GDC MapReduce Notes

Josh Snider

October 26, 2015

FP Review

Data structures are not modified, new data structures are made instead. This means we don't need to lock data structures. It also means that since we don't have side effects, we make everything a separate thread and put them together at the end.

Functional programming also allows you to use functions as arguments as the name implies. Functional programming is also good for generic yet typesafe programming, but languages like OCaml can still screw this up.

Standard functional programming functions:

- Map - Apply a function to each element of a list and return a list made my concatenating the results together.
- Fold - Track the result of calling f on each element and an accumulator, return the final accumulator. Can be either a left or right fold.

These are basically the Map and Reduce in MapReduce.

MapReduce

Motivated by large scale data processing, parallelize across thousands of computers, and make it easy for programmers to use. MapReduce has built-in fault tolerance and network monitoring. The user basically has two functions to provide

- map (in_key, in_value) -> (out_key, intermediate_value) list - Input consists of records from various sources.
- reduce (out_key, intermediate_value list) -> out_value list

After the map phase happens, we combine all the intermediate values for a given key into one list and then start the reduce phase. It's good practice for reduce to only have out_value. We use barrier synchronization, to prevent people from going to the reduce step while people are still mapping.

Optimization

Each map function runs parallelly and the reduction for each key runs in parallel. This means we have a bottleneck waiting for mapping to finish before we start reduction. How do we optimize this?

- Slow-moving "map" tasks are run on multiple hosts and then we take the first one to finish.
- Google considered making reducers lazy, but that has some design flaws that made it not used.
- We can have "combiners" which run mini-reduce phases before the actual reduce in order to save bandwidth. If our reduce is both associative and commutative, then we can use it as our combine as well.

Written in C++, with Java and Python bindings. The dividing program tries to divide up map tasks so that mappers are on the same computer as the data. Tasks are chunked in 64MB which is the same size as the Google File System's chunks.

Ensuring fault tolerance

- We give up on tasks if certain key-value pairs reliably crash.
- Mappers that fail before reporting results are redone.
- Reducers don't claim to be done, until their results are reliably backed up.

Impact

Has had great results at Google and is a shining jewel of functional programming. Greatly simplifies large-scale computations. Lets programmer focus on problem and let library handle messy details.

File Systems

- A filesystem is a system to join data in a tree of files, by writing them to disk. Sometimes, they can support remote files or local caching.
- Folders are the same as namespaces.
- A "distributed filesystem" is one that can reach files on other machines.
- Any decent "distributed filesystem" guarantees that many people can access many files simultaneously. It needs to be performant at the same time and maintain data integrity / consistency.

NFS

- Made by Sun in the 80s. Pretty standard Unix FS. We mount remote drives onto local host.
- Original was completely stateless. Higher-level protocols handle that stuff.
- NFS defines a virtual filesystem (like an interface, not an implementation).
- NFS locking is done with leases. Clients request locks. Servers tell them if that succeeds or not and take back locks that aren't renewed.
- When we close a file, we push changes. When we open a file, we pull changes.

- An NFS volume is managed by a single server. This makes concurrency easier, but puts stress on it.
- POSIX compliance makes it portable, but very generalized.

GFS

- Google needed a way to redundantly store data on cheap and unreliable systems. Also, wanted to optimize it for Googly purposes.
- Assumptions:
  - Components suck.
  - Modest number of HUGE files.
  - Files are write-once. Mostly appending to data.
  - Want to read large serial chunks.
- Design Decisions:
  - 64 MB fixed-size chunks.
  - Each thing replicated on 3+ chunk servers.
  - Master coordinates access and stores metadata.
  - No data caching (files are too big).
  - Interfaces customized for Google, but POSIX-esque.
- Master server is a single point of failure. We solve this by having backups of the master called "shadow masters".
- What metadata does the master have?
  - File and chunk namespaces.
  - Mapping from files to chunks.
  - Locations of chunk's replicas.
  - An operations log for metadata updates.
- Except for the operations log, the metadata is super small and can stay in RAM. The operations log needs to be stored on disk for robustness reasons.
- We can use the operations log to restore the master to a good state if it becomes corrupted.
- GFS Mutation = write or append. Our goal for mutations is to minimize master involvement.
- We use a lease mechanism. The master picks a replica as primary and gives it a lease for mutations, the primary defines a serial order for the changes, and the replicas follow it.
- Atomic record append - GFS lets us append a record to a chunk atomically. Message is at-least-once, so we may do it multiple times, and order may differ between replicas.
- So, we have a relaxed consistency model, except for our master's metadata. Google's fine with that, but it's something to keep in mind.

- What are the master's responsibilities?
  - Metadata storage
  - Namespace management/locking.
  - Monitor system health.
  - Chunk creation, re-replication, and rebalancing.
  - Garbage collection - We delete things by renaming them to something hidden, then we do the actual clean up when we're not busy.
  - Prompt people with out of date chunks to update them.
- This is a highly fault tolerant system, with high availability and specified data integrity.
- In conclusion, sometimes simple solutions are good and you should expect hard drives to break.

Clustering Algorithms

- Clustering is partitioning data into subsets so that the members of each are close in some sense.
- Google News clusters by odds that two stories are the same story. We can alsop cluster hospital records, scientific images, survey data etc.
- Some common distance metrics are Euclidean distance, Manhattan distance, maximum norm, but you can always define your own.
- Clustering can either be
  - Hierarchical - building a tree (dendogram) of clusters either by division or by agglomeration.
  - Partitioning - Divide set into all clusters simultaneously.
- K-means clustering is a form of partition clustering. This can be expensive and hard to do with MapReduce.
- Canopy clustering is a preliminary step to clustering to make it easier to parallelize. We divide the stuff into canopies that are close. This works best if our canopy distance is super cheap compared to our K-means distance.
- The Elbow Criteria is a way to estimate the number of clusters in a thing.

Graph Algorithms

- How can we do a BFS in MapReduce? Iterate passes through MapReduce. Map some nodes, result include additional nodes which are fed into successive MapReduce passes.
- Problem: Sending entire graph to mapper is expensive. Solution: Carefully consider how we represent graphs.

  - The most straightforward representation is for each node to have a list of its neighbors. This doesn't make iteration easy and is hard to serialize.
  - Adjacency matrix (possibly as a sparse matrix).

- Finding the shortest path can be done easily on one machine with Dijkstra's algorithm, but is highly serial. How can we parallelize this with MapReduce?
  1. A map task receives $\{n : (D, points - to)\}$ n is a node, d away from start, and points to the guys in $points - to$.
  2. Map returns $(p, D + 1)$ for p in $points - to$.
  3. Reduce takes this and finds the minimum.

  Each MapReduce moves one layer further.

- First, iteration is fast, but then we rapidly blow up. We eventually terminate because we don't recalculate for a node unless we find a faster way to it.
- Adding edge weights its trivial.
- Complexity is at most n times as hard as Dijkstra which works out well if we have n machines.

PageRank

- If a user clicks random links and every so often goes to a random page, what are the odds they will eventually be at a given page? This is used as a measure of site importance.
- The MapReduce algorithm to compute this is similar to the shortest path where we keep track of PR_I after I steps.
- PageRank doesn't necessarily converge, but gets within a few percent accuracy after a few hours.

Phase 1: Parse HTML - Map takes $(URL, page - content)$ and spits out $(URL, (PR_{init}, list - of - urls))$. Reduce is id.

Phase 2: PageRank Distributions - Compute PageRanks of a website's children. (Number of mappers = number of nodes)

Phase 3: Terminate? If we've sufficiently converged, do so. Otherwise, go back to phase 2.

- This isn't the actual algorithm used by Google, because it doesn't scale to the very high end.