Introduction to Big Data

Pooya Jamshidi
pooya.jamshidi@ut.ac.ir

Ilam University
School of Engineering,
Computer Group

March 6, 2025

Ilam University

Storage for Big Data

Overview

- Network File System (NFS)
- Google File System (GFS)
- Hadoop File System (HDFS)

Motivation

- Moore's law:
  - The number of transistors in a dense integrated circuit (IC) doubles about every two years.
- Kryder's Law:
  - *Inside of a decade and a half, hard disks had increased their capacity 1,000-fold, a rate that Intel founder Gordon Moore himself has called flabbergasting.*
  - This is much faster than the two-year doubling time of semiconductor chip density suggested by Moore's law!
- Unfortunately, disk speeds don't increase at the rate of capacity.

Replacement Rates

HPC	%	COM1	%	COM2	%
Hard drive	30.6	Power supply	34.8	Hard drive	49.1
Memory	28.5	Memory	20.1	Motherboard	23.4
Misc/Unk	14.4	Hard drive	18.1	Power supply	10.1
CPU	12.4	Case	11.4	RAID card	4.1
Motherboard	4.9	Fan	8	Memory	3.4
Controller	2.9	CPU	2	SCSI cable	2.2
QSW	1.7	SCSI Board	0.6	Fan	2.2
Power supply	1.6	NIC Card	1.2	CPU	2.2
MLB	1.0	LV Pwr Board	0.6	CD-ROM	0.6
SCSI BP	0.3	CPU heatsink	0.6	Raid Controller	0.6

- HPC dataset was collected in a large cluster systems using supercomputers.
- Data sets COM1 and COM2 were collected in at two different cluster systems at a large internet service provider with many distributed and separately managed sites.

Why use multiple disks?

- Capacity
  - More disks allow us to store more data.
- Performance
  - Access multiple disks in parallel.
  - Each disk can be working on independent read or write.
  - Overlap seek and rotational positioning time for all.
- Reliability
  - Recover from disk (or single sector) failures.
  - Will need to store multiple copies of data to recover.

Redundant Array of Inexpensive Disks (RAID)

- Redundant Array of Inexpensive Disks (RAID): A data storage virtualization technology that combines multiple physical disk drive components into one or more logical units for the purposes of data redundancy, performance improvement, or both.
- **Hardware RAID**
  - Storage box you attach to computer.
  - Same interface as single disk, but internally much more
    - Multiple disks
    - More complex controller
    - NVRAM (holding parity blocks)

HP Smart Array P420 6Gb/s PCI-E x8 SAS Raid Controller

Redundant Array of Inexpensive Disks (RAID)

- **Software RAID**
  - OS (device driver layer) treats multiple disks like a single disk.
  - Software does all extra work.
  - The performance and reliability are lower than that of hardware RAID.
- **Interface for both**
  - Linear array of bytes, just like a single disk (but larger).

Network File Systems (NFS)

Introduction

- The most commercially successful and widely available remote file system protocol.
  - Note that NFS is not a distributed filesystem though.
- Designed and implemented by Sun Microsystems in 1985.
- Reasons for success:
  - The NFS protocol is *public domain*.
  - *Sun* used to sell the implementation to all people for less than the cost of implementing it themselves.

NFS Overview

- Views a set of interconnected workstations as a set of independent machines with independent file systems.
- The goal is to allow some degree of sharing among these file systems (on explicit request).
- Sharing is based on client-server relationships.
- A machine may be both client and server.
- Designed to support UNIX file system semantics.
- The protocol design is transport independent.

NFS Versions

- Version 1 (1985)
  - Only used by Sun Microsystems internally.
- Version 2 (March 1989)
  - Originally developed to use UDP.
  - A stateless protocol. When a packet drops, the entire RPC request must be repeated.
- Version 3 (June 1995)
  - Support for 64-bit file sizes and offsets, to handle files larger than 2 gigabytes (GB).
  - Support for asynchronous writes on the server, to improve write performance.
  - Additional file attributes in many replies, to avoid the need to re-fetch them.

NFS Versions

- Version 4 (December 2000; revised in April 2003; revised again in March 2015)
  - Better performance; mandatory security.
  - Uses a single port and hence plays nicely with firewalls (only uses port 2049).
  - Uses TCP protocol (a stateful protocol).
  - Designed by *Internet Engineering Task Force (IETF)* (not Sun).

NFS Configuration Files

- `/etc/exports`: Its a main configuration file of NFS, all exported files and directories are defined in this file at the NFS Server end.
- `/etc/fstab`: To mount a NFS directory on your system across the reboots, we need to make an entry in `/etc/fstab`.
- `/etc/default/nfs-common` and `/etc/default/nfs-kernel-server`: Configuration file of NFS to control on which port RPC and other services are listening and the NFS version used.
    - These are config files used in Debian Linux.

Why not just run NFS?

- NFS Shortcomings:
  - Scalability
  - Performance
  - Elasticity
  - Reliability

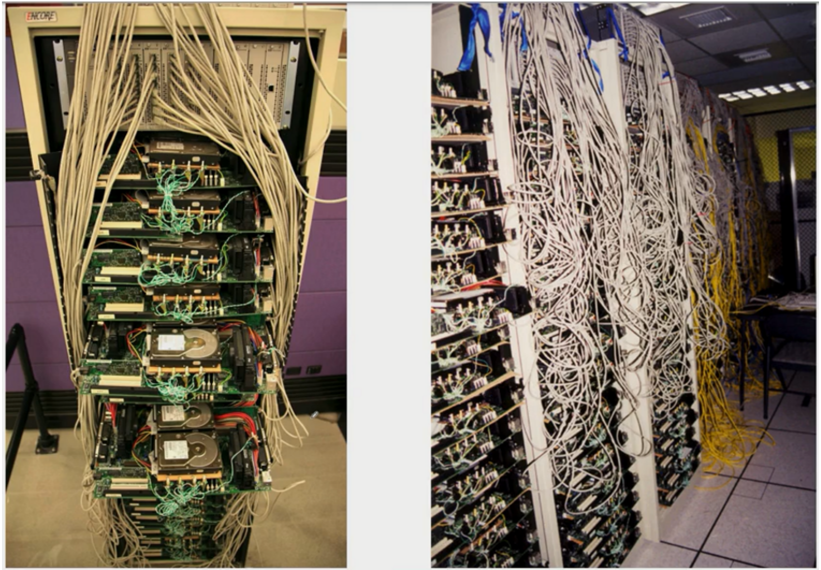
Google File Systems (GFS)

GFS Motivation

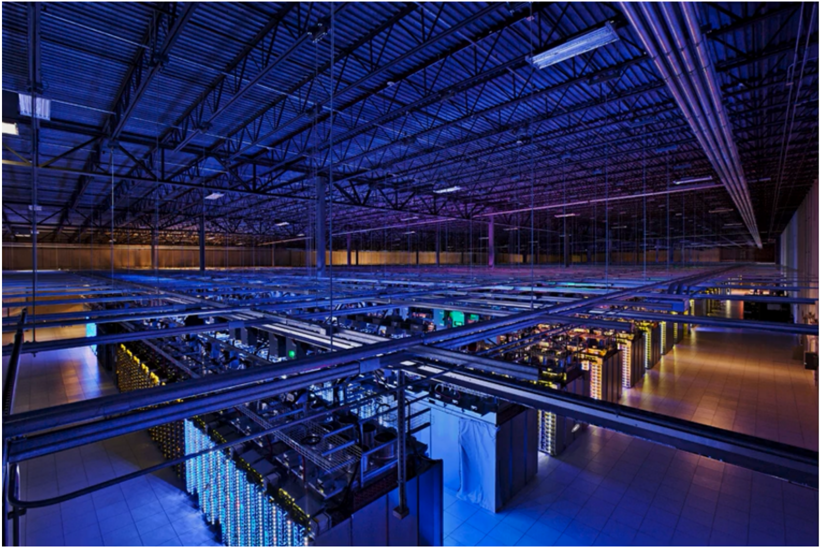
- Google applications exercise specific read/write patterns (Gmail, YouTube, etc.).
- General purpose file systems (like Ext4, NTFS, etc.) are not designed to exploit specific workloads.
- POSIX API (standard for file system communication) is an overkill for specific applications and their requirements.
- Solution – design your own storage system.
- GFS is a *distributed fault-tolerant file system*.

Recommended Reading: Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google file system." *Proceedings of the nineteenth ACM symposium on Operating systems principles*. 2003.

Google's Old Days. . .



GFS Operation Environment



GFS Operation Environment

- Hundreds of thousands of commodity servers.
- Millions of commodity disks.
- Failures are normal (expected):
  - App bugs, OS bugs.
  - Human error.
  - Disk failure, memory failure, net failure, power supply failure.
    - Remember the replacement rate of hard disks.
  - Connector failure.
- Huge number of concurrent readers/writers.

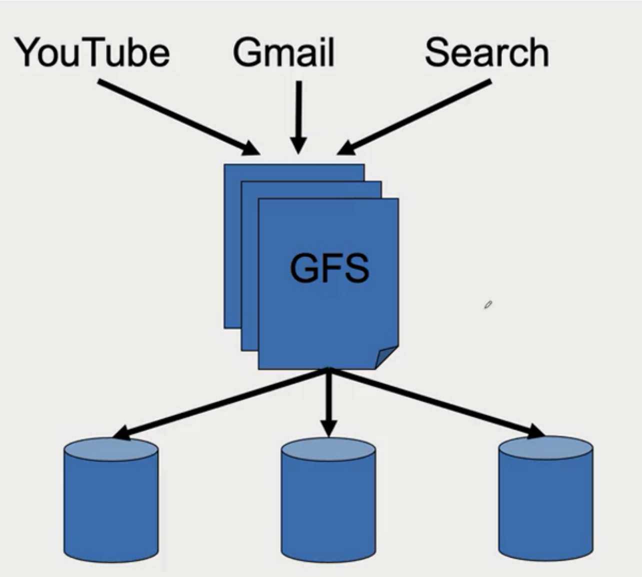
GFS Workload Assumptions

- (Relatively) Small *(in the millions)* number of large files.
- Large files are $\geq$ 100 MB in size (multi-GB files common).
- Large, streaming reads ($\geq$ 1 MB in size).
- Large, sequential writes that append.
- Concurrent appends by multiple clients (e.g., producer-consumer queues).

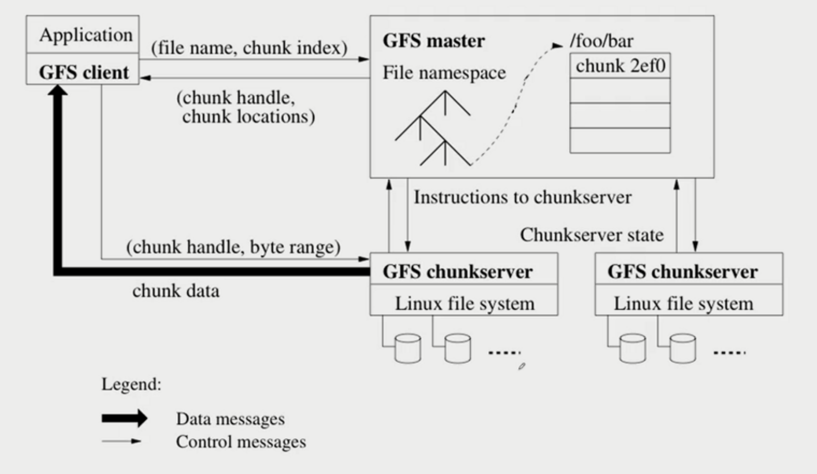
GFS Design Aims

- Maintain data and system availability.
- Handle failures gracefully and transparently.
- Low synchronization overhead between entities of GFS.
- Exploit parallelism of numerous entities.
- Ensure high sustained throughput over low latency for individual reads/writes.

GFS Contex



GFS Architecture



GFS Architecture

- One master server (state replicated on backups).
- Many chunk servers (100s – 1000s):
  - Spread across racks; intra-rack bandwidth greater than inter-rack.
  - Chunk: 64 MB portion of file, identified by 64-bit, globally unique ID.
- Many clients accessing same and different files stored on same cluster.

GFS Architecture: Master Server

- Holds all metadata:
  - Namespace (directory hierarchy)
  - Access control information (per-file)
  - Mapping from files to chunks
  - Current locations of chunks (chunkservers)
- Delegates consistency management.
- Garbage collects orphaned chunks.
- Migrates chunks between chunkservers.
  - Why is migration needed?

Holds all metadata in RAM; very fast operations on file system metadata

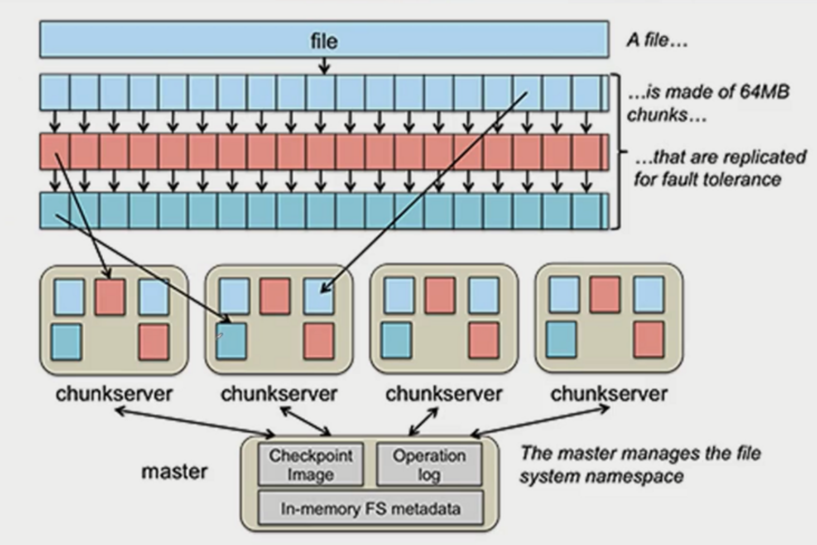
GFS Architecture: Chunkserver

- Stores 64 MB file chunks on local disk using standard Linux filesystem (like Ext4), each with version number and checksum.
- What is the traditional file system chunk/block size?
  - Ext4 uses 4KB.
- Why 64 MB? A key design parameter (Much larger than most file systems.)
  - **Advantages:**
    - Lower the loading of master.
    - Reduce network overhead through a persistent TCP connection.
    - Reduce metadata size stored in the master.
  - **Disadvantages:**
    - Wasted space due to internal fragmentation.
    - Small files consist of a few chunks, which then get lots of traffic from concurrent clients.
    - This can be mitigated by increasing the replication factor.
- GFS uses lazy space allocation to avoid wasting space due to internal fragmentation.

GFS Architecture: Chunkserver (cont'd)

- Has no understanding of overall file system (just deals with chunks).
- Read/write requests specify chunk handle and byte range.
- Chunks replicated on configurable number of chunkservers (default: 3).
- No caching of file data (beyond standard Linux buffer cache).
- Send periodic heartbeats (♡) to Master.

GFS Architecture: File Layout



GFS Architecture: Client

- Issues control (metadata) requests to master server.
- Issues data requests directly to chunkservers.
  - This exploits parallelism and reduces master bottleneck.
- Caches metadata.
- Does no caching of data.
  - No consistency difficulties among clients.
  - Streaming reads (read once) and append writes (write once) don't benefit much from caching at client.

GFS Architecture: Client (cont'd)

- No file system interface at the operating-system level (e.g., under the virtual file system layer.)
  - User-level API is provided.
  - Does not support all the features of POSIX file system access – but looks familiar (i.e., `open`, `close`, `read`, ...).
- Two special operations are supported (not available through POSIX):
  - Snapshot: An efficient way of creating a copy of the current instance of a file or directory tree.
  - Append: Allows a client to append data to a file as an atomic operation without having to lock a file.
    - Multiple processes can append to the same file concurrently without fear of overwriting one another's data.

GFS Working Client Read

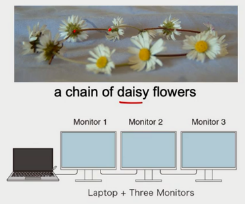
- Client sends master:
  - `read(file name, chunk index)`
- Master's reply:
  - chunk ID, chunk version number, locations of replicas
- Client sends a request to the closest chunkserver with a replica:
  - `read(chunk ID, byte range)`
  - Closest is determined by IP address on a simple rack-based network topology.
- Chunkserver replies with data.

GFS Working Client Write (1)

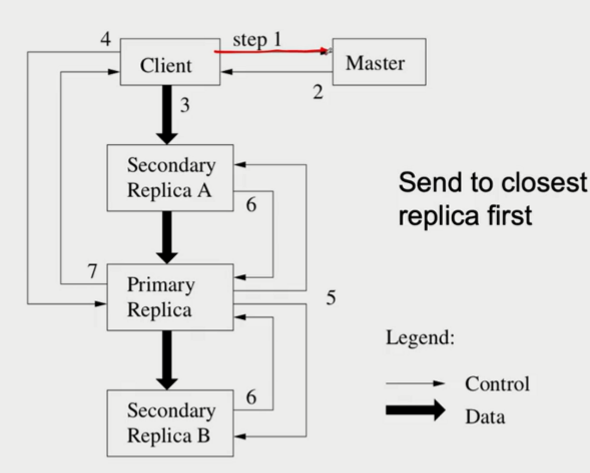
- 3 replicas for each block $\rightarrow$ must write to all.
- When block is created, Master decides placements.
  - Default: two within a single rack, third on a different rack.
- Why?
  - Access time / safety tradeoff.

GFS Working Client Write (2)

- Some chunkserver is primary for each chunk.
  - Master grants lease to primary (typically for 60 sec.)
  - Leases renewed using periodic heartbeat messages between master and chunkservers.
- Client asks master for primary and secondary replicas for each chunk.
- Client sends data to replicas in daisy chain:
  - Pipelined: Each replica forwards as it receives.
  - Takes advantage of full-duplex Ethernet links.



GFS Working Client Write (3)



GFS Working Client Write (4)

- All replicas acknowledge data write to client.
- Client sends write request to primary (commit phase).
- Primary assigns serial number to write request, providing ordering.
- Primary forwards write request with the same serial number to secondary replicas.
- Secondary replicas all reply to primary after completing writes in the same order.
- Primary replies to client.

GFS Working Client Record Append

- Google uses large files as queues between multiple producers and consumers.
- Same control flow as for writes, except...
- Client pushes data to replicas of last chunk of file.
- Client sends request to primary.
- Common case:
  - Request fits in current last chunk:
    - Primary appends data to own replica.
    - Primary tells secondaries to do the same at the same byte offset in theirs.
    - Primary replies with success to client.

GFS Working Client Record Append

- When data won't fit in the last chunk:
  - Primary fills current chunk with padding.
  - Primary instructs other replicas to do the same.
  - Primary replies to client, *"retry on next chunk."*

GFS Working File Deletion

- When client deletes file:
  - Master records deletion in its log.
  - File renamed to hidden name including deletion timestamp.
- Master scans file namespace in background:
  - Removes files with such names if deleted for longer than 3 days (configurable).
  - In-memory metadata erased.
- Master scans chunk namespace in background:
  - Removes unreferenced chunks from chunkservers.

GFS Working Logging at Master

- Master has all metadata information
  - Lose it, and you've lost the filesystem!
- Master logs all client requests to disk sequentially.
- Replicates log entries to remote backup servers.
- Only replies to client after log entries are safe on disk on self and backups!
- Logs cannot be too long – why?

GFS Fault Tolerance (Master)

- Replays log from disk
  - Recovers namespace (directory) information
  - Recovers file-to-chunk-ID mapping (but not location of chunks)
- Asks chunkservers which chunks they hold:
  - Recovers chunk-ID-to-chunkserver mapping
- If chunk server has older chunk, it's stale.
  - Chunk server down at lease renewal.
- If chunk server has newer chunk, adopt its version number.
  - Master may have failed while granting lease.

GFS Fault Tolerance (Chunkserver)

- Master notices missing heartbeats
- Master decrements count of replicas for all chunks on dead chunkserver
- Master re-replicates chunks missing replicas in background
  - Highest priority for chunks missing the greatest number of replicas

GFS Fault Tolerance (Master)

- **Replays log from disk**
  - Recovers **namespace** (directory) information
  - Recovers **file-to-chunk-ID** mapping (but not location of chunks)
- **Asks chunkservers which chunks they hold:**
  - Recovers **chunk-ID-to-chunkserver** mapping
- **If chunk server has older chunk, it's stale.**
  - Chunk server down at lease renewal.
- **If chunk server has newer chunk, adopt its version number.**
  - Master may have failed while granting lease.

GFS Fault Tolerance (Chunkserver)

- **Master notices missing heartbeats**
- **Master decrements count of replicas** for all chunks on **dead chunkserver**
- **Master re-replicates chunks missing replicas in background**
  - Highest priority for **chunks missing greatest number of replicas**

GFS Limitations

- **Security?**
  - Trusted environment, trusted users.
  - But that doesn't stop users from interfering with each other...
- **Does not mask all forms of data corruption**
  - Requires application-level checksum.

GFS Limitations (cont'd)

- **Master biggest impediment to scaling.**
    - Performance bottleneck.
    - Holds all data structures in memory.
    - Takes long time to rebuild metadata.
    - Most vulnerable point for reliability.
    - **Solution:**
        - Have systems with multiple master nodes, all sharing set of chunk servers.
        - Not a uniform namespace.
- **Large chunk size.**
    - Can't afford to make smaller, since this would create more work for master.

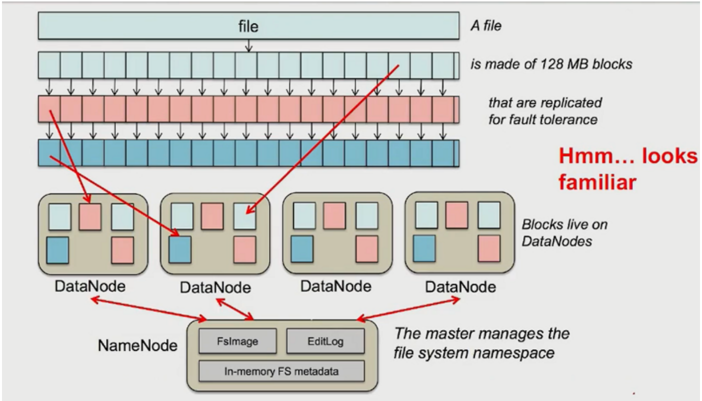
GFS Summary

- **Success: Used actively by Google to support search service and other applications.**
  - Availability and recoverability on cheap hardware
  - High throughput by decoupling control and data
  - Supports massive data sets and concurrent appends
- **Semantics not transparent to apps**
  - Must verify file contents to avoid inconsistent regions, repeated appends (at-least-once semantics)

GFS Summary

- **Performance not good for all apps**
  - Assumes read-once, write-once workload (no client caching!)
- **Replaced in 2010 by Colossus (GFS v2)**
  - Eliminate master node as single point of failure
  - Targets latency problems due to more latency-sensitive applications
  - Reduce block size to be between **1∼8 MB**
- **Recommended read:** *"Cluster-Level Storage @ Google"* presentation at International Workshop on Parallel Data Storage & Data Intensive Scalable Intensive Computing Systems.

HDFS



file
A file
is made of 128 MB blocks
that are replicated
for fault tolerance
Hmm… looks
familiar
Blocks live on
DataNodes
DataNode
DataNode
DataNode
DataNode
NameNode
FsImage
EditLog
In-memory FS metadata
The master manages the
file system namespace

GFS vs. HDFS

GFS	HDFS
Master	NameNode
chunkserver	DataNode
operation log	journal, edit log
chunk	block
random file writes possible	only append is possible
multiple writer, multiple reader model	single writer, multiple reader model
chunk: 64KB data, 32-bit checksum	data & metadata file (checksum, timestamp)
default block size: 64MB	default block size: 128MB

A Glance at HDFS Commands

Command & **Description**
`hdfs dfs -ls /` & List all files/directories for the given HDFS destination path.
`hdfs dfs -ls -d /hadoop` & Lists directories as plain files (details of hadoop folder).
`hdfs dfs -ls -h /data` & Formats file sizes in human-readable fashion.
`hdfs dfs -ls -R /hadoop` & Recursively lists all files and subdirectories.
`hdfs dfs -ls /hadoop/dat*` & Lists all files matching the pattern (starting with 'dat').

A Glance at HDFS Commands

Command & **Description**
`hdfs dfs -cp /hadoop/file1 /hadoop1` Copies file from source to destination in HDFS.
`hdfs dfs -cp -p /hadoop/file1 /hadoop1` & Copies file while preserving metadata (ownership, timestamps).
`hdfs dfs -cp -f /hadoop/file1 /hadoop1` & Copies file, overwriting if it exists.
`hdfs dfs -mv /hadoop/file1 /hadoop1` & Moves file from source to destination.
`hdfs dfs -rm /hadoop/file1` & Deletes file (sends to trash).

For a more comprehensive list of HDFS commands, checkout the "Hadoop HDFS Command Cheatsheet".

Summary

- **GFS / HDFS**
  - Data-center customized API, optimizations
  - Append-focused distributed file system
  - Separate control (filesystem) and data (chunks)
  - Replication and locality
  - Rough consistency $\rightarrow$ apps handle rest