THE GOOGLE FILE SYSTEM

An Architectural Overview & Deep Dive

original by Firas Abuzaid

Motivation: Designing for Failure & Scale

Google's environment presented unique challenges that standard file systems could not handle:

1. Component Failure is the Norm
- Thousands of commodity machines mean frequent failures.
- Solution: Constant monitoring, error detection, recovery.

2. Massive Files
- Multi-GB files are common (web crawls, indexes).
- Billions of objects; managing block overhead is hard.

3. Specific Mutation Patterns
- Most files are mutated by appending new data.
- Random writes are practically non-existent.

4. Bandwidth > Latency
- Priority: Processing data in bulk (high throughput).
- Low latency is secondary.

Workload Assumptions

The design is optimized for two specific access patterns:

READS

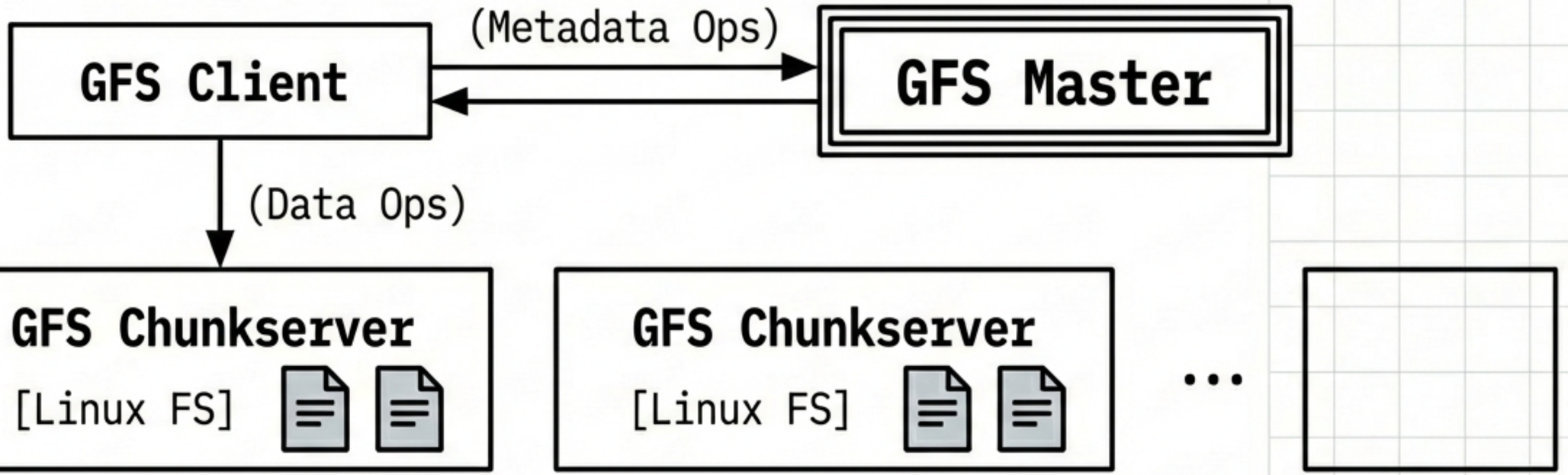
1. **Large Streaming**
   - > 1MB contiguous
2. **Small Random**
   - A few KBs
   - Performance not critical

WRITES

1. **Large Sequential Append**
   - Main data source
2. **Concurrent Appends**
   - Producer-Consumer queues

Files are seldom modified again once written.

High-Level Architecture



- **GFS Master:** Central coordinator & metadata holder.
- **GFS Client:** Code linked to app implementing the API.
- **Chunkservers:** Linux commodity machines storing files.

The Critical Design Choice: Decoupling

Why separate the flows?

1. Decoupled Control & Data

- Clients talk to Master *only* for metadata (location).
- Clients talk directly to Chunkservers for data.

2. Prevention of Bottlenecks

- The Master is a single point of failure and traffic.
- By removing data transfer from the Master, it can scale.

3. Network Efficiency

- Expensive data flows are scheduled based on network topology.

⚠️ **Note:** No Client/Chunkserver Caching. Working sets are too large. Chunkservers rely on Linux buffer cache.

The Master Node

The “Brain” of the system. Maintains global state in memory.

Responsibilities:

- Namespace management (No inodes, lookup table with locks).
- Access Control Lists (ACLs).
- Mapping files to chunks.
- Current location of chunks (polled via HeartBeats).

Memory Architecture:

- All metadata is kept in memory for speed.
- Master scans entire system state periodically in background.
- Master decides placement, but Chunkservers have the “final word” on what they actually hold.

Persistence & Reliability

The Master is a single point of failure. Reliability is key.

1. The Operation Log

- Only persistent record of metadata.
- Defines serialized order of concurrent operations.
- Replicated to remote machines.

2. Fast Recovery (Checkpoints)

- Log is compacted into B-tree checkpoints.
- Mapped directly to memory for fast restart (~1 min).

Shadow Masters (Read-Only)

- Provide read access if primary is down.
- Lag slightly (fractions of a second).

Chunks & Chunkservers

CHUNK SIZE: 64 MB

Much larger than standard FS blocks (usually 4KB)

Advantages:

- Reduces client interaction with Master.
- Persistent TCP connections reduce network overhead.
- Keeps Master's metadata size small (fits in RAM).

Disadvantages:

- Internal fragmentation.
- "Hot spots" on small files.
  → Mitigated by increasing replication factor.

GFS Relaxed Consistency Model

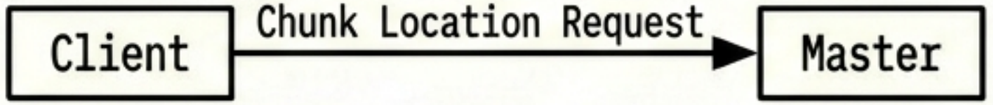
- **Consistent:** All clients see same data (regardless of replica).
- **Defined:** Consistent AND clients see entire mutation.

State	Write	Record Append
Serial Success	Defined	Defined (interspersed w/ inconsistent padding)
Concurrent Success	Consistent Undefined	Inconsistent
Failure	Inconsistent	Inconsistent

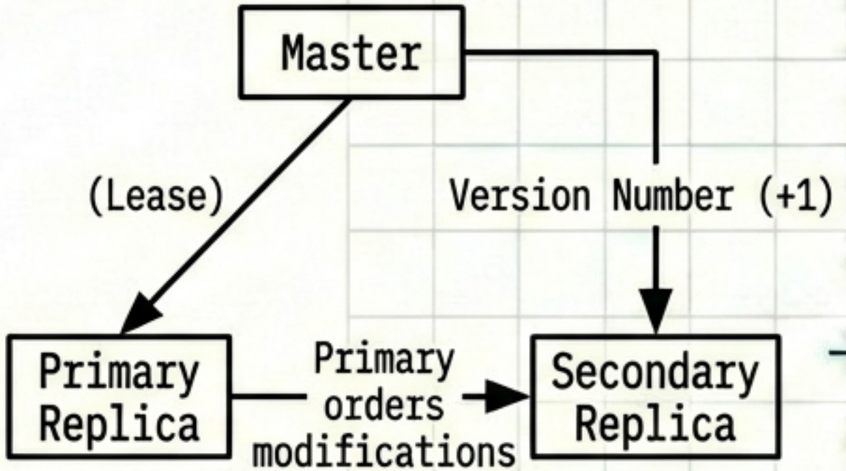
Applications must adapt (e.g., checksums, self-validating records)

Write Control Flow: Preparation (Steps 1-3)

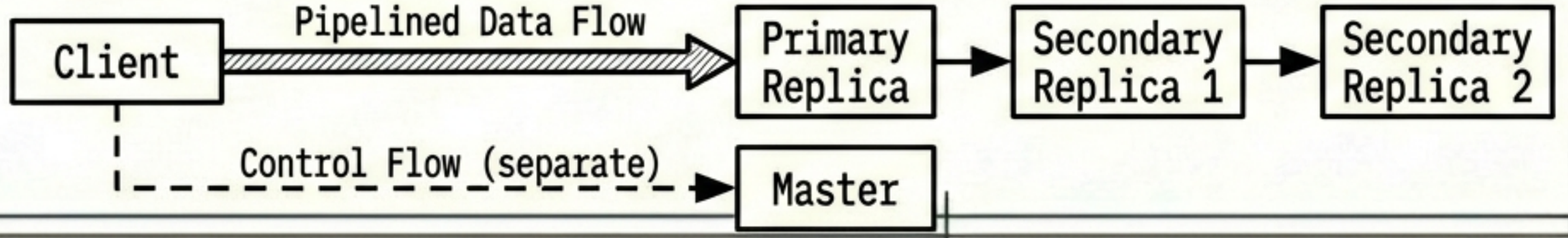
1. Client asks Master for chunk location.



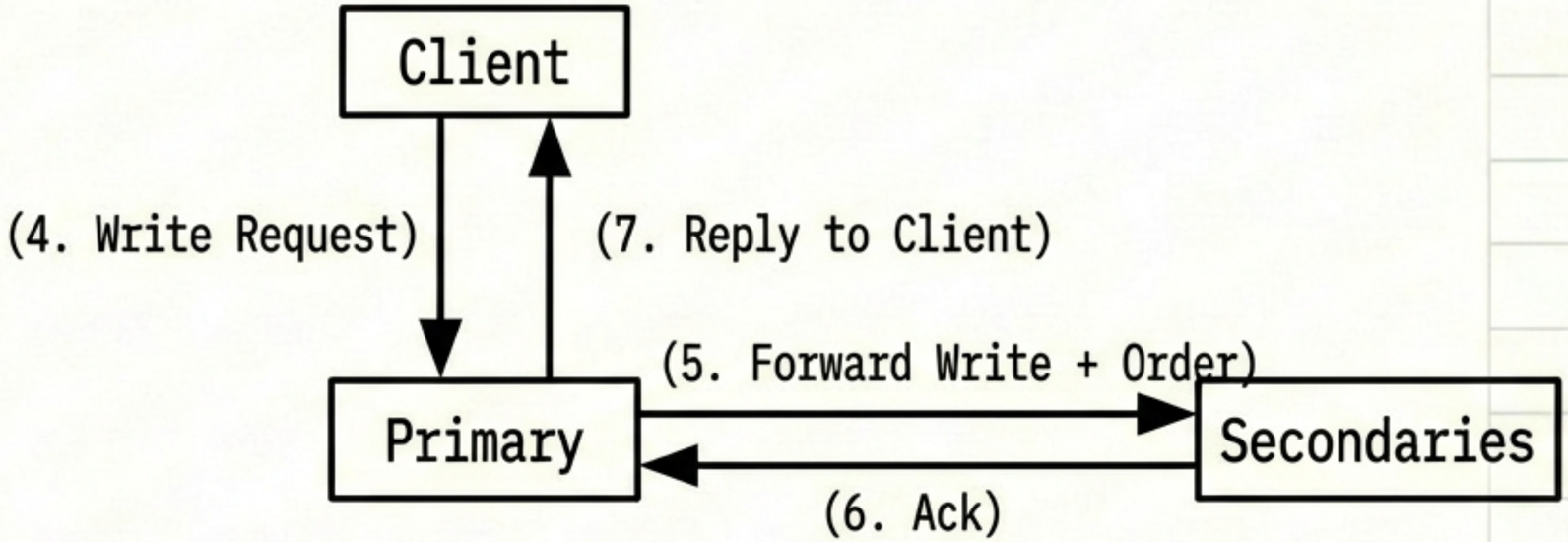
2. Master grants LEASE to one replica (Primary).
- Primary determines serialization order.
- Master increments version number.



3. Client pushes data to ALL replicas.
- Pipelined through network topology (chain replication).
- Data flows separately from control.



Write Control Flow: Execution (Steps 4-7)



Execution Details:

- Step 5: Primary assigns serial number to mutation.
- Step 6: Secondaries apply data in that exact order.
- Step 7: Primary reports success only if ALL succeed.
  → If any fail, returns Error. Client must retry.

The Atomic “Record Append”

Specialized for concurrent queues (Producer-Consumer).

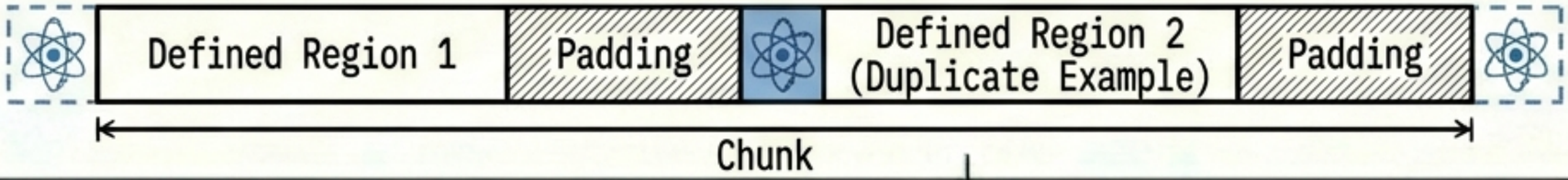
Logic: “Append this data at an offset of YOUR choosing.”
Guarantee: Data written atomically at least once.

The Process:

1. Primary checks if data fits in current chunk.
2. Fits?
   - If Yes: Writes, tells secondaries, returns offset.
     - Secondaries
     - Client
   - If No: 3. Pads chunk to end (max 16MB padding).
     - Padding (Max 16MB)
     - Client
     - • Tells client “Try Next Chunk”.

Result:

- Defined regions mixed with padding/duplicates.
- Readers must handle duplicates (checksums/IDs).



Conclusion: The Power of Specialization

GFS proves specialized designs outperform general standards for specific workloads.

Key Takeaways:

- ✅ Decoupling Data & Control. Removing the Master from the data path enabled scalability.
- ✅ Single Master Simplicity. Centralized management simplified complex logic like placement and garbage collection.
- ✅ Consistency Trade-offs. Relaxing consistency allowed for high-throughput concurrent appends (Record Append).