Summary of MapReduce

MapReduce is a programming model and an associated implementation for processing and generating large data sets. The advantages of the MapReduce are not only providing an abstraction for complex data, but also hiding the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. To illustrate the idea of MapReduce, this methodology can be separated into three phases: map phase, intermediate phase, and reduce phase. This process can be expressed by the following:

Map (k , v) → [(k’ , v’), …]

Reduce (k’ , [v1,v2,…]) → [(k” , v”), …]

The detail execution of MapReduce is the following.

Step1-Initialization:

Split the input file into M pieces of typically 16 to 64 MB per piece. Then, it starts up many copies of the program on a cluster of machines.

Step2-Master:

The master is the special copy among the all of copies, while the rest are workers which are assigned by master. There are M map tasks and R reduce (See Step4) tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task. The master keeps several data structures. For each map task and reduce task, it stores the state (idle, in-progress, or completed). The master must make

O(M + R) scheduling decisions and keeps O(M \* R) state in memory as described above.

Step3-Map Worker:

Map workers read the content of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined Map function. The intermediate key/value pairs produced by the Map function are buffered in memory.

Step4-Partition:

The buffered pairs are written to local disk and partition into R regions. Then, we passed back the locations of these buffered pairs on the local disk to master, which will forward these locations to the reduce workers.

Step5-Reduce Worker:

When reduce worker receive the forwarding of locations from master, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys that all occurrences of the same key are grouped together.

Step 6-Iterate to Reduce:

The reduce workers iterate over sorted intermediate data and for each unique intermediate key encountered, and pass the key and the corresponding set of intermediate values to the user’s Reduce function. The output of the Reduce function is added to a final output file for reduce partition.

Step7-Termination:

When every map and reduce tasks completes, the master wakes up the user program. At the moment, the program returns backs to the user code. The output of MapReduce execution is available in the R output files.

The MapReduce library tolerate machine failures, including worker failure, master failure, and semantics in presence of failures.

1. Worker Failure

The master pings every worker periodically. If no response is received from a worker in a certain amount of time, the master marks the worker as failed. At the moment, map tasks completed or in-progress at worker are reset to idle. Idle tasks eventually rescheduled on other worker(s)

1. Master Failure

If the master task dies, a new copy can be started from the last check pointed state. However, given that there is only a single master, its failure is unlikely. Thus, MapReduce task is aborted and client is notified.

1. Semantics in presence of failures.

When the user-supplied 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.

Besides fault-tolerance, another optimization of MapReduce is to adopt backup task to reduce “straggler” effect. Straggler lengthens the time taking to do last few map or reduce tasks in the computation. To address this challenge, 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.

Last but not least, we have had the basic concept of MapReduce. There are following extensions of MapReduce to do refinement for data dealing: Partitioning Function, Ordering Guarantees, Combiner Function, Input and Output Types, Side-Effects, Skipping Bad Records, Local Execution, Status Information, and Counters.