of disks against those in memory and microprocessors
  - A large array of small independent disks acting as a single higher-performance logical disk
  - Data striping distributes data transparently over multiple disks to make them appear as a single large, fast disk.
    - Bit-level striping
    - Block-level striping

RAID - Redundant Arrays of Inexpensive/Independent Disks



Striping of data across multiple disks.

(a) Bit-level striping across four disks.

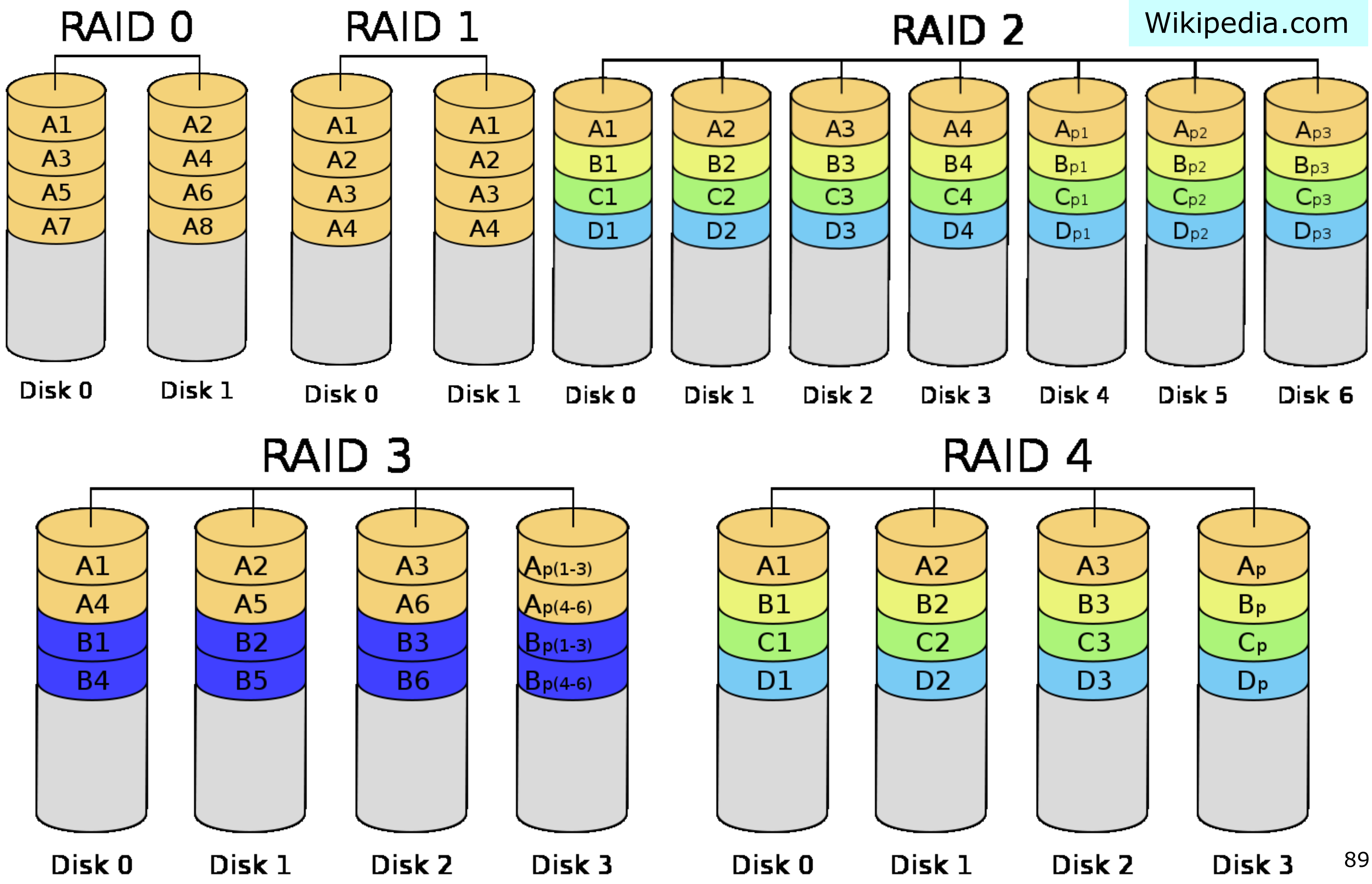
(b) Block-level striping across four disks.

Figure 16.13, pp. 585, [1]

RAID - Redundant Arrays of Inexpensive/Independent Disks

- Improving reliability
  - Redundancy of data so that disk failures can be tolerated
    - Mirror or shadow
    - Store extra information that is not normally needed but that can be used to reconstruct the lost information in case of disk failure
- Improving performance
  - Data striping to achieve higher transfer rates
  - Striping level: bit (vs. byte) vs. block

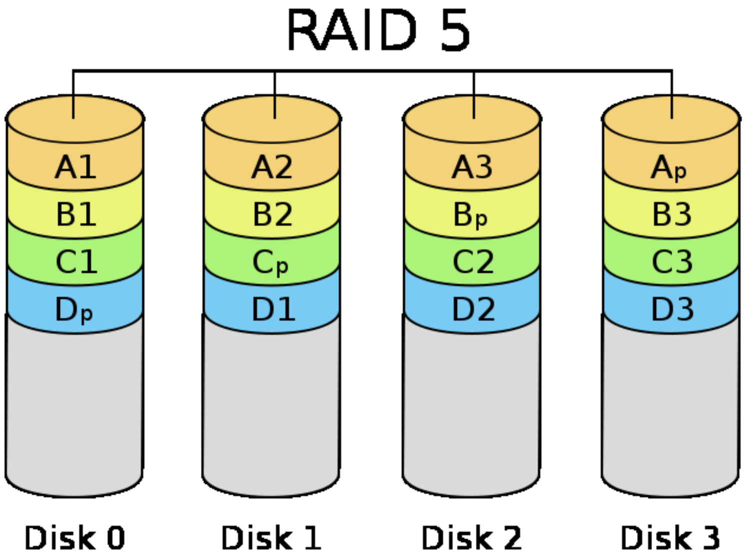
RAID - Redundant Arrays of Inexpensive/Independent Disks

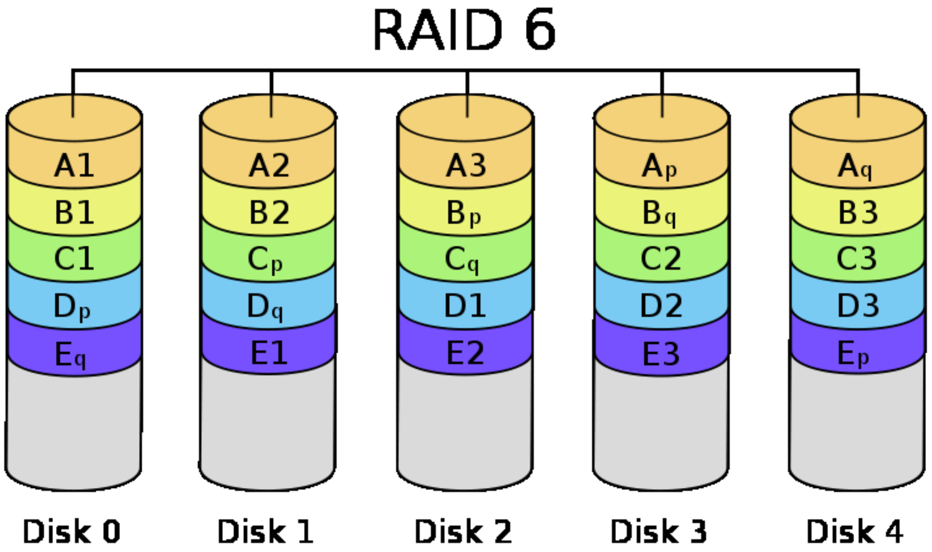


Wikipedia.com

RAID - Redundant Arrays of Inexpensive/Independent Disks

Wikipedia.com





RAID - Redundant Arrays of Inexpensive/Independent Disks

- Other RAIDs
  - RAID 01
  - RAID 10
  - RAID 1.5
  - …
- How many disks can be failed?
  - RAID 0
  - RAID 1
  - RAID 3
  - RAID 5