Introduction

- A **file system** is a method and data structure that an operating system (OS) uses to manage, store, organize, and retrieve files on a storage device such as a hard drive, SSD, USB drive, or memory card. It defines how data is stored and accessed on the storage medium.
- **File Organization:**
- Storage Management
- **File Naming and Identification:**
- Access Control:
- File Operations
- **Types of File Systems:**
- **Disk-based File Systems:** Used in hard drives and SSDs.
    - **FAT32, exFAT, NTFS** (Windows)
    - **EXT3, EXT4, XFS** (Linux)
    - **APFS, HFS+** (Mac)
- **Network File Systems:** Enable remote access.
    - **NFS (Network File System)**
- **Specialized File Systems:** Designed for specific use cases.
    - **ZFS (high reliability, used in servers)**
    - **ReFS (Resilient File System by Microsoft)**
- **Virtual File Systems**
    - **procfs (/proc in Linux):**

GFS

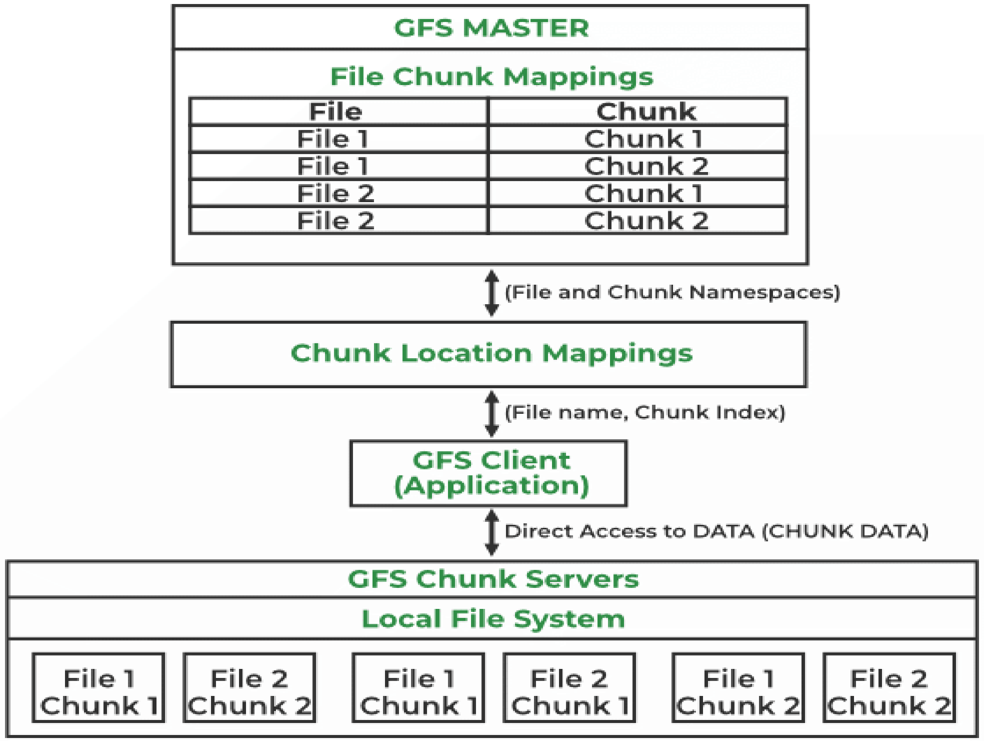
- Google File System (GFS) is designed to meet the rapidly growing **demands of Google’s data processing needs.**
- GFS shares many of the same goals as previous distributed file systems such as **performance, scalability, reliability, and availability.**
- The Google File System (GFS) is **a proprietary DFS** developed by Google. It is designed to provide efficient, reliable access to data using large clusters of commodity hardware.
- A GFS cluster consists of a **single *master* and multiple *chunk servers*** and is accessed by multiple *clients*.
- Each of these is typically a **commodity Linux machine** running a user-level server process. It is easy to run both a chunk server and a client on the same machine
- Files are divided into fixed-size ***chunks*.** Each chunk is identified by an immutable and globally unique **64 bit *chunkhandle*** assigned by the master at the time of chunk creation.
- Chunk servers store on local disks as Linux files and read or write chunk data specified by a chunk handle and byte range. **For reliability, each chunk is replicated on multiple servers.**
- By default chunks **replicated three times.**

Need for GFS

- Handling Large-Scale Data
- High Fault Tolerance
- Efficient Storage for Large Files
- Scalability for Distributed Systems
- Efficient Metadata Management
- High Throughput for Big Data Processing
- Optimized for Write-Once, Read-Many Workloads

- **Handling Large-Scale Data :**Google processes petabytes of data daily (e.g., web crawling, indexing).
- Traditional file systems couldn't efficiently handle such massive datasets.
- **2️⃣High Fault Tolerance:** Hardware failures (disk crashes, node failures) are common in large clusters.
- GFS ensures **automatic fault recovery** and data redundancy.
- **3️⃣Efficient Storage for Large Files:**GFS is optimized for **very large files (GBs to TBs)** instead of many small files.
- Data is stored in **64 MB chunks**, reducing metadata overhead.
- **4️⃣Scalability for Distributed Systems:**Traditional file systems struggle to scale across thousands of machines.
- GFS supports **horizontal scaling** across thousands of servers.
- **5️⃣Optimized for Write-Once, Read-Many Workloads:**Many Google applications (like web search indexing) involve **appending data rather than modifying it**.
- GFS is optimized for **append-heavy workloads** rather than frequent modifications.
- **6️⃣Efficient Metadata Management:**Unlike traditional file systems, GFS uses a **single Master Node** for metadata storage and lookup.
- This reduces metadata lookup latency.
- **7️⃣High Throughput for Big Data Processing:**GFS allows **parallel processing** by splitting files across multiple servers.
- This speeds up tasks like **MapReduce**, which Google heavily relies on.

Google File System



GFS Architecture

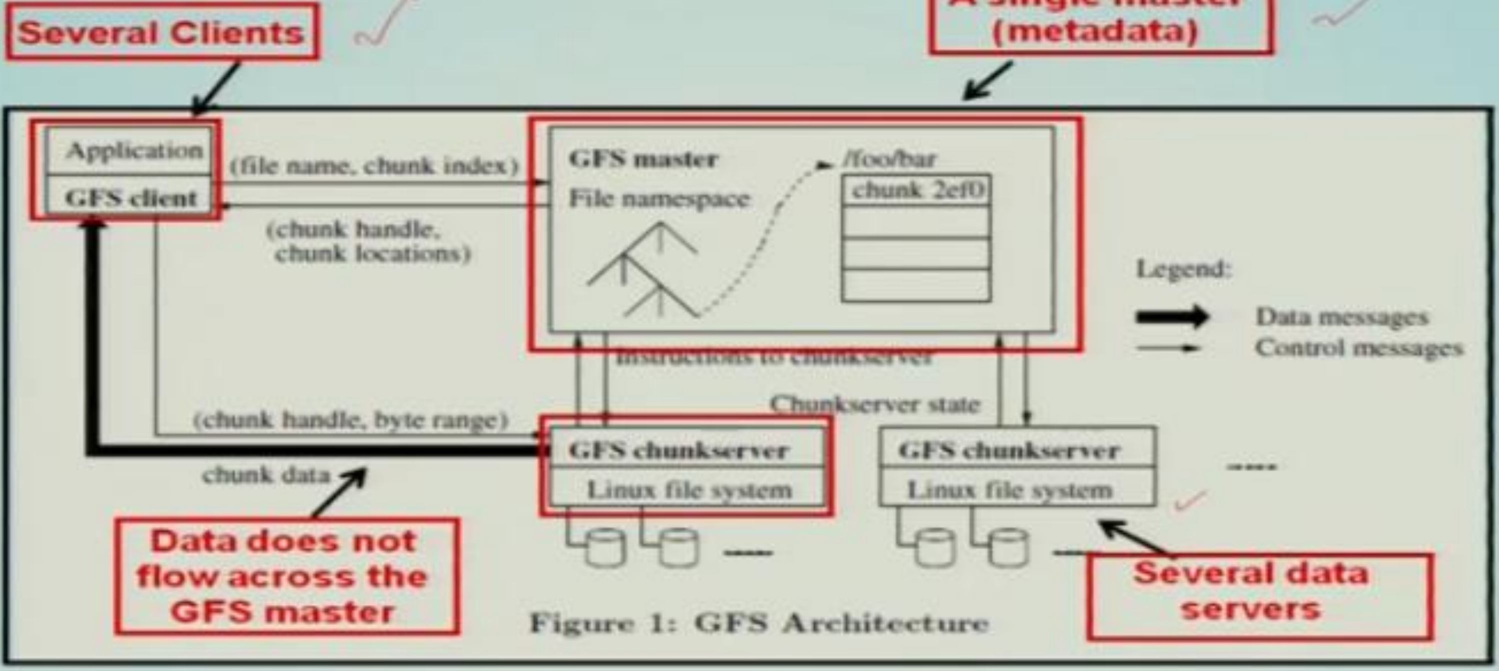


Figure 1: GFS Architecture

GFS

- **Master:**
- The master maintains all file system metadata. This includes the namespace, access control information, the mapping from files to chunks, and the current locations of chunks.
- It also controls system-wide activities such as chunk lease management, garbage collection of orphaned chunks, and chunk migration between chunkservers.
- The master periodically communicates with each chunkserver in ***HeartBeat*** messages to give it instructions and collect its state.
- GFS client code linked into each application implements the file system API and communicates with the master and chunkservers to read or write data on behalf of the application.
- Clients interact with the master for metadata operations ,but all data-bearing communication goes directly to the chunkservers.
- Neither the client nor the chunkserver caches file data.
- most applications stream through huge files or have working sets too large to be cached.
- Chunkservers need not cache file data because chunks are stored as local files

Chunk Size

- Chunks size is one of the key design parameters.
- **Default size 64 MB**, which is much larger than typical file system blocks sizes.
- Each chunk replica is stored as a plain Linux file on a chunk server.
- **Advantages of large chick size:**
- It reduces clients' need to interact with the master because reads and writes on the same chunk require only one initial request to the master for chunk location information.
- since on a large chunk, a client is more likely to perform many operations on a given chunk, it can reduce network overhead
- it reduces the size of the metadata stored on the master.

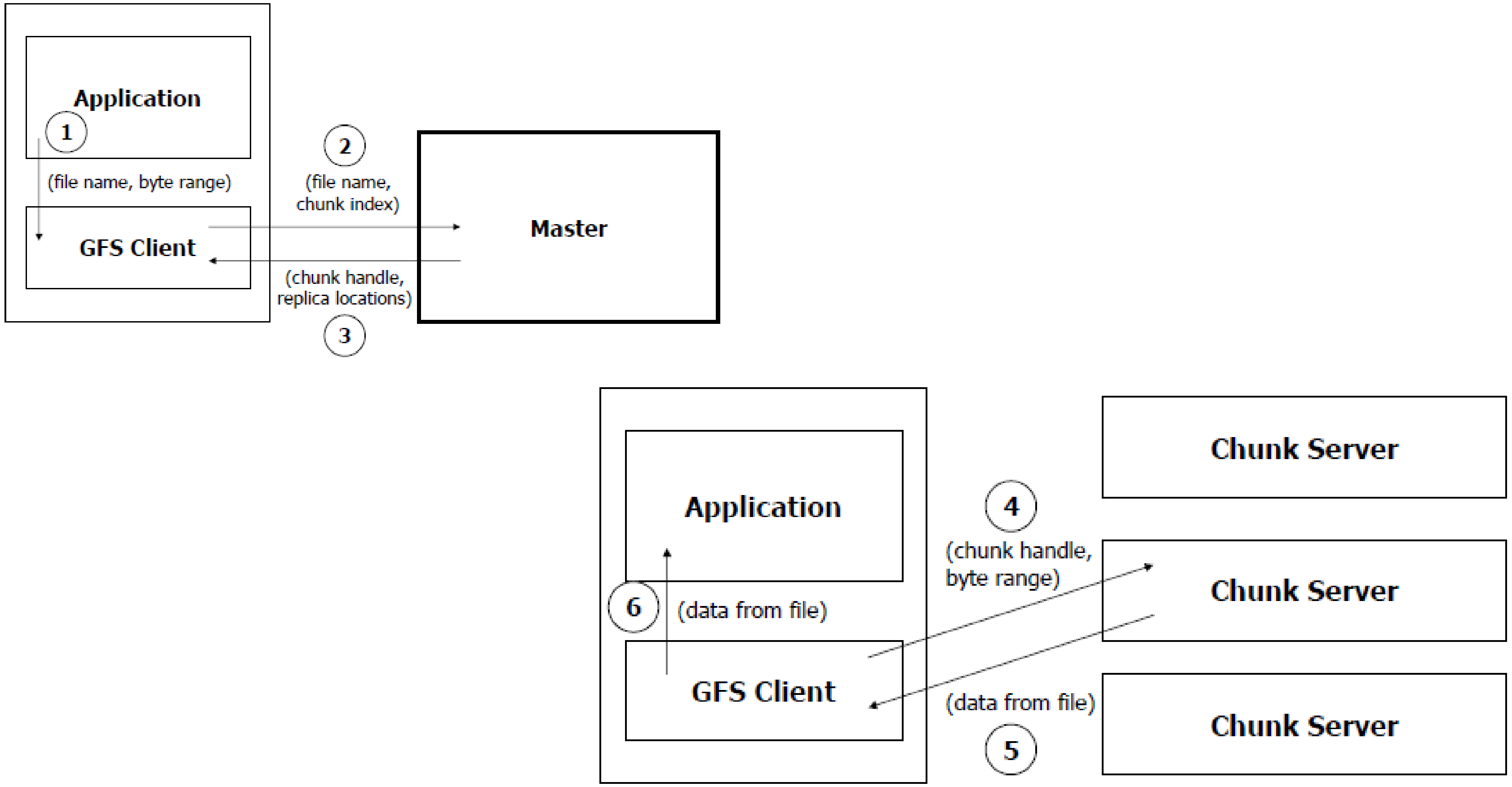
Metadata

- The master stores three major types of metadata: the file and chunknamespaces, the mapping from files to chunks,and the locations of each chunk’s replicas.
- All metadata is kept in the master’s memory.
- The first two types (namespaces and file-to-chunkma pping) are also kept persistent whereas chunk location information is not stored persistently.
- Instead, it asks each chunkserver about its chunks at master startup and whenever a chunkserver joins the cluster.

□ File Metadata Maintained by the Master Node

Metadata Type	Description
Namespace (File & Directory Structure)	The hierarchical structure of files and directories (similar to a filesystem tree).
File-to-Chunks Mapping	Tracks which chunks belong to each file.
Chunk Locations (Chunk-to-Chunkserver Mapping)	Stores the locations of chunk replicas across different Chunkservers.
Chunk Version Numbers	Helps detect stale or outdated chunks.
Chunk Lease Information	Identifies the Primary Chunkserver for each chunk (important for consistency).
Access Control & Permissions	Manages user permissions for file access.
Operation Logs (Metadata Change History)	Logs all metadata changes for fault recovery and replication.

Read Algorithm

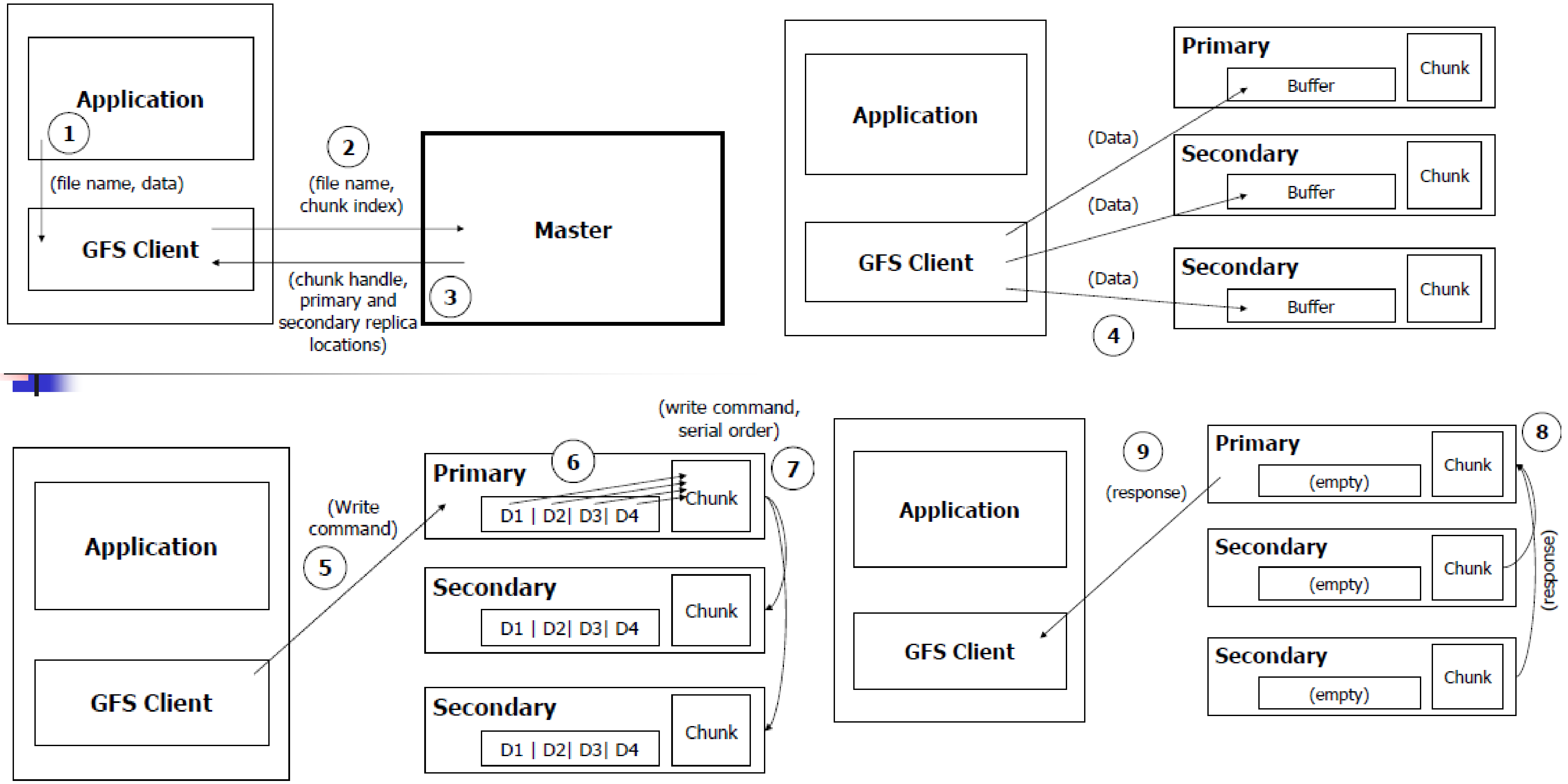


Read Algorithm

1. Application originates the read request.
2. GFS client translates the request from (filename, byte range) -> (filename, chunk index), and sends it to master.
3. Master responds with chunk handle (unique ID of the chunk) and replica locations (i.e. chunkservers where the replicas are stored).
4. Client picks a location and sends the (chunk handle, byte range) request to that location.
5. Chunkserver sends requested data to the client.
6. Client forwards the data to the application.

- **Client Caches Metadata**
- To avoid repeated requests to the Master, the client **caches metadata** for future reads.
- This improves performance by reducing **Master Node bottlenecks**.

Write Algorithm



Write Algorithm

1. Application originates write request.

2. GFS client translates request from (filename, data) -> (filename, chunk index/byterange), and sends it to master.

The **Master Node** checks which **chunk** corresponds to this byte range. If the required chunk **does not exist**, the Master **allocates a new chunk** and assigns **Chunkservers** to store it.

3. Master responds with chunk handle and (primary + secondary) replica locations.

4. Client pushes write data to all locations. Data is stored in chunkservers' internal buffers.

5. Client sends write command to primary.

6. Primary determines serial order for data instances stored in its buffer and writes the instances in that order to the chunk.- determines the **write order** for consistency

7. Primary sends serial order to the secondaries and tells them to perform the write.

8. Secondaries respond to the primary.

9. Primary responds back to client.

- Note: If write fails at one of chunkservers, client is informed and retries the write.

Write Algorithm

- **Error Handling & Recovery (If Needed)**
- If a **Secondary Chunkserver fails**, the write still **succeeds on other replicas**.
- The **Master Node detects failures** and may reassign chunk replicas.
- If the **Primary Chunkserver fails**, the **Master reassigns** a new Primary and retries the operation.

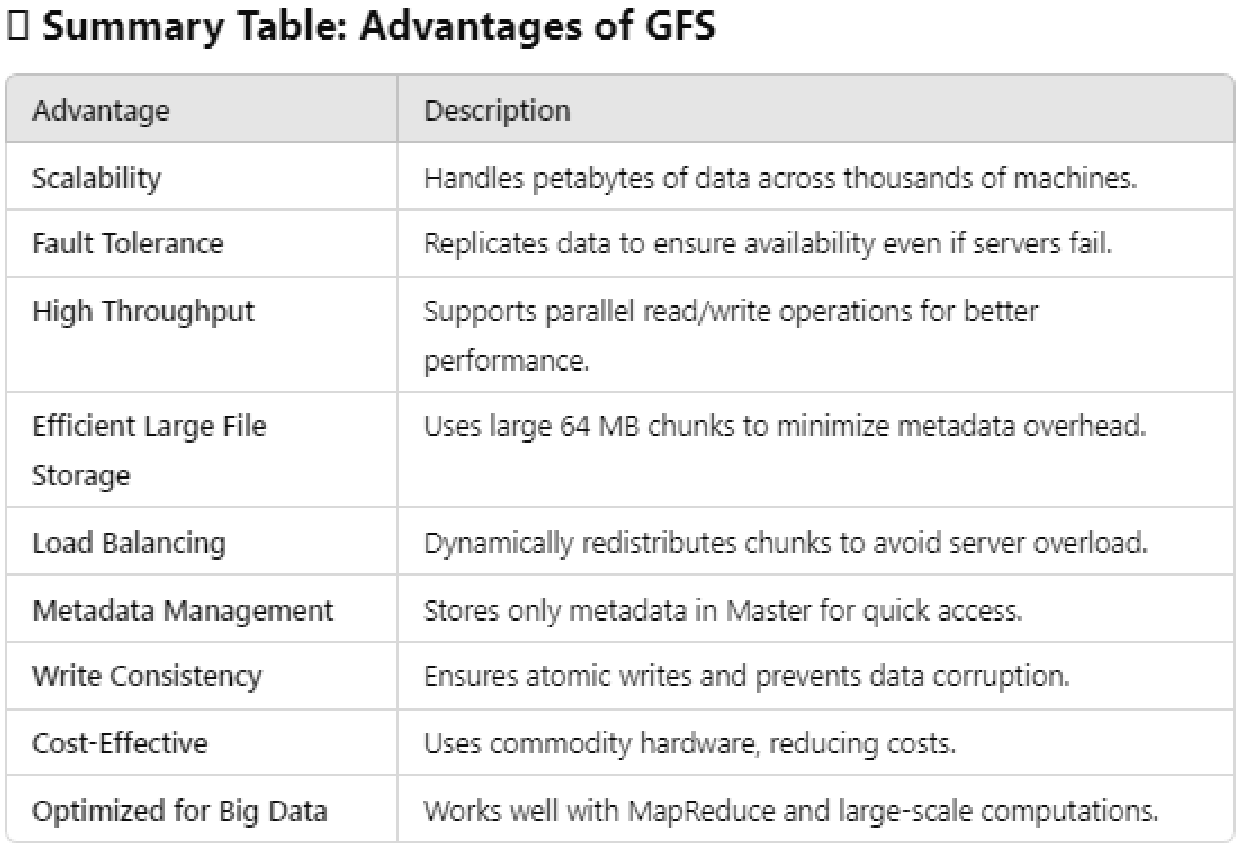
Write Algorithm:Example

- **Example Scenario**
  - A client wants to write **"Hello GFS"** to **File.txt** at **offset 4096**.
- **Client → Master:** "Where is chunk for File.txt, offset 4096?“
  Master → Client: "Chunk ID = CHK_123, Primary = Server A, Replicas = Server B, C.“
  Client → Chunkservers (A, B, C): "Here’s the data!" (Buffered)
  Primary Chunkserver A: Assigns **write order** and sends a commit request.
  Secondary Chunkservers B, C: Write the data and confirm success.
  Primary → Client: "Write successful!"

Advantages of GFS

- High Scalability
- Fault Tolerance & Data Reliability
- High Performance & Throughput
- Efficient Storage of Large Files: Supports **multi-terabyte files** efficiently., Designed for **sequential reads/writes**, making it ideal for big data processing.
- Automatic Load Balancing
- Simplified Metadata Management: The **Master stores only metadata**, not actual file data. Write Consistency & Atomic Operations

Summary



□ Summary Table: Advantages of GFS

Advantage	Description
Scalability	Handles petabytes of data across thousands of machines.
Fault Tolerance	Replicates data to ensure availability even if servers fail.
High Throughput	Supports parallel read/write operations for better performance.
Efficient Large File Storage	Uses large 64 MB chunks to minimize metadata overhead.
Load Balancing	Dynamically redistributes chunks to avoid server overload.
Metadata Management	Stores only metadata in Master for quick access.
Write Consistency	Ensures atomic writes and prevents data corruption.
Cost-Effective	Uses commodity hardware, reducing costs.
Optimized for Big Data	Works well with MapReduce and large-scale computations.

Disadvantages of GFS

- Not the best fit for small files.- **64 MB chunk size** is inefficient for small files, leading to **wasted storage**.
- Single Point of Failure: Master may act as a bottleneck.
- unable to type at random.
- Suitable for procedures or data that are written once and only read (appended) later.
- High Replication Overhead- No built-in **erasure coding**, which could save space.
- No Built-in Support for Windows
- High Memory Consumption on Master Node
- Chunkserver Failures Cause Performance Drops

□ Summary Table: Limitations of GFS

Limitation	Description	Impact
Master Node Dependency	Single Master is a bottleneck and failure point	Can slow down file operations
Inefficient for Small Files	Large chunks cause overhead for small files	High memory usage & wasted storage
Slow Random Writes	Optimized for sequential writes, not updates	Not ideal for real-time applications
High Storage Overhead	Uses multiple replicas for fault tolerance	High storage & network bandwidth cost
Limited Security	Lacks encryption and fine-grained access control	Not ideal for sensitive data
No Windows Support	Designed for Linux-based clusters	Requires extra development effort
High Master Memory Usage	Metadata stored in RAM limits scalability	Performance drops with large metadata
Chunkserver Failure Impact	Failure triggers re-replication, affecting network	Can cause latency and network congestion