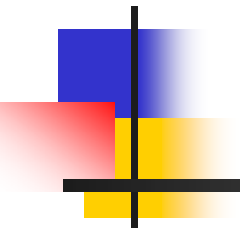


Data Storage - I:
Memory Hierarchies & Disks

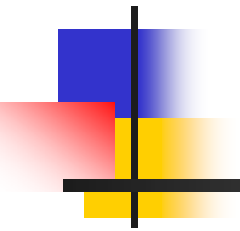
W7-C, Spring 2005

Updated by M. Naci Akkøk, 27.02.2004 and 23.02.2005,
based upon slides by Pål Halvorsen, 11.3.2002.
Contains slides from:
Hector Garcia-Molina, Ketil Lund, Vera Goebel

Overview

- Implementing a DBS is easy!!??
- Memory hierarchies
  - caches
  - main memory
  - secondary storage
  - tertiary storage
- Disks
  - characteristics
  - access time
  - throughput
  - complicating issues
  - disk controllers

Example: A Simple DBS Implementation



Isn't Implementing a DBS Simple?

- A DBMS is can be easily implemented!!??

  - just storing data on persistent storage devices (usually disks)
  - represent data as characters / text strings in plain files
  - use traditional file systems like FFS, LFS, etc.
  - when performing operations, just read data from the files using the underlying operating system

Megatron 2002 Relational DBS

- Store everything in their own files
- Relations stored in files (ASCII),
  e.g., relation R(a, b, c) is in /usr/db/R

```
Smith # 123 # CS
Jones # 522 # EE
...
```

- Directory (schema) stored in file (ASCII),
  e.g., /usr/db/directory

```
R # A # TEXT # B # INTEGER # C # TEXT
S # D # TEXT # E # CHARACTER
...
```

Megatron 2002: Queries

- How do you execute
  `SELECT * FROM R WHERE <condition>` ?

- It is "easy"
  - read schema from the /usr/db/directory file
  - check that <condition> is semantically valid for R
  - draw the resulting table, i.e., with columns a, b, and c
  - read data from relation R from the /usr/db/R file
  - for each line, check <condition>
  - if condition evaluates to TRUE, print tuple

Megatron 2002: Queries

- How do you execute
  `SELECT * FROM R,S WHERE a = d ?`
- Again, it is "easy"
  - read schema from file
  - join table R and S and check condition in `WHERE` clause:
    - read R file, for each line (tuple)
      - read S file and for each line (tuple):
        - create join tuple
        - check condition for resulting tuple, i.e., if "R.a" = "S.d"
        - print the tuple-pair if condition is TRUE
  - pseudo code:

```
FOR  EACH tuple x in relation R DO
        FOR  EACH tuple y in relation S DO
                IF x.a = y.d THEN print("x.a  x.b  x.c  y.d  y.e  \n");
```

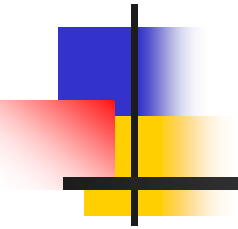
Megatron 2002: What's Wrong??? – I

- No flexibility when updating the database, e.g.,
  - if we change string from 'Cat' to 'Cats', we must rewrite file
  - ASCII storage is expensive
  - deletions are expensive
- Search expensive; no indexes, e.g.,
  - cannot find tuple with given key quickly
  - always have to read full relation
- "Brute force" query processing, e.g.,
  - do select first?
  - more efficient join?
- No appropriate buffer manager, e.g.,
  - caching – frequently accessing slow devices (disks)

Megatron 2002: What's Wrong??? – II

- No concurrency control, e.g.,
  - several users modify a file at the same time
- No reliability, e.g.,
  - can lose data

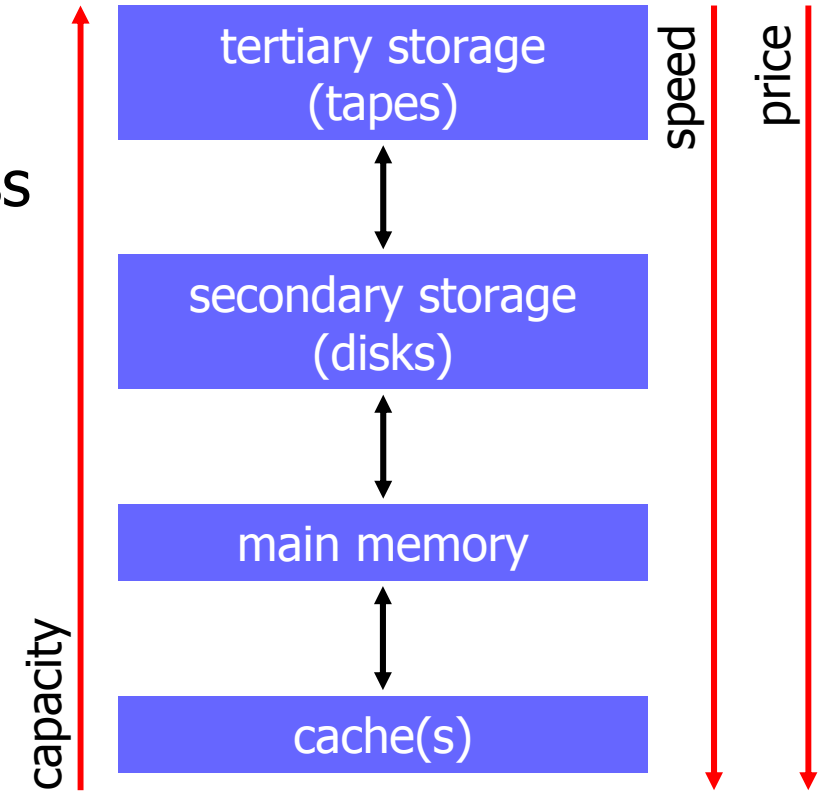
- Luckily, Megatron 2002 is a *fictitious* system
- The remainder of the course will more or less look at mechanisms that address *efficient implementations* of DBSes
- Today, we look at some hardware "facilities"



Memory Hierarchies

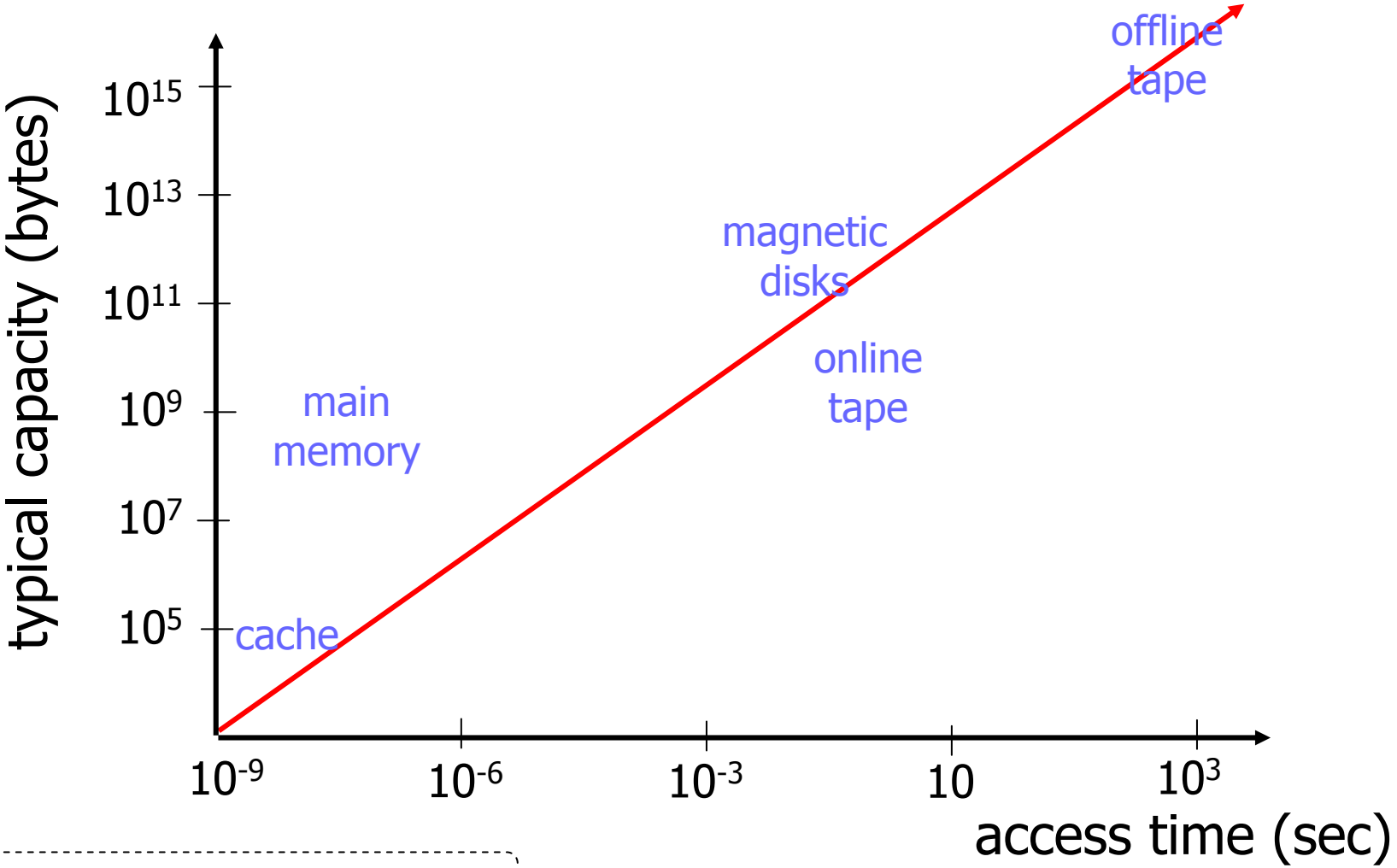
Memory Hierarchies

- We can't access the disk each time we need data
- Typical computer systems therefore have several different components where data may be stored
  - different capacities
  - different speeds
  - less capacity gives faster access and higher cost per byte
- Lower levels have a copy of data in higher levels
- A typical memory hierarchy:



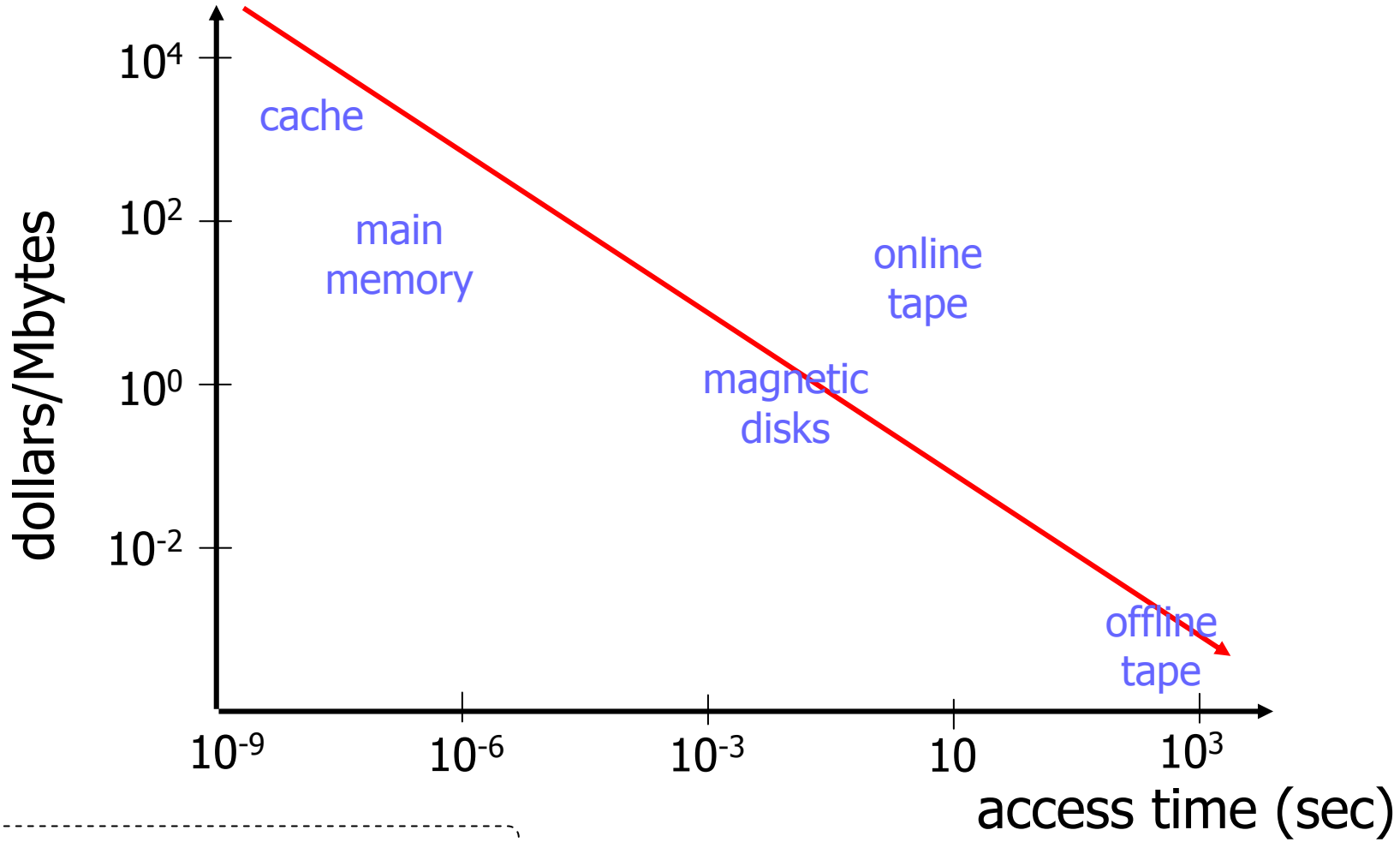
Note:
this hierarchy is sometimes reversed, i.e., cache as the highest level and tertiary storage at the bottom

Storage Costs: Access Time vs Capacity



from Gray & Reuter

Storage Costs: Access Time vs Price

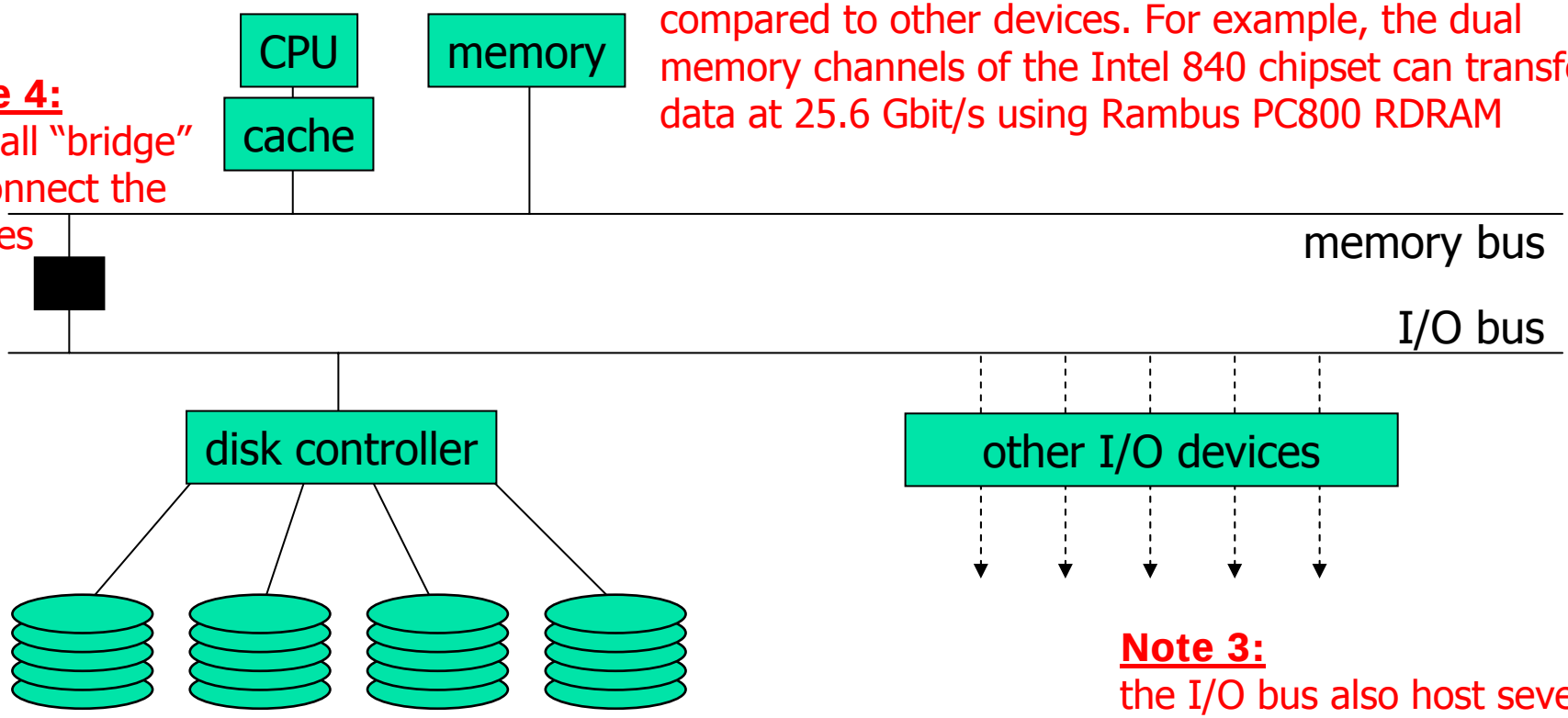


from Gray & Reuter

The Big Picture

Note 1:
the CPU and the memory is connected to a *memory bus* or *front end bus* due to speed mismatch compared to other devices. For example, the dual memory channels of the Intel 840 chipset can transfer data at 25.6 Gbit/s using Rambus PC800 RDRAM

Note 4:
a small "bridge" to connect the busses



Note 2:
secondary storage is connected to the *I/O bus*. For example, a 64 bit, 66 MHz PCI bus can at maximum transfer 4.2 Gbit/s

Note 3:
the I/O bus also host several other I/O devices such as network cards, sound cards, etc.

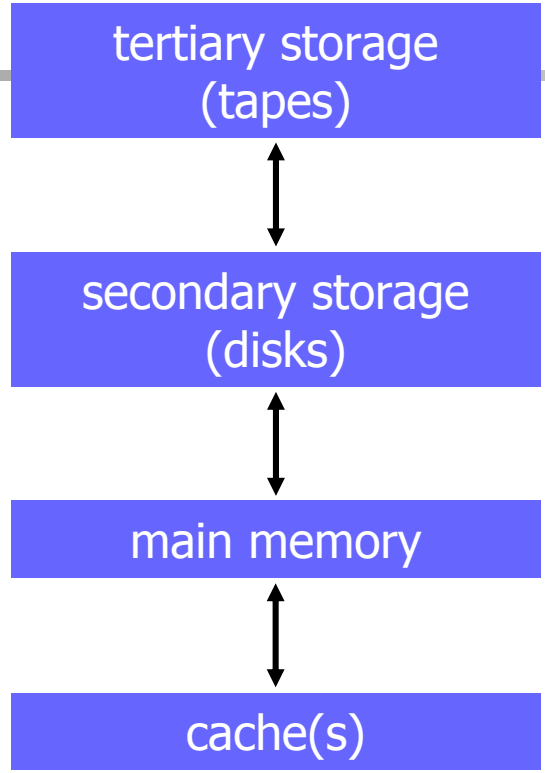
Data Movement in the Memory Hierarchy – I

- Lower level components are closer to the CPU
- Each data element in a lower level is a mapping from (a piece of) a data element in a higher level
- If data is not in the current level, we have to see in the layer above (slower, but more capacity)
- If we retrieve a data element from a higher layer, we have to replace one of the elements that already is in this component (i.e., if there are no free space)
- There exists many different algorithms for selecting an appropriate "victim" for replacement in a lower level
  - different components
  - different data access patterns
  - usually based on *reference locality* in *time* and *space*

Data Movement in the Memory Hierarchy – II

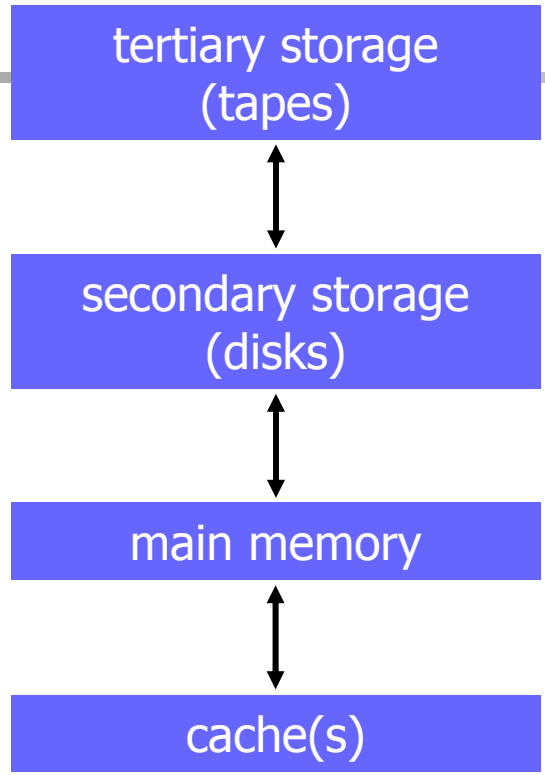
- If data is modified, we also has to modify the data elements in the higher layers to have a consistent data set
  - *delayed write* –data in higher layers are modified at a time "appropriate" for the system (typically when the system is idle)
  - *write through* – we modify data in slower components at once, i.e., right after the update (the other operations has to wait for the update operation)
  - if (1) a data element has been updated, (2) we are using delayed write, and (3) the data element is being replaced, we cannot wait for the system itself to write changes to higher levels, i.e., *it must be written back no later than replacement time*

Caches



- Caches are at the lowest level
- Often we have two levels
  - L1: on CPU chip
    - up to 32 KB or more?
    - often partitioned – data and instruction cache
  - L2: on another chip
    - up to 1024 KB or more?
    - typical access time is a few nanoseconds, i.e., 10^{-9} seconds
- If each CPU has a private cache in a multi-processor system, we must use a *write-through* update strategy

Main Memory



- Main memory is *random access*, i.e., we can usually obtain any byte in the same amount of time, but ...
  - ... accessing different parts of memory takes a different amount of time (multi-processor)
  - ... accessing a memory location close to the previous is sometimes faster than accessing one further away
- Typical numbers:
  - size from a couple hundred MB to several GB
  - access times from 10 to 100 nanoseconds ($10^{-8} – 10^{-7}$)
- The memory manager usually uses *virtual memory*

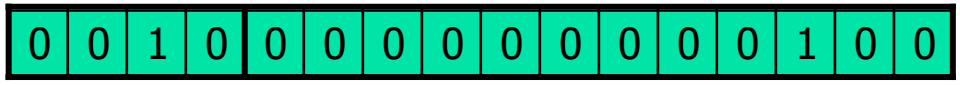
Virtual Memory

- ✓ Mono-programming
- ✓ Multi-programming with memory partitions
- ✓ As the number of concurrent processes and the amount of data used in each process increased, physical memory become a scare resource that had to be carefully managed
- ✓ *Virtual memory* introduce the idea of allowing processes use more memory than physically available
  - typical a 2^{32} bit virtual address space – 4 GB
  - (usually) much larger than available memory per process
  - the operating system keeps track those parts of data that are in memory and keeps the rest on disk, e.g., using *paging*

Paging

- The virtual address space is divided into units called *pages*, i.e., transfers between disk and memory is in units of pages
- The *memory management unit* (MMU) maps virtual addresses to physical addresses using a *page table*
- Example:
  - page size: 2^{12} (4 KB)
  - virtual address space: 2^{16} (64 KB) → 2^{4} (16) virtual pages
  - physical memory: 2^{15} (32 KB) → 2^{3} (8) physical pages

Paging: Virtual to physical Mapping



index	physical page	valid bit
0	010	1
1	001	0
2	110	1
3	000	1
4	100	1
5	101	0
...	...	...
14	111	0
15	011	0

Incoming (16-bit) virtual address: 8196

16 virtual pages
→ 16 entries in the page table
→ 4-bit index to page table

use the 4-bit index to look up physical address page table (0010 = 2)

check valid bit:
→ 1: page in memory
→ 0: page on disk (page fault)

if page in memory,
read physical page number

copy (12-bit) page offset from virtual address

Outgoing (15-bit) physical address: 24580 (page 6 – byte 4)

Paging: Page Tables

- ✓ Thus, page tables map virtual addresses to physical addresses
- ✓ Two major issues:
  1. page tables may be very large:
     - modern computers use at least 32-bit addresses
     - 4 KB pages gives 2^{20} ($\sim 10^6$) pages – and thus page table entries
     - each process has its own page table
     - ⇨ *multi-level page tables* keeping only recently used parts resident in memory
  2. mapping must be fast
     - mapping done for each memory reference
     - a typical instruction has more than one reference
     - ⇨ *associative memory* (translation lookaside buffer) working as a cache for the page table, i.e., most programs tend to make a their references to a small number of pages

Paging: Page Replacement

- As we will see later, retrieving data from disk is *very expensive* compared to memory references
- Thus, we must make a careful selection when choosing which page to replace in case of a page fault
- Many different algorithms exist based on reference locality, e.g.:
  - Traditional: random, FIFO, NRU, LRU, clock, etc.
  - "New": LRU-K, 2Q, L/MRP, interval caching, distance, etc.
- Which one that is most appropriate may differ from application to application

Paging: Page Fault Management

- If valid bit in page table is 0, a page fault occur:
  1. hardware trap (saving program counter, ...)
  2. assembly routine saves registers and volatile information
  3. operating system (OS) finds the needed virtual page
  4. OS checks whether the address is valid and access rights
  5. OS checks whether the page to replace is dirty
  6. OS finds disk address and sends a disk request

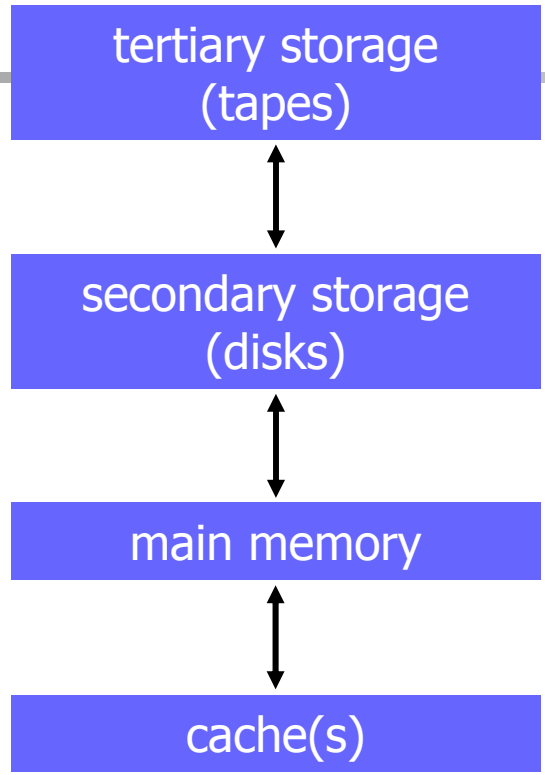
  <storage system finds disk block, rest of system continues>

  7. OS receives a disk interrupt when disk block is retrieved and page table is updated
  8. faulting instruction is backed up as ready to continue
  9. faulting instruction is scheduled
  10. assembly routine restores registers and volatile information

Memory Management and DBS

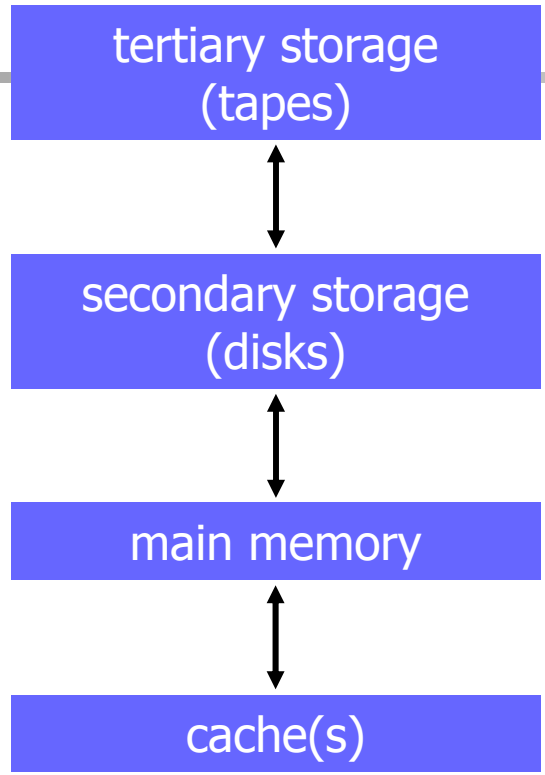
- Usually, one tries to keep the database in memory to increase performance
- A general rule is not to use a write-through updating scheme between memory and disk, but a DBS require that a change in the database state is persistent, i.e., every update must be written back to disk and logged
- *Small systems*, i.e., having less data than the size of virtual memory (~4 GB in 32-bit systems), ...
  - ...can use the OS's memory management
  - ...sometimes access the disk directly and manage the allocated memory itself due to OS and DBS requirement mismatch
- *Large systems* must manage their data directly on disk

Secondary Storage



- Secondary storage is significantly slower and significantly more capacious
- Many types: various hard disks, floppy disks, etc.
- Secondary storage are used both as persistent storage for files and to hold pages of a program's virtual memory
- Data is moved between disks and memory in blocks (a multiple of pages)
- A DBS often manage I/O by itself, but the issues are basically the same as in a file system
- Typical numbers for disks:
  - capacities of 100 GB or more
  - access times from 10 to 30 milliseconds (10^{-3} – 3 x 10^{-3}), i.e., an order of at least 10^5 (typical 10^6) slower than to main memory

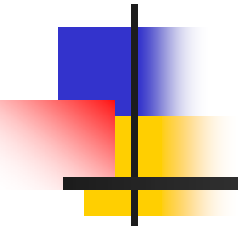
Tertiary Storage



- Some systems are too large even for large disk(s) → tertiary storage
- Tertiary storage is again significantly slower and significantly more capacious
- Many types: ad-hoc tape storage, optical juke boxes, tape silos, etc.
- Typical numbers for disks:
  - capacities of 50 GB or more per tape cassette
  - access times from a few seconds to a few minutes, i.e., an order of at least 10^3 slower than to disks

Volatile vs Nonvolatile Storage

- Volatile device: "forgets" everything when power goes off – data is cleared, e.g., main memory
- Nonvolatile devices: keep content intact even with no power – persistent, e.g., magnetic devices (disks, tapes,..)
- A DBS must retain its data even in presence of errors such as power failures
- A DBS running with volatile memory must therefore back up every change on nonvolatile devices, i.e., giving a LOT of write operations
- We therefore look closer on *disks* which are most commonly used....



Disks

Disks

- Disks are orders of magnitude *slower* than main memory, but are *cheaper* and have *more capacity*

- Disks are used to have a persistent system and manage huge amounts of information
- Because...
  - ...a DBS often manage the disks by them self,
  - ...there are a *large* speed mismatch compared to main memory (this gap will increase according to Moore's law), and
  - ...we need to minimize the number of accesses,

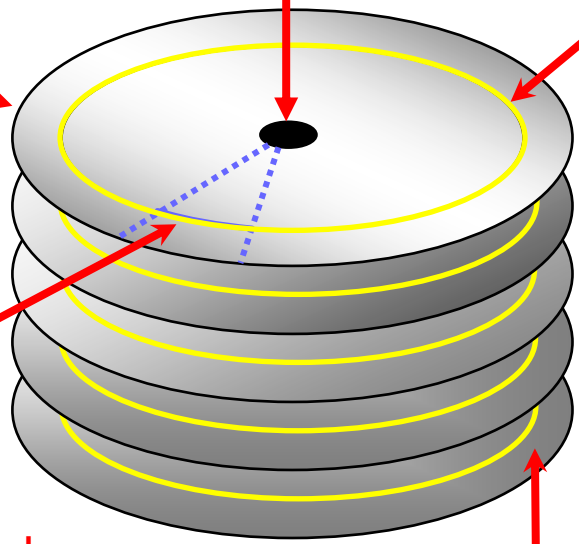
  we look closer on how to manage disks

Mechanics of Disks

Platters
circular platters covered with magnetic material to provide nonvolatile storage of bits

Spindle
of which the platters rotate around

Tracks
concentric circles on a single platter



Disk heads
read or alter the magnetism (bits) passing under it. The heads are attached to an arm enabling it to move across the platter surface

Sectors
segments of the *track* circle separated by non-magnetic gaps. The gaps are often used to identify beginning of a sector

Cylinders
corresponding tracks on the different platters are said to form a cylinder

Disk Specifications

- ✓ Disk technology develops "fast"
- ✓ Some existing (Seagate) disks today:

Note 1:
disk manufacturers usually denote GB as 10^9 whereas computer quantities often are powers of 2, i.e., GB is 2^{30}

	Barracuda 180	*Cheetah 36*	*Cheetah X15*
Capacity (GB)	181.6	36.4	36.7
Spindle speed (RPM)	7200	10.000	15.000
#cylinders	24.247	9.772	18.479
average seek time (ms)	7.4	5.7	3.6
min (track-to-tack) seek (ms)	0.8	0.6	0.3
max (full stroke) seek (ms)	16	12	7
average latency	4.17	2.99	2
internal transfer rate (Mbps)	282 – 508	520 – 682	522 – 709

Note 3:
there is a difference between internal and formatted transfer rate. *Internal* is only between platter. Formatted is after the signals interfere with the electronics (cabling loss, interference, retransmissions, checksums, etc.)

Note 2:
there might be a trade off between speed and capacity

Disk Capacity

- The size of the disk is dependent on
  - the number of platters
  - whether the platters use one or both sides
  - number of tracks per surface
  - (average) number of sectors per track
  - number of bytes per sector

Note 1:
the tracks on the edge of the platter is larger than the tracks close to the spindle. Today, most disks are *zoned*, i.e., the outer tracks have more sectors than the inner tracks

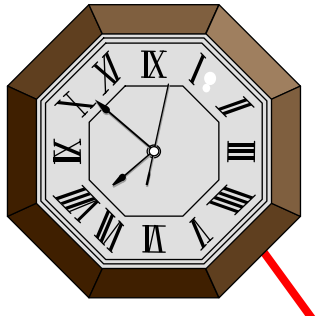
- Example (Cheetah X15):
  - 4 platters using both sides: 8 surfaces
  - 18497 tracks per surface
  - 617 sectors per track (average)
  - 512 bytes per sector
  - Total capacity = $8 \times 18497 \times 617 \times 512 \approx 4.6 \times 10^{10}$ = 42.8 GB
  - Formatted capacity = 36.7 GB

Note 2:
there is a difference between formatted and total capacity. Some of the capacity is used for storing checksums, spare tracks, gaps, etc.

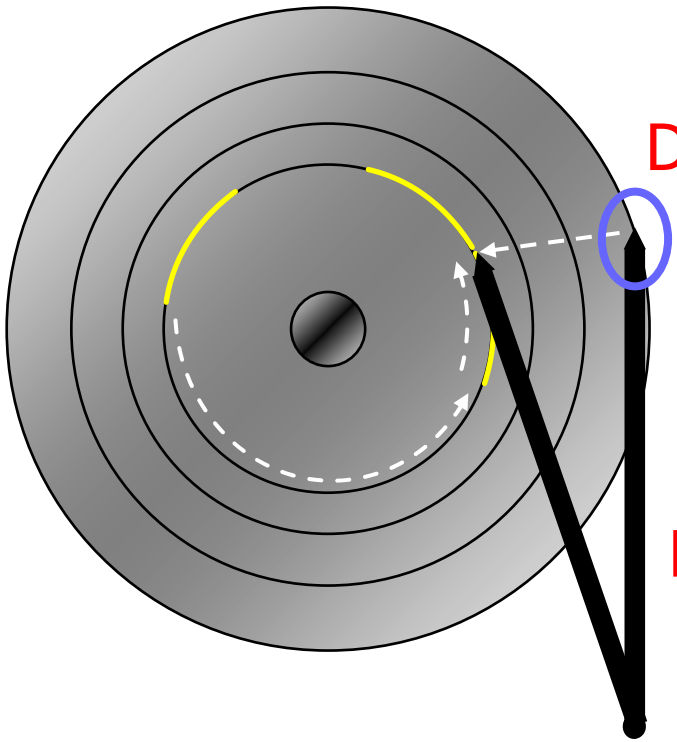
Disk Access Time – I

- How do we retrieve data from disk?
  - position head over the cylinder (track) on which the block (consisting of one or more sectors) are located
  - read or write the data block as the sectors move under the head when the platters rotate
- The time between the moment issuing a disk request and the time the block is resident in memory is called *disk latency* or *disk access time*

Disk Access Time – II

I want block X →  → block x in memory

Disk platter



Disk access time =

Seek time

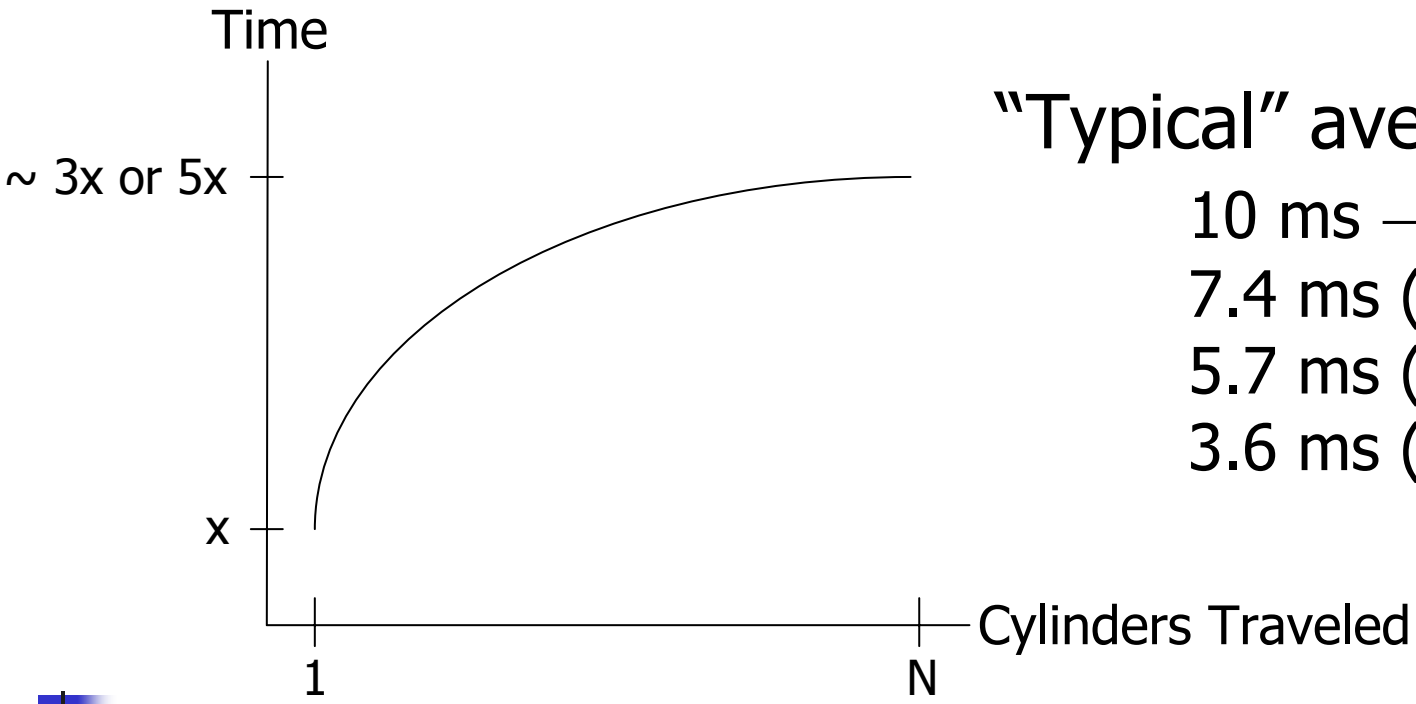
\+ Rotational delay

\+ Transfer time

\+ Other delays

Disk Access Time: Seek Time

- Seek time is the time to position the head
  - the heads require a minimum amount of time to start and stop moving the head
  - some time is used for actually moving the head – roughly proportional to the number of cylinders traveled

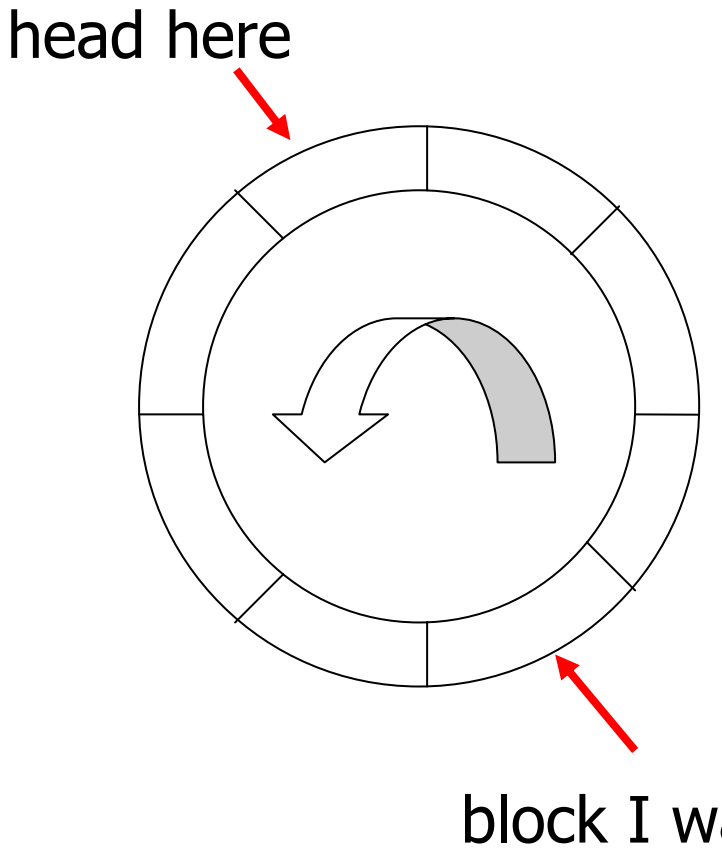


"Typical" average:

10 ms → 40 ms
7.4 ms (Barracuda 180)
5.7 ms (Cheetah 36)
3.6 ms (Cheetah X15)

Disk Access Time: Rotational Delay

- Time for the disk platters to rotate so the first of the required sectors are under the disk head



Average delay is 1/2 revolution

"Typical" average:

8.33 ms	(3.600 RPM)
5.56 ms	(5.400 RPM)
4.17 ms	(7.200 RPM)
3.00 ms	(10.000 RPM)
2.00 ms	(15.000 RPM)

Disk Access Time: Transfer Time

- Time for data to be read by the disk head, i.e., time it takes the sectors of the requested block to rotate past the head
- Transfer time = $\frac{\text{amount of data per track}}{\text{time per rotation}}$
- Example 1:
  If a disk has 250 KB per track and operates at 10.000 RPM, we can read from the disk at 40.69 MB/s
- Example 2 – *Barracuda 180:*
  406 KB per track x 7.200 RPM ≈ 47.58 MB/s
- Example 2 – *Cheetah X15:*
  316 KB per track x 15.000 RPM ≈ 77.15 MB/s

Note:
one might achieve these transfer rates reading continuously on disk, but time must be added for seeks, etc.

- Access time is dependent on data density and rotation speed
- If we has to change track, time must also be added for moving the head

Disk Access Time: Other Delays

- There are several other factors which might introduce additional delays:
  - CPU time to issue and process I/O
  - contention for controller
  - contention for bus
  - contention for memory
  - verifying block correctness with checksums (retransmissions)
  - ...
- Typical values: "0"
  (at least compared to the other factors – ns/µs vs ms)

Disk Access Time: Example – I

- Disk properties:
  - rotation speed: 7200 RPM
  - bytes per sector: 512
  - sectors per track (average): 128
  - maximum seek: 15 ms
  - average seek: 8 ms
  - track-to-track seek: 0.8 ms
  - assume 10 % of a track is used by the gap between tracks

- Example:
  Calculate time to transfer a (logical) disk block of 4096 bytes
  - we neglect the "other delays"
  - a logical block contains 8 physical sectors
  - gaps cover 36 degrees of the track, sectors 324 degrees
  - one 4096 bytes block "contains" 8 sectors and 7 gaps
    → 36 degrees x (7/128) + 324 degrees x (8/128) = 22.22 degrees
  - one rotation takes 8.33 ms

Disk Access Time: Example – II

- Example-A, minimum time:
  - head directly under first required sector
  - all sectors in same track

time = (22.22 / 360) x 8.33 ms = 0.51 ms

Note 1:
fraction of a rotation needed for our block

Note 2:
time to make one rotation

Disk Access Time: Example – III

- Example-B, maximum time:
  - head as far away required sector as possible (max seek)
  - sectors in two different tracks (track-to-track seek)
  - max rotational delay (twice)
  - transmission time as Example-A (0.51 ms)

time = (15 + 8.33 + 0.8 + 8.33 + 0.51) ms = 32.97 ms

Note 1:
time to position head (max seek)

Note 2:
max rotational delay

Note 3:
must change track in-between (track-to-track seek)

Note 4:
max rotational delay once more

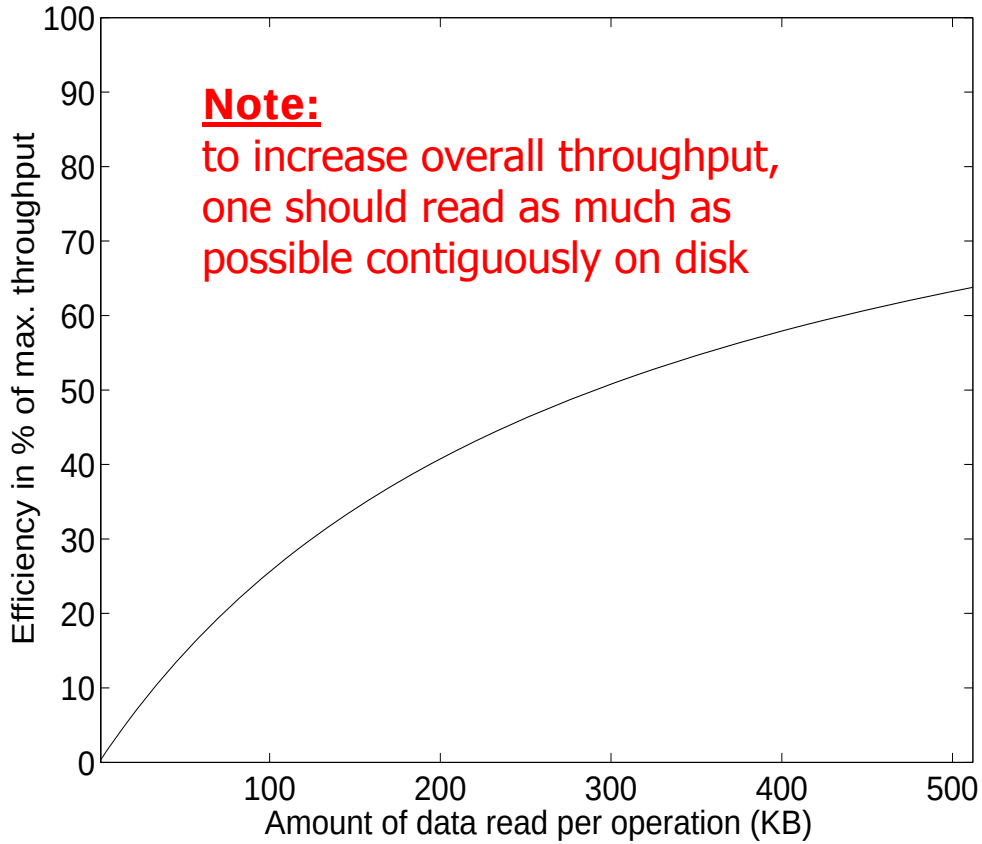
Note 5:
transmission time (time to rotate 22.22 degrees). This time will be equal even though it is split into two parts when changing track

Disk Throughput

- How much data can we retrieve per second?
- $\text{Throughput} = \frac{\text{data size}}{\text{transfer time (including all)}}$
- Example:
  for each operation we have
  - average seek
  - average rotational delay
  - transfer time
  - no gaps, etc.

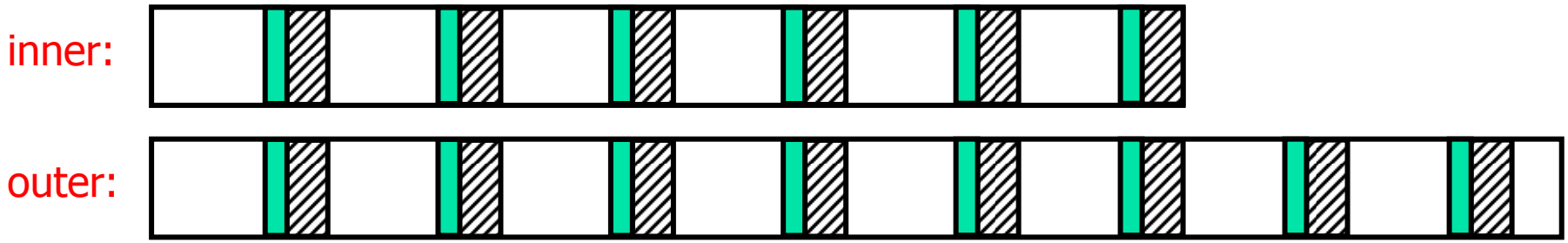
  - Cheetah X15
    4 KB blocks → 0.71 MB/s
    64 KB blocks → 11.42 MB/s

  - Barracuda 180
    4 KB blocks → 0.35 MB/s
    64 KB blocks → 5.53 MB/s



Some Complicating Issues

- There are several complicating factors:
  - The "other delays" described earlier like consumed CPU time, resource contention, etc.
  - zoned disks, i.e., outer tracks are longer and therefore usually have more sectors than inner
  - checksums are also stored with each the sectors



Note 1:
transfer rates are higher on outer tracks

Note 2:
gaps between sectors

Note 3:
the checksum is read for each track and used to validate the track

Note 4:
the checksum is usually calculated using Reed-Solomon interleaved with CRC

Note 5:
for older drives the checksum is 16 bytes

Note 6:
SCSI disks may be changed by user to have other sector sizes

Writing and Modifying Blocks

- A write operation is analogous to read operations
- A complication occurs if the write operation has to be *verified* – must wait another rotation and then read the block to see if it is the block we wanted to write
- Total write time ≈ read time + time for one rotation

- Cannot modify a block directly:
  - read block into main memory
  - modify the block
  - write new content back to disk
  - (verify the write operation)
- Total modify time ≈ read time + time to modify + write operation

Disk Controllers

- To manage the different parts of the disk, we use a *disk controller*, which is a small processor capable of:
  - controlling the actuator moving the head to the desired track
  - selecting which platter and surface to use
  - knowing when right sector is under the head
  - transferring data between main memory and disk
- Newer controllers acts like small computers them self
  - both disk and controller now has an own buffer reducing disk access time
  - data on damaged disk blocks/sectors are just moved to spare room at the disk – the system above (OS) does not know this, i.e., a block may lie elsewhere than the OS thinks

Summary

- ✓ Implementing a DBS is NOT easy!!
- ✓ Memory hierarchies
  - caches
  - main memory
  - secondary storage
  - tertiary storage
- ✓ Disks
  - characteristics
  - access time
  - throughput
  - complicating issues
  - disk controllers