15-719/18-709: Advanced Cloud Computing

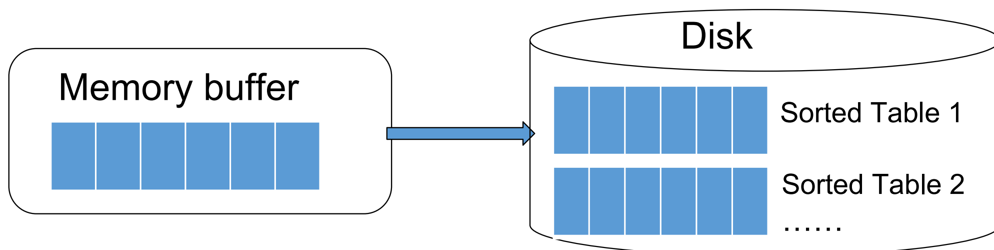
Greg Ganger
George Amvrosiadis
Majd Sakr

Agenda: Cloud-scale storage

- Scalable storage: essential for scalable cloud systems
- Approach 1: extend familiar distributed file systems (Covered Wednesday)
  - Basic design tradeoffs: statelessness, caching, etc.
  - NASD: scaling the data transfer path
  - Haystack: optimize for specific workload
  - GFS: fault-tolerance, targeted consistency model
  - TableFS: efficiency for small files too
- Approach 2: abandon traditional file system model (Covered today)
  - Examples: AWS S3, AWS EBS, Docker

Log Structured Merge (LSM) Trees

- Insert / Updates
  - Buffer and sort recent inserts/updates in memory
  - Write-out sorted buffers into local file system sequentially
  - Less random disk writes than traditional B-Tree
- Lookup / Scan
  - Search sorted tables one by one from the disk
    - Compaction is merge sort into new files, deleting old (cleaning)
  - Bloom-filter and in-memory index to reduce lookups

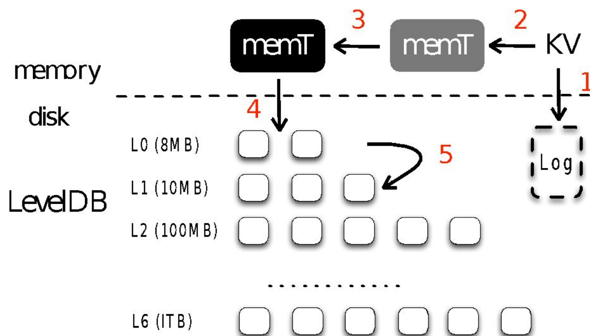


Write Optimized like LFS (cleaning = compaction)

LSM-trees: Insertion

1. Write sequentially 2. Sort data for quick lookups
3. Sorting and garbage collection are coupled

Clean so there is no overlap in SSTables in each level after 0



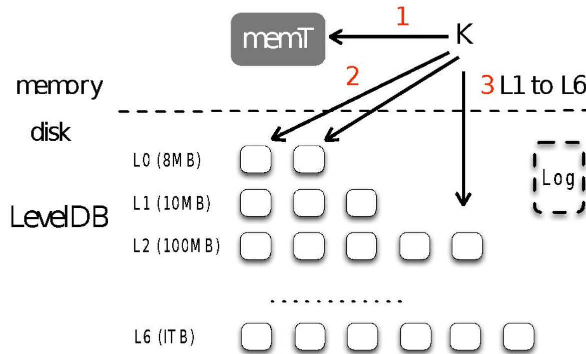
- (Cacheable) index per SSTable
- Lists 1st & last key per SSTable

[Lanyue Lu, FAST16]

O(log size) lookup like B-tree

LSM-trees: Lookup

1. Random reads
2. Travel many levels for a large LSM-tree

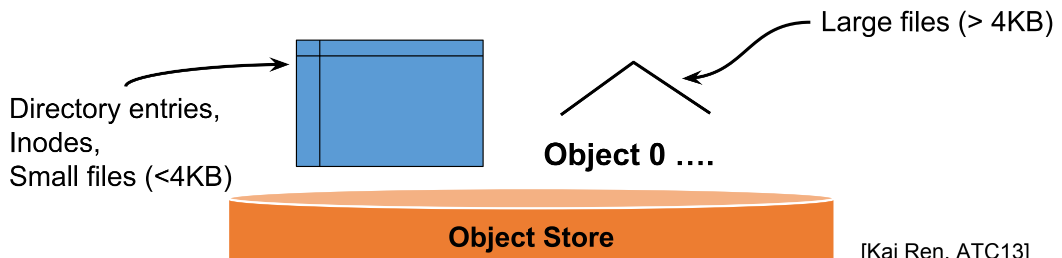


- (Cacheable) Bloom filter per SSTable
- Skip ~99% unneeded lookups

[Lanyue Lu, FAST16]

TableFS: metadata in LSM Trees

- Small objects embedded in LSM tree (tabular structure)
  - E.g. directory entries, inodes, small files
  - Turn many small files into one large object (~ 2MB)
- Larger files stored in object store indexed by TableFS-assigned IDs

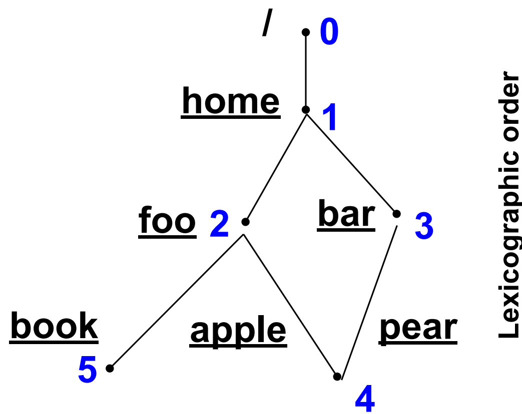


[Kai Ren, ATC13]

Table Schema

- Key: <Parent inode number, hash(filename)>
  - Inodes with multiple hard links: <inode number, null>
- Value: filename, inode attrs, inlined file data (or symlink to large object)

[Kai Ren, ATC13]

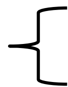


Key	Value
<0,hash(*home*)>	1, “home”, struct *stat*
<1,hash(*foo*)>	2, “foo”, struct *stat*
<1,hash(*bar*)>	3, “bar”, struct *stat*
<2,hash(*apple*)>	4, “apple”, hard link
<2,hash(*book*)>	5, “book”, struct *stat,* inline small file
<3,hash(*pear*)>	4, “pear”, hard link
<4,null>	4, struct *stat*, large file pointer

Table Schema (cont)

- Advantages:
  - Fewer random lookups by co-locating dir entries with inode attrs, small files
  - “readdir” performs sequential scan on the table

Entries in the same directory



Key	Value
<0,hash(*home*)>	1, “home”, struct *stat*
<1,hash(*foo*)>	2, “foo”, struct *stat*
<1,hash(*bar*)>	3, “bar”, struct *stat*
<2,hash(*apple*)>	4, “apple”, hard link
<2,hash(*book*)>	5, “book”, struct *stat,* inline small file
<3,hash(*pear*)>	4, “pear”, hard link
<4,null>	4, struct *stat*, large file pointer

[Kai Ren, ATC13]

Popular cloud storage options

- 1. Provide a ”traditional” filesystem
  - The OS running in each VM mounts file service
    - just like any client would in client-server distributed FS
  - E.g., NFS, AFS, **Google file system**, HDFS
  - Discussed on Monday
- 2. Provide block stores (virtual disks)
- 3. Provide a “union” filesystem on each client
- 4. Provide an “object store”

2. Provide block stores (virtual disks)

- A common option in VM-based environments
  - Guest OS running in a VM has code for FSs on disks
    - assumes that it has private access to disk capacity
  - So, give it a “disk” to use
    - but, giving it a physical disk isn’t VM/cloud style
- Virtual disk looks to guest OS just like real disk
  - Same interface
    - read/write of fixed-size blocks, ID’d by block number
  - Guest OS can format it, implement an FS atop it, etc.
    - VMM makes guest OS disk operations access the right content
- Most cloud infrastructures have this option
  - E.g., AWS Elastic Block Store (EBS), OpenStack Cinder

Virtual Disk (VD) implementation

- Client OSs think that they are using a real disk
  - So, they use disk-like block interfaces
    - e.g., SCSI rather than NFS
  - Guest OS may or may not know virtual disk is local
    - Non-local interface: network-disk interface (iSCSI)
    - Local interface: VMM translates to other protocol as needed
- VDs often implemented as files
  - A file is a sequence of bytes
    - So, a file can hold a sequence of fixed-sized blocks
  - So, a file server can be used for VDs
    - E.g., each VD is a file
    - May be accessed by block protocol or file protocol
      - E.g., via non-local or local from above,

More Virtual Disk (VD) stuff

- Thin provisioning
  - Promise more space that you have
    - E.g., tell 20 VMs they each get 1TB, but only have 10TB
  - Allocate physical space only for blocks that get written
    - Most devices are not used to full capacity
    - Benefits from TRIM and other storage class stuff ☺
- Performance interference
  - Each VM may have a virtual disk
    - OS in VM assumes it will behave like a real disk
      - Including performance behavior!
  - We expect time-sharing to have fairness / QoS
    - Need it for storage too
    - But, it's very difficult
      - Interference in caches, on-disk placement, metadata, ….

One aggressive demonstration of Quality of Storage

- **IOFlow: a Software-Defined Storage Architecture**.

  Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis,

  Antony Rowstron, Tom Talpey, Richard Black, Timothy Zhu.

  SOSP 2013, Farmington PA, Nov 2013.
  - SDN "forwarding rules" replaced with "request queue ordering"
  - Flows are abstraction of SLO, service binding, data & requests
    - Used for bandwidth allocation & sharing, content checking, prioritization for latency

3. Provide "union" filesystems

- A common option in container-based environments
  - Container runs atop OS
  - Container is given access to (part of) file system
    - usually thinks it has entire FS (via chroot)
  - Needs some "system-wide" and some "private" files
    - so, we want to give it both
- Make a single FS view from multiple FSs
  - Show contents of a directory as merge of several
    - With a sorted order when there are name conflicts
  - Implemented by a layer atop the individual FSs
    - Each operation accesses "unioned" FSs as appropriate

4. Provide "object" store

- A common option in large clouds
  - A simplified, generic "file" storage system
    - Like files, objects are sequences of bytes
    - Unlike FSs, usually just numerical object IDs
  - Example: AWS S3
    - Some (e.g., box or iCloud) provide simple directories too
- Usually limited interface and semantics
  - E.g., CRUD API: Create, Read (get), Update (put), Delete
    - No open/close, rename, links, locks, etc.
  - Often assumes single writer, sequential (or all-at-once)
    - No promises re: sharing/concurrency, interrupted writes, etc.

Next

- Cool Saurabh talk about analytics storage atop S3
- Next
  - Wednesday: tail latency
  - Next week: frameworks-2 and key-value stores

A Case for Packing and Indexing in Cloud File Systems

Saurabh Kadekodi
Bin Fan*, Adit Madan*, Garth Gibson
PARALLEL DATA LABORATORY
Carnegie Mellon University
, *Alluxio Inc.

Workload

- Spark job processing all data in memory and producing 3.2 million 8KB files
- Packing tiny files improves throughput and reduces cost
- By how much?

Improved Throughput Guess?

- 10x
- 25x
- 50x
- 100x
- more

61x more

Reduced Experiment Price?

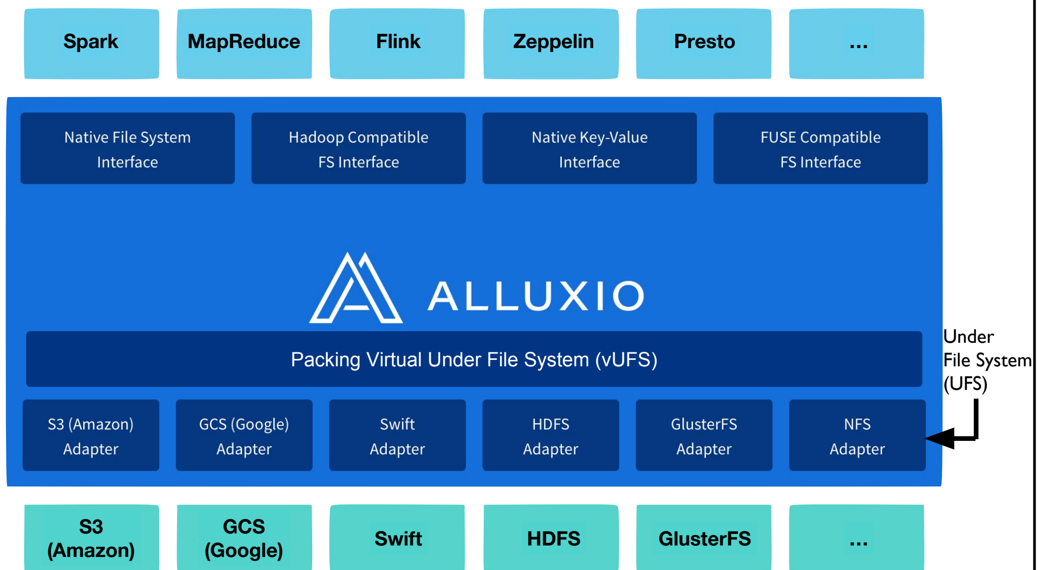
- 10x
- 25x
- 50x
- 100x
- more

25000x less

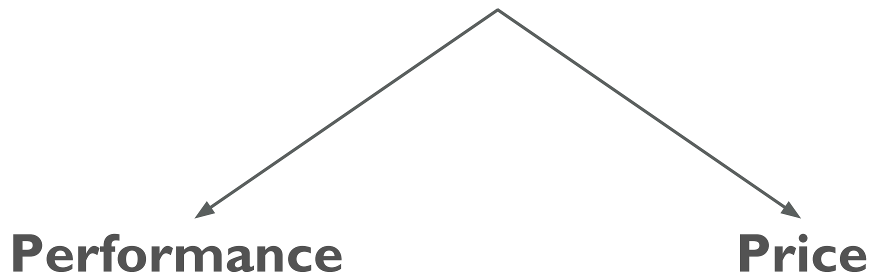
Problem Statement

- To augment a cloud file system's write-back cache with a packing and indexing layer that coalesces small files or segments of slow-growing files to transform arbitrary user workload(s) to a write pattern more ideal for cloud storage in terms of — *transfer sizes, number of objects* and *price*.
- tl;dr — Batch cloud writes and make large transfers.
- Invariant: Never write small files to backing cloud stores.

Alluxio



Motivation

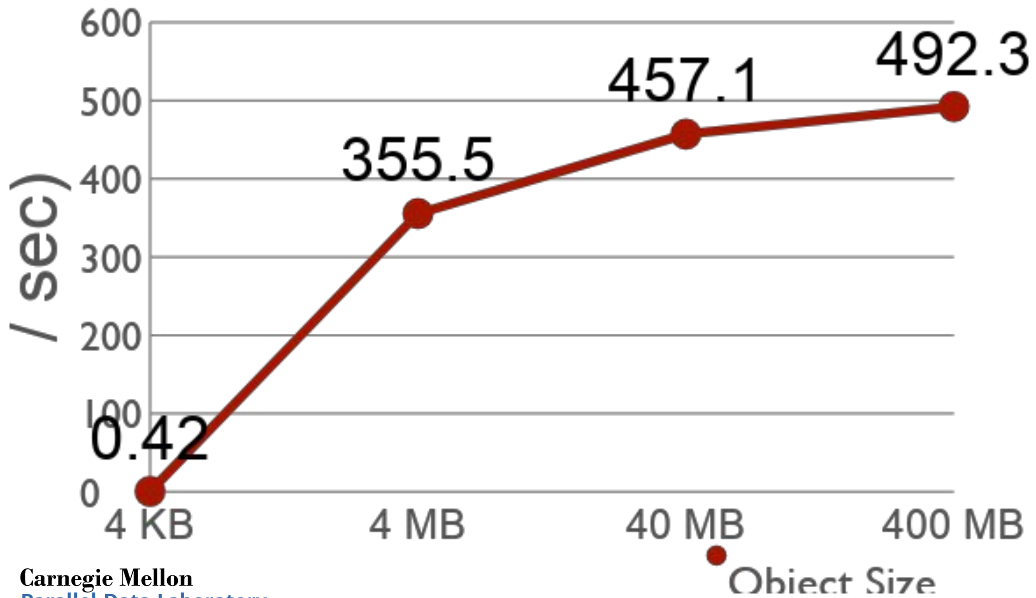


Performance **Price**

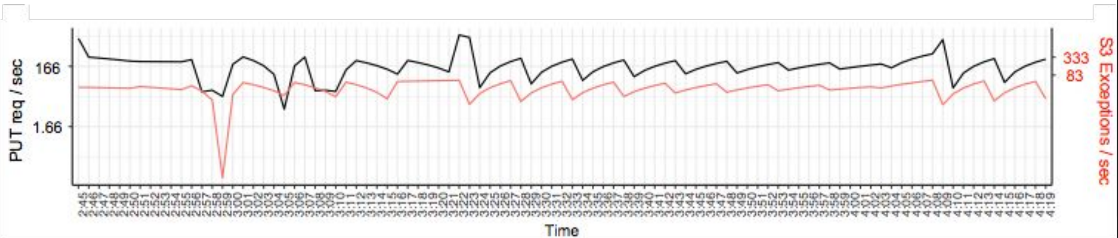
Performance Motivation

- r4.4xlarge EC2 instances
- 4 Alluxio workers
- 1 Alluxio masters
- ~13 GB data
- Increasing file sizes (4KB - 400MB)
  - 3.2M files of size 4KB (smallest)
  - 32 files of 400 MB (largest)

Throughput By Object Size

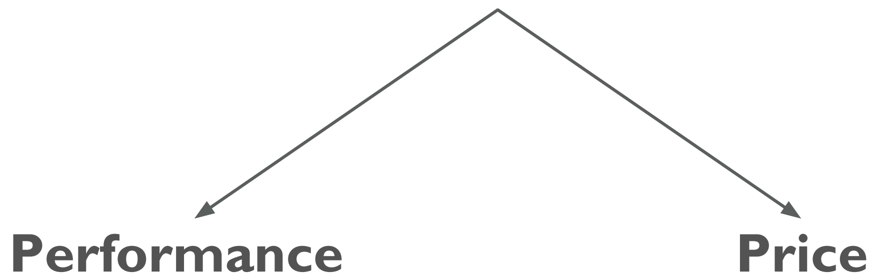


S3 Throttling in Action



- Request rate throttling by S3 for 4KB writes
- S3 warns routinely requesting >100 PUT req / sec subject to throttling
- NW bandwidth of r4.4xlarge instances ~ 10 Gigabit / sec => minimum 13MB writes to avoid being throttled => packing

Motivation



Performance **Price**

Price Motivation

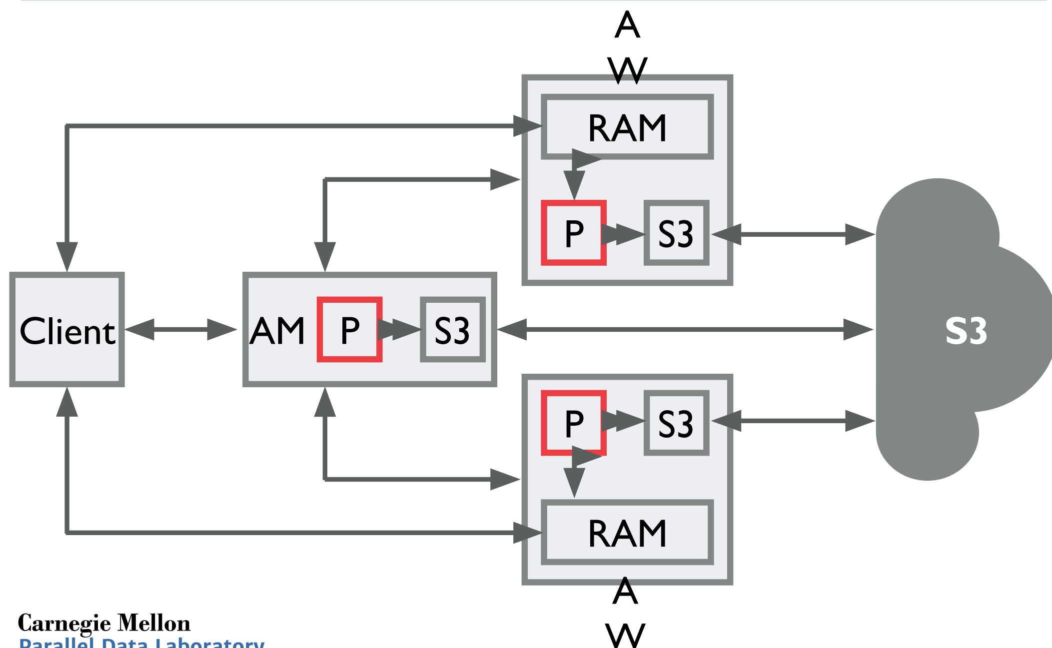
S3 Pricing Model	PUT, COPY, POST	GET	Data Retrieval Cost
Standard	$0.05 / 10000 req, same for retries	$0.04 / 100000 req, same for retries	Free (for certain data center locations)
Standard w/ Infrequent Access	$0.1 / 10000 req, no retries	$0.1 / 100000 req, no retries	$0.01 / GB

- For just one million files, the put cost = $5
- Packing can reduce cost by ***at least*** the packing factor

Design

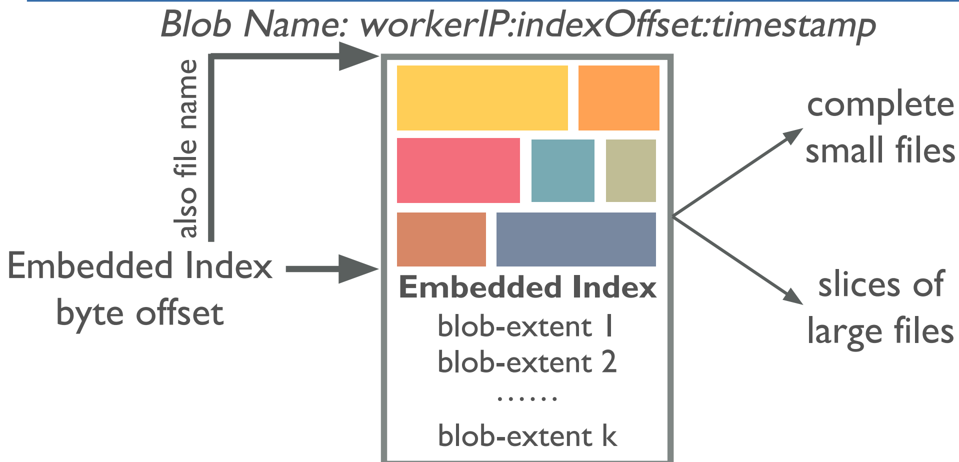
Carnegie Mellon
Parallel Data Laboratory

The Packing Modules



Carnegie Mellon
Parallel Data Laboratory

A Packed Blob



- Packing policy determines what to pack
- Triggered by dirty bytes & timeout

Blob Descriptor Table - BDT (Index)

- Maps Alluxio files → Current Location
- Implemented as LevelDB to bound memory usage
- Global BDT in centralized location
- Each worker has BDT as optimization

Evaluation

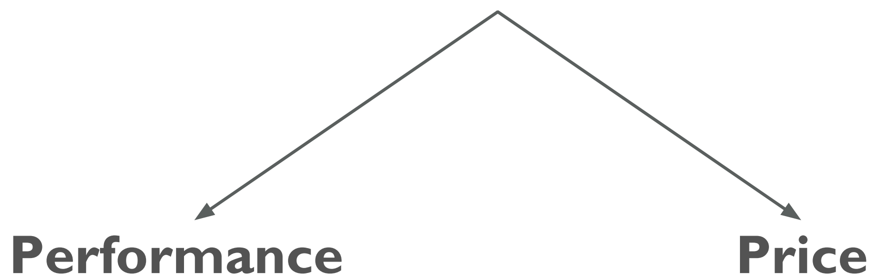
Configuration

- Experiment - Small file concurrent create (avg of 2 runs)
  - 1 AlluxioMaster (i.e. 1 PackingMaster)
  - 4 AlluxioWorkers (i.e. 4 PackingWorkers)
  - 32 concurrent clients (workload generators) — 8 per AW
  - 100K files (each 8KB) per client ~ totally 3.2M files
  - Total workload size: 24.4 GB

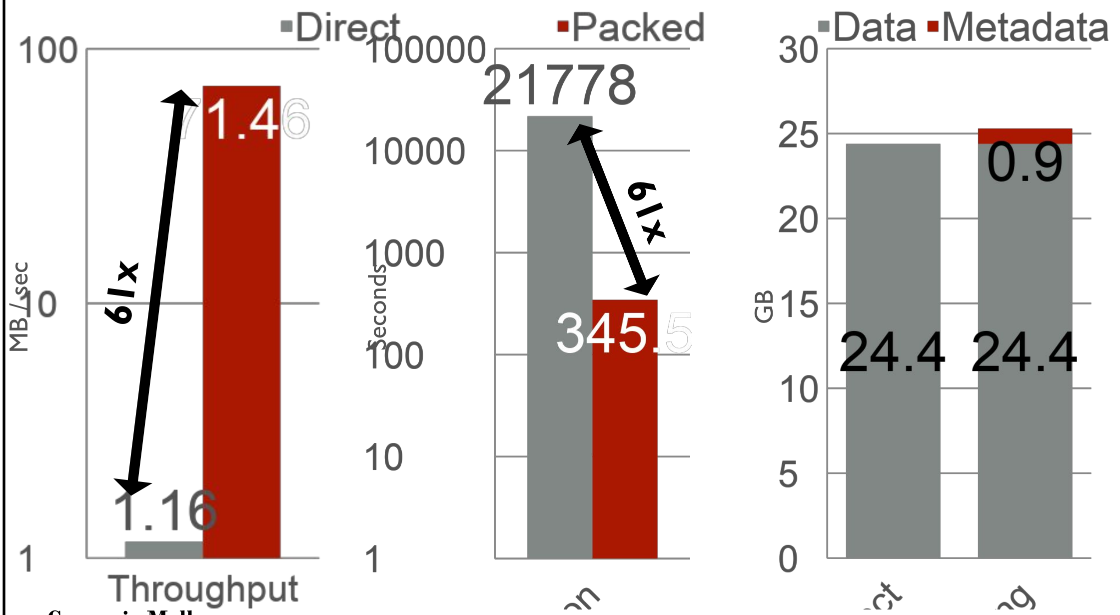
Packing Configuration

- Max blob size: 1 GB
- Packing interval: 5 sec
- # Packing threads: 16
- # Master threads: 16
- Backup interval: 1 min

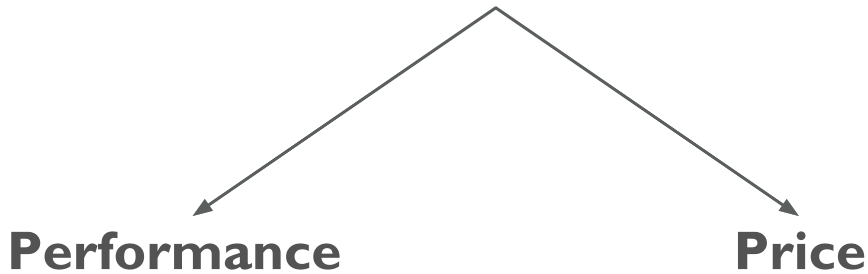
Motivation Revisited



Write Performance Comparison



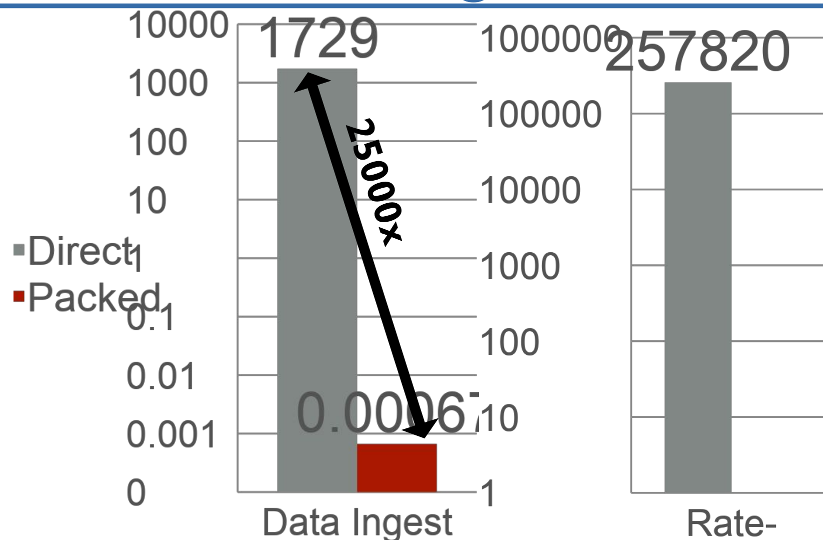
Motivation Revisited



Cloud Storage Price

S3 Pricing Model	PUT, COPY, POST	GET	Data Retrieval Cost
Standard	$0.05 / 10000 req, same for retries	$0.04 / 100000 req, same for retries	Free (for certain data center locations)
Standard w/ Infrequent Access	$0.1 / 10000 req, no retries	$0.1 / 100000 req, no retries	$0.01 / GB

S3 Data Ingest Price



- Request rate is throttled much more than data rate

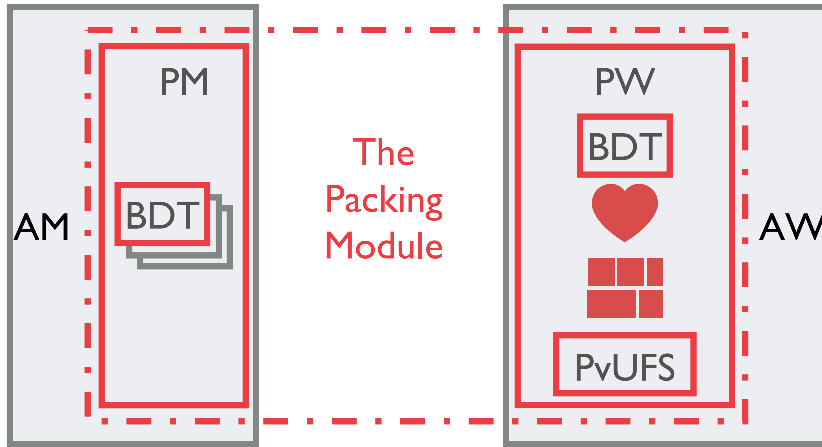
Conclusion

- S3 prefers large objects
- S3 rate limits ops / sec to their buckets
- Packing eliminates this problem by:
  - Reducing ops made to S3 by at least 1000x
  - Making much more infrequent accesses

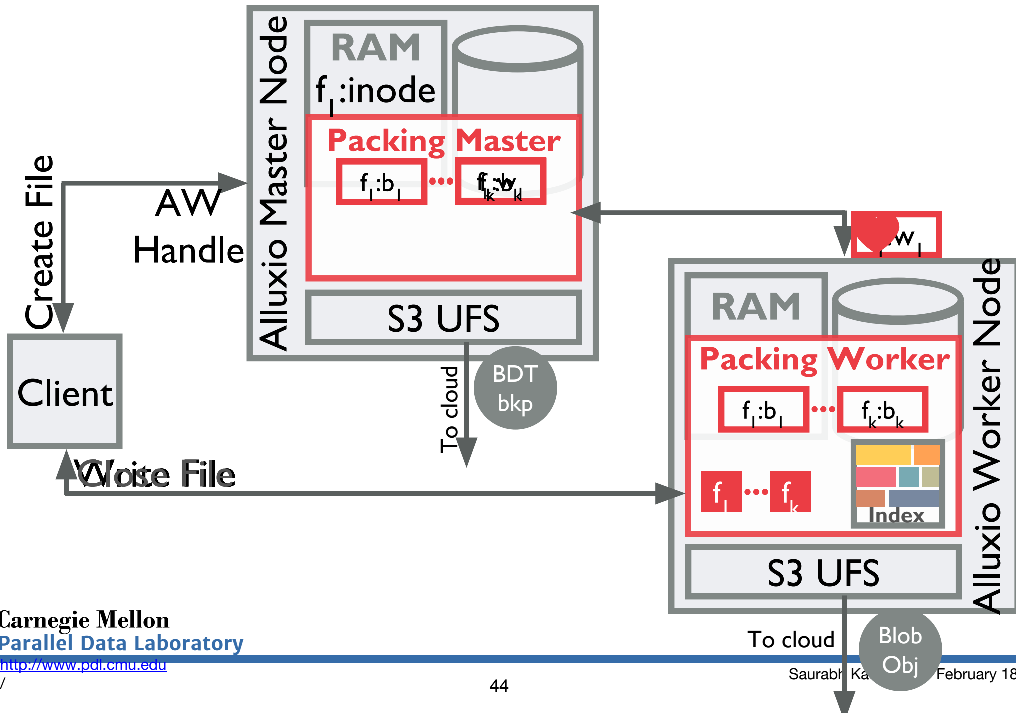
Questions?

Thank You!

The Packing Modules



Packing Write Flow



Packing Read Flow