# Data WareHousing & Mining: A Three-Pronged View

Jeevesh Juneja

Delhi Technological University

## Google File System

The Google File System(GFS) is a distributed file systems for large distributed data-intensive applications. It is a fault tolerant storage optimized for storing multi-GB sized files that are: mostly mutated by appending data, and the data from which is mostly read in huge sequential accesses. It has a relaxed consistency model that relies on the application code to verify checksums, and validate data.

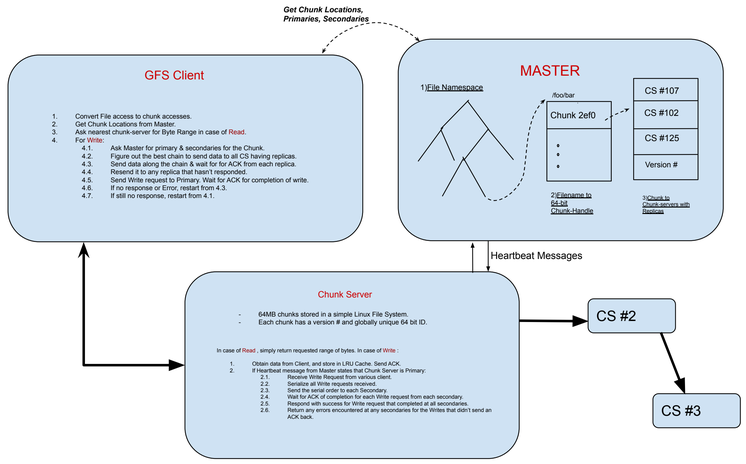


Figure 1. An overview of the GFS components. File Data flows through the solid lines. Heartbeat messages check whether chunk servers are alive. Version numbers of various chunks, the commands for creating, garbage collecting, re-replication, re-balancing are often piggybacked on these Heartbeat messages.

### Record Append

In addition to these operations, there is an atomic **Record Append** operation. The client only specifies the data to be appended and based on the multiple append requests received by the primary, it decides an offset in a chunk for each request so that the data from various clients can be concurrently appended. Each secondary is sent those offset, and all are expected to put same record in same offset. In case the append fails for some secondary, the corresponding location in that secondary will be filled with some pad entry. Client will be told that the append failed at secondaries.
In case it chooses to re-issue the append, a new offset will be decided by the primary, and it will ask all secondaries to put that record at the new offset. If all of them succeed, then some chunk-servers will have double entries, and some will have one actual record, and one pad record. This is what leads to **Relaxed Consistency** rather than strong consistency for GFS, since each replica may be in slightly different state.

### Operation Log and Persistence

The **Operation Log** maintained by the Master and logs the mutations to the file namespace, file-to-chunk mapping to persistent storage. It doesn't keep the chunk locations persistent, but gets them piggybacked on the Heartbeat messages. Also, each time it grants a **lease** to any of the chunk-servers, it also logs the **chunk version number**, to its Operation Log, after all the primaries and secondaries have done so too. Only after updating version number on disk, does it return the location of primaries and secondaries to the client. This ensures that the master can identify and garbage collect stale copies of data. The **lease** expires after 60 seconds, so any changes beyond that time period, are guaranteed to happen with a new version number.

## Fault Tolerant Virtual Machines

While the previous paper dealt with file systems, here with deal with compute. The aim here is to build a compute service tolerant to fail-stop faults. The basic idea is to model the servers as deterministic state machines that are kept in sync by starting them from the same initial state and ensuring that they receive the same input requests in the same order. The two machines are known as **Primary** and **Backup**. The amount of extra information need to keep the primary and backup in sync is far less than the amount of state (mainly memory updates) that is changing in the primary.

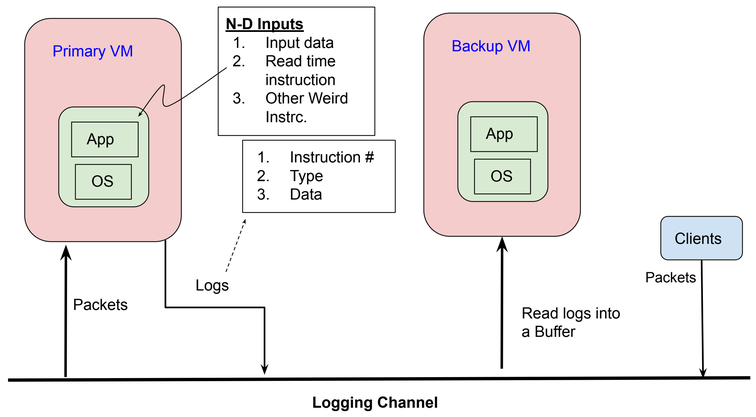


Figure 2. Keeping the two servers in sync. N-D inputs are non-deterministic inputs, they also include packets sent to primary by clients. These inputs are captured by the Primary VM and then logged via the logging channel. The type of the and their output along with exact instruction number at which the ND input was received is sent to the backup VM. The outputs of backup are dropped.

### Consistency

The requirement is that if the primary ever fails, and the backup takes over, then the backup VM should continue execution consistent with all the outputs primary VM has sent to the external world. To ensure this we have the following output rule: *the primary VM may not send output to the external world, until the backup has received and acknowledged the log entry(if any) operation producing the output.* It is not necessary that the primary VM stop execution, it can do something else in the meantime, before sending output.

### Going Live

"**Go live**" refers to a server becoming primary. To go live, a server must first replay all log entries in its buffer. If the backup stops receiving heartbeat messages from the primary, it assumes that it is dead and tries to become the primary.

To avoid split-brain problem in case only the communication with primary is cut off, but the primary is well, it is required that an **atomic test-and-set** operation be performed on a shared memory regularly by the primary, and by the backup when it wants to become the primary.

### Storage

The two servers may have a **shared virtual storage disk**, in which case we allow the Primary to read/write to it. And the backup can only read from it. Writing to it, is identical to output to external world, and should function similarly. In case the two servers have **different storage disks**, the backup writes to its disk, the same way Primary writes and keeps it in sync. This is usually used when the primary and backup are far away.

## Map-Reduce

After data and compute, we move to actually running processes on huge data to process and draw insights from it. This framework was developed by Google too, and provides a programming model where the user have specify two functions: **map** and **reduce**. The **map** function processes a key, value pair(here value is usually few MB's of data to be processed) and produces an intermediate key, value pair. A huge database can be broken into parts of few MB's where each part is processed by a map function independently. The outputs of each **map** call are collected and the **reduce** function is used to aggregate over all intermediate key/value pairs corresponding to a particular key. The user only needs to specify the **map** and **reduce** functions. The sharding of data, parallelization of processing, accumulation & re-grouping of intermediate key-value pairs, is all done automatically by the Map-Reduce framework.

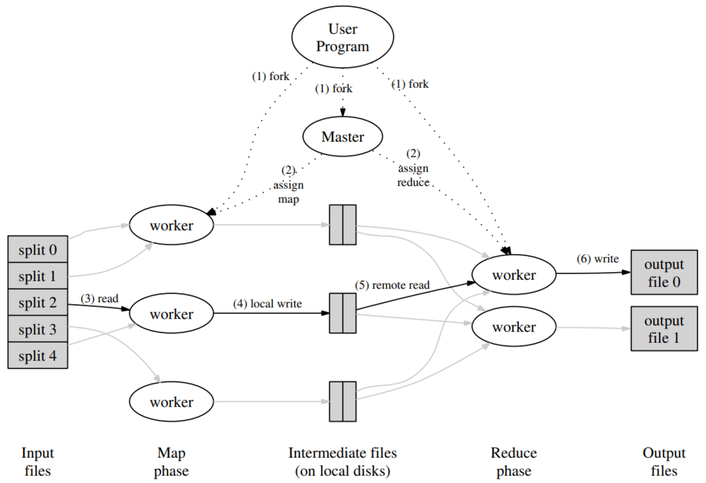


Figure 3. An overview of MapReduce framework. Master assigns map/reduce tasks to workers and keeps tracks of workers' completion/failure. Each in-progress task writes its output to private temporary files. A reduce task produces one such file, and a map task produces $R$ such files (one per reduce task). When a map task completes, the worker sends a message to the master and includes the names of the $R$ temporary files in the message. If the master receives a completion message for an already completed map task, it ignores the message. Otherwise, it records the names of $R$ files in a master data structure.

### Relation to GFS

The input files are stored in GFS. So, to minimize data transfer, the master tries to cleverly schedule the map task for a chunk of the file on the chunk-server having that chunk. Failing that, it tries to schedule the task on another nearby replica. The intermediate files are also written to GFS, but the reduce tasks will likely require integrating files from all over GFS and require a lot of remote reads.

### Efficient Reducing

We make sure to write an output intermediate key-value pair to the file number *hash(k) mod* $R$ out of total $R$. In fact, the *hash(k) mod* $R$ number output file of each **map** task, can all be grouped together and reduced at once.

## References

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[2] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 20–43, Bolton Landing, NY, 2003.

[3] Daniel J. Scales, Michael N. Nelson, and Ganesh Venkitachalam. The design of a practical system for fault-tolerant virtual machines. *ACM SIGOPS Oper. Syst. Rev.*, 44:30–39, 2010.

jeeveshjuneja2k18it061@dtu.ac.in