Distributed File Systems

Managing Big Data Storage at Scale

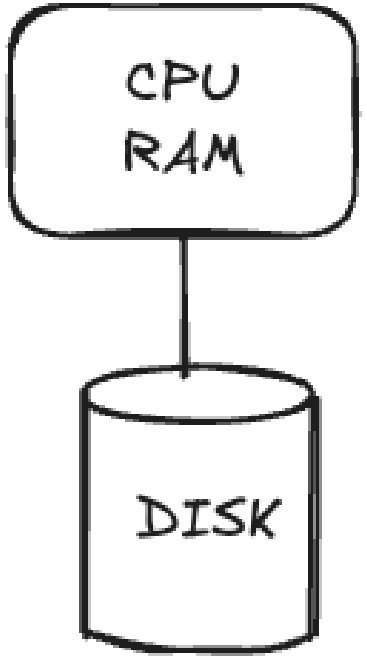
Big Data Storage Requirements

- **Volume**: Large capacity + low cost per byte
- **Velocity**: Efficient streaming reads/writes ("batch" vs "point" access)
- **Variety**: Store unstructured/semi-structured data
- **Concurrency**: Coordinate concurrent access without expensive transactions

Bottom line: We need a file system (FS) that can handle these requirements

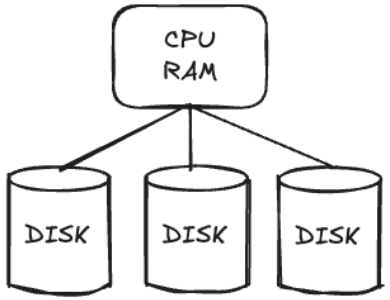
Local File System Architecture

Single machine with CPU/RAM and attached storage



Can we just add more disks?

Scaling Local FS

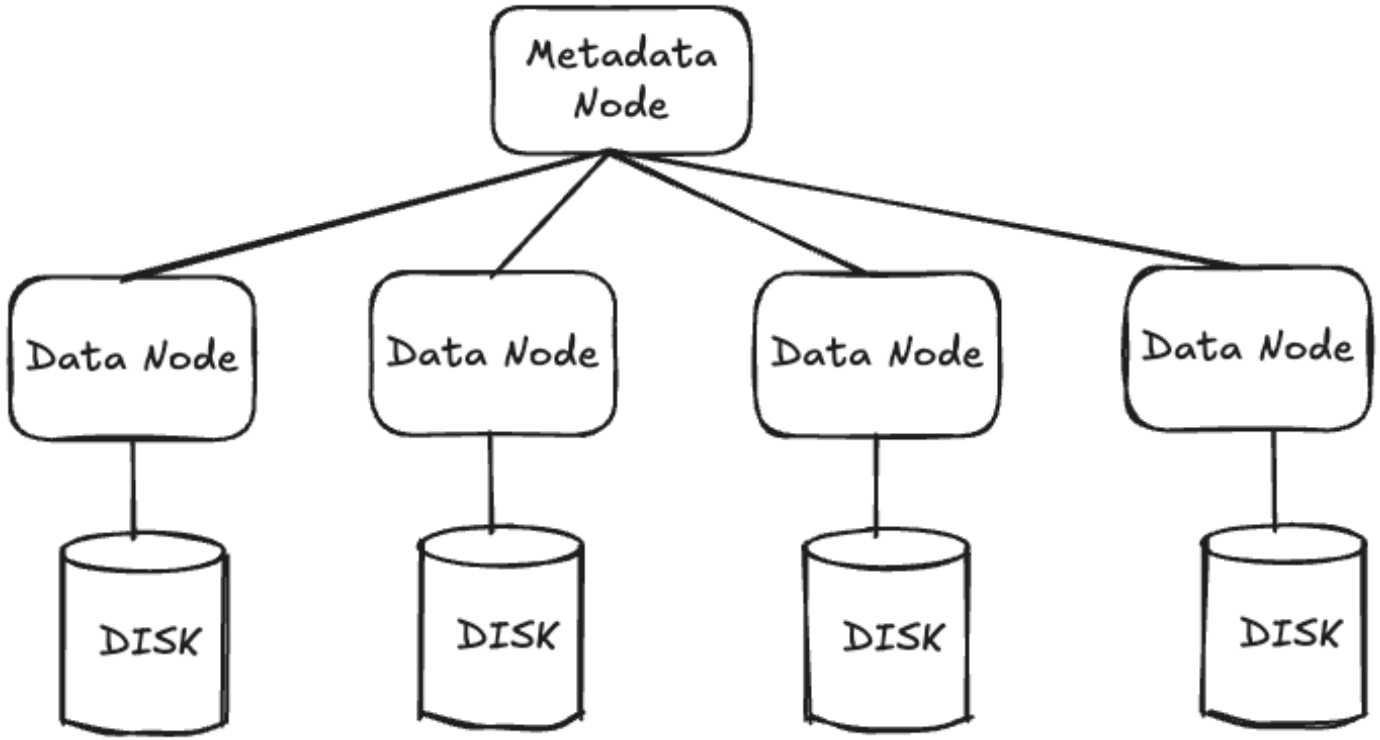


Problems:

1. Limit on storage one machine can drive
2. Limit on request traffic one machine can handle:
    - CPU for request processing
    - RAM for disk buffers
    - Network interface for data transmission

❌ **Local FS cannot handle Big Data Volume and Velocity**

Distributed File System Architecture



- **Data nodes**: Store actual file content
- **Metadata node**: Tracks which data is stored where

Addressing the Metadata Bottleneck

Question: Have we just moved the bottleneck from data node → metadata node?

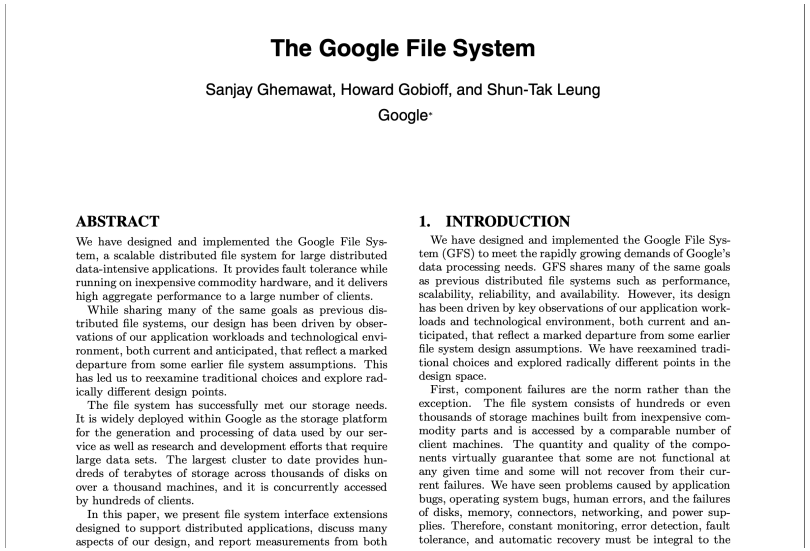
Answer: No! Here's why:

1. **Infrequent metadata access**
    - Clients consult metadata node only when opening files
    - Clients cache metadata locally
2. **Separate control and data paths**
    - Metadata node: Control path only
    - Data nodes: Direct data transfer with clients
    - Metadata node NOT in the data path!

GFS/HDFS: The Pioneers

Historical Context:

- **GFS** (Google File System): 2003
  - Now replaced by Colossus File System (CFS)
- **HDFS** (Hadoop Distributed File System)
  - Open source version of GFS



The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung
Google*

ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points.

The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our service as well as research and development efforts that require large data sets. The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds of clients.

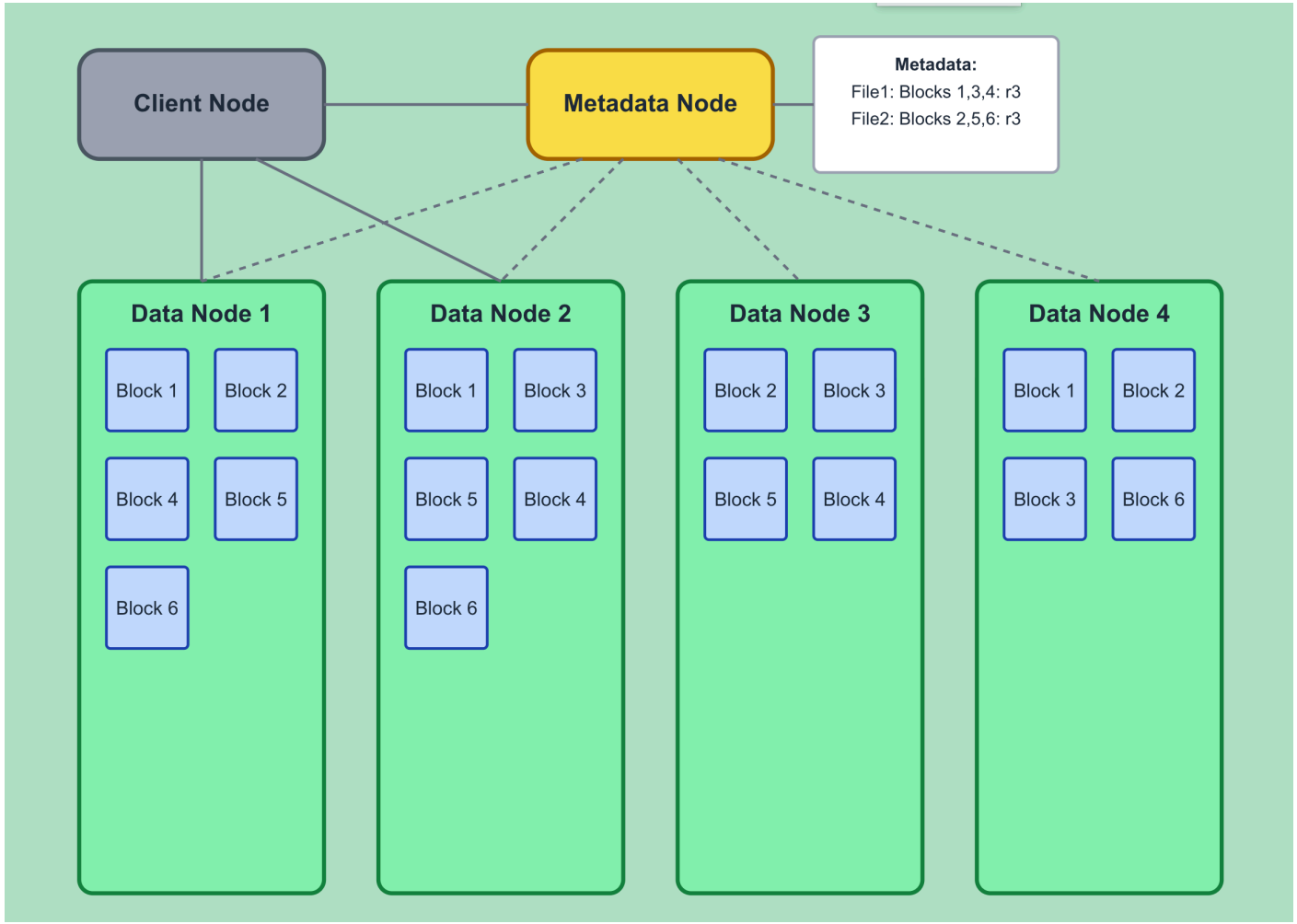
In this paper, we present file system interface extensions designed to support distributed applications, discuss many aspects of our design, and report measurements from both

1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at any given time and some will not recover from their current failures. We have seen problems caused by application bugs, operating system bugs, human errors, and the failures of disks, memory, connectors, networking, and power supplies. Therefore, constant monitoring, error detection, fault tolerance, and automatic recovery must be integral to the

GFS/HDFS Architecture Overview



Design Goal 1: High Volume

Strategies:

1. **Multiple data nodes** → increased capacity
2. **Commodity hardware** → reduced cost
3. **Redundancy** → handle failures
    - Commodity hardware more likely to fail
    - **Replication (R=3)**: Simple but costly
    - **Erasure coding (5,3)**: Same fault-tolerance (2 losses) at ~half the storage cost
4. **"Repair" plane** → re-replicate on failures

Design Goal 2: High Velocity (Part 1)

Large block size: 64 MiB (vs 4 KiB in local FS)

Benefits:

1. **Fewer blocks per file**
    - Reduced metadata (can stay in memory!)
    - Lower metadata node traffic
2. **Higher effective throughput**
    - Amortize seek time over larger reads

Block Size Impact on Throughput

Example: 1 ms seek + 1 KiB/ms disk read speed

Block Size	Time to Read 1 Block	Effective Throughput
4 KiB	(1 + 4) ms = 5 ms	80%
64 MiB	(1 + 2^{16}) ms ≈ 65 sec	~100%

Key insight: With large blocks, seek time becomes negligible!

Design Goal 2: High Velocity (Part 2)

Additional velocity benefits:

3. **Reduced wasted storage**
    - Small file in 64 MiB block seems wasteful…
    - But GFS/HDFS designed for *large files*
    - Wasted space is minimal in practice
4. **Redundancy enables parallel reads**
    - Multiple replicas → read from multiple nodes
    - Further increases effective throughput

Design Goal 3: High Concurrency

Write-Once-Read-Many (WORM) model

- Simplifies concurrency management
- No need for complex locking protocols

Novel append semantics:

- **Traditional FS**: Specify file offset → concurrent appends not serializable
- **GFS/HDFS**: No offset specification → metadata node serializes appends
- Enables concurrent appends without conflicts!

Key Takeaways

1. **Local FS can't scale** for Big Data (Volume + Velocity limits)
2. **Distributed FS separates concerns**:
    - Metadata management vs. data storage
    - Control path vs. data path
3. **GFS/HDFS design optimizations:**
    - Large blocks for batch access
    - Replication/erasure coding for fault tolerance
    - WORM + novel appends for concurrency
4. **Design for common case**: Large files, sequential access, append-only

Further Reading

📄 **Recommended Paper:**

The Google File System (GFS, 2003)

- Foundational distributed systems paper
- Clear explanation of design decisions
- Real-world performance data
- Great example of systems thinking