Big Data Computing

Master's of Science in Computer Science

Sapienza University of Rome

2026/2027

Gabriele Tolomei

tolomei@di.uniroma1.it

Who Am I?

Gabriele Tolomei

Associate Professor of Computer Science, Sapienza University of Rome

- Leads the **HERCOLE Lab** (**H**uman-**E**xplainable, **R**obust, and **CO**llaborative **LE**arning)
- Co-founder & CSO at **tellmewAI** (AI Act compliance platform)

Research pillars: Explainable AI · Adversarial & Robust ML · Federated & Scalable Learning

Background: Ph.D. Ca' Foscari Venice · M.Sc. & B.Sc. University of Pisa · Previously at Yahoo Labs (London)

Useful Information

Class Schedule

- Wed, 8:00–11:00 (Room 1L)
- Thu, 10:00–12:00 (Room 2L)

Contacts

- Email: tolomei@di.uniroma1.it
- Office: Room 106, 1st floor, Building E, Viale Regina Elena 295 (map)
- Course website: https://gtolomei.github.io/teaching/bdc.html
- Google Classroom (mandatory): click to join

Office Hours

Drop me a message and we'll arrange a meeting time, either in person or online! 🙂

Class Material

- Released on the course website
- Suggested books (though not mandatory):
  - ***Mining of Massive Datasets*** — Leskovec, Rajaraman & Ullman (freely available online)
  - ***Foundations of Data Science*** — Blum, Hopcroft & Kannan (freely available online)
  - ***Designing Data-Intensive Applications*** — Kleppmann
  - ***Introduction to Information Retrieval*** — Manning, Raghavan & Schütze
  - ***Understanding Machine Learning*** — Shalev-Shwartz & Ben-David
- Additional research papers referenced throughout lectures!

Assessment and Grading

The final grade is determined through an **oral seminar** on a recent, high-impact research paper.

- The paper must be **chosen autonomously by the student** and approved by the instructor.
- Papers should come from **top-tier venues** (e.g., SIGMOD, VLDB, OSDI, SOSP, NeurIPS, ICML, KDD).

Seminar Format

The seminar can be delivered **individually or in pairs**:

Format	Presentation	Q&A	Total
Individual	~12 min	~3 min	~15 min
Pair	~25 min	~5 min	~30 min

For pair presentations, the presentation and Q&A should be **equally balanced between the two students**.

Honors (30 cum laude) require exceptional mastery of the topic, deep critical insight, and flawless academic communication.

Course Organization

The course is structured into **four main modules**:

- **1. Big Data Infrastructure**

  Scale-out storage, fault tolerance via lineage, MapReduce, memory-centric DAG execution (Spark), streaming.

- **2. High-Dimensional Data Representations**

  Geometry of high-dimensional spaces, dimensionality reduction (Johnson–Lindenstrauss), columnar layouts, text/image/graph embeddings.

- **3. Classical Non-Learning Tasks**

  Web-scale indexing, sampling theory, Approximate Query Processing, sub-linear sketching (HyperLogLog, Count-Min), Locality-Sensitive Hashing.

- **4. Scale-Out Learning Systems**

  Large-scale ERM, parallel/asynchronous SGD, Parameter Server, Federated Learning, memory-bandwidth optimization.

Module 1: Big Data Infrastructure

Lecture 1.1: The Scale-Out Paradigm & Distributed Storage

Philosophy of Scale

> *"At scale, everything fails. The art of distributed systems is making that survivable."*
> *— Folklore of distributed computing*

A single, infinitely fast and reliable machine does not exist.
Big data computing begins by accepting this, and designing around it.

Lecture Roadmap

This lecture covers the storage foundations of big data systems — the substrate on which all later computation (MapReduce, Spark, streaming) is built.

Specifically, we will study:

1. **Why Scale Out** — the limits of single-machine computing
2. **Foundations of Distributed Storage** — design goals and abstractions
3. **The Google File System (GFS)** — architecture and mechanics
4. **Data Locality & Replication** — moving computation, not data
5. **HDFS** — the open-source realization of GFS
6. **Failure Handling** — fault tolerance as a first-class concern
7. **Synthesis** — connecting storage to computation

> *Let's build the foundation everything else stands on.*

Why This Matters: The Scale of Modern Data

Big data systems exist because data volumes exceed what any single machine can store or process:

- **Web-scale crawls:** hundreds of billions of pages, petabytes of raw HTML.
- **Log and telemetry data:** sensor and clickstream data accumulating at terabytes per day.
- **Scientific and enterprise data:** genomics, astronomy, and large enterprise transaction logs reaching exabyte scale.

Key question for this lecture: if a single disk, single machine, and single point of failure cannot hold or serve this data, what structural alternative exists?

§ 1 — Why Scale Out?

The Data Growth Problem

Data volumes have grown far faster than the capacity of any individual machine:

- Doubling disk capacity every few years is not enough when data doubles every few months.
- Some workloads (web indices, sensor networks) generate data faster than a single disk can even write it.

Single-Machine Limits

A single server, no matter how powerful, hits hard physical ceilings:

- **CPU:** finite cores per socket, finite sockets per motherboard.
- **RAM:** finite DIMM slots, finite addressable memory per machine.
- **Disk:** finite drive bays, finite I/O bandwidth per controller.
- **Network:** finite NIC bandwidth in and out of one box.

Even a "big" machine is still bounded — it just moves the ceiling, not eliminates it.

Vertical Scaling

Vertical scaling ("scale up") means making a single machine bigger:

- Add more CPU cores, more RAM, faster/larger disks to the *same* node.
- Software often requires no changes — the machine just does more.
- Conceptually simple: one bigger box instead of many small ones.

The question is not whether this works, but how far it can go — and at what cost.

Vertical Scaling: The Cost Curve

High-end hardware does not scale linearly in price:

Configuration	Relative Capacity	Relative Cost
Commodity server	1×	1×
High-end server	4×	~10×+
Top-of-line mainframe-class	16×	~100×+

- Premium hardware (more sockets, exotic interconnects, specialized memory) carries a steep price premium.
- Past a certain point, doubling capacity can cost far more than double.

Vertical Scaling: The Hard Ceiling

Beyond cost, vertical scaling eventually hits **physical limits**:

- Motherboards have a fixed maximum number of CPU sockets and DIMM slots.
- A single machine has a single power/cooling envelope.
- At some point, **no amount of money buys a bigger single machine** for a given workload.

Conclusion: vertical scaling buys time, but it is not a long-term strategy for web-scale or exabyte-scale data.

Horizontal Scaling

Horizontal scaling ("scale out") means adding *more machines*, not bigger ones:

- A workload is spread across many independent nodes.
- Each node is relatively modest in capability.
- Capacity grows by adding nodes to a cluster, not by upgrading any single node.

This is the foundational strategy behind virtually all big data systems.

Horizontal Scaling: Commodity Hardware Philosophy

A key insight behind systems like GFS and HDFS: build clusters from **cheap, unreliable, commodity machines**, not expensive, reliable ones.

- Thousands of inexpensive nodes can be cheaper in aggregate than a handful of premium ones.
- Individual node failure becomes *expected*, not exceptional.
- The system, not the hardware, is responsible for reliability.

This philosophy shift — software guarantees reliability instead of hardware — defines the rest of this lecture.

Vertical vs. Horizontal Scaling: Comparison

Aspect	Vertical (Scale Up)	Horizontal (Scale Out)
Approach	Bigger single machine	More machines
Ceiling	Hard physical limit	Effectively unbounded
Cost growth	Superlinear at high end	Roughly linear (commodity HW)
Failure handling	Hardware reliability	Software-level fault tolerance
Complexity	Low (same software model)	High (distributed coordination)

The New Failure Model: Failure Is Normal

At cluster scale, hardware failure stops being an edge case:

- With thousands of disks, **some disk is failing right now**, statistically speaking.
- Networks partition, machines crash, racks lose power.
- A system designed assuming "failures are rare" will fail constantly in production at this scale.

Big data infrastructure must treat failure as a routine, expected event — not an exception.

Cost-per-Byte vs. Cost-per-FLOP at Scale

Two different cost dimensions matter differently when scaling out:

- **Cost-per-byte (storage):** commodity disks make raw storage cheap; the dominant cost becomes replication and durability overhead.
- **Cost-per-FLOP (compute):** commodity CPUs make raw compute cheap; the dominant cost becomes coordination and data movement.

Both observations point to the same conclusion: at scale, the bottleneck shifts from raw hardware cost to how well the system coordinates many cheap parts.

✅ Checkpoint 1 — Scaling Fundamentals

Let's test our basic understanding:

1. What is the fundamental difference between vertical and horizontal scaling?
2. Why does vertical scaling eventually hit a hard physical ceiling?
3. Why do big data systems prefer many commodity machines over few premium ones?
4. Why must failure be treated as a "normal" event in scale-out systems, rather than an exception?

Write down your thoughts before moving to Section 2.

§ 2 — Foundations of Distributed Storage

What Is a Distributed File System?

A **Distributed File System (DFS)** presents a single, unified file system interface backed by storage spread across many machines.

- Files appear to clients as if stored on one logical file system.
- In reality, file contents are split, replicated, and distributed across a cluster of machines.
- The DFS hides this complexity behind familiar file operations (open, read, write, close).

Goal: petabyte-scale storage that looks, to applications, like a normal (if large) file system.

Design Goal: Throughput vs. Latency

Big data DFSs are optimized for a specific trade-off:

- **Throughput-oriented:** maximize total data moved per second across the whole cluster.
- **Latency-tolerant:** individual operations may take longer than on a local disk.

This is a deliberate choice: batch analytics over petabytes cares about aggregate bandwidth, not millisecond-level response time for any single request.

Design Goal: Huge Files, Sequential Access

Big data DFSs are designed around a specific access pattern:

- Files are typically **huge**: gigabytes to terabytes per file, not kilobytes.
- Access is predominantly **sequential**: read the whole file (or large chunks) start to finish.
- Random small reads/writes are explicitly **not** the primary target workload.

This assumption shapes nearly every architectural decision that follows.

Design Assumption: Write-Once, Read-Many

Most big data DFSs assume a specific lifecycle for files:

- A file is **written once** (e.g., appended, then closed).
- It is then **read many times** by different analytics jobs.
- In-place modification of existing file content is rare or disallowed.

This relaxes a major engineering burden: the system does not need to support efficient arbitrary in-place updates.

The Block/Chunk Abstraction

Rather than storing a file as one contiguous unit, a DFS splits it into fixed-size **blocks** (or **chunks**):

- A large file becomes a sequence of same-size blocks.
- Each block is stored, replicated, and managed independently.
- Different blocks of the same file can live on different machines.

This is the single most important structural decision in distributed storage design.

Why Fixed-Size Blocks?

Fixing block size (rather than variable-size file fragments) simplifies the whole system:

- **Simplified metadata:** the system tracks block locations, not arbitrary byte ranges.
- **Simplified load balancing:** blocks are interchangeable units of work to place and move.
- **Simplified replication:** each block is replicated independently, uniformly.

Block Size Trade-offs

Choosing the right block size balances competing concerns:

Block Size	Pros	Cons
Small (e.g., 4 KB)	Fine-grained, low waste for small files	Huge metadata overhead at scale
Large (e.g., 64–128 MB)	Low metadata overhead, efficient sequential I/O	Wasteful for small files, coarser load balancing

Big data DFSs choose large blocks (tens to hundreds of MB) — the opposite of traditional local file systems.

Metadata vs. Data Separation

A central architectural decision: **separate the system that tracks "where things are" from the system that stores "the actual bytes."**

- **Metadata service:** namespace, directory structure, block-to-location mappings.
- **Data servers:** the actual block contents, stored on local disks.

This separation allows each subsystem to be optimized independently — metadata for fast lookups, data servers for raw I/O throughput.

Single Point of Coordination: Pros and Cons

Many DFS designs (including GFS and HDFS) use a **single coordinating node** for metadata:

Pros:

- Simple, globally consistent view of the namespace.
- No need for distributed consensus on every metadata change.

Cons:

- Potential single point of failure.
- Potential scalability bottleneck if metadata operations grow too large.

We will see exactly how GFS and HDFS each manage this risk later in this lecture.

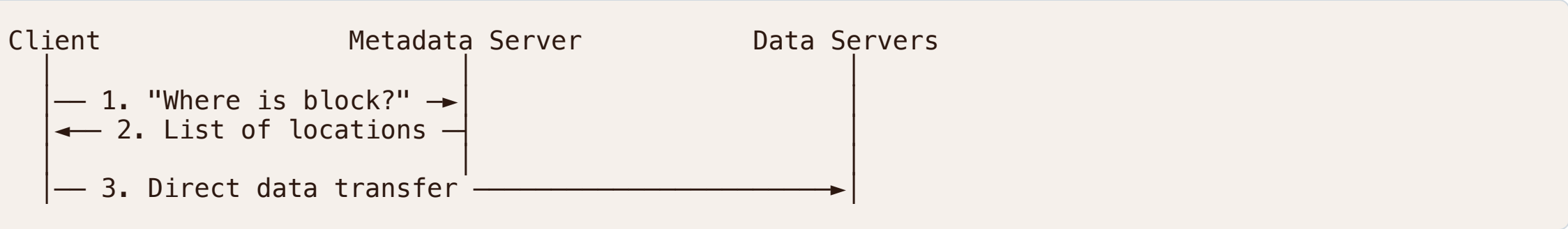
Namespace Management

The metadata service maintains the **namespace**: the hierarchical structure of directories and files.

- Tracks file-to-block mappings (which blocks make up which file).
- Tracks block-to-location mappings (which servers hold each block's replicas).
- Held primarily **in memory** for fast lookups — critical for performance at scale.

Client Interaction Model

Clients never read or write data *through* the metadata server — only its **location information**:



This decouples metadata lookups (small, frequent) from data transfer (large, bandwidth-heavy) — avoiding a metadata-server bottleneck.

Network Topology Awareness

Distributed storage systems are aware of **physical network topology**, not just logical machine identity:

- Machines are grouped into **racks**, connected by top-of-rack switches.
- Intra-rack bandwidth is typically much higher than inter-rack bandwidth.
- Placement and scheduling decisions account for this hierarchy (machine → rack → datacenter).

This awareness becomes essential when we discuss replica placement in Section 4.

Why Replication Is Necessary

In a single machine, RAID can protect against disk failure. At cluster scale, this is insufficient:

- RAID protects against *disk* failure, not *machine*, *rack*, or *network* failure.
- With thousands of machines, the *expected* number of failures at any moment is non-zero.
- The DFS itself must replicate data **across machines** (and ideally across racks) to survive these failures.

Replication is not an optimization here — it is the core durability mechanism.

✅ Checkpoint 2 — DFS Design Principles

Let's test our understanding:

1. Why do big data DFSs use large block sizes instead of small ones?
2. Why is metadata kept separate from the actual data storage servers?
3. Walk through the client interaction model: what does the metadata server *not* do?
4. Why is RAID-style protection insufficient at cluster scale?

Discuss with your colleague or write down your answers.

§ 3 — The Google File System (GFS)

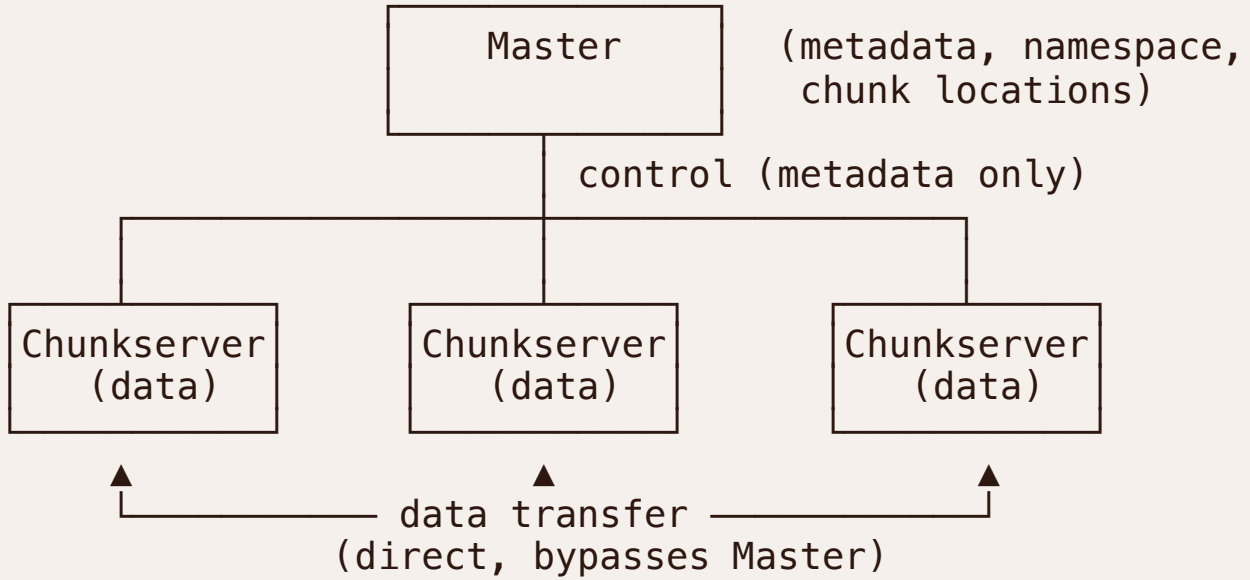
GFS: Motivation and Original Context

The **Google File System (GFS)**, published in 2003, was designed for Google's internal workloads:

- Storing and processing massive web crawl data, logs, and index files.
- Running on **thousands of commodity machines**, where component failure is the norm, not the exception.
- Optimized for huge files, sequential reads, and append-heavy writes — not general-purpose POSIX semantics.

GFS is the direct architectural ancestor of HDFS, which we study later in this lecture.

GFS Architecture Overview



One Master coordinates; many Chunkservers store and serve data directly to clients.

The GFS Master: Role and Responsibilities

The **Master** is the single metadata-managing node in a GFS cluster:

- Maintains the file system **namespace** (directory structure, file → chunk mapping).
- Maintains the **chunk location map** (which chunkservers hold which chunks).
- Manages **chunk lease** assignment for writes (covered shortly).
- Coordinates chunk creation, replication, and garbage collection.

GFS Master: Metadata Held in Memory

All Master metadata is kept **in memory** for speed:

- Namespace tree, file-to-chunk mappings, chunk-to-chunkserver mappings.
- In-memory access allows the Master to handle metadata operations extremely fast.
- Chunk location information is **not persisted** — it is rebuilt by polling chunkservers at startup.

Trade-off: speed in exchange for a bounded total namespace size (limited by Master memory).

GFS Master: Single Master Rationale and Risk

GFS deliberately uses a **single Master**, not a distributed metadata cluster:

Rationale:

- Drastically simplifies design — no distributed consensus needed for metadata changes.
- A single, globally consistent view of chunk placement decisions.

Risk:

- Single point of failure (mitigated — see fault tolerance later in this section).
- Potential bottleneck if metadata operation volume grows too large.

GFS Chunkservers: Role

Chunkservers are the workhorses of GFS — they store the actual chunk data:

- Each chunkserver stores chunks as regular files on its local (Linux) file system.
- Chunkservers serve read/write requests for the chunks they hold, directly to clients.
- Chunkservers periodically report their chunk inventory to the Master via heartbeats.

GFS Chunk Size: 64 MB

GFS uses a notably large chunk size: **64 MB** (much larger than typical local file system blocks).

Why so large?

- Reduces the number of chunks per file → reduces Master metadata size.
- Reduces the frequency of client-Master interactions (fewer chunk lookups per large read).
- Encourages large, sustained, sequential data transfers per chunk.

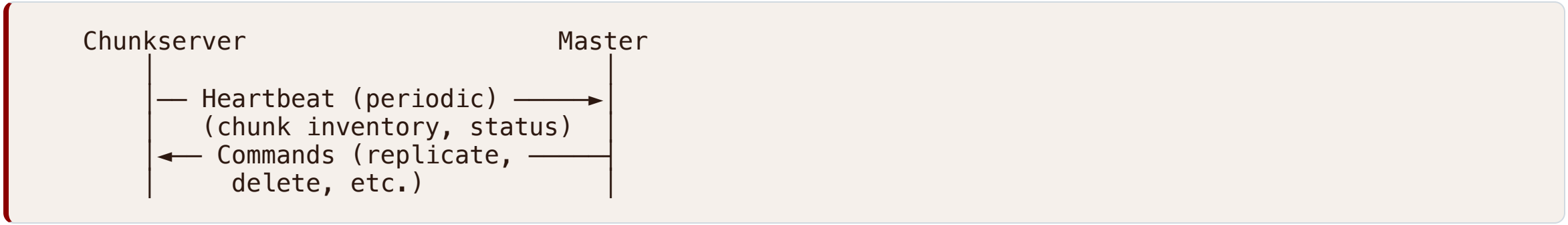
Downside: small files waste a disproportionate fraction of a chunk — an accepted trade-off given GFS's target workload.

GFS Chunk Replication Factor

Each chunk is replicated, by default, **3 times** across different chunkservers:

- Provides durability against individual chunkserver/disk failure.
- Replicas are placed with rack-awareness (detailed in Section 4).
- The replication factor is configurable per file, allowing more critical data to be replicated further.

Master–Chunkserver Interaction Flow

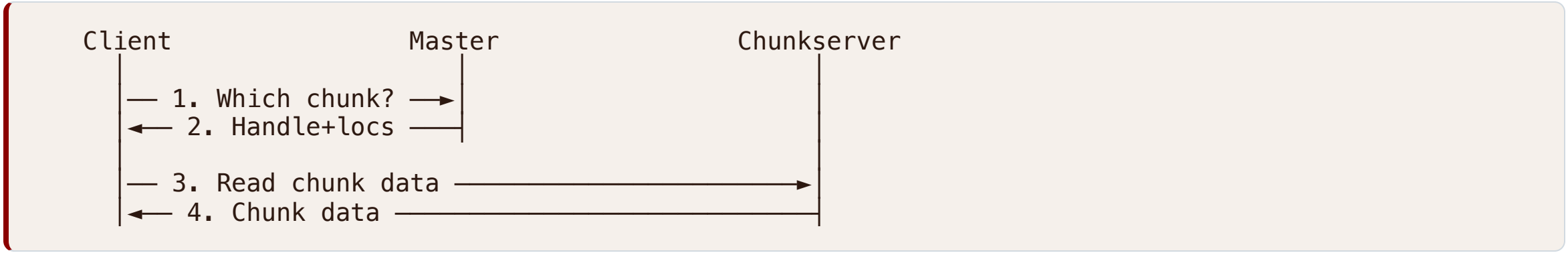


- Heartbeats let the Master detect chunkserver liveness and inventory without storing persistent chunk-location metadata.
- The Master piggybacks instructions (e.g., "replicate this chunk elsewhere") on heartbeat responses.

GFS Read Path — Step by Step

1. Client computes the chunk index for the requested file offset.
2. Client asks the **Master** for the chunk handle and the list of chunkserver locations.
3. Master replies with chunk handle + locations (cached briefly by the client).
4. Client contacts the **nearest replica chunkserver directly** (bypassing the Master).
5. Chunkserver streams the requested data directly to the client.

GFS Read Path — Diagram

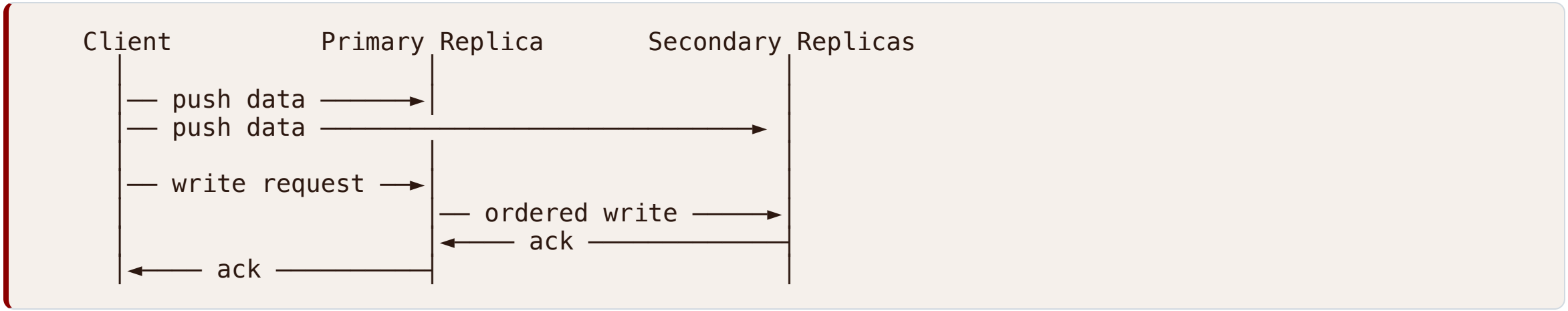


Note: the Master is involved only in step 1–2 (metadata); all bulk data flows directly between client and chunkserver.

GFS Write Path — Step by Step

1. Client asks the Master for the chunkservers holding the chunk's replicas, including which one holds the current **lease** (the primary).
2. Client pushes data to **all replicas** (in any order), which buffer it temporarily.
3. Client sends a **write request** to the **primary** replica.
4. The primary assigns a serial order to the write and forwards it to **secondary** replicas in that order.
5. Secondaries acknowledge; the primary replies to the client once all replicas confirm.

GFS Write Path — Diagram



Data flow (push) and control flow (write order) are deliberately decoupled for performance.

Primary/Secondary Replica Roles During Writes

GFS uses a **lease mechanism** to designate one replica as primary for a period of time:

- The **primary** replica decides the serial order of concurrent write/mutation requests.
- **Secondary** replicas simply apply mutations in the order dictated by the primary.
- This avoids needing a separate consensus protocol for every single write — ordering is delegated to one replica at a time.

Mutation Order and Consistency in GFS

Because all replicas apply mutations in the **same order** (as dictated by the primary), GFS achieves:

- **Consistency across replicas:** all replicas converge to the same chunk state.
- **Relaxed consistency guarantees overall:** GFS does not guarantee strict POSIX semantics; concurrent writers may see "defined but undefined" interleavings of data — an accepted trade-off for performance.

GFS Heartbeat Mechanism

Heartbeats serve multiple purposes in GFS beyond simple liveness detection:

- Chunkservers report which chunks they currently hold.
- The Master uses this to detect under-replicated chunks (after a failure) or over-replicated chunks (after recovery).
- The Master issues replication/deletion commands piggybacked on heartbeat responses.

Chunk Lease Mechanism

The **lease** is a time-bounded grant of "primary" status to one replica for a given chunk:

- The Master grants a lease to one chunkserver replica for a chunk.
- The lease has a timeout; it can be extended if the chunkserver is actively handling mutations.
- If the primary becomes unreachable, the Master can grant a new lease to another replica once the old one expires.

This avoids the Master needing to make an ordering decision for every single write — it delegates that to whichever replica holds the lease.

Garbage Collection in GFS

GFS does **not** immediately reclaim space when a file is deleted:

- A "deleted" file is renamed to a hidden name and kept for a few days.
- Actual chunk space is reclaimed later via a **lazy garbage collection** process during regular Master/chunkserver communication.

Benefit: simpler, more robust deletion — accidental deletes can be undone within the retention window, and deletion does not require an immediate, synchronous cluster-wide operation.

Master Fault Tolerance: Operation Log & Checkpoints

Although the Master keeps metadata in memory, it must survive crashes:

- All namespace **mutations** are appended to a persistent **operation log**, replicated to multiple machines.
- Periodically, the Master writes a **checkpoint** (a compact snapshot of its full state) to avoid replaying an ever-growing log from scratch.
- On restart: load latest checkpoint, then replay the log entries since that checkpoint.

Master Fault Tolerance: Shadow Masters

GFS also runs **shadow masters** — read-only replicas of the Master's state:

- Shadow masters continuously apply the same operation log to mirror the primary Master's state.
- They can serve **read-only** metadata queries, reducing load on the primary Master.
- They do **not** automatically take over write/coordination duties if the primary Master fails — that requires separate recovery/promotion.

GFS Limitations

The single-Master design, while simple, introduces known limitations:

- **Metadata scalability:** the entire namespace must fit in one machine's memory.
- **Master as a bottleneck:** very high rates of metadata operations (e.g., huge numbers of small files) can overwhelm a single Master.
- **Failover is not instantaneous:** recovering from a Master crash takes time, even with checkpoints and logs.

These limitations directly motivate later architectural evolutions, including HDFS Federation, covered in Section 5.

✅ Checkpoint 3 — GFS Architecture

Let's test our understanding:

1. What two types of information does the GFS Master manage, and why is metadata kept entirely in memory?
2. Walk through the GFS read path: what is the Master's role, and what is the chunkserver's role?
3. Explain the function of the chunk lease mechanism in the GFS write path.
4. How does GFS tolerate a Master crash without losing metadata?

Review your answers before moving on.

§ 4 — Data Locality & Replication Strategy

The Principle of Data Locality

At big data scale, **moving computation to where data already resides** is far cheaper than moving data to computation:

- Network bandwidth between machines is orders of magnitude slower than local disk/memory bandwidth.
- Moving terabytes of data across a network for every job is prohibitively slow and expensive.
- Instead: schedule the computation **on or near** the chunkserver that already holds the relevant data.

"Move Computation to Data, Not Data to Computation"

This principle, central to MapReduce (covered in the next lecture), originates directly from the storage layer studied here:

- The storage system (GFS/HDFS) exposes **where** each chunk/block physically resides.
- A scheduler can then place a computing task on (or near) that same machine.
- Only the *much smaller* computation results need to travel over the network — not the raw input data.

Why Network Bandwidth Is the Scarce Resource

In a typical cluster:

Resource	Approximate Bandwidth
Local disk read	Hundreds of MB/s – a few GB/s
Same-rack network	~1–10 Gb/s shared across the rack
Cross-rack network	Lower, shared across the datacenter backbone

Local I/O is fast and "free." Network I/O — especially cross-rack — is the bottleneck resource that data locality is designed to minimize.

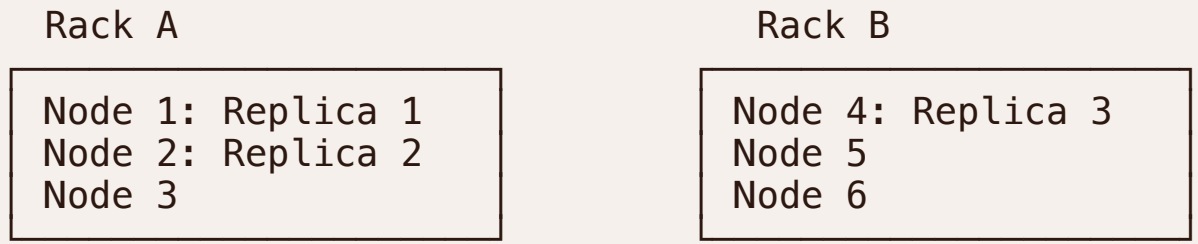
Rack-Aware Replica Placement Strategy

Replica placement is not random — it deliberately balances two competing goals:

- **Durability:** replicas should be spread across failure domains (different machines, ideally different racks).
- **Write/read efficiency:** too much spreading increases cross-rack network traffic during writes.

The common compromise: place some replicas within the same rack, and at least one replica in a different rack.

Replica Placement: Diagram



- Two replicas in Rack A (fast intra-rack replication for the write path).
- One replica in Rack B (survives an entire rack failure, e.g., shared power/switch failure).

Replication Factor Trade-offs: Durability vs. Cost

Increasing the replication factor improves durability but at a direct storage cost:

Replication Factor	Storage Overhead	Durability
1 (no replication)	1×	None — any failure loses data
2	2×	Tolerates 1 failure
3 (typical default)	3×	Tolerates 2 simultaneous failures

Higher replication factors are used selectively for especially critical or frequently-read data.

Replication Factor Trade-offs: Write Overhead

Every additional replica also adds **write cost**:

- Each write must be propagated to *every* replica before being acknowledged (as seen in the GFS write path).
- More replicas → more network traffic and more time before a write is considered durable.
- This is a direct trade-off against the durability gained from extra replicas.

Failure Domains: Disk, Node, Rack, Datacenter

Replication strategy must account for **correlated failures** across different scopes:

- **Disk failure:** affects one chunk's local copy.
- **Node failure:** affects all chunks on that machine.
- **Rack failure:** affects all nodes sharing a switch/power source.
- **Datacenter failure:** affects an entire facility (power outage, network outage, disaster).

Spreading replicas across these domains protects against progressively larger-scale failures.

Worked Example: Probability of Data Loss vs. Replication Factor

Assume each node has an independent annual failure probability of $p = 0.02$ (2%). For a chunk replicated across r *independent* nodes, all replicas failing **simultaneously within the same short window** has probability:

$$P(\text{total loss}) \approx p^r$$

Replication Factor r	$P(\text{loss}) \approx p^r$
1	0.02
2	0.0004
3	0.000008

Even a modest replication factor reduces loss probability by orders of magnitude — assuming failures are independent, which rack-aware placement helps ensure.

Re-replication After Node Failure

When a node fails, the system does not simply leave affected chunks under-replicated:

1. The Master/NameNode detects the failure (missed heartbeats).
2. It identifies all chunks that were stored on the failed node, now under-replicated.
3. It schedules **new replication** of those chunks from surviving replicas to other healthy nodes.
4. Once complete, the system returns to full replication factor — automatically, with no manual intervention.

Balancing Cluster Storage Utilization

Beyond simple replication, the system also tries to **balance storage usage** across nodes:

- Avoid overloading a subset of nodes while others sit underused.
- New chunk placement and re-replication both consider current disk utilization across the cluster.
- A well-balanced cluster avoids hotspots that could become bottlenecks or premature failure points.

✅ Checkpoint 4 — Locality & Replication

Let's test our understanding:

1. Why is "moving computation to data" preferable to "moving data to computation" at scale?
2. Describe a rack-aware replica placement strategy and explain why it balances durability against write cost.
3. Using the simplified independent-failure model, why does increasing replication factor from 2 to 3 reduce loss probability so dramatically?
4. What happens, step by step, when a chunkserver/DataNode permanently fails?

Consolidate your answers before proceeding.

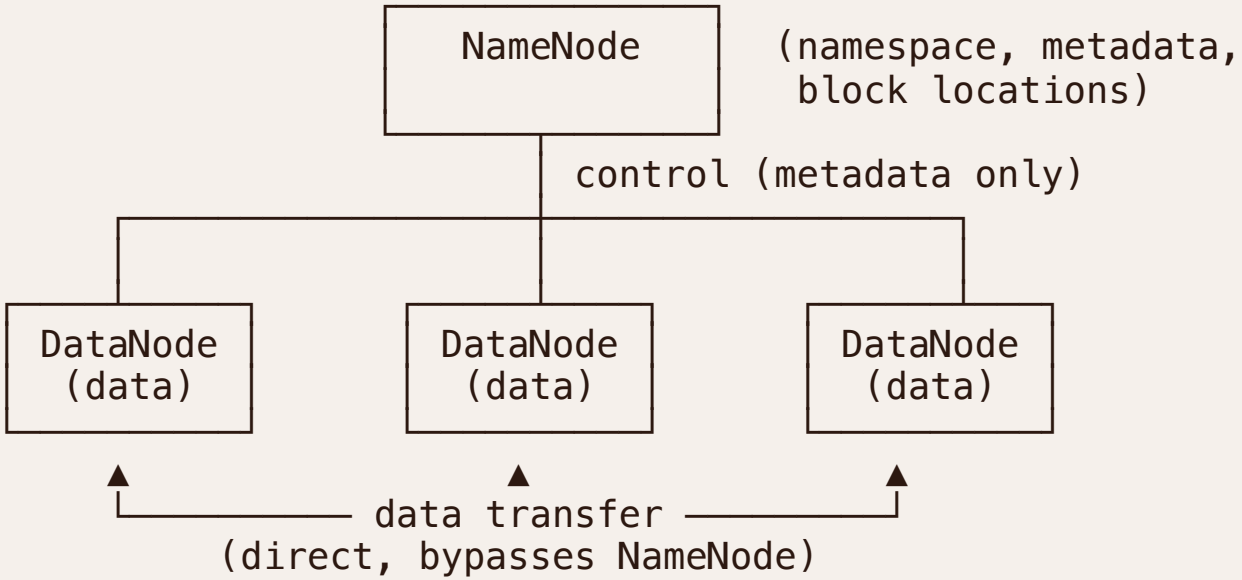
§ 5 — Hadoop Distributed File System (HDFS)

HDFS Origins: GFS Paper Inspiration

HDFS was designed as an open-source storage system directly inspired by the 2003 GFS paper:

- Built as part of the Apache Hadoop project to support large-scale, fault-tolerant batch processing (originally for MapReduce, the topic of the next lecture).
- Adopts the same fundamental architecture: one metadata-managing node, many data-storing nodes, large blocks, replication.
- Terminology differs slightly from GFS, but the underlying design philosophy is nearly identical.

HDFS Architecture Overview



Structurally identical to GFS — NameNode plays the role of the Master; DataNodes play the role of Chunkservers.

HDFS NameNode: Role and Responsibilities

The **NameNode** is HDFS's metadata-managing node, directly analogous to the GFS Master:

- Maintains the file system namespace (directories, files, permissions).
- Maintains the file-to-block mapping and block-to-DataNode location mapping.
- Coordinates block replication, deletion, and rebalancing decisions.
- Keeps all of this metadata in memory for fast access.

HDFS DataNodes: Role

DataNodes are HDFS's data-storing nodes, directly analogous to GFS Chunkservers:

- Store HDFS blocks as files on their local underlying file system (e.g., ext4, XFS).
- Serve read and write requests for blocks **directly** to clients.
- Periodically send **block reports** and **heartbeats** to the NameNode.

HDFS Block Size vs. GFS Chunk Size

HDFS uses a similar but distinct default block size compared to GFS:

System	Default Size	Typical Range
GFS chunk	64 MB	Fixed at 64 MB
HDFS block	128 MB	Configurable, commonly 128–256 MB

Same underlying rationale as GFS: large blocks minimize metadata overhead and favor large sequential transfers.

NameNode Metadata Structures: FsImage and EditLog

HDFS persists NameNode metadata using two complementary structures:

- **FsImage:** a complete, point-in-time snapshot of the entire file system namespace.
- **EditLog:** an append-only log of every namespace mutation (create, delete, rename, etc.) since the last FsImage.

This is the HDFS equivalent of GFS's checkpoint + operation log mechanism.

NameNode Startup Process

On restart, the NameNode reconstructs its full in-memory state as follows:

1. Load the most recent **FsImage** into memory.
2. Replay every entry in the **EditLog** recorded since that FsImage was taken.
3. The resulting in-memory state reflects the namespace exactly as it was before the restart.
4. A new FsImage checkpoint is typically taken soon after, to bound future EditLog replay time.

Secondary NameNode: Common Misconception Clarified

The name **Secondary NameNode** is often misunderstood:

- It is **not** a hot standby or failover replica for the primary NameNode.
- Its actual job: periodically merge the FsImage and EditLog into a new checkpoint, then send it back to the primary NameNode.
- This offloads the (potentially expensive) checkpointing work from the primary NameNode's critical path.

For real failover capability, HDFS requires a separate High Availability configuration — covered next.

HDFS High Availability: Active/Standby NameNode

To address the single-point-of-failure risk, production HDFS deployments run **HA NameNode pairs**:

- One **Active** NameNode handles all client requests.
- One or more **Standby** NameNodes stay synchronized, ready to take over.
- A shared, consistent edit log (often via a separate consensus-based logging service) keeps Standby nodes up to date.
- Automatic failover mechanisms detect Active NameNode failure and promote a Standby.

DataNode Block Reports and Heartbeats

DataNodes communicate with the NameNode through two periodic mechanisms:

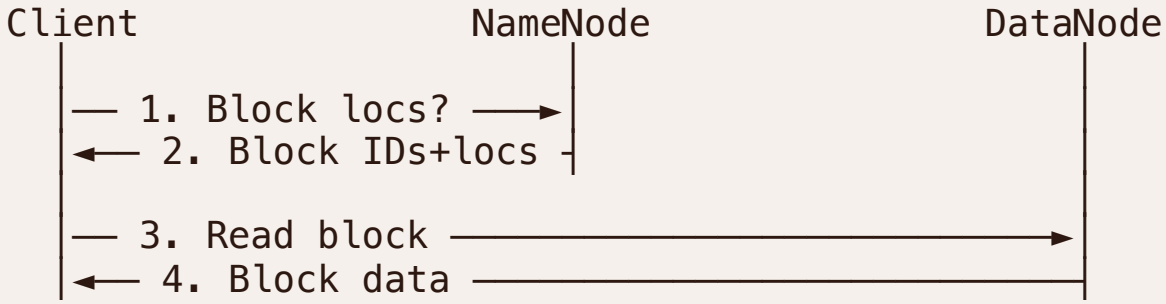
- **Heartbeats** (frequent, e.g., every few seconds): simple liveness signal, "I am alive."
- **Block reports** (less frequent): the full list of blocks currently stored on that DataNode.

Together, these let the NameNode maintain an accurate, near-real-time view of the cluster's block distribution and node health.

HDFS Read Path — Step by Step

1. Client asks the **NameNode** for the block locations comprising the target file.
2. NameNode returns block IDs and an ordered list of DataNodes holding each block (ordered by proximity to the client when possible).
3. Client connects **directly** to the nearest available DataNode for each block.
4. DataNode streams the block data directly to the client; the NameNode is not involved in this transfer.

HDFS Read Path — Diagram

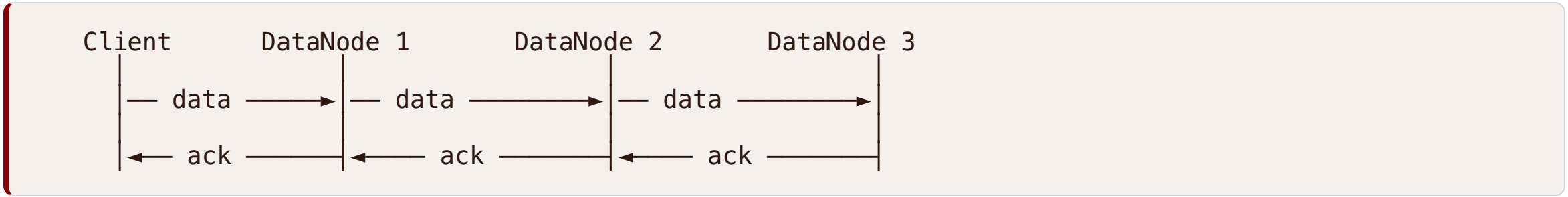


Identical structural pattern to the GFS read path — metadata lookup, then direct data transfer.

HDFS Write Path — Step by Step

1. Client asks the NameNode to create a new file; NameNode checks permissions and allocates the first block.
2. NameNode returns a list of DataNodes to form a **replication pipeline** for that block.
3. Client streams data to the **first** DataNode in the pipeline.
4. Each DataNode forwards data to the next DataNode in the pipeline, in sequence.
5. Acknowledgments flow back up the pipeline once all DataNodes have persisted the data.

HDFS Write Path — Diagram



Note the pipelined structure: data flows once through a chain of DataNodes, rather than the client pushing to each replica independently as in classic GFS.

HDFS Replica Placement Policy

HDFS's default placement policy (for replication factor 3) follows a rack-aware pattern:

- **1st replica:** on the DataNode where the client is writing (or a nearby node).
- **2nd replica:** on a different node, in a **different rack** from the first.
- **3rd replica:** on a different node, in the **same rack** as the second replica.

This balances cross-rack durability (replicas 1 and 2/3 are on different racks) against limiting cross-rack write traffic (replicas 2 and 3 share one rack-to-rack hop).

Rack Awareness Configuration in HDFS

HDFS does not automatically infer network topology — administrators must configure it explicitly:

- A configurable script or mapping tells HDFS which rack each DataNode belongs to.
- Without this configuration, HDFS treats all nodes as if they were in a single rack, losing the durability benefits of cross-rack placement.

Operationally, this is a key deployment step often overlooked in naive cluster setups.

HDFS Federation: Scaling the NameNode

To address the single-NameNode metadata bottleneck (the same limitation discussed for GFS), HDFS supports **Federation**:

- Multiple independent NameNodes, each managing a distinct portion of the namespace.
- DataNodes register with, and serve blocks for, **all** NameNodes in the federation.
- Each NameNode still has its own in-memory metadata limit, but the *aggregate* namespace capacity scales with the number of NameNodes.

Comparing GFS vs. HDFS: Terminology Mapping

Concept	GFS Term	HDFS Term
Metadata node	Master	NameNode
Data-storing node	Chunkserver	DataNode
Storage unit	Chunk	Block
Default unit size	64 MB	128 MB
Checkpoint helper	Shadow Master	Secondary NameNode

Comparing GFS vs. HDFS: Architectural Differences

While structurally similar, a few differences are worth noting:

- HDFS's write path uses an explicit **replica pipeline** (data flows node-to-node in sequence); GFS's write path has the client **push to all replicas directly**, then coordinate via a primary/lease mechanism.
- HDFS persists metadata via **FsImage + EditLog**; GFS uses an analogous **checkpoint + operation log**.
- HDFS Federation and HA NameNode pairs are explicit, separately-configured extensions; GFS's original paper describes shadow masters but a comparable, fully productized HA/federation story evolved later in industry practice.

Block-Based Storage Revisited: Benefits at Scale

Returning to the block/chunk abstraction introduced in Section 2 — now grounded in two concrete systems:

- Fixed-size blocks made metadata for petabyte-scale files tractable for a single in-memory NameNode/Master.
- Uniform block size simplified replication, load balancing, and re-replication after failures.
- The same abstraction underlies virtually every distributed storage system that followed GFS and HDFS.

Why Block-Based Storage Suits Batch Processing

The block abstraction is not incidental — it directly enables efficient batch computation:

- Each block can be processed largely **independently**, enabling parallel computation across many blocks at once.
- Block locations are exposed to schedulers, enabling the data-locality principle from Section 4.
- This sets up exactly the structure that **MapReduce** (next lecture) exploits: one "Map" task per block, scheduled near that block's data.

HDFS Access Patterns: Append-Only, No Random Writes

Consistent with the write-once-read-many assumption from Section 2:

- HDFS traditionally supports **append** operations to existing files, but not arbitrary in-place modification.
- Random writes into the middle of an existing file are not supported in the classic HDFS model.
- This restriction simplifies consistency and replication — exactly the same trade-off GFS makes.

✅ Checkpoint 5 — HDFS Architecture

Let's test our understanding:

1. Map each HDFS component (NameNode, DataNode, block) to its GFS equivalent.
2. What is the actual role of the Secondary NameNode, and what is it commonly mistaken for?
3. Walk through the HDFS write path — how does the "pipeline" structure differ from GFS's write path?
4. How does HDFS Federation address the single-NameNode metadata bottleneck?
5. Why does block-based storage naturally support parallel batch computation?

Make sure you understand these mechanisms before wrapping up.

§ 6 — Failure Handling & Fault Tolerance

Failure Is the Default Assumption at Scale

Returning to the theme introduced in Section 1, now grounded in concrete mechanisms:

- With thousands of disks and machines, failures are a **continuous background process**, not isolated incidents.
- Every mechanism studied so far — replication, heartbeats, leases, checkpoints — exists specifically to make failure survivable.
- A distributed storage system's quality is measured largely by how gracefully it degrades and recovers, not by how rarely it fails.

Types of Failures

Not all failures look the same, and systems must handle several categories:

- **Transient failures:** temporary network blips, brief process pauses (e.g., garbage collection stalls) — the node is fine but briefly unresponsive.
- **Permanent failures:** disk death, hardware failure, node decommissioning — the node or data is gone for good.
- **Correlated failures:** failures that strike multiple machines simultaneously (e.g., rack power loss, top-of-rack switch failure).

Detecting Failure: Heartbeats and Timeouts

Both GFS and HDFS rely on the same basic detection mechanism:

- Each data-storing node sends periodic **heartbeats** to the metadata node.
- If no heartbeat is received within a configured **timeout window**, the node is presumed dead.
- This is inherently a **trade-off**: too short a timeout risks false positives (a healthy-but-slow node marked dead); too long delays real failure detection.

Recovering from Chunkserver/DataNode Failure

When a data-storing node is presumed dead, recovery follows a consistent pattern:

1. Metadata node marks all chunks/blocks on that node as **under-replicated**.
2. It schedules re-replication of each affected chunk/block from a surviving replica to a healthy node.
3. Re-replication proceeds in the background, prioritized to restore full redundancy as quickly as network/disk capacity allows.

This is the same re-replication mechanism introduced in Section 4, now framed explicitly as a recovery process.

Recovering from Master/NameNode Failure

Recovering the **metadata node** itself is more delicate, since it holds the single authoritative namespace view:

- **Without HA configured:** the cluster becomes unavailable for new operations until the Master/NameNode is manually restarted (replaying checkpoint + log, as covered earlier).
- **With HA configured:** a Standby node is promoted to Active, using the shared, continuously-synchronized edit log to resume with minimal interruption.

This is precisely why production deployments almost always run HA configurations despite the added complexity.

Checksums and Silent Data Corruption

Not all failures are obvious crashes — some are **silent**: a disk returns subtly corrupted bits without reporting an error.

- Both GFS and HDFS store **checksums** alongside each block/chunk.
- On every read, the checksum is verified against the data; mismatches indicate corruption.
- A detected corruption triggers the same re-replication mechanism — read from a different (presumably valid) replica, and recreate the corrupted one.

Self-Healing: Continuous Re-replication

The combination of heartbeats, block reports, and checksums creates a continuously **self-healing** system:

- Under-replicated blocks (from node failure) are automatically restored to full replication.
- Corrupted blocks (detected via checksum mismatch) are automatically replaced from valid replicas.
- No human operator needs to manually intervene for routine, individual failures — only for large-scale or unusual events.

Trade-off: Detection Speed vs. False Positives

Tuning the heartbeat timeout illustrates a recurring theme in distributed systems design:

Timeout	Detection Speed	Risk
Short	Fast failure detection	False positives on slow-but-alive nodes; wasted re-replication
Long	Fewer false positives	Slower response to real failures; longer window of reduced redundancy

There is no universally "correct" timeout — it is tuned to the specific cluster's network reliability and workload.

Real-World Numbers: Disk Failure Rates at Scale

Disk failure rates are typically reported as an **Annualized Failure Rate (AFR)**:

- Large-scale studies of commodity disks commonly report AFRs in the range of roughly **1–5% per year**, varying by disk age, model, and operating conditions.
- In a cluster of 10,000 disks at a 2% AFR, this implies an *expected* multiple disk failures **per week**, not per year.

This single number is the empirical justification for the entire "failure is normal" design philosophy introduced in Section 1.

Modeling Disk Failures in One Week

Consider a cluster with:

$$N = 10{,}000 \text{ disks}$$

and an annual failure rate of:

$$\text{AFR} = 2\% = 0.02$$

Let p be the probability that a given disk fails during **one week**.

Assuming, for simplicity, that failures are uniformly distributed throughout the year:

$$p \approx \frac{0.02}{52} \approx 0.0003846$$

Thus, for each disk:

$$\begin{cases} \text{failure during the week} & \text{with probability } p \\ \text{no failure during the week} & \text{with probability } 1 - p \end{cases}$$

The Number of Failures Is a Random Variable

Define the indicator variable for disk i:

$$X_i = \begin{cases} 1 & \text{if disk } i \text{ fails during the week} \\ 0 & \text{otherwise} \end{cases}$$

Then:

$$X_i \sim \text{Bernoulli}(p)$$

The **total** number of disk failures during the week (X) is another random variable:

$$X = \sum_{i=1}^{N} X_i$$

Assuming disk failures are **independent and identically distributed**, X is the sum of n iid Bernoulli random variables:

$$\boxed{X \sim \text{Binomial}(N, p)}$$

with:

$$N = 10{,}000, \qquad p \approx 0.0003846$$

Probability of k Failures in One Week

Since:

$$X \sim \text{Binomial}(10{,}000, 0.0003846)$$

the probability of observing exactly k failures is:

$$\boxed{P(X = k) = \binom{N}{k} p^k (1-p)^{N-k}}$$

For example, the probability of **exactly 4 failures** in any given week is:

$$P(X = 4) = \binom{10{,}000}{4} (0.0003846)^4 (1 - 0.0003846)^{9996} \approx \boxed{19.4\%}$$

Almost 20% chance of experiencing 4 failures!

Expected Number of Failures

For a binomial random variable, it holds that:

$$\mathbb{E}[X] = Np$$

Therefore:

$$\mathbb{E}[X] = 10{,}000 \times \frac{0.02}{52}$$

$$\boxed{\mathbb{E}[X] \approx 3.85}$$

So the cluster experiences **3.85 failures per week on average**.

Importantly:

$$\boxed{\mathbb{E}[X] = 3.85 \neq X = 3.85}$$

The actual number of failures must be an integer:

$$X \in \{0, 1, 2, 3, \ldots\}$$

while 3.85 is the long-run average.

How Variable Is the Number of Failures?

For a binomial random variable:

$$\mathrm{Var}(X) = Np(1-p)$$

Hence:

$$\mathrm{Var}(X) = 10{,}000(0.0003846)(1-0.0003846) \approx 3.84$$

and therefore:

$$\sigma_X = \sqrt{\mathrm{Var}(X)} \approx \boxed{1.96}$$

So, roughly speaking:

$$X \approx 3.85 \pm 1.96$$

The weekly number of failures naturally fluctuates around its expected value.

From AFR to "Failures per Week"

The complete calculation is therefore:

$$\boxed{\begin{aligned} \text{AFR} &= 0.02 \\ p_{\text{week}} &\approx \frac{0.02}{52} \approx 0.0003846 \\ X &\sim \text{Binomial}(10{,}000, 0.0003846) \\ \mathbb{E}[X] &= Np \approx 3.85 \end{aligned}}$$

Hence:

$$\boxed{\text{about 4 disk failures per week, on average}}$$

This is the key scaling effect:

> *A 2% annual failure rate sounds small for one disk, but becomes a regular stream of failures at the scale of 10,000 disks.*

A Useful Consequence

What is the probability that **at least one disk fails** during a given week?

Instead of summing all the probabilities for $X = 1, 2, \ldots$, we can use the complement, i.e., we calculate what is the probability of none fails ($P(X = 0)$), and take the complement:

$$P(X \geq 1) = 1 - P(X = 0)$$

For a binomial random variable:

$$P(X = 0) = (1 - p)^N$$

Therefore:

$$P(X \geq 1) = 1 - (1 - p)^N$$

Substituting our values:

$$P(X \geq 1) = 1 - (1 - 0.0003846)^{10{,}000}$$

$$\boxed{P(X \geq 1) \approx 97.9\%}$$

So, under these simplifying assumptions, **a 10,000-disk cluster has a disk failure in roughly 98% of weeks**.

Putting It Together: A Chunkserver Dies Mid-Read

Walking through a concrete failure scenario end to end:

1. A client is reading a chunk from Chunkserver A; Chunkserver A crashes mid-transfer.
2. The client's read request times out or errors.
3. The client re-queries the Master for alternate replica locations for the same chunk.
4. The client retries the read against a different chunkserver (B or C) — the application-level read succeeds, transparently to the end user.
5. Separately, the Master detects Chunkserver A's missed heartbeats and schedules re-replication of all its chunks elsewhere.

✅ Checkpoint 6 — Failure Handling

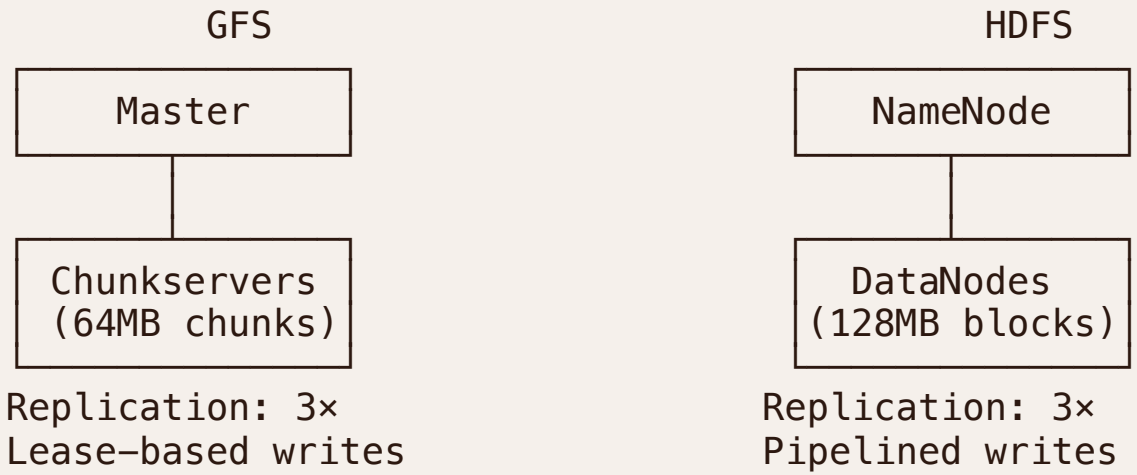
Let's test our understanding:

1. Why must heartbeat timeouts balance detection speed against false positives?
2. Walk through the recovery steps after a DataNode permanently fails.
3. Why are checksums necessary even when replication already provides redundancy?
4. Given a 2% annual disk failure rate, why does a cluster of thousands of disks experience failures on a weekly, not yearly, basis?
5. In the "chunkserver dies mid-read" scenario, which steps are visible to the client, and which happen entirely in the background?

Make sure you understand these mechanisms before wrapping up.

§ 7 — Synthesis & Wrap-up

GFS/HDFS Side-by-Side Architecture Recap



Same core architecture, two implementations — one proprietary, one open-source and industry-standard.

Connecting to the Bigger Picture: Storage Enables Compute

Everything covered in this lecture is, in isolation, "just" a storage system. Its real significance becomes clear once paired with computation:

- The block/chunk abstraction enables **parallel** processing — one task per block.
- Exposed block locations enable **data locality** — schedule computation where data already lives.
- Replication and fault tolerance at the storage layer let the **computation layer** assume reliable storage, simplifying its own design.

Where MapReduce Fits on Top of This Storage Layer

A preview of the next lecture's central idea:

- A MapReduce job reads its input as a set of **blocks** stored in HDFS (or chunks in GFS).
- The scheduler assigns each **Map task** to run on (or near) the node that already holds its input block.
- Output is written back into the same distributed file system, replicated the same way as any other data.

MapReduce does not reinvent storage — it is built directly on top of everything covered in this lecture.

Limits of This Architecture

The GFS/HDFS model, while foundational, has known limitations that motivate later developments in the course:

- Optimized for large files and sequential access — poor fit for huge numbers of small files or random access workloads.
- Single metadata node (even with HA/Federation) imposes namespace scalability limits.
- Write-once-read-many semantics are restrictive for workloads requiring frequent in-place updates.

Later lectures (Spark, streaming, lakehouse architectures) address several of these limitations directly.

Summary: The Three Pillars of Scale-Out Storage

This lecture's content can be distilled into three structural pillars:

1. **Horizontal scaling over commodity hardware:** many cheap, unreliable machines instead of few expensive, reliable ones.
2. **Block-based storage with replication:** fixed-size chunks/blocks, replicated across failure domains, enable both parallelism and durability.
3. **Separated, fault-tolerant metadata management:** a single coordinating node (Master/NameNode) tracks locations; bulk data flows directly between clients and data-storing nodes.

Key Terms Glossary

Term	Meaning
Chunk / Block	Fixed-size unit of file storage (64 MB / 128 MB)
Master / NameNode	Metadata-managing coordinator node
Chunkserver / DataNode	Node storing and serving actual data
Replication factor	Number of copies maintained per chunk/block
Heartbeat	Periodic liveness signal from data node to coordinator
Data locality	Scheduling computation near the data it needs

Preview: Lecture 1.2 — The MapReduce Execution Model

In the next lecture, we build the computation layer on top of this storage foundation:

- **The MapReduce abstraction:** Map, Shuffle, Sort, and Reduce phases.
- **Scheduling with data locality:** placing Map tasks near their input blocks.
- **Fault tolerance during computation:** stragglers, speculative execution, task re-execution.

> *Understanding distributed storage is the key to understanding distributed computation. See you in Lecture 1.2!*

Lecture 1.1 Complete

Recommended Reading

- Ghemawat, S. et al. (2003). *The Google File System*. SOSP.
- Course reference materials on HDFS architecture

Contact & Q&A

- **Lecturer:** Gabriele Tolomei
- **Email:** tolomei@di.uniroma1.it
- **Office Hours:** Drop me a message, and we'll arrange a time to meet, either in person or online! 🙂

Thank you for your attention!