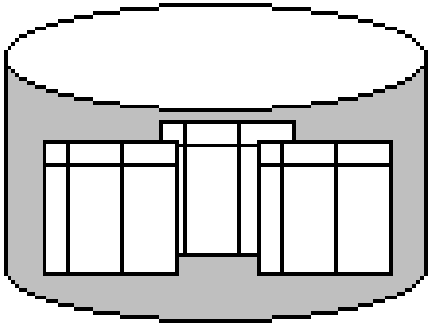
ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ

Συστήματα Διαχείρισης Βάσεων Δεδομένων

Διάλεξη 1η: Data storage, Record and file structures

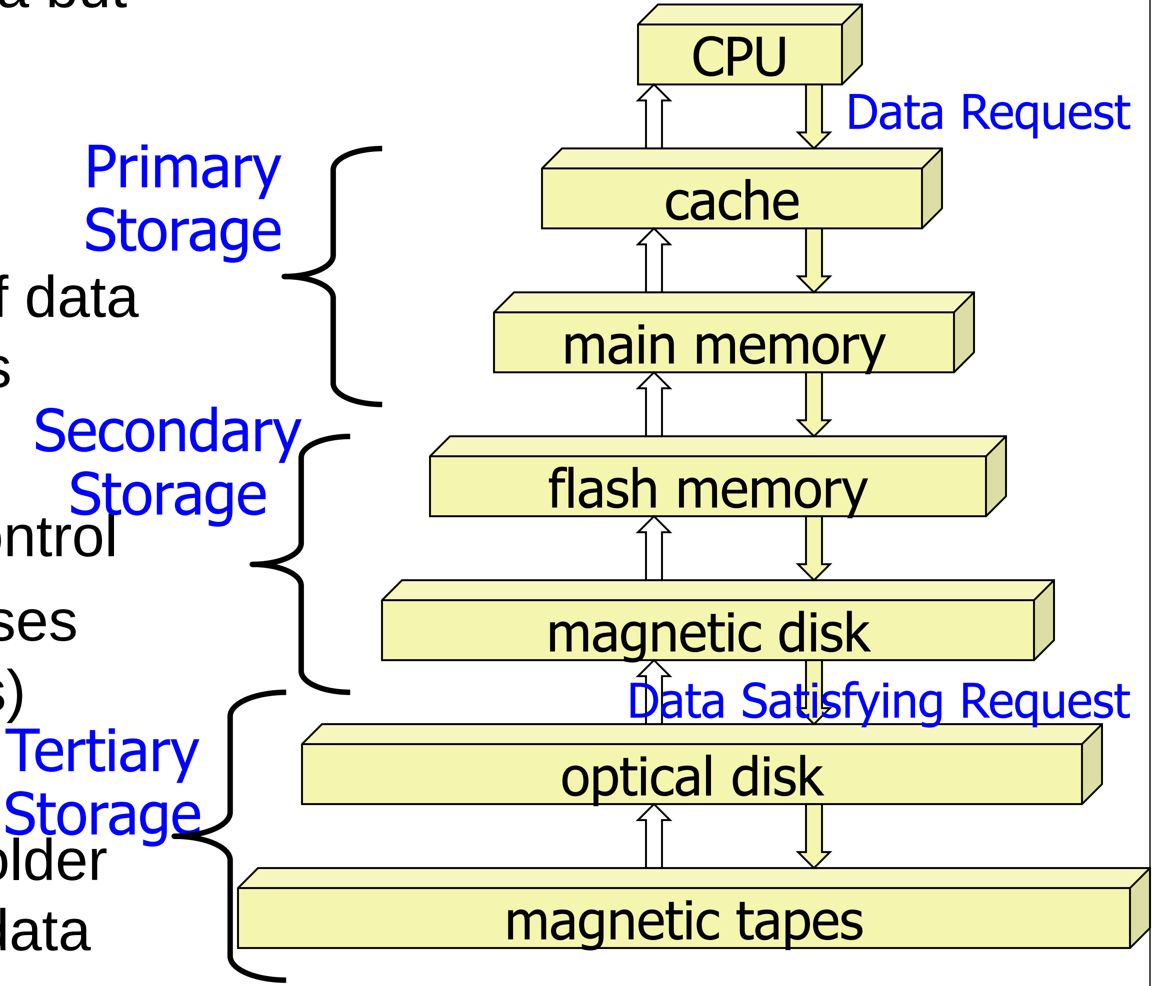
Δημήτρης Πλεξουσάκης
Τμήμα Επιστήμης Υπολογιστών

DATA STORAGE, RECORD and FILE STUCTURES

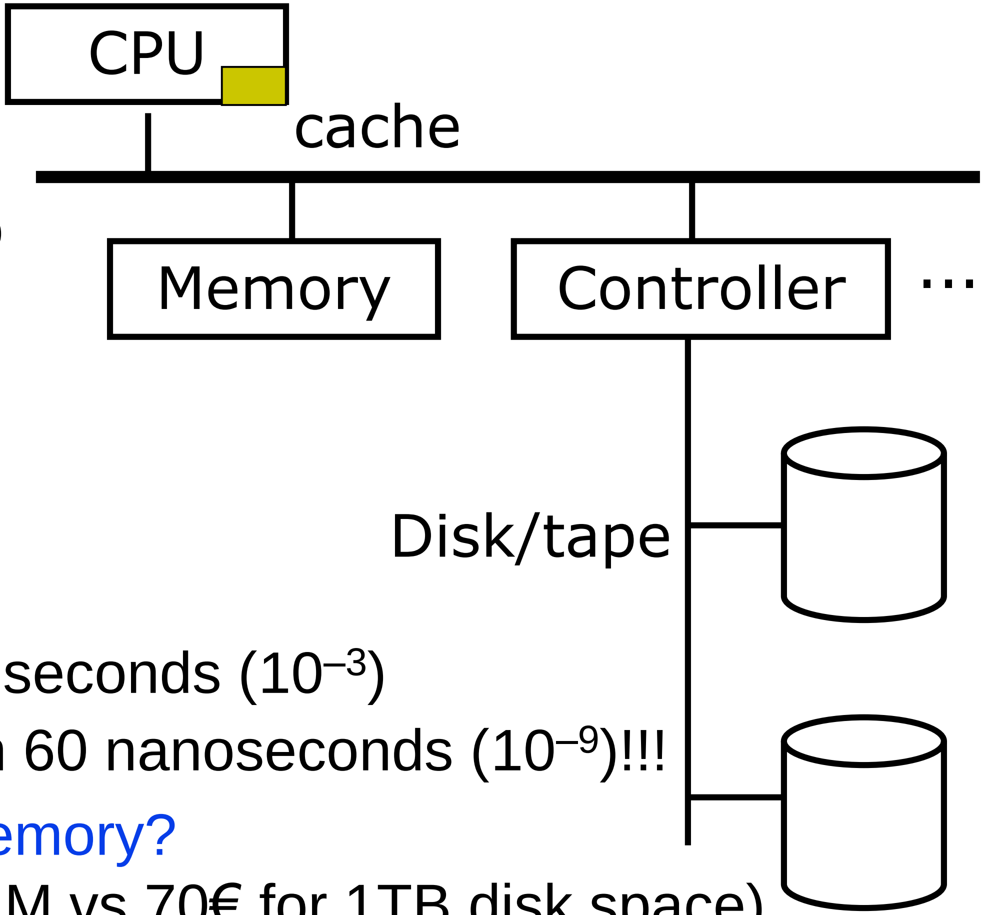


Typical Memory Hierarchy

- Primary storage: Fastest media but volatile (cache, main memory)
  - ◆Main memory for currently accessed data
  - ◆Cache for small amounts of data and/or machine instructions
    - on-chip (L1) and L2
    - outside of DB system control
- Secondary storage for databases (flash memory, magnetic disks)
  - ◆also called on-line storage
- Tertiary storage for archiving older versions of infrequently used data (tapes, DVDs, jukeboxes)
  - ◆also called off-line storage

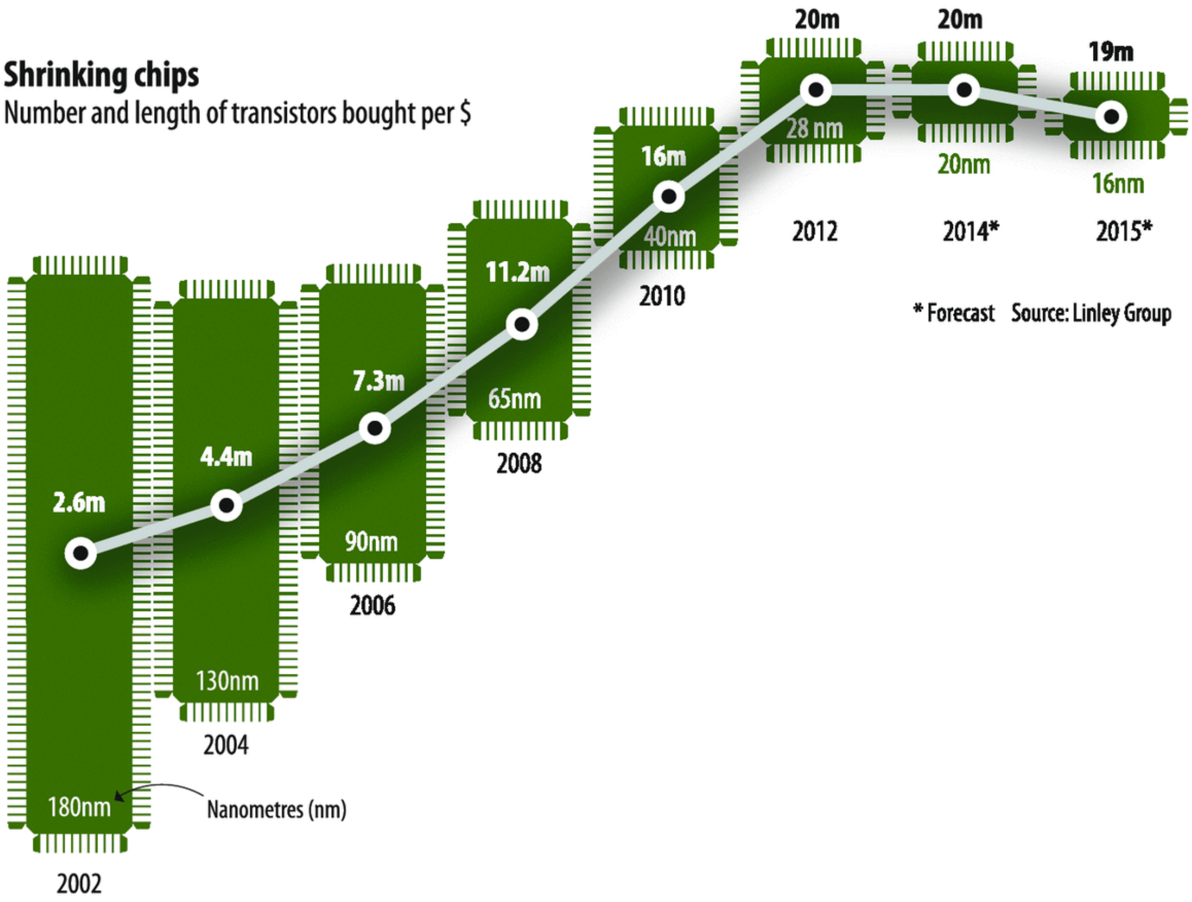


Data Storage



- A DBMS stores information on disk; manipulation of data takes place in main memory
  - READ: transfer data from disk to main memory
  - WRITE: transfer data from main memory to disk
- Both are high-cost operations, relative to in-memory operations
  - Typical disk access takes 10 milliseconds (10^{-3})
  - Main memory access is less than 60 nanoseconds (10^{-9})!!!
- Why not store everything in main memory?
  - costs too much! (70€ for 8GB RAM vs 70€ for 1TB disk space)
  - main memory is volatile: contents are usually lost if a power failure or system crash occurs

Moore’s Low



- Processor speed doubles every 18 months (2x18 months ~ 100x10 years)
  - ◆CPUs will get faster, disks will get bigger, and so do communication speeds… (http://www.intel.com/research/silicon/mooreslaw.htm)

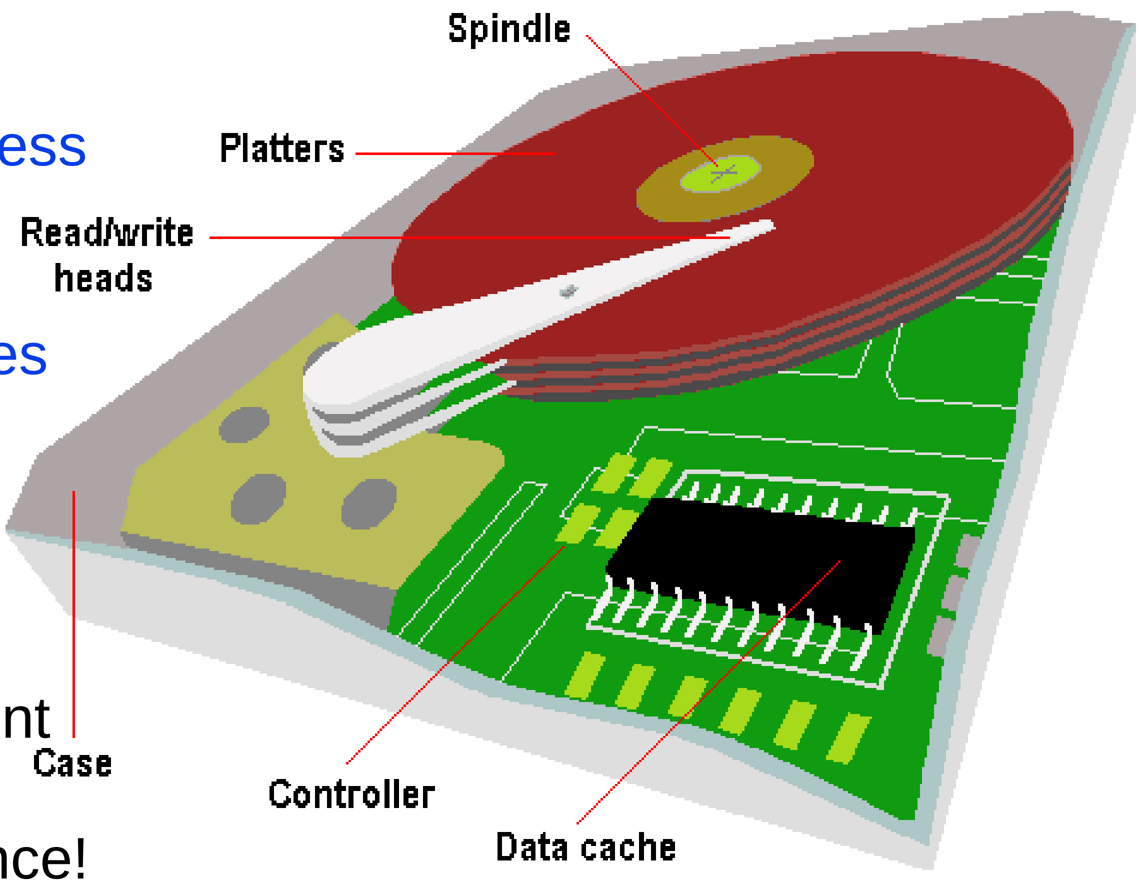
Exponential Growth

	1969	**2001**	**Factor**
main memory	200 KB	200 MB	10^3
cache	20 KB	20 MB	10^3
cache pages	20	5000	$<10^3$
disk size	7.5 MB	20 GB	$3*10^3$
disk/memory size	40	100	**-2.5**
transfer rate	150 KB/s	15 MB/s	10^2
random access	50 ms	5 ms	10
scanning full disk	130 s	1300 s	**-10**

- Over the last decade:
    - 10x better memory access time
    - 10x more bandwidth
    - 100x more capacity
    - 4000x lower media price
    - Disk scan takes 10x longer
    - Data on disk are 2.5x bigger than the memory size

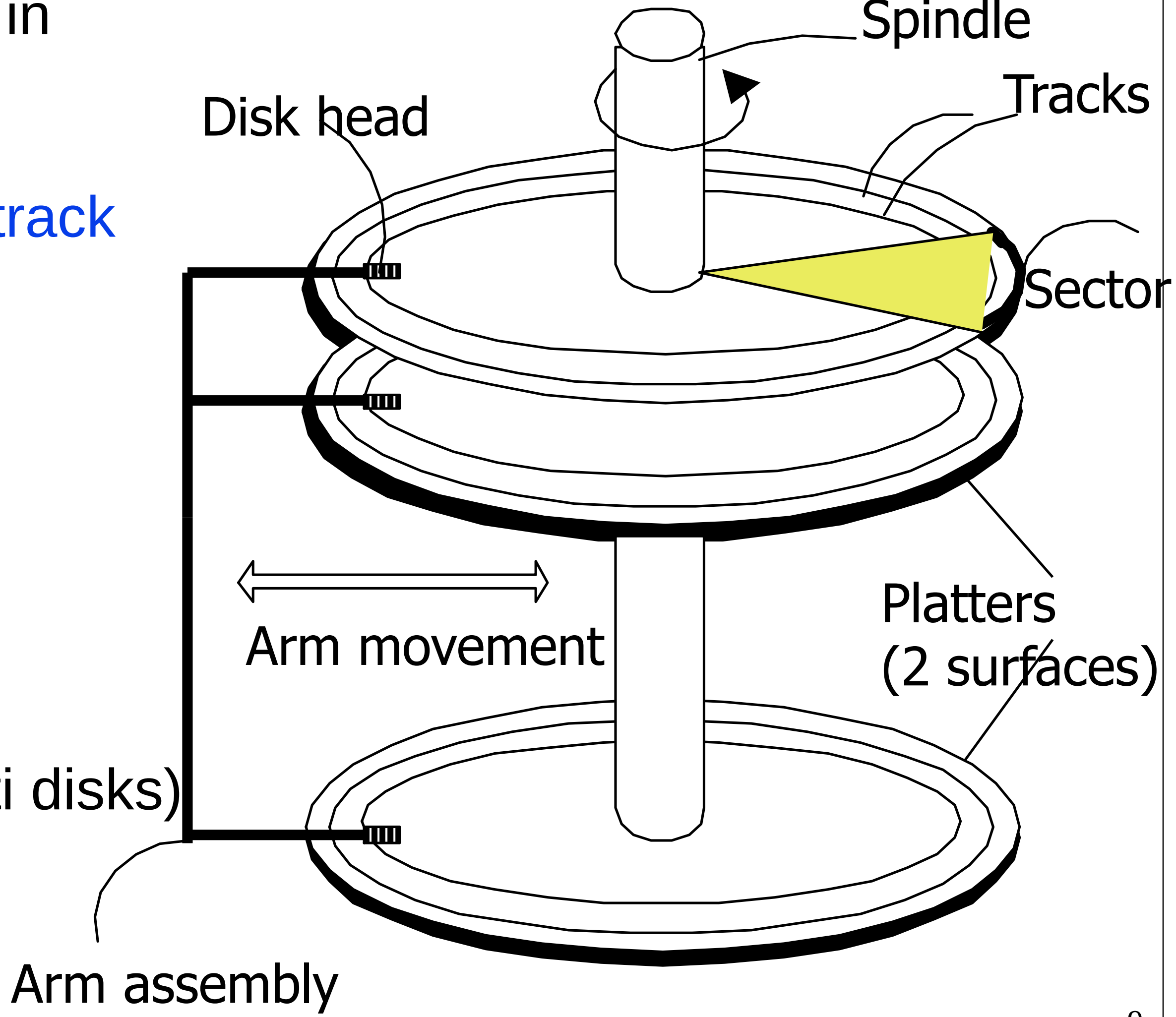
Secondary Storage

- Disks: preferred secondary storage device
  - random access is the main advantage over tapes that provide only sequential access
- Data is stored and retrieved in units called disk blocks or pages
- Unlike RAM, time to retrieve a disk block varies depending upon location on disk
  - Therefore, relative placement of blocks on disk has major impact on DBMS performance!



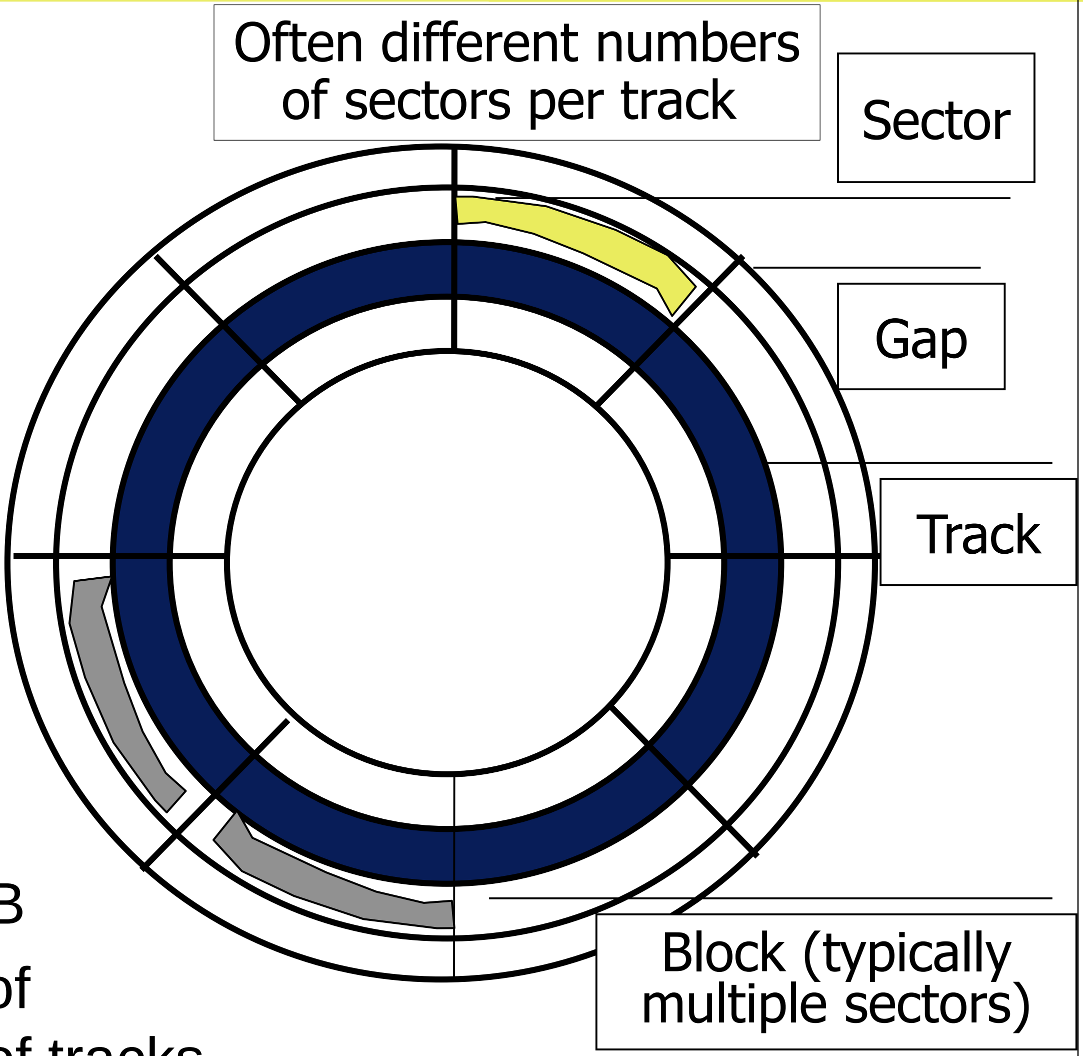
Components of a Disc

- A typical disk is made up of several platters, which are separated in tracks, organized in sectors
- The arm is moved in or out to position a head on a desired track
- Tracks under heads make a cylinder (virtual)
- Only one disk head reads a sector at a time
- Block size is a multiple of sector size (fixed)
- Block address consists of:
  - ◆Physical device # (for multi disks)
  - ◆Cylinder #
  - ◆Surface #
  - ◆Sector #

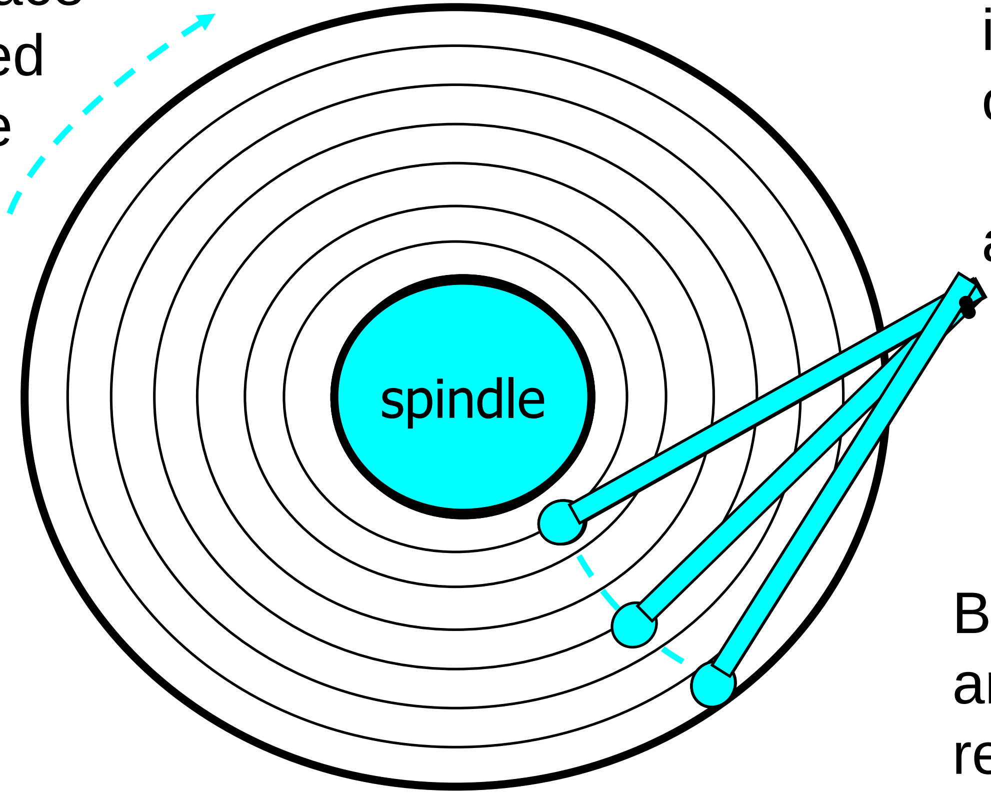


Disk Characteristics

- Diameter: 1 - 15 inches
- Cylinders: 100 - 2000
- Surfaces: 1 (CDs) - many
- Tracks/Cyl: 2 (floppies) - 30
- Sector Size: 512B - 50K
- Capacity: 360 KB (floppies) – 8TB
  - ◆Capacity of disk is a function of number of cylinders, number of tracks per cylinder, and capacity of track



Disk Operation (Single-Platter View)



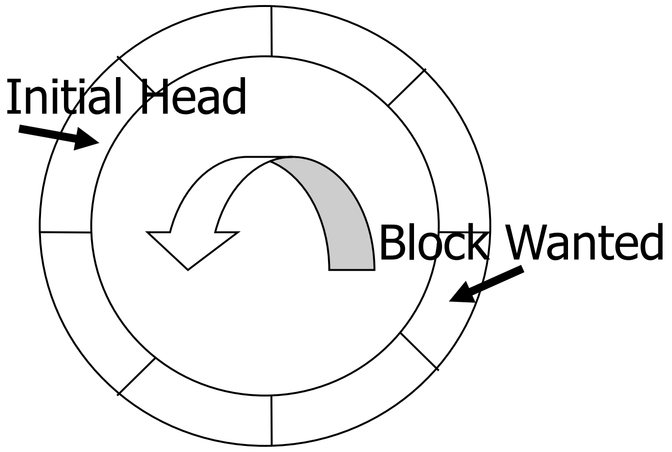
The disk surface spins at a fixed rotational rate

The read/write head is attached to the end of the arm and flies over the disk surface on a thin cushion of air

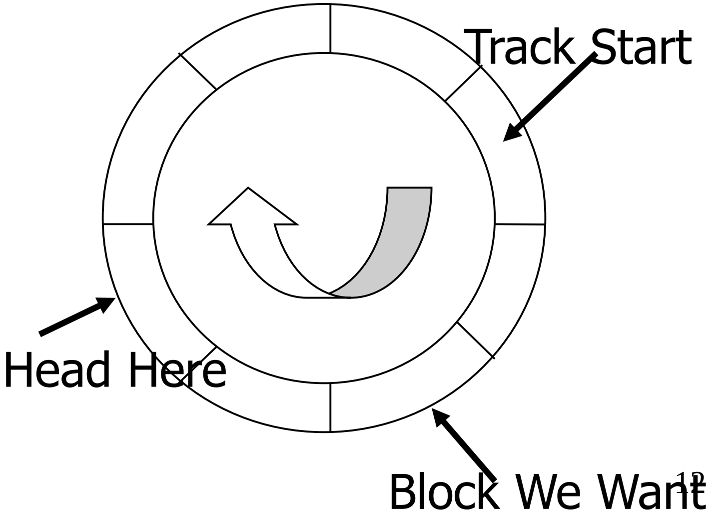
By moving radially, the arm can position the read/write head over any track

Accessing a Disk Page

- Time to access (read/write) a disk block:
  - seek time: Time it takes to reposition the arm over the correct track
    - 4 to 10 ms on typical disks
  - rotational latency: Time it takes for the sector to be accessed to appear under the head
    - 4 to 11 ms on typical disks (5400 to 15000 rpm)
  - transfer time rate: The rate at which data can be retrieved from or stored to the disk
    - 4 to 8 MB per second is typical
    - Multiple disks may share a controller, so the rate that the controller can handle is also important
- Seek time and rotational latency dominate



May have to wait for start of track before we can read desired block



Access Time for the IBM Deskstar 14GPX

- 3.5 inch hard disk, 14.4 GB capacity
- 5 platters of 3.35 GB of user data each, platters rotate at 7200/min
- average seek time 9.1 ms (min: 2.2 ms [track-to-track], max: 15.5 ms)
- average rotational delay 4.17 ms
- data transfer rate 13 MB/s

- access time$_{\text{8 KB block}}$
    - ~ 9.1 ms + 4.17 ms + 1 s/13 MB/8 KB ~ 13.87 ms

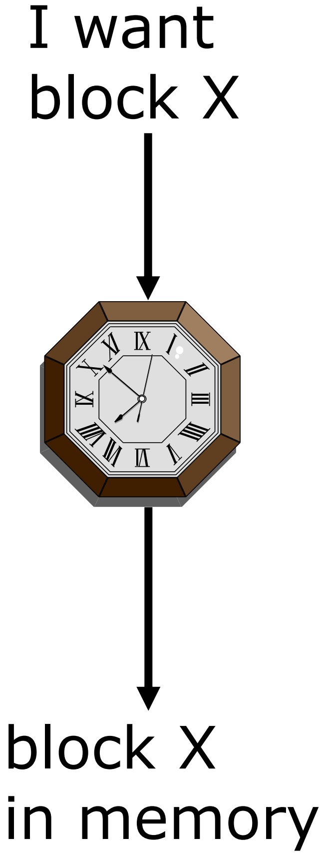
- Accessing a main memory location typically takes < 60 ns !!!

Arranging Pages on Disk

- Key to lower the duration and/or number of page transfers (I/O)
- DBMSs take the geometry and mechanics of hard disks into account
  - Current disk designs can transfer a whole track in one platter revolution, active disk head can be switched after each revolution
  - Blocks in a file should be arranged sequentially on disk (by ‘Next’) to minimize average latency i.e., reduce seek/rotation delays!
- This implies a closeness measure (relative positioning): for data records r1, r2 on disk to reduce the duration of I/Os
  - Place r1 and r2 inside the same block (single I/O operation!)
  - Place r2 inside a block adjacent to r1’s block on the same track
  - Place r2 in a block somewhere on r1’s track
  - Place r2 in a track of the same cylinder than r1’s track
  - Place r2 in a cylinder adjacent to r1’s cylinder
- For a sequential scan, pre-fetching several pages at a time is a big gain to reduce the number of I/Os (more latter)

Example

- Compute time taken to read a 2048000 byte file that is divided into 8000 256 byte records assuming the following disk characteristics?
  - average (random) seek time 18 ms
  - track-to-track seek time 5 ms
  - rotational delay 8,3 ms
  - maximum transfer rate 16,7 ms/track
  - bytes/sector 512
  - sectors/track 40
  - tracks/cylinder 11
  - tracks/surface (cylinders) 1331



Example

- 1 track contains 40*512 = 20480 bytes
- File needs 100 tracks ~10 cylinders
- Store records randomly
  - reading the file requires 4000 random accesses (Why?)
  - each access time

    = 18 (average seek) + 8.3 (rotational delay) + 0,4 (transfer one sector) = 26,7 ms
  - total access time

    = 4000 * 26,7 = 106800 ms = 106,8 s
- Store records on adjacent cylinders
  - read first cylinder

    = 18 + 8,3 + 11*16,7 = 210 ms
  - read next 9 cylinders

    = 9 * (5 + 8,3 + 11*16,7) = 1773 ms
  - total access time

    = 1983 ms = 1,983 s
- Ideally, a request for a sequence of pages should be satisfied by pages stored sequentially in disk
  - responsibility of the disk space manager

DBMS vs. OS File System

- The disk space manager is the lowest layer of the DBMS software managing space on disk

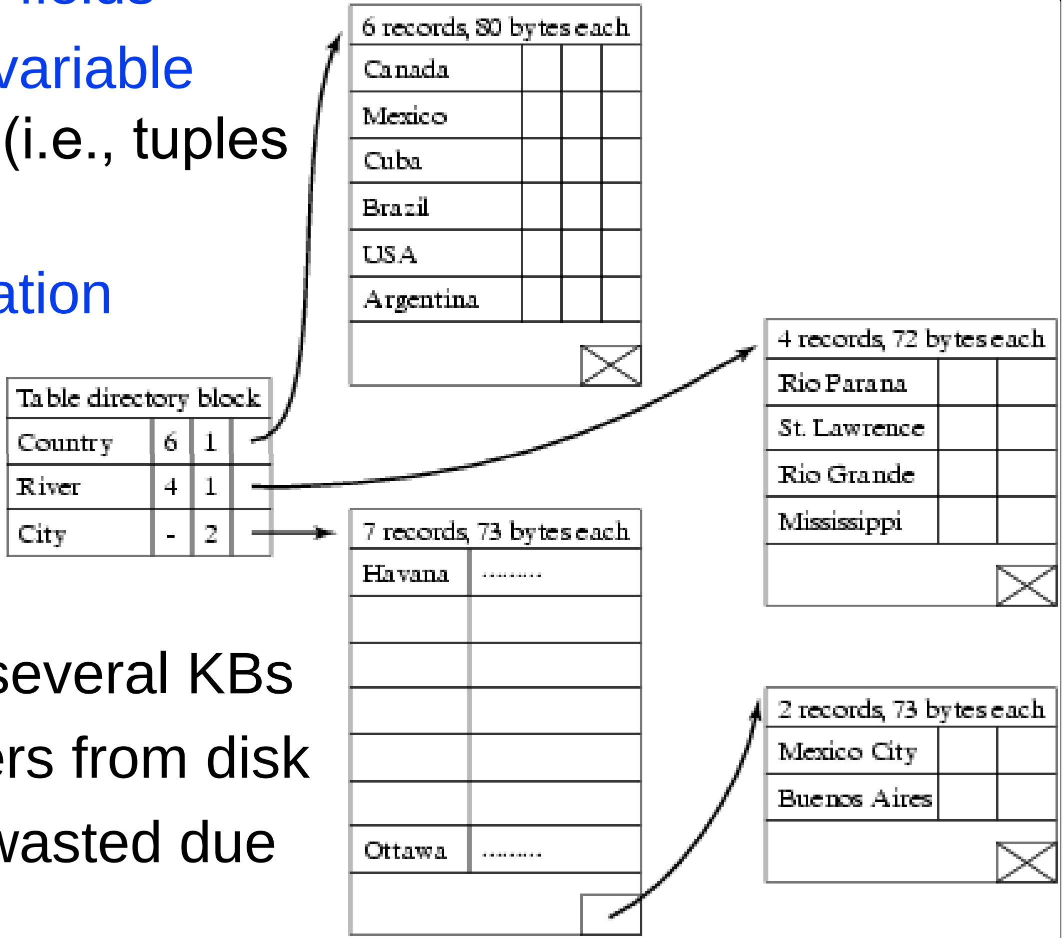
- Operating systems (OS) does disk space but also buffer management (more later)
  - why not let OS manage these tasks?

- In OS terminology a file (a document, a spreadsheet, an executable, etc.) is simply a sequence of bytes

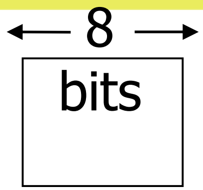
- In BDMS terminology, the term is used somewhat differently: Page or block is OK when doing I/O, but…
  - higher levels of DBMS operate on records, and files of records (i.e., databases) which can't span disks

Representing Data Elements

- Attributes are represented by fixed or variable length sequences of bytes, called “fields”
- Fields are put together in fixed or variable length collections called “records” (i.e., tuples or objects)
- A collection of records forms a relation which is stored as a collection of blocks called a “file”
- A block is a contiguous sequence of sectors from a single track
  - ◆sizes range from 512 bytes to several KBs
    - smaller blocks: more transfers from disk
    - larger blocks: more space wasted due to partially filled blocks
  - ◆typical block sizes today 4-16 KBs



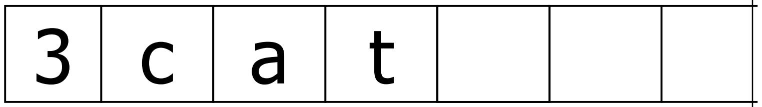
Representing Data Elements



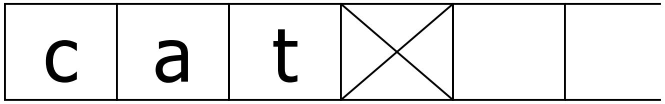
- Ultimately, all data is represented as a sequence of bytes
- Integer (short): 2 bytes (~ -32000…+32000)
  - e.g., 35 is 00000000 00100011
- Integer (long): 4 bytes (~ -2x10^9…+ 2x10^9)
- Real, floating point (SQL FLOAT) 4 or 8 bytes
  - arithmetic interpretation by hardware
- Characters: various coding schemes suggested, most popular is ASCII
  - Example: 8 bit ASCII
- Boolean: e.g., TRUE 1111 1111 FALSE 0000 0000
- Dates, e.g.: Integer: # days since Jan 1, 1900
  - 8 chars: YYYYMMDD
  - 7 chars: YYYYDDD
  - 10 chars: YYYY-MM-DD (SQL2)
- Time, e.g. Integer: seconds since midnight
  - chars: HH:MM:SS[.FF…] (SQL2)

Representing Data Elements

- Fixed-length character STRING is an array of n bytes
  - ◆If the value for the attribute is a string of length shorter than n, then the array is filled with special pad character
- Variable-length character STRING
  - ◆Allocate array of n+1 bytes
  - ◆Two common representations
    - Length plus content



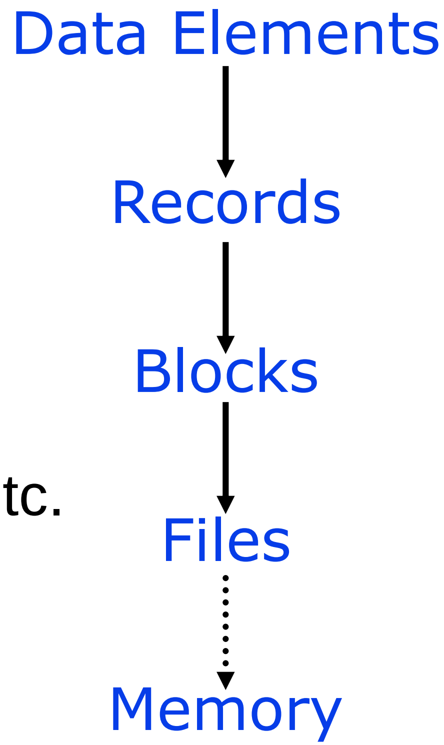
      - First byte holds number of bytes in string
      - Actual string cannot exceed n bytes (n < 255)
      - Unused bytes in the array are ignored
    - Null-terminated string



      - Allocate array of n+1 bytes
      - Fill array with characters of string, followed by null character

Records

- System catalog (more latter)
  - Information about field types common to all records in a file
  - e.g,. number of fields, field names and data types
- How to organize fields within a record ?
  - Retrieve, modify fields in a record
- Main choices:
  - Fixed vs variable length records
    - fixed or variable size fields, repeated fields, etc.
  - Fixed vs variable format records
    - follow or not a given record schema



Data Elements

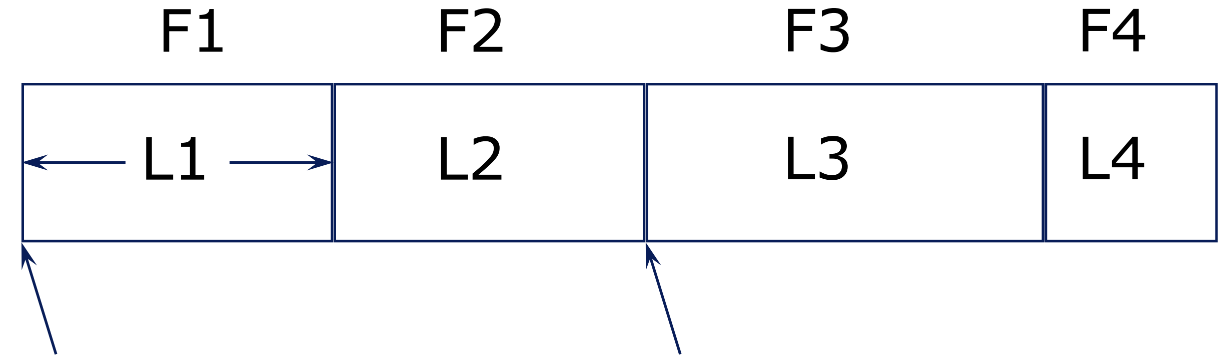
Records

Blocks

Files

Memory

Fixed Length Records



Base address (B)

Address = B+L1+L2

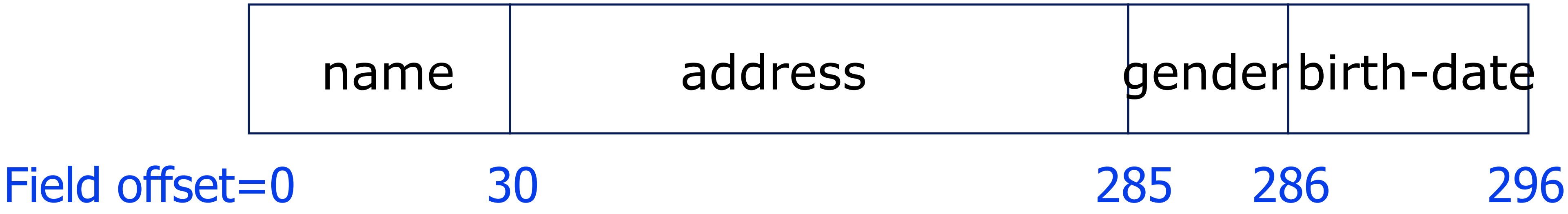
- Fixed length representation
  - Each field has fixed length
  - Number of fields is fixed
  - Store fields consecutively

- Finding `i`th field done via arithmetic

Fixed Length Records: Example

- MovieStar relation
  - ◆name: 30 byte string of characters
  - ◆address: varchar(255)
  - ◆gender: 1 byte
  - ◆birth-date: 10 byte

- Record of type MoveStar will take 30+255+1+10 = 296 bytes



name	address	gender	birth-date

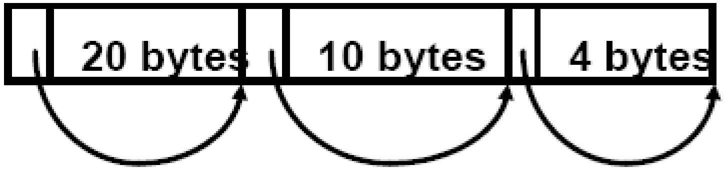
Field offset=0 30 285 286 296

Variable Length Records

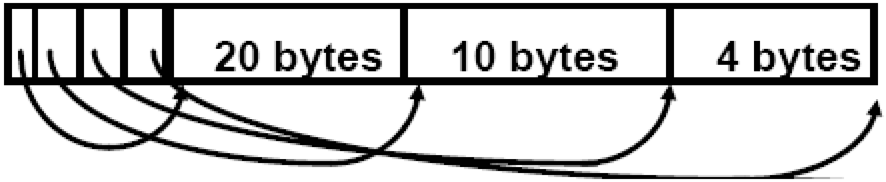
- Fields whose size varies
  - E.g.,: address field of up to 255 bytes
- Repeating fields
  - E.g.,: the set of movies in which an actor appears in
- Enormous fields
  - E.g.,: include picture of the actor -GIF image
- Variable format records
  - E.g.,: some actors also direct/produce movie

Mark end of attributes

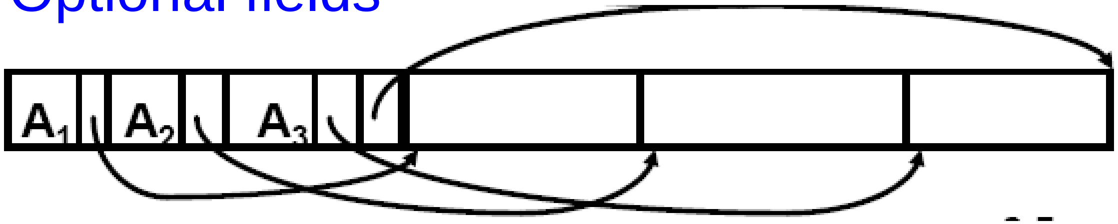
Indicator of length



Record dictionary



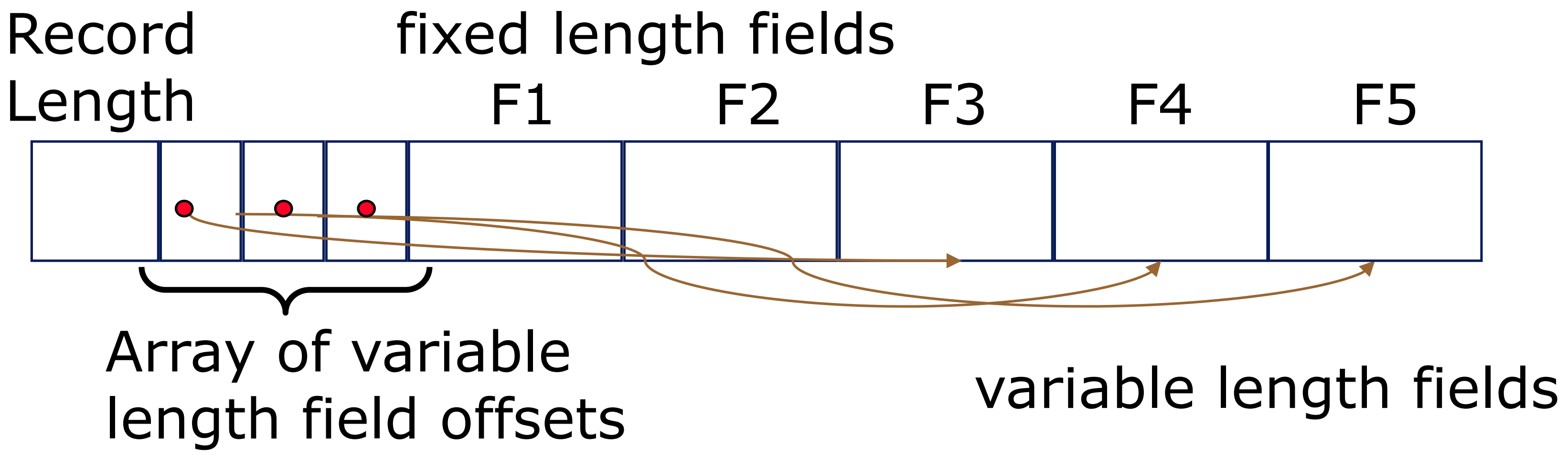
Optional fields



Combination of above

Variable Length Records: Variable Length Fields

- Put all fixed-length fields ahead of the variable-length fields (# fields is fixed):

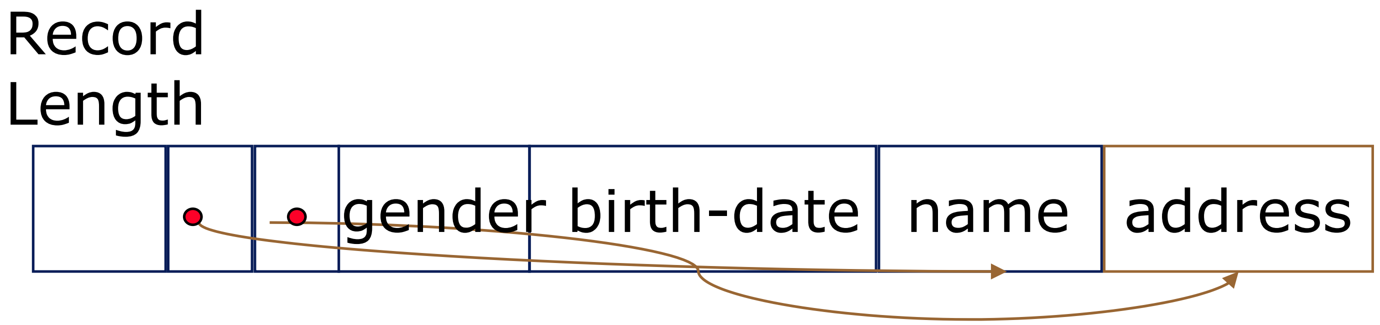


- Record header contains
  - ◆Length of record
  - ◆Pointers to (or offsets of) the beginning of all variable length fields

- Offers direct access to `i`th field, efficient storage of nulls (special don't know value); small directory overhead

Variable Length Records: Example

- Example MovieStar relation
  - name: variable length
  - address: variable length
  - gender: fixed length 4 bytes
  - birth-date: fixed length 12 bytes



Variable Length Records: Repeated Fields

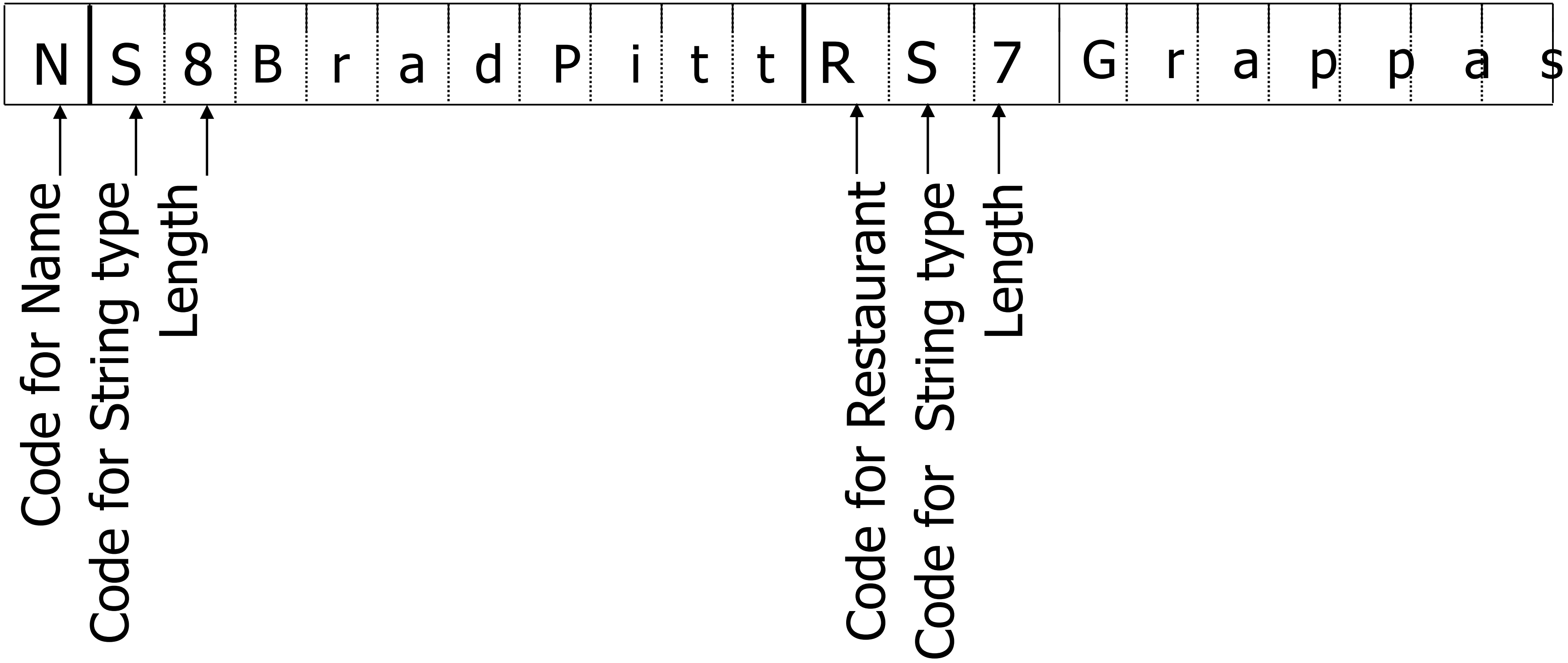
- Record contains a variable number of occurrences of a fixed-length field F
  - Group all occurrences of F together
  - Record header has a pointer to the first occurrence
  - Locate all occurrences of F as follows:
    - Let the length of field F be L
    - Add to the offset for field F all integer multiples of L, starting from 0, L, 2L, 3L etc
    - Stop when offset of the field following F is reached

Fixed vs. Variable Format Records

- Fixed format records
  - follow a given record schema that contains:
    - # fields
    - type of each field
    - order in record
    - meaning of each field
- Variable format records
  - do not follow a fixed record schema (e.g., in information integration and scientific applications)
  - Represented by a sequence of tagged fields (“self-description”)
    - Attribute or field name
    - Type of field, if it is not obvious from the field name or schema information
    - Length of field
    - Value of field

Variable Format Records: Example

- Example: MovieStar relation
  - Some movie stars have information such as movies directed, former spouses, restaurants owned etc
  - Use single byte codes for various possible field names and types



N	S	8	B	r	a	d	P	i	t	t	R	S	7	G	r	a	p	p	a	s	

Code for Name

Code for String type

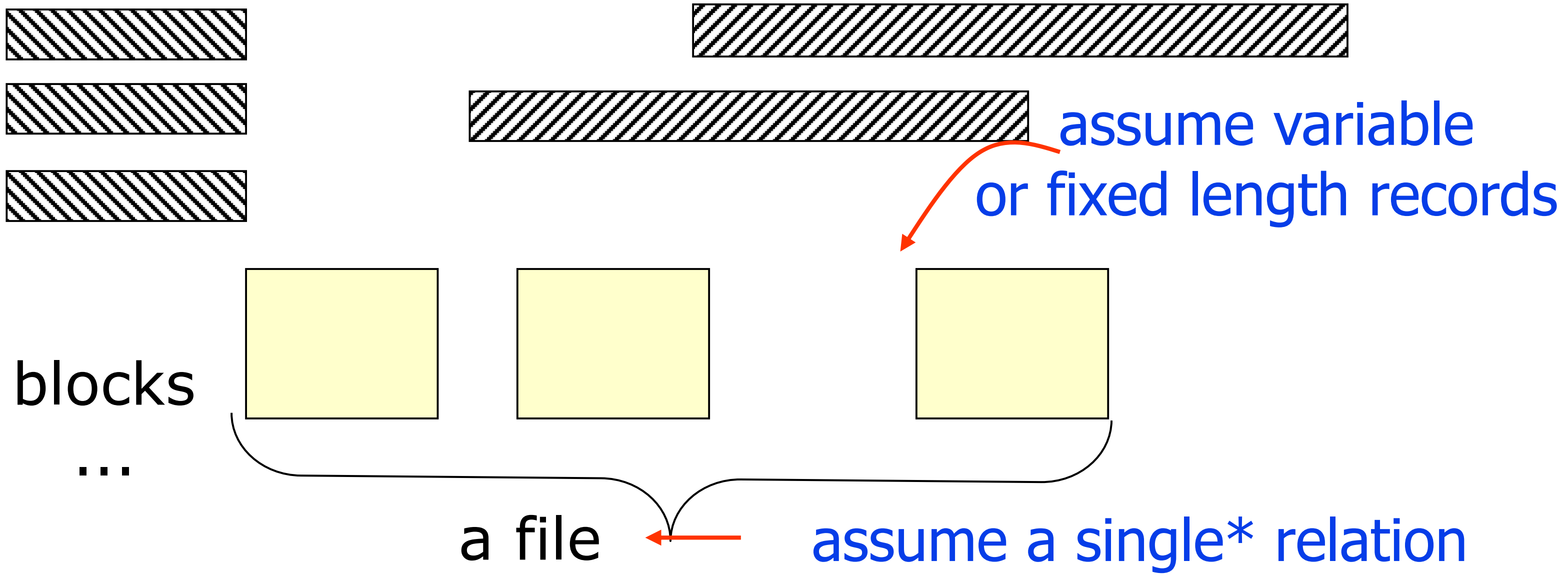
Length

Code for Restaurant

Code for String type

Length

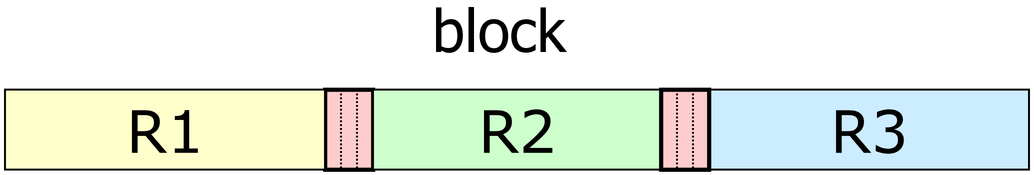
Placing Records on Blocks



- Different options for record placement:
  - Separating records (by type)
  - Spanned vs un-spanned
  - Mixed record types – clustering
  - Split records
  - Sequencing
  - Addressing

Placing Records on Blocks

❶ Separating records:

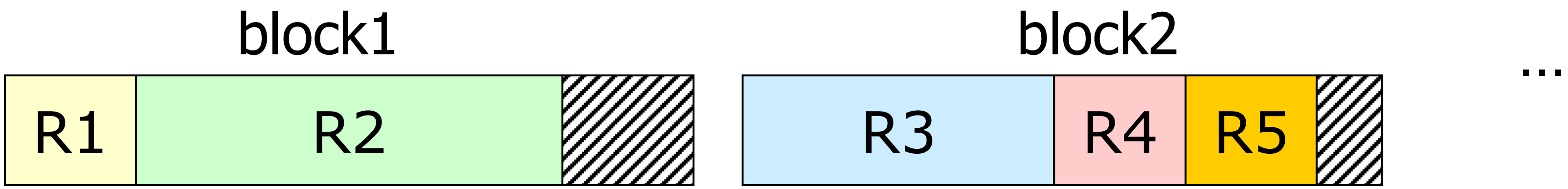


- Must use special marker or include record lengths/offsets within each record or in block header
- No need to separate if records are of fixed length

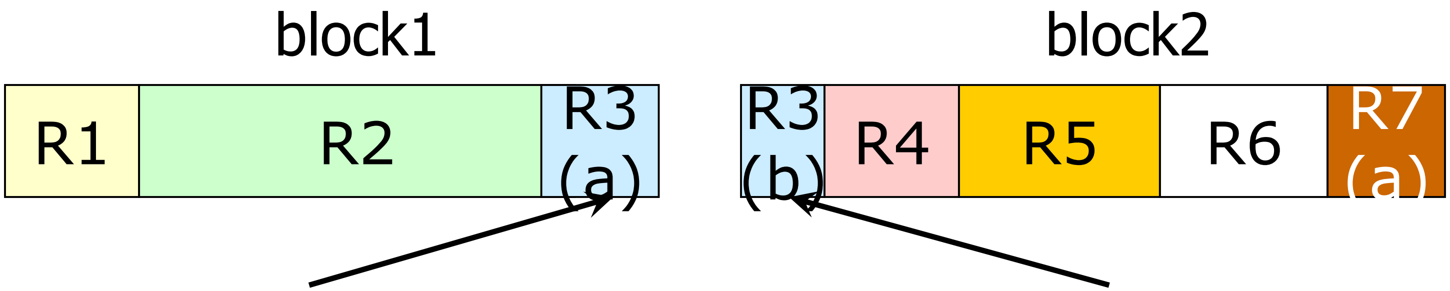
Placing Records on Blocks

❷ Spanned vs un-spanned

- Un-spanned: records are within one block but may waste space



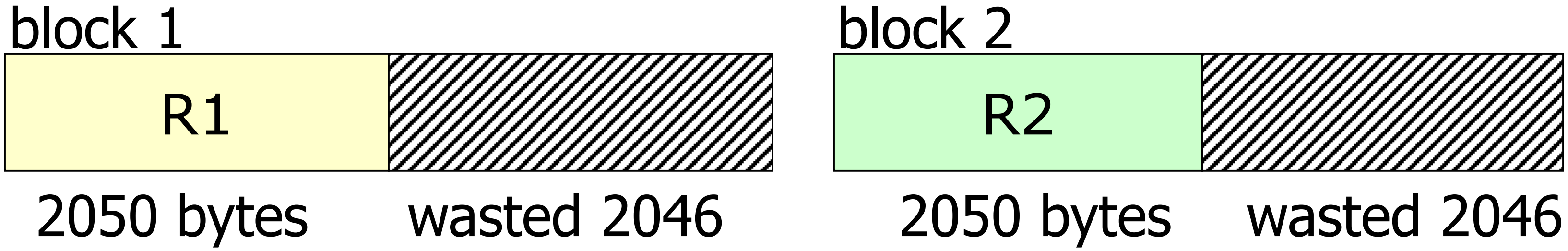
- Spanned: necessary when record size > block size



- Bits to indicate record fragment, first or last fragment, pointer to next
  - must indicate that a record is partially stored in a block and use a pointer to the rest of it;
  - must also indicate that a field is the continuation of another

Spanned vs un-Spanned: Example

- Need to store 10^6 records, each of size 2050 bytes (fixed) using block size = 4096 bytes

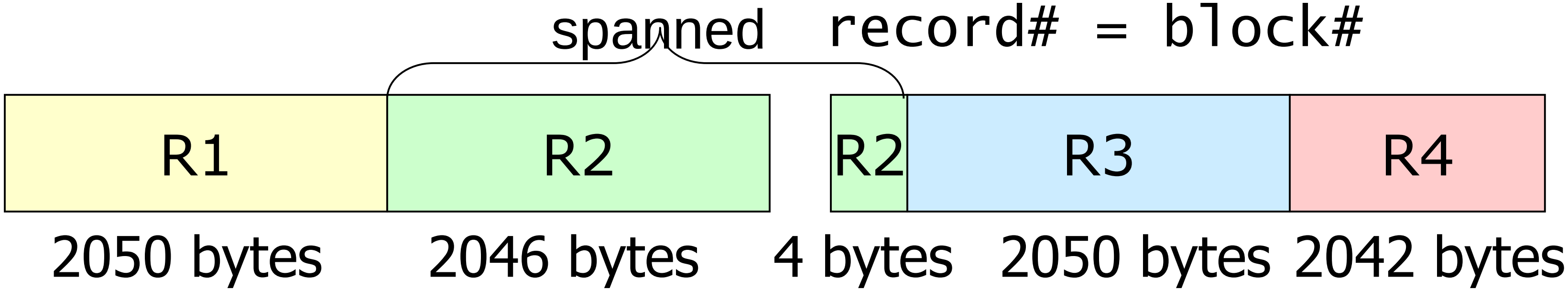


◆Total wasted = 2 x 10^9 (10^{6+3})

◆Total space = 4 x 10^9

Block space utilization = 50%

But… easy to find any record, since `record# = block#`



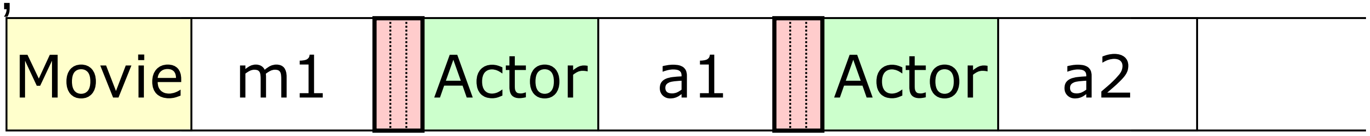
Block space utilization = 100%

◆Blocking factor is the number of logical records included in a single read or write operation aka a block

Placing Records on Blocks

❸ Mixed record types: records of different types allowed in the same block

- E.g.,



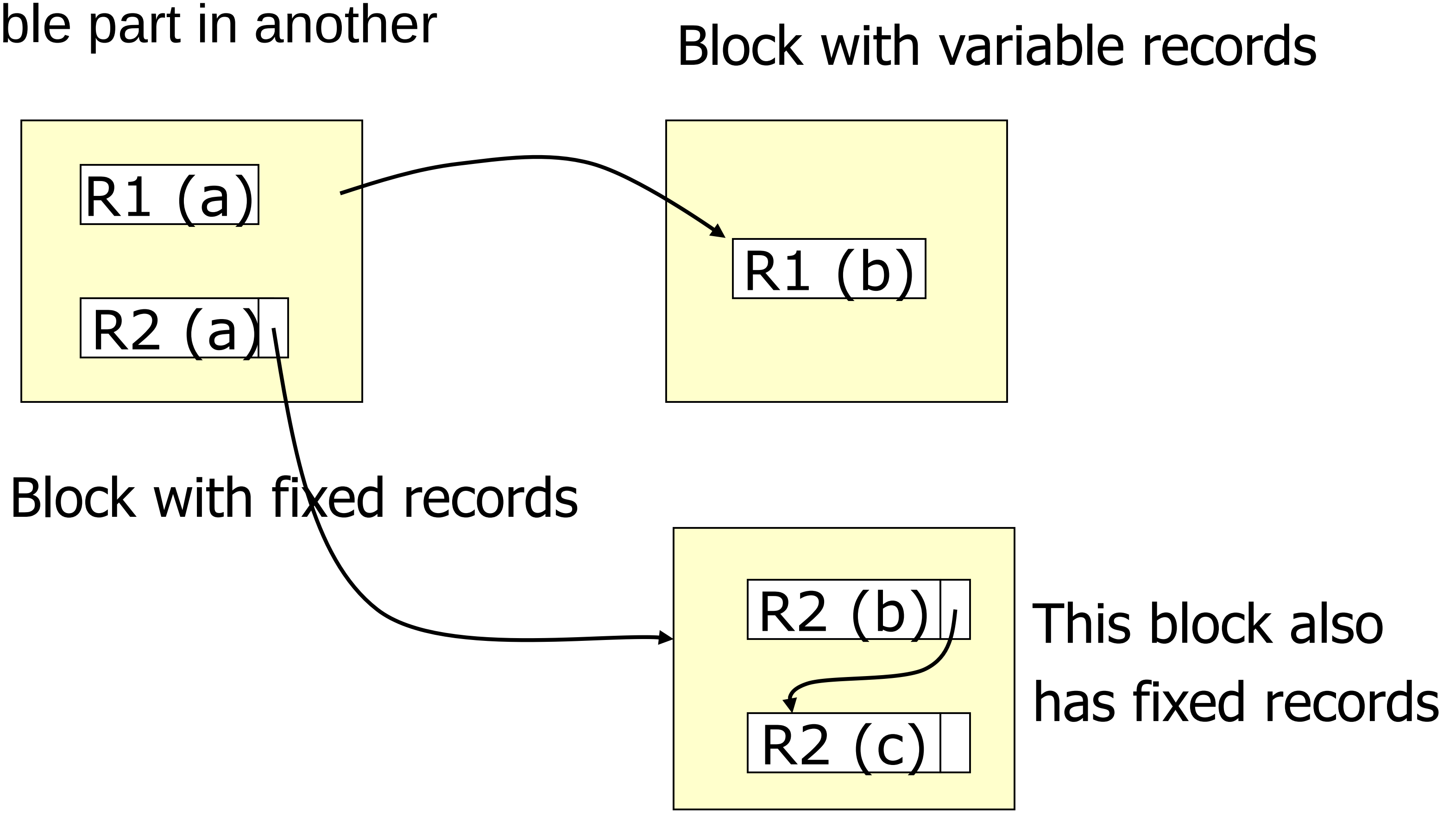
- Why would we want to mix them?
  - Clustering: Records that are frequently accessed together should be in the same block

- Compromise: don't mix them but keep them on the same disk cylinder

- Deciding whether to cluster or not presupposes knowledge about the expected types of queries

Placing Records on Blocks

- ❹ Split records: used for hybrid formats
  - ◆Fixed part in one block
  - ◆Variable part in another



Block with variable records

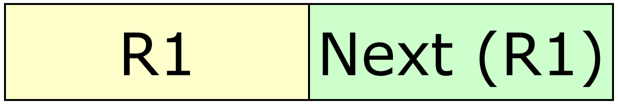
Block with fixed records

This block also has fixed records

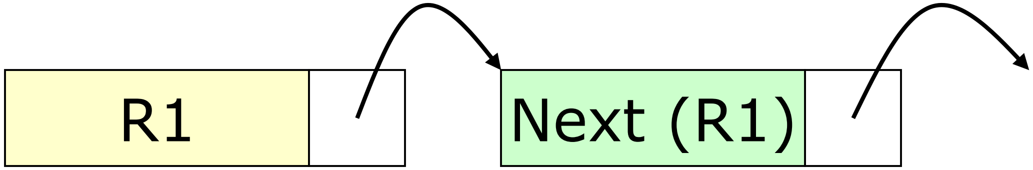
Placing Records on Blocks

❺ Sequencing: order records in file (and block) by some key value

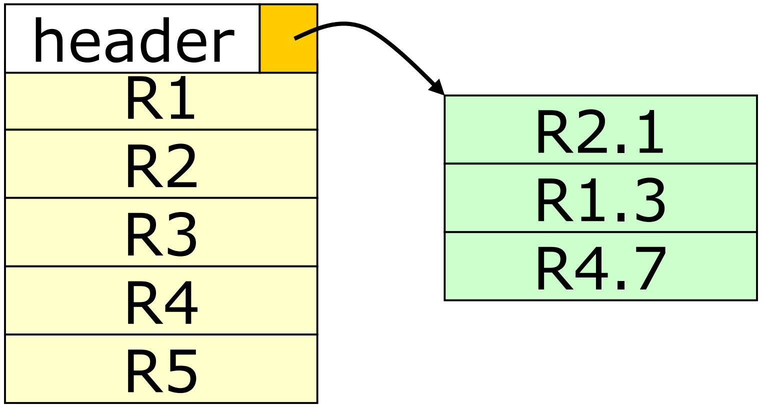
- Why do sequencing?
  - Typically to make it possible to efficiently read records in order (e.g., to do a merge-join)
- Options for sequencing:
  - Next record physically contiguous



  - Linked

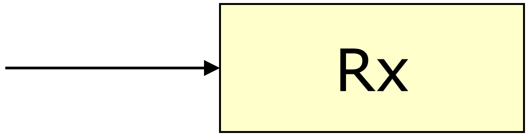


  - Overflow area

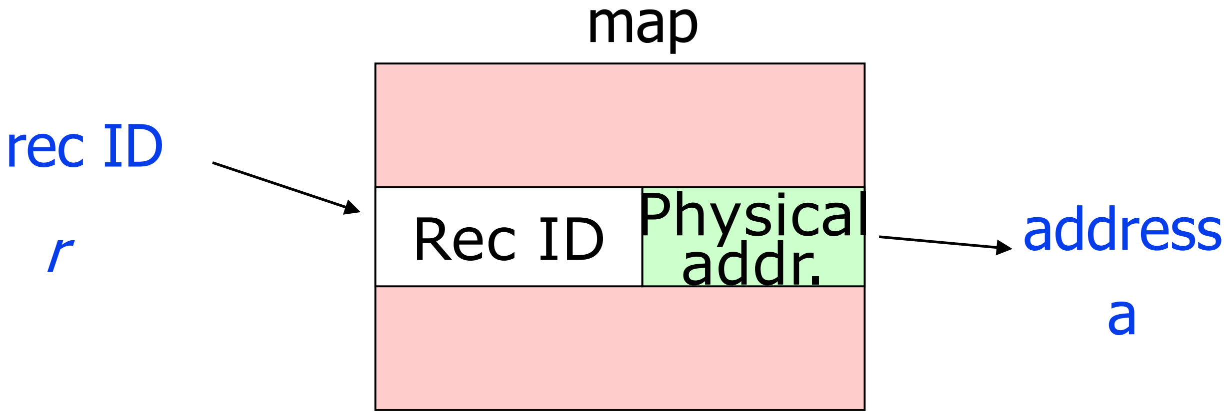


Placing Records on Blocks

❻ Addressing



- How does one refer to a record?
  - DB address: physical location on secondary storage by using record address (`<device id, cylinder#, track#, block#, record-offset in block>`)
  - Memory address: record location when loaded into (main or virtual) memory (full indirection) by using an arbitrary byte string



- Which one to use, and when?
  - Tradeoff: Flexibility to move records (for deletions, insertions) vs. cost of indirection

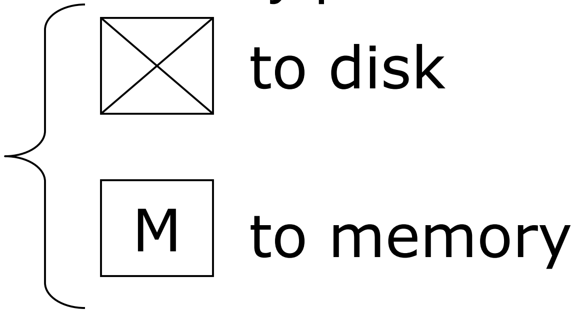
Pointer Swizzling

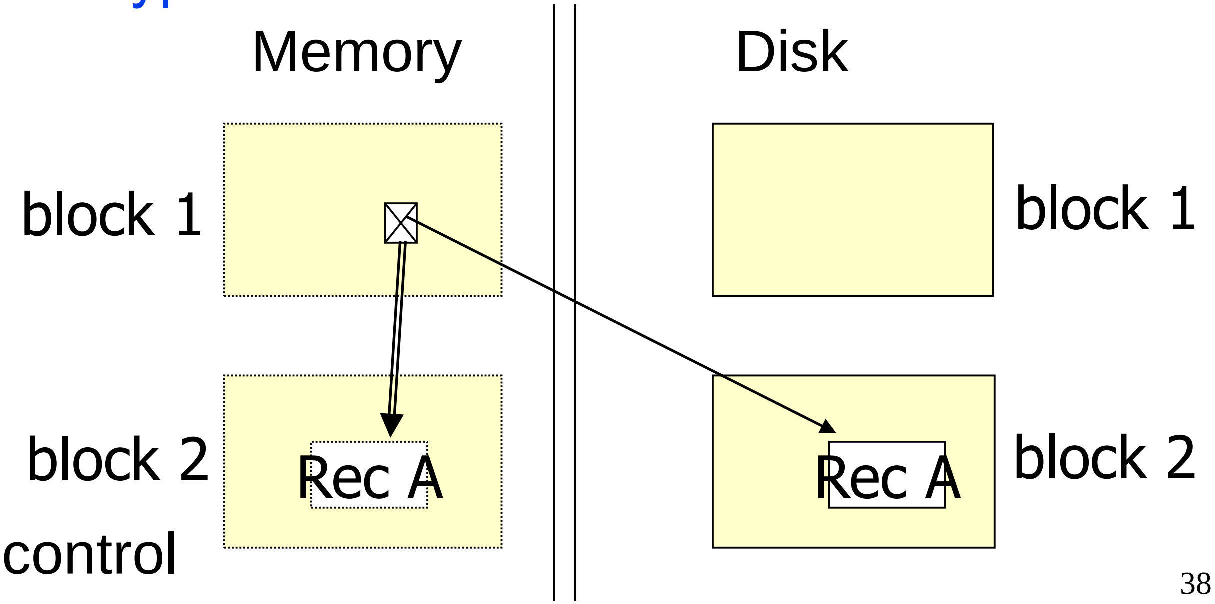
- First Option
  - For all records copied to memory use a map that contains records containing the record address

Translation Table

DB Addr	Mem Addr
Rec-A	Rec-A-inMem

- Another Option
  - In memory pointers - need “type” bit





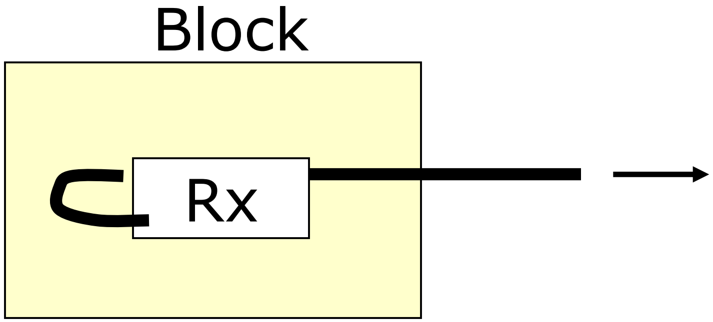
- Swizzling
  - Automatic ("eager")
  - On-demand ("lazy")
  - No swizzling / program control

Record Deletion

- When a record is deleted the following options are available:
  - immediately reclaim space
  - mark as "deleted" (may need a chain of deleted records for reuse)
    - Need a way to mark (special characters, deleted field, in map)

- Many tradeoffs to consider:
  - How expensive is to move valid record to free space for immediate reclaim?
  - How much space is wasted?

- Problem with dangling pointers. Solutions?
  ❶ Do not worry about it
  ❷ Use a special mark (tombstone) in old location or in map

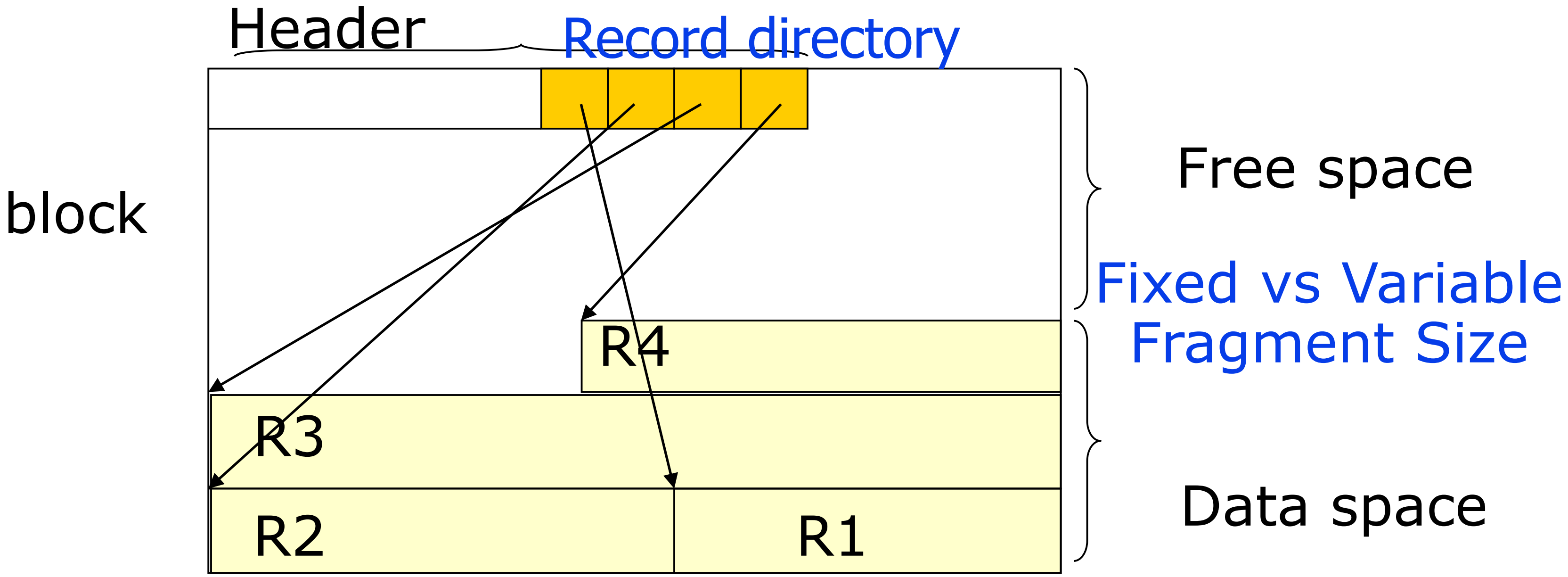


Record Insertion

- If records are not in sequence, insert new record at end of file (last block) or in deleted slot
  - Not as easy if records are of variable size

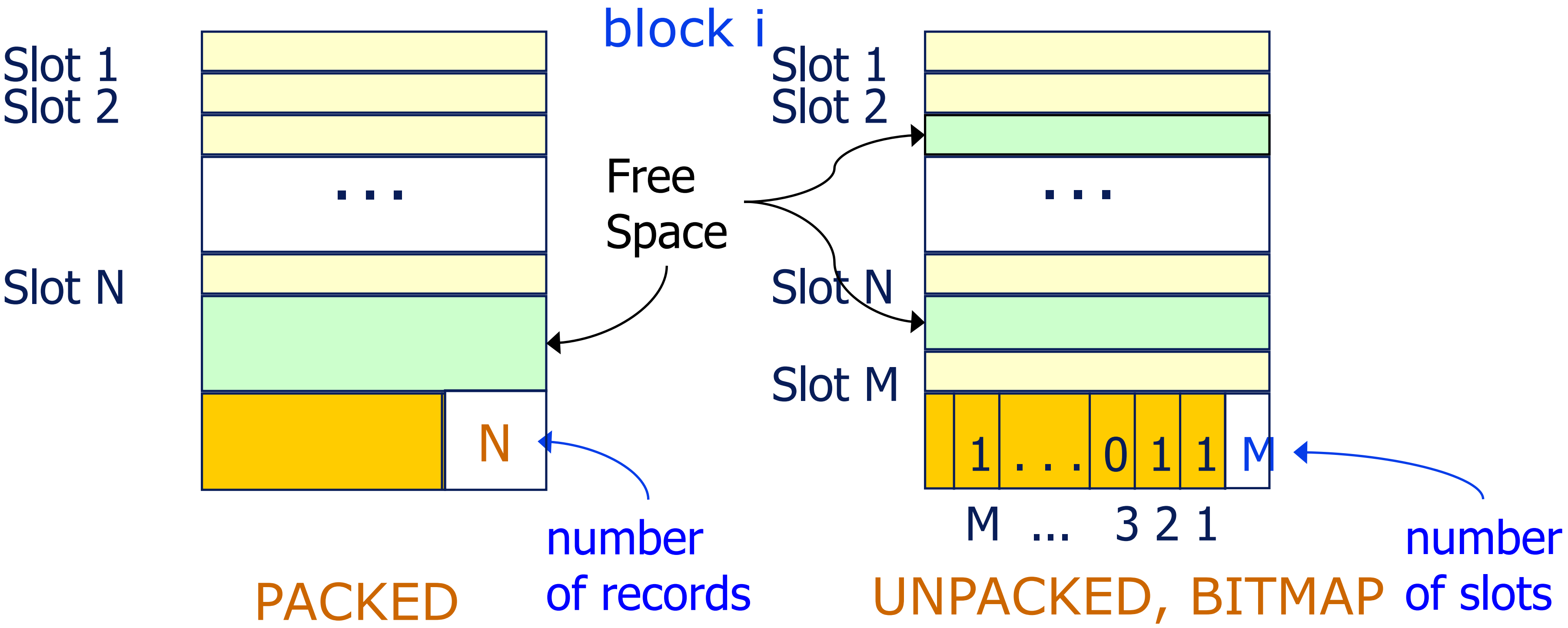
- If records are in sequence, use nearby free space or overflow area
- But…
  - How much free space to leave in each block, track, cylinder?
  - How often do I reorganize file + overflow?

Block Header



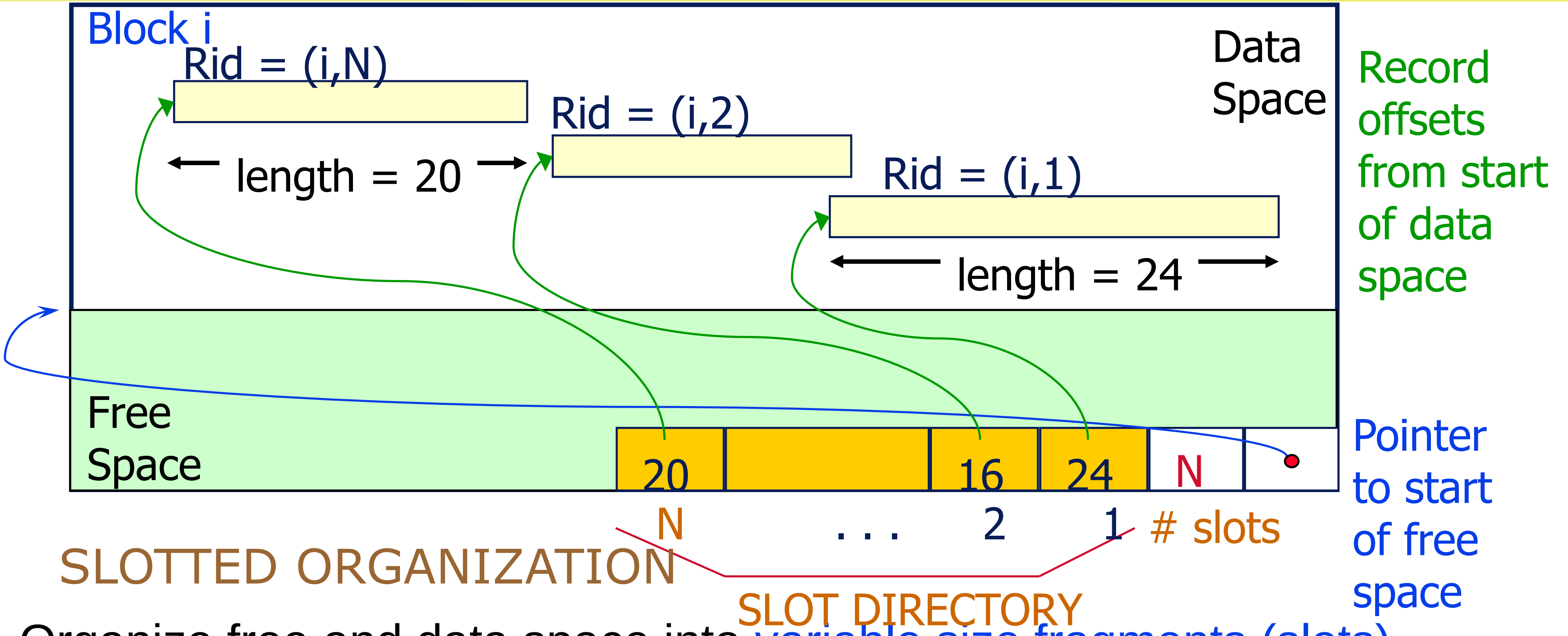
- Block (Page) is a collection of slots each containing a record
- Header data describing block may contain:
  - ◆File ID (or RELATION or DB ID); the ID of this block – Record directory; Pointer to free space
  - ◆Type of block (e.g., contains records of type 4; is overflow, …)
  - ◆Pointer to other blocks "like it" (say, if part of an index structure)
  - ◆Timestamp ...

Block (Page) Formats: Fixed Length Records



- Organize free and data space into fixed size fragments (slots)
- Packed: moving records for free space management (to keep records contiguous) or for sorting them, changes the `rid <page#, slot#>`
  - ◆may not be acceptable
- Bitmap: If slot `i` is free the `i`th bit of the header is set to 0, otherwise 1
- In both cases we have positioned the page header at the end of its page

Block (Page) Formats: Variable Length Records



- Organize free and data space into variable size fragments (slots)
  - ◆To get rid of holes produced by deletions compact the remaining records to maintain a contiguous area of free space on the page
- Slotted: we can move records on page without changing `rid <page#, record_index>` (indirection); so, attractive for fixed-length records too
  - ◆Record (slot) directory entries: `<record-offset,record-length>`

File Structure

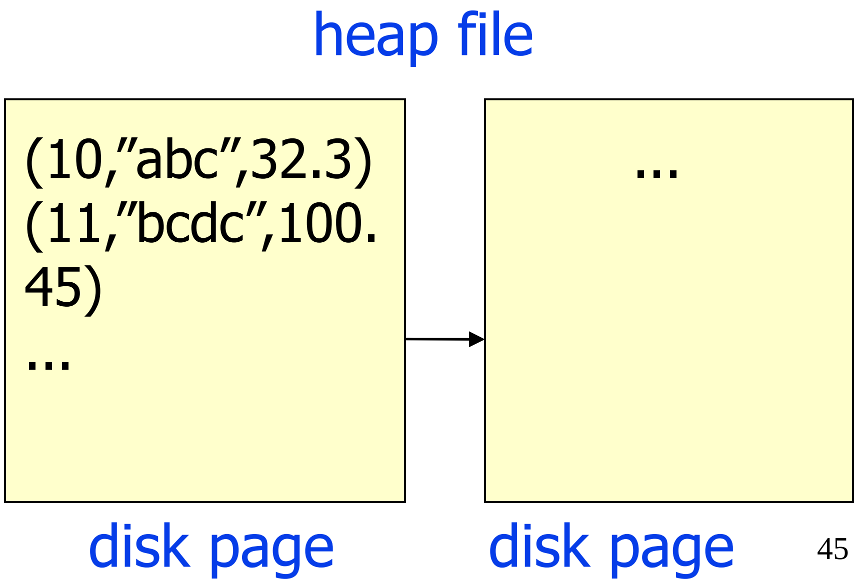
- File
  - Collection of pages (blocks), each containing a collection of records
- File structure must support
  - insert / delete / modify record
  - read a particular record (specified using `rid`)
  - scan all records (possibly with some conditions on the records to be retrieved)
- Many alternatives exist, each good for some situations, and not so good in others:
  - Heap files: Suitable when typical access is a file scan retrieving all records
  - Sorted (Sequential) Files: Best for retrieval in search key order, or only a `range’ of records is needed (more latter)
  - Hashed Files: Good for equality selections (more latter)

Unordered (Heap) Files

- Simplest file structure contains records in no particular order
- As file grows and shrinks, disk pages are allocated and de-allocated
- To support record level operations, we must:
  - keep track of the *pages* in a file
  - keep track of *free space* on pages
  - keep track of the *records* on a page
- There are many alternatives for keeping track of this
  - We'll consider two

table

A1	A2	A3
10	"abc"	32.3
11	"bcdc"	100.45
…	…	…



heap file

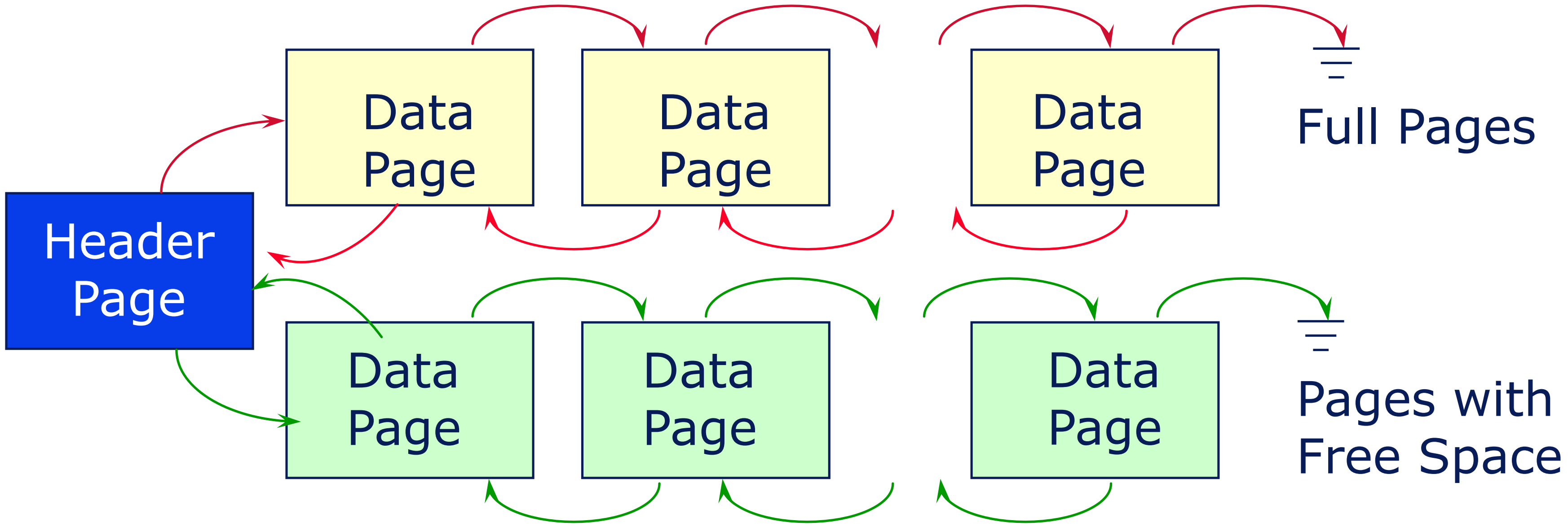
(10,"abc",32.3)
(11,"bcdc",100.45)
…

…

disk page

disk page

Heap File Implemented as a List

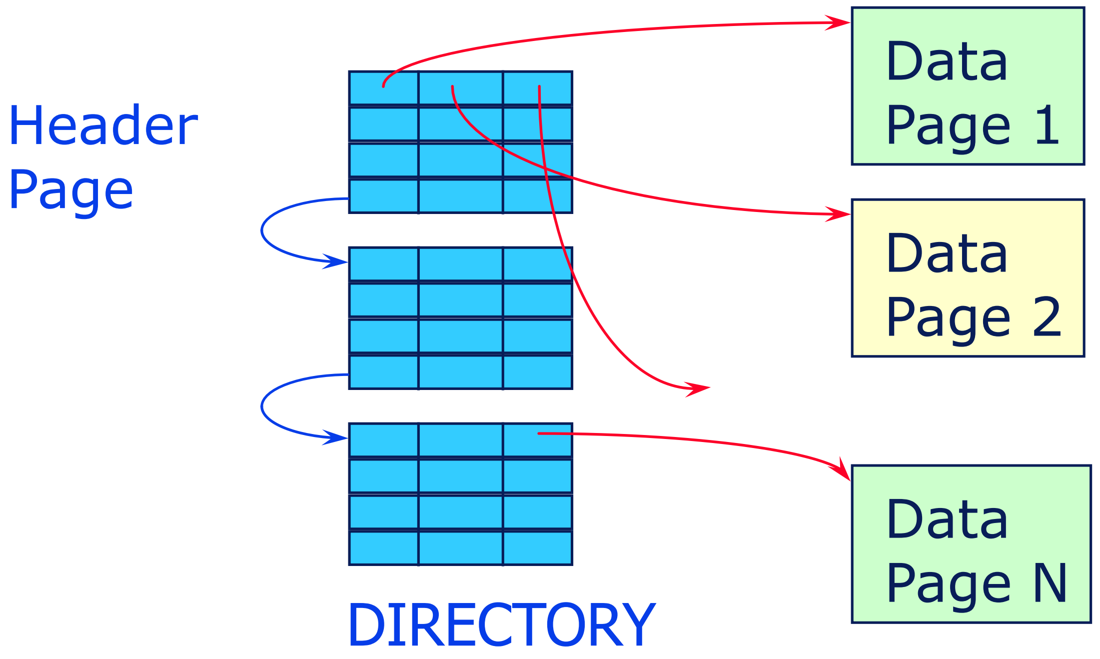


- DBMS allocates a free page (the file header) and writes an appropriate entry <`heapFileName, headerPageID`> to a known location on disk;
  - Database “catalog”
- Header page is initialized to point to two doubly linked lists of page ids
  - Initially, both lists are empty
- Scan several pages on free list before finding one with enough free space to insert a record

Heap File Implemented as a List

- For insertRecord(f , r):
  - try to find a page p in the free list with free space > |r |; should this fail, ask the disk space manager to allocate a new page p
  - record r is written to page p
  - since generally |r | << |p|, p will belong to the list of pages with free space
  - a unique rid for r is computed and returned to the caller
- For openScan(f):
  - both page lists have to be traversed
- A call to deleteRecord(f , rid)
  - may result in moving the containing page from full to free page list,
  - or even lead to page deallocation if the page is completely free after deletion
- Finding a page with sufficient free space is an important problem to solve
  - How does the heap file structure support this operation? (How many pages of a file do you expect to be in the list of free pages?)

Heap File Using a Page Directory



- DBMS maintains information on the first directory page for each heap file
  - ◆Each entry in a directory page can include the number of free bytes available on the page <`PageID`, `nfree`>
- The directory is a collection of pages; linked list (LL) implementation is just one alternative
  - ◆Much smaller than LL of all HF pages! |page directory| << |data pages|

Data Dictionary

- Data dictionary (also called system "catalog") stores metadata: that is, data about data, such as
  - information about relations
    - names of relations
    - names and types of attributes
    - physical file organization information
    - statistical data such as number of tuples in each relation
  - integrity constraints
  - view definitions
  - user and accounting information
  - information about indices
- Catalog structure: can use either
  - specialized data structures designed for efficient access
  - a set of relations, with existing system features used to ensure efficient access
    - the latter alternative is usually preferred

Disk Space Manager

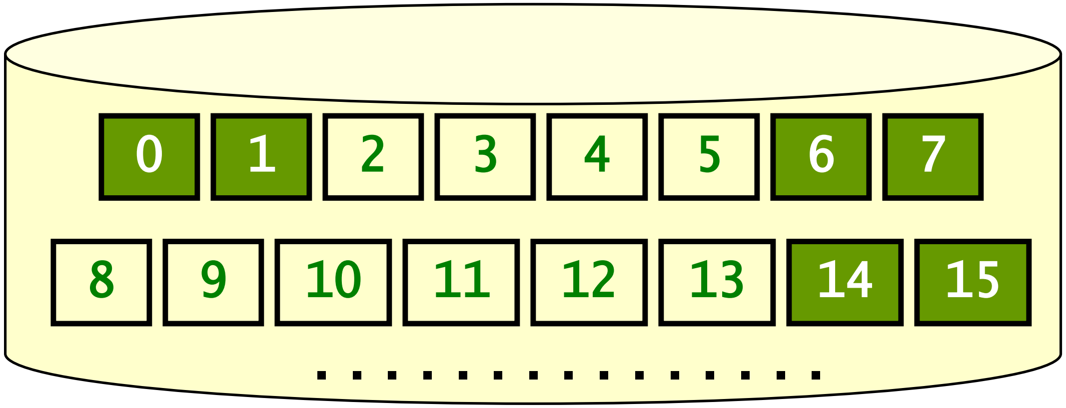
- It is the lowest DBMS software layer supporting the concept of page as a unit of data: accessing one disk block is one seek
  - Many files will be stored on a single disk
- Higher DBMS software levels call upon this layer to:
  - allocate/de-allocate a page
  - read/write a page
- Best if a request for a sequence of pages is satisfied by pages for a file stored as a contiguous sequence of blocks on disk!
  - Higher levels don’t know how this is done, or how free space is managed
  - Though they may assume sequential access for files!
    - Hence disk space manager should do a decent job
      - Disk space is effectively utilized
      - Files can be quickly accessed

Disk Space Management

- Two issues:
  - ◆Management of free space in a disk
    - System maintains a list of free pages (blocks)
      - keep a pointer to the first free block in a known location on disk
      - when a block is no longer needed, append/prepend this block to the free block list for future use
      - next pointers may be stored in disk blocks themselves
    - Implemented as bitmaps or linked lists
      - reserve a block whose bytes are interpreted bit-wise (bit n = 0: block n is free)
      - toggle bit n whenever block n is (de-)allocated
  - ◆Allocation of free space to files
    - Granularity of allocation (blocks, clusters, extents)
    - Allocation methods (contiguous, linked)
      - Subsequent deallocations and new allocations however will, in general, create holes

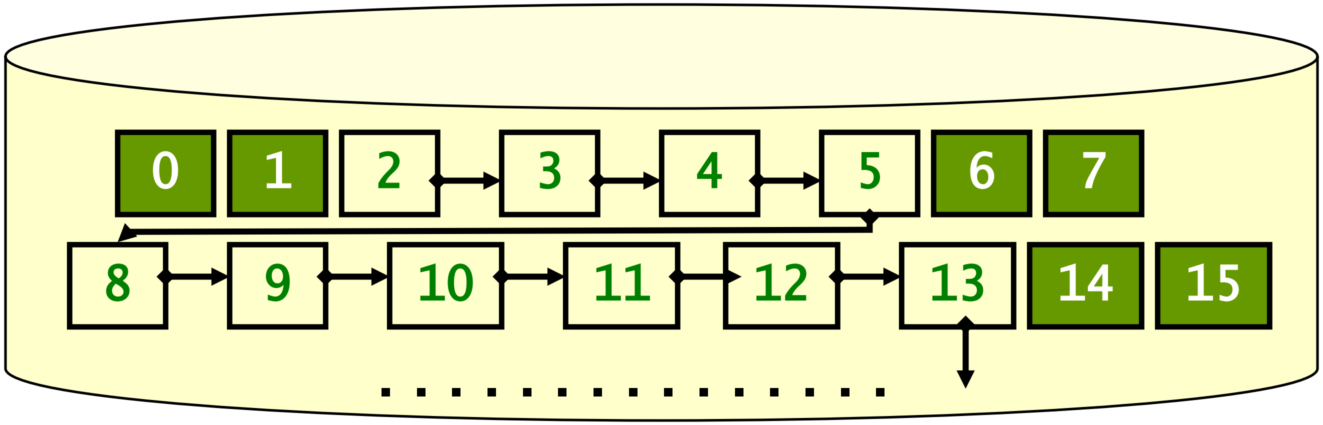
Bitmap of Free Blocks (Pages) on Disk

- A bitmap is kept for all blocks in the disk
  - ◆Each block is represented by one bit
    - If a block is free, its corresponding bit is 0
    - If a block is in use, its corresponding bit is 1
  - ◆To allocate space, scan the bitmap for 0s
    - Free block bitmaps allow for fast identification of contiguous sequences of free blocks
- Example: Consider a disk whose blocks `2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17,` etc. are free
  - ◆The bitmap would be 110000110000001...



Link Lists of Free Blocks (Pages) on Disk

- Link list of all the free blocks
  - Each free block points to the next free block
  - DBMS maintains a free space list head (FSLH) to the first free block
- To allocate space
  - look up FSLH
  - follow the pointers
  - reset the FSLH



Contiguous Block Allocation

- Each file occupies a set of contiguous block addresses

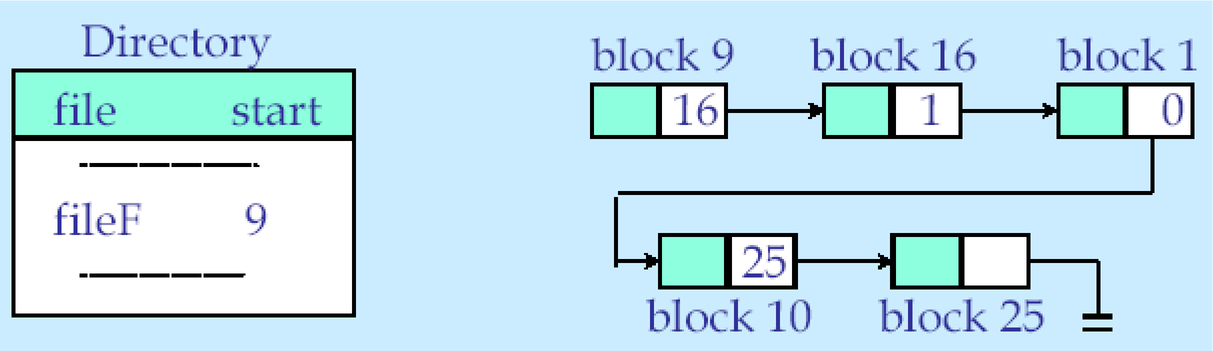
- Efficient access
  - ◆At most only one track-to-track movement for sequential accesses
  - ◆Minimal head-movement (seek time) for random accesses

- External fragmentation
  - ◆Only contiguous blocks can be allocated
  - ◆Limited file growth/shrunk

- Periodic compaction of disk space
  - ◆Disk is reorganized to group all free space as a single chunk
  - ◆Prevent poor space utilization of disk space
  - ◆Cost: time

Linked Block Allocation

- Each file is a linked list of disk blocks
  - ◆Blocks may be scattered anywhere on disk
  - ◆A directory contains a pointer to the first block of a file
- Example: A file of 5 blocks starts at block 9, continues at blocks 16, 1, 10 and 25
  - ◆Each block contains a pointer to the next block



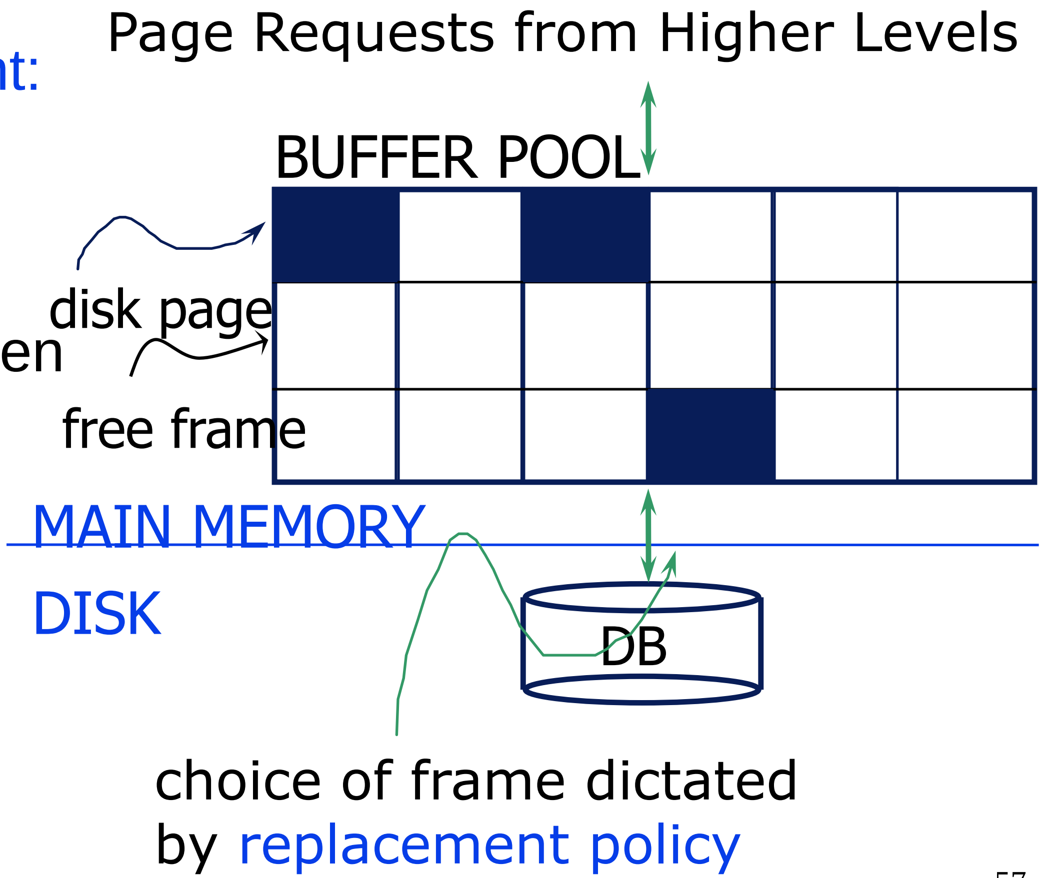
- No external fragmentation
  - ◆Facilitates file growth/shrunk
- Poor random access performance

Buffer Manager

- The buffer manager enables the higher levels of the DBMS to assume that the needed data is in main memory
  - ◆Manages buffer pool: the pool provides space (called frames) for a limited number of pages from disk
- If the block is already in the buffer:
  - ◆the requesting program is given the address of the block in main memory
- Otherwise:
  - ❶The buffer manager allocates space in the buffer for the block
    - discard some other block, if necessary for space
    - the block that is thrown out is written back to disk if it was modified since the last time it was fetched
  - ❷The buffer manager reads the block from the disk to the buffer
    - Passes the address of the block in main memory to requester

Buffer Manager

- Buffer pool information table contains tuples of the form: *<frame#, page#, pin_count, dirty>*
- If requested page is not in pool:
    - ◆Choose a frame for replacement: only "unpinned" pages are candidates!
    - ◆If frame is "dirty", write it to disk
    - ◆Read requested page into chosen frame, pin it and return address
- Page in pool may be requested several times:
    - ◆a pin_count is used, to pin a page, pin_count++
    - ◆a page is a candidate for replacement iff pin count == 0 ("unpinned")



Buffer Manager

- Sometimes it is useful to pin blocks to keep them available during an operation and not let the replacement strategy touch them
  - a pinned block is a memory block that is not allowed to be written back to disk
- Requestor of page must eventually unpin it, and indicate whether page has been modified:
  - dirty bit is used for this
- Buffer frame is chosen for replacement by an appropriate policy:
  - Least-recently-used (LRU), Most-recently-used (MRU), Clock, First In First Out (FIFO), Random, etc.
- Replacement policy can have big impact on the # of I/O's
  - If requests can be predicted i.e., access patterns, (e.g., sequential scans) pages can be pre-fetched (several pages at a time)
- Concurrency control and recovery may entail additional I/O (forced output) when a frame is chosen for replacement
  - Write-Ahead Log protocol

Least Recently Used Replacement Policy

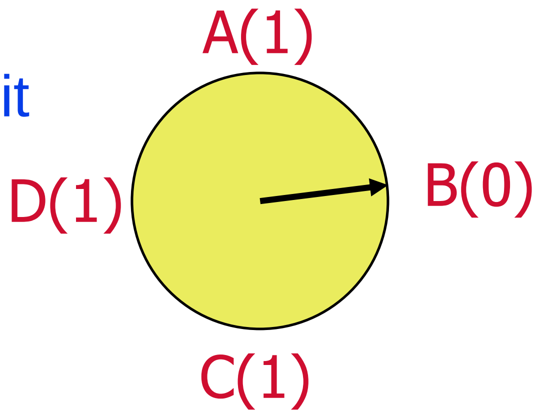
- LRU Strategy:
  - Buffer blocks not used for a long time are less likely to be accessed
  - Past usage often predicts future
- Rule: Throw out the block that has not been read or written for the longest time
  - for each page in buffer pool, keep track of time when last unpinned
  - replace the frame which has the oldest (earliest) time
  - very common policy: intuitive and simple
    - Works well for repeated accesses to popular pages
- Problem: Sequential flooding
  - LRU + repeated sequential scans of the same table (e.g., nested-loop joins)
  - `#buffer frames < #file pages` means each page request causes an I/O
  - Is MRU better in this scenario?

Most Recently Used Replacement Policy

- Toss-immediate Strategy:
    - If iterating through table, then most recent buffer block will be unused the longest (works very well with joins)
- Rule: Free the space occupied by a block as soon as the final tuple of that block has been processed
    - System must pin the block currently being processed
    - After the final tuple of that block has been processed, the block is "unpinned", and it becomes the most recently used block
- Buffer manager can use statistical information regarding the probability that a request will reference a particular relation
    - E.g. the data dictionary is frequently accessed
        - Heuristic: always keep data dictionary blocks pinned in main memory
    - if several pages are available for overwrite; choose the one that has the lowest number of recent access requests to replace

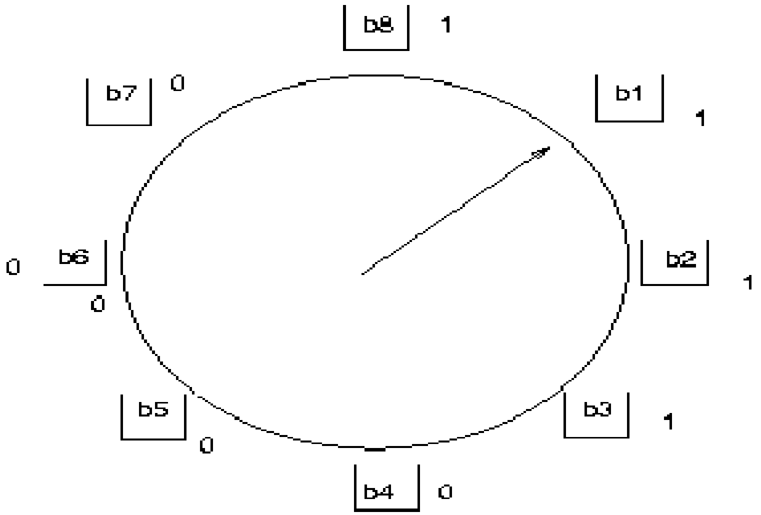
“Clock” Replacement Policy

- “Clock” Strategy:
  - An approximation of LRU
  - Arrange frames into a cycle (`current++`), store one reference bit (`ref_bit`) per frame
    - Can think of this as the second chance bit
  - When pin_count reduces to 0, turn on reference bit
  - When replacement necessary

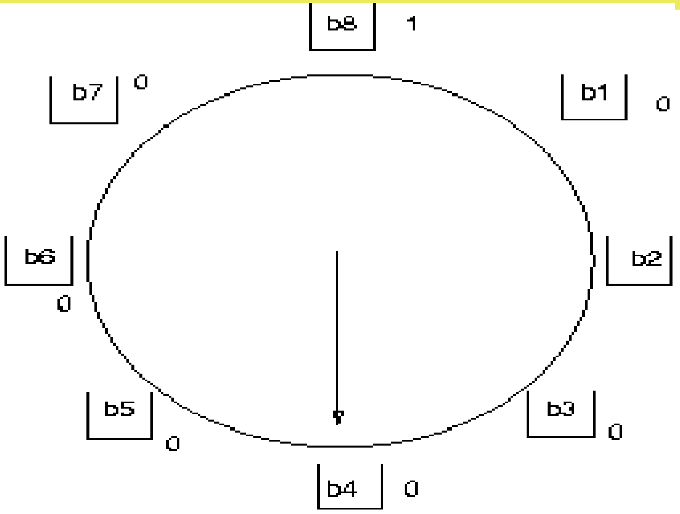


```
do for each page in cycle {
        if (pin_count == 0 && ref_bit is on)
              turn off ref_bit;
        else if (pin_count == 0 && ref_bit is off)
              choose this page for replacement;
  } until a page is chosen;
```

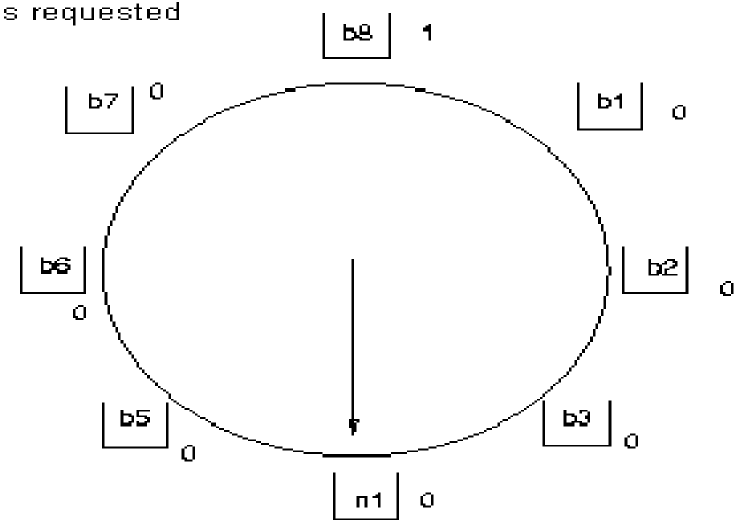
“Clock” Replacement Policy



before FIX(n4) is requested



A victim is found.



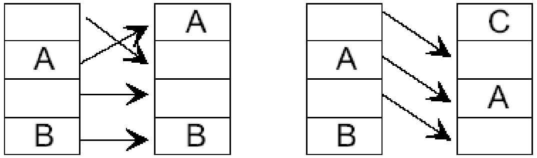
Block n1 replaces block b1.

Criteria of Buffer Replacement Policies

Criteria		Age of page in buffer		
		no	since last ref.	total age
References	none	Random		FIFO
	last		LRU CLOCK GCLOCK(V1)	
	all	LFU	GCLOCK(V2) DGCLOCK LRD(V2)	LRD(V1)

Schematic Overview of Buffer Replacement Policies

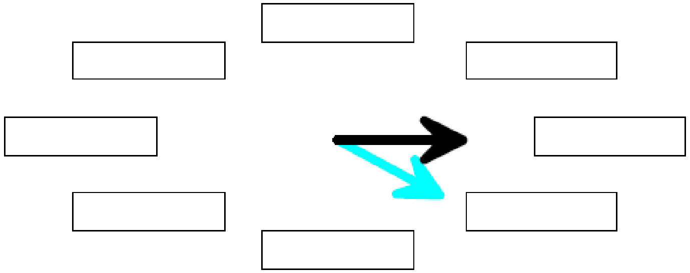
LRU



ref to A
in buffer

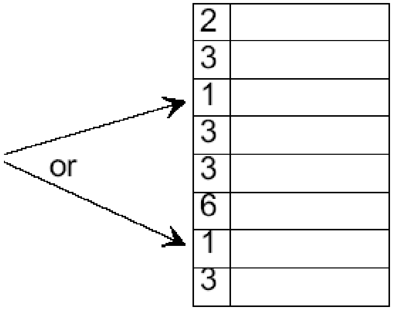
ref to C
not in buffer

FIFO



LFU

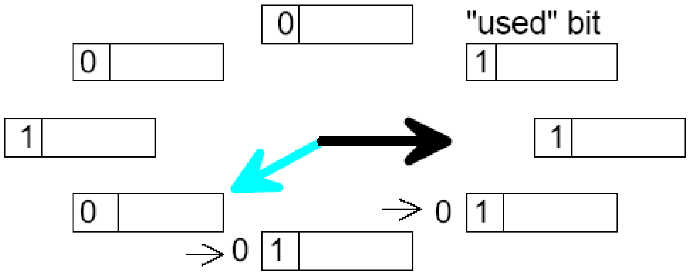
victim
page



2
3
1
3
3
6
1
3

CLOCK

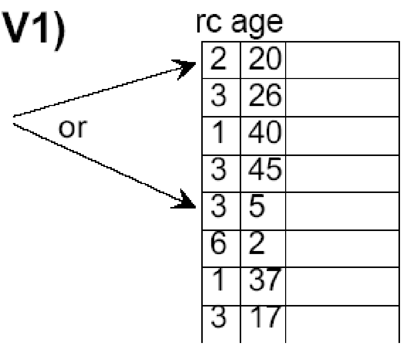
("Second
Chance")



LRD(V1)

victim
page

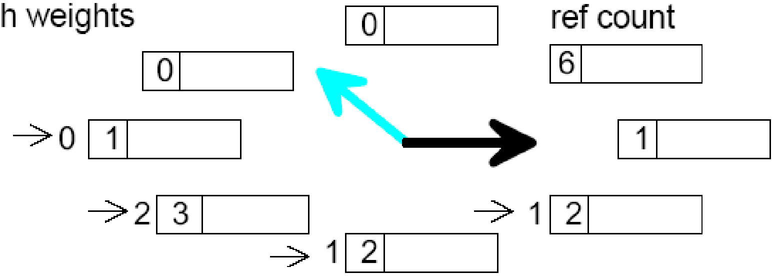
gc
50



rc	age
2	20
3	26
1	40
3	45
3	5
6	2
1	37
3	17

GCLOCK

possibly initialized
with weights

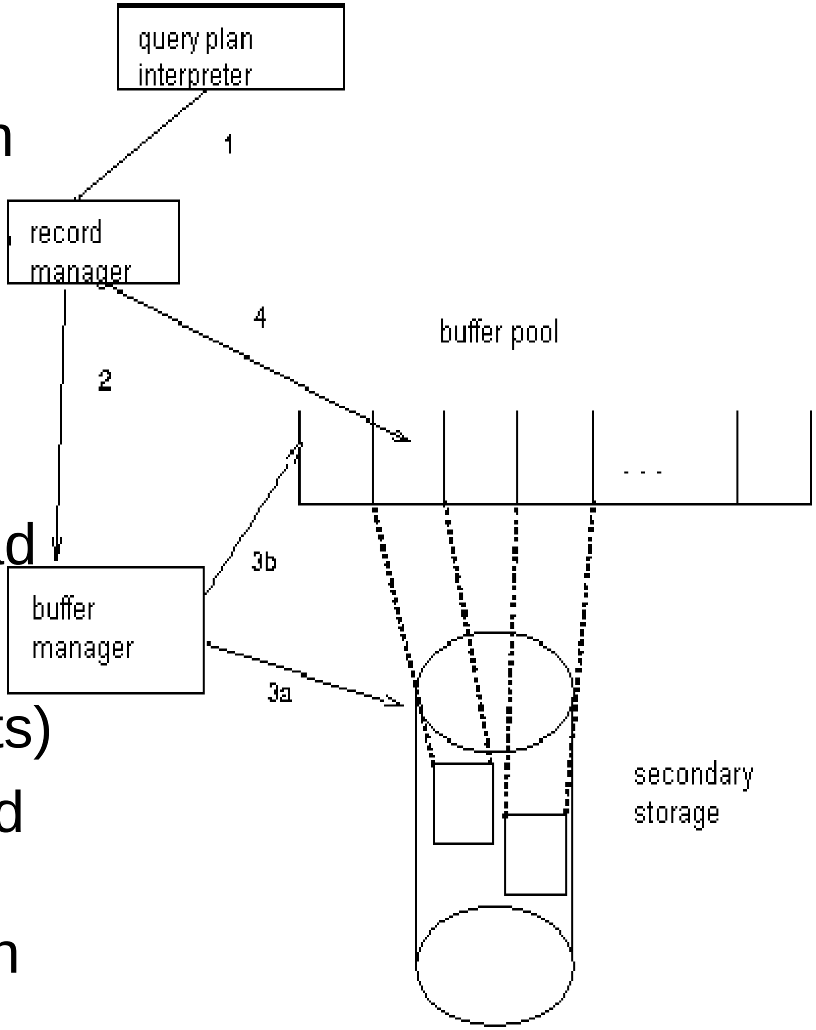


Buffer Management

- Existing OS affect DBMS operations by:
  - read ahead, write behind
  - uniform replacement strategies (DBMS is just an OS application!)
  - Unix is not good for DBMS to run on top
  - Most commercial DBMS implement their own I/O on a raw disk partition
- DBMS buffer management is more tricky
  - More semantics to pages: pages are not all equal
  - More semantics to access patterns: queries are not all equal
  - More concurrency on pages: often prescribe the order in which pages are written back to disk
  - Facilitates prefetching: on-demand(asynchronous),heuristic(speculative)
- Variations of buffer allocation
  - common buffer pool for all relations
  - separate buffer pool for each relation
  - as above but with relations borrowing space from each other
  - prioritised buffers for very frequently accessed blocks (data dictionary)

Buffer Management (DBMS) vs. Virtual Memory (OS)

- Goal in both cases: provide access to more data than will fit in main memory
  - ◆Page access patterns in DBMS can often be predicted (vs. in OS) e.g., in a query
  - ◆Pre-fetching of pages based on well-defined access patterns
    - when the buffer manager receives requests for (single) page(s), it may decide to (asynchronously) read ahead
    - reading contiguous page blocks is faster (vs. reading the same pages at different times as per several requests)
  - ◆WAL (Write-Ahead Log) protocol required by DBMS for crash recovery
    - forces some buffer pages to be written in the disc before others in order to implement the WAL protocol



Double Buffering

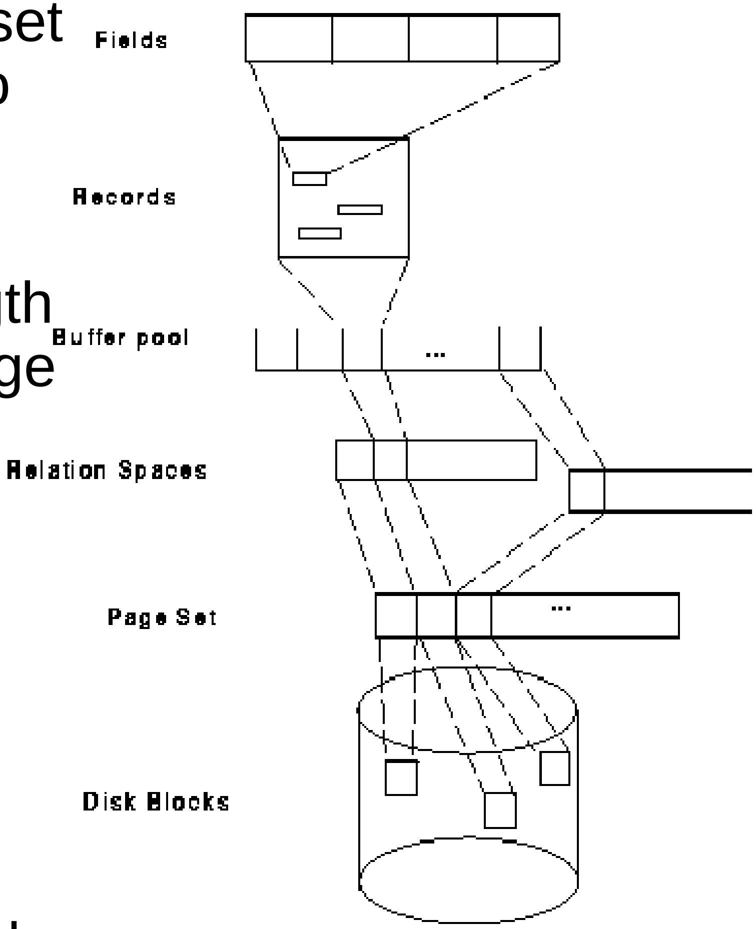
- If the DBMS uses it’s own buffer manager (within the virtual memory of the DBMS server process), independently from the OS VM manager, we may experience the following:
- Virtual page fault: page resides in DBMS buffer. However, frame has been swapped out of physical memory by OS VM manager
  - An I/O operation is necessary that is not visible to the DBMS
- Buffer fault: page does not reside in DBMS buffer, frame is in physical memory
  - Regular DBMS page replacement, requiring an I/O operation
- Double page fault: page does not reside in DBMS buffer, frame has been swapped out of physical memory by OS VM manager
  - Two I/O operations necessary: one to bring in the frame (OS); another one to replace the page in that frame (DBMS)
- => DBMS buffer needs to be memory resident in OS

Summary

- Disks provide cheap, non-volatile storage:
  - ◆Random access
  - ◆Cost depends on location of page on disk
  - ◆Goal: arrange data sequentially to minimize seek and rotation delays
- Blocks:
  - ◆a fixed-length unit for storage allocation and data transfer
  - ◆database files are organized into blocks
- Block Transfers
  - ◆Want to minimize the number of block transfers between disk and memory
  - ◆Keep as many blocks as possible in main memory
- Buffer
  - ◆portion of main memory available to store copies of disk blocks
- Buffer Manager
  - ◆subsystem responsible for allocating buffer space in main memory

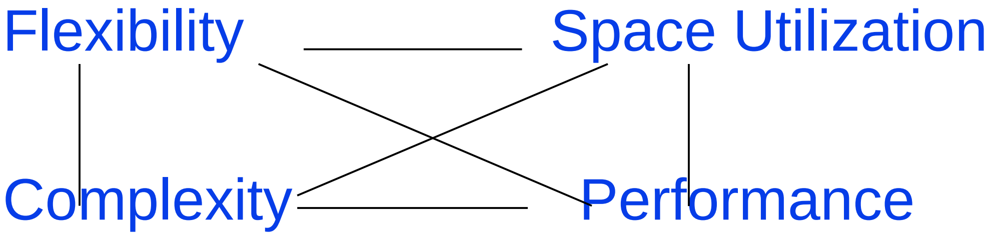
Summary

- Record format:
  - ◆Variable length record format with field offset directory offers support for direct access to i’th field and null values
- Page format:
  - ◆Slotted page format supports variable length records and allows records to move on page
- File Structure:
  - ◆Linked list or page directory structure to keeps track of pages in the file and pages with free space
- Disk Manager:
  - ◆Bitmap or linked list to keep track of free blocks on disk
  - ◆Contiguous or linked allocation of free blocks



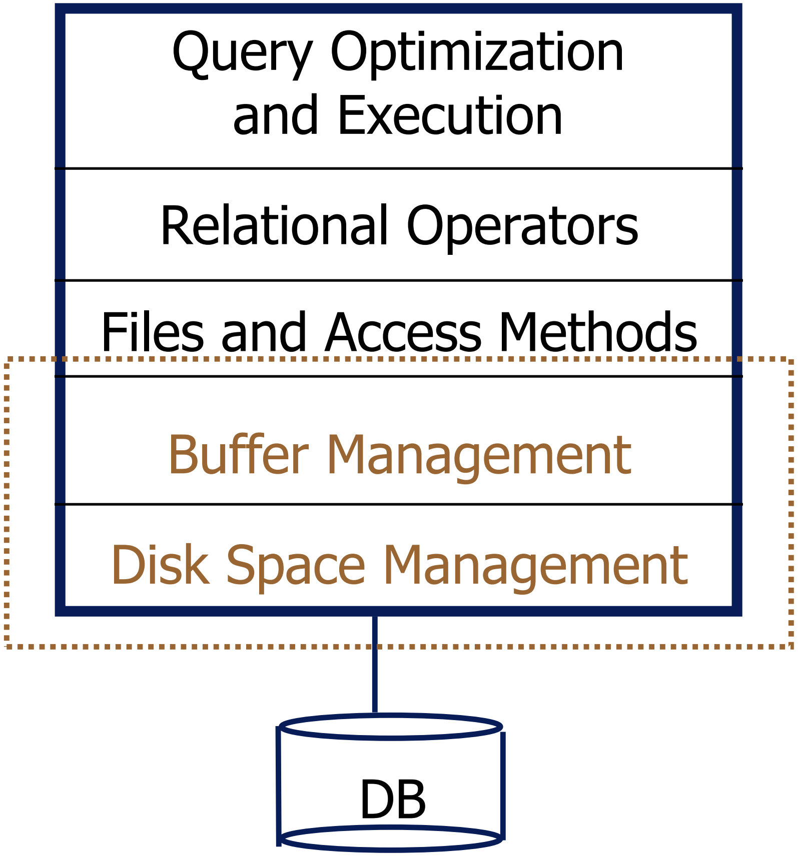
Summary

- There are 10,000,000 ways to organize the data on disk
- Which one is right? Issues:



- To evaluate a specific strategy, compute:
  - expected space usage
  - expected time to: fetch record given key, fetch record with next key, insert record, append record, delete record, update record, read all file, reorganize file

The BIG picture...



References

- Based on slides from:
  - R. Ramakrishnan and J. Gehrke
  - H. Garcia Molina
  - J. Hellerstein
  - L. Mong Li
  - P. Kilpeläinen
  - M. H. Scholl

Τέλος Ενότητας

Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο

ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ & ΘΡΗΣΚΕΥΜΑΤΩΝ, ΠΟΛΙΤΙΣΜΟΥ & ΑΘΛΗΤΙΣΜΟΥ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ
Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

ΕΣΠΑ
2007-2013
πρόγραμμα για την ανάπτυξη
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ

Χρηματοδότηση

- Το παρόν εκπαιδευτικό υλικό έχει αναπτυχθεί στα πλαίσια του εκπαιδευτικού έργου του διδάσκοντα.
- Το έργο «**Ανοικτά Ακαδημαϊκά Μαθήματα στο Πανεπιστήμιο Κρήτης**» έχει χρηματοδοτήσει μόνο τη αναδιαμόρφωση του εκπαιδευτικού υλικού.
- Το έργο υλοποιείται στο πλαίσιο του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» και συγχρηματοδοτείται από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο) και από εθνικούς πόρους.

Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο
ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ ΚΑΙ ΘΡΗΣΚΕΥΜΑΤΩΝ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ
ΕΣΠΑ
2007-2013
πρόγραμμα για την ανάπτυξη
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ
Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

Σημειώματα

Σημείωμα αδειοδότησης

.

Σημείωμα Αναφοράς

Copyright Πανεπιστήμιο Κρήτης, Δημήτρης Πλεξουσάκης. «**Συστήματα Διαχείρισης Βάσεων Δεδομένων. Διάλεξη 1η: Data storage, Record and file structures**». Έκδοση: 1.0. Ηράκλειο/Ρέθυμνο 2015. Διαθέσιμο από τη δικτυακή διεύθυνση: http://www.csd.uoc.gr/~hy460/