# Reading Assignment 1

October 23, 2018

**Anna Hedstrom** annaheds@kth.se
**Sandra Pico Oristrell** sandrapo@kth.se

## 1 Motivation

The papers **"The Google File System"** by Ghemawat et al., in 2003 [3] and **"MapReduce: Simplified Data Processing on Large Clusters"** by Dean and Ghemawat., in 2004 [2], can most generally, be viewed as an attempt to simplify everyday computing tasks at Google. At their release, Google had begun to generate a pervasive amount of terabyte-scaled data through its various services. So, in response to issues arising such as quickly outgrown systems, better architectures for *storing*- and *processing* large-scale data were due.

The **interestingness** of the approaches arguably lies in their **demonstrated practicality**, that is, the clear connection that the authors make between their self-encountered problems while handling data-intensive systems at work and the proposed systems' architectural design. Both the GFS and MapReduce are motivated by the same core idea of **scaling a system horizontally** rather than vertically, i.e., adding more nodes (inexpensive commodity machines) to a system instead of adding capacity to just one node.

Arguably, the dominant assumption leading to the development of the GFS was the observation that *"component failures are the norm rather than the exception"* [3]. As cited directly from the GFS paper, it was clear that the motivation behind its creation was to **build a "reliable" storage system out of "unreliable" machines**. Similarly, the model of MapReduce was designed to leverage low-cost machines so to simplify data-intensive processing. The main idea here was to have a program that could **process large data files in a parallel fashion** (back then, just to build an index for web pages one had to write complex and lengthy special-purpose code and latch hundreds to thousands of machines). While today, one can argue that at least the most basic version of MapReduce has somewhat fallen out of flavor, back then, as found in the active stream of research and many open source implementations that followed [5], it definitely received a lot attention at the time of its release. Regardless, there was a true need for better *storage*- and *processing* solutions that could **capacitate and simplify big data analysis on top of cheap machine hardware**.

## 2 Contributions

In both papers, the authors elaborate extensively on the benefits that could be reaped from using a distributed cluster of nodes.

In particular, the interface of GFS was designed to address problems revolving **concurrent append operations** by multiple clients and **large streaming reads** of huge multi-GBs files, where at the time, no alternative option would suffice. Most generally, the centralised design with a single master proved both simple and efficient. We find other value propositions as follows.

Firstly, GFS is **fault tolerant**. Given that the data- and control flow is decoupled, the system provides a good fault tolerance solution (and the master does not necessarily become a bottleneck for read and write operations). Specifically, two different plans are provided for the master and the chunk servers: **i. Master fault tolerance**. As it is a single-master system, recovery is fast given that the master state often remains small. At the same time, the limitation of the log size (removing the log operations after the checkpoint) and the replication of them in different backups implies a persistent system. **ii. Chunk-server fault tolerance**, one of the parts of the system that creates good fault-tolerance is the way that the client is pushing data to the replicas. In case that one replica does not respond, client retries. Nevertheless, if the number of replicas drop below some number, the master re-balances the replicas, replicating more chunk-servers to guarantee that the system is still working properly.

Secondly, GFS provides **high throughput for reading**. The solution generates three different copies for reading a file, with 125MB/sec in aggregation (see p.11).

Thirdly, **data integrity** is another important contribution. Considering that the GFS often handles thousands of disks on hundreds of machines, data corruption problems due to disk failures is a common experience. Thus to detect corruption (without any need of comparison), each chunk-server verifies its own data independently using checksumming.

For **MapReduce**, we would attribute most of its popularity to its ability to maintain a very **simple user interface** while still delivering significant **performance gains** on a variety of data analysis tasks (see p.11). We conclude that the main contributions are the following items.

Firstly, the **library is easy-to-use**. Given that the program hides most of the details of parallelisation, even inexperienced programmers can use it.

Secondly, to address the issue of machines breaking, MapReduce is designed to be **resilient to failure.** MapReduce replicates each map task up to three times and ensures that each task is small in size (to speed up the recovery in case of failure). It leverages periodic heartbeats and checkpoints for instances of master failure.

Thirdly, the model is **scalable.** To increase the capacity of the system, one can easily do it across an arbitrary number of machines.

Lastly, it also provides refinements such as **back-ups, load balancing** and **I/O locality.** Back-ups allow for smart re-scheduling that allocates work from slow workers (aka "stragglers") to nodes with more slack. To optimise resource use via load balancing, MapReduce further ensures that every worker is assigned only smaller tasks (instead of "one big task"). To reduce network traffic, it takes advantage of I/O locality which basically means that one "move task closer to data".

# 3 Solution

The **GFS** cluster consists of a **single master** with **multiple chunkservers** that can be **accessed by multiple clients**.

- The **single master** is responsible for maintaining all file system metadata, including e.g., names space, chunk location (including replicas), file-to-chunk mappings, garbage collection etc. The master communicates with chunkservers with periodic heartbeat messages to detect potential failure. E.g., for a read operation, the master would first be contacted by one (or multiple) client(s) to get the file's chunk locations, whose clients then deals directly with the chunkservers.
- The **chunkservers** are the locations/ disks where data is stored. The files are broken into 64 MB sized chunks, where each chunk consists of an immutable globally unique 64-bit chunkhandle. At chunk creation, these are assigned by the master to map the chunk location to the specific file. For reliability, the default is to have each chunk replicated across 3 servers.
- The **clients** communicates with the master for caching meta data and with the chunkservers for direct data interaction/ exchange. The clients are linked to the application using the file system's API, which supports operations such as *create, delete, open, close, read,* and *write* files.

The **MapReduce** model also follows a master-slave architecture. Here, the **map- and reduce operations** are controlled by the single master (the job tracker) which schedules and allocates the input data to different worker nodes (task tracker). Very briefly, we have:

- The **map operation** that parses unsorted input and produces a set of intermediate key/value pairs.
- The **reduce operation** that then take these tuples and apply a reduce function that combines all the values that shares the same key to produce the final output (that is written on a local disk).

- In-between the two operations lies additional intermediate steps, including the optional *combiner* function (for partial **merging**) as well as **shuffling**, **sorting**.

At the most fundamental level, MapReduce is "just" (distributed) list processing: it reads the input in forms of a list and transforms its elements to a desired output.

## 4 Discussion/ Our Opinion

As explained before, **GFS** is a really good distributed file system that provides reliable access to a huge amount of data using large clusters. Nevertheless, as every system, it has also its own limitations.

Firstly, it is **not possible to "randomly" write**. Since GFS is based on the assumption that most files are no longer based on random writes and overwrites, but modified by appending at the end.

Secondly, GFS is **not optimised for small sized files.** Because the minimum chunk size is 64MB (which is much larger than most file systems), GFS does not efficiently support for smaller/ mix of data sets. For example, we wonder how GFS performs in the context of many small files, such as a Gmail application? We found the reasoning behind the standard chunk size poorly explained in the paper.

Thirdly, by reading the authors' description of the single-master architecture, one might think that the GFS is completely relieved from all "master-bottleneck" issues. But we ask what happens when the amount of meta data maintained by the master increases drastically, e.g., when storage increase to petabyte scale or/and thousands of requests happens simultaneously?

Moreover, for **MapReduce** we learn that it has often been regarded as one of the key enabling approaches to meet big data demands [4]. Given how easy it is to implement a basic MapReduce program, it is not hard to understand *why*. However, we also find that the model's usability is **limited set of domains**, namely in "lists- associated-with-a-key" problem set-ups, such as word counting. We note that other drawbacks include the following.

Firstly, there is **no support for real-time writes.** Potentially the biggest limitation of MapReduce, is its batch-oriented, sequential nature. Since its creation, it is unarguably true that the production environment for data processing models has drastically changed - today there are higher requirements, including handling of stream data (where data often grows and evolves) which invitably reduce the applicability and relevance of MapReduce.

Secondly, **complexity of code rises quickly.** While the claimed "simplicity" is true as long as it involves counting on lists, writing distributed applications for more complex queries, e.g., iterative computing algorithms, becomes vastly more troublesome (requiring manual programming of multiple jobs).

Thirdly, it suffers from **slow processing speed**. Given that the MapReduce model requires intermediate writes of data, as a result, it becomes slow on a per-task basis. Even worse, since most applications needs to chain multiple MapReduce jobs together, performance deteriorates due to costly disk usage. Interestingly, Bortnikov et al., measured that as much as 90% of tasks are "waste" in MapReduce [1].

## References


[1] Edward Bortnikov, Ari Frank, Eshcar Hillel, and Sriram Rao. Predicting execution bottlenecks in map-reduce clusters. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, pages 18–18. USENIX Association, 2012.

[2] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[3] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. *The Google file system*, volume 37. ACM, 2003.

[4] Katarina Grolinger, Michael Hayes, Wilson A Higashino, Alexandra L'Heureux, David S Allison, and Miriam AM Capretz. Challenges for mapreduce in big data. In *Services (SERVICES), 2014 IEEE World Congress on*, pages 182–189. IEEE, 2014.

[5] Ren Li, Haibo Hu, Heng Li, Yunsong Wu, and Jianxi Yang. Mapreduce parallel programming model: A state-of-the-art survey. *International Journal of Parallel Programming*, 44(4):832–866, 2016.