Disk Storage, Basic File Structures, and Hashing

406.426 Design & Analysis of Database Systems

Jonghun Park

jonghun@snu.ac.kr

Dept. of Industrial Engineering

Seoul National University

chapter outline

- disk storage devices
- files of records
- operations on files
- unordered files
- ordered files
- hashed files
- RAID technology

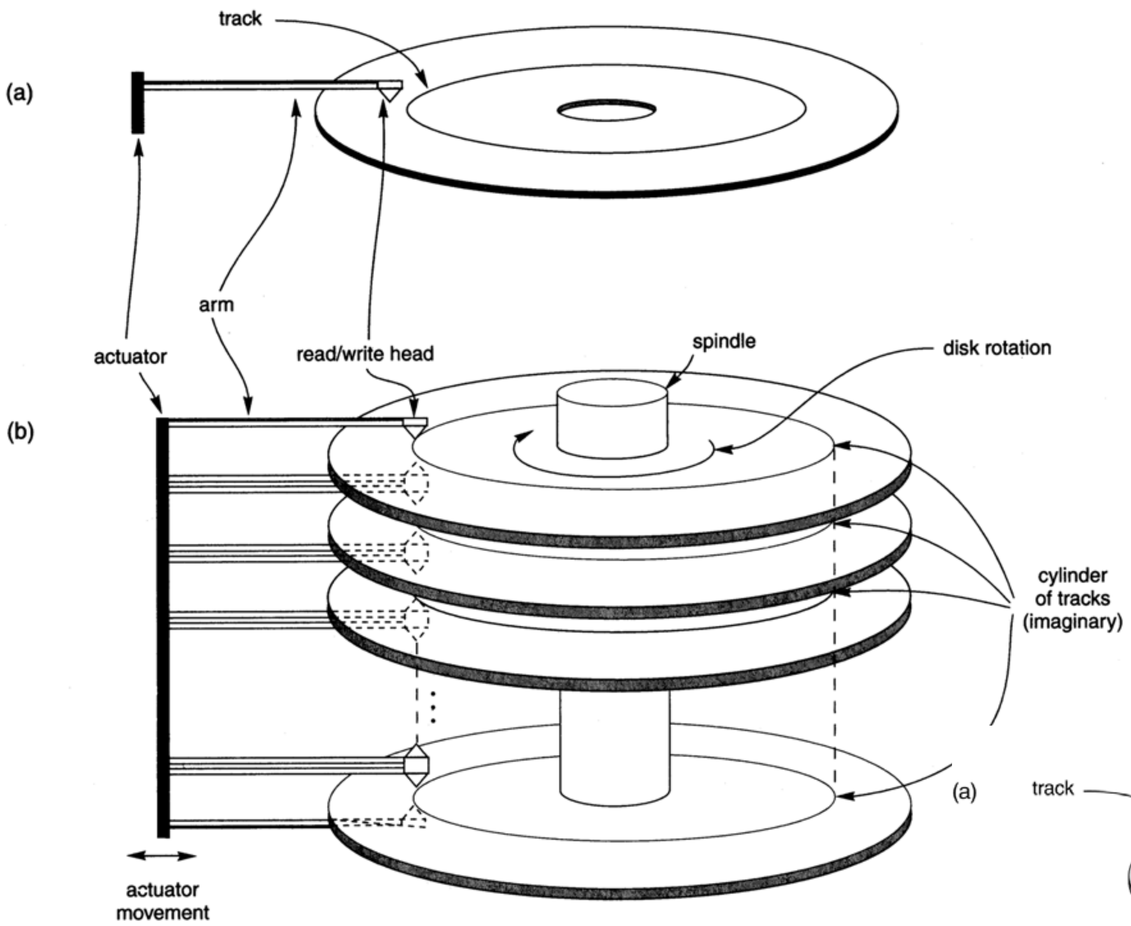
storage hierarchy

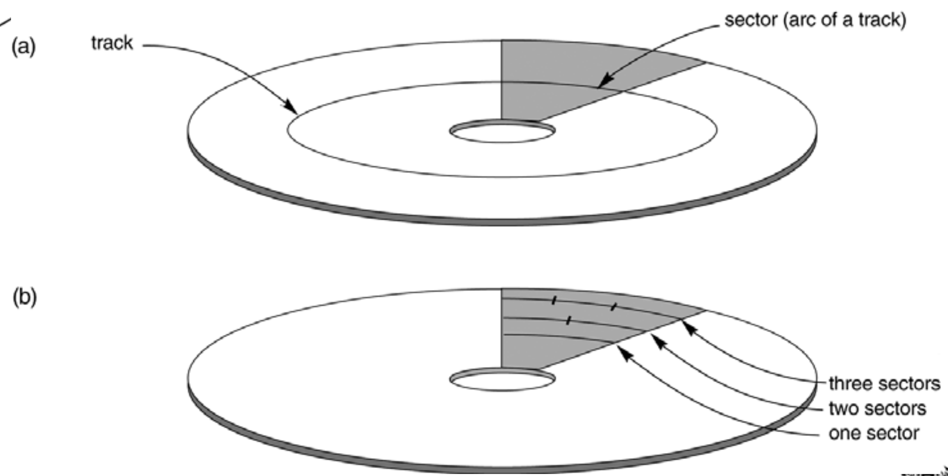
- primary storage
  - storage media that can be operated on **directly by CPU**
  - RAMs: main memory, cache memory
- secondary storage
  - magnetic disks, optical disks, and tapes
  - larger capacity, cost less, slower access than primary storage devices
- flash memory
  - in between DRAM and magnetic disk storage
  - nonvolatile
  - appearing in cameras, MP3P, USB storage, ...
- MMDBMS: entire DBs are kept in main memory
- flash memory DBMS: asymmetric read/write time

storage of DBs

- data stored on disk is organized as **files of records**
- each record is a collection of data values that can be interpreted as facts about entities, their attributes, and their relationships
- primary file organizations: determines how the records of a file are **physically placed** on the disk, and hence how the records can be accessed
  - heap: no particular order
  - sequential file: records are ordered
  - hashed file: uses a hash function applied to a particular field
  - B-tree: uses tree structures

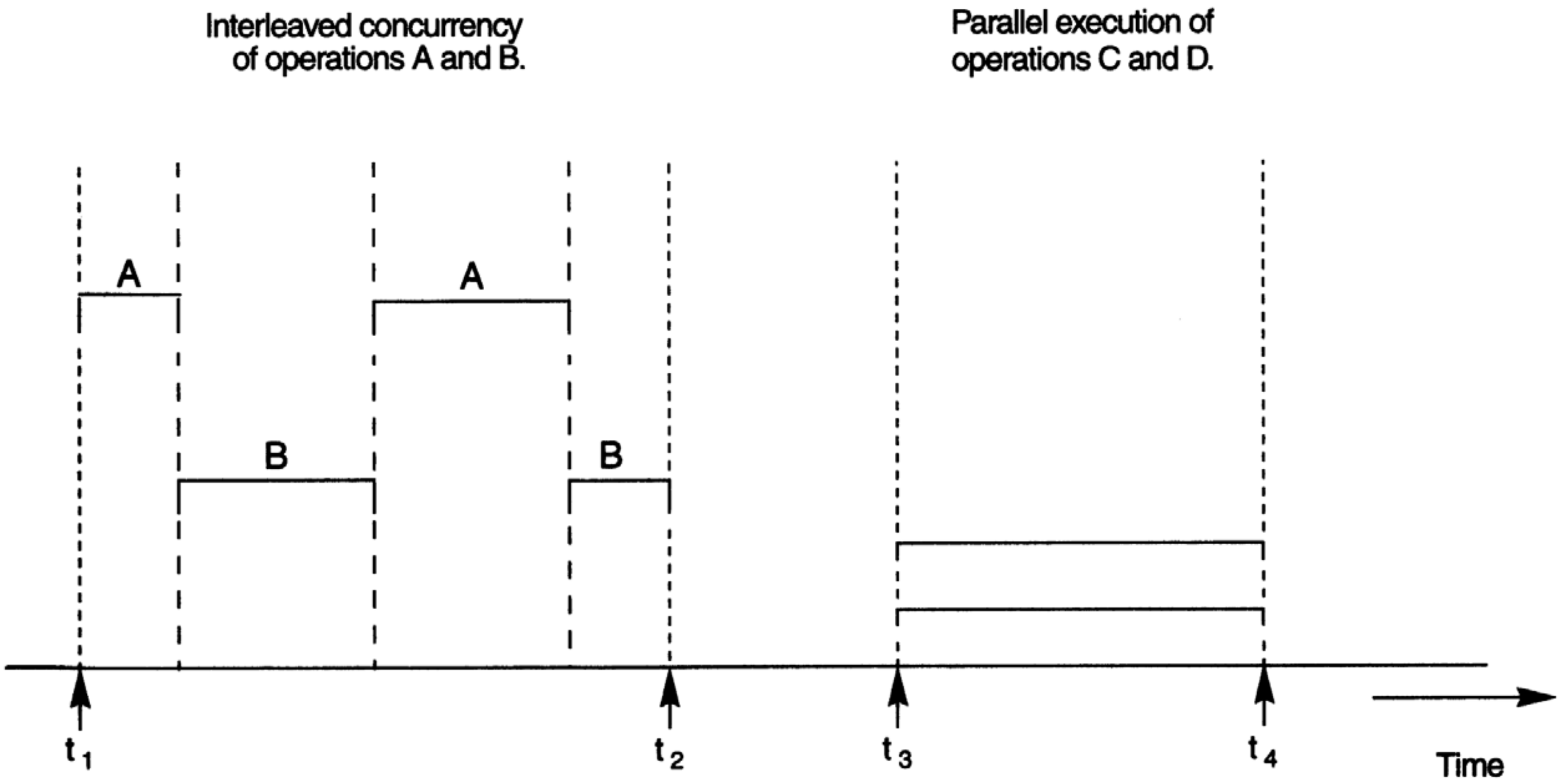
hardware description of disk devices





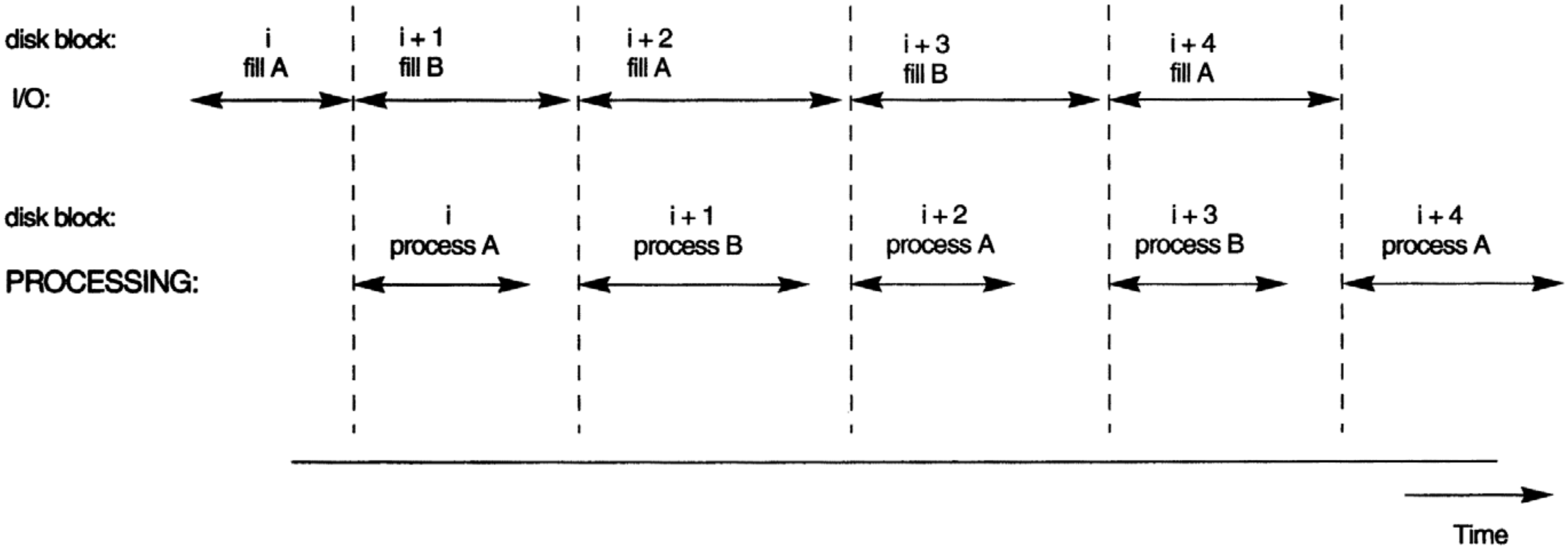
interleaved concurrency vs. parallel execution

- processes A and B are running concurrently in an interleaved fashion, whereas processes C and D are running concurrently in a parallel fashion



buffering of blocks

- double buffering: reading and processing can proceed in parallel when the time required to process a disk block in memory is less than the time required to read the next block and fill a buffer



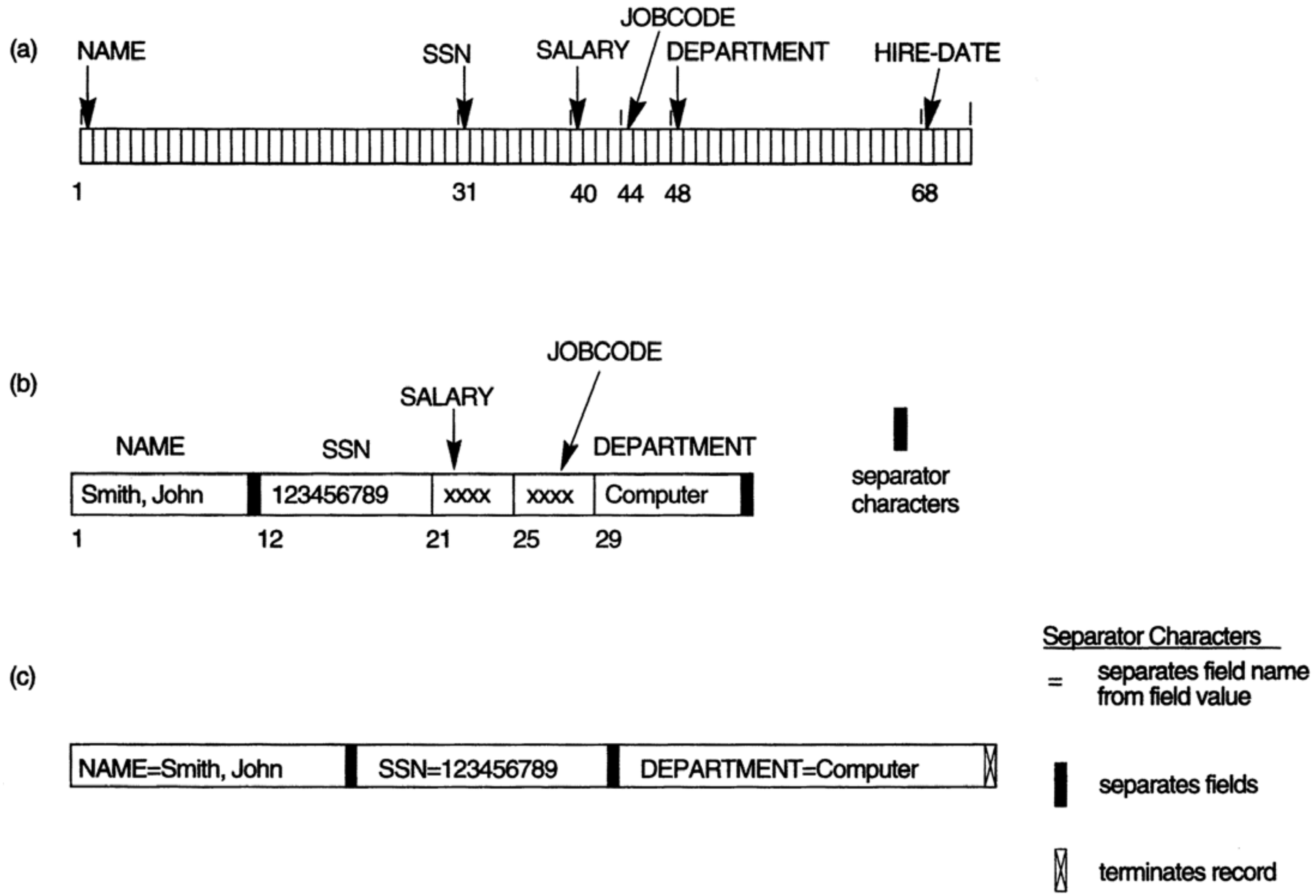
records and record types

- data is usually stored in the form of **records**
- each record consists of a **collection of related data values** or items, where each value is formed of one or more bytes and correspond to a particular **field** of the record
- a collection of **field names** and their corresponding **data types** constitute a **record type**
- data type of a field is usually one of the standard data types used in programming
    - numeric (integer, long integer, or floating point)
    - string of characters (fixed-length or varying)
    - Boolean
    - date and time
- BLOBs (Binary Large Objects)
    - data items that consist of large unstructured objects, which represent images, digitized videos, or audio streams, or free text

files, fixed-length records, and variable-length records

- file: a sequence of records
- in many cases, all records in a file are of the **same record type**
- fixed-length records: every record in the file has exactly the same size
- variable-length records: different records in the file have different sizes
- reasons for having the variable-length records
  - one or more of the fields are of varying size: e.g., NAME
  - one or more of the fields may have multiple values for individual records: called a repeating field
  - one or more of the fields are optional
  - file contains records of different record types

record storage formats

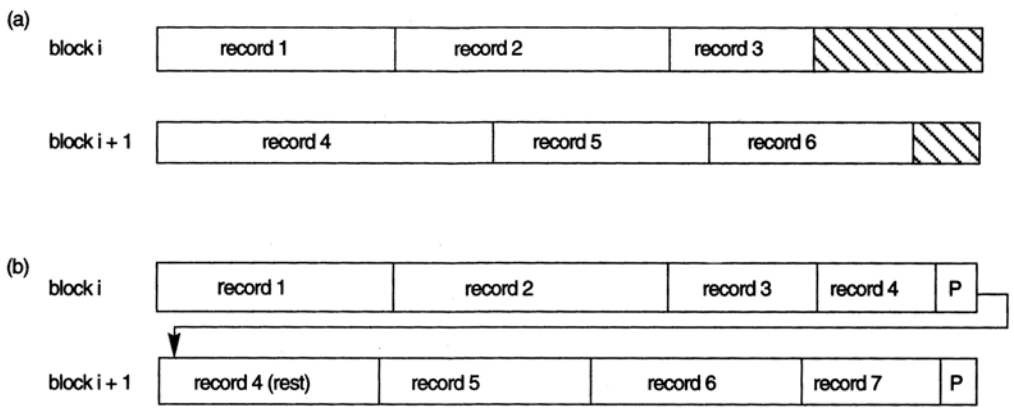


representation of the variable-length records

- optional fields
  - let every field be included in every record, but store a **special null value** if no value exists
  - or include in each record a sequence of **<field-name, field-value>** pairs
- repeating fields
  - allocate as many spaces in each record as the **maximum number of values** that the field can take
  - or use one **separator character** to separate the repeating values of the field and another separator character to indicate termination of the field
- variable-length fields
  - use special **separator** characters which do not appear in any field value to terminate variable-length fields

record blocking, spanned vs. unspanned records

- block: the **unit of data transfer** between disk and memory
- records of a file must be allocated to disk blocks
- blocking factor: bfr
  - *B*: the block size (in 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
  - unused space in each block: $B - (\text{bfr} * R)$ bytes
- spanned record
  - store part of a record in one block and the rest on another
  - pointer at the end of the first block points to the block containing the remainder of the record
  - whenever a record is larger than a block, we must use a spanned organization



allocating file blocks on disk

- contiguous allocation
  - blocks are allocated to consecutive disk blocks
  - makes reading the whole file very fast
  - **makes expanding the file difficult**
- linked allocation
  - each block contains a pointer to the next block
  - easy to expand the file but makes it **slow to read** the whole file
- combination of the above
  - allocates **clusters** of consecutive disk blocks and the cluster are linked
- indexed allocation
  - one or more index blocks contain pointers to the actual file blocks

file headers

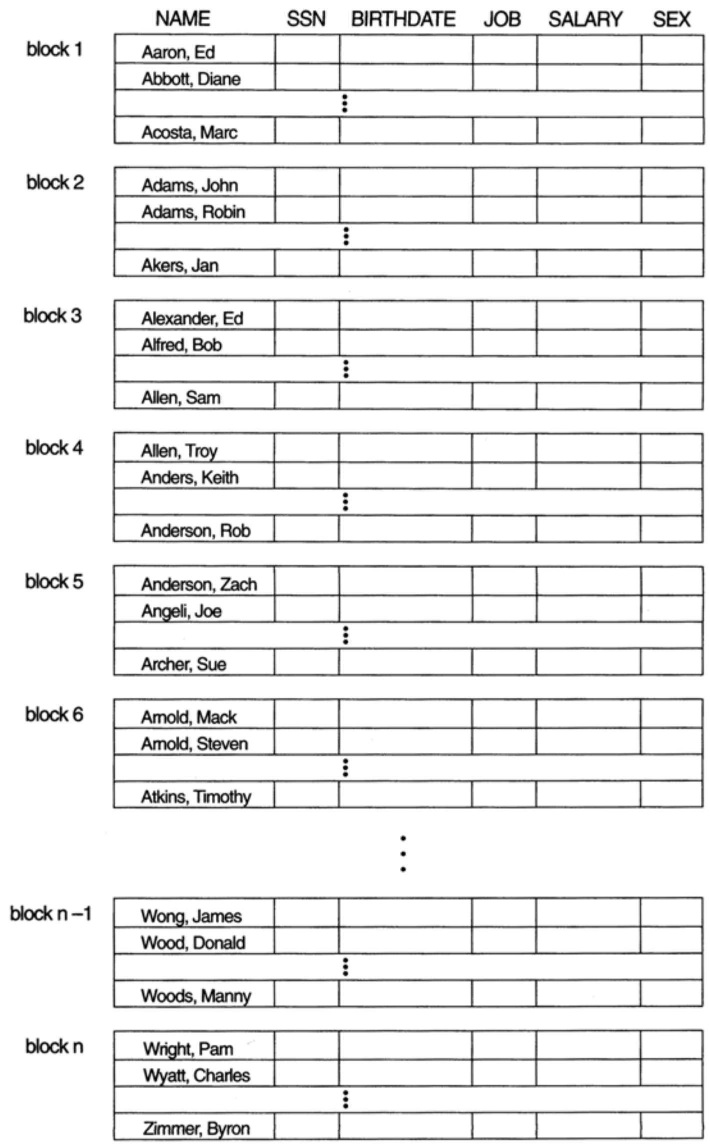
- header includes information to determine the disk **addresses** of the file blocks as well as to record **format descriptions**, which may include field lengths and order of fields within a record for fixed-length unspanned records and field type codes, separator characters, and record type codes for variable-length records

heap files

- **simplest** and most basic type of organization
- records are placed in the file **in the order in which they are inserted**, so new records are inserted at the end of the file
- inserting a new record is very efficient: the last block of the file is copied into a buffer; the new record is added; and the block is then rewritten back to disk
- searching for a record using any search condition involves a **linear search**
  - when only one record satisfies the search condition: for a file of *b* blocks, searching (*b*/2) blocks is required on the average
  - when no records or several records satisfy the search condition: searching all *b* blocks is required
- deletion
  - find the block and delete the record
  - deletion marker: a record is deleted by setting the **deletion marker** to a certain value
- accessing a record by its position in the file of fixed-length records using unspanned blocks and contiguous allocation
  - records in the file are numbered 0, 1, 2, ..., *r*-1
  - records in each block are numbered 0, 1, 2, ..., bfr-1
  - the *i*-th record of the file is located in block $\lfloor i/\mathrm{bfr} \rfloor$ and is the (i mod bfr)-th record in that block

sorted files (sequential files)

- physically order the records of a file on disk based on the **values of one of their fields** (called ordering field)
- advantages
  - reading the records in order of the ordering key values becomes extremely efficient
  - finding the next record from the current one in order of the ordering key usually requires no additional block access
  - **binary search** can be used for a search condition based on the value of an ordering key field
    - max time to access a specific record is $\log_2 b$



binary search on an ordering key of a disk file

- searching for a record whose ordering key field value is K
- b is the number of blocks
- $l \leftarrow 1$; $u \leftarrow b$;
  while ($u \geq l$) do
    $i \leftarrow (l + u) / 2$
    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
- cf. number guessing game based on “high” / “low” hints

sorted files (cont.)

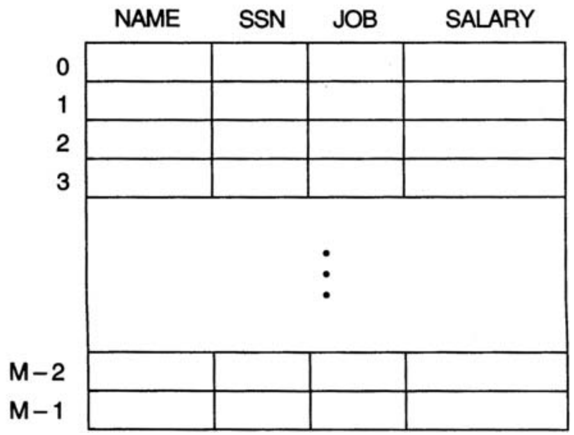
- linear search for the nonordering fields
- inserting and deleting records are expensive operations because the records must remain physically ordered
  - on the average, half the records of the file must be moved to make space for the new record
  - for the record deletion, the problem is less severe if deletion markers and periodic reorganization are used
- one option for making insertion more efficient is to keep some unused space in each block for new records
- ordered files are rarely used

hash files

- provides very fast access to records on certain search conditions
- search condition must be an **equality condition on a single field**, called the **hash field** of the file
- idea
  - provide a function *h*, called a **hash function** that is applied to the hash field value of a record and **yields the address of the disk block** in which the record is stored

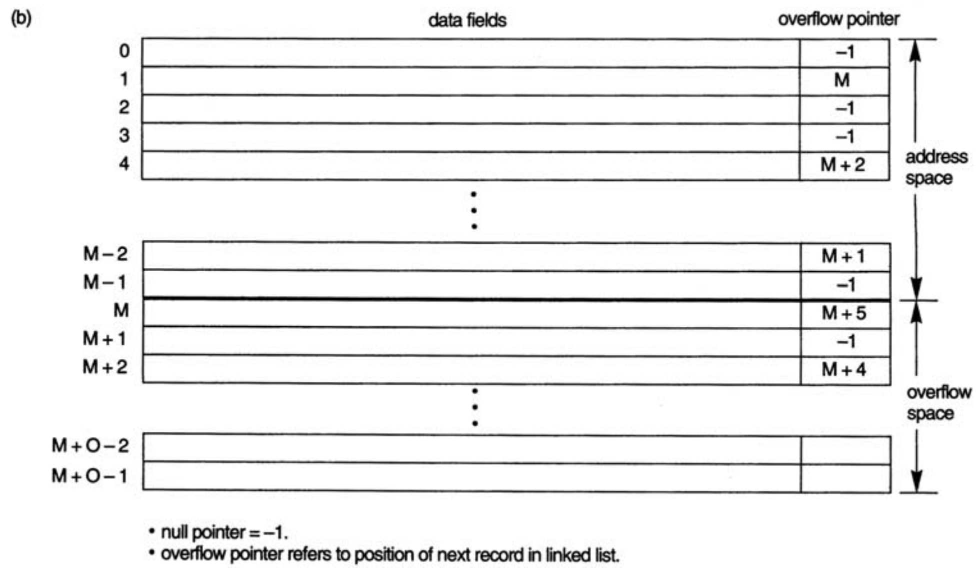
internal hashing

- hashing is implemented as a hash table through the use of an array of records
- array index: 0, ..., M-1
- choose a hash function that transforms the hash field value into an integer between 0 to M-1
  - e.g., $h(K) = K \bmod M$, where hash key field value is K
- problem
  - # of possible values for a hash field >> # of available addresses for records
  - does not guarantee distinct values will has to distinct addresses



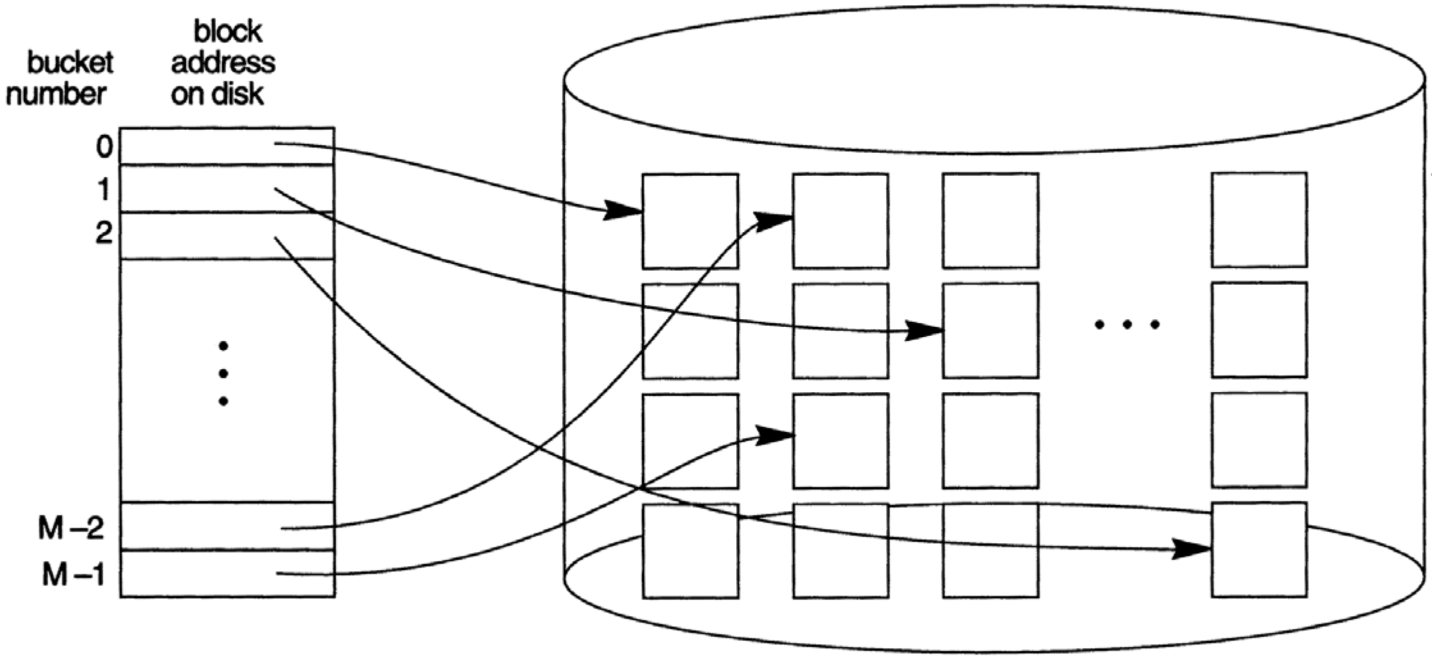
collision resolution

- a collision occurs when the hash field value of a record that is being inserted hashes to an address that already contains a different record
- 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 position is found
  - chaining: place the new record in an **unused overflow location** and set the pointer of the occupied hash address location to the address of that overflow location
  - multiple hashing: **applies the second, third, ... hash function** if the first results in a collision. if another collision results, the program uses open addressing
- goal of a good hashing function: to **distribute the records uniformly** over the address space so as to **minimize collisions while not leaving many unused locations**



external hashing

- target address space is made of **buckets**, each of which holds **multiple records**
- bucket is either one disk block or a cluster of contiguous blocks
- hash function **maps a key into a relative bucket number**, rather than assign an absolute block address to the bucket



collision resolution

- **collision problem is less severe** with buckets, because as many records as will fit in a bucket can hash to the same bucket without causing problems
- use 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: includes both a block address as well as a relative record position within the block

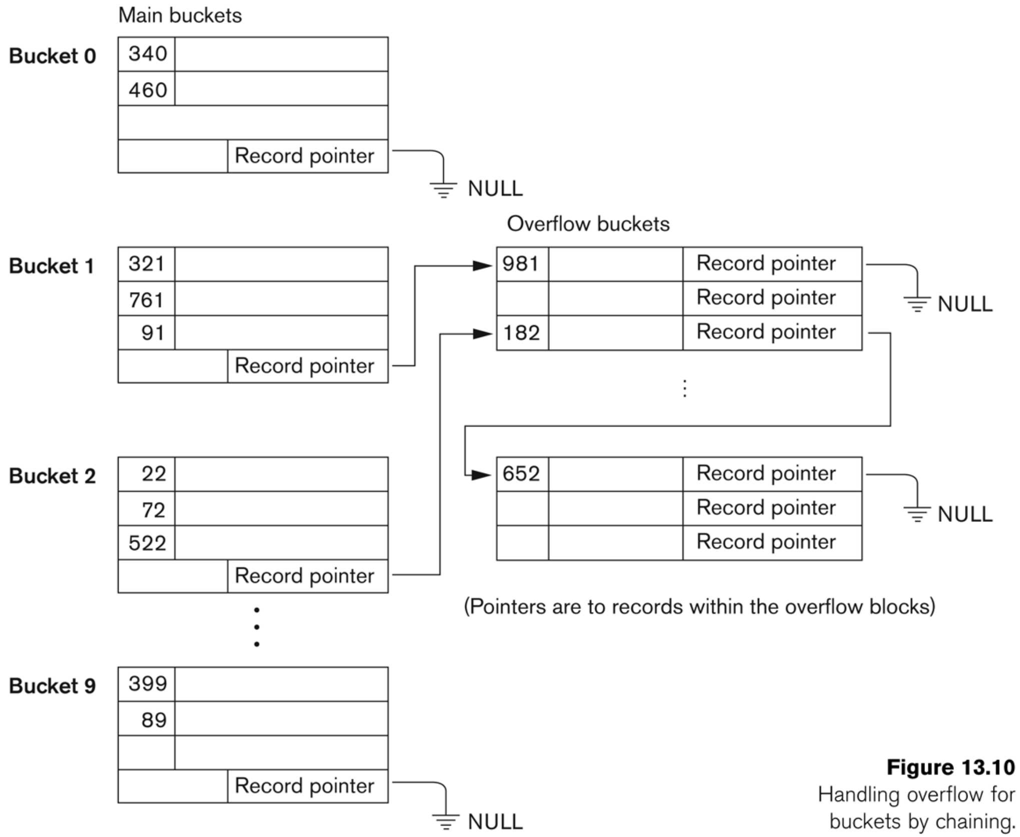


Figure 13.10
Handling overflow for buckets by chaining.

more on hashing

- hashing provides **the fastest possible access for retrieving an arbitrary record** given the value of its hash field
- order preserving hashing
  - maintains records in order of hash field values
  - e.g., take the leftmost three digits of an invoice number field as the hash address and keep the records sorted by invoice number within each bucket
- static hashing
  - a fixed number of buckets *M* is allocated, and each bucket may have up to *m* records
  - a serious drawback for dynamic files
  - what if the # of records turns out to be << (or >>) (m*M)?
- dynamic hashing
  - extendible hashing, linear hashing

extendible hashing

- stores an **access structure** in addition to the file
- access structure is built on the binary representation of the hashing function result
- a type of directory, **an array of 2^d bucket address**, is maintained, where ***d*** is called the **global depth** of the directory
  - initially, $d = 1$
- integer value corresponding to the **first *d* bits** of a hash value is used as an index to the array to determine a **directory entry**, and the address in that entry determines the **bucket** in which the corresponding records are stored
- several directory locations with the same first ***d'*** (called **local depth**; <= *d*) bits for their hash values many contain the same bucket address if all the records that hash to these locations **fit in a single bucket**

structure of the extendible hashing scheme

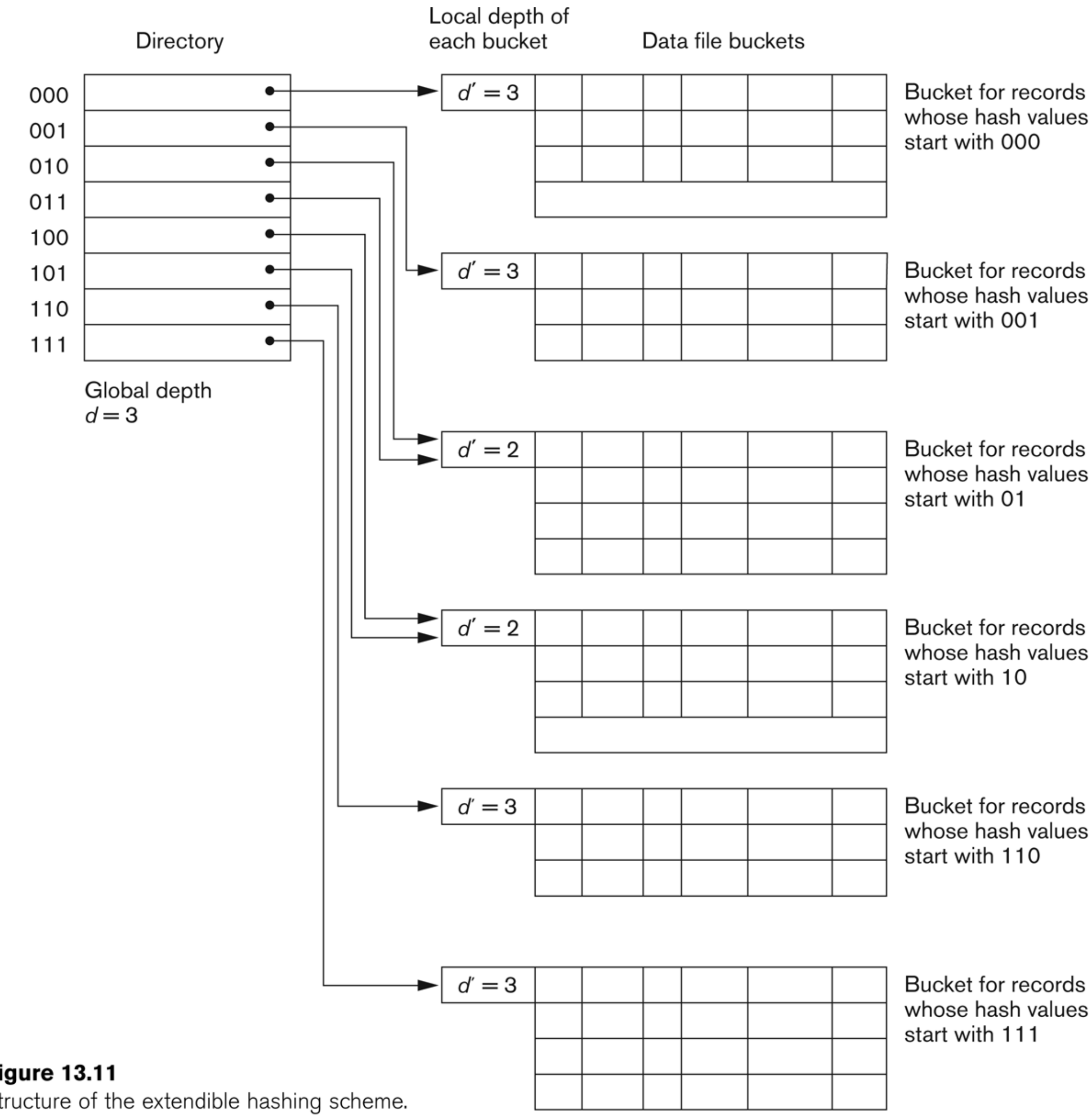


Figure 13.11
Structure of the extendible hashing scheme.

extendible hashing (cont.)

- bucket splitting
  - bucket whose hash values start with 01 **overflows** -> the bucket that contains all records whose hash values start with 010, and the bucket that contains all records whose hash values start with 011
- 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
- halving occurs if *d* > *d'* for all the buckets after some deletions occur
- advantages
  - performance of the file does not degrade as the file grows
  - splitting causes minor reorganization in most cases, since only the records in one bucket are redistributed to the two new buckets
- disadvantages
  - directory must be searched before accessing the buckets themselves, resulting in **two block access** instead of one in static hashing

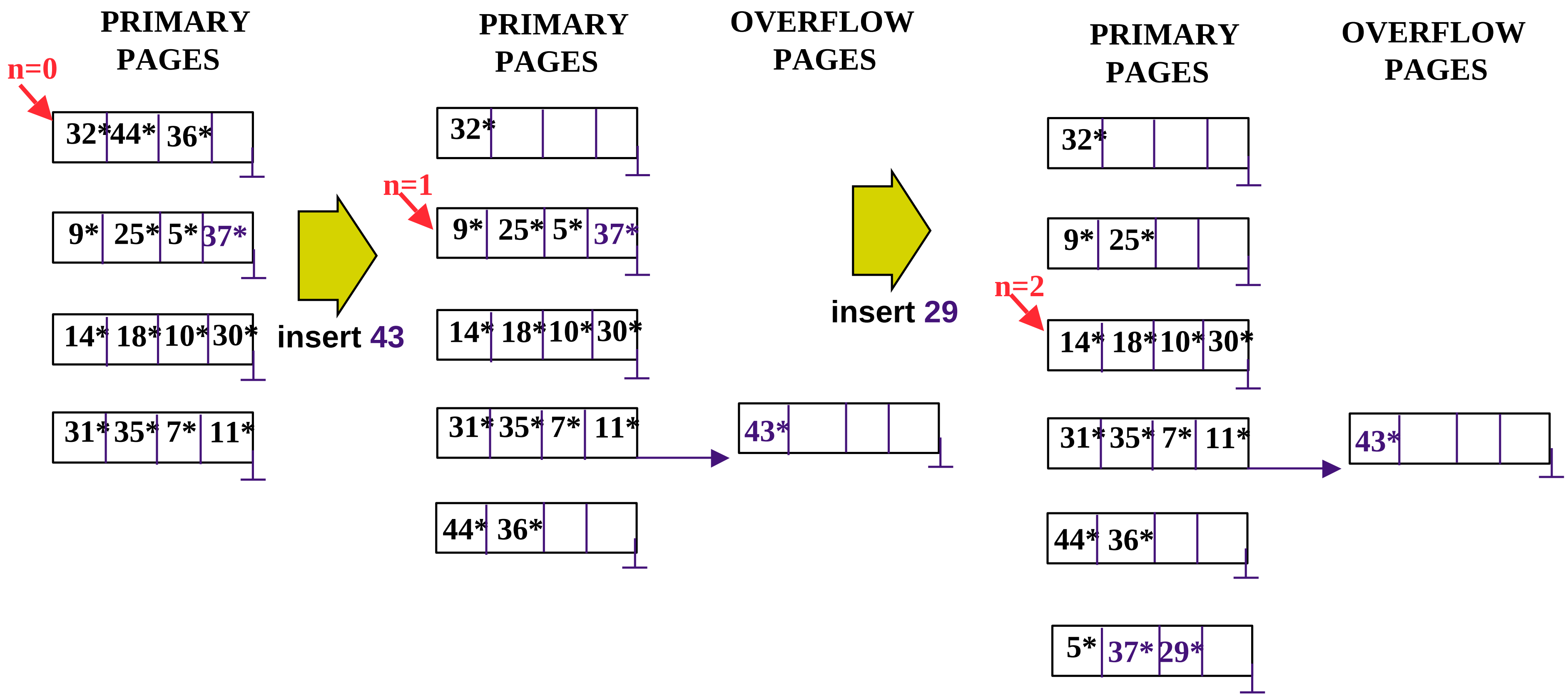
linear hashing

- to allow a hash file to expand and shrink its number of buckets dynamically **without needing a directory**
- file starts with *M* buckets: 0, 1, ..., *M*-1
- initial hash function $h_i(K) = K \bmod M$
- when a collision leads to an overflow record in **any** bucket, bucket 0 is split into two buckets: the original bucket 0 and a new bucket *M* at the end of the file
- records originally in bucket 0 are distributed between the two buckets based on $h_{i+1}(K) = K \bmod 2M$
  - any record that hashed to bucket 0 based on h_i will hash to either bucket 0 or bucket *M* based on h_{i+1}
- splits are performed in **linear order (bucket 0 first, then bucket 1, then 2, ...)**, and a split is performed when **any** bucket overflows
- if the bucket that overflows is not the bucket that is split (which is the common case), overflow techniques such as chaining are used
- if enough overflows occur, all the original file buckets, 0, 1, ... *M*-1 will have been split, so the file now has 2*M* buckets, and all buckets use the hash function h_{i+1}

linear hashing (cont.)

- no directory is needed, only a value $\boldsymbol{n}$, which is initially set to 0 and is **incremented by 1 whenever a split occurs**, is needed to determine which buckets have been split
- to retrieve a record with hash value K, first apply the function h_i to K; if $h_i(K) < n$, then apply the function h_{i+1} on K because the bucket is already split
- when $n = M$, this signifies that all the original buckets have been split and the h_{i+1} applies to all records in the file -> at this point, n is reset to 0, and any new collisions that cause overflow lead to the use of a new hashing function $h_{i+2}(K) = K \bmod 4M$
- in general, a sequence of hashing functions, $h_{i+j}(K) = K \bmod (2^j M)$ is used, where j = 0, 1, 2, ...; a new hashing function h_{i+j+1} is needed whenever all the buckets, 0, 1, ..., $(2^j M) - 1$ have been split and n is reset to 0

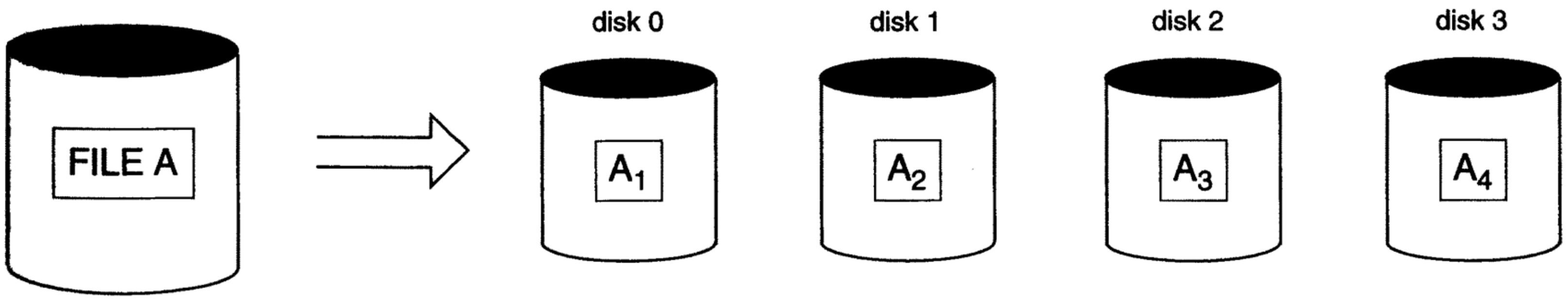
example: *M* = 4



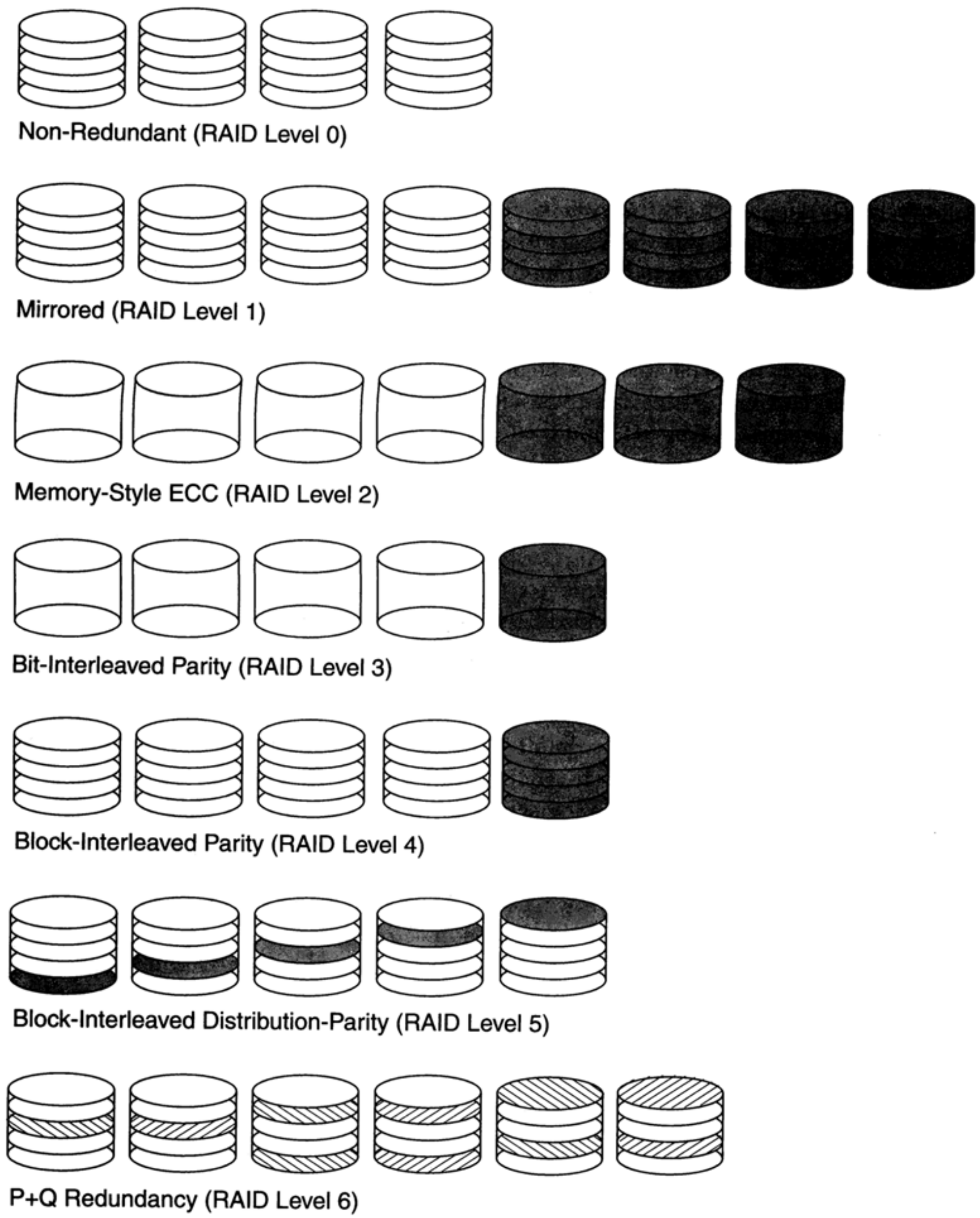
PRIMARY PAGES
n=0
32* 44* 36*
9* 25* 5* 37*
14* 18* 10* 30*
31* 35* 7* 11*
insert 43
PRIMARY PAGES
OVERFLOW PAGES
32*
n=1
9* 25* 5* 37*
14* 18* 10* 30*
31* 35* 7* 11*
43*
44* 36*
insert 29
PRIMARY PAGES
OVERFLOW PAGES
32*
9* 25*
n=2
14* 18* 10* 30*
31* 35* 7* 11*
43*
44* 36*
5* 37* 29*

RAID

- redundant arrays of independent disks
- to 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**
- a concept called data striping is used, which utilizes parallelism to improve disk performance
  - improves overall I/O performance by allowing multiple I/Os to be service in parallel
- by storing redundant information on disks using parity or some other error correction code, reliability can be improved



use of RAID technology



SAN

- storage area networks
- online storage peripherals are configured as nodes on a high-speed network and can be attached and detached from servers in a very flexible manner