COMP163

Database Management Systems

Sections 17.1-17.3 & Appendix B
Physical Storage

Cost Analysis

- What is the cost of the following problems?

1. Given 100,000 unordered names count all the names beginning with ‘R’.
2. Given 100,000 names (unordered) find the alphabetic median.
3. Given 10,000,000 insurance customer records, find a customer with the median premium.
4. Given 250,000,000 US tax records, find the citizen with the largest tax payment in 2009.
5. Given 250,000,000 US tax records, find a citizen with the median tax payment in 2009.

Cost Analysis

		data in memory	moving data from disk to memory
1	O(n) n=100,000	100,000 comparisons	100 disk block reads
2	O(n log(n)) n=100,000	1,000,000 comparisons	100 disk block reads
3	O(n log(n)) n=10,000,000	100,000,000 comparisons	10,000 disk block reads
4	O(n) n=250,000,000	250,000,000 comparisons	250,000 disk block reads
5	O(n log(n)) n=250,000,000	2,500,000,000 comparisons	250,000 disk block reads

Rough estimate: 1000 records / disk block

Which takes more time, the in memory algorithm, or the disk reads?

Cost Analysis

		data in memory		moving data from disk to memory?	
1	O(n) n=100,000	100,000 comparisons	.0001 sec	100 disk reads	.1 sec
2	O(n log(n)) n=100,000	1,000,000 comparisons	.001 sec	100 disk reads	.1 sec
3	O(n log(n)) n=10,000,000	160,000,000 comparisons	.1 sec	10,000 disk reads	10 sec
4	O(n) n=250,000,000	250,000,000 comparisons	.25 sec	250,000 disk reads	4 min
5	O(n log(n)) n=250,000,000	4,800,000,000 comparisons	4.8 sec	250,000 disk reads (but will it all fit in memory?)	4 min

Assume 1 GHz (10^9 ops/sec) processor
Assume 1 msec block (10^3 blocks/sec) transfer time

Joins are Expensive

- R: 300,000,000 US citizens $3x10^8$
- T: 8,000,000 airline passengers $8x10^6$
- Join on name, count flights per citizen

- Naïve solution: cross product → $2x10^{15}$ records
  - No way it will fit in memory

- Better solution:
  - Read each citizen once,
    then read each passenger 300,000,000 times

Inefficient Join Algorithm

```
for c = 0, citizens.size()
    citizen_record = read_from_disk(citizens, c)
    flights = 0
    for p = 0, passengers.size()
        passenger_record = read_from_disk(passengers, p)
        if (citizen_record.name == passenger_record.name)
            flights++;
    write_to_disk(flight_count, citizen_record.name, flights)
```

10^{15} reads → 10^{12} seconds → 10^{8} hours

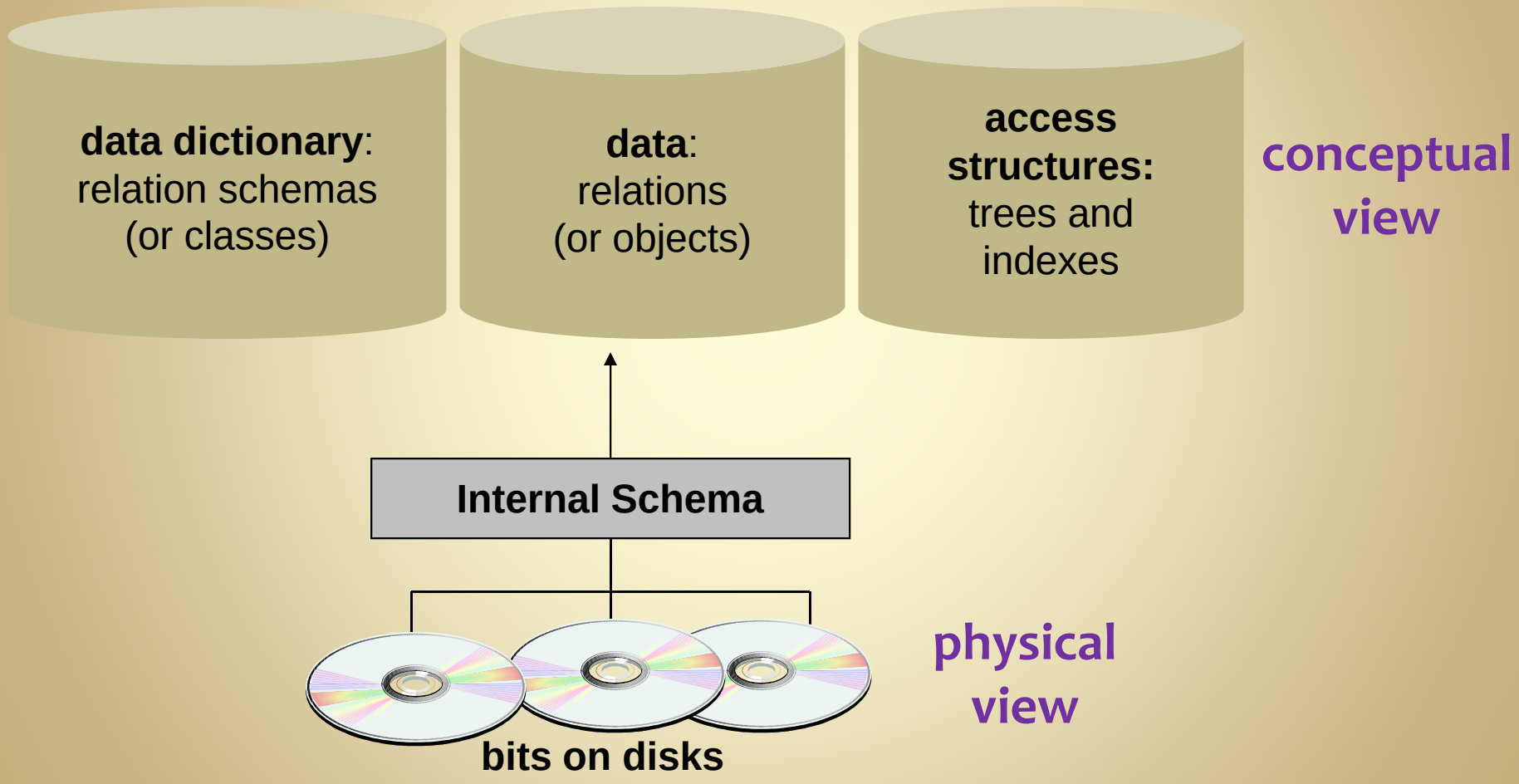
Clearly, we need a better algorithm

Physical Data Independence

- conceptual schema (tables)
  and external schema (views)
  are not affected by changes
  to the physical layout of the data

- Database (application) designers still need to understand the internals of the DBMS
  - to optimize performance
  - to perform maintenance
  - data structures and algorithms are applicable in other areas of computer science

Storage: Bits and Bytes



Storage Media

- electronic storage (cache, main memory)
  volatile fast (speed of light)
- flash memory (USB drives)
  non-volatile fast (limited by USB)
- magnetic disks
  non-volatile slow (moving parts)
- optical disks (CD-ROM, DVD)
  non-volatile slow (moving parts)
- tape
  non-volatile very slow (moving parts, sequential access)

Storage Media Prices

	typical retail price	GB / $1
main memory	1GB / $50	0.02
flash memory	32GB / $65	0.5
magnetic disk	500GB / $60	8
CD	100*700MB / $20	7
DVD	100*5GB / $35	10
tape	100GB / $20	10

Prices are out of date, relative prices should still be accurate.

exponent prefixes

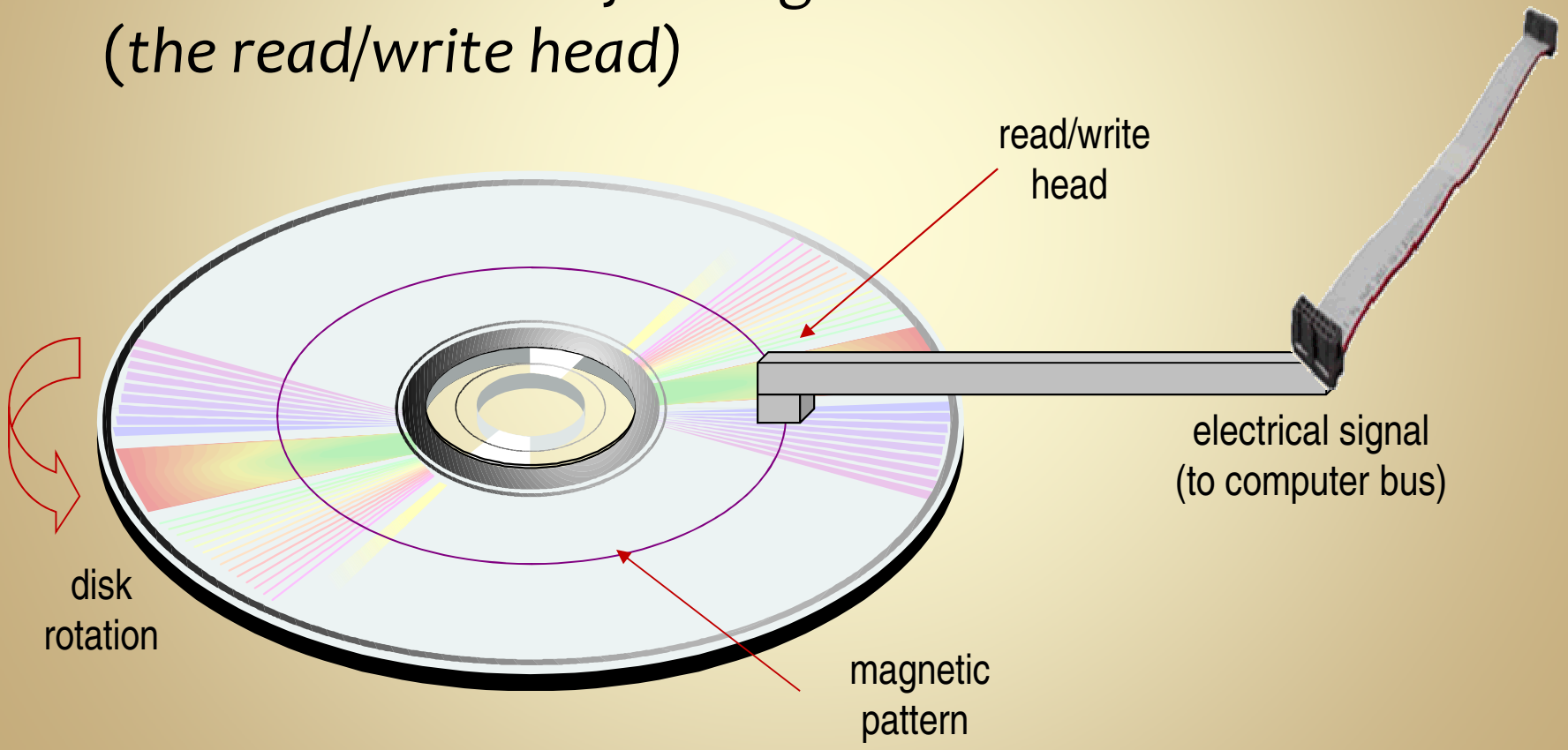
tera	10^{12}
giga	10^{9}
mega	10^{6}
kilo	10^{3}
milli	10^{-3}
micro	10^{-6}
nano	10^{-9}
pico	10^{-12}

Database Storage Needs

- To support a DBMS, the data store must be
  - non-volatile
  - readable and writeable
  - random access
  - cheap (large amounts required)
- A DBMS needs ***magnetic disk storage***
  - consequence: internal schema must be designed to optimize access in order to minimize the effect of slow physical parts
  - corollary: We need to know how a disk works: parameters that define access time are needed to optimize performance

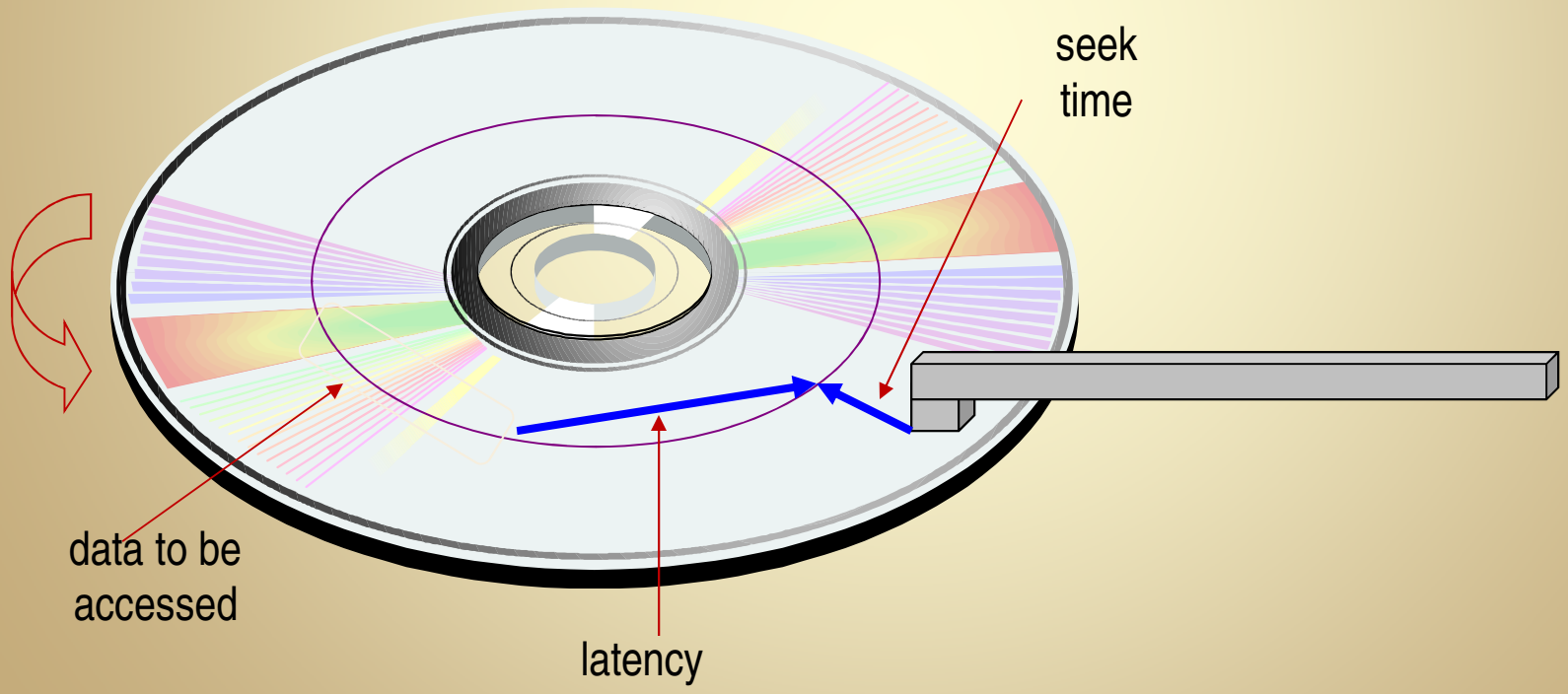
Disk Mechanics

- Data is stored in concentric *tracks*
- Data is accessed by a magnetic reader (*the read/write head*)



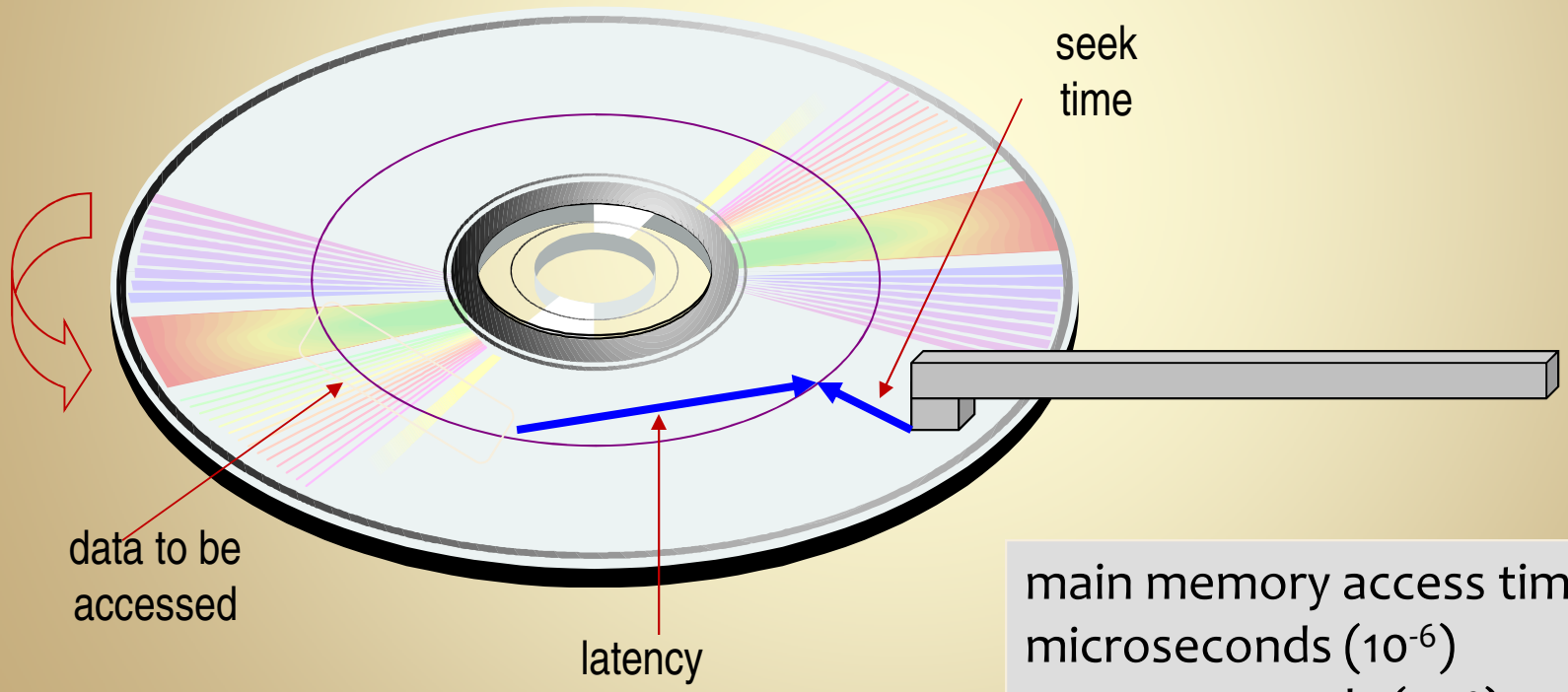
Disk Data Access

- To read or write data on a disk, the head must
  - move to the correct track (seek time)
  - wait for rotation to move data to it (latency)



Seek Time and Latency

- Average seek time: s = 3-4 msec
- Average Latency (rotational delay): rd = 2-3 msec



main memory access time is microseconds (10^{-6}) or nanoseconds (10^{-9})

Disk Speed and Rot. Delay

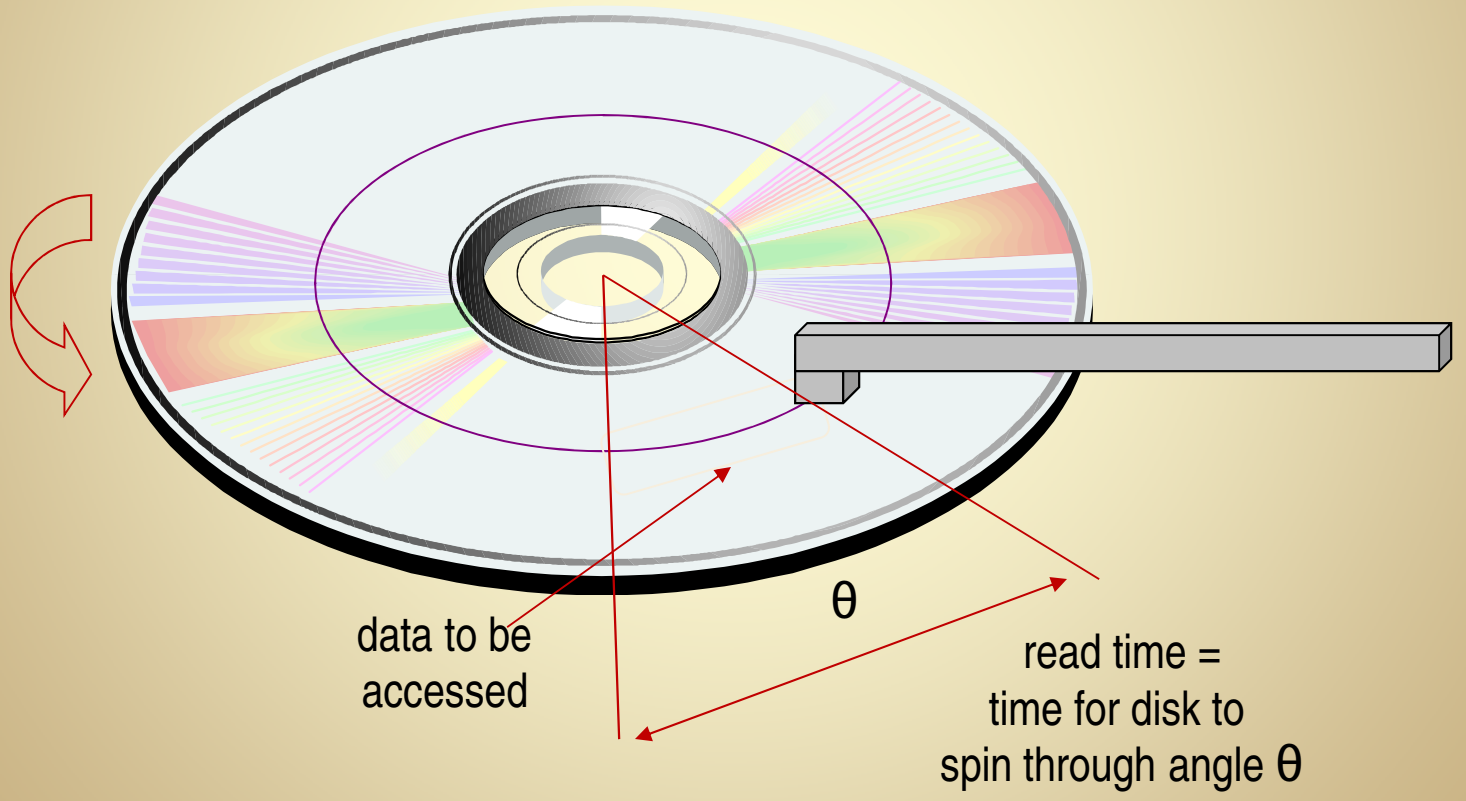
- Disk speed (p) is typically given in RPM
  - p = 10,000 rotations/minute is typical
- Rotational delay (rd) is the time for ½ rotation
  - rd = 0.5 / p

$$\frac{0.5}{10000\,rpm} = \frac{0.5\,\text{min}}{10000} \times \frac{60\,\text{sec}}{\text{min}} = 0.003\,\text{sec} = 3\text{msec}$$

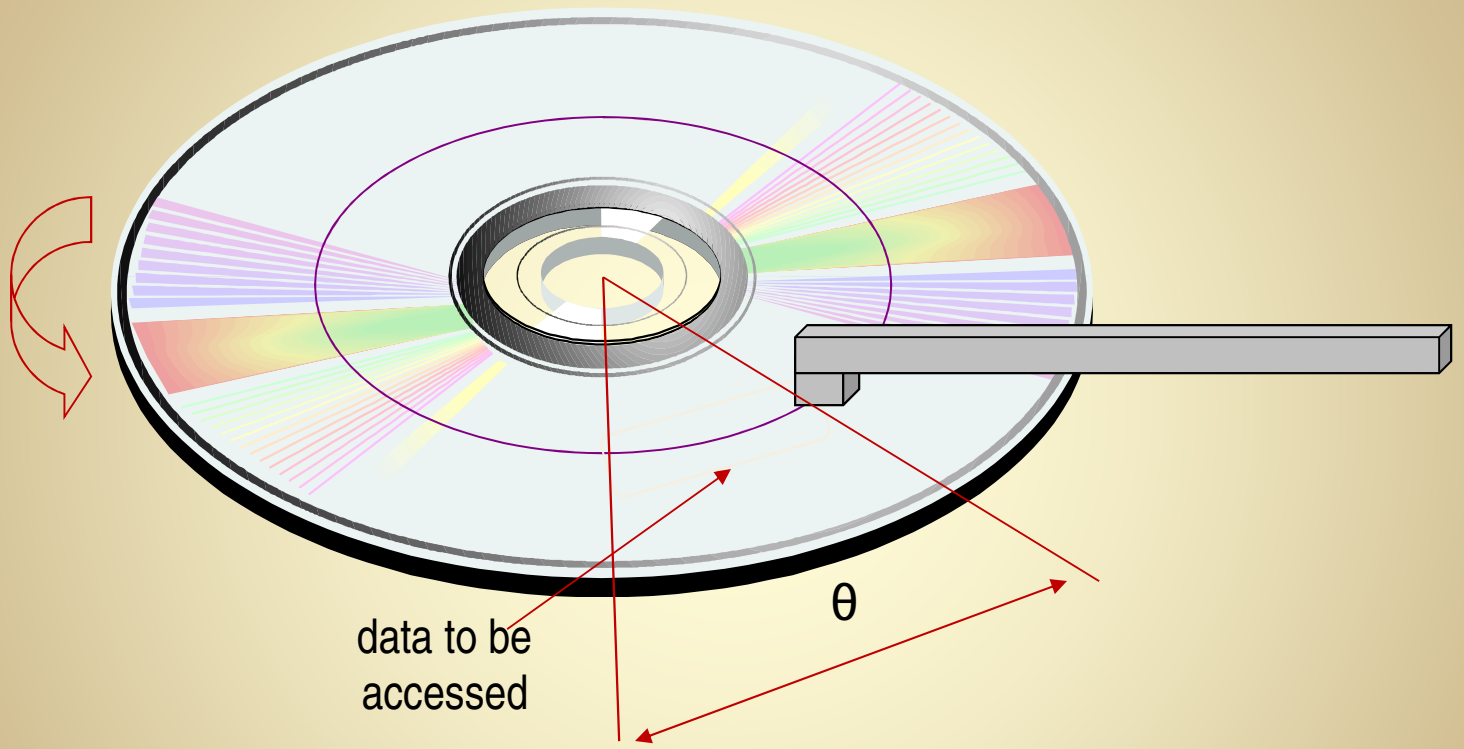
- Time for one full rotation
  is twice the rotational delay (2 * rd)

Reading Data from Disk

- Once start of data is located, it must be read or written



Angular Spin Time



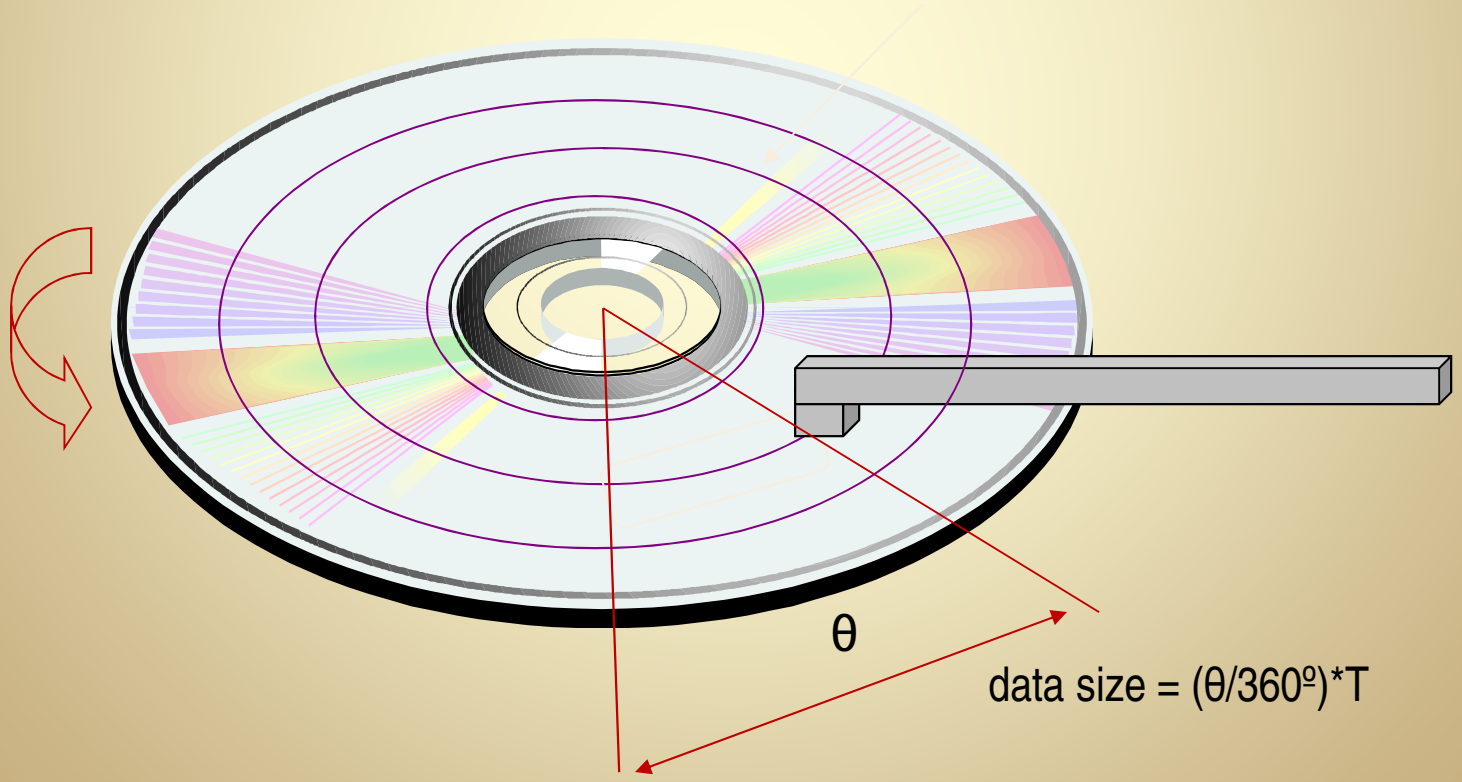
data transfer time: (θ/360º) * (2*rd) = (θ/360º) / p

Track Size

Assumption: all tracks hold the same amount of data and disk velocity is constant → constant transfer rate.

T = track size

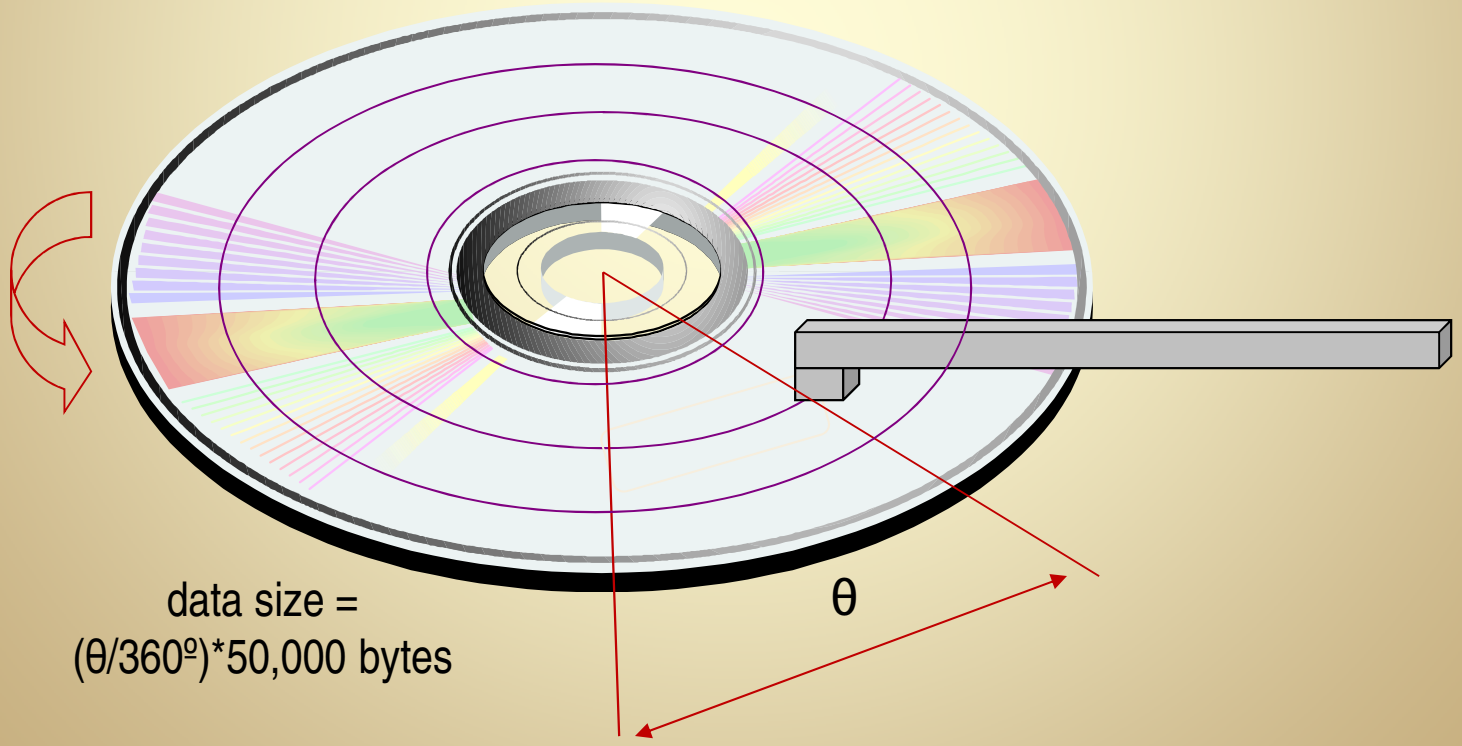
example = 50,000 bytes/track



Data Transfer Rate

track size: T = 50,000 bytes/track
velocity: p = 10000 rpm = 167 tracks/sec

transfer rate: tr = T * p = T /(2*rd)



data size =
(θ/360º)*50,000 bytes

Data Transfer Rate

track size: T = 50,000 bytes/track
velocity: p = 10000 rpm = 167 tracks/sec

transfer rate (tr):

tr = T*p
= 50000 bytes/track * 167 tracks/sec
= 8,350,000 bytes/sec = 8 MB/sec

alternate calculation:

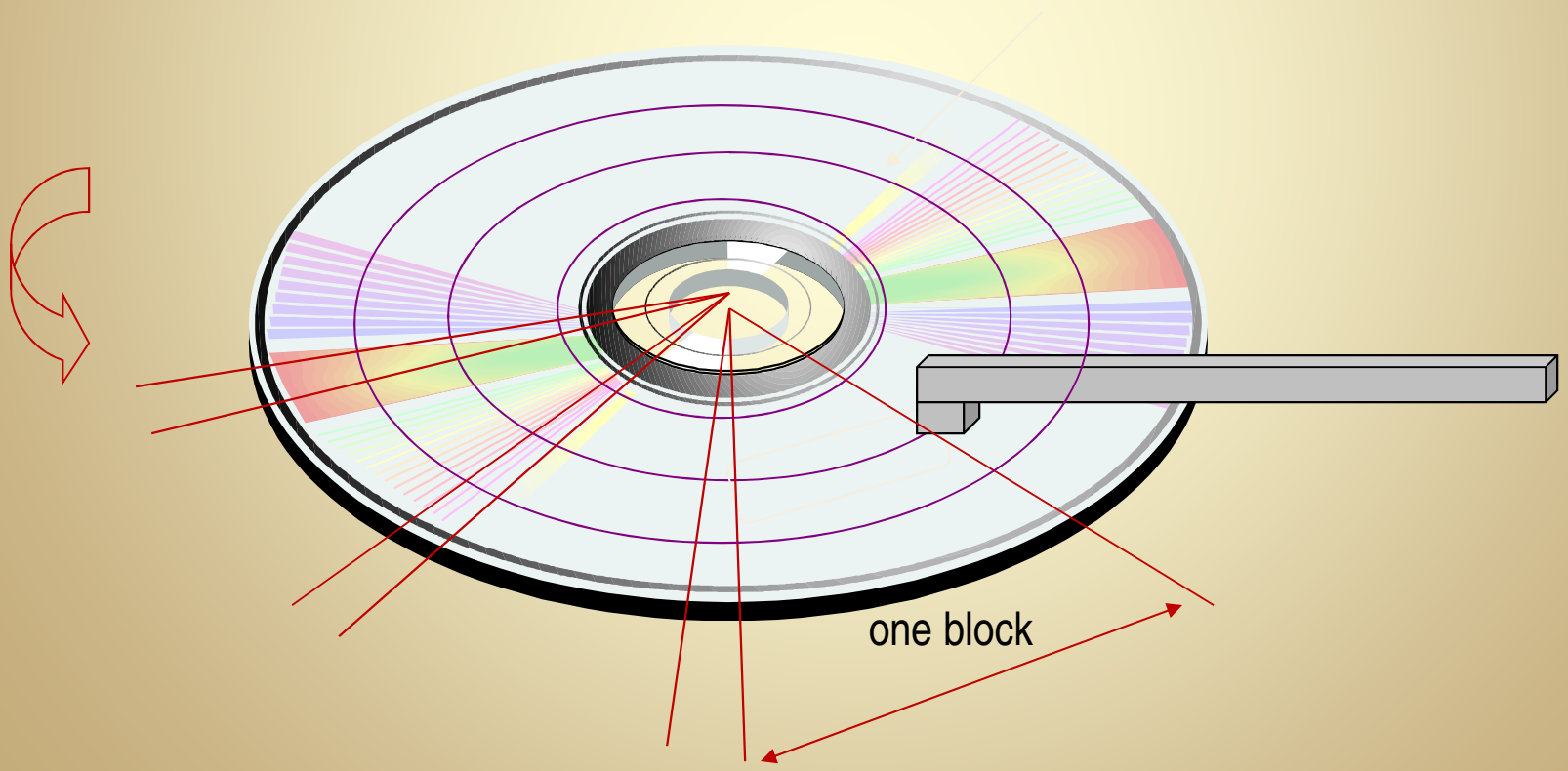
tr = T/(2*rd)
= 50000/(2*3e-3)
= 8 MB/sec

Blocking

Tracks are divided into *blocks*, separated by *inter-block gaps*

typical block size: B = 1024 bytes (50 blocks/track)

typical gap size: G = 128 bytes

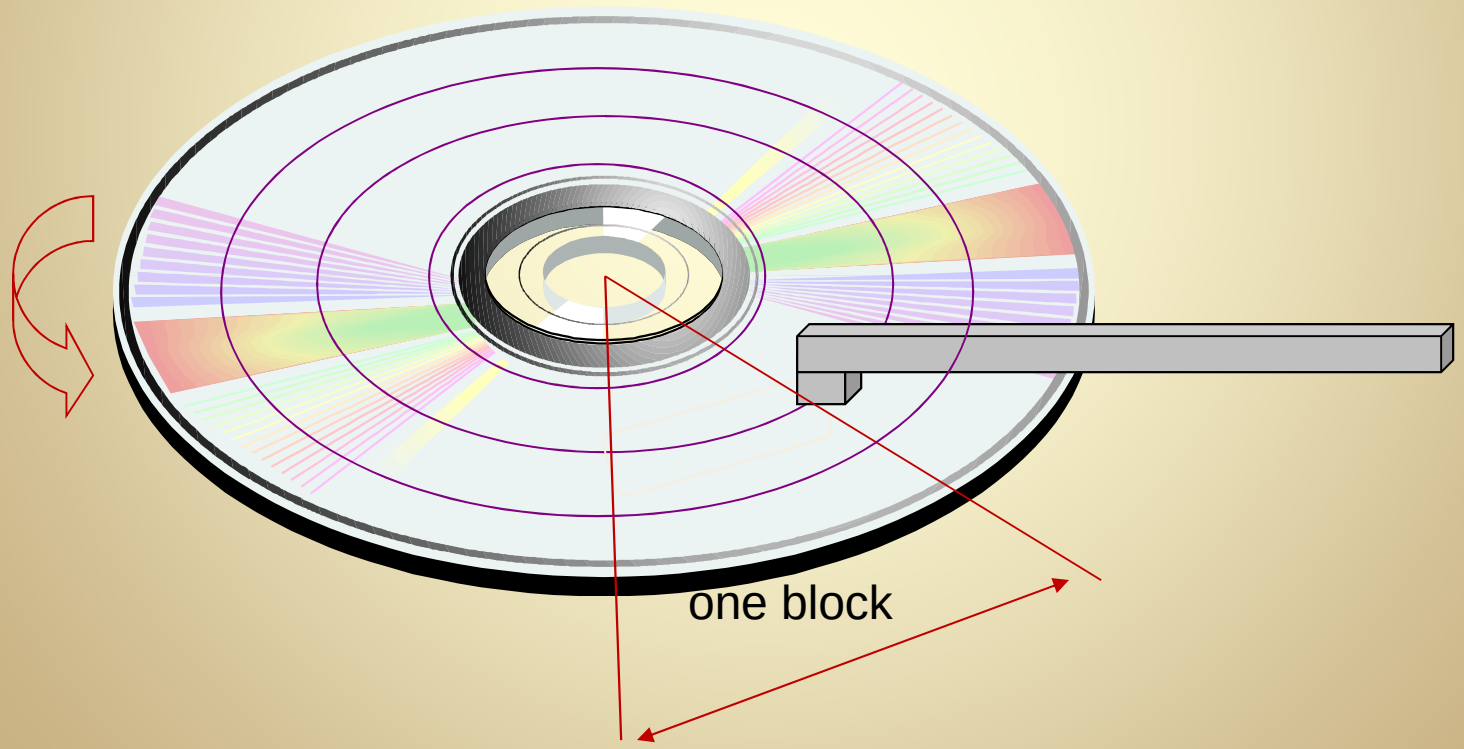


Block Transfer Rate

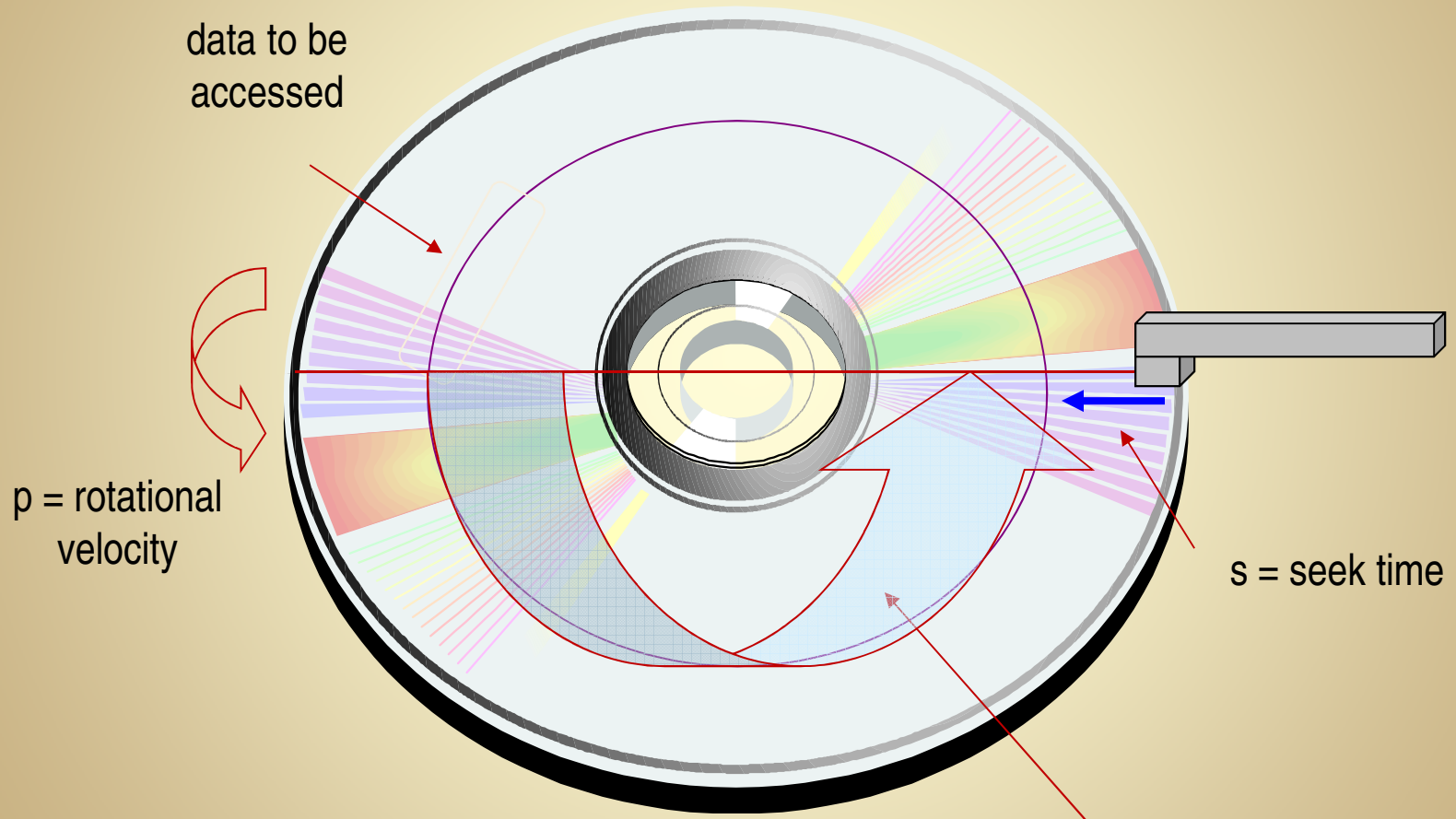
block transfer time:

btt = B / tr =

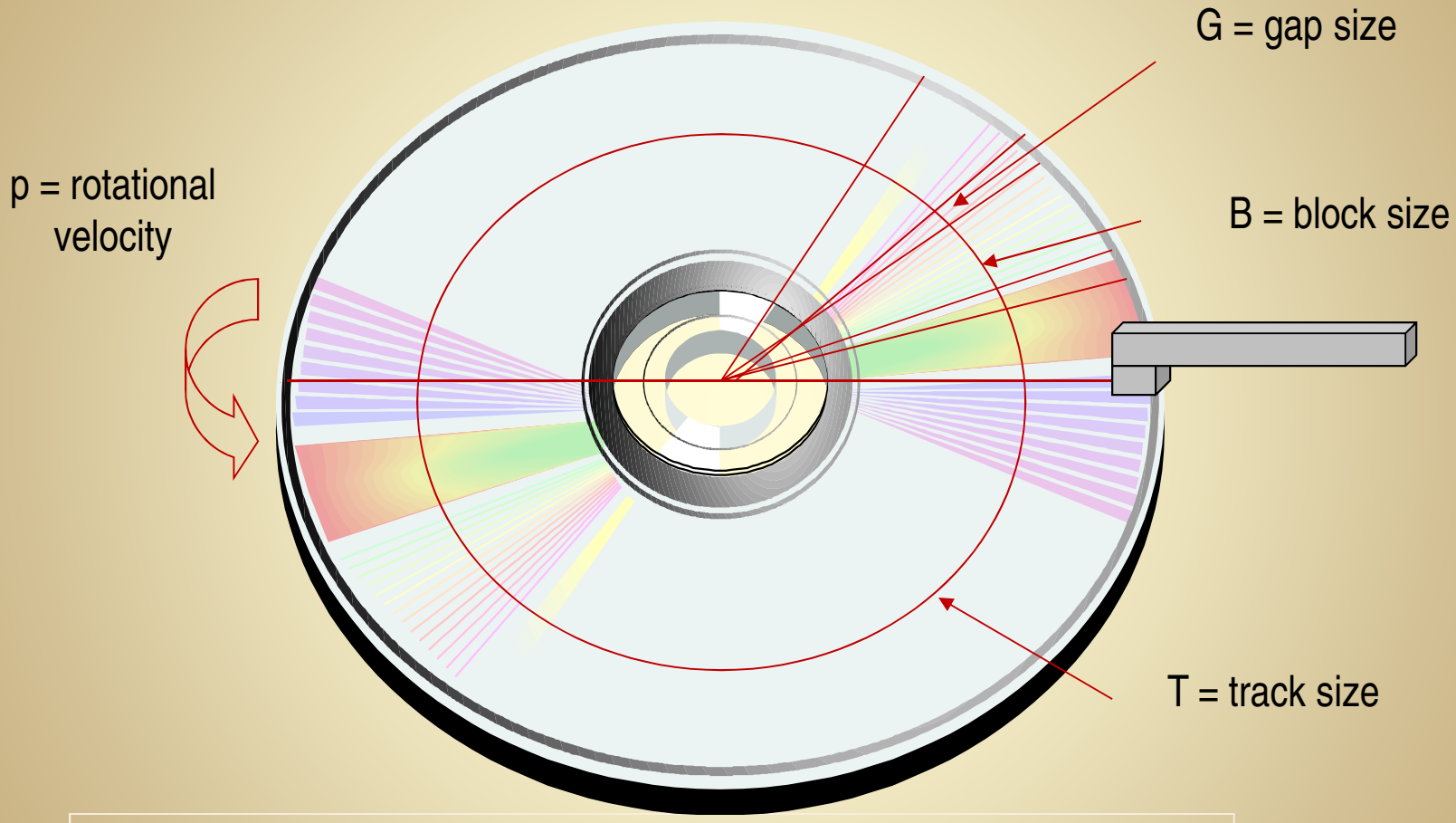
= 1024 bytes / (8Mb/sec) = 0.128 msec



Seek Time, Latency and Rotational Velocity



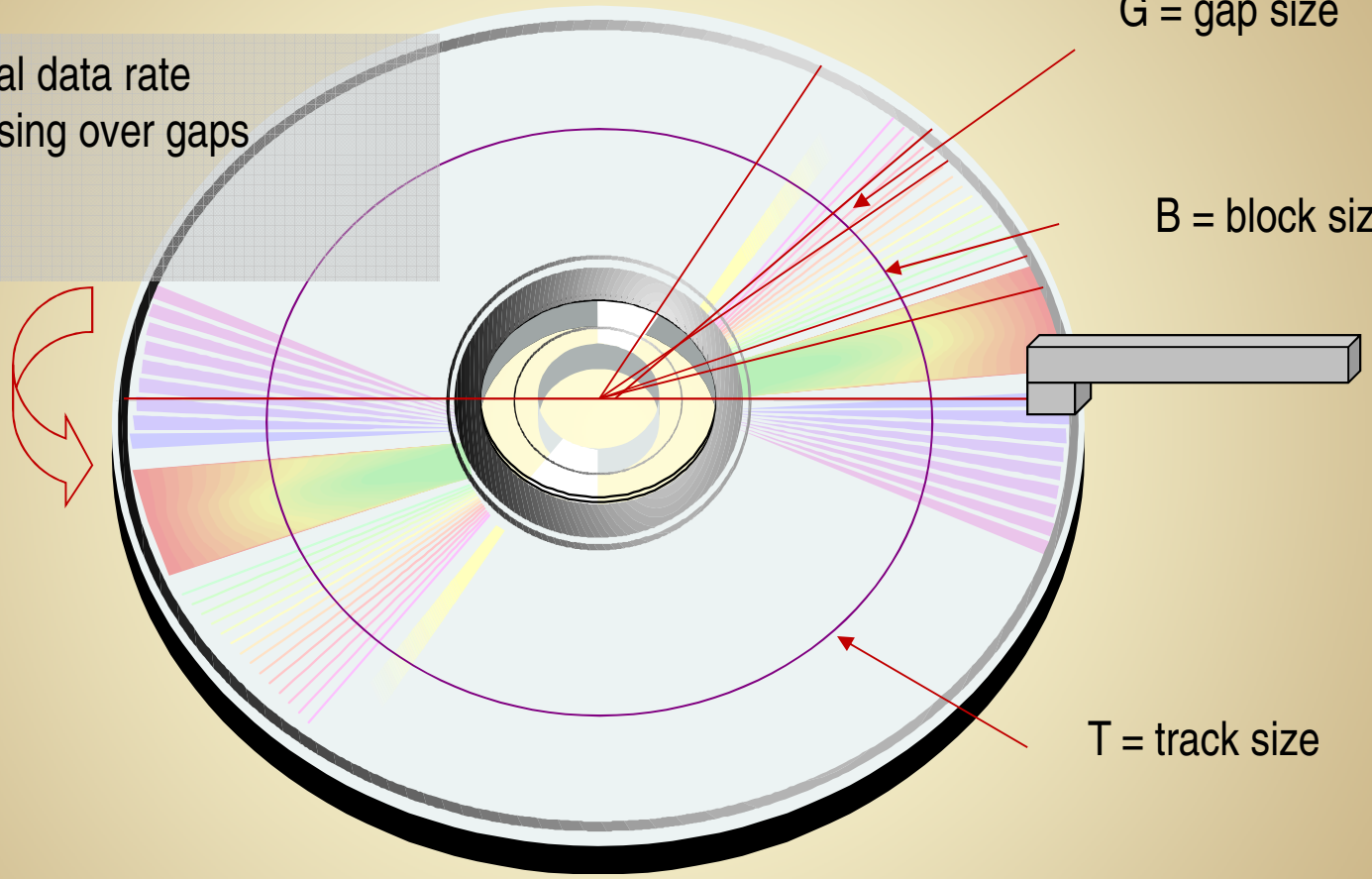
Block Transfer Time



btt = time for one block to pass under read/write head
= B/tr

Bulk Transfer Rate

btr = best actual data rate
since time passing over gaps
is "wasted"



btr = transfer rate for *consecutive* blocks = $\frac{B}{B+G} \times tr$

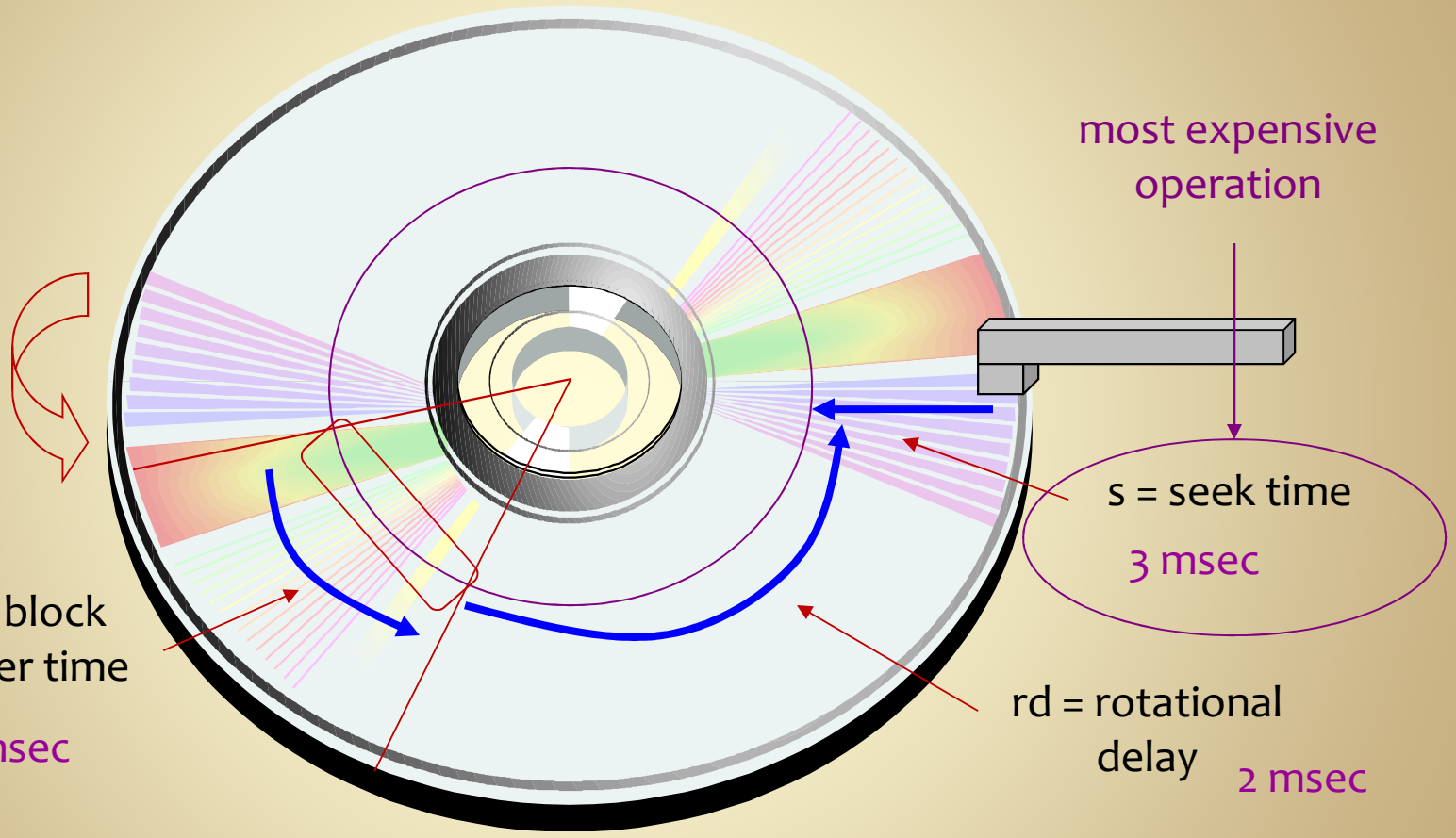
Bulk Transfer Rate

- btr = best actual data rate,
  since time passing over gaps is "wasted"

- For our purposes, we'll generally
  ignore the gaps to simplify the computations,
  thus

btr = transfer rate for *consecutive* blocks = B * tr

Random Block Transfer Time



rbtt = time to locate and transfer one random block
= s + rd + btt (5.1msec)

Transferring Multiple Blocks

Time to transfer n blocks of data:

if blocks are randomly located, we pay seek and latency for each block:

rbtt = n*(s + rd + btt)

if blocks are consecutively located, we only pay seek and latency once

cbtt = s + rd + n*btt

Fundamental Results

- Organize data in blocks
  - this is the basic unit of transfer
- Whenever possible, layout data to maximize possibility of consecutive block retrieval
  - avoids seek and latency costs
- This will impact
  - record layout in files
  - access structure (indexes, trees) organization

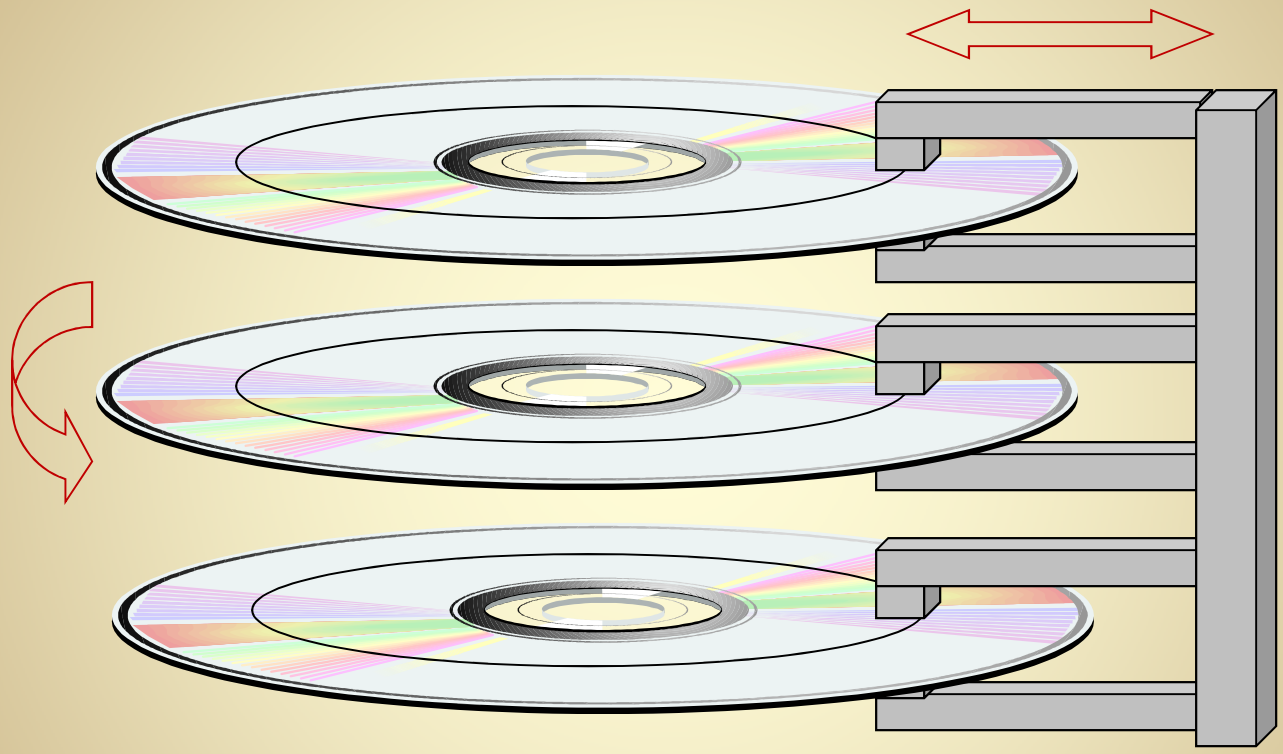
Disk Parameters

parameter		typical value	source
T	track size	50 KB	fixed
B	block size	1 KB	formatted
G	interblock gap size	128 bytes	formatted
s	seek time	10 msec	fixed
p	rotational velocity	1,000 rps	fixed
rd	(average) rotational delay	3 msec	.5*(1/p)
tr	transfer rate	1MB/sec	T*p
btt	block transfer time	1 msec	B/tr
btr	bulk transfer rate (consecutive blocks)	700KB/sec	(B/(B+G))*tr

Disk Packs

- Typical disk drives have multiple disk surfaces
  - surfaces are sometimes called platters
- Disks are connected to same spindle
  - disks rotate together
- Each surface has its own read/write head
  - Heads are connected to single motor, they all move together
- We can read/write the same block on multiple disks simultaneously

Cylinders



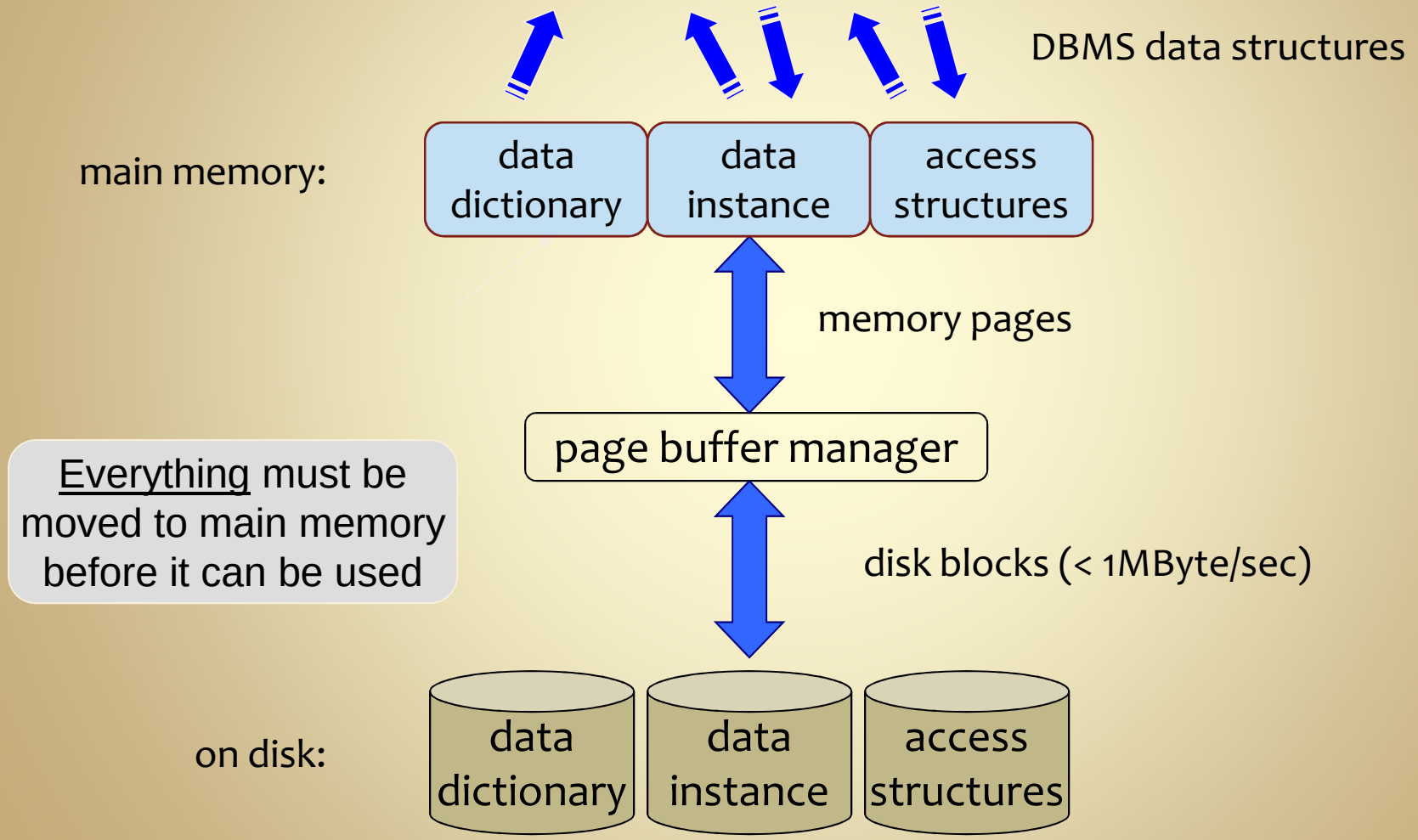
a *cylinder* is made up of the same track on all platters

COMP163

Database Management Systems

Sections 17.3-17.7
File Organization

Page Buffer Management



Blocks and Pages

- A *page* is a unit of data transfer from the DBMS point of view

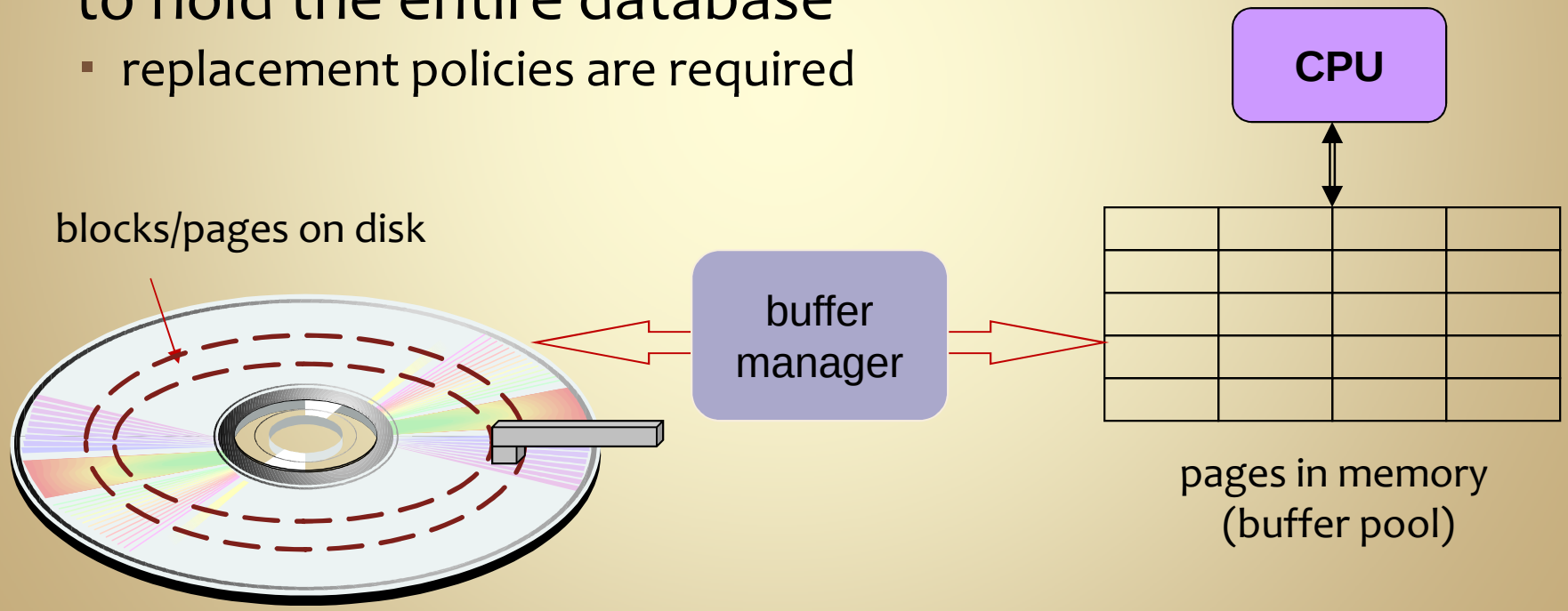
- Disk blocks are the smallest practical page size
- Larger pages are also possible:
  - all data on one track
  - all data on one cylinder
  - same block, same cylinder on all surfaces

- typical page size: 1-10 Kbytes
- typical page transfer time (ptt): 1-10 msec

for this course: we'll generally assume one page = one block

Page Management

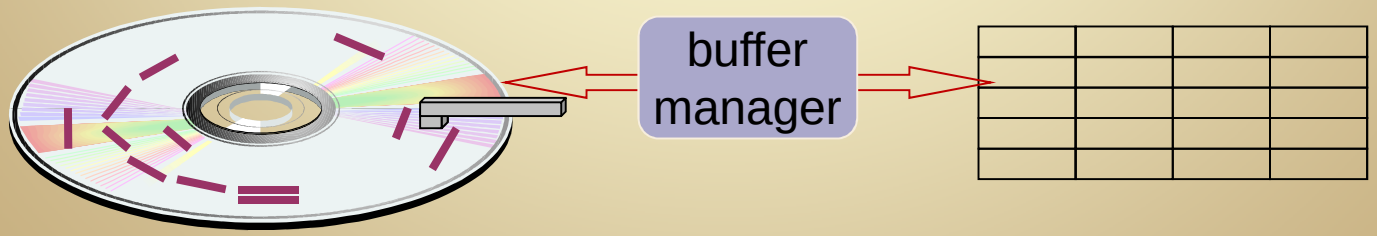
- pages are held in main memory in a *buffer pool*
- the buffer pool is typically not large enough to hold the entire database
  - replacement policies are required



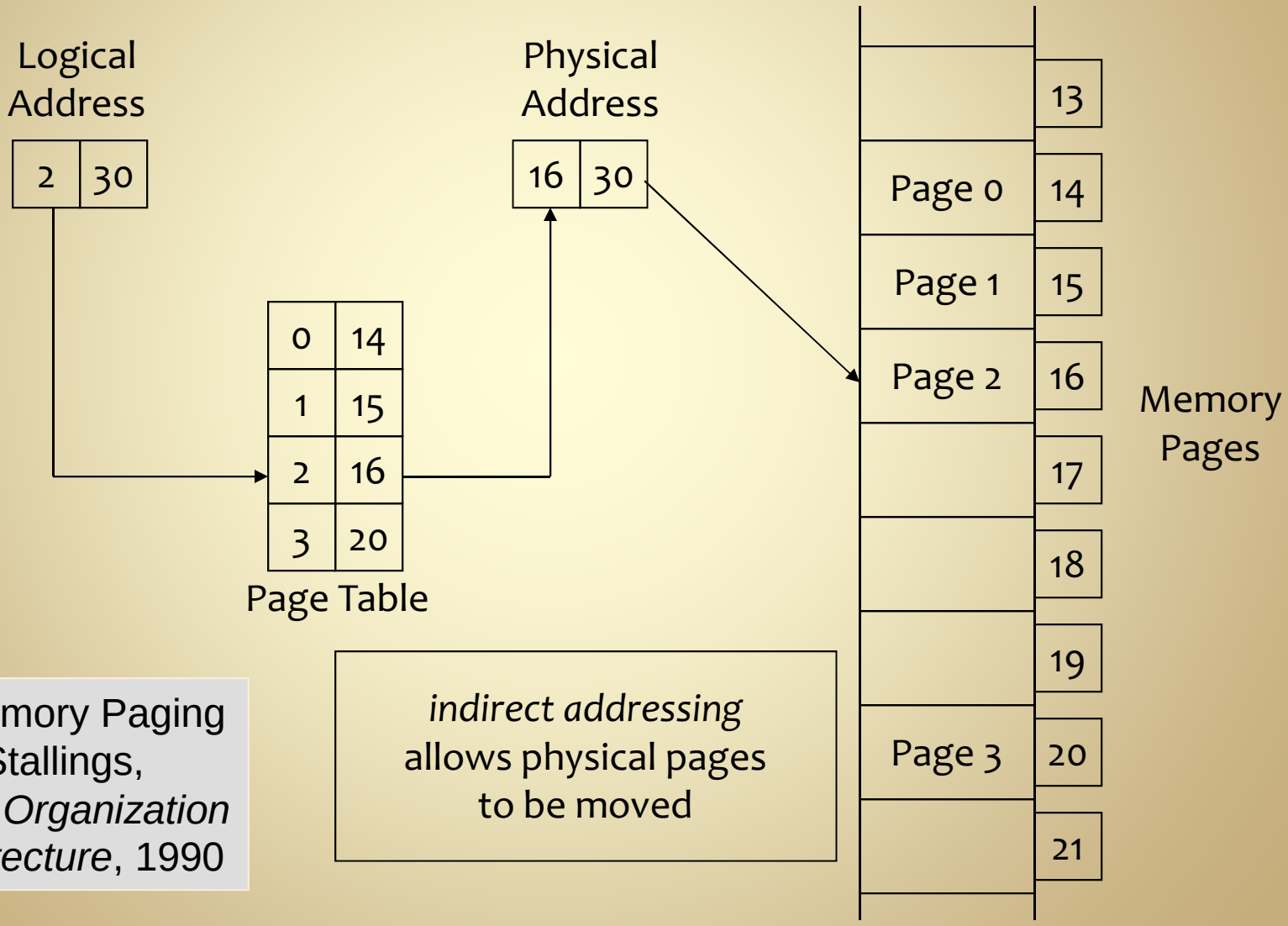
Page Buffer Manager

- The buffer manager must know
  - which pages are in use (pinned pages)
  - which pages have modified data (dirty pages)

- replacement policies:
  - FIFO: oldest non-pinned page is replaced
  - LRU: page that has been unpinned the longest is replaced
  - semantic strategies: if page m is replaces, replace page n next, because pages m and n are related



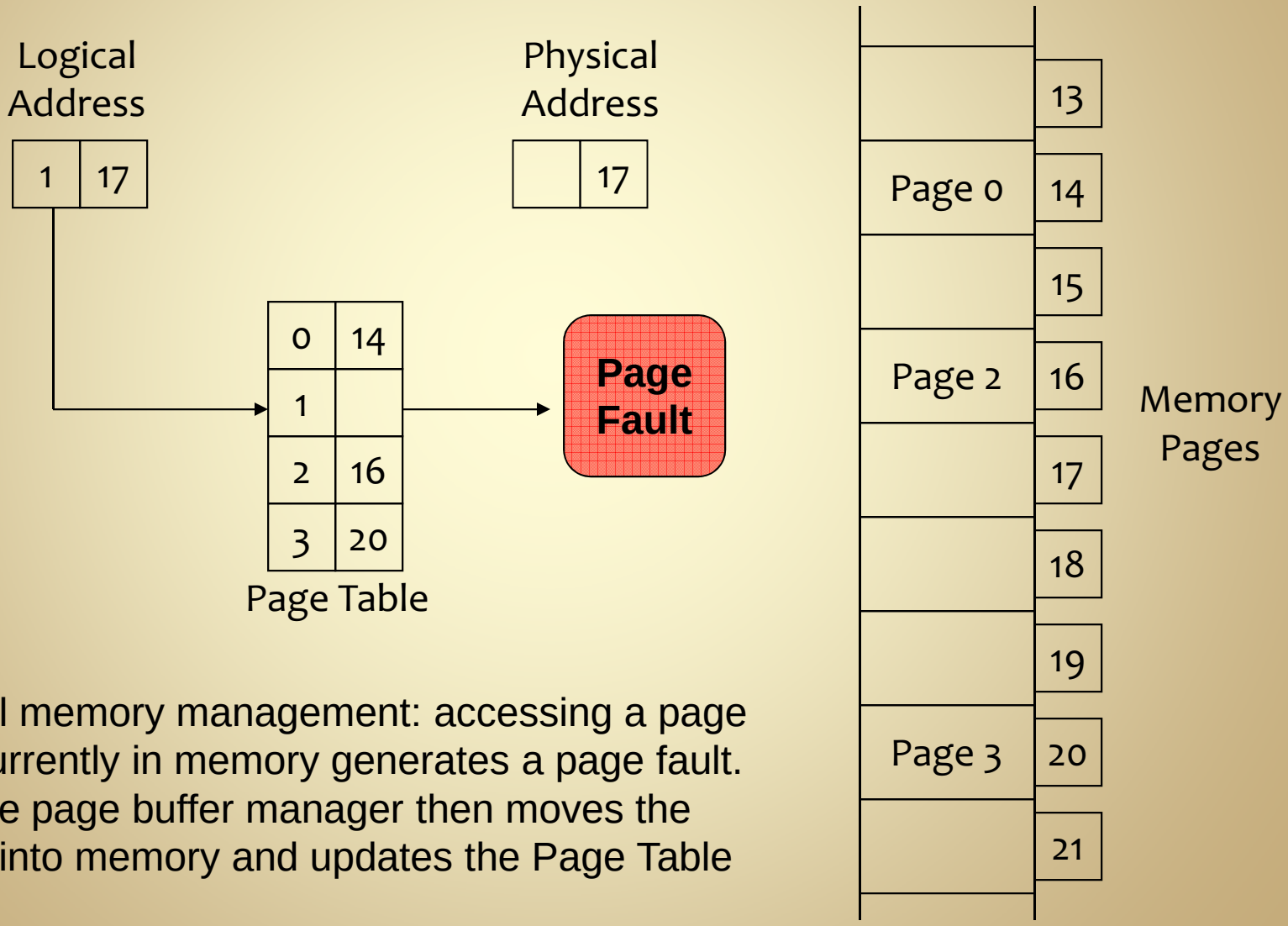
Memory Page Management



Virtual Memory Paging from Stallings, *Computer Organization and Architecture*, 1990

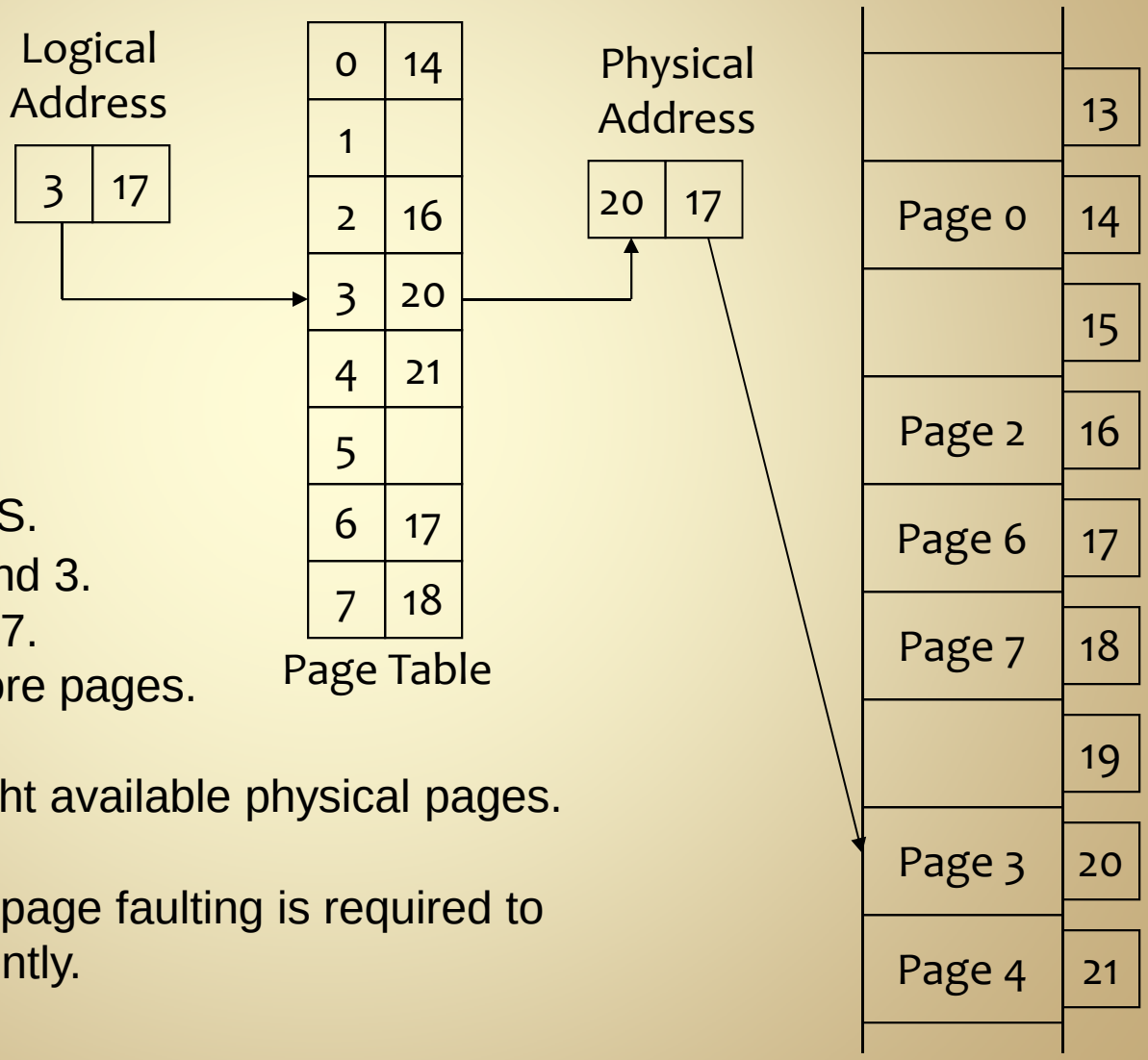
indirect addressing allows physical pages to be moved

Virtual Memory Management



Virtual memory management: accessing a page not currently in memory generates a page fault. The page buffer manager then moves the page into memory and updates the Page Table

DBMS Page Buffer Mngt



Example: compute R ⋈ S.
R occupies pages 1, 2 and 3.
S occupies pages 6 and 7.
Result will require six more pages.

Suppose there's only eight available physical pages.

Something smarter than page faulting is required to implement the join efficiently.

DBMS vs. OS Paging

- A DBMS understands the algorithms being applied to the data
  - It can utilize replacement policies appropriate for the current data and operations.
  - It can influence data access patterns.
- An OS does not understand the algorithms being applied to the data structures
  - It must use generic replacement policies.
  - It can only see the data access patterns as they happen.

Exercise

- Conceptual Schema:
  WorksOn(EmpID, ProjectID, Hours, Role, Week)
  - sizeof(EmpID) = 8 bytes
  - sizeof(ProjectID) = 4 bytes
  - sizeof(Hours) = 8 bytes
  - sizeof(Role) = 30 bytes
  - sizeof(Week) = 6 bytes
- Instance: 10,000 records
- Disk block size: 1024 bytes

- How many blocks?
- Should we sort/order the file? What order?

Blocking Factors

- Data must be organized in *blocks*
  - a block (or page) is the unit of transfer between disk and memory
- A *record* is a data unit
  - tuple, object or portion of an access structure
- *Blocking factor* determines how many records can fit in a block
  - bfr = $\lfloor B / R \rfloor$ where B = block size and R = record size
- *Number of blocks* required is determined by the blocking factor and the number of records
  - b = $\lceil r / bfr \rceil$ where r = number of records

Blocking and Sorting

- Let R be a relation with key attribute(s) K
- Options for storing R on disk:
  - unsorted, random blocks
  - sorted by key, random blocks
    (impractical, requires "next-block" pointers,
    yielding worst case performance)
  - unsorted, consecutive blocks
  - sorted by key, consecutive blocks

Disk Access Costs

rbtt = random block transfer time = s + rd + btt (bad)
cbtt = consecutive block transfer time = btt (good)

	insert	delete	select (on key)
unsorted non-consecutive	O(1)*rbtt	O(n)*rbtt	O(n)*rbtt
unsorted consecutive	O(1)*rbtt	O(n)*cbtt	O(n)*cbtt
sorted consecutive	O(n)*cbtt	$O(\log_2 n)$*rbtt	$O(\log_2 n)$*rbtt

assumes we don't reorganize file, simply mark the record as deleted

binary search

deletion cost is same as selection:
we have to find the record
that we want to delete

Records and Files

- A ***record*** is one unit of structured data
  - **tuple** in a relation
  - **node** in a tree or index
  - object or other structure
- Records may be *fixed length* or *variable length*
- A ***file*** is a set or records stored as a unit on disk
  - a file is stored on some set of disk blocks
- *spanning* file organization:
  records can be split among blocks
- *non-spanning* file organization:
  whole records must be stored in a single block

Blocking

- ***Blocking***: storing a number of records in one block on the disk.
- ***Blocking factor*** (**bfr**) refers to the number of records per block.
- There may be empty space in a block if an integral number of records does not fill a block (non-spanning)

File Organization

- Physical disk blocks allocated to hold a file can be *contiguous, linked, or indexed.*
- ***contiguous***: Once you find the first block, keep reading blocks in sequence until the end
- ***linked***: Each block has a block pointer to the next block
- ***indexed***: an access structure (index or tree) holds block pointers to all blocks in the file

Unordered Files

- Also called a **heap** or a **pile** file.
- New records are inserted at the end of the file.
  - Very efficient
- A **linear search** through the file records is necessary to search for a record.
  - Reading ½ the file blocks on the average → expensive: O(n)

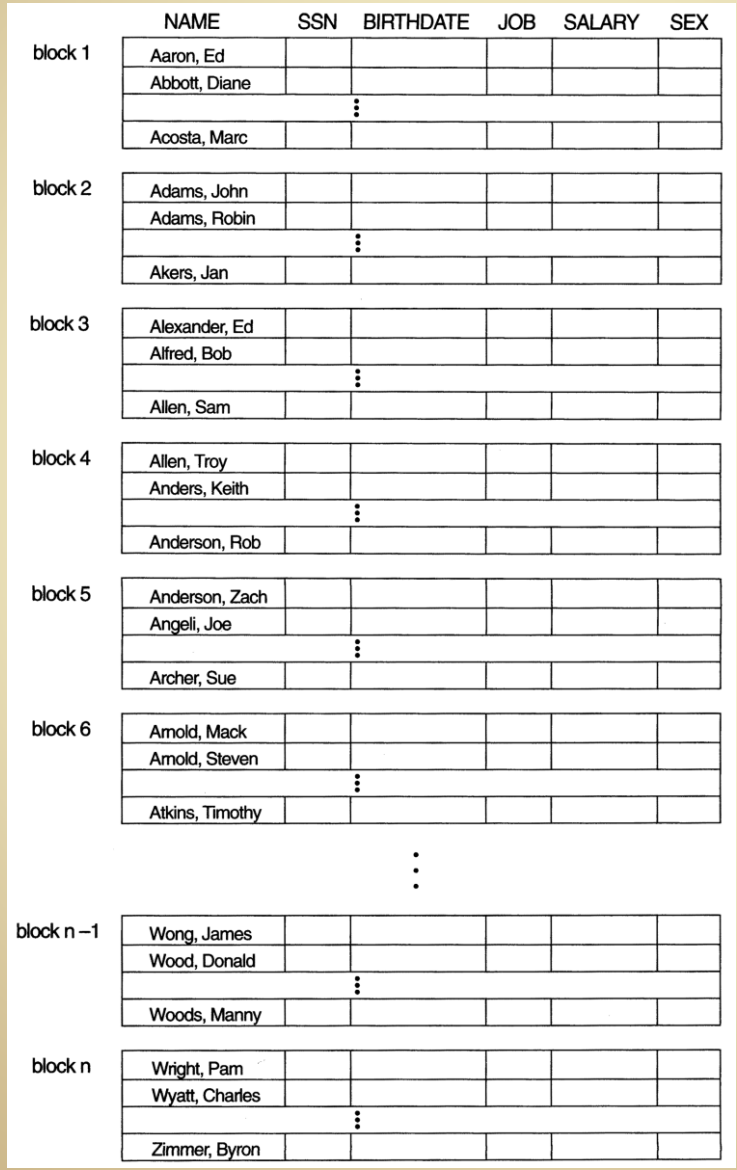
Ordered Files

- Also called a **sequential** file.
- File records are kept sorted
  by the values of some *ordering field*.
- Records must be inserted in the correct order.
  - Insertion is expensive: n/2 reads & writes (on average)
- **Binary search** can be used
  to search for a record on its *ordering field* value.
  - O(log n) reads

Faster ins/del on Ordered Files

- Normally, insertion requires moving a lot of records
  - O(n) block read/writes
- Improvement:
  - Keep a separate unordered *overflow* file for new records to improve insertion efficiency
  - Periodically merge overflow file with the main ordered file.
- Deletion also requires moving a lot of records
- Improvement:
  - Simply mark deleted records (and ignore them on subsequent access)
  - Periodically scan the file and removed the marked records
  - Requires a "deleted flag" in each record

Ordered File Example



Logarithms and Exponents

- logarithms are the inverse of exponents

base 2

$2^1 = 2$	$\log_2(2) = 1$
$2^2 = 4$	$\log_2(4) = 2$
$2^3 = 8$	$\log_2(8) = 3$
$2^{10} = 1024$	$\log_2(1024) = 10$

Quick Quiz:
What is $\log_{10}(345{,}768)$?

base 8

$8^1 = 8$	$\log_{10}(8) = 1$
$8^2 = 64$	$\log_8(64) = 2$
$8^3 = 512$	$\log_8(512) = 3$
$8^{10} = 1{,}073{,}741{,}824$	$\log_8(1{,}073{,}741{,}824) = 10$

Logarithmic Behavior

Which is better:
O(n)*cbtt
or
O(log n)*rbtt ?