Distributed Systems: Assignment 2

Saurabh Mathur

February 10, 2019

MapReduce: Simplified Data Processing on Large Clusters

Jeffery Dean and Sanjay Ghemawat

Summary

This paper tackles the problem of efficiently processing a large amount of data. There are two approaches to such a problem - we can either use a single computer with a large enough memory and a fast enough processor or we can use a cluster of many computers of moderate hardware specifications. The authors focus on the second approach. They describe an abstract programming model and an associated distributed implementation that allows a user to run distributed computations without having to worry about the details of parallelization.

The programming model is inspired by functional programming and consists mainly of two functions - map and reduce. Specifically, the mapper function takes in a key-value pair and outputs a set of key-value pairs and the reducer function groups together the intermediate key-value pairs and outputs a smaller set of pairs. The key idea here is that the intermediate values are sent to the reducer using an iterator. This allows handling of lists too big to fit in memory.

The implementation described in this paper assumes a cluster of thousands of commodity hardware machines connected via commodity networking hardware. So, machine failures are common. One machine in the cluster is denoted as the master node and the rest as worker nodes. The implementation requires three pieces of user code to work - a splitting function that splits the input into roughly 16-64MB chunks, the mapper function and the reducer function. The master node uses the splitting function to split the data and assigns each chunk to a worker node as a map task. All map tasks run in parallel and when all of them finish, their output is sorted and grouped by key and passed to other worker nodes as reduce tasks. The reducer function is applied on the grouped data and the final output from each reducer is sent to the master node.

Apart from this, following practical considerations also come into play -

- *Worker Failure:* Master pings each worker periodically and re-executes failed tasks. This is possible because the mapper and the reducer are idempotent.
- *Master Failure:* Master's state is checkpointed periodically and the master is restored from the most recent checkpoint on failure.
- *Locality:* Tasks are assigned to machines that are as close to the data as possible to minimize network latency.
- *Task Granularity:* A larger the number of map tasks with each worker having multiple tasks corresponds to more efficient recovery from worker failures. However, this must be balanced against the number of scheduling decisions that the master will have to make.
- *Stragglers:* Multiple copies of the same tasks are run on different machines and the output of the first one to finish is used. This prevents a single unusually slow task from slowing down the entire job.

The authors also describe some extensions of the MapReduce system -

- *Partitioning function:* The user is allowed to specify a custom function that partitions the output of the mapper and divides it among the reducers.
- *Combiner function:* If there is a lot of repetition in the key of mapper output, a combiner function is applied to each mapper's output. Typically, this is the same as the reducer function.
- *Bad records:* In some cases, it is acceptable to ignore a few records. So, if repeated failures are detected, the master node marks the problematic records to be ignored in the next re-execution.

While MapReduce was originally developed as a generic framework for internal distributed computing jobs at Google, it has been for a variety of tasks of similar scale while having an easy to use abstraction.

Observations

- The authors assume that all the machines in the cluster are homogeneous which might not be the case for many distributed computing clusters outside Google.
- It is unclear how the number of workers is determined.
- While the rest of the paper assumes idempotent and deterministic mapper and reducer functions, the authors briefly talk about the case where the functions are non-deterministic. However, it is unclear how the guarantees would be affected by the functions being non-deterministic.

The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Summary

This paper tackles the problem of large scale data storage for a large number of clients. The authors describe the Google File System which supports only the bare minimum operations while providing high fault tolerance and concurrent access. The GFS works on assumptions that are very similar to MapReduce - large amount of data, high availability of commodity hardware and high failure rate. Apart from these, GFS also assumes that most file writes are appends and reads are either large streaming reads or small reads in random order which can be batched together. GFS has a very minimal interface. It supports create, delete, open, close, read and write operations. It also supports snapshot and record append operations. The record append operation is atomic.

A cluster consists of a master node and many chunk-servers. Each file is split into many chunks and each chunk is replicated on several chunk-servers. The file-chunk replicas are validated using checksums. The master node maintains the mappings and locations of all the file-chunks in memory. It directs clients to the appropriate chunk-server based on this mapping. This information is collected by polling the chunk-servers.

Write operations are implemented using *Leases*. The master grants a chunk-server lease to a chunk and marks the server as primary. The server now has a fixed amount of time to decide an order of write and propagate it to other replicas. Leases initially have a timeout of 60 seconds but the chunk-server can request extensions. This minimizes management overhead at the master node. The authors also discuss the following practical considerations -

- *Data Flow:* Each machine transfers data to the machine that

is closest to it and has not received the data.

- *Snapshots:* Copy-on-write operations are used to implement snapshots. The master prevents any writes from disturbing the snapshot by revoking an issued leases.
- *Replica Placement:* Replicas are placed not only to maximize reliability but also to maximize availability and bandwidth utilization. Thus, replicas are placed across racks.
- *Chunk Placement:* Replicas are placed on chunk-servers with below average disk space utilization to equalize disk utilization. Also, the number of recently created chunks for a given server should be limited as creation is usually followed by heavy write traffic.
- *Namespace management:* The locking mechanism is very fine-grained and all operations require very specific locks. Thus, concurrent mutations are possible in the same directory.
- *Garbage collection:* Deleted files are not removed from the filesystem. They are renamed to a hidden name and deleted after 3 days. This allows reversal of accidental deletions.

Observations

- GFS does not implement a standard file system interface.
- The master could be a bottleneck if there are very frequent accesses. Also, the metadata is stored in memory in the master. The size of master's memory could place an upper limit on the number of chunks that can be saved.
- Distances (to optimize data flow) are measured using IP Addresses. This might not be a good heuristic if we do not have full control over network topology.

The Hadoop Distributed File System

Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler

Why

I chose this paper because Hadoop is open-source and as a result the paper can cover the actual implementation details in much greater depth than GFS.

Summary

Hadoop is a collection of open-source software component that allow storage of large amounts of data and computations on such data by distributing the load among a cluster of machines using the MapReduce programming model. While Google's implementation of MapReduce is supported by the Google File System (GFS), Hadoop's MapReduce is supported by the Hadoop Distributed File System (HDFS). While GFS served as the primary inspiration for this work, this work also focused on adapting the same ideas for a more general purpose computing framwork and making fewer assumptions about the data and operations.

Since I have already discussed GFS in detail above, I will discuss some key places where HDFS diverges from the GFS design.

- *Architecture:* While the general architecture is same as GFS, the naming and implementation are different. NameNode and DataNode are analogous to Master and Chunk-Servers in GFS.
- *File Structure:* Each file is split in blocks of fixed size. The default is 128MB but the size is configurable.
- *Data Flow:* HDFS directly exposes block locations and leases are granted directly to the client. However, data is not guaranteed to be visible to other clients unless the file is

closed. Alternatively, the *hflush* operation can be called to ensure that the changes are synchronized.

- *Chunk placement:* Disk space utilization at each DataNode is not considered when placing a block. While this avoids a small set of nodes getting a lot of traffic, the data is not guaranteed to be placed uniformly across nodes. So, a balancer tool is provided that can be configured with a threshold value and deployed as an application program.
- *Replica policy:* The replica placement policy in this paper is defined more clearly than in GFS - no node can have more than one replica of any block and no rack can have more than two replicas of same block.
- *POSIX Standard:* While HDFS does not implement the full standard, it does have some of its features modeled after the POSIX standard. The file permissions scheme is an example of this.

Observations

- While it does diverge from GFS at some places, HDFS still has the same shortcoming of the NameNode being a bottleneck.
- While running the load balancer as a separate program make the system more configurable, this would be at the cost of performance and bandwidth usage.

Improving MapReduce Performance in Heterogeneous Environments

Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica

Why

I chose this paper because I had the same question about how would MapReduce perform in a cluster in which all machines do not have the same hardware.

Summary

One of the assumptions of the MapReduce implementation (as discussed above) is that all the nodes are homogeneous, that is, they have similar hardware configurations and all tasks progress linearly. This assumption is key when deciding which node is a *straggler* and running backup tasks for such nodes. However, this assumption does not hold in practice. This paper tackles this problem by proposing a scheduling algorithm called Longest Approximate Time to End (LATE).

The authors make specific arguments for each assumption made by the Hadoop scheduler that breaks down in practice.

- *Homogenity:* The assumption that a detectably slow node is faulty breaks down as in practice, nodes can be slow for many reasons - differences in hardware, contention from other users, disk and network bottle-necking. Any of these could cause the node slow down. Further, fixed threshold based decision making could result in the scheduler launching too many speculative tasks or worse, it could also launch wrong tasks.
- *Backup tasks on idle nodes are free:* When resources are shared in a cluster, this could worsen performance at bottlenecks like the network bandwidth.

- *Progress score = fraction of work completed:*This implies that each phase like copy, sort and reduce roughly takes an equal amount of time. This is not true as a copy operation which is over the network could easily dominate the other phases.

LATE is implemented to pick tasks for speculative execution which are expected to complete the furthest in the future. LATE uses two heuristics to ensure that unnecessary backup tasks are not executed. The first is *SpeculativeCap* which is a cap on the maximum number of backup tasks that can be running at once. The second is *SlowTaskThreshold* which is used to judge if a task is slow enough to be speculated upon.

Observations

- The Hadoop task scoring scheme makes wrong assumptions and it is not fair. However, the LATE algorithm still uses this scheme.
- While the authors provide their values for the new threshold parameters that they have introduced, this still adds another set of parameters that need to be tuned for each cluster.

Map-Reduce for Machine Learning on Multicore

Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y. Ng, and Kunle Olukotun

Why

I chose this paper because it applied the MapReduce framework in a non-distributed system setting.

Summary

This paper is from 2007 when multi-core systems were becoming popular. However, most machine learning algorithms were single threaded. This paper approaches the task of parallelizing machine learning algorithms by adapting the MapReduce model for this task.

The authors talk about three classes of machine learning algorithms. The first is Leslie Valiant's Probably Approximately Correct (PAC) model in which the algorithm uses randomly drawn examples to learn the target function. The second is the Statistical Query Model by Michael Kearns in which the algorithm only has access to some type of aggregate over the examples and not the actual values. The authors talk about a third model, called the *summation form* that they developed. The authors claim that any algorithm that can be written in Statistical Query form can also be written in summation form.

The key idea in this paper is that we can think of each core as a worker node and perform the computation of aggregates by applying MapReduce. The aggregated result is then used to update the machine learning model. However, for some machine learning algorithms, additional scalar variables are needed in mapper and reducer functions. To allow this, a special query_info interface is provided which can be implemented specific to the algorithm. The

authors implemented many algorithms using this framework including Locally Weighted Linear Regression, Naive Bayes, Logistic Regression, Neural Networks and Expectation Maximization.

Observations

- The parallelization framework does not modify the algorithms. So, this is an exact implementation.
- The speed up scales linearly with the number of cores.