1. GFS has three main components: a single master and multiple chunkservers accessed by multiple clients. Chunkservers store chunks on local disks, which are the place where data is handled. Clients never read or write data through the master; instead, it asks the master which chunk to connect to. To read a file, the client first determines the chunk index using the byte offset within the file. It then sends a request to the master, including the filename and chunk index, to ask for a read operation. The master then responds with the corresponding chunk handle and the replica locations. From there, the client sends a request that has the chunk handle and a byte range within that chunk to one of the replicas, probably the closest one. Finally, the chunkmaster returns the data to the client. Further requests can be made here without recontacting the master until the cache expires.
2. GFS masters store their metadata in their local memory. As discussed in the paper, there are some concerns. The one that Google mentioned was that the number of chunks and, hence, the whole system's capacity was limited by how much memory the master had. Google later claimed that it shouldn’t be a problem since there is often available space since the last chunk is partially filled. In addition, the cost of adding more memory is less expensive than the cost of simplicity, reliability, performance, and flexibility. However, one thing that Google might not have considered was how quickly and large files can evolve over time. GFS masters’ capacity is limited by how much space they can store their metadata, suggesting limited future scalability when files grow. Moreover, it is also more volatile to failure since the risk of memory overflow is inherently more significant. In addition, as the masters only store files locally, all metadata will be lost if a master fails, leading to substantial downtown and recovery efforts.

From there, Google redesigned GFS and implemented Colossus, which used BigTable to address those issues. The metadata are now stored in a distributed file system instead of locally on the masters. This allows easier scalability since you can add more nodes, higher availability since it is more equipped for fault tolerance, and a more efficient storage system.

1. A lease is a mechanism the master grants to a chunkserver to manage access to data. The lease allows the client to write into that chunk of data in a given period without being interfered with by other clients, causing inconsistency. The chosen chunkserver becomes the primary, among other replicas of itself, responsible for managing access and modification to that chunk for a given lease period, ensuring consistency during that time.

When a client wants to write a file, it first contacts the master, who then grants a lease and provides the client with the primary and secondary replica locations. The client then sends the data to the primary, producing a sequence (for serializability) to the file and forwarding it to all secondary replicas (to ensure consistency). After the operation finishes, the primary reports back to the master, which then reports to the client. Furthermore, the primary can contact the master to extend the lease if it needs more time.

Snapshot is created to capture a copy of all replicas of a chunk periodically. This enables the system to revert to an earlier state if necessary. Before a snapshot is taken, the master revokes all leases to prevent any modifications to the system to ensure consistency. It then logs the operation to the operating disk. After the snapshot operation finishes, new leases are granted, and the system continues its operation.

1. GFS is the backend data storage system for MapReduce. Recall that GFS splits the data into smaller chunks and stores them across machines in clusters. Input files used by MapReduce are typically large, so GFS provides a storage solution that is reliable, scalable, resilient, and efficient for accessing them by splitting them up and storing them in different machines within clusters. GFS also allows parallel access to multiple mappers to execute tasks concurrently, improving efficiency and performance. On top of that, between the Map and the Reduce task, intermediate files are generated, which are also stored using GFS on local machine disks using GFS. GFS then facilitates data movement between the tasks and provides fault tolerance ability.

Borg is Google’s cluster management system. Borg is used to allocate necessary resources to run the tasks, schedule the tasks to execute, and provide fault tolerance ability to the entire system. As the operation spans lots of clusters and, hereby, machines, it is probably impossible to implement this system without using them.

1. MapReduce is an expansive system that can run many tasks concurrently. Because of that, parallelization is crucial to achieve such performance. The authors point out some interesting facts about the system in the paper. First is scalability—MapReduce leverages Borg and GFS, which utilize a mass network of commodity machines to power. The ability to expand is endless as more machines can be added to the cluster. Then, it is fault tolerance. Checkpoints and snapshots are created on top of replicas made by Borg, which reserves the ability to recover from failures as we have learned they are the norms. We also realize that network bandwidth is a scarce resource. With that, MapReduce has a mechanism to store files locally on the machine during the intermediate steps to boost performance and reduce network congestion. Besides, MapReduce also can parallelize its tasks to leverage the clusters. The system is also easy to use, even for less-experienced users, since it hides the complex implementation and offers a more straightforward workflow.

One thing that I find interesting about the discovery in this paper is that the number of applications increases over time. In one and a half years, the number of applications using MapReduce increases exponentially, from 0 to nearly 1000.

1. I want to develop a system that counts how many mispayments there are from a credit card issuer in a month.

* Map 1: Extract information from all active credit cards.
  + Parse and concatenate the credit card number with its payment history. Note that there can be many payments within one credit report period.
  + Input: credit card number – this number should be unique across all issuers.
  + Output: emit a list of key-value pairs of credit card numbers and their on-time payment performance: 1 for on-time and 0 for missed.
* Reduce 1: Transform and count all values from the Map 1 task
  + Count the total number of times the user has had a mispayment or on-time payment for each credit card number.
  + Input: A list of key-pair values generated from Map 1 task
  + Output: A tuple of (credit\_card\_number, [num\_on\_time, num\_missed])
* From here, the credit issuers can leverage this data to understand better how their business is working. They can extend credit lines for those with a percentage of on-time performance beyond a certain threshold while putting accounts that have missed many times on a more careful review.