# CS647 Modern Storage Systems

- Logistics
- Introduction (Fast-forward to Key-Value Stores)

CS647 Modern Storage Systems, Fall 2023
(Modern Key-Value Store Design and Implementation)
Angelos Bilas, bilas@csd.uoc.gr
Computer Science Department, University of Crete

# CS647: The Course

- Logistics
  - Instructors: Angelos Bilas, Georgios Saloustros
  - TA: Georgios Saloustros
  - Web: http://www.csd.uoc.gr/~hy647
  - Mailing lists:
    - Full class: hy647-list@csd.uoc.gr
      - Send mail to majordomo today
    - Instructors + TAs: hy647@csd.uoc.gr
- Work
  - Lectures
  - Guided paper discussion
  - 3-4 assignments (one every ~2 weeks)
  - 1 project (proposed by students, ~3-4 weeks)
  - Project presentations in last class
  - No final exam
- 1st assignment available on web page, due +2 weeks

# Assignment 1: Simple blob store

- Write a library for a simple “blob” store + workload generator
- Blob library
  - A blob is a set of (fixed-size) blocks with few/simple operations
    - create, delete, read/write full blob
  - Impl. 1: use the filesystem and one blob per file
    - FS will do allocation/freeing of space on device
  - Impl. 2: allocate blobs yourself over a block device (or in a single file)
    - You need an allocator to know which blobs/blocks are free
    - Each blob can consist of consecutive blocks on the device
  - Shutdown and restart a store (no failures yet)
  - Extra credit: implement asynchronous I/O operations for read/write
- Workload generator (driver program)
  - Create a store with N blobs of size B
  - Issue I/Os to blobs: read/write, random/sequential, K threads
    - (extra) asynch I/O from 1 thread with increasing queue depth as well
  - Plot results for ramdisk, HDD, SSD, NVMe, (NVM)
- Questions to Giorgos Saloustros

# Storage Eras (pre-cloud)

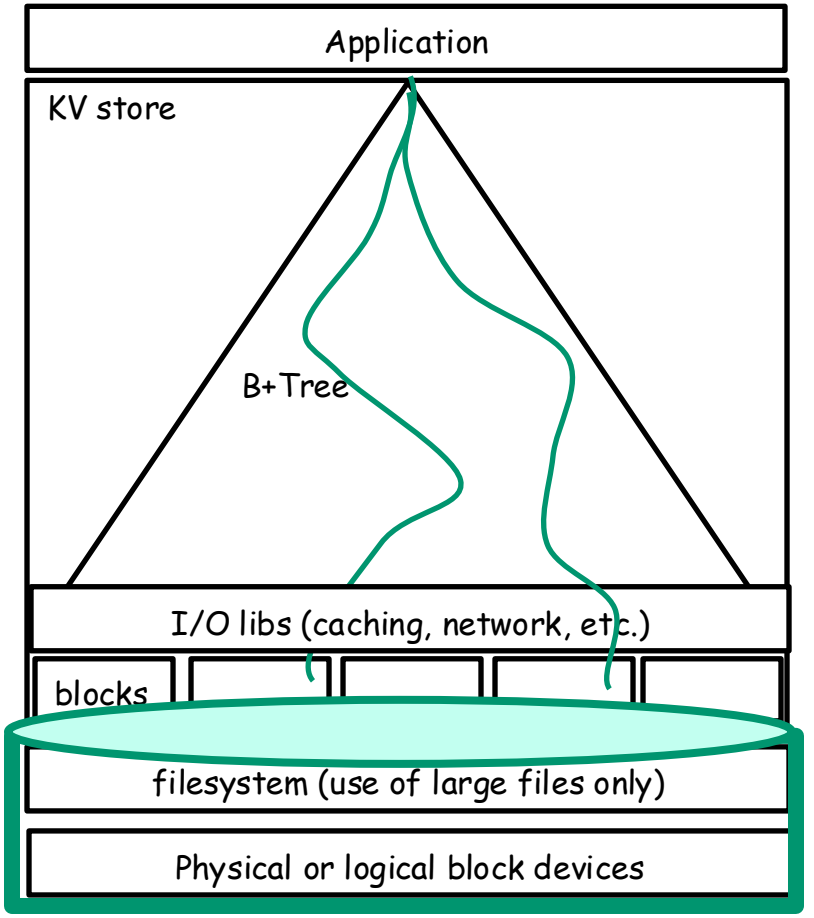
- ...–1985: **Device era**
  - Trend: HDDs in PCs/workstations
    Full I/O stack already available: block, fs, I/O libs (not objects)
    People complain about I/O → I/O is important (even then)
  - Effort: Functionality, e.g. recovery, performance (xput, latency), capacity optimization
  - Result: I/O is slow – storage is expensive
- 1985–1995: **Parallel era**
  - Trend: Throughput is about parallelism → use many devices (e.g. RAID)
  - Effort: Take advantage of parallelism in the I/O stack
  - Result: + You can have a lot of throughput, at cost close to the #devices
    (This works well for large/concurrent/asynchronous IOs)
    - I/O is also about small persistent writes and latency remains high
- 1995–2005: **Network era**
  - Trend: Shared storage has many benefits → commodity networks become fast
  - Effort: Network vs. direct attached storage, distribution, consistency, scaling
  - Result: + You can have shared capacity, high I/O throughput
    (OK when there is single "owner/user" for each data item = share capacity but not data)
    - "True, distributed sharing" is difficult to scale
- 2005–2015: **Flash era**
  - Trend: NAND FLASH devices appear (at high capacity, otherwise available late '80s)
  - Effort: Devices, adaptation of software stack, deep(er) hierarchies
  - Result: + You can have shared capacity, high I/O throughput, low avg latency
    - CPU cycles are the I/O bottleneck

# Storage Eras (post-cloud)

- 2015–2025: **Cloud era (it's all about the workload)**
  - Trend: Workloads change dramatically
    Consolidation of workloads and sharing of resources at scale
  - Effort: In all layers of the stack → Partitioning, Interference
  - Result: + Improved performance under mixed/collocated workloads
    - Load still grows faster than anything else – storage is expensive
- Today (gather all "-"): (Effort to design such systems non-trivial)
  - Throughput scales with parallelism over devices + network
  - Shared capacity but we really partition to address scaling
  - Use FLASH for avg latency but tail latency is still poor
  - Consume a lot of CPU for I/Os – main bottleneck today
  - Cannot keep up with load - grows faster than anything else
  - Storage is still expensive and remains a bottleneck !
- **In the course we will discuss (understand) where we stand today**
  - Not as much what has happened until now
- 2025–?: **Workload specialization (AI) – basic questions**
  - Is it about small or large requests (or both)?
  - It is about throughput or latency (or both)?
  - Is it about storage access or memory access (or both)?

# Storage Stack Then (=FS) and Now (=KVS)

- Then
  - Application
  - I/O libs (cache, network)
  - OS (block + FS)
  - Device (+ firmware)
- Now
  - Application
  - Key-value store (KVS)
  - I/O libs (cache, network)
  - OS (block + FS)
  - Device (+ firmware)
- Database?
  - About arbitrary transactions



# Is the Storage Stack merely an "Index"?

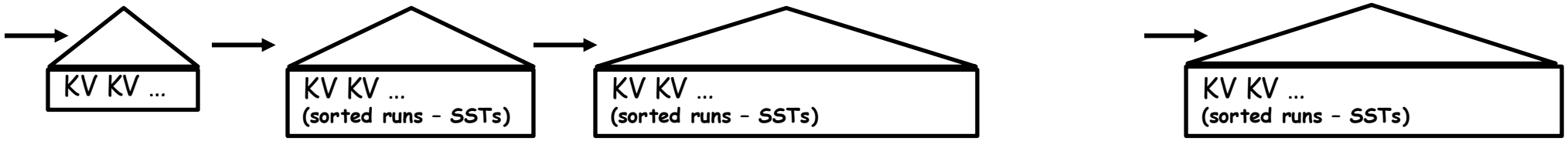
- No… (obviously?)
- But an index (or more than one) is an important component
  - All storage systems require an index (B+ Tree and many other alternatives and variants)
  - We have to translate application requests to device I/O operations, e.g (file,offset) → physical block
    - Mapping a large logical namespace to a (large?) physical address/location
  - Alternative: access physical blocks directly; not convenient; no sharing/protection/etc.
- A lookup can be a table
  - E.g. virtual to physical address translation (multi-level tables)
- Why isn't a table enough for storage?
  - 1. Ranges for input/output namespaces are too big and stored items vary a lot
  - 2. The full index is not expected to fit in memory in most cases
  - Tree-type structures are more appropriate for such lookups (although hash tables are used in cases)
  - B+-Trees generally dominate (block addressable devices)
- Today, the role of indexing has shifted to the Key-Value Store
  - FS has an index (file, offset → block)
  - Not used in the common path today under a KVS
  - FS merely used as a persistent allocator for large extents (files)
- Other forms of translation in the I/O path as well
  - E.g. network I/O, internal to storage device (FTL), caches
  - Interactions with the storage index not always clear nor well understood

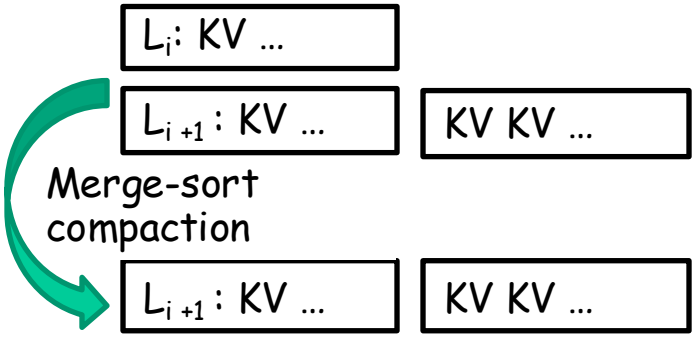
# Key-Value (KV) Stores

- "New" approach for organizing, accessing data
  - Has dominated the datacenter / cloud / storage
- Why? (Sep 2022):
- 1. Higher-level abstraction than files/offsets/blocks
  - Dictionary operations: put(), get(), delete(), and scan()
  - (a) More convenient for apps
  - (b) More room for optimization at the system level
- 2. Best known performance for small writes/updates
  - Traditionally, persistent I/O layers use a B+-tree
  - Optimal for read I/Os, very good for scans
  - Poor for writes: one random I/O per insert
  - Really poor for bursts of small updates: read-modify-write
  - E.g. with HDDs 100x degradation in throughput
- 3. Insensitive to workload mix
  - Cloud workloads demand this due to consolidation/blender effect
- 4. Best space utilization due to packing of data (no half full leaves)
  - 30% less space on device, possibly saving in memory as well
- Many other differences but these are most important

# (LSM) KV-Stores: Constant Data Re-organization

- LSM organizes data in N levels (L0 in memory) – sizes grow by f (e.g. f=10)
  - Data (keys) in each level are unique and sorted (for reads/scans)

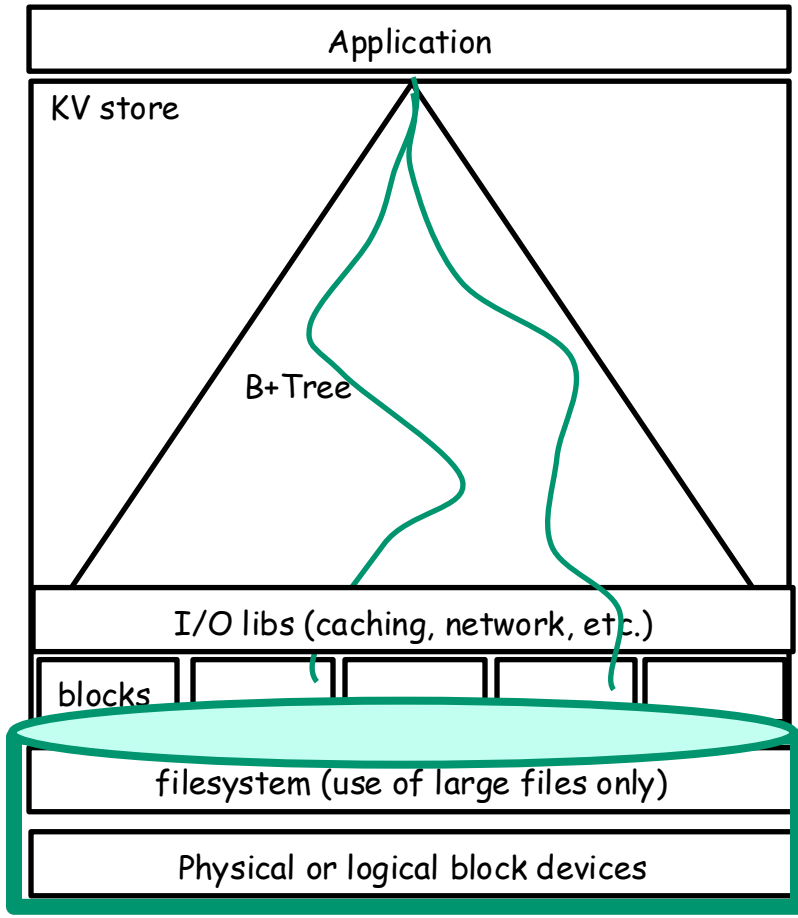



- Reorganizing data from level to level
  - Compaction process
  - Merge-sort operations: read – sort – write
  - + Generates only large I/Os, reclaims garbage space
  - - Increases I/O traffic over application I/O requests
    - I/O amplification: each data item read/written several times
  - Incurs CPU overhead, memory usage
    - Data traversals – poor match for memory/cpu trends

# Storage Stack Today

- Device (+ firmware)
  - Many peculiarities of NAND FLASH (and Non-Volatile Memory -- NVM)
- OS (block + FS)
  - FS (+Block layer) are there merely for space allocation = glorified space allocator on device
- I/O libs (cache, network)
  - A lot of activity (then and now): direct device access, network, maybe caching
- Key-value store
  - Data organization on device
  - Data access (insert/read/update)
  - Recovery
  - ++: transactions, multi-level indexes, scans, etc.
- Application
  - What runs over KV stores?
  - Typically, some data access framework to perform distributed/elastic queries and convert them to I/O operations to the KV store
  - Then an application that issues queries



# What is our Target?

- A “perfect” storage system
  - Go figure…
- What could it be?
- I suppose match device capabilities without much loss
  - Assume a very large amount of data (big data…)
  - Device capacity/density keeps growing
  - Fast storage devices available and getting faster
    - Internal NVMe/SSD device parallelism + PCI throughput driven by GPUs
    - Latency?
  - Performance
    - 1 random I/O per read operation (at smallest possible granularity)
      - 1 random I/O per scan + additional sequential I/Os for scan size
      - Unclear role for prefetching
    - 1 sequential I/O operation for many write requests
    - No additional read/write traffic (amplification)
      - Read/scan performance = device IOPS
      - Write performance = device throughput
    - Average latency: close to device (a few µs)
    - Tail latency: 10s of µs?
  - Availability, elasticity, load balancing
- This course
  - How close are we to this? Why? Can we do better?

# Storage vs. Memory

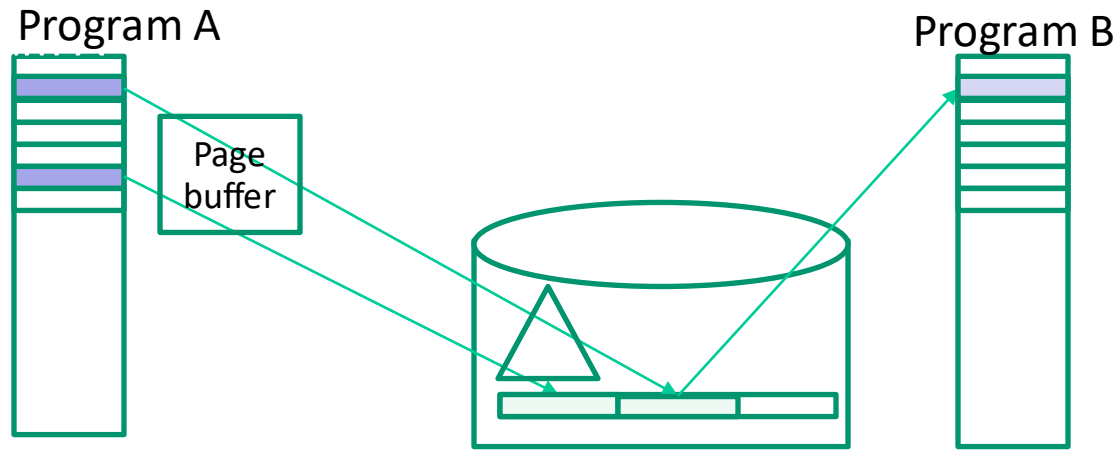
- Storage is expensive: KV stores are still expensive today; thousands of cycles per operations (get, put) on average
- Conceptually, storage is a simple function: move an item between (volatile) memory and (non-volatile) device
- Memory is (relatively) cheap: Why the difference?

serialization/des
Role of ←
application ←

- **select** what to persist
- small items, scattered
- transform to external layout
- abstraction (file, KVs, …)

Role of storage ←
system ←

- track order of requests
- write atomically (failures)
- I/O cache
- index / map keys to storage
- organize/reclaim space




- Key to storage is the notion of **persisting** data
  - Memory is heap → lifetime of a single process
  - Storage is "file/object/block" → beyond a single process lifetime
  - People looked at devices for "persistent" + "memory" (NVM)
  - Very interesting but did not pan out (yet); Why?
- Can/should Storage = Memory?
  - Why do applications need to see two different abstractions?
  - Is there a fundamental reason or is this a result of technology limits?

# Scope & Syllabus (roughly per week)

- Workloads
- Device technology
- LSM KV stores
- Indexing
- Fault tolerance & recovery
- Data organization on the devices
- Write-optimized structures for I/O
- I/O path from memory to device
- Memory-mapped I/O
- Persistent memory
- Network access, RDMA
- Evolving memory hierarchy

# Reading

- Review
  - Course web page
- Read
  - Slides, Your class notes
  - Assignment 1
  - Start working on it, due +2 weeks
- Next class
  - Cloud workloads