DBMS & Physical Storage (With A Focus On HDD)

COM 3563: Database Implementation

Avraham Leff

Yeshiva University

avraham.leff@yu.edu

COM3563: Fall 2020

Today's Lecture: Overview

1. Introduction
2. Physical Storage Media
3. Hardware Characteristics of HDD Disks
4. Disk-Block Access: Optimizations
5. RAID
6. Some Take Away Lessons

Introduction

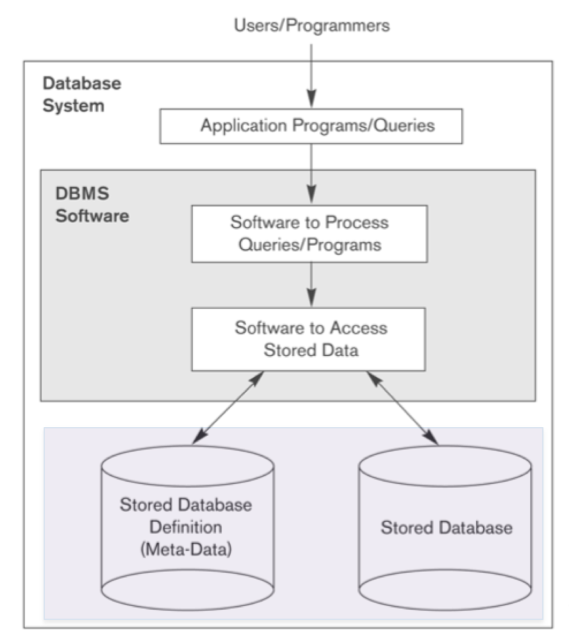
Context

- Textbook: Skipping Part 3 (*Application Design and Development*) entirely
- Textbook: Skipping Part 4 (*Big Data Analytics Development*) entirely
  - Definitely worth reading (should at least skim), simply not enough time ☹
- Major transition in course "flavor"
  - Road map so far: relational algebra, database design, normalization ...
  - Today's lecture: physical storage

Higher & Lower Layers

- Higher, logical, DBMS layer
  - Create a model of the enterprise (e.g., using E-R)
  - Create a logical “implementation”
    - Using a relational model and normalization techniques
  - Key point: this layer is created independently of any physical implementation!
- Lectures earlier in the semester focused on using SQL to write queries and to drive CUD operations from the database client to the “logical” DBMS layer
- We now examine the “lower”, physical, DBMS layer
  - Uses a file system to store the relations
  - Requires some knowledge of hardware and operating system’s characteristics
- (Note: no pejorative meaning intended with use of “higher” *versus* “lower”)

Next Few Lectures Focus on Lower Architectural Box



Physical Storage Layer: Previously Ignored In This Course

- Until now, you've been executing DBMS programs without even being aware of physical storage issues
  - *"Main-memory, disk, files . . . it's all the same to me ☺"*
- Now: as DBMS implementors (or at least "computer scientists") we have to address the following key point
- Typically, a database's data will not fit into a computer's RAM
- Implication: the data are "really" stored on other storage media
  - Brought into RAM as needed by the DBMS
- Efficient management of this data-flow to/from RAM and other storage media is crucial to system performance!

Physical Storage Layer

- Topics we'll have to understand
  - Storage media
  - File structures
  - Indexing
- We'll be making some important assumptions
- Primary storage location of the database is on non-volatile disk
- DBMS components manage movement of data between non-volatile (disk) and volatile (RAM) storage
- We'll be discussing centralized (in contrast to distributed) database architectures
- Database stores data on disk using a "standard" file system
  - Not a storage-system that is "custom-tailored" for databases
- Database uses an unmodified general-purpose OS to interact with disk

Physical Storage Media

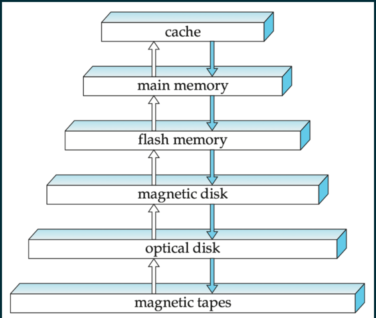
Can Classify Physical Media Along Multiple Axes

- Speed with which data can be accessed
- Cost per unit of data
- How much data are accessed at a time?
  - RAM: byte-addressable
  - disk: block-addressable
- Limitations on how data are accessed
  - RAM: random access
  - disk: sequential access
- Reliability
  - Data loss on power failure or system crash
  - Physical failure of the storage device
- Terminology
  - Volatile storage: loses contents when power is switched off
  - Non-volatile storage: contents persist even when power is switched off
    - Non-volatile storage includes secondary and tertiary storage, as well as main-memory that is backed up by e.g., battery-power

Storage Hierarchy: I

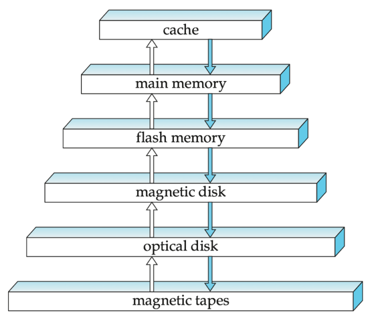
- A storage hierarchy exists because of a fundamental tradeoff between
  - Media's cost per data unit and
  - Media's performance (e.g., "bytes per second")
- In an ideal world: store database in a single "cheap & fast" media
- Currently: that world doesn't exist ☹
  - Typically, buying enough memory to store all data is prohibitively expensive
  - And: RAM is volatile, so can't serve as the "persistent" version of the data
- Implication: lower levels of the storage hierarchy serve as a cache for higher levels ...

Storage Hierarchy: II



- Cache memory: fastest and most costly form of storage; volatile; managed by the computer system hardware
- Main memory: fast access; volatile
  - 10s to 100s of nanoseconds
  - $1 nanosecond = 10^{-9} seconds$
  - Typically too small (or too expensive) to store the entire database
  - Even though "per-byte" cost continues to decrease (Moore's Law)
  - Requirements for storage capacity keep on going up ☺

Storage Hierarchy: III

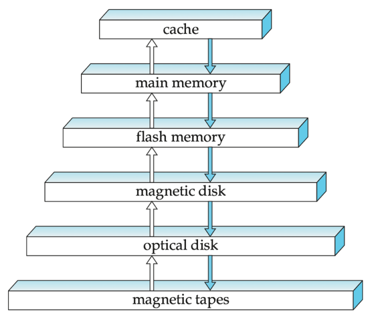


- Flash memory: fast (150 μs), survives power failure!
  - Data can be written at a location only once, but location can be erased and written to again
  - Can support only a limited number (10K - 1M) of write/erase cycles
  - Erasing memory has to be done to an entire bank (or *block*) of memory
  - Reads are roughly as fast as main memory
  - But writes are slow (few microseconds), erase is slower

Meta-point: Cost *versus* performance tradeoff isn't always straightforward …

Storage Hierarchy: IV

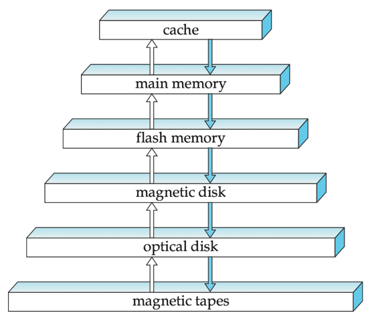
- Magnetic-disk: data are stored on spinning disk, and read/written magnetically
- Primary medium for the long-term storage of data; typically stores entire database



- Data must be moved from disk to main memory for access, and written back for storage
- Much slower access (10s of milliseconds) than main memory
- Direct-access: possible to read data on disk in any order
  - Unlike magnetic tape
- Survives power failures and system crashes
  - Disk failure can destroy data, but is rare

Storage Hierarchy: V

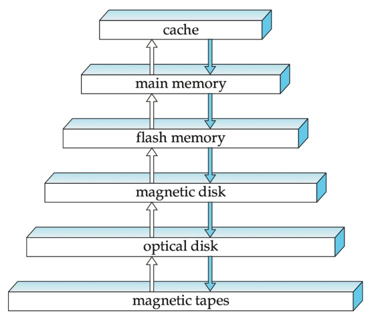
- Optical storage: non-volatile, data is read optically from a spinning disk using a laser
- Alternative to magnetic disks



- CD-ROM (640 MB) and DVD (4.7 to 17 GB) most popular forms
- Blu-ray disks: 27 GB to 54 GB
- Write-one, read-many (WORM) optical disks used for archival storage (CD-R, DVD-R, DVD+R)
- Multiple-write versions also available (CD-RW, DVD-RW, DVD+RW, and DVD-RAM)
- Reads and writes are slower than with magnetic disk
  - Frustrating: couldn't get "hard-numbers" ☹

Storage Hierarchy: VI

- Tape storage: non-volatile, used primarily for backup (to recover from disk failure), and for archival data



- Sequential access: much slower than disk
- Very large capacity: 40 to 300 GB tapes available
- Tape can be removed from drive, so storage costs much cheaper than disk
  - But tape drives are more expensive than disk drives
- Tape jukeboxes available for storing massive amounts of data
  - Hundreds of terabytes ($1 terabyte = 10^9 bytes$) to even multiple petabytes ($1 petabyte = 10^{12} bytes$)

Storage Hierarchy: Summary

- Primary storage: Fastest media but volatile (cache, main memory)
- Secondary storage: next level in hierarchy, non-volatile, moderately fast access time
  - Also called on-line storage
  - Examples: flash memory, magnetic disks
- Tertiary storage: lowest level in hierarchy, non-volatile, slow access time
  - Also called off-line storage or “cold storage”, used to restore a system if something bad happens
  - Alternatively: unusually large (often archival in nature) data-sets
  - Examples: magnetic tape, optical storage

Sequential *Versus* Random Access

- Random access on an HDD is much slower than sequential access
  - Disk "seek" ≈ 2ms
  - Disk read of a million bytes: ≈ 825 μsec
- Implication: DBMS are designed to maximize sequential access
  - Algorithms try to reduce number of writes to "random" pages so that data is stored in contiguous blocks

DBMS Design Goals

- Allow the DBMS to manage databases that exceed the amount of RAM memory available
- Reading/writing to disk is expensive, so it must be managed carefully to avoid
  - Large “stall” episodes in which system is only transferring data back and forth (not doing anything useful from the client point of view)
  - Performance degradation (reducing throughput and increasing latency)

Before We Leave This Topic: I

	Disk (HHD/SSD)	LTO (Tape)	Cloud
Initial Investment	Moderate	High	Low, but constant expense
Maintenance Cost	High	Moderate	Included
Expandability	Easy to Moderate depending on workflow and set-up	Easy to add additional tape media	Easy, but incurs higher monthly fees
Access Time	Fast	Moderate (with loading time)	Slow (depends on connection)
Sensitivity	High sensitivity, relatively high failure rate, maintenance required for off-site and long-term storage	Low sensitivity, low failure rate, low maintenance for off-site and long-term storage	Low sensitivity, low failure rate, low maintenance for off-site and long-term storage
Ideal Uses	Good for backup and failover, long-term storage for moderate amounts of data	Good for long-term archiving of large amounts of date	Good for archiving and backup if quick retreval is not required, cost prohibitive for large amounts of data

Source: Disk vs Tape vs Cloud: What Archiving Strategy is Right for Your Business? February 20, 2018

Not vouching for accuracy, but seems on-target ☺

Before We Leave This Topic: II

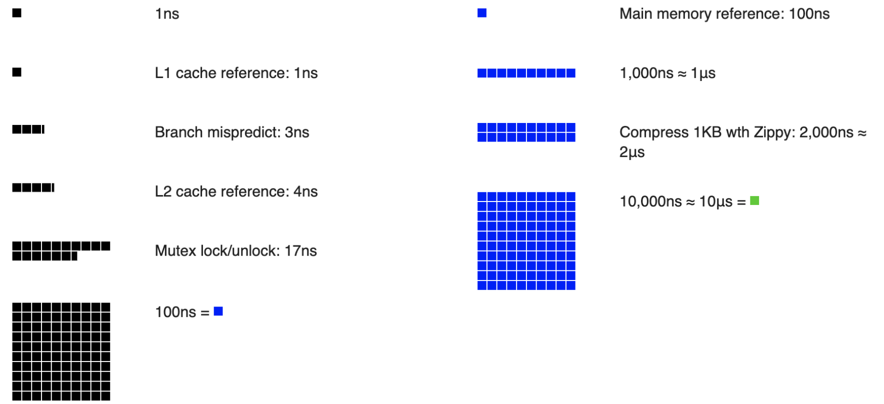
```
L1 cache reference ......................... 0.5 ns
Branch mispredict .......................... 5 ns
L2 cache reference ......................... 7 ns
Mutex lock/unlock ......................... 25 ns
Main memory reference ...................... 100 ns
Compress 1K bytes with Zippy ............. 3,000 ns  =   3 µs
Send 2K bytes over 1 Gbps network ....... 20,000 ns  =  20 µs
SSD random read ........................ 150,000 ns  = 150 µs
Read 1 MB sequentially from memory ..... 250,000 ns  = 250 µs
Round trip within same datacenter ...... 500,000 ns  = 0.5 ms
Read 1 MB sequentially from SSD* ..... 1,000,000 ns  =   1 ms
Disk seek ........................... 10,000,000 ns  =  10 ms
Read 1 MB sequentially from disk .... 20,000,000 ns  =  20 ms
Send packet CA->Netherlands->CA .... 150,000,000 ns  = 150 ms
```

Source: Latency numbers every programmer should know (circa 2012)

Not vouching for accuracy, but seems on-target ☺

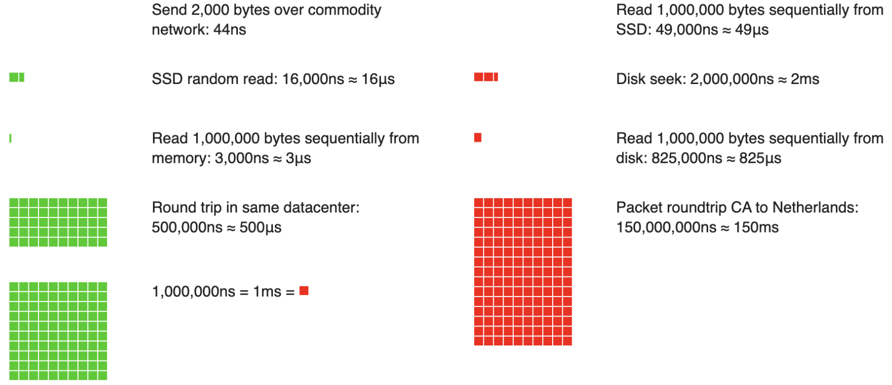
Updated For 2020: Registers & Main-Memory

See this interactive (drag the slider to set the “year”) latency numbers display



Updated For 2020: Network & Disk

See this interactive (drag the slider to set the “year”) latency numbers display



Hardware Characteristics of HDD Disks

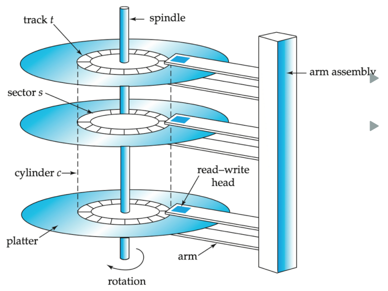
If you have a "pro-software bias", easy to let your eyes glaze over here ☺

Make an effort! Hardware characteristics drive the basic algorithms in the "DBMS file storage and access" space

- Magnetic disks still provide the bulk of DBMS secondary storage
- Most of your experience with data access has tended to be with "main memory"
- So: let's begin by discussing how magnetic disk storage works
  - At a very high-level ☺

Magnetic Hard Disk Mechanism: I

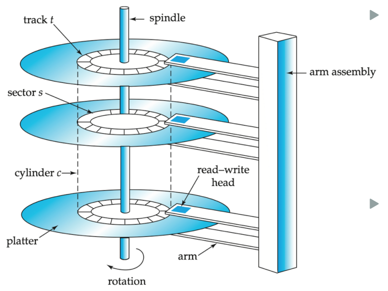
Read-write head is positioned very close to the platter surface (almost touching it): reads or writes magnetically encoded information.



- Surface of platter divided into circular tracks
  - Over 50K-100K tracks per platter on typical hard disks
- Each track is divided into sectors
- A sector is the smallest unit of data that can be read or written
  - Sector *size* typically 512 bytes
  - *Number* of sectors per track: typically 500 to 1000 (on inner tracks) to 1000 to 2000 (on outer tracks)

Magnetic Hard Disk Mechanism: II

To read/write a sector disk arm swings to position head on the correct track. The platter spins continually; data is read/written as sector passes under head



- Head-disk assemblies
  - Multiple disk platters on a single spindle (typically 1 to 5)
  - One head per platter, mounted on a common arm.
- Cylinder *i* (a "virtual concept") consists of i_{th} track of all the platters

Disk Controller

Disk controller: interfaces between the computer system and the disk drive hardware

- Accepts high-level commands to read or write a sector
- Initiates actions such as moving the disk arm to the right track and actually reading or writing the data
- Computes and attaches checksums to each sector to verify that data is read back correctly
  - If data are corrupted, with very high probability stored checksum won't match recomputed checksum
- Ensures successful write by reading back sector after writing it
- Performs remapping of bad sectors
  - We'll discuss the mapping concept later

Accessing a Disk Block

- I/O time dominates the time taken for database operations!
- To minimize I/O time, the DBMS must make "good decisions" about
  - Where to store data on disk
  - How to reduce the time needed to locate that data
- Disk I/O time is comprised of
  - The time to move disk heads to the track on which a desired block is located
  - The waiting time for the desired block to rotate under the disk head
  - The time to actually read or write the data in the block once the head is positioned

Disk I/O time = seek time + rotational time + transfer time

Disk Storage: Performance Metrics (I)

- Access time: time it takes from when a read or write request is issued to when data transfer begins
- Access time is comprised of
  - Seek time: time it takes to reposition the arm over the correct track
    - Average seek time is 1/2 the worst case seek time
    - Would be 1/3 if all tracks had the same number of sectors, and we ignore the time to start and stop arm movement
    - 4 to 10 ms on typical disks
  - Rotational latency: time it takes for the sector to be accessed to appear under the head.
    - Average latency is 1/2 of the worst case latency
    - 4 to 11 ms on typical disks (5400 to 15000 rpm)

Disk Storage: Performance Metrics (II)

- Data-transfer rate: rate at which data can be retrieved from or stored to the disk
  - 25 to 100 MB per second maximum rate, lower for inner tracks (because they have fewer sectors)
- Multiple disks may share a controller, so rate that controller can handle is also important
- Examples
  - SATA: 150 MB/sec, SATA-II 3Gb (300 MB/sec)
  - Ultra 320 SCSI: 320 MB/s, SAS (3 to 6 Gb/sec)
  - Fiber Channel (FC2Gb or 4Gb): 256 to 512 MB/s

- As we've seen, seek time and rotational delay dominate HDD I/O time
- Implication: to reduce I/O time, DBMS should reduce seek & rotational delay
  1. Place blocks on the same track
  2. Place blocks on the same cylinder
  3. Place blocks on <u>adjacent</u> cylinder

In other words: sequential arrangement of file blocks can be a "big win" for a DBMS

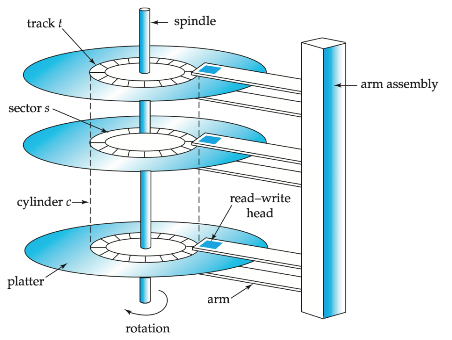
Disk-Block Access: Optimizations

Requests for Disk-Blocks

- Requests for "data on disk" must reference a given address
- This address is expressed as a block number
- Given a block number, the disk subsystem will return a fixed (quite small) number of bytes, beginning at the specified address
- Example: the cylinder-head-sector (CHS) scheme specifies block addresses using a triple
  1. cylinder number (essentially "track number")
     - Number of cylinders of a disk drive is the number of tracks on a single surface in the drive
     - A given cylinder corresponds to a given track number for all the platters in the disk-storage system
  2. head number (essentially "which platter")
  3. sector number (within a given track)
- CHS was replaced by the logical block addressing scheme
  - Blocks are specified using a single integer index

What Sort Of Optimizations Do We Need?

- You're used to main-memory (RAM) addressing
  - Offers byte-addressable & random access capabilities



- Disk access differs from RAM in more ways than just access rate
- Disks are rotating media
- Seek time: "the time it takes the head assembly on the actuator arm to travel to the track of the disk where the data will be read or written"

- Sequential access pattern
  - Successive disk-block requests are for successive block numbers
  - Same track, adjacent track …
  - Seek time penalty is relatively small (first seek is amortized over multiple requests)
- Random access pattern
  - Successive disk-block requests are for blocks that are randomly located on disk
  - Each request incurs seek penalty
  - We need optimizations that address this problem

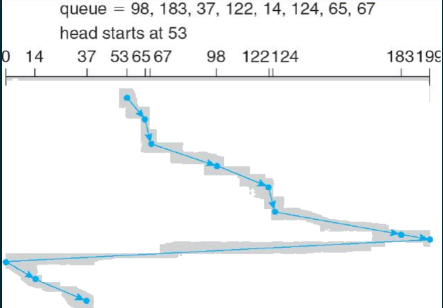
"Read-Ahead" Optimization

- Read-ahead: don't just transfer requested block into main-memory!
- Read the "next n blocks" as well ...
  - You've already paid the "seek time" penalty
  - Amortize that penalty over n blocks
- One problem: who says those "next blocks" will be used?
- Good point! Only useful for sequential access pattern
- Note: optimization applies to both OS and DBMS

Disk-Arm Scheduling

- This technique explicitly addresses the fact that disk storage cannot access block addresses in random order
  - Instead: must first do a seek operation
- Key idea: rearrange a set of disk-block requests so that they get serviced in the order that the disk heads will pass over the requested blocks
  - Optimize to minimize disk-arm movement
  - Order in which the requests were made? irrelevant ☺

Elevator Algorithm



- So-called because also (used to be?) used to schedule elevator movement
    1. Move from current track towards the outside of the disk
        - Servicing block requests as you go ...
    2. When no more requests reference blocks in the outside of the disk
    3. Reverse direction, move from outside → to inside
        - Servicing block requests as you go ...

File (Re)Organization

- The file organization technique attempts to improve block access time by organizing the blocks to correspond to how data will be accessed
- Example: store related information on the same or nearby cylinders
- Remember: DBMS has past history on how file was accessed, what file accesses were followed by other file accesses
- Problem: Files may get fragmented over time
  - Example: data are is inserted into or deleted from the file
  - Example: because free blocks are scattered on disk → new files have their blocks scattered over the disk
- Defragmentation technique: coalesce a file's disk blocks so that they're nicely organized again

Techniques To Get Disks Behave Like RAM: I

Use (non-volatile) RAM for write operations ☺

- Key idea: complete main-memory as soon as possible, have disk get around to it later
  - Definition: non-volatile RAM: RAM that's backed up with a battery or flash memory
- DBMS first writes blocks to a non-volatile RAM buffer
  - Data are now safe (even if power fails)
- Disk controller writes data in RAM buffer to disk whenever it makes time for the request
- In the meantime, DBMS continues processing other requests

Techniques To Get Disks Behave Like RAM: II

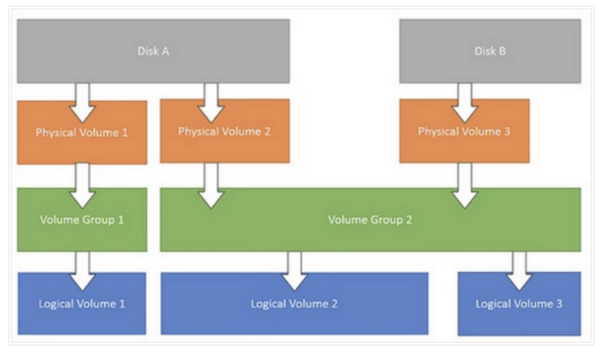
- All disk block updates are written in sequential order to a log disk
  - Dedicated to this function, so can just keep on appending
- Improves performance because no need to do any seeks!
  - Making a disk behave like non-volatile RAM ☺
- If DBMS crashes while doing “regular” (seek-based) disk writes
  - No problem: when DBMS comes back up, reads and applies the data in the log file
- FYI: file systems that use this technique are called journaled file systems
  - System has recorded its intentions in a journal

RAID: Benefits Of Using Many Disks

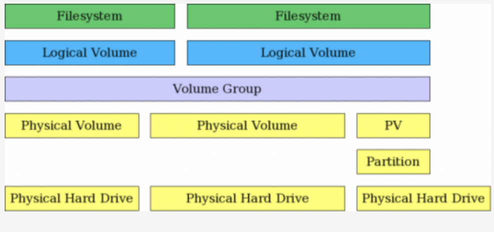
- Previous slides: HDD provide cheap, non-volatile storage for DBMS, but HDD are a performance bottleneck
- Q: will using multiple disks make the problem better or worse?
- A: "multiple disks", cleverly deployed, are the basis of the RAID technology solution
  - RAID: *Redundant Array of Inexpensive Disks*
  - RAID: *Redundant Array of Indepedent Disks*
- Key ideas: multiple disks enable DBMS
  - To store data redundantly: if one disk fails, the data can be found on another disk (a reliability benefit)
  - To make multiple disk requests in parallel (a performance benefit)
- All this besides the obvious benefit of being able to store more data than can be stored on one disk (a capacity benefit)

Logical Volume Managers (LVMs) (I)

- Q: given that "disk addresses" in a file system refer to an address in a single disk, how can multiple disks be harnessed in a practical way?
- A: the usual solution, an abstraction layer, specifically a logical volume manager (or LVM)
  - An LVM provides an abstraction that makes multiple disks appear as a single disk



Logical Volume Managers (LVMs) (II)



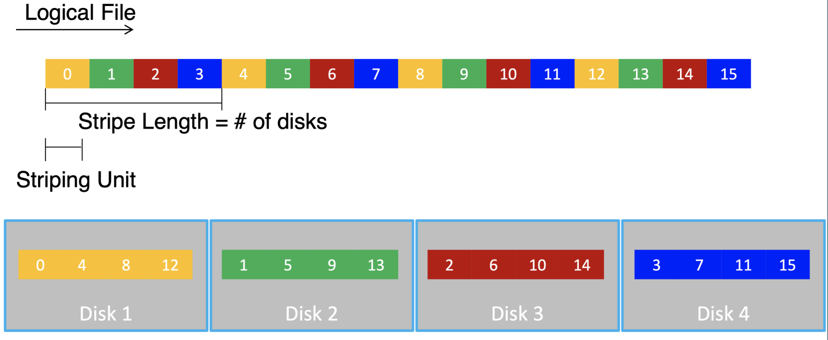
- An LVM allows combining partitions and entire hard drives into volume groups
- In this figure, two complete physical hard drives and one partition from a third hard drive have been combined into a single volume group
- Two logical volumes have been created from the space in the volume group
- A filesystem (e.g., EXT3 or EXT4) has been created on each of the two logical volumes

LVM Capabilities

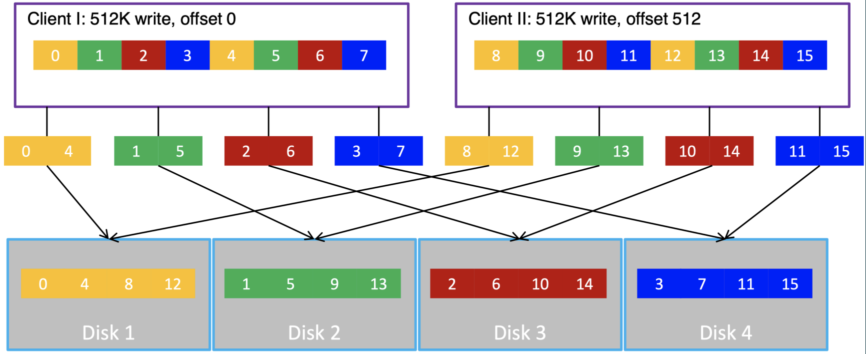
- LVMs provide a spanning capability
  - Transparently map a larger address space to different disks
- And also provide a mirroring capability
  - Each disk can hold a separate, identical copy of data
  - The LVM directs writes to the same block address on each disk
  - LVM directs a read to any disk (e.g., to the “less busy” disk)

Parallelism: Data Striping (I)

- To achieve parallel data access, we use a technique called data striping
- Typically, a stripe unit is either:
  - A bit ("bit interleaving")
  - A byte ("byte interleaving")
  - A block ("block interleaving")
- Key point: each stripe is written across all disks at once



Parallelism: Data Striping (II)



How Large Should We Set The "Stride Unit"?

- If we set a "small" striping unit value ...
  - Increases parallelism
  - Less data to transfer
  - But: increased seek & rotational delays
- The advantages/disadvantages from setting a "large" striping unit are the converse of the above points
- Additional advantages from setting a "large" striping unit
  - May be able to completely satisfy a request with a <u>single</u> disk
  - Increase the concurrency factor by satisfying multiple requests simultaneously

Data Striping: Performance Benefits

- Two potential benefits
  - Load balance multiple, small, disk access requests
    - Increase throughput
  - Parallelize large disk access requests
    - Reduce response time (latency)

Redundancy

- Mean time to failure (or MTTF): average amount of time a single disk is expected to run continuously without any failure
  - Typically 3 to 5 years
- But: the probability that one disk out of a set of *n* disks will fail is much higher than the probability that a specific single disk will fail
  - If probability of one disk failure is f, then probability of overall system failure is $(1 - (1 - f)^n)$
  - Example: a system with 100 disks, each with MTTF of $100,000$ hours ($\approx$ 11 years)
  - Overall system has an MTTF of 1000 hours ($\approx$ 41 days)
  - So: overall, not as impressive as it sounds ☺
- Conclusion: we must use "redundancy techniques" to avoid data loss when the system uses large numbers of disks

Reliability *Via* Redundancy (I)

- Mirroring (or “shadowing”)
- Duplicate every disk: e.g., every “logical disk” consists of two “physical disks”
- Every write is performed on both disks
  - Reads can be done from either disk
- If one disk in the pair fails, data still available in the other
- Actual data loss occurs only if
  - One disk fails
  - And: its mirror disk <u>also fails</u> before the other disk is repaired
- Probability of combined event is very small
  - Note: doesn’t help for dependent events such as fire or building collapse or electrical power surges ☹

Reliability *Via* Redundancy (II)

- MTTR ("mean time to repair")
- The "mean time to data loss" depends on both the MTTF and the MTTR
- Example: given a
  - MTTF of $100,000$ hours
  - MTTR of 10 hours
- The mean time to data loss is 500×10^6 (or 57 years ☺) for a mirrored pair of disks
  - Again: assuming independent failure modes for the two disk

RAID "Levels" (I)

- People have devised different ways to benefit from RAID: we refer to these as RAID levels
  - These levels have differing cost, performance and reliability characteristics
- The idea is to provide redundancy at lower cost by using disk striping combined with parity bits (next slide)
- RAID level 0: provides block striping in non-redundant fashion
  - Typically used in high-performance applications where data loss is not critical
- RAID level 1: provides block striping <u>and</u> mirrored disks
  - Best write performance for a "mirrored" approach (see next slides)
  - Popular for applications such as storing log files in a database system

RAID "Levels" (II)

- There are 7 RAID levels, but we'll ignore levels 2-4 because they aren't used nowadays
- Other RAID levels involve the notion of parity block
- For a given set of blocks, we can compute (and store) a parity block
- The i_{th} bit of the parity block is the XOR of the i_{th} bits of all of the blocks in the set
- When writing a new block, we must recompute the corresponding parity block
  - Can do an XOR of the previous parity block, the previous value of the "data" block and the new value of the "data" block (involves 2 block reads + 2 block writes)
  - Or: by reading all blocks in the set, and recomputing the parity block from scratch
- Key benefit: if the contents of a single block are lost, can be recovered by doing an XOR of the remaining blocks with the parity block ☺

RAID "Levels" (III)

- RAID level 5: provides block-interleaved distributed parity
- Data and parity are partitioned among $n + 1$ disks
- That is: given n "logical blocks" ...
  - One disk stores the parity bit
  - The other disks store the n logical blocks
  - Parity bits for different logical blocks are stored on different disks
- Example: with 5 disks, parity block for i_{th} set of blocks is stored on disk$(i \bmod 5) + 1$
  - The data blocks stored on the other 4 disks
- Note: a parity block cannot store parity for "data blocks" on the same disk: otherwise a disk failure will destroy the parity information as well as the data ☹

RAID "Levels" (IV)

- Yes, there is a RAID level 6 ☺
- Extends RAID level 5 by adding another parity block
  - Uses block-level striping with two parity blocks distributed across all member disks
  - Protects against (even) two concurrent disk failures
- Some observations
  - RAID level 1 provides much better write performance than level 5 (because no parity bits are computed)
    - Level 5 requires (at least) 2 block reads and 2 block writes per block written; level 1 only requires 2 block writes
  - Use level 5 for applications with high read ratios
  - The penalty for level 5 writes is mitigated when applications perform sequential writes
    - The parity can usually be computed from the newly written blocks

Some Take Away Lessons

- In case you didn't realize ☺
  - We didn't discuss all that much about "hardware" and "how do DBMS physical storage media work"
- Intent was to make you aware that a DBMS can't just be "software"
- And that using hardware effectively means reading and understanding the implications of many "low-level" details
- You can count on the hardware characteristics of a DBMS storage system changing over time
- But slowly … if only because businesses are
  - Reluctant to throw away existing infrastructure investments
  - Conservative with respect to adopting new technologies
    - Are averse to "shiny object syndrome" ☺

- For foreseeable future, we'll always have a memory hierarchy
  - Small(er) amounts of expensive, very fast, memory
  - Larg(er) amounts of cheaper, (relatively) slower, memory
- Many of the techniques discussed in today's lecture will be useful to you even when faced a with "different" memory hierarchy
- Key point: although the tradeoffs (or the inflection points) will probably differ from today's discussion ...
  - ...If you understand the "why" for the algorithms, you'll be able to apply them usefully to new memory hierarchy and environment

Example

- Flash ("solid state") memory is becoming a serious alternative to magnetic disk storage
- Random reads/writes per second (from the textbook)
  - Typical 4KB reads: 10,000 (10,000 IOPS)
  - Typical 4KB writes: 40,000 IOPS
- SSDS support parallel reads (here are some numbers for 4KB reads)
  - 100,000 IOPS with 32 requests in parallel (QD-32) on SATA
  - 350,000 IOPS with QD-32 on NVMe PCIe
  - SSDS 4KB writes: 100,000 IOPS with QD-32, even higher on some models
  - Data transfer rate for sequential reads/writes: 400MB/sec for SATA3, 2 to 3 GB/sec using NVMe PCIe
- Flash memory doesn't have to do the sort of sequential seek that disks do
  - In other words: think of flash as "slower main memory"
- Flash memory lies somewhere in-between RAM and disk in the cost *versus* speed tradeoff

War Story

- Back in 2010, some smart guys in IBM saw an opportunity in flash memory technology …
- Inverted the usual database paradigm, and explored the idea of offloading from main-memory to disk
- Idea: enhance the Memcached "object caching" system to offload to *solid-state drive* if too many objects in main-memory
- Dragged IBM Research into this exploration: Disk-Offload Middleware for Web-Services Using the Application-Caching Paradigm
- Similar ideas in Redis nowadays …
- Key point: new storage technologies allow you to find new opportunities for your DBMS's capabilities
  - To exploit: you must have a real understanding of material discussed today

Readings

- The textbook discusses DBMS *Physical Storage* in Chapter 12
- The textbook discusses *Data Storage Structures* in Chapter 13
- Textbook, Chapter 10 through 10.2, skim 10.2.4
- Not responsible for Chapter 10.3 (RAID material) and Chapter 10.4 (beyond the basic concepts and terminology)
- Textbook, Chapter 10.5 - 10.7: next lecture
- Textbook, Chapter 10.8, this lecture