TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA
Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

(Big) Data Storage Systems

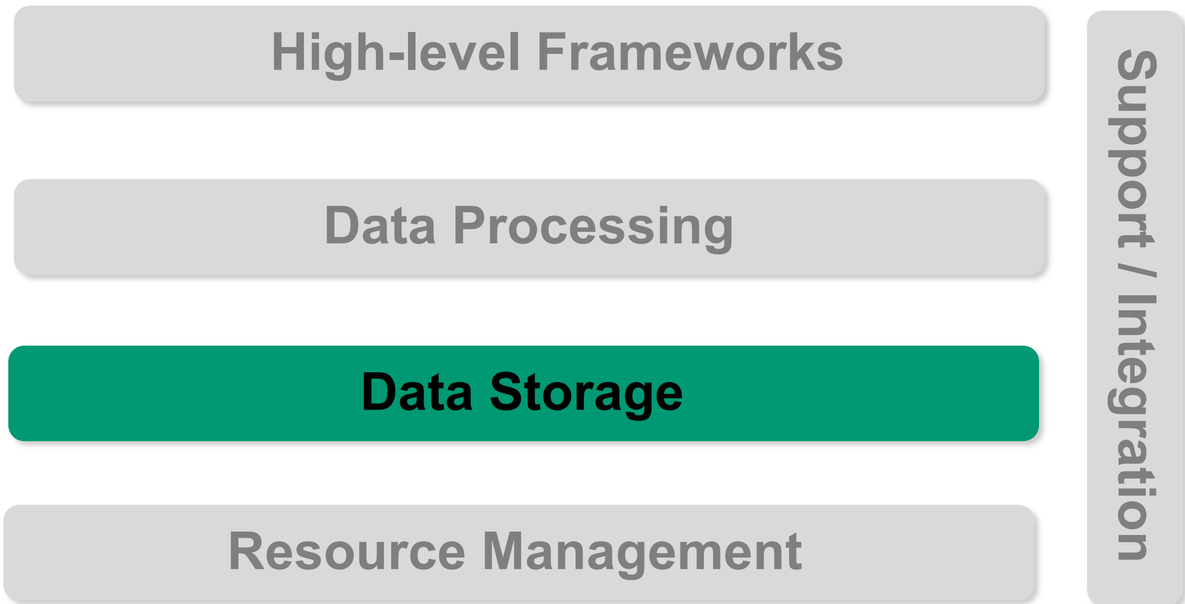
Corso di Sistemi e Architetture per Big Data

A.A. 2024/25

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

The reference Big Data stack

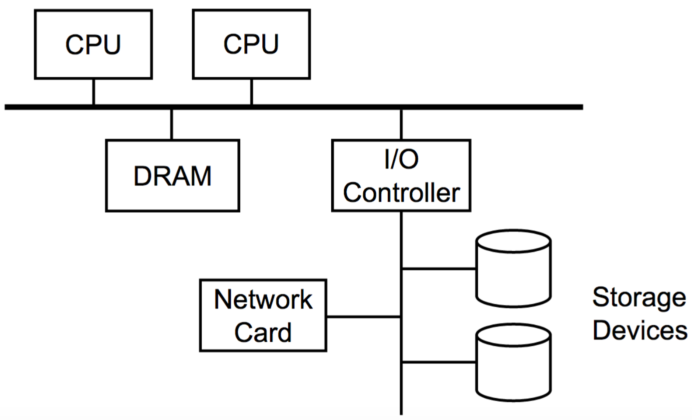


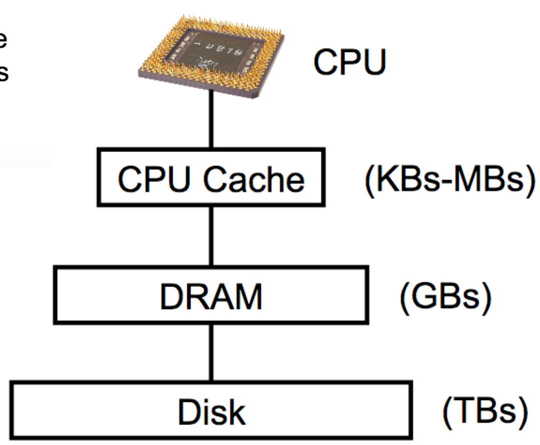
Where storage sits in Big Data stack

- Some frameworks and tools in a data lake architecture

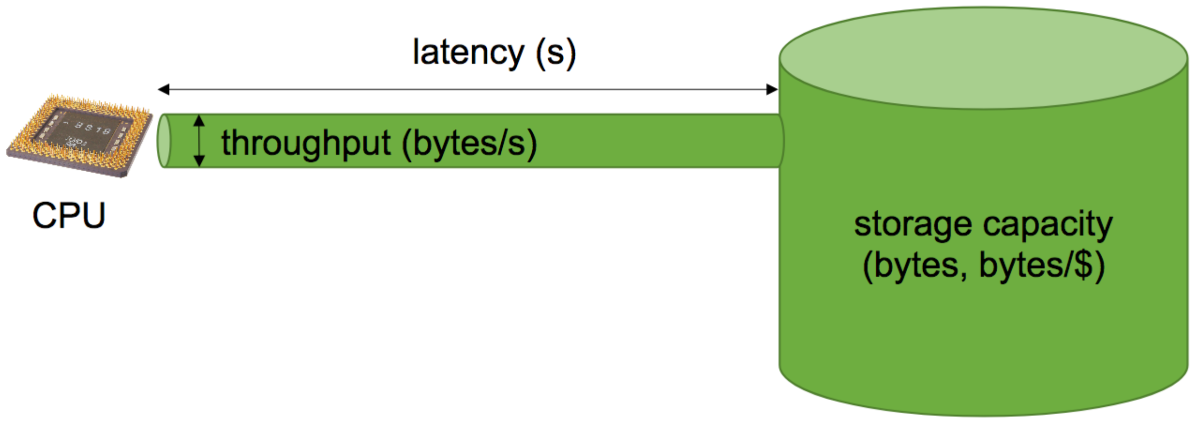
Hadoop MapReduce, Apache Spark, Flink, presto, TensorFlow	Processing engines
Parquet, protobuf (Protocol Buffers), {JSON} (JavaScript Object Notation), ICEBERG, AVRO	File formats & metadata
Apache Ozone™, ceph, hadoop HDFS, Amazon S3	Large-scale file systems or object stores

Typical server architecture and storage hierarchy



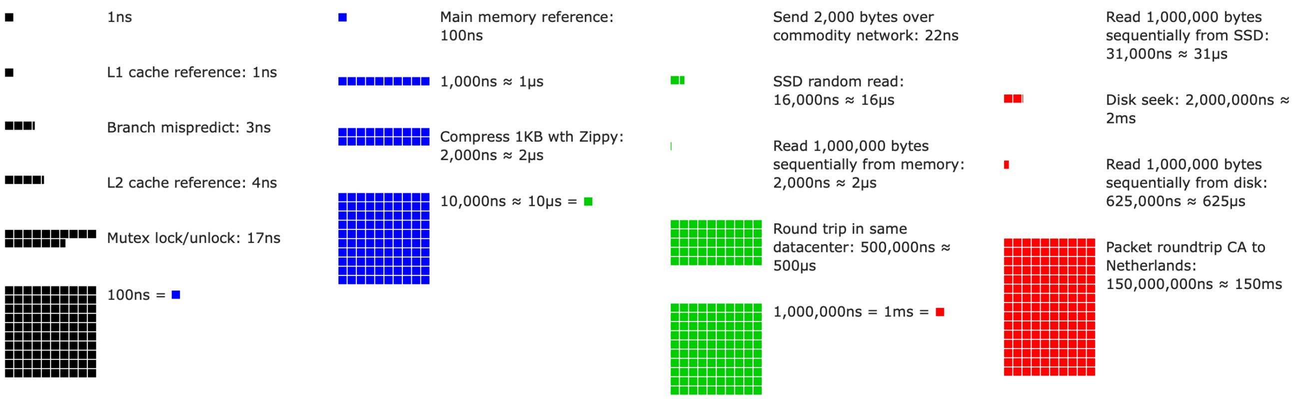


Storage performance metrics



Where to store data?

- See “Latency numbers every programmer should know” (presented by Jeff Dean from Google in 2010, numbers updated in 2020)



Maximum attainable throughput

- Varies significantly by device
  - 50 GB/s for RAM
  - 3 GB/s for NVMe SSD
    - SSD: Solid State Drive
    - NVMe: Non-Volatile Memory Express
    - NVMe is a storage access and transport protocol for flash and next-generation SSDs
  - 130 MB/s for hard disk

- Assumes large reads (≫1 block)

Hardware trends over time

- Capacity/$ grows at a fast rate (e.g., doubles every 2 years)

- Throughput grows at a slower rate (~5% per year), but new interconnects help

- Latency does not improve much over time

Data storage: the classic approach

- **File**
  - Group of data, whose structure is defined by file system
- File system
  - Controls how data are structured, named, organized, stored and retrieved from disk
  - Single (logical) disk (e.g., HDD/SDD, RAID)

- **Relational database**
  - Organized/structured collection of data (e.g., entities, tables)
- Relational database management system (RDBMS)
  - Provides a way to organize and access relational data
  - Enables data definition, update, retrieval, administration

What about Big Data?

Storage capacity and data transfer rate have increased massively over the years

HDD
Capacity: ~1TB
Throughput: 250MB/s

SSD
Capacity: ~1TB
Throughput: 850MB/s

Let's consider the latency (time needed to transfer data*)

Data Size	HDD	SSD
10 GB	40s	12s
100 GB	6m 49s	2m
1 TB	1h 9m 54s	20m 33s
10 TB	?	?

* we consider no overhead

We need to scale out!

General principles for scalable data storage

- Scalability and high performance
  - Need to face continuous growth of data to store
  - Use multiple nodes to store data
- Ability to run on commodity hardware
  - But hardware failures are the norm rather than the exception
- Reliability and fault tolerance
  - Transparent data replication
- Availability
  - Data should be available to serve requests when needed
  - CAP theorem: trade-off with consistency

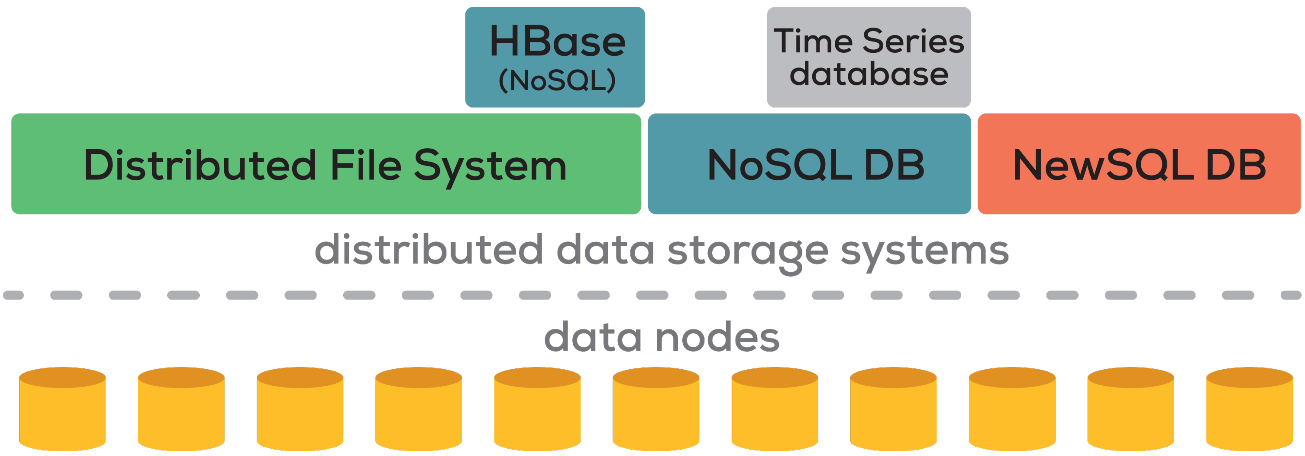
Scalable and resilient data storage solutions

Various forms of storage for Big Data:

- **Distributed file systems and object stores**
  - Manage **files** and **objects** on multiple nodes
  - E.g., Google File System, HDFS, Ozone, Ambri
- **NoSQL data stores**
  - Simple and flexible **non-relational** data models: key-value, column family, document, and graph
  - Horizontal scalability and fault tolerance
  - E.g., Redis, BigTable, Hbase, Cassandra, MongoDB, Neo4J
  - Also time series DBs built on top of NoSQL (e.g.,: InfluxDB, KairosDB)
- **NewSQL databases**
  - Add horizontal scalability and fault tolerance to **relational** model
  - E.g., VoltDB, Google Spanner, CockroachDB

Scalable and resilient data storage solutions

Whole picture of different storage solutions we consider



Cloud data storage

- Goals:
  - On-demand (elastic) and geographic scale
  - Fault tolerance
  - Durability (versioned copies)
  - Simplified application development and deployment
  - Support for cloud-native apps (serverless)
- Some public Cloud services for data storage
  - DFSs: Amazon EFS
  - Object stores: Amazon S3, Google Cloud Storage, Azure Storage
  - Relational DBs: Amazon RDS, Amazon Aurora, Google Cloud SQL, Azure SQL Database
  - NoSQL data stores: Amazon DynamoDB, Amazon DocumentDB, Google Cloud Bigtable, Google Datastore, Azure Cosmos DB, MongoDB Atlas
  - NewSQL databases: Google Cloud Spanner
  - Serverless databases: Google Firestore, CockroachDB

Distributed File Systems (DFS)

- Primary support for data management
- Manage data storage across a network of servers
  - Usually locally distributed, in some case geo-distributed
- Usual interface to store data as files and later access them for reads and writes
- Several solutions with different design choices
  - **GFS**, **HDFS** (GFS open-source clone): batch applications with large files
  - **Alluxio**: in-memory (high-throughput) storage system
  - Lustre https://www.lustre.org: open-source, large-scale distributed file system
  - Ceph https://docs.ceph.com/: open-source, unified system for object, block, and file storage

Case study: Google File System (GFS)

Assumptions and motivations

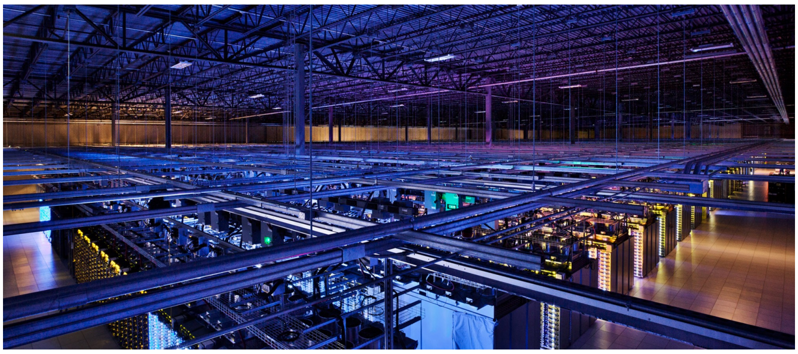
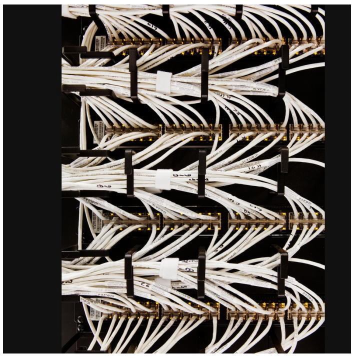
- System is built from inexpensive commodity hardware that often fails
  - 60,000 nodes, each with 1 failure per year: 7 failures per hour!
- System stores large files
- Large streaming/contiguous reads, small random reads
- Many large, sequential writes that append data
  - Concurrent clients can append to same file
- High sustained bandwidth is more important than low latency

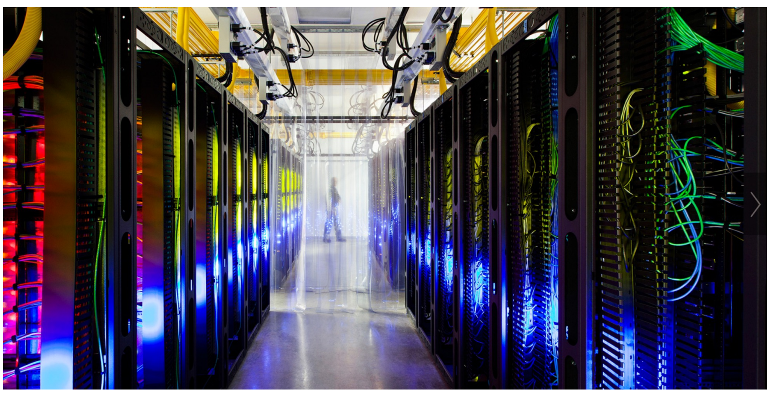
Ghemawat et al., The Google File System, *SOSP '03*
https://static.googleusercontent.com/media/research.google.com/it//archive/gfs-sosp2003.pdf

GFS: Main features

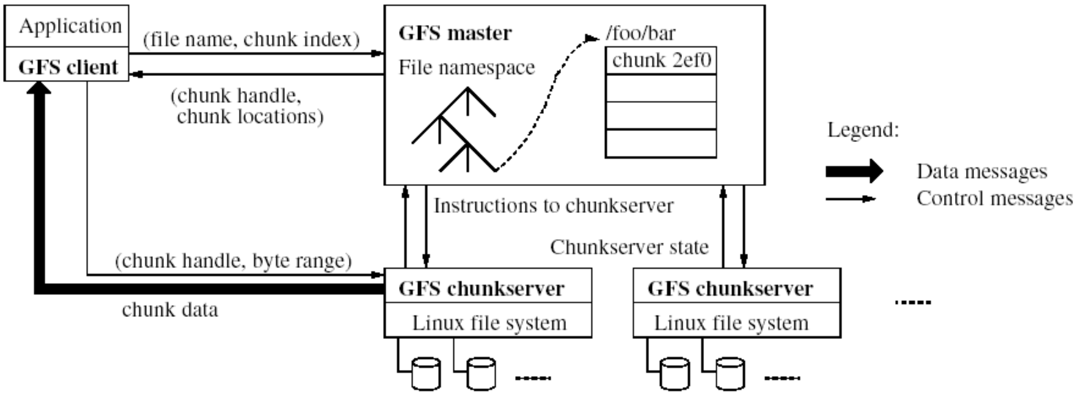
- Distributed file system implemented in user space
- Manages (very) large files: usually multi-GB
- **Data parallelism** using *divide et impera* approach: file split into fixed-size chunks
- ***Chunk***:
  - Fixed size (either 64MB or 128MB)
  - Transparent to users
  - Stored as plain file on chunk servers
- Write-once, read-many-times pattern
  - Efficient *append* operation: appends data at the end of file *atomically at least once* even in the presence of concurrent operations (minimal synchronization overhead)
- Fault tolerance and high availability through chunk replication, no data caching

GFS: Operation environment

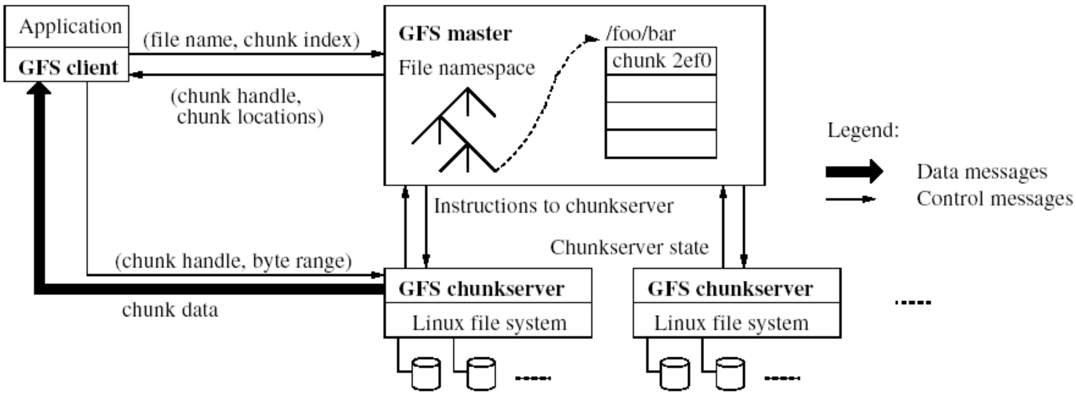


GFS: Architecture



- Master
  - Single, centralized entity (to simplify the design)
  - Manages file metadata (stored in memory)
    - Metadata: access control information, mapping from files to chunks, locations of chunks
  - Does not store data (i.e., chunks)
  - Manages operations on chunks: create, replicate, load balance, delete

GFS: Architecture



- Chunk servers (100s – 1000s)
  - Store chunks as files
  - Spread across cluster racks
- Clients
  - Issue *control* (metadata) requests to GFS master
  - Issue *data* requests to GFS chunkservers
  - Cache metadata, do not cache data (simplifies system design)

GFS: Metadata

- Master stores 3 major types of metadata:
  - File and chunk namespace (directory hierarchy)
  - Mapping from files to chunks
  - Current locations of chunks
- Metadata are stored in memory (64B per chunk)
  - ✓ Fast, easy and efficient to scan the entire state
  - ✗ Number of chunks is limited by amount of master's memory
    "The cost of adding extra memory to the master is a small price to pay for the simplicity, reliability, performance, and flexibility gained"
- Master also keeps an operation log where metadata changes are recorded
  - Log is persisted on master's disk and replicated for fault tolerance
  - Master can recover its state by replaying operation log
  - Checkpoints for fast recovery

GFS: Chunk size

- Chunk size is either 64 MB or 128 MB
  - Much larger than typical block sizes
- Why? Large chunk size reduces:
  - Number of interactions between client and master
  - Size of metadata stored on master
  - Network overhead (persistent TCP connection to chunk server)
- Each chunk is stored as a plain Linux file
- Cons
  - ✗ Wasted space due to internal fragmentation
  - ✗ "Small" files consist of a few chunks, which get lots of traffic from concurrent clients (can be mitigated by increasing replication factor)

GFS: Fault tolerance and replication

- Master controls and maintains the replication of each chunk on several chunk servers
  - At least 3 replicas on different chunk servers
  - Replication based on primary-backup schema
  - Replication degree > 3 for highly requested chunks
- Multi-level placement of replicas
  - Different machines, same rack + availability and reliability
  - Different machines, different racks + aggregated bandwidth
- Data integrity
  - Chunk divided in 64KB blocks; 32B checksum for each block
  - Checksum kept in memory
  - Checksum checked every time app reads data

GFS: Master operations

- Stores metadata
- Manages and locks namespace
  - Namespace represented as a lookup table
  - Read lock on internal nodes and read/write lock on leaves: read lock allows concurrent mutations in the same directory and prevents deletion, renaming or snapshot
- Communicates periodically with each chunk server using RPC
  - Sends instructions and collects chunk server state (*heartbeat* messages)
- Creates, re-replicates and rebalances chunks
  - Balances chunk servers' disk space utilization and load
  - Distributes replicas among racks to increase fault tolerance
  - Re-replicates a chunk as soon as the number of its available replicas falls below the replication degree

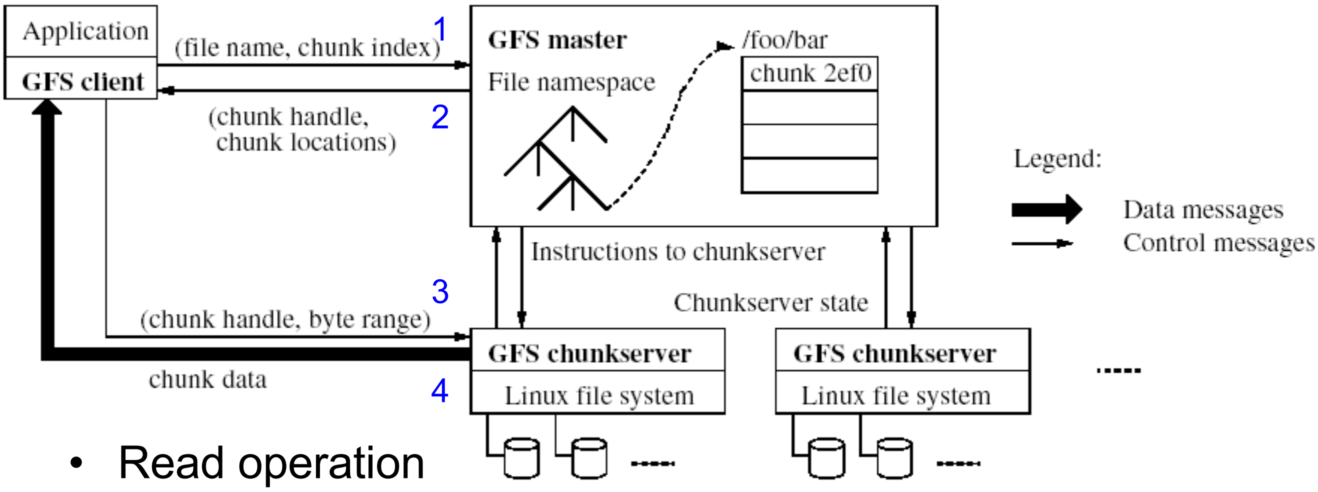
GFS: Master operations

- Garbage collection
  - File deletion logged by master
  - Deleted file is renamed to a hidden name with deletion timestamp, so that real deletion is postponed and file can be easily recovered in a limited timespan
- Stale replica detection
  - Chunk replicas may become stale if a chunk server fails or misses updates to chunk
  - For each chunk, the master keeps a chunk version number
  - Chunk version number updated at each chunk mutation
  - Master removes stale replicas during garbage collection

GFS: Interface

- Files are organized in directories
  - But no data structure to represent directory
- Files are identified by their pathname
  - Bu no alias support
- GFS supports traditional file system operations (but not Posix-compliant)
  - **`create`**, **`delete`**, **`open`**, **`close`**, **`read`**, and **`write`**
- Supports also 2 special operations:
  - **`snapshot`**: makes a copy of file or directory tree at low cost (based on copy-on-write techniques)
  - **`record append`**: allows multiple clients to append data to the same file concurrently, without overwriting one another's data

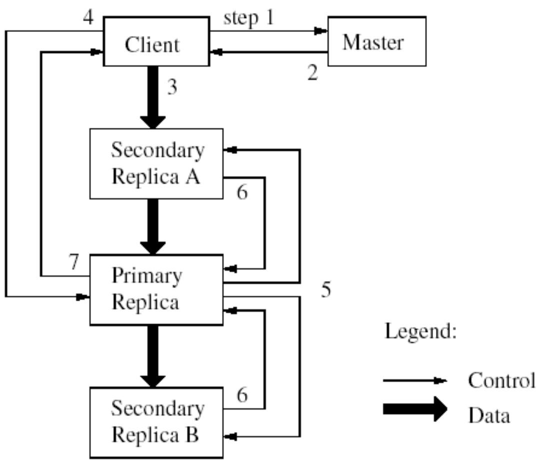
GFS: Read operation



- Read operation
  - Data flow is decoupled from control flow
  1) Client sends read(file name, chunk index) to master
  2) Master replies with *chunk handle* (globally unique ID of chunk), chunk version number (to detect stale replica), and chunk locations
  3) Client sends read(chunk handle, byte range) to the closest chunk server among those serving the chunk
  4) Chunk server replies with chunk data

GFS: Mutation operation

- Mutations are `write` or append
  - Performed at all chunk's replicas in same order
- Based on *lease* mechanism
  - Goal: minimize management overhead at master
  - Master grants chunk lease to primary replica
  - Client sends command to primary (4)
  - Primary picks serial order for all mutations to chunk and secondaries follow order when applying mutations
  - Secondaries reply to primary, then primary replies to client (7)
  - Lease is renewed using periodic heartbeat messages between master and chunk servers



- Data flow is decoupled from control flow
- Client sends data to *any* of the chunk servers identified by master, which in turn pushes data to other replicas in a chained fashion so to fully utilize network bandwidth

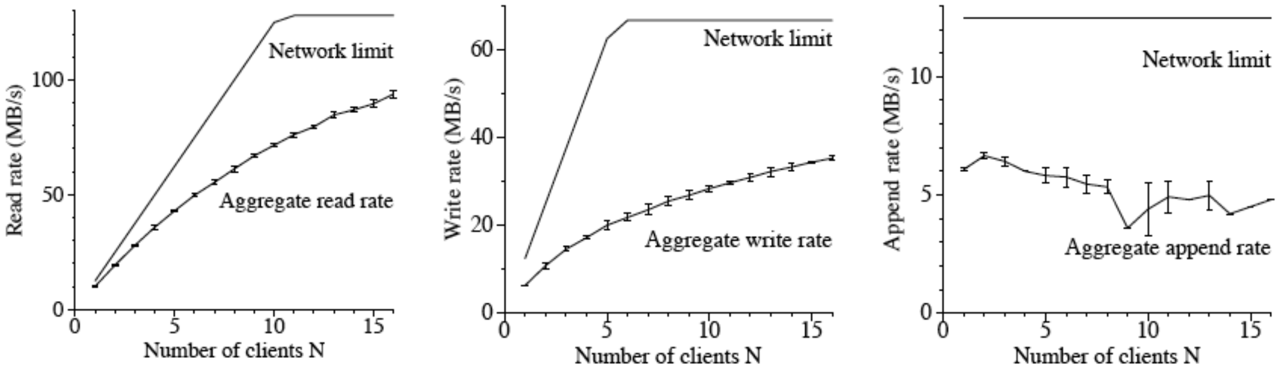
GFS: Atomic append

- GFS provides an atomic append operation
- Client sends only data (without specifying offset)
- GFS appends data to file *at-least-once* atomically (i.e., as one continuous sequence of bytes)
  - At offset chosen by GFS
  - Works with multiple concurrent writers
  - At least once: applications must cope with possible duplicates
- Append operations were heavily used by Google’s distributed apps
  - E.g., files often serve as multiple-producers/single-consumer queue or contain results merged from many clients (MapReduce)

GFS: Consistency model

- Changes to namespace (e.g., file creation) are atomic
  - Managed by GFS master with locking
- Mutations are ordered as chosen by primary replica, but chunk server failures can cause inconsistency
- GFS has a “relaxed” consistency model: eventual consistency
  - Simple and efficient to implement

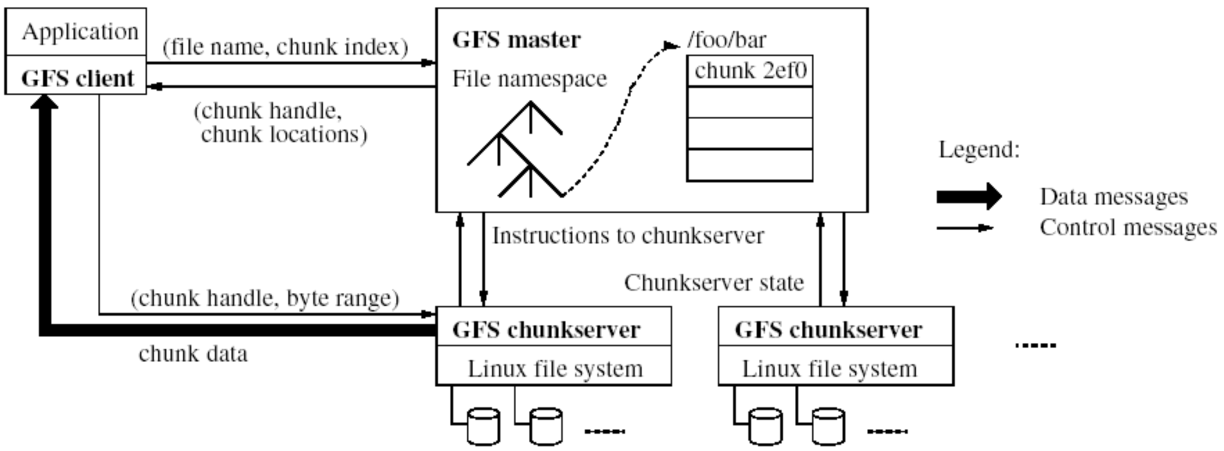
GFS performance (in 2003)



(a) Reads (b) Writes (c) Record appends

- Read performance is satisfactory (80-100 MB/s)
- But reduced write performance (30 MB/s) and relatively slow (5 MB/s) in appending data to existing files

GFS problems



Main architectural problem is…

Single master — Single point of failure (SPOF)
Scalability bottleneck

GFS problems: Single master

- Solutions adopted to overcome issues related to single master
  – Overcome SPOF: by having multiple “shadow” masters that provide read-only access when primary master is down
  – Overcome scalability bottleneck: by reducing interaction between master and clients
    - Master stores only metadata
    - Clients can cache metadata
    - Chunk size is large
    - Chunk lease: master delegates authority to primary replica
- Overall, simple solutions

GFS summary

- GFS success
  – Used by Google to support search service and other services
  – Availability on commodity hardware
  – High throughput by decoupling control and data
  – Supports massive data sets and concurrent appends
- GFS problems (besides single master)
  – Metadata stored in master memory
    - “Limited” scalability: approximately 50M files, 10PB
  – Semantics not transparent to apps
  – Slow failover
  – Client’s delay when recovering from failed chunk server
  – Not good for all services: focus on throughput, no guarantee on latency

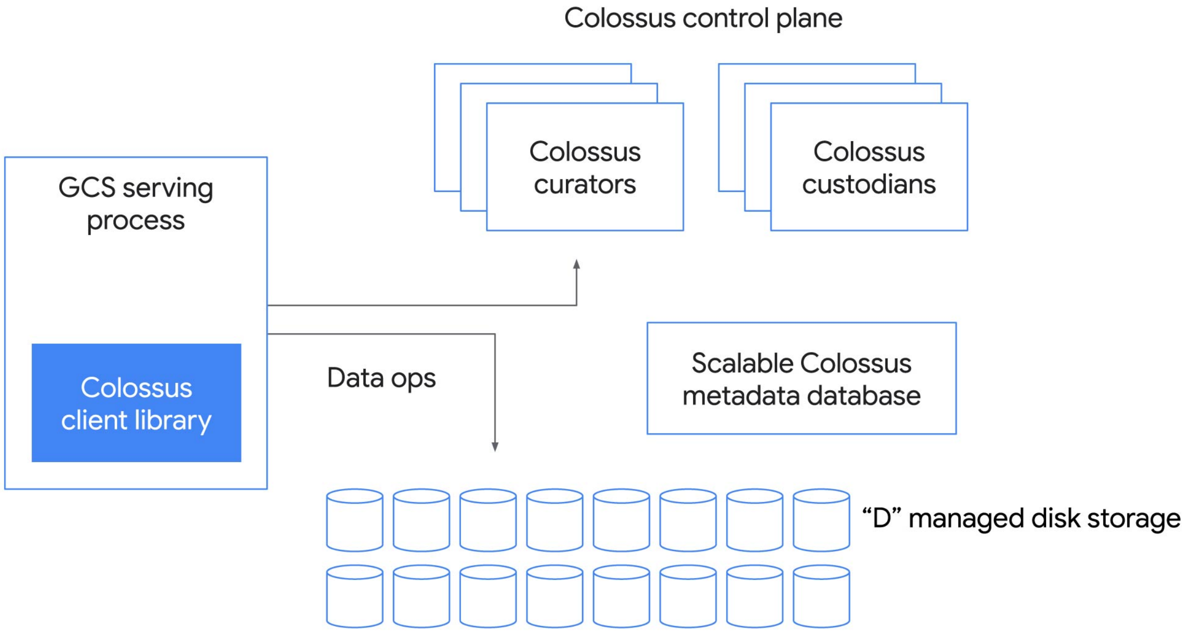
Google Colossus

- Successor to GFS (since 2010)
- Designed for a wide range of apps (YouTube, Maps, Photos, search ads)
- At Google scale: EB of storage, 10K servers
- Distributed masters, chunk servers replaced by D servers
- Scalable metadata layer, built on top of Bigtable
- Error-correcting codes (e.g., Reed-Solomon)
- Client-driven encoding and replication
- Hardware diversity: mix of flash memory and disks
- Google Cloud services built on top
  – Cloud Storage (object store), Cloud Firestore (NoSQL data store)

https://cloud.google.com/blog/products/storage-data-transfer/a-peek-behind-colossus-googles-file-system

https://www.youtube.com/watch?v=q4WC_6SzBz4

Colossus: key components



Hadoop Distributed File System (HDFS)

- Open-source user-level DFS https://hadoop.apache.org
- GFS clone: shares many features with GFS (including pros and cons)
  - Master/worker architecture
  - Large files, data parallelism
  - Commodity hardware
  - Fault-tolerant and throughput-oriented
- Integrated with processing frameworks and ingestion tools, e.g., Hadoop MapReduce, Spark, Flink, NiFi

https://www.databricks.com/glossary/hadoop-distributed-file-system-hdfs

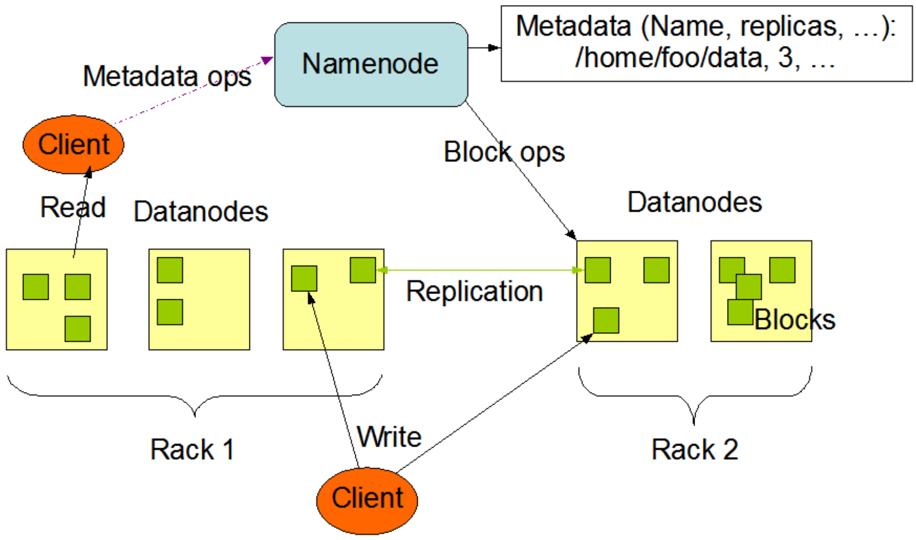
Shafer et al., The Hadoop Distributed Filesystem: Balancing Portability and Performance, *ISPASS 2010*
https://www.jeffshafer.com/publications/papers/shafer_ispass10.pdf

HDFS: Design principles

- Designed to handle large datasets: typical file size is GBs or TBs
- Write-once, read-many-times access pattern to files
  - E.g., MapReduce apps, web crawlers
- Commodity, low-cost hardware
  - Designed to work without noticeable interruption even when failures occur
- Portability across heterogeneous hardware and software platforms

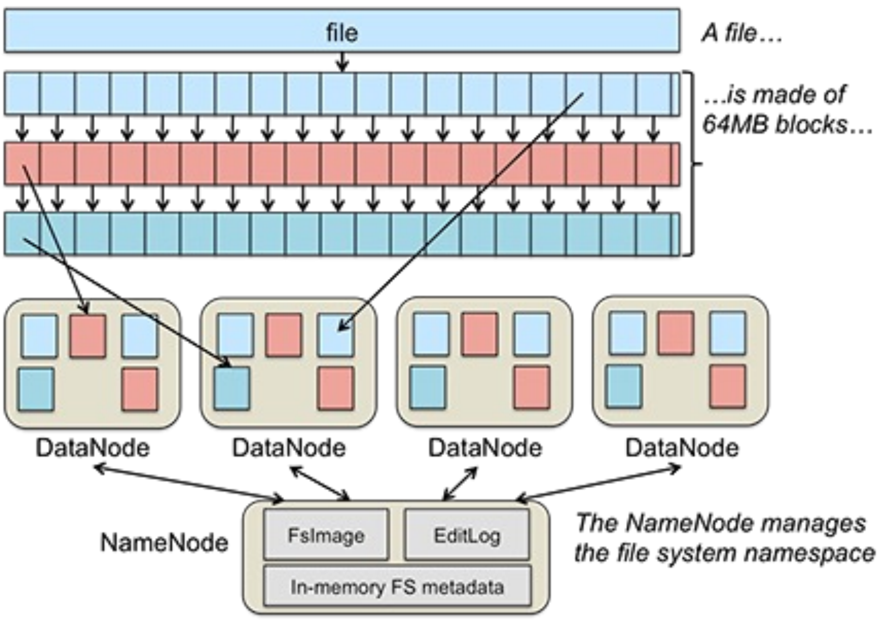
HDFS: Architecture

- Master/workers, nodes in HDFS cluster:
  - One *NameNode* (GFS master)
  - Multiple *DataNodes* (GFS chunk servers)



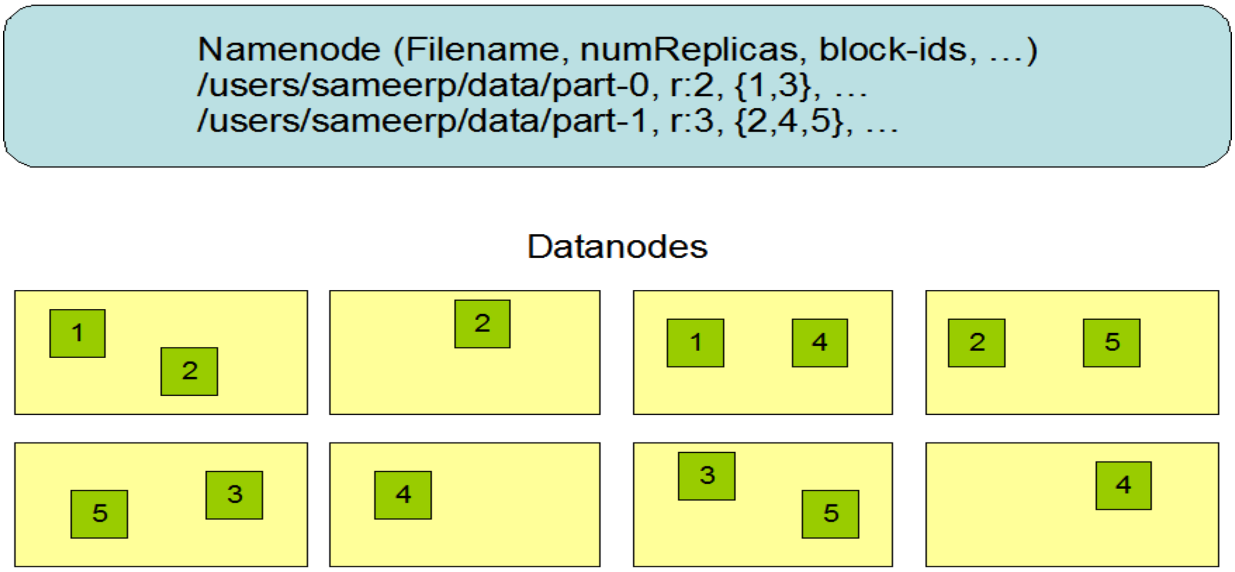
HDFS: File management

- Data parallelism: file split into blocks (GFS chunks) which are stored on DataNodes
- Large size blocks (default 64 MB)

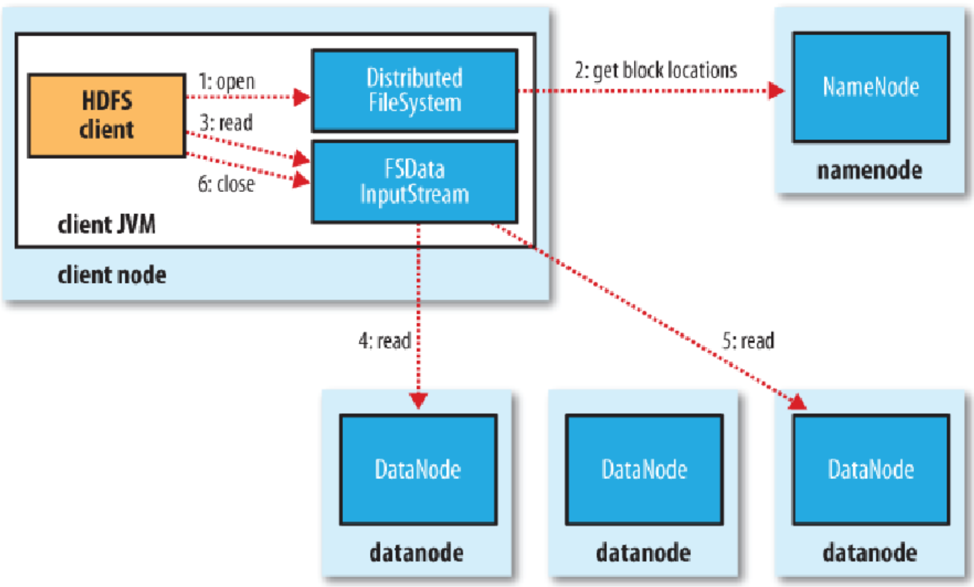


HDFS: Block replication

- NameNode periodically receives heartbeat and blockreport from each DataNode
  - Blockreport: list of blocks on a DataNode



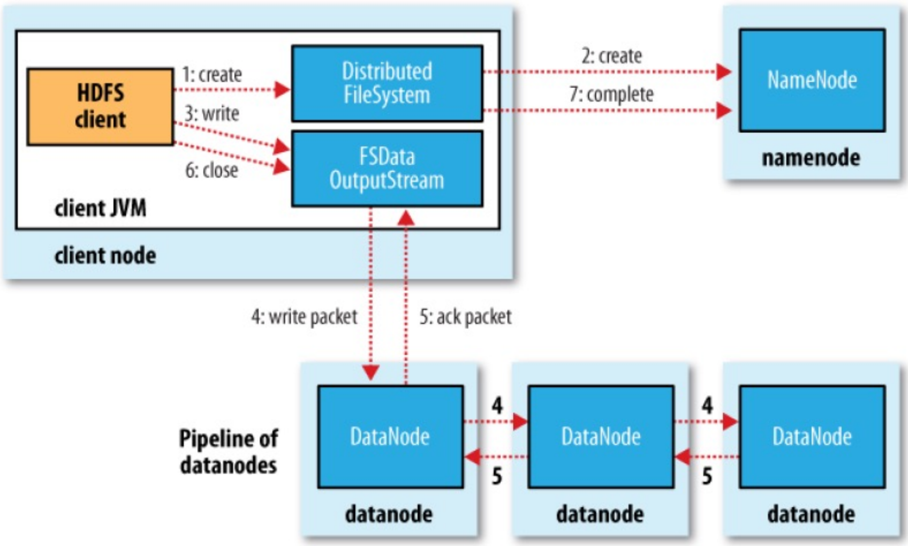
HDFS: File read



Source: “Hadoop: The definitive guide”

- NameNode is used to get block location

HDFS: File write



Source: "Hadoop: The definitive guide"

- Clients ask NameNode for a list of suitable DataNodes
- This list forms a chain: first DataNode stores the block, then forwards it to the second, and so on

Enhancements in HDFS 3.x

- High availability
  – Support for >= 2 NameNodes (1 active and >=1 standby)
  https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithNFS.html

- Erasure coding as alternative strategy to replication in order to provide fault tolerance
  ✓ Same level of fault tolerance with less storage overhead: from 200% (when replication degree is 3) to 50%
  ✗ Increase in network and processing overhead
  – 2 codes: XOR and Reed-Solomon
  – Erasure coding can be enabled on a per-directory basis
  https://docs.cloudera.com/runtime/7.3.1/scaling-namespaces/topics/hdfs-ec-overview.html

HDFS: security

- HDFS initially lacked robust security mechanisms
- Recent versions support authentication (Kerberos and LDAP), authorization (ACLs), and encryption (data at rest and in transit)
- Can be integrated with Apache Ranger, which provides security across Hadoop ecosystem https://ranger.apache.org
  - Centralized security administration
  - Fine-grained authorization
  - Different authorization methods (role-based AC, attribute-based AC, etc.)
  - Centralize auditing of user access and administrative actions
- Data governance can be provided by third-party tools, e.g., Cloudera Navigator

Distributed Object Stores (DOS)

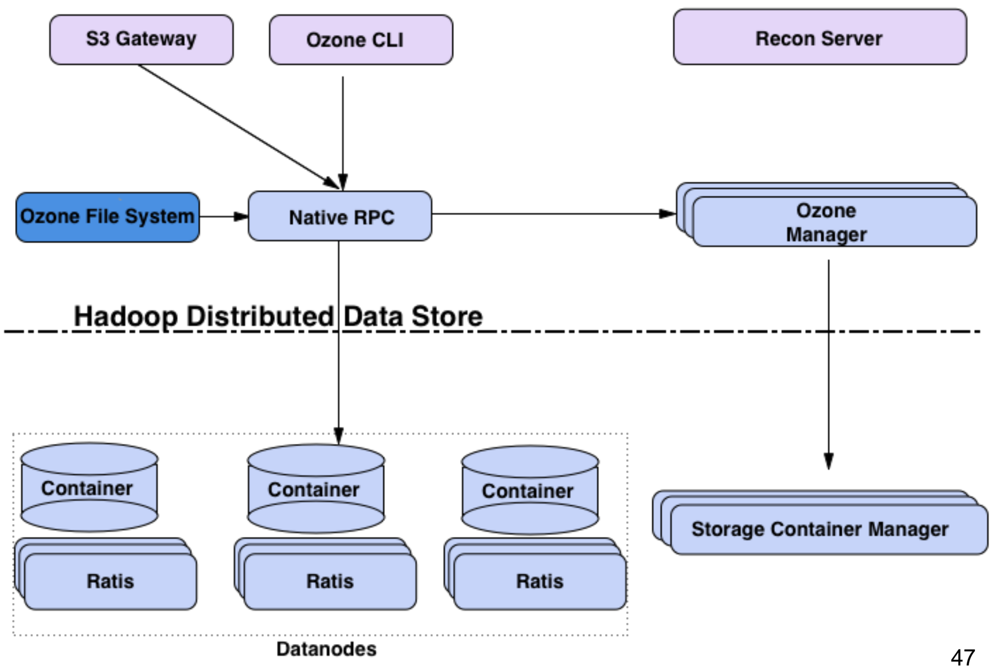
- Designed to handle large volumes of unstructured data by storing objects rather than files
- Data is stored as object with unique identifier, metadata, and content
  - Object aka blob
- No hierarchical directory structure
- Mostly read-intensive workloads
- Challenges
  - Variety of media types (photos, videos, documents, …)
  - Variety of sizes: from tens of KBs (e.g., profile pictures) to a few GBs (e.g., videos)
  - Volume: ever-growing number of blobs to be stored and served

Object store: Apache Ozone

- Highly scalable, distributed object store
  https://ozone.apache.org
- Built on Hadoop Distributed Data Store, a highly available, replicated block storage layer
- Separation of metadata management layer and data storage layer
- Strongly consistent distributed storage thanks to Raft protocol
  - Apache Ratis https://ratis.apache.org: high-performance Java library for Raft protocol
- Secure: access control and transparent data encryption

Ozone: architecture

- Ozone Manager: name space
- Storage Container Manager: physical and data layer
- Recon: management interface



Object store: Ambry

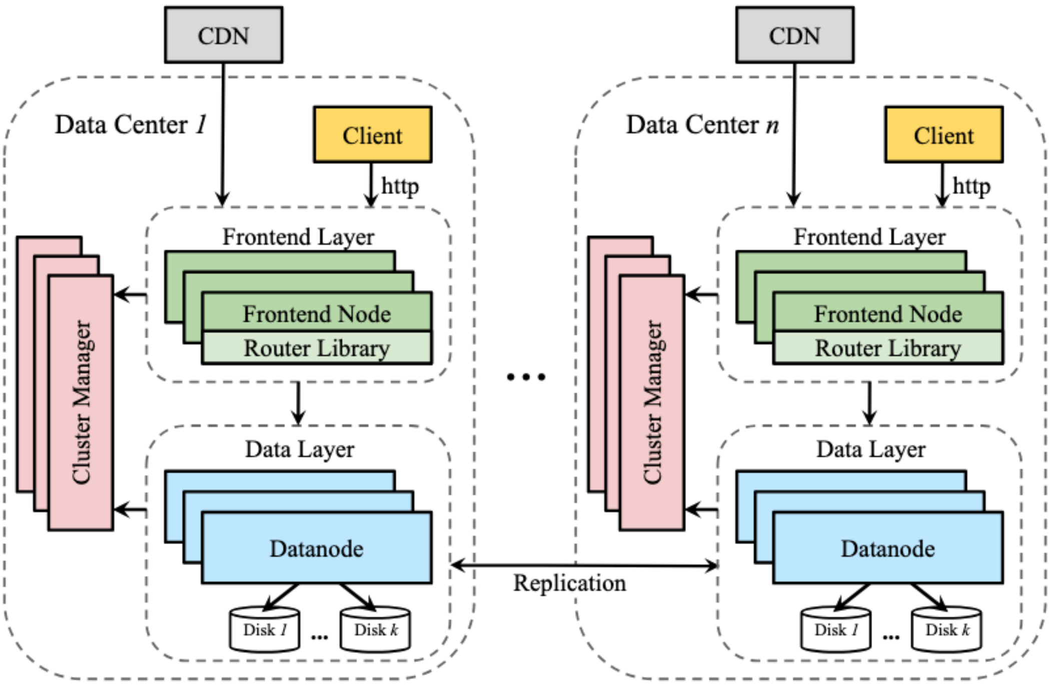
- LinkedIn's object store
- 800M put and get ops/day (over 120 TB in size), 10K reqs/sec. (in 2016)
- Immutable objects (designed for media objects)
- Low-latency, high-throughput
- Optimized for both small and large objects
- Geo-distributed: high durability and availability
- Decentralized architecture
- A number of techniques
  - Logical blob grouping, asynchronous replication, rebalancing mechanisms, zero-cost failure detection, and OS caching

Noghabi et al. Ambry: LinkedIn's Scalable Geo-Distributed Object Store, SIGMOD '16 https://dl.acm.org/doi/pdf/10.1145/2882903.2903738

https://github.com/linkedin/ambry

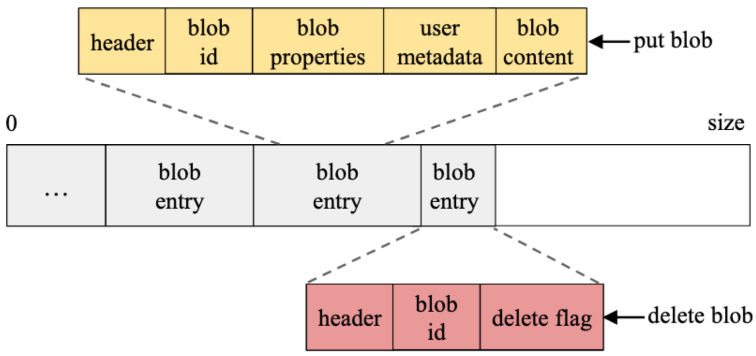
Ambry: architecture

- Decentralized multi-tenant system across geographically distributed data centers



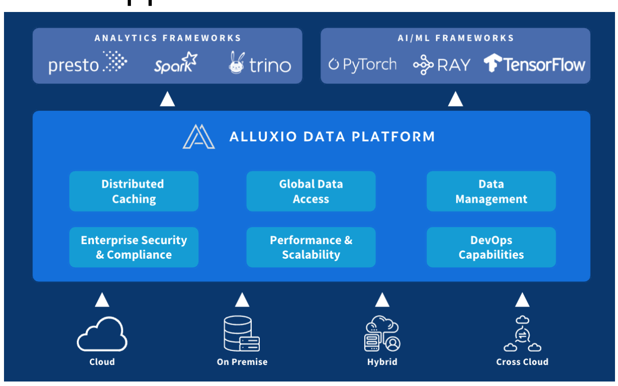
Ambry: partitions and blobs

- Data is organized in virtual units called *partitions*
  - Partition: logical grouping of a number of blobs, implemented as a large, fixed-size file, replicated on multiple Datanodes
- Physical placement of partitions on machines
- Decoupling of logical and physical placement
  - Transparent data movement (necessary for rebalancing)
  - No rehashing of data during cluster expansion



Storing in memory: Alluxio ALLUXIO

- Distributed **in-memory** storage system www.alluxio.io
- Adds a data access layer between storage and computation
  - Interposed between persistent storage layer (e.g., HDFS, AWS S3, …) and processing frameworks for analytics and AI (e.g., Spark, Flink, TensorFlow, …)
- Goal: storage unification and abstraction
  - Brings data from storage closer to applications
  - Enables applications to connect to different storage systems through a common interface and a global namespace

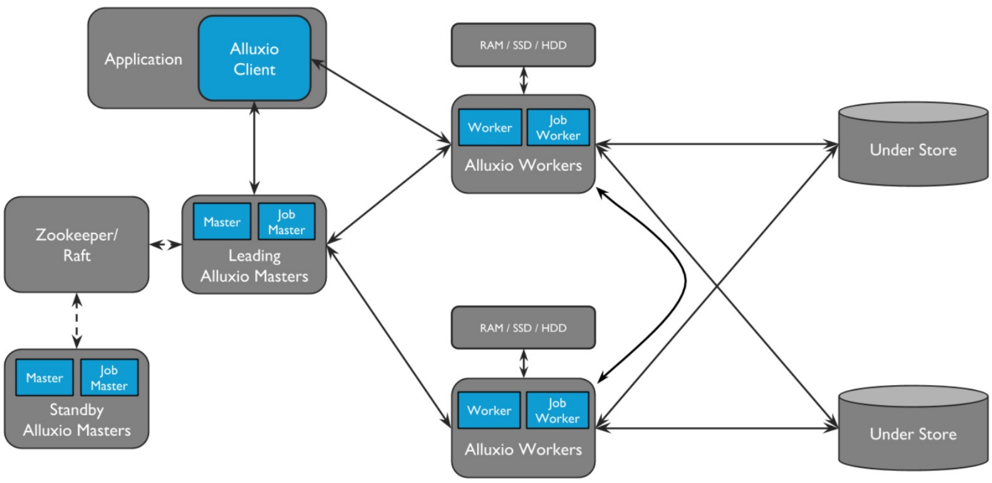


Alluxio

- History
  - Originated from Tachyon project at AMPLab (UC Berkeley)
  - Evolved as data orchestration technology for analytics and AI for the cloud
- Features
  - High read/write throughput, at memory speed
  - Commonly used as distributed shared caching service
  - How to address RAM volatility? Avoid replication and use re-computation (**lineage**) to achieve fault tolerance
    - One copy of data in memory (fast)
    - Upon failure, re-compute data using lineage: keep track of executed ops and, in case of failure, recover lost output by re-executing ops that created the output
    - Borrowed from Spark

Alluxio: Architecture

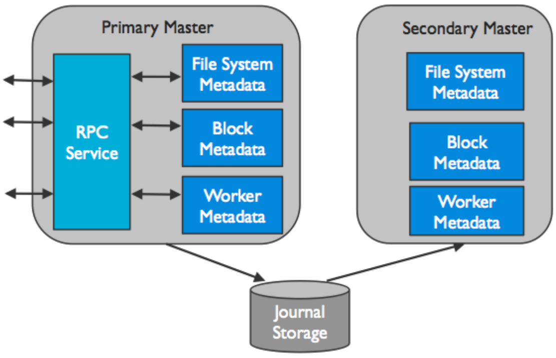
- Master-worker architecture (like GFS, HDFS)
- Replicated masters, multiple workers
  - Passive standby approach to ensure master fault tolerance
  - Consensus: Zookeeper, Raft



Alluxio: Architecture

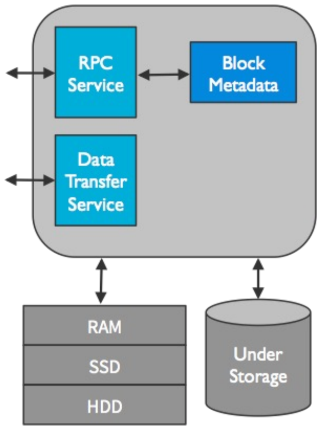
Master

- Stores metadata of storage system
- Responds to client requests
- Tracks lineage information
- Computes checkpoint order
- Secondary master(s) for fault tolerance



Workers

- Manage local storage (RAM, SSD, HDD)
- Access to “under storage” (e.g., HDFS, S3), not managed by Alluxio
- Periodically heartbeat to primary master



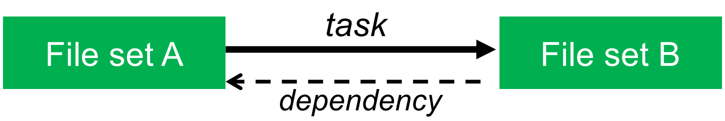
docs.alluxio.io/os/user/stable/en/overview/Architecture.html

Alluxio: Lineage and persistence

Alluxio consists of two (logical) layers:

- **Lineage layer**: tracks sequence of operations that have created a particular data output
  - Write-once semantics: data is immutable once written
  - Frameworks using Alluxio *track* data dependencies and *recompute* them when failure occurs
  - API for managing and accessing lineage information

Task reads file set A and writes file set B



- **Persistence layer**: persists data onto storage, used to perform asynchronous checkpoints
  - Efficient checkpointing algorithm
    - Avoids checkpointing temporary files
    - Checkpoints hot files first (i.e., the most read files)
    - Bounds re-computation time

Data storage so far: Summing up

- Distributed file systems: GFS and HDFS
  - Master/worker architecture, originally single master
  - Decouple metadata from data, also control and data flows
  - Designed for high-throughput, large files, batch applications
- Distributed object stores: Ozone and Ambri
  - Master/worker architecture, multi-master
  - Decouple data control and data storage
- Alluxio
  - In-memory storage system
  - Master/worker architecture
  - No replication: tracks changes (lineage), recovers data using checkpoints and re-computations

References

- Ghemawat et al., The Google File System, *Proc. ACM SOSP '03* https://static.googleusercontent.com/media/research.google.com/it//archive/gfs-sosp2003.pdf
- Hildebrand and Serenyi, Colossus under the hood: a peek into Google's scalable storage system, 2021 https://cloud.google.com/blog/products/storage-data-transfer/a-peek-behind-colossus-googles-file-system
- Video on Colossus: A peek behind the VM at the Google Storage infrastructure, 2020 https://www.youtube.com/watch?v=q4WC_6SzBz4
- Shafer et al., The Hadoop Distributed Filesystem: Balancing Portability and Performance, *Proc. ISPASS '10* https://www.jeffshafer.com/publications/papers/shafer_ispass10.pdf
- Li, Alluxio: A Virtual Distributed File System, 2018 https://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-29.pdf