The Google File System vs MapReduce: Simplified Data Processing on Large Clusters

Ottar Gudmundsson Xin Ren

I. Motivation

A. GFS

The Design of GFS has been driven by key observations of authors' application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions.

1) Component failures are the norm rather than the exception.
2) Files are huge by traditional standards.
3) Most files are mutated by appending new data rather than overwriting existing data.
4) Co-designing the applications and the file system API benefits the overall system by increasing flexibility.

B. MapReduce

Over the past five years, the authors and many others at Google have implemented hundreds of special-purpose computations that process large amounts of raw data to compute various kinds of derived data. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, authors designed a new abstraction that allows them to express the simple computations they were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library.

II. Contributions

A. GFS

The Google File System demonstrates the qualities essential for supporting large-scale data processing workloads on commodity hardware. While some design decisions are specific to authors' unique setting, many may apply to data processing tasks of a similar magnitude and cost consciousness. The file system has successfully met authors' storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our service as well as research and development efforts that require large data sets. The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds of clients. GFS is an important tool that enables authors to continue to innovate and attack problems on the scale of the entire web.

B. MapReduce

The major contributions are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs. The implementation makes efficient use of these PC resources and therefore is suitable for use on many of the large computational problems encountered at Google. It is one of the reasons that the MapReduce programming model has been successfully used at Google for many different purposes.

III. Solution

A. GFS

GFS provides a familiar file system interface and supports the usual operations plus snapshot and record append. A GFS cluster consists of a single master and multiple chunkservers and is accessed by multiple clients. Files are divided into large fixed-size chunks which stored locally on chunkservers. The master maintains all file system metadata which includes the namespace, access control information, the mapping from files to chunks, and the current locations of chunks. All metadata are kept in memory, while namespaces and file-to-chunk mappings are also stored persistently in operation log. The master polls chunkservers for chunk locations at startup and keep itself up-to-date thereafter because it controls all chunk placement and monitors chunkserver status with regular HeartBeat messages. Clients interact with the master for metadata operations, but all data-bearing communication goes directly to the chunkservers.

Each mutation (data change) is performed at all the chunks replicas. Leases are used to maintain a consistent mutation order across replicas. The master grants a chunk lease to one of the replicas, which is called the primary. The primary picks a serial order for all mutations to the chunk. All replicas follow this order when applying mutations. A lease has an initial timeout of 60 seconds, but can be extended indefinitely.

The master's involvement in reads and writes is minimized by having Data flow decoupled from control flow so that it is possible for the single master to vastly simplify the design and make sophisticated chunk placement and replication decisions using global knowledge without itself being a bottleneck.

1) For read operation, a client contacts the master to get the locations of the replicas and caches this information, then sends a request to one of the replicas, most likely the closest one. Further reads of the same chunk require no more client-master interaction until the cached information expires or the file is reopened.
2) For write operation, a client contacts the master to get the identity of the primary and the locations of the other (secondary) replicas and caches this data for future mutations. The client pushes the data to all the replicas linearly along a carefully picked chain of chunkservers in a pipelined fashion. Each chunkserver will store the data in an internal LRU buffer cache until the data is used or aged out. Once all the replicas have acknowledged receiving the data, the client sends a write request to the primary. The primary assigns consecutive serial numbers to all the mutations and applies the mutation to its own local state in serial number order. The primary forwards the write request to all secondary replicas which apply mutations in the same order as the primary. The secondaries all reply to the primary after completing the operation. The primary replies to the client. Any errors encountered at any of the replicas are reported to the client.
3) GFS handles record append similarly to write operations with only a little extra logic at the primary to check if appending the record to the current chunk would cause the chunk to exceed the maximum size (64 MB) and react.

Besides chunk lease management, the master also controls system-wide activities such as namespace locking, garbage collection of orphaned chunks, and chunk migration between chunkservers.

1) Each master operation acquires read lock on internal nodes, and read/write lock on the leaf before it runs.
2) Chunk replicas are created for three reasons: chunk creation, re-replication as soon as the number of available replicas falls below a user-specified goal, and rebalancing periodically for better disks pace and load balancing. To maximize data reliability and availability, and maximize network bandwidth utilization, when master is choosing a place to create a chunk replica, it considers: place new replicas on chunkservers with below-average disk space utilization; limit the number of recent creations on each chunkserver; spread replicas of a chunk across racks.
3) When a file is deleted by the application, the file is just renamed to a hidden name and removed after three days (configurable interval) during the masters regular scan of the file system namespace. Before it is removed, the file can still be read under the new, special name and can be undeleted by renaming it back to normal.
4) In a HeartBeat message regularly exchanged with the master, each chunkserver reports a subset of the chunks it has, and the master replies with the identity of all chunks that are no longer present in the masters metadata. The chunkserver is free to delete its replicas of such chunks.
5) Whenever the master grants a new lease on a chunk, it increases the chunk version number and informs the up-to-date replicas. The master and these replicas all record the new version number in their persistent state. The master will detect that this chunkserver has a stale replica when the chunkserver restarts and reports its set of chunks and their associated version numbers. The master removes stale replicas in its regular garbage collection.

For high availablility, both the master and the chunkserver are designed to restore their state and start in seconds no matter how they terminated, each chunk is replicated on multiple chunkservers on different racks, the master's operation log and checkpoints are replicated on multiple machines and shadow masters provide read-only access to the file system even when the primary master is down.

For data intergrity, each chunkserver uses checksumming to detect corruption of stored data and for troubleshooting, extensive and detailed diagnostic logging has helped immeasurably in problem isolation, debugging, and performance analysis, while incurring only a minimal cost.

File namespace mutations are atomic and it is guaranteed by the master: namespace locking guarantees atomicity and correctness; operation log defines a global total order of these operations. Mutated file region is guaranteed to be defined and contain the data written by the last mutation and it is achieved with applying same order on all replicas and chunk version to detect stale replicas.

B. MapReduce

The MapReduce interface implementation is targeted to the computing environment in wide use at Google with storage provided by GFS. When the user program calls the MapReduce function, the following sequence of actions occurs:

1) The MapReduce library in the user program first splits the input files into M pieces(size configurable for user). It then starts up many copies of the program on a cluster of machines.
2) One of the copies of the program is special - the master. The rest are workers that are assigned work by the master. The master picks idle workers and assigns each one a map task or a reduce task.
3) A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined Map function. The intermediate key/value pairs produced by the Map function are buffered in memory.
4) Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.

5) When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together.
6) The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's Reduce function. The output of the Reduce function is appended to a final output file for this reduce partition.
7) When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the MapReduce call in the user program returns back to the user code.

The number of partitions (R) and the partitioning function are specified by the user. After successful completion, the output of the MapReduce execution is available in the R output files.

The master keeps several data structures. For each task, it stores the state (idle, in-progress, or completed), and the identity of the worker machine (for non-idle tasks). For each completed map task, the master stores the locations and sizes of the R intermediate file regions produced by the map task.

The master pings every worker periodically. If no response is received from a worker in a certain amount of time, the master marks the worker as failed. Any map tasks completed by the worker are reset back to their initial idle state, and therefore become eligible for scheduling on other workers. Similarly, any map task or reduce task in progress on a failed worker is also reset to idle and becomes eligible for rescheduling. If the master fails, the MapReduce computation is aborted and Clients can check for this condition and retry the MapReduce operation if they desire.

When the user-supplied map and reduce operators are deterministic functions of their input values, the distributed implementation produces the same output as would have been produced by a non-faulting sequential execution of the entire program. It is achieved by:

1) 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.
2) When a reduce task completes, the reduce worker atomically renames its temporary output file to the final output file. If the same reduce task is executed on multiple machines, multiple rename calls will be executed for the same final output file. The atomic rename operation provided by GFS to guarantees that the final file system state contains just the data produced by one execution of the reduce task.

When the map and/or reduce operators are nondeterministic, the implementation provides weaker but still reasonable semantics.

The implementation conserves network bandwidth by taking advantage of the fact that the input data is stored on the local disks of the machines that make up the cluster. Having each worker perform many different tasks improves dynamic load balancing, and also speeds up recovery when a worker fails: the many map tasks it has completed can be spread out across all the other worker machines. To avoid "straggler" issue, when a MapReduce operation is close to completion, the master schedules backup executions of the remaining in-progress tasks. The task is marked as completed whenever either the primary or the backup execution completes.

IV. Your Opinions

1) GFS and MapReduce have similar architecture, single master and multiple slaves (chunkservers or workers). MapReduce using GFS as data storage and the fact that in the cluster, a machine can be chunkserver and worker at the same time, make it possible for MapReduce to schedule map tasks on the machines that store the data which conserves network bandwidth. This is a good design.
2) In both GFS and MapReduce, single master is single point of failure. System availability can be improved if there is a redundancy.
3) In both GFS and MapReduce, data flow is decoupled from control flow.
4) In GFS, when master is unavailable, caches in GFS client and shadow masters can still maintain read availability for files that are not being actively mutated or applications that do not mind getting slightly stale results. But in MapReduce, if master is unavailable, the system can not be functional.
5) In MapReduce, garbage collection for intermediate key/value pairs produced by mappers is not mentioned.
6) In MapReduce, when a mapper worker is unavailable, the output of its completed tasks can be replicated to other nodes as the data on local disks are managed by GFS, this then can avoid re-executing its completed tasks if re-executing is more expensive than replicating
7) The semantics of MapReduce is partially guaranteed by atomic rename operation provided by GFS.
8) It is good that MapReduce paper has page number and GFS paper can be improved by having it.
9) It is good that in the MapReduce paper Introduction, the last paragraph describes the structure of the paper and GFS paper can be improved by having it.
10) It is good that both papers describe the related work and note references.
11) In GFS, when Table 2 is referred to, it mentions "the table" instead of "Table 2".