# The Google File System and its application in MapReduce

Johannes Passing
johannes.passing@hpi.uni-potsdam.de

Hasso-Plattner-Institute for Software Engineering, D-14482 Potsdam

**Abstract.** In the need for a scalable infrastructure supporting storage and processing of large data sets, Google has developed a number of cluster-based technologies which include the Google File System (GFS) and MapReduce.
This paper aims at explaining the key ideas and mechansims deployed in the Google File System and illustrates the architecture and fault tolerance features implemented. To highlight the role of the Google File System in Google's software landscape, MapReduce as a technology for data processing that leverages the services provided by GFS is presented. Along the discussion of these systems, key differences to other approaches regarding design and implementation are pointed out.

## 1 Introduction

Operating the market-leading internet search engine [4], Google's infrastructure faces unique performance and scalability needs. Although exact numbers have not been leaked to the public, Google is estimated to run approximately 450.000 servers [5].

Software runnable on clusters consisting of large numbers of computers require sophisticated distributed algorithms which raise the complexity of software systems [12]. Moreover, providing highly availabe services requires fault tolerance and resilience to be a key requirement in the design of a distributed system. Challenged by these requirements, Google has developed a number of technologies, the *Google File System* (GFS) [1] and *MapReduce* [2] being two of them, that constitute a basic infrastructure supporting the development of distributed software in this specific context. Most prominently, these two technologies serve as the technical basis for indexing web content [2] as well as for the database system *BigTable* [3], which plays a key role in the Google search engine.

The paper is structured as follows. Section 2 introduces GFS and illustrates design and architecture of the system. Following, a walkthrough of key operations such as reading and writing to the file system is provided, along with a discussion of the relaxed consistency model employed. Although this paper focuses primarily on GFS, section 3 discusses MapReduce in order to highlight the interplay between GFS and MapReduce, in which GFS is able to play off its strengths. After laying out the design rationale and core ideas of MapReduce, a sample walkthrough is discussed and fault tolerance features are presented.

While both GFS and MapReduce are proprietary technologies not available outside Google, projects adopting the core ideas of these technologies such as Apache Hadoop [16] exist. Moreover, research papers such as [15] have evaluated the possibilities of leveraging the ideas of MapReduce in a different context, namely multi-core computing.

## 2 Google File System

As a consequence of the services Google provides, Google faces the requirement to manage large amounts of data – including but not being limited to the crawled web content to be processed by the indexing system. Relying on large numbers of comparably small servers [6], GFS is designed as a distributed file system to be run on clusters up to thousands of machines. In order to ease the development of applications based on GFS, the file system provides a programming interface aimed at abstracting from these distribution and management aspects.

Running on commodity hardware, GFS is not only challenged by managing distribution, it also has to cope with the increased danger of hardware faults. Consequently, one of the assumptions made in the design of GFS is to consider disk faults, machine faults as well as network faults as being the norm rather than the exception. Ensuring safety of data as well as being able to scale up to thousands of computers while managing multiple terabytes of data can thus be considered the key challenges faced by GFS.

Having distilled the aims and non-aims of a prospective file system in detail, Google has opted not to use an existing distributed file system such as NFS [7] or AFS [8] but instead decided to develop a new file system. Whereas file systems like NFS and AFS are targeted at being generally applicable, GFS has been fully customized to suite Google's needs. This specialization allows the design of the file system to abstain from many compromises made by other file systems. As an example, a file system targeting general applicability is expected to be able to efficiently manage files with sizes ranging from very small (i.e. few bytes) to large (i.e. gigabyte to multi-terabyte). GFS, however, being targeted at a particular set of usage scenarios, is optimized for usage of large files only with space efficiency being of minor importance. Moreover, GFS files are commonly modified by appending data, whereas modifications at arbitrary file offsets are rare. The majority of files can thus, in sharp contrast to other file systems, be considered as being append-only or even immutable (*write once, read many*).

Coming along with being optimized for large files and acting as the basis for large-volume data processing systems, the design of GFS has been optimized for large streaming reads and generally favors throughput over latency. In expectance of large working sets, client side caching techniques as used by various other file systems such as AFS have been deemed ineffective and are thus not used by GFS.

Another drastic difference to other distributed file systems is the fact that GFS does not provide a POSIX [9] interface – in the case of Linux, which is the platform GFS is operated on, this means that GFS also does not integrate

with the Linux *Virtual File System (VFS)* layer. Instead, GFS implements a proprietary interface applications can use.

### 2.1 Architecture

As indicated before, GFS is a distributed system to be run on clusters. The architecture relies on a master/slave pattern [12]. Whereas the master is primarily in charge of managing and monitoring the cluster, all data is stored on the slave machines, which are referred to as *chunkservers*. In order to provide sufficient data safety, all data is replicated to a number of chunkservers, the default being three. While the exact replication algorithms are not fully documented, the system additionally attempts to use machines located in different racks or even on different networks for storing the same piece of data. This way, the risk of loosing data in the event of a failure of an entire rack or even subnetwork is mitigated.

To implement such distribution in an efficient manner, GFS employs a number of concepts. First and foremost, files are divided into *chunks*, each having a fixed size of 64 MB. A file always consists of at least one chunk, although chunks are allowed to contain a smaller payload than 64 MB. New chunks are automatically allocated in the case of the file growing. Chunks are not only the unit of management and their role can thus be roughly compared to *blocks* in ordinary file systems, they are also the unit of distribution. Although client code deals with *files*, files are merely an abstraction provided by GFS in that a file refers to a sequence of chunks. This abstraction is primarily supported by the *master*, which manages the mapping between files and chunks as part of its *meta data*. Chunkservers in turn exclusively deal with chunks, which are identified by unique numbers. Based on this separation between files and chunks, GFS gains the flexibility of implementing file replication solely on the basis of replicating chunks.

As the master server holds the meta data and manages file distribution, it is involved whenever chunks are to be read, modified or deleted. Also, the meta data managed by the master has to contain information about each individual chunk. The size of a chunk (and thus the total number of chunks) is thus a key figure influencing the amount of data and interactions the master has to handle. Choosing 64 MB as chunk size can be considered a trade-off between trying to limit resource usage and master interactions on the one hand and accepting an increased degree of internal fragmentation on the other hand.

In order to safeguard against disk corruption, chunkservers have to verify the integrity of data before it is being delivered to a client by using checksums. To limit the amount of data required to be re-read in order to recalculate a checksum if parts of a chunk have been modified during a write operation, these checksums are not created on chunk-granularity but on block-granularity. Blocks, which do not correspond to file system blocks, are 64 KB in size and are a logical subcomponent of a chunk.

GFS is implemented as user mode components running on the Linux operating system. As such, it exclusively aims at providing a distributed file system

while leaving the task of managing disks (i.e. the role of a file system in the context of operating systems) to the Linux file systems. Based on this separation, chunkservers store each chunk as a file in the Linux file system.

### 2.2 Master

Although all payload data is exclusively handled by chunkservers, the master plays a key role in the dynamics of a GFS cluster. Besides managing meta data, which includes information about chunks, their location and their replication status, the master also implements the file system namespace. The namespace allows files to be named and referred to by hierarchical names such as `/foo/bar/foobar` as known from other file systems. It is notable that due to the comparatively large size of a chunk and the fact that the meta data of a chunk is as small as 64 bytes, the master server is able to hold all meta data in memory, which not only simplifies data structures and algorithms but also ensures good performance. In order to avoid the master becoming the bottleneck of the cluster, the master has additionaly been implemented using multi-threading and fine grained locking. Any communication performed between chunkservers and the master is performed using a remote procedure call facility.

The master not only decides about the chunk servers used whenever a new chunk is requested to be allocated, it also arranges for the chunks being re-replicated whenever the replication count is determined to have fallen below a chosen minimum replication count. Such situations can occur whenever chunks have become unavailable due to failures such as chunkserver crashes, disk failures or failed integrity checks. Another reason to deliberately raise the replication count of certain chunks is to improve speed of access when chunks turn out to be part of 'hot files', i.e. files accessed very frequently. GFS employs a sophisticated scheme to ensure that traffic and effort required for such re-replication do not slow down the cluster operation in a significant manner.

To provide safety of meta data in case of a crash, all changes made by the master is written (using a write-through technique) to the *operations log*, which may be compared to the transaction log of a database system [10]. To limit the size of the log and thus also the time required to replay the log, snapshots of the meta data are taken periodically and written to disk. After a crash, the latest snapshot is applied and the operation log is replayed, which – as all modifications since the last snapshot have been logged before having being applied to the in-memory structures – yields the same state as existed before the crash. Location information of chunks, however, is not persisted in the same manner. The primary rationale of this decision is the fact that location information may have become obsolete during the downtime of the master (e.g. a chunk server may have crashed at the same time as the master, rendering all chunks housed by this servers as unavailable). To avoid situations where meta data is not in sync with actual state, all location data is therefore queried from chunkservers during startup of the master. Although more expensive, this assures that location information is up to date and in sync.

The master, as discussed so far, poses a single point of failure, i.e. as soon as the master fails, the cluster becomes inoperable. However, GFS mitigates this risk by employing *shadow masters* acting as hot-standby master servers. By continuously replaying a replica of the operations log generated by the primary master, shadow masters keep themselves up to date and can thus – in the case of a primary master failure – provide read-only access to meta data. Shadow masters, however, as indicated by [14] do not take over all responsibilities of the primary master, so that the cluster may still be considered to run in a slightly degraded mode during primary master outage.

Although not discussed in detail in this paper, further duties of the master also include monitoring the health of chunkservers by exchanging *heartbeat* messages. As files (and thus chunks) are not immediately deleted but rather marked as being deleted, the master also periodically runs a *garbage collector*, which purges all chunks that have become orphaned or have been marked as deleted for at least three days.

### 2.3 Walkthrough

The following section aims at providing an overview of the inner workings and the dynamics of a GFS cluster by providing a walkthrough of three basic operations – reading, modifying and appending to a chunk.

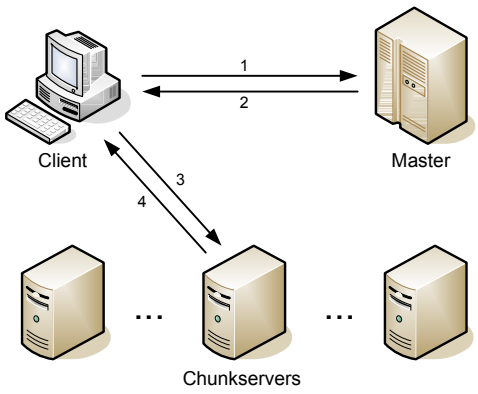


**Fig. 1.** Reading a chunk

**Reading a chunk** In order to read a portion of a file (see figure 1), the client application uses an API provided by a GFS client library all applications using GFS link to. When being requested to read from a file, this library sends an appropriate request to the master (1). The master, provided the filename and

the offset within the file, looks up the identifiers of affected chunks in the meta data database and determines all chunkservers suitable to read from (For the following steps, it is assumed that only a single chunk is affected by the read operation). The chunk identifier along with all chunk locations is returned to the client (2). The client library, taking the individual locations and their 'distances' into account, chooses a chunkserver from the list of servers offered by the master and requests it to read the desired range of data (3). The chunkserver reads the affected blocks from disk, verifies their checksums, and – assuming a successful integrity check – returns the data to the client (4). If the checksum validation reveals data to be corrupted, the chunkserver reports the corruption to the master and returns an appropriate failure to the client. The requester will then choose a different server from the list of chunkservers initially retrieved from the master, and retry the request.

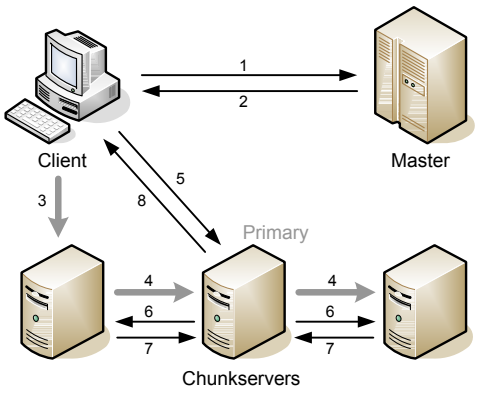


**Fig. 2.** Modifying a chunk

**Modifying a chunk** Modifying a file at an arbitrary offset requires more provisions to be made by GFS (see figure 2). The client, again utilizing the GFS library, sends the name of the affected file along with the desired offset to the master server (1). As data is about to be altered, the master has to synchronize the process properly with any other concurrent modifications affecting the same chunk. To attain this synchronization, the master designates one chunkserver to be the *primary replica* by granting it a *lease* for the specific chunk. If a lease has already been granted to a server for this chunk, this lease will be reused. Using this scheme, the risk of applying conflicting changes to the same chunk on different chunkservers is mitigated. The chunk identifier, along with the location of the primary replica and the remaining chunkservers holding the affected chunks (*secondary replicas*) is returned to the client (2). In the next step, the client pushes

the data (illustrated as thick arrow) to *any* of the chunkservers (3), which in turn will propagate this data to the other chunkservers in a pipelined fashion (4). No modifications are applied yet – instead the data can be considered to be stored in a staging area, ready for further application. Once all affected chunkservers have acknowledged receipt of the data, the client sends a write request to the primary (5). As multiple clients may simultaneously request modifications to a chunk, the primary now defines an execution plan referred to as *mutation order*, which basically constitutes a schedule [11] defining the sequence of modifications to be performed. Following this order, the primary applies the modifications using the data received before. The write request, along with the mutation order is then forwarded to the secondary replicas (6), which apply the modifications in the predefined order. Whenever a chunk has been modified, its version number is incremented. By versioning each chunk, stale replicas, i.e. replicas that have missed certain modifications and thus provide outdated data, can be detected by the master and appropriate action (e.g. re-replication) can be taken. If all steps of the modification procedure have been completed successfully, the requests are acknowledged (7) and success is reported to the client (8).

If a file modification spans multiple chunks, the described procedure is executed for each affected chunk in isolation. Whenever a different client attempts to read from a block currently being modified, the master will direct this read request to the primary copy. This ensures that a client reads up-to-date data. It is also worth noting that the serialization scheme used does not protect against *lost updates* [11] – instead, the last writer always 'wins'.

**Appending to a chunk** Although appending to a file is usually interpreted as a mere specialization of modifying a file, GFS handles this action in a special, optimized manner in order to improve performance of this action. In particular, GFS ensures that while trying to minimize locking, no race conditions between two concurrent append operations can occur – a problem commonly encountered whenever two applications append to a write-shared file opened with `O_APPEND` on a POSIX-compliant file system. This optimization is primarily owed to the fact that applications, such as MapReduce, heavily rely on file appends to be atomic, race-free and quick.

Instead of choosing the current end of file as the offset and scheduling a write request, a client uses a special operation provided by GFS to append to a file. Consequently, the client sends the name of the file to be appended to the master, which in turn decides on the offset to use and – as in the case of file modification – designates a primary replica and returns chunkserver locations back to the client. Again, the client pushes the data to one of the chunkservers, which will propagate the data to the remaining affected chunkservers. Rather than sending a write request, however, the client now sends an append request to the primary. The primary checks whether appending the data to the chunk would let the size of the last chunk exceed 64 MB. If this is the case, it pads the chunk, requests secondaries to do the same and indicates the client to retry the request in order to trigger allocation of a new chunk. If size does not exceed the

maximum chunk size or a new chunk has been allocated, the primary appends the data received before to the chunk and requests secondaries to do the same. Finally, success is reported to the client. In case the modification has failed on one of the replicas, the whole append operation is retried by the client. As such, the append operation has *At Least Once* semantics [12].

### 2.4 Relaxed Consistency Model

Retrying a failed append request increases the likelihood of a successful operation outcome, yet it has an impact on the consistency of the data stored. In fact, if a record append operation succeeds on all but one replica and is then successfully retried, the chunks on all servers where the operation has succeeded initially will now contain a duplicate record. Similarly, the chunk on the server that initially was unable to perform the modification now contains one record that has to be considered garbage, followed by a successfully written new record (illustrated by record B in figure 3).

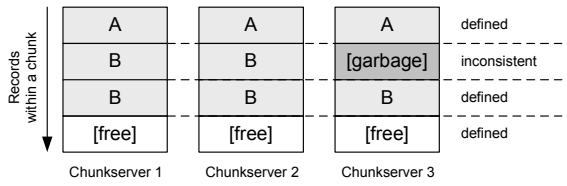


**Fig. 3.** Consistency state of records on different chunkservers

In order not to have these effects influence the correctness of the results generated by applications, GFS specifies a special, relaxed consistency model and requires clients to cooperate. GFS classifies a file region, i.e. a part of a chunk, as being in one of three states:

- Consistent. A region is *consistent* iff all clients see the same data.
- Defined. A region is *defined* with respect to a change iff it is consistent and all clients see the change in its entirety.
- Inconsistent. A region is *inconsistent* if it is not consistent.

Based on this classification, the situation discussed above yields the first record in an inconsistent state, whereas the second record is considered defined. Consistent but not defined regions can occur as a result of concurrent successful modifications on overlapping parts of a file.

As a consequence, GFS requires clients to correctly cope with file regions being in any of these three states. One of the mechanisms clients can employ to attain this is to include a unique identifier in each record, so that duplicates can be identified easily. Furthermore, records can be written in a format allowing proper self-validation.

The relaxed nature of the consistency model used by GFS and the requirement that client code has to cooperate emphasizes the fact that GFS is indeed a highly specialized file system neither intended nor immediately applicable for general use outside Google.

## 3 MapReduce

While the Google File System serves as a solid basis for managing data, it does not prescribe applications to follow certain patterns or architectures. On the one hand, this freedom underlines the flexibility of GFS, on the other hand, it raises the question how applications can make efficient use of both, GFS and a cluster infrastructure. With BigTable [3] and MapReduce [2], Google has published two approaches how a distributed database system as well as a system for batch data processing can be implemented on top of GFS and a cluster system.

The primary role of MapReduce is to provide an infrastructure that allows development and execution of large-scale data processing jobs. As such, MapReduce aims at efficiently exploiting the processing capacity provided by computing clusters while at the same time offering a programming model that simplifies the development of such distributed applications. Moreover and similar to the requirements of GFS, MapReduce is designed to be resilient to failures such as machine crashes.

Google uses MapReduce to process data sets up to multiple terabytes in size for purposes such as indexing web content.

### 3.1 Inspiration

To achieve the goals mentioned, MapReduce has been inspired by the idea of *higher order functions*, in particular the functions *map* (also referred to as *fold*) and *reduce*. These functions are an integral part of functional programming languages such as Lisp or Haskell, but are also commonly provided by non purely-functional languages such as Python.

The primary benefit the functional programming paradigm and these functions in particular promise is to allow the creation of a system that incorporates automatic parallelization of tasks.

Although the basic idea of these functions has been retained in MapReduce, it is worthwhile to notice that the exact semantics of the terms *map* and *reduce* in MapReduce literature deviates from the semantics of their archetypes [13]: Not *map* and *redurce* but rather the mapper and reducer (provided by the runtime) implement the idea of a higher order function. The functions *map* and *redurce* merely denote the user-provided functions passed to the mapper and reducer respectively.

The following paragraphs summarize the ideas of map and reduce as applied by MapReduce.

### 3.2 Idea

One of the assumptions made by MapReduce is that all data to be processed can be expressed in the form of key/value pairs and lists of such pairs. Both keys and values are encoded as strings. Based on these assumptions, the key idea of MapReduce is to implement the application exclusively by writing appropriate map and reduce functions. Provided these functions, the infrastructure not only transparently provides for all necessary communication between cluster nodes, it also automatically distributes and load-balances the processing among the machines.

*Map* is a function written by the user that takes a key/value pair as input and yields a list of key/value pairs as result. A canonical use case for *map* is thus to digest raw data and generate (potentially very large quantities of) unaggregated intermediate results.

*Reduce* is the second function implemented by the user. It takes a key and a list of values as input and generates a list of values as result. The primary role of *reduce* is thus to aggregate data.

### 3.3 Walkthrough

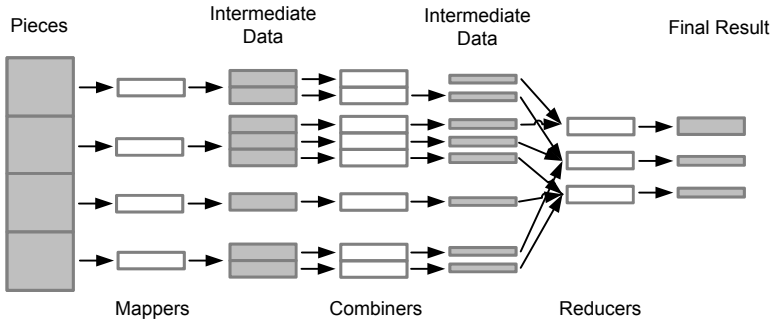


**Fig. 4.** A MapReduce job

To illustrate the functionality of MapReduce and its operation in a computing cluster, the following section provides a walkthrough over the various stages of data processing in a MapReduce job.

Similar to GFS, an application wishing to take advantage of MapReduce has to be linked against a certain client library simplifying the use of the infrastructure to the user. In order to be able to properly parallelize the map stage, all input data has to be split into $M$ *pieces*, each being 16-64 MB in size. Although not clearly indicated by [14] and [2] it seems likely that the library has been

specially adapted to GFS, so that chunks of a file, also being 64 MB in size, can be readily used as pieces. Besides being able to read data from GFS files as well as from BigTable, the MapReduce implementation can also be provided data in a customized manner by implementing a *Reader* interface.

Like GFS, MapReduce employs a master/slave architecture. The master, which is spawned as soon as the data pieces have been prepared, plays a key role in the execution of a MapReduce job. Duties of the master include assigning work items to the slave machines, referred to as *workers* as well as managing load balancing and monitoring the execution of jobs.

Whereas there is only one master process, a job involves a number of worker processes, which are distributed over a potentially large number of machines. It is important to notice that MapReduce jobs are usually run on the same machines that constitute a GFS cluster. The primary rationale of sharing machines for these two different purposes is performance. Attempting to schedule workers to run on the machine hosting the data to be processed yields the benefit of enabling the worker to read the input data from local disks. Not only does reading from local disks provide a higher throughput rate than transferring data over the network, it also decreases the overall traffic on the network. The fact that GFS stores multiple copies of each chunk on different machines increases the likelihood of MapReduce being able to schedule computation close to the data. Whenever using the optimal machine is not feasible, MapReduce tries to use a machine *close* to the data.

Having $M$ pieces of data to be processed, the master will assign $M$ workers the task of executing the map function, each being supplied a different piece. Running in parallel, each map task produces a set of intermediate results in the form of key/value pairs, which is buffered and periodically flushed to the local disk of the respective machine. Rather than writing all data to a single file, a partioning scheme is used. Using a stock or custom *partitioner*, intermediate data is – based on the key of each key/value pair – split into $R$ buckets.

According to the operation performed during the map stage, intermediate data can turn out to be rather large, requiring significant amounts of data to be transferred over the network during the next stage. Depending on the nature of the reduce function, an optimization can be used to alleviate this situation. Whenever a reduce function is both associative and commutative, it is possible to 'pre-reduce' each bucket without affecting the final result of the job. Such a 'pre-reduce' function is referred to as *combiner*, although the reduce function can often be used both for reducing and combining. Using a combiner, the buckets on each machine are processed again, yielding new intermediate files which can be assumed to be smaller than the files generated in the map stage. In order to simplify further processing, MapReduce automatically sorts each intermediate file by key.

As soon as intermediate data is ready for further processing, its location is reported to the master. Not before the map stage (and the combine stage, if applicable) has been completed on *all* workers, the master will assign $R$ reduce tasks. Again, the same machines are being used as during the map stage. Each

reduce task is assigned one bucket, i.e. one intermediate file per machine having been involved in the map stage. Due to the input data being spread throughout the network, the reduce stage usually involves a large number of network reads. The additional effort of using a combiner in the previous stage can thus be expected to have significant impact on the performance of the reduce stage as less data has to be transferred over the network.

Each reduce task now iterates over the (sorted) intermediate data and generates the final results in the form of a list of values. Unlike intermediate data, which is stored temporarily on local disks only, the data generated by the reduce stage constitutes the final result of the job and is thus written to GFS to be universally accessible and stored in a safe manner.

When all reduce tasks are completed, the job is considered to have finished and the user program is woken up. It is worthwhile to notice that the result of the overall job is provided as a set of $R$ (assuming R is larger than one) files (i.e. the files generated by the reducers) rather than a single file.

### 3.4 Scheduling

As indicated by the previous section, choosing $M$ and $R$ has significant impact on the execution of a job. Choosing a small R has the potential benefit of generating a small number of output files only, yet it limits the degree of parallelization possible in the reduce stage. Similarly, parallelization as well as load balancing of the map stage depends on the relation of M to the number of machines available. Google suggests choosing R as a small multiple of the number of machines [2]. To improve load balancing, $M$ should be significantly larger than (i.e. a multitude of) $R$.

Besides trying to schedule computation close to the location of data, MapReduce implements additional mechanisms for reducing the total duration of jobs. In a manner similar to BitTorrent [12], MapReduce tries to mitigate the danger of *stragglers*. Stragglers are tasks that – possibly due to hardware-related problems – exhibit significantly degraded performance and thus – as all tasks are required to finish before a job is considered competed – delay the overall job. To avoid such situations from arising, MapReduce, when a job is close to completion, schedules *backup copies* of remaining tasks. Although increasing the overall workload by spawning additional tasks, by picking the result of the first task finished, a straggler looses its impact on the performance of the job.

### 3.5 Fault Tolerance

To account for the risk of failing hardware, MapReduce implements several features related to fault tolerance.

Whenever a machine currently running map tasks fails, any intermediate data already generated becomes unavailable, too. But even if only a map process fails, any intermediate data created by this process is considered to be not usable as it cannot be clearly determined whether the task has already completed processing

(but has not yet reported so) or whether it failed before completion. Therefore, the master, recognizing the crash, re-schedules execution of the entire map task.

In a similar fashion, failing reduce tasks are automatically re-executed. As reduce tasks write their output to GFS and checkpoint their progress, however, the data already generated remains accessible so that already completed portions do not have to be repeated.

An interesting additional feature of MapReduce is the treatment of 'bad' input data, i.e. input data that repeatedly leads to the crash of a task. The master, tracking crashes of tasks, recognizes such situations and, after a number of failed retries, will decide to ignore this piece of data.

### 3.6 Interplay with GFS

Although MapReduce is not restricted to work exclusively in combination with GFS, there are several points the seamless integration between these two technologies become evident. As mentioned before, the fact that GFS splits files into chunks of fixed size and replicates chunks across machines simplifies data access and scheduling of mappers for MapReduce. Furthermore, the fault tolerance features of GFS and the fact that data is likely to remain accessible even in the case of machine of network outages also improves the fault tolerance of MapReduce job executions, as indicated in the discussion of the reduce stage. The main and most striking property of the interplay between GFS and MapReduce, however, can be considered the idea of using the same cluster for both data storage and data processing. Scheduling computation close to data and thus reducing the total amount of data required to be moved over the network, promises significant performance benefits as well as high throughput.

## 4 Conclusion

With GFS and MapReduce, Google has developed two remarkable technologies for large scale data storage and processing. While significantly lowering the effort required by developers to develop distributed applications, these technologies have proven to be capable of scaling to clusters consisting of thousands of machines managing and processing large amounts of data [6].

On a technical perspective, it is notable that Google has made the effort to question common and familiar features such as strict consistency models for file systems and has developed a highly specialized infrastructure developing much of its strengths out of omitting features not being absolutely required.

Although simplifying development of distributed systems, both GFS and MapReduce are technologies that require users to develop code on a slightly more technical level than may be considered to be common. Expressing the entire functioality of a data processing application in a set of map and reduce functions can be challenging. Similarly, correctly coping with inconsistent GFS data requires the user to have a thorough understanding of the infrastructure and its workings.

Having been developed with Google's own requirements in mind, both systems are highly specialized solutions and thus cannot be considered to be easily applicable to other use cases or even for general purpose use. GFS and MapReduce should thus be seen as two innovative approaches complementary to existing technologies such as other distributed file or database systems.

## References


1. S. Ghemawat, H. Gobioff, S.-T. Leung, The Google File System, ACM SOSP 10/2003.
2. J. Dean, S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, OSDI 2004.
3. F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, R.E. Gruber, Bigtable: A Distributed Storage System for Structured Data, OSDI 2006.
4. Hitwise Pty. Ltd., Google Accounted For 64 Percent Of All U.S Searches In March 2007, http://www.hitwise.com/press-center/hitwiseHS2004/searchenginesmarch2007.php (retrieved 02/01/2008).
5. The New York Times, 06/14/2006, Hiding in Plain Sight, Google Seeks More Power, http://www.nytimes.com/2006/06/14/technology/14search.html (retrieved 02/01/2008).
6. InformationWeek, Google Revealed: The IT Strategy That Makes It Work, 08/28/2006 (retrieved 02/01/2008).
7. S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, D. Noveck, Network File System (NFS) version 4 Protocol, RFC 3530.
8. J.H. Morris, M. Satyanarayanan, M.H. Conner, J.H. Howard, D.S.H. Rosenthal, F.D. Smith, Andrew, a Distributed Computing Environment, Communications of the ACM 29, 03/1986
9. IEEE, Information Technology – Portable Operating System Interface (POSIX) Part 1: System Application Programming Interface (API), 1003.1-1990, 1990.
10. P.A. Bernstein, E. Newcomer Principles of Transaction Processing, Morgan Kaufmann, 1997
11. G. Vossen, G. Weikum, Fundamentals of Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery, Morgan Kaufmann, 2001.
12. G. Coulouris, J. Dollimore, T. Kindberg, Distributed Systems. Concepts and Design, 4th rev. ed., 2005.
13. Ralf Lämmel, Google's MapReduce Programming Model Revisited, SCP journal, 2006.
14. Google, Cluster Computing and MapReduce, http://code.google.com/edu/-content/submissions/mapreduce-minilecture/listing.html, 2007 (retrieved 02/01/2008).
15. C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, C. Kozyrakis, Evaluating MapReduce for Multi-core and Multiprocessor Systems, HPCA, 2007.
16. Apache Hadoop Project, http://hadoop.apache.org/core/ (retrieved 02/01/2008).