MapReduce

Prakhar Srivastav (ps2894)

Summary

The key goal of the MapReduce framework is to allow programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system. Mapreduce is an abstraction that allows programmers to express the simple computations they're trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. It is inspired by two key functions in functional programming languages - map and reduce.

Map operation - takes a set of logical "records" in and outputs a set of intermediate key/value pairs

Reduce operation - takes a set of values that share the same key and combines them to get the derived data appropriately.

A key difference between the systems in the past (eg. MPI) and MapReduce is that MapReduce exploits a restricted programming model to parallelize the user program automatically and to provide transparent fault-tolerance.

Implementation

One of the key goals of the MapReduce framework was to make it on commodity hardware which could be horizontally scalable. In Google's implementation all resources amongst the machines were off-the-shelf: 2-4gb ram, commodity networking hardware and inexpensive disks. Beneath the mapreduce layer, however, GFS (a distributed filesytem) is used on which all the data was stored in chunks. Lastly, a job scheduler is used to submit jobs to the mapreduce system.

Execution

The execution of a mapreduce job is quite simple. Primarily because the framework abstracts out most of the plumbing. The user is responsible for submitting a program which describes the map task and the reduce task. The MapReduce library first splits the file into a number of chunks (`M`) each of which can be processed in parallel on different machines. Once the split is complete, the library starts up many copies of the program on the cluster of machines. One of these machines is designated as a master which coordinates with other worker machines. There are `M` map tasks and `R` reduce tasks to assign.

After the `M` map tasks complete, the data (intermediate key-value) pairs are written to disk and partitioned into `R` regions by the partition function. When the reduce worker gets this data, it sorts it and then executes the reduce program on these key-value pairs to produce the final answer. This final answer then gets appended to a final output file which, on completion, is returned back to the user.

Fault Tolerance

As a typical MapReduce job typically involves hundreds of machines, failures are bound to happen. Since one of the design goal of the library was to hide the dirty distributed system issues with failures it implies that

the framework has to manage failures in a transparent manner so that job completions are ensured. More importantly, all of this has to happen without intervention of the programmer and completely transparently.

To keep a track of the running workers, the master server participates in pinging the workers for a health check. A worker, is in a *idle* state by default and hence is ready to accept jobs.When a map task is executed first by worker A and then later executed by worker B (because A failed), all workers executing reduce tasks are notified of the reexecution.

Lastly, to ensure the master does not become the single-point-of-failure, the master keep logging out periodic checkpoints. When the master dies, a new node can be spun up from the last checkpointed state. From a distributed systems point-of-view, the semantics that the system guarantees is -

> *The same output as would have been produced by a non-faulting sequential execution of the entire program.*

Other Aspects

Locality: The MapReduce library conserves network bandwidth by exploiting the fact that the input data is stored on the local disks of the machines. Hence, instead of bringing data to the computation it bring computation to the data.

Backup Tasks: 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. This is to ensure that the task is not kept running for a long time due to a *straggler*.

Two More Functions

Apart from *map* and *reduce* there are two more functions that are employed in the MapReduce library - both of which can be customized by the programmer (if need be).

Partitioning Function: The partition function is responsible for partitioning the output of the map tasks based on the intermediate key. By default it uses a hash based scheme so that the distribution is balanced amongst workers but this can be supplanted by a custom written partition function depeneding on the data locality needs.

Combiner Function: The combiner function is an optional function that does partial merging of intermediate key-value data before it is sent over the network. The function is basically like a reduce task but it is run on the map worker so as to congregate data before spending precious bandwidth repeated / duplicate data over the network.

Key Takeaways

In essence, MapReduce is a framework for running computations on a cluster of machines that abstracts out (to a significant extent) the issues prevalant in a distributed system thereby freeing the programmer to only think about the problem at hand. A few takeaways of the authors as mentioned in the paper are as follows -

- Restricting the programming model makes it easy to parallelize and distribute computations and to make such computations fault-tolerant.
- Network bandwidth is a scarce resource. Hence the data locality optimization is essential as it allows the system to read data from local disks, and writing a single copy of the intermediate data to local disk saves network bandwidth.
- Redundant execution can be used to reduce the impact of slow machines, and to handle ma- chine failures and data loss.