CS 138: Google

This material is covered in the textbook in Chapter 21.

Google Environment

- Lots (tens of thousands) of computers
  - all more-or-less equal
    - processor, disk, memory, network interface
  - no specialized servers
  - even if only .01% down at any one moment, many will be down

Google File System

- **Not your ordinary file system**
  - small files are rare
  - large files are the rule
    - typically 100 MB or larger
    - multi-GB files are common
  - reads
    - large sequential reads
    - small random reads
  - writes
    - large concurrent appends by multiple clients
    - occasional small writes at random locations
  - high bandwidth better than low latency

"The Google File System" paper, by S Ghemawat, H Gobioff, and S-T Leung, was published in the proceedings of the ACM Symposium on Operating Systems Principles in October 2003 and may be found at http://labs.google.com/papers/gfs-sosp2003.pdf.

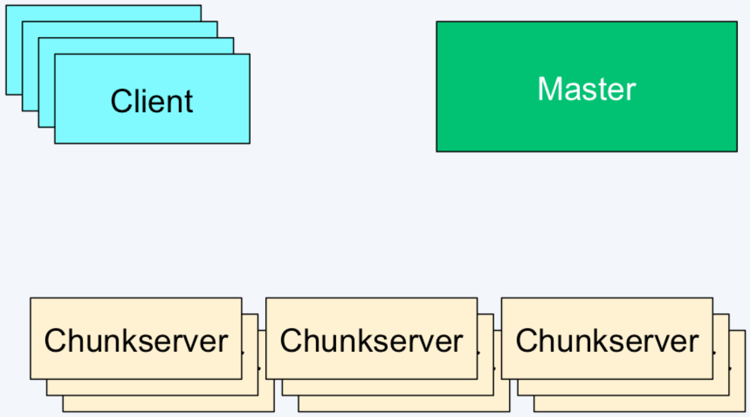
Some Details

- GFS master computer holds metadata
  - locations of data
  - directory
- Files split into 64-MB chunks
  - each chunk replicated on three computers (chunkservers)
  - master assigns chunks to chunkservers
    - does load balancing
    - takes into account communication distance from clients

More Details

- Fault tolerance
  - chunkserver crash
    - master re-replicates as necessary from other chunkservers
  - master crash
    - restart is quick
    - if not possible (disk blew up), a backup master takes over with checkpointed state

Architecture



Master and Chunkservers

- Master assigns chunks to chunkservers
  - assignment kept in volatile storage on master
  - … on disk on chunkservers
    - in Linux local file system
  - master recovers assignment from chunkservers on restart
- Master and chunkservers exchange *heartbeat* messages
  - keep track of status
  - exchange other information

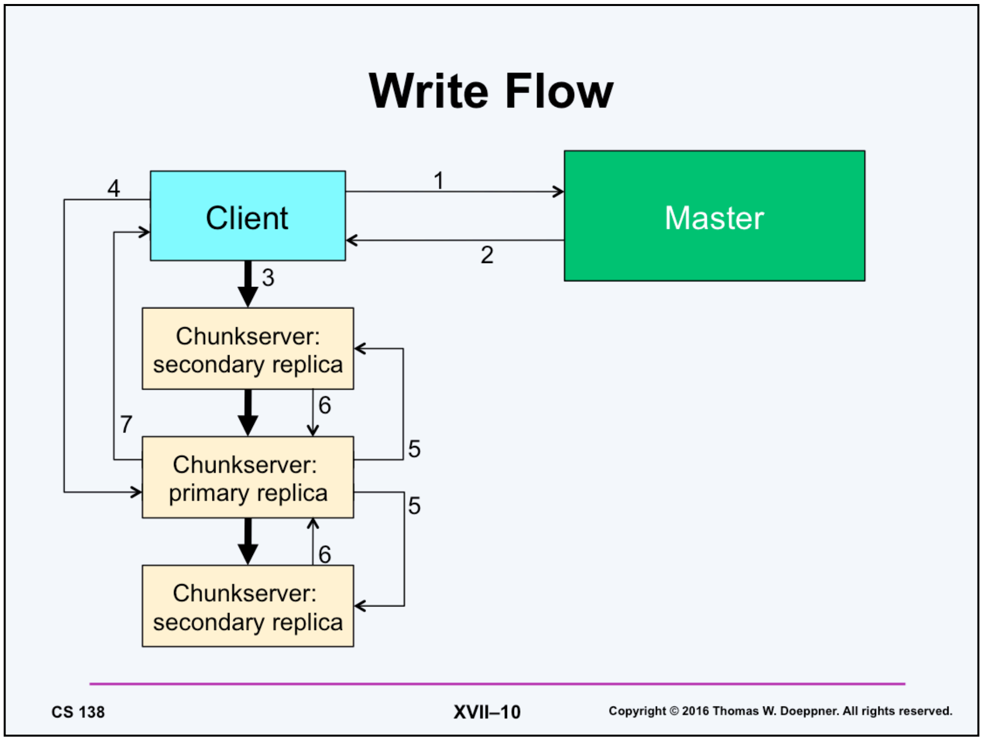
Issues

- Consistency
  – all replicas are identical*
- Atomicity
  – append operations are atomic, despite concurrency

*for a suitable definition of identical:
hold the same data, modulo duplicates

Chunks and Mutations

- Operations that modify chunks are "mutations"
- When a chunk is to be mutated, master grants one replica a *lease*
  - that replica becomes the *primary*
  - it determines order of concurrent mutations
    - assigns serial numbers
  - lease lasts 60 seconds
  - can be extended via heartbeat messages



This slide and the discussion below are from the aforementioned paper, "The Google File System."

1)"The client asks the master which chunkserver holds the current lease for the chunk and the locations of the other replicas. If no one has a lease, the master grants one to a replica it chooses (not shown).

2)"The master replies with the identity of the primary and the locations of the other (*secondary*) replicas. The client caches this data for future mutations. It needs to contact the master again only when the primary becomes unreachable or replies that it no longer holds a lease.

3)"The client pushes the data to all the replicas. A client can do so in any order. Each chunkserver will store the data in an internal LRU buffer cache until the data is used or aged out. By decoupling the data flow from the control flow, we can improve performance by scheduling the expensive data flow based on the network topology regardless of which chunkserver is the primary.

4)"Once all the replicas have acknowledged receiving the data, the client sends a write request to the primary. The request identifies the data pushed earlier to all of the replicas. The primary assigns consecutive serial numbers to all the mutations it receives, possibly from multiple clients, which provides the necessary serialization. It applies the mutation to its own local state in serial number order.

5)"The primary forwards the write request to all secondary replicas. Each secondary replica applies mutations in the same serial number order assigned by the primary.

6)"The secondaries all reply to the primary indicating that they have completed the operation.

7)"The primary replies to the client. Any errors encountered at any of the replicas are reported to the client. In case of errors, the write may have succeeded at the primary and an arbitrary subset of the secondary replicas. (If it had failed at the primary, it would not have been assigned a serial number and forwarded.) The client request is considered to have failed, and the modified region is left in an inconsistent state. Our client code handles such errors by retrying the failed mutation. It will make a few attempts at steps (3) through (7) before falling back to a retry from the beginning of the write."

Data Flow

- Independent of control flow
  - client sends data to nearest replica
  - replica sends data to nearest remaining replica
  - etc.
  - data is pipelined

Atomic Record Appends

- Data appended to end of file
  - atomic in spite of concurrency
  - must fit in a chunk
    - limited to ¼ chunk size
    - if doesn't fit, chunk is padded out to chunk boundary and data put in next chunk
      - applications know to skip over padding

Append Details

- Client pushes data to all replicas
- Client issues record-append request to primary
- Primary checks to make sure data fits in chunk
  - if not, primary deletes data and adds padding, tells secondaries to do likewise, tells client to start again on next chunk
  - otherwise primary writes data at end of file and tells secondaries to do likewise at same file offset

More Details

- Append could fail at a replica
  - perhaps replica crashed
  - new replica enlisted
- Client retries operation
  - duplicate entry at replicas where original succeeded
    - client must detect duplicates

Snapshots

- Quick file snapshots using copy-on-write
  – snapshot operation logged
  – leases recalled
  – metadata copied
    - reference count on chunks incremented
  – first mutation operation on each chunk causes a copy to be made
    - reference count of original decremented

Replica Placement

CS 138 XVII–16

The discussion below is copied from the aforementioned paper, “The Google File System.”

“A GFS cluster is highly distributed at more levels than one. It typically has hundreds of chunkservers spread across many machine racks. These chunkservers in turn may be accessed from hundreds of clients from the same or different racks. Communication between two machines on different racks may cross one or more network switches. Additionally, bandwidth into or out of a rack may be less than the aggregate bandwidth of all the machines within the rack. Multi-level distribution presents a unique challenge to distribute data for scalability, reliability, and availability.

“The chunk replica placement policy serves two purposes: maximize data reliability and availability, and maximize network bandwidth utilization. For both, it is not enough to spread replicas across machines, which only guards against disk or machine failures and fully utilizes each machine’s network bandwidth. We must also spread chunk replicas across racks. This ensures that some replicas of a chunk will survive and remain available even if an entire rack is damaged or offline (for example, due to failure of a shared resource like a network switch or power circuit). It also means that traffic, especially reads, for a chunk can exploit the aggregate bandwidth of multiple racks. On the other hand, write traffic has to flow through multiple racks, a tradeoff we make willingly.”

Chubby

- Coarse-grained distributed lock service
- File system for small files
- Election service for determining primary nodes
- Name service

Chubby is discussed in http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en/us/archive/chubby-osdi06.pdf. These bullets come from the textbook, page 940.

Lock Files

- File creation is atomic
  - two processes attempt to create files of the same name concurrently
    - one succeeds, one fails
- Thus the file is the lock

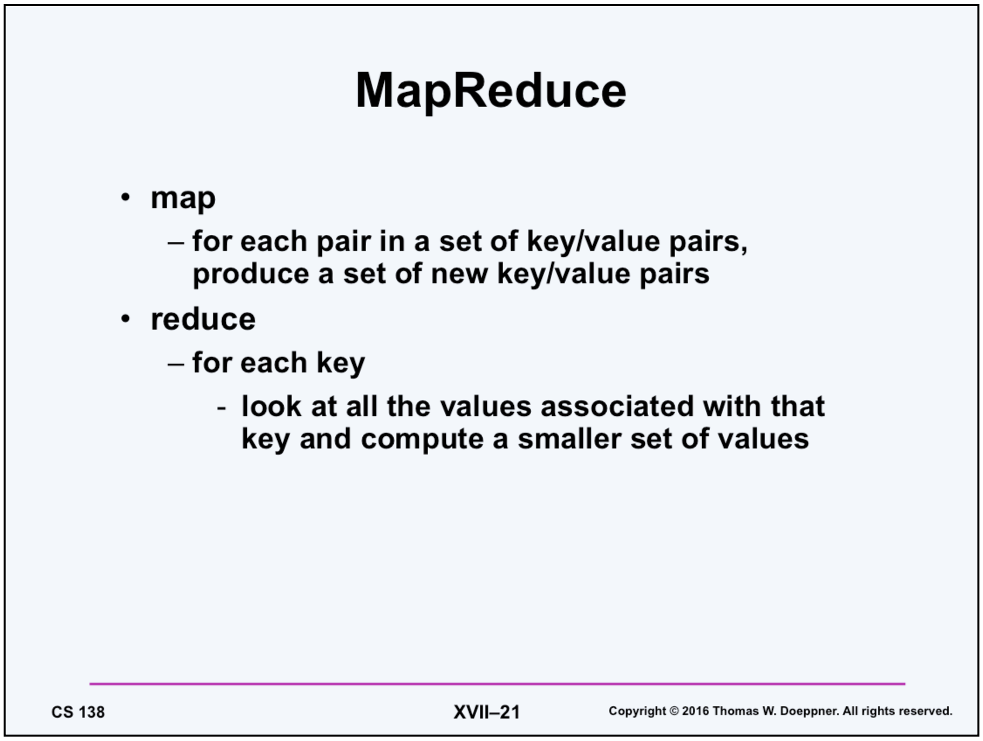
This is approximately what Chubby does. For exact details, see the textbook.

Electing a Leader

- Participants vie to create /group/leader
  - whoever gets there first, creates file and stores its ID inside
  - others see that file exists and agree that the value in the file identifies the leader

More

- Processes may register to be notified of file-related events
  - file contents are modified
  - file deleted
  - etc.
- Caching
  - clients may cache files
  - a file’s contents aren’t changed until all caches are invalidated



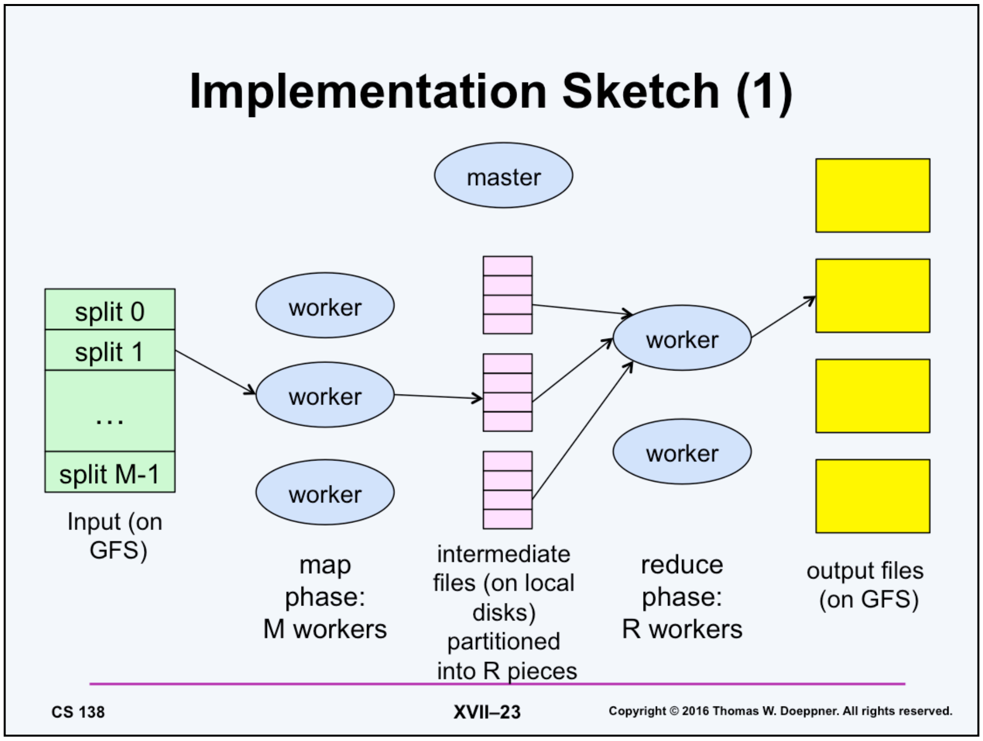
A paper discussing MapReduce can be found at http://labs.google.com/papers/mapreduce.html.

Example

```
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
   for each v in values
      result += v;
   Emit(result);
}
```


This example is from the aforementioned paper. It counts the number of occurrences of each word in a collection of documents.



This slide is taken from the aforementioned paper on map reduce.

Implementation Sketch (2)

- Map's input pairs divided into *M* splits
  - stored in GFS
- Output of Map/Input of Reduce divided into *R* pieces
- One master process is in charge: farms out work to W (<< M+R) worker machines

Implementation Sketch (3)

- Master partitions splits among some of the workers
  – each worker passes pairs to user-supplied map function
  – results stored in local files
    - partitioned into pieces
      - e.g., *hash(key) mod R*
  – remaining workers perform reduce tasks
    - the R pieces are partitioned among them
    - place remote procedure calls to map workers to get data
    - put output in GFS

Distributed Grep

- Map function
  - emits a line if it matches pattern
- Reduce function
  - identity function

Count of URL Access Frequency

- Map function
  - processes logs of web-page requests
  - emits: <URL, 1>
- Reduce function
  - adds together all values for same URL
  - emits <URL, total count>

Reverse Web-Link Graph

- Map function
  - outputs <target, source> for each link to target URL found in source
- Reduce function
  - concatenates list of all source URLs associated with given target
  - emits <target, list(source)>

Distributed Sort

- Map function
  - extracts key from each record
  - emits <key, record>
- Reduce function
  - emits all pairs unchanged
    - depends on partitioning properties to be described

Details (1)

1) Input files split into M pieces, 16MB-64MB each
2) A number of worker machines are started
   - master schedules M map tasks and R reduce tasks to workers, one task at a time
   - typical values:
     - M = 200,000
     - R = 5000
     - 2000 worker machines
3) Worker assigned a map task processes the corresponding split, calling the map function repeatedly; output buffered in memory

Details (2)

4) Buffered output written periodically to local files, partitioned into R regions by *partitioning function*
   - locations sent back to master
5) Reduce tasks
   - each handles one partition
   - accesses data from map workers via RPC
   - data is sorted by key
   - all values associated with each key are passed collectively to reduce function
   - result appended to GFS output file (one per partition)

Committing Data

- Map task
  - output kept in R local files
  - locations send to master only on task completion
- Reduce task
  - output stored on GFS using temporary name
  - file atomically renamed on task completion (to final name)

Coping with Failure (1)

- Master maintains state of each task
  - idle (not started)
  - in progress
  - completed
- Master pings workers periodically to determine if they're up

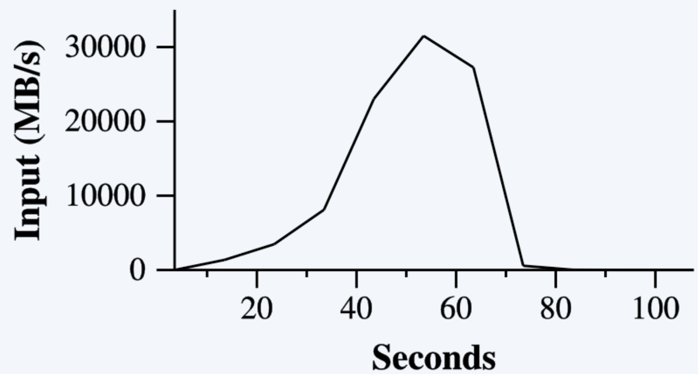
Coping with Failure (2)

- Worker crashes
  - *in-progress* tasks have state set back to idle
    - all output is lost
    - restarted from beginning on another worker
  - *completed* map tasks
    - all output lost
    - restarted from beginning on another worker
    - reduce tasks using output are notified of new worker

Coping with Failure (3)

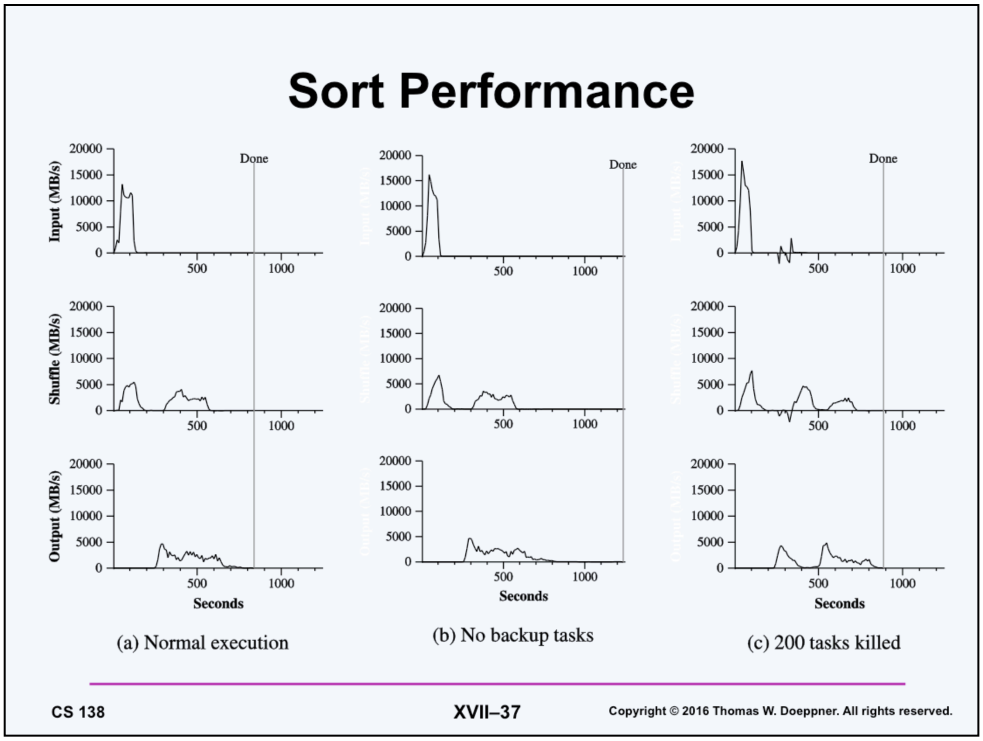
- Worker crashes (continued)
  - completed reduce tasks
    - output already on GFS
    - no restart necessary
- Master crashes
  - could be recovered from checkpoint
  - in practice
    - master crashes are rare
    - entire application is restarted

Performance: Data Transfer Rate



- grep through 10^{10} 100-byte records for rare 3-character pattern

This figure is from the aforementioned paper on map reduce. 15,000 map tasks and 1 reduce task were used. 1800 computers were employed.



This figure is also from the aforementioned paper. It shows the results of sorting 10^{10} 100-byte records, with 15,000 map tasks and 4000 reduce tasks running on 1800 computers. The top row shows the rate at which input is read; the second row shows the rate at which data is transferred from map tasks to reduce tasks, and the last row shows the rate at which final output is produced. The "Normal execution" column refers to the use of extra "backup tasks" which are used to deal with stragglers: a few map or reduce tasks take far longer than others, perhaps because of hardware problems. When a MapReduce application is close to completion, the master schedules a few backup tasks to execute the remaining in-progress tasks redundantly. The outputs of whichever finish first — the backup tasks or the original tasks — are used first. The middle column shows how useful this is: without the backup tasks it took far longer to run. The last column shows how quickly the system dealt with the loss of 200 tasks.

Counterpoint

- See
  http://www.databasecolumn.com/2008/01/mapreduce-a-major-step-back.html
  - MapReduce: the opinion of the database community:
  1) a giant step backward in the programming paradigm for large-scale data intensive applications
  2) a sub-optimal implementation, in that it uses brute force instead of indexing
  3) not novel at all — it represents a specific implementation of well known techniques developed nearly 25 years ago
  4) missing most of the features that are routinely included in current DBMS
  5) incompatible with all of the tools DBMS users have come to depend on

A more recent version of these arguments, taking into account the response of the map-reduce people (next slide), can be found at http://database.cs.brown.edu/papers/stonebraker-cacm2010.pdf.

Countercounterpoint

1) MapReduce is not a database system, so don't judge it as one
2) MapReduce has excellent scalability; the proof is Google's use
3) MapReduce is cheap and databases are expensive
4) The DBMS people are the old guard trying to defend their turf/legacy from the young turks

These points are from http://www.databasecolumn.com/2008/01/mapreduce-continued.html.