Tony Nguyen

Dr. Shawn Bowers

CPSC 324 01

12 February 2024

Reading 2

Part 1: Ghemawat, S., Gobioff, H., and Leung, S.-T. (2003). The Google File System. *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 29–43.

Reading Notes

1. Abstract
    - Scaleable distributed file system
    - For large distributed data-intensive applications
    - Fault tolerance while running on inexpensive commodity machines
    - High aggregate performance to a large number of clients
    - Accessible to hundreds of clients
2. Introduction
    - Google insights:
        - Component failures are the norm
        - Files are huge by traditional standards
        - Files are mutated by appending data instead of overwriting.
            - Allows performance optimization and atomicity guarantees
        - Relaxed GFS consistency model
        - Design an atomic appending operation to allow multiple concurrent appending without extra synchronization

3. Design Overview

Assumptions

- Since machines are built on a constant fail basis, they must monitor themselves and detect, tolerate, and recover from failures.
- Able to manage large files (very large)
- Two kinds of reading
    - Large streaming reads
    - Small random reads
- The workloads have many large, sequential writes that append data to files
    - Once written, seldomly modified again
    - Arbitrary modification is still supported
- Well-defined semantics for multiple clients
    - Allows concurrent appending to files
    - Atomicity with minimal synchronization overhead is essential
- High sustained bandwidth is more important than low latency

Interface

- Pretty much similar to Linux directories
- No API
- Has *snapshot* and *record append* operations
    - Snapshot: Create a copy of a file or directory tree at a low cost
    - Record append: Allows multiple clients to append data to the same file concurrently while guaranteeing the atomicity of each individual client's append

Architecture

- Has a single *master* and multiple *chunkservers*, accessed by multiple *clients*
- Commodity Linux machine
- Run both a chunkserver and a client on the same machine
- Files
    - Divided into fixed-size chunks
    - Each is identified by an immutable and globally unique 64-bit *chunk handle*
    - Replicated on multiple chunkservers (3 replicas by default)
- Metadata
    - Maintained by the master
    - Periodically communicate with each chunkserver in *HeartBeat*
    - GFS client code linked into each application implements the file system API and communicates with the master and chunkservers
    - Don't provide POSIX API
    - No cache file data (client and chunkserver).

Single Master

- Simplifies design and enables the master to make sophisticated chunk placement and replication decisions
- Clients never write and read data through the master. Instead, they ask a master which chunkserver to connect to

Chuck Size

- Chosen to be 64MB. Avoid wasting space due to internal fragmentation
- Advantages:

  - Reduces clients' needs to interact with the master
  - Clients are more likely to perform more operations on a given chunk, reduce network overhead
  - Reduce the size of metadata stored on the master, keeping them in memory
- Disadvantage
  - A small file may have small chunks, which are all stored in a chunkserver. It can become a hot spot if multiple clients access it at the same time or when the system is used by a batch-queue system.

Metadata

- The master stores 3 major types of metadata:
  - File and chunk namespaces
  - Mapping from files to chunks
  - Locations of each chunk's replicas
- All metadata is kept in the master's memory
- The first two types are kept persistent by logging mutation to an operation log, stored on the master's local disk and replicated on remote machines
  - Advantage: update the master state simply, reliably, and not risking inconsistency
- In-Memory Data Structure
  - Master operations are fast since metadata is stored in memory.
  - Also, it is easier for the master to scan through its entire state
    - Used to implement chunk garbage collection, re-replication (when a chunkserver fails), and chunk migration (balance load and disk space .)

    - Concern: the number of chunks and hence the capacity of the whole system is limited by how much memory the master has
        - It shouldn't be a problem since there is often available space since the last chunk is partially filled.
        - Has the ability to add extra memory to the master (relatively cheap)

- Chunk locations
    - The master doesn't keep a persistent record of which chunkserver has a replica of what chunk. It only pulls it at startup
        - Keep itself updated using HeartBeat messager to communicate with the chunkservers
    - Not keeping a persistent record to not have to deal with chunkservers starting, restarting, etc. They happen really often.
        - Also, chunkservers give decisions on what to do with the disks, which don't have to be reflected on the disk as errors occur and they could be shut off (by the chunkservers)
- Operation Log
    - Contains a historical record of critical metadata changes, which is central to GFS
        - The only persistent record of metadata
        - Serves as a logical timeline that defines the order of concurrent operations
    - Replicate it on multiple remote machines and respond to a client operation after flushing the log to disk both locally and remotely

Consistency Model

- Relaxed consistency model. Relatively simple but efficient
- Guarantees by GFS
    - File namespace are atomic by locking
    - The state of the file region after mutation depends on the type of mutation
        - Consistent: all clients always see the same data, regardless of replicas
        - Defined (after a mutation): consistent and clients will see what the mutation writes entirely
        - Mutations succeed without interference from writes: defined
        - Concurrent successful mutations: undefined but consistent because it might not reflect what is being written
        - Failed mutations make the region inconsistent.

	Write	Record Append
Serial success	*defined*	*defined* interspersed with *inconsistent*
Concurrent successes	*consistent* but *undefined*	
Failure	*inconsistent*	

Table 1: File Region State After Mutation

        -
    - Write mode:
        - Data is written at an application-specified file offset
    - Record append
        - Data is appended atomically at least once even in the presence of concurrent mutations, but at an offset of GFS's choosing
        - Compared to the normal append, it appends to the end of the file

        - The offset is returned to the client and marks the beginning of a defined region that contains the record
        - GFS may insert padding and or record duplicates in between
        - The region is hereby inconsistent, typically dwarfed by the amount of the user data
        - See note for more
    - After a sequence of successful mutations, the region is guaranteed to be defined and contain the data written by the last application
    - Achieved by
        - Applying mutations to a chunk in the same order on all replicas
        - Use the chunk version number to detect any replica that has become stale (missed mutation since the chunkserver was down). Stale replicas are never used and will end up in garbage collections
    - The clients may read information from a stale replica, which is okay since it will be refreshed, and the old information will be purged out. When a reader retries and contacts the master, it will get the current chunk locations.
    - Failed components can corrupt or destroy data. GFS identifies failed chunkservers by handshakes and data corruption by checksumming
        - Data is recovered from valid replicas
        - If all replicas fail before GFS can react, an error message is returned instead of corrupt data (all or nothing)
- Implications for Applications
    - Relaxed consistency model

        - Appends over overwrites
        - Checkpointing and writing self-validating
        - Self-identifying record
    - Appending instead of overwriting
        - Generate a file from beginning to end
        - Atomically renames a file to a permanent name after writing all data
    - Appending is more efficient and resilient to application failure
        - Checkpointing allows writers to restart incrementally and keeps readers from processing successfully written file data that is still incomplete from the application's perspective
    - Many writes concurrently append to a file instead of a queue
    - Deal with padding and duplicates by
        - Each record has extra information like checksums so that it can be verified
        - Checksums are used to identify and discard extra padding and record fragments
        - If it can't tolerate the occasional duplicates, it can filter them out using unique identifiers, which are needed anyway to name the corresponding application entity

4. System Interactions

- System design: Minimize the master's involvement

Leases and Mutation Order

- Mutation

- Operation that changes the contents or metadata of a chuck
- Append or write
- Each mutation is performed at all replicas
- Use leases to maintain a consistent mutation order across all chunks
- The primary (a replica granted by the master to use as a lease) decides the order of all mutations order
- The lease is designed to minimize management overhead. Typically has a 60s timeout but can be extended

Data Flow

- Decouple the flow of data from the flow of control to use the network efficiently
- Goal:
  - Utilize each machine's network bandwidth
    - Data is pushed linearly along a chain of chunkservers instead of in some kind of topology
    - Each machine's outbound bandwidth is used to transfer AFAP rather than divided among recipients (all for one, then the next one, etc.)
  - Avoid bottlenecks and high-latency links
    - Each machine forwards the data to the closest machine that has not received
    - Distance is calculated using the IP address
  - Minimize the latency to push through all data over a TCP connection
    - Once a chunkserver receives some data, it forwards immediately
    - See paper for time formula

Atomic Record Appends

- Traditional write
    - Client specifies the offset at which data is to be written
    - Concurrent write to the same region are not serializable
        - May have data fragments from multiple clients
- Record append
    - Client only specifies the data
    - GFS appends it to the file at least once atomically at an offset that GFS chooses and returns that offset to the client
    - Simplifies synchronization effort compared to the traditional writes
    - A kind of mutation that follows the same data flow logic as above
- GFS does not guarantee that all replicas are bytewise identical, only that data is written at least once as an atomic unit

Snapshot

- Snapshot operation
    - Make a copy of a file or a directory tree (the source) almost instantaneously
    - Minimize interruption to the mutation process
    - Use to quickly create branch copies or to checkpoint the current state
- Use standard copy-on-write to implement snapshots
    - When the master receives a snapshot request, it revokes any outstanding leases to make sure any writes need interaction with the master to find the holder
    - After the leases are revoked, the master logs the operation to the disk

Answers to Questions

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.

3.

- 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.

Additional Questions and Observations

1. The way GFS implements its system relaxes some of the normal forms. I find it actually interesting since I have learned that it is better to reach the highest normal form possible. However, in this case, Google actually has traded off some of that consistency to achieve higher performance and efficiency.
2. I also find it pretty interesting the way Google implements the GFS system. I have always wondered how Google would implement such systems that can store a large amount of information. I do have a question at the point when Google decides to switch to Colossus. I know the decision arises when GFS comes with many drawbacks, but I wonder how the team at Google knew when it was the switching point.

Part 2: Dean, J., Ghemawat, S. (2004). MapReduce: Simplified Data Processing on Large Clusters. *Proceedings of the 6th USENIX Symposium on Operating System Design and Implementation (OSDI)*, pp. 137–150.

Reading Notes

1. Introduction

- MapReduce handles computations of very large data, which is usually distributed across machines to finish in a reasonable time
- MR allows a simple expression of the computation but hides messy details
- Inspired by the map and reduce primitives
- Apply a map operation to compute a set of intermediate key/value pairs, then apply the reduce operation to all values that have the same key to combine data appropriately
- Allows easy parallelization of large computations and to use of re-execution as the primary mechanism for fault tolerance

2. Programming Model
    - Takes a set of input key/value pairs and produces a set of output key/value pairs
    - Map
        - Takes an input pair and produces a set of intermediate key/value pairs
        - The MR groups all intermediate key I and passes them to the reduce function
    - Reduce
        - Accepts an intermediate key I and a set of values for that key
        - Merges these values together to form a possibly smaller set of values
        - Typically one or zero output value produced
        - The intermediate values are supplied via an iterator

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

- The map function emits each word plus an associated count of occurrences (1)
- The reduce function sums together all counts emitted for a particular word

map (k1,v1) → list(k2,v2)
reduce (k2,list(v2)) → list(v2)

I.e., the input keys and values are drawn from a different domain than the output keys and values. Furthermore, the intermediate keys and values are from the same domain as the output keys and values.

- See paper for more example applications

3. Implementation

Execution Overview

- The Map is distributed across machines by automatically partitioning the input data into a set of M splits, which can be processed in parallel by different machines
- The Reduce is distributed by partitioning the intermediate key into R pieces using a function (hash(key) mod R)
- The sequence of action:
    - The MR splits the input files into M pieces of 16-64MB per piece. It then starts up many copies of the program on a cluster of machines

  - One of the copies is the master. The rest are workers who are assigned tasks by the master (M map tasks and R reduce tasks). The master picks idle workers and assigns each one a task
  - A worker assigned a Map task then reads the contents of the corresponding input split. It then parses the key/value pairs out of the input data and passes each pair to the Map function. The intermediate key/value pairs produced by the Map function are buffered in memory
  - The buffered pairs are written to the local disk and partitioned into R regions by the partitioning function. The location of these pairs is passed back to the master, who then forwards to the reduce workers
  - When a reduce worker is notified, it uses remote procedure calls to read the data. When it finishes reading all intermediate data, it sorts them by the keys so that all occurrences are grouped together.
  - The reduce worker iterates over the sorted intermediate data. For each unique key, it passes the key and its set of intermediate values to the reduce function. The output of the reduce function is appended to the final output file for this reduce partition
  - When all tasks are completed, the master wakes up the program. At this point, the MR call in the user program returns back to the user code
- The output is available in the R output files

Master Data Structures

- The master keeps several data structures
- For each task, it stores the state and the identity of the worker machine

- The master stores the location and size of the R-intermediate regions that are propagated from the map to the reduce task. Updates to this location and size are received as map tasks are completed, which is pushed incrementally to workers that have in-progress reduce tasks.

Fault Tolerance

- Worker failure
    - The master marks a worker failed if it fails to respond to the ping. Any map works completed are reset and turned back to rescheduling. Similar to in progress map and reduce work.
    - However, the failed completed reduce task will not be rescheduled since the output is stored in a global file system, while the completed map task is stored on the local disk
    - MR is resilient to large-scale worker failures
- Master failure
    - Abort MR computation and notify the user, who can elect to run again if they wish
- Semantics in the Presence of Failures
    - When the map and reduce operators are deterministic functions of their input values, our distributed implementation produces the same output as would have been produced by a non-faulting sequential execution of the entire program
    - Relies on atomic commits of maps and reduces task output

Locality

- Network bandwidth is scarce, and we need to conserve it
- MR takes advantage of GFS, which stores information in 3 replicas. MR attempts to get data from the closest replica. If it fails, it will try the closest to the failed replica
- With that, most input data is read locally and consumes no bandwidth

Task Granularity

- Ideally, we should have several tasks for each machine
- There are constraints on how large M and R can be. See paper for more details

Backup Tasks

- One of the common causes for lengthening MR operation is a "straggler": a machine that takes an unusually long time to complete tasks
- Resolve mechanism
    - When an operation is close to completion, the master schedules a backup of remaining in-progress tasks. The operation will be marked as completed when either the primary or backup one is completed.
    - Though we increase the computational resource, the runtime is significantly reduced

4. Refinements

Partitioning Function

- The default partitioning function uses hashing – hash(key) mod R
- MR library supports additional function to better fit the required situation

Ordering Guarantees

- Increasing key ordering is guaranteed
- Makes it easier to generate s sorted output file per partition

Combiner Function

- Does partial merging of data before it is sent over the network to the reduce task
- Executed on each machine that performs a map task
- Typically the same code as the reduce function
- Difference in the way MR handles the output. The output of the Combiner Function is written to an intermediate file and then sent to a reduce task

Input and Output Types

- Supports different formats

Side-effects

- In some cases, users want to have additional outputs. MR relies on the application writers to make them atomic and idempotent

Skipping Bad Records

- Sometimes, there are bugs in the map or reduce functions. MR has an optional mode that can detect and slip bad records to move forward

Local Execution

- Debugging can be tricky since the system is expansive
- MR provides an execution mode that sequentially executes all operations on a local machine
- Flags and debugger tools are available

Status Information

- The master runs an internal HTTP server to generate the report page that

Counters

- A facility to count occurrences of various events

5. Performance

Cluster Configuration

- A cluster of 1800 machines, each has two 2GHz Intel Xeon, 4GB RAM

Grep

- The computation rate picks up as more machines are assigned to the computation

Sort

- The partitioning function has built-in knowledge of the distribution of keys
- Add a pre-pass MapReduce operation that would collect a sample of the keys and use the distribution of sampled keys to compute the split point for the final sorting pass
- The input rate is higher than the shuffle rate and the output rate since they are read locally
- The shuffle rate is higher than the output rate since the output phase writes two copies of the sorted data (for reliability and availability)

Effect of Backup Tasks

- Increase of 44% in time though minimal write actions are made

Machine Failures

- Increase or 5% of total time

6. Experience

Large-Scale Indexing

- Use for the production indexing system that produces the data structures used for the Google web search service
- See paper for more notes

Answers to Questions

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.

2.

- 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.

3.

- 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.

Additional Questions and Observations

1. Something that sets MapReduce apart is that it simplifies the process of implementing batch processing. MapReduce creates an abstract layer that helps users easily code up and process a large amount of data together. It streamlines productivity, which is surely helpful in regard to big data.
2. I am more interested in knowing why Google has decided to quit MapReduce and switch to an entirely new system. I actually looked up the two authors of this paper and realized that they are some of the people who have revolutionized the way big data is processed. I reckon that there must be some drawbacks that are so significant that they have to ditch an entire system that takes time for them to implement and move to a new one.