This paper details the design and implementation of the Google File System (GFS), a scalable distributed file system created to handle Google's massive data processing needs. It provides fault tolerance on inexpensive commodity hardware and delivers high aggregate performance.

1. Introduction GFS was designed to meet performance, scalability, reliability, and availability goals, driven by observations of Google's specific application workloads and technological environment, which differ significantly from earlier file system assumptions[cite: 3, 9, 10]. Key departures include:

- **Component failures are the norm:** The system uses thousands of inexpensive commodity machines, making failures frequent. Constant monitoring, error detection, fault tolerance, and automatic recovery are integral[cite: 12, 13, 16].
- **Files are huge:** Multi-GB files are common, containing many application objects (e.g., web documents), necessitating reevaluation of I/O operation and block sizes[cite: 17, 19, 20].
- **Append-heavy workloads:** Most files are mutated by appending new data rather than overwriting; they are then read often sequentially. This makes appending the focus for performance optimization and atomicity, reducing the utility of client-side data caching[cite: 21, 22, 26].
- **Co-design of applications and API:** Relaxing GFS's consistency model simplified the file system without overly burdening applications. An atomic append operation was introduced for concurrent client appends without extra synchronization[cite: 27, 28, 29].

2. Design Overview

2.1 Assumptions

- The system is built from many inexpensive commodity components that fail often; it must routinely detect, tolerate, and recover from these failures[cite: 34, 35].
- It stores a modest number of very large files (typically 100 MB or larger, with multi-GB files being common)[cite: 36, 37, 38]. Small files are supported but not optimized for[cite: 39].
- Workloads consist primarily of large streaming reads and small random reads. Writes are mainly large, sequential appends; files are seldom modified after being written[cite: 40, 41, 45, 46].
- Efficient, well-defined semantics for multiple clients concurrently appending to the same file are essential, requiring atomicity with minimal synchronization[cite: 48, 51].
- High sustained bandwidth is prioritized over low latency for individual operations[cite: 52, 53].

2.2 Interface GFS provides a familiar file system interface (hierarchical directories, pathnames) but not a standard API like POSIX[cite: 54, 55]. It supports usual operations (create, delete, open, close, read, write) plus two new operations[cite: 56, 57]:

- **Snapshot:** Creates a low-cost copy of a file or directory tree[cite: 57].
- **Record append:** Allows multiple clients to append data to the same file concurrently, guaranteeing atomicity for each client's append[cite: 58].

2.3 Architecture (Figure 1) A GFS cluster has a single **master**, multiple **chunkservers**, and is accessed by multiple **clients**[cite: 62].

- **Files:** Divided into fixed-size **chunks** (default 64 MB), each identified by an immutable, globally unique 64-bit chunk handle assigned by the master[cite: 64, 93].
- **Chunkservers:** Store chunks on local disks as Linux files and handle read/write requests for data specified by a chunk handle and byte range[cite: 65]. Each chunk is replicated on multiple

chunkservers (default 3) for reliability[cite: 66, 67].
- **Master:** Maintains all file system metadata (namespace, access control information, file-to-chunk mapping, current chunk locations)[cite: 68]. It controls system-wide activities like chunk lease management, garbage collection, and chunk migration[cite: 69]. Communicates with chunkservers via periodic HeartBeat messages[cite: 70].
- **Client:** Client code linked into applications implements the GFS API. It interacts with the master for metadata operations but communicates directly with chunkservers for all data-bearing operations[cite: 71, 72].
- **Caching:** GFS clients do not cache file data due to large file sizes and streaming access patterns; they do cache metadata[cite: 74, 76]. Chunkservers also do not cache file data, relying on the underlying Linux buffer cache[cite: 76].

2.4 Single Master A single master simplifies design and allows for global knowledge in chunk placement and replication decisions[cite: 77, 78]. To avoid being a bottleneck, clients interact with the master only for metadata (e.g., chunk locations), caching this information and then communicating directly with chunkservers for data[cite: 79, 80, 81, 82]. For a read, the client gets chunk locations from the master, caches them, and then requests data from the nearest replica[cite: 83, 85, 86, 87, 88].

2.5 Chunk Size The chunk size is a key parameter, set to 64 MB[cite: 93]. Replicas are stored as plain Linux files using lazy space allocation to avoid internal fragmentation[cite: 94, 95].

- **Advantages:** Reduces client-master interaction for large sequential workloads; enables persistent TCP connections to chunkservers, reducing network overhead; reduces metadata size on the master, allowing it to be kept in memory[cite: 96, 97, 99, 100].
- **Disadvantages:** Can lead to hot spots on chunkservers storing small, frequently accessed files, though this is mitigated by higher replication and staggering application start times[cite: 104, 105, 108].

2.6 Metadata The master stores three types of metadata: file and chunk namespaces, file-to-chunk mapping, and locations of each chunk's replicas[cite: 110].

- **In-Memory Data Structures (2.6.1):** All metadata is kept in the master's memory for fast operations and efficient background scanning (for garbage collection, re-replication, chunk migration)[cite: 111, 115, 116, 117]. The memory required is modest (less than 64 bytes per 64 MB chunk and per file for names using prefix compression)[cite: 120, 122].
- **Chunk Locations (2.6.2):** The master does not store chunk locations persistently. Instead, it polls chunkservers at startup and when they join, controlling all placement and monitoring status via HeartBeats[cite: 113, 114, 124, 125]. This simplifies handling chunkserver joins, leaves, and failures[cite: 127].
- **Operation Log (2.6.3):** Critical metadata changes (namespaces, file-to-chunk mappings) are logged persistently on the master's local disk and replicated on remote machines[cite: 111, 131]. The log serves as a logical timeline defining the order of operations[cite: 132]. Changes are visible to clients only after log records are durably flushed[cite: 134, 136]. The master batches log records to reduce flushing impact[cite: 137]. Master recovery involves replaying the operation log. To minimize startup time, the master checkpoints its state when the log grows large, allowing recovery by loading the latest checkpoint and replaying only subsequent log records[cite: 138, 139, 140, 141]. Checkpoints are created efficiently without delaying incoming mutations[cite: 144, 145].

2.7 Consistency Model GFS employs a relaxed consistency model.

- **Guarantees (2.7.1, Table 1):**
    - File namespace mutations (e.g., file creation) are atomic, handled exclusively by the master using locking and the operation log[cite: 154, 155, 156].
    - The state of a file region after a data mutation depends on the mutation type (write or record append), success/failure, and concurrency[cite: 157]:
        - **Consistent:** All clients always see the same data[cite: 158].
        - **Defined:** Consistent, and clients see the entirety of what the mutation wrote[cite: 159].
        - Successful non-concurrent mutation: Region is defined[cite: 160].
        - Concurrent successful mutations: Region is consistent but undefined (may contain mingled fragments)[cite: 161].
        - Failed mutation: Region is inconsistent (and undefined)[cite: 162].
    - **Record Appends:** Data is appended atomically at least once at a GFS-chosen offset, which is returned to the client[cite: 166, 167]. GFS may insert padding or record duplicates between successful appends (these intervening regions are inconsistent)[cite: 168, 169].
    - After successful mutations, the mutated region is defined (reflects the last mutation) by applying mutations in the same order on all replicas and using chunk version numbers to detect stale replicas (which are not used and eventually garbage collected)[cite: 170, 171, 172].
    - Clients might read from a stale replica due to cached locations, but this window is limited by cache timeout and file re-open[cite: 173, 174]. For append-only files, this usually results in a premature end-of-chunk[cite: 175].
    - Data loss occurs only if all replicas of a chunk are lost before GFS can re-replicate (typically within minutes); applications receive errors, not corrupted data[cite: 180, 181].
- **Implications for Applications (2.7.2):** Applications adapt using techniques like relying on appends, checkpointing, and writing self-validating/identifying records[cite: 182]. Writers often rename files atomically after completion or use checkpoints; readers process data up to the last valid checkpoint[cite: 185, 186]. For concurrent appends, readers use checksums and unique IDs within records to handle padding and potential duplicates[cite: 192, 193, 194].

3. System Interactions The design minimizes master involvement.

- **3.1 Leases and Mutation Order:** Mutations (writes, appends) are performed on all replicas[cite: 199, 200]. The master grants a **lease** to one replica (the **primary**), which determines a serial order for all mutations to that chunk. All replicas follow this order[cite: 200, 201, 202]. Leases have a timeout (e.g., 60s) but are extended via HeartBeat messages if the chunk is active[cite: 205, 206].
    - **Write Control Flow (Figure 2):**
        1. Client asks master for primary and replica locations. Master grants lease if none.
        2. Master replies; client caches this.
        3. Client pushes data to all replicas (any order); replicas buffer it. Data flow is decoupled from control flow for network efficiency.
        4. After all replicas ACK data receipt, client sends write request to primary. Primary assigns serial numbers to mutations and applies locally.
        5. Primary forwards request to secondaries; they apply in the same serial order.
        6. Secondaries ACK primary.
        7. Primary replies to client. Errors are reported; client retries failed mutations.
- **3.2 Data Flow:** Data is pushed linearly along a chain of chunkservers in a pipelined fashion over TCP to fully utilize network bandwidth and minimize latency[cite: 234, 235, 236, 245]. Each machine forwards data to the "closest" machine in the network topology that hasn't received it[cite: 239].

- **3.3 Atomic Record Appends:** GFS appends client data to a file at least once atomically at an offset of GFS's choosing, returning the offset[cite: 252, 256]. This is useful for multi-producer/single-consumer queues[cite: 261]. The primary checks if the append fits the current chunk; if not, it pads the chunk and instructs the client to retry on the next chunk[cite: 264, 265]. If it fits, the primary appends, tells secondaries the offset, and replies success[cite: 266]. Retries can lead to duplicates on different replicas, but data is written at least once atomically[cite: 267, 268, 270].
- **3.4 Snapshot:** Creates a nearly instantaneous copy of a file/directory tree using copy-on-write[cite: 275, 277]. The master revokes leases on chunks in the source, logs the operation, then duplicates metadata in memory[cite: 278, 281, 282, 283]. On the first client write to a shared chunk post-snapshot, the master directs chunkservers to create a new copy of that chunk (locally, not over network) before granting a lease on the new chunk copy[cite: 284, 285, 286, 287, 288].

4. Master Operation The master manages namespace operations, chunk replicas (placement decisions, new chunk creation), and system-wide activities (replication, load balancing, garbage collection)[cite: 289].

- **4.1 Namespace Management and Locking:** Allows multiple master operations concurrently using read-write locks over namespace regions[cite: 290, 292]. The namespace is a lookup table mapping full pathnames to metadata, represented efficiently in memory with prefix compression[cite: 295, 296]. Locks are acquired on directory paths and the final file/directory name to ensure proper serialization (e.g., preventing file creation during a snapshot of its parent directory)[cite: 299, 300, 303]. This allows concurrent mutations in the same directory[cite: 307]. Locks are acquired in a consistent total order to prevent deadlocks[cite: 313].
- **4.2 Replica Placement:** Policy aims to maximize data reliability, availability, and network bandwidth[cite: 320]. Spreads replicas across machines *and* across different racks to survive rack failures and utilize aggregate rack bandwidth[cite: 321, 322, 323].
- **4.3 Creation, Re-replication, Rebalancing:**
  - **Creation:** Master places new empty replicas on chunkservers with below-average disk utilization, limits recent creations on any single server (as creation implies imminent write traffic), and spreads replicas across racks[cite: 326, 327, 329, 330].
  - **Re-replication:** Master re-replicates chunks when available replicas fall below a goal[cite: 331]. Priority is given based on replica deficit, liveness of the file, and whether a chunk is blocking client progress[cite: 333, 334, 335, 337]. Master instructs a chunkserver to copy data from an existing valid replica, following similar placement goals[cite: 338, 339]. Cloning traffic is limited[cite: 340].
  - **Rebalancing:** Master periodically moves replicas for better disk space and load balancing, gradually filling new chunkservers[cite: 342, 343]. Prefers to remove replicas from chunkservers with below-average free space[cite: 346].
- **4.4 Garbage Collection:** Physical storage is reclaimed lazily after file deletion[cite: 347, 348].
  - **Mechanism:** When a file is deleted, the master logs it and renames the file to a hidden name including a deletion timestamp[cite: 350, 351]. Master's regular scan removes hidden files older than a configurable period (e.g., 3 days); until then, files can be undeleted[cite: 352, 353]. Removing the hidden file erases its in-memory metadata, orphaning its chunks[cite: 354]. A separate scan identifies orphaned chunks; their metadata is erased[cite: 355]. Chunkservers report their chunks via HeartBeat; master informs them of chunks no longer in its metadata, and chunkservers delete these replicas[cite: 356, 357].
  - **Discussion:** This is simpler and more reliable than eager deletion in a distributed system with failures[cite: 363]. It handles lost messages and component failures uniformly[cite: 366]. Cost is

amortized as part of background master activities[cite: 367, 368]. Provides a safety net against accidental deletion[cite: 371]. Expedited reclamation is possible for explicitly re-deleted files, and users can set different policies per namespace region[cite: 374, 375].

- **4.5 Stale Replica Detection:** Replicas become stale if a chunkserver is down and misses mutations[cite: 377]. Master maintains a **chunk version number** for each chunk[cite: 378]. When granting a new lease, the master increments this version number and informs up-to-date replicas; all record it persistently before client writes begin[cite: 379, 380, 381]. The master detects a stale replica when a restarted chunkserver reports its chunks with older version numbers[cite: 383]. Stale replicas are removed during garbage collection and are not given to clients requesting chunk locations[cite: 385, 386]. The version number is included in communications with clients/chunkservers to ensure operations use up-to-date data[cite: 387, 388].

5. Fault Tolerance and Diagnosis Dealing with frequent component failures is a major challenge[cite: 389].

- **5.1 High Availability:** Achieved through fast recovery and replication[cite: 394].
  - **Fast Recovery:** Master and chunkservers restart in seconds, irrespective of normal/abnormal termination[cite: 395, 396].
  - **Chunk Replication:** Each chunk replicated on multiple chunkservers on different racks (default 3)[cite: 399, 400]. Master re-replicates as needed[cite: 401].
  - **Master Replication:** Master state (operation log, checkpoints) replicated on multiple machines[cite: 404]. Mutations committed only after log record flushed locally and on replicas[cite: 405]. If master machine fails, external monitoring starts a new master elsewhere using replicated log[cite: 407]. Clients use a DNS alias for the master[cite: 408]. **Shadow masters** provide read-only access to metadata (possibly slightly stale) if the primary master is down, enhancing read availability[cite: 409, 410, 413].
- **5.2 Data Integrity:** Each chunkserver uses **checksumming** to detect corruption of its stored data (chunks broken into 64KB blocks, each with a 32-bit checksum kept in memory and stored persistently) [cite: 417, 421, 422, 423]. For reads, checksums are verified before returning data; on mismatch, error returned, master notified, and data cloned from another replica[cite: 424, 425, 426]. Checksumming has little effect on read performance[cite: 428]. For appends (dominant workload), checksums are updated incrementally[cite: 432, 433]. Overwrites require reading and verifying the first/last blocks of the range being overwritten before the write[cite: 435]. Chunkservers scan inactive chunks during idle periods to detect corruption[cite: 437].
- **5.3 Diagnostic Tools:** Extensive diagnostic logging (significant events, RPC requests/replies) on servers helps isolate problems, debug, and analyze performance with minimal cost[cite: 441, 443]. Logs reconstruct interaction history[cite: 447].

6. Measurements Micro-benchmarks (1 master, 16 chunkservers, 16 clients) and real-world cluster data were presented.

- Micro-benchmarks showed good read throughput, nearing network limits. Write throughput was lower, attributed to network stack interactions with GFS pipelining, but aggregate bandwidth to many clients was less affected. Record appends to a single file were limited by chunkserver bandwidth for the last chunk.
- Real-world clusters (hundreds of chunkservers, many TBs) showed master metadata was small (tens of MBs), enabling fast recovery. Read rates were significantly higher than write rates, aligning with workload assumptions. Master operation rates (200-500 ops/sec) were well within capacity. Chunk recovery after server failure was reasonably fast (e.g., 600GB restored in ~23 minutes). Workload

analysis confirmed data sizes, dominance of appends over writes, and distribution of master operations.

7. Experiences Key experiences included dealing with disk and Linux kernel issues (IDE protocol mismatches causing silent corruption leading to checksum use; fsync() costs; single reader-writer lock for mmap). Access to Linux source code was invaluable for diagnosing and fixing issues.

8. Related Work GFS is compared to AFS (location independence), xFS/Swift (data spreading), RAID (GFS uses replication), Frangipani/Intermezzo (GFS no client data cache), other GFSs/GPFS (GFS centralized master for simplicity and control), Harp (GFS could adapt primary-copy for HA), Lustre (GFS simpler, non-POSIX, focus on fault tolerance), and NASD (GFS uses commodity machines, fixed-size chunks). GFS's record append is compared to River's distributed queues.

9. Conclusions GFS supports large-scale data processing on commodity hardware by treating component failures as the norm, optimizing for huge append-mostly/read-sequentially files, and using a specialized interface. Fault tolerance is achieved via monitoring, replication, and fast recovery. High aggregate throughput comes from separating control (master) and data flow (direct client-chunkserver), minimized master involvement (large chunks, leases). GFS is a vital storage platform for Google.