# Google File System

## Introduction

This paper describes the Google File system, a scalable distributed file system built for the large data-intensive applications. The requirement of the file system is generated from the workloads and technology environment of the Google. The authors describe the design decisions and mechanism behind a scalable, reliable and fault tolerant file system which runs on inexpensive hardware.

## Design Assumptions

Currently available file systems are not suitable for Google, because the scale at which Google generates and processes data is very large. Traditional file systems do not guarantee consistency in the face of multiple concurrent updates, whereas using locks to achieve consistency hampers scalability by becoming a concurrency bottleneck. Therefore, there are few key observation and assumptions are made which will be the key points in the design of the file system. They are:

1. The system is built from inexpensive components that often fail due to which continuous monitoring, fault detection, tolerance and recover immediately from the system failure are routine tasks.
2. There are modest number of large files, each mainly of 100 MB or large. Multi-gigabytes files are common and should be managed efficiently.
3. There are two type of reads: large streaming reads and small random reads. The workloads also have many large, sequential writes that append data to files.
4. High sustained bandwidth is more important than low latency.

## Design Architecture

A GFS cluster consists of a single master and multiple chunkservers and multiple clients. Files are stored in cluster. Files are divided into 64 MB chunks, each with a globally unique 64bit handle and a version number. Each chunk is stored as a file in the underlying file system and is replicated on multiple chunkservers, which improves bandwidth for read access patterns in a distributed environment. All metadata operations happen at master, data transfer happens directly between client and chunkserver.
Instead of using a directory structure, a lookup table along with prefix compression is used which leads to faster name to chunkserver translation. All metadata is In-Memory. Copy on write mechanism is used based on the observation of frequent append access patterns. Data transfers happen directly on the chunkserver after metadata operations. Overall, all optimization that are needed for the file access inside Google's distributed data processing needs have been done. POSIX compliance is not provided. An atomic append operation is provided, which ensures consistency even with concurrent updates. Control and Data flow is decoupled for maximum network bandwidth utilization. Master maintains Operation log to recover from crashes. Master state is also replicated on a remote site. Lazy garbage collection: deleted files don't immediately get freed, they are renamed and hidden and cleaned up at a later time. Block checksumming is used to ensure data consistency within a chunk.

## System Interaction

Lease and Mutation Order: Leases maintain consistent mutation order across the replicas. Master picks one replica as primary and primary defines serial order for mutations. Replicas follow same serial order.
This is mainly design to reduce the overhead of the master. GFS offers append of record in the file. Multiple clients can append to the same file concurrently on different machines, it is regulated as undefined and defined region. Changes are visible to other clients if the region is considered as defined. The data is written at least once as an atomic unit.

# Google File System

## Design Flaws

1. The single master with multiple hot replicas will face problems when metadata update operations pressure builds up in cases like small file update. (Note however that metadata read pressure can be alleviated with shadow read only masters. And also when a single master fails, then a replica master can assume mastership almost instantly). As in any centralized system, the server becomes the bottleneck for the system. Since only the master knows all the metadata maps, any downtime of the master leads to downtime of the whole cluster.
2. The paper mentions the problem of hotspots where-in small files which get accommodated in a single chunk are accessed heavily. This problem could occur with large files too where a specific portion of the file, extending to a few 10s of KBs could be very important and be heavily used. This is an artifact of large chunk size where-in the entire portion of the important data would often be present in the same chunk.
3. The consistency guarantees given by the system is not bulletproof: it is up to the client application to interpret the file contents while reading and ensure logical consistency. Lazy reclamation of deleted file space might stop new file creation because of quota restrictions.
4. One general flaw is that this filey stem is very specific to Google's data access patterns; it is unclear if any of these techniques can be used for general purpose file systems.

## Tradeoffs

1. The Google file system is designed and optimized to run on Google's computing clusters, the nodes of which consist of cheap, "commodity" computers, which means precautions must be taken against the high failure rate of individual nodes and the subsequent data loss. To prevent data lose they then must implement the replication algorithms and checkpoint systems.
2. Other design decisions select for high data throughputs, even when it comes at the cost of latency.
3. Writes have to flow through multiple racks. Redundancy vs Fault Tolerance: Increased redundancy wastes disk space but provides higher fault tolerance.
4. Space utilization vs management overhead: Smaller chunk sizes waste lesser space, but increase the metadata that needs to be managed by the master.
5. Any log structured file system, with the same append-at-end characteristic can adopt some of the features of Google File System.

## Techniques used to improve performance

1. Batching: most operations like writing the log, chuck server requests, garbage collection are not done synchronously, but batched to increase disk bandwidth.
2. Peer-to-Peer transfer model: Master is used only to establish a connection; actual data transfer doesn't involve master.
3. Copy-on-write snapshots and lazy garbage collection: examples of deferring operations to a later time or when there is a need.

## References

https://en.wikipedia.org/wiki/Google_File_System
https://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf
http://pages.cs.wisc.edu/~swift/classes/cs736-fa08/blog/2008/11/the_google_file_system.html