MapReduce: Simplified Data Processing on Large Clusters

Key Takeaways

- This paper provides some fine details about the implementation of original map-reduce system.

How Reducers Work With Partitioning

- When each mapper finishes their job and generates intermediate key-value pairs, the mappers write the intermediate result into their own local storage. The storage is partitioned locally on each mapper into R partitions where R is the number of reducer, and each partition is prepared for future reading of corresponding reducer.
- The location of the partitions are passed back to the master, then the master forwards the location of files to corresponding reducer.
- The reducers use RPC to read the files from mappers, and at the end the intermediate result from mappers are stored in the memory of each reducer.
- Then the reducer sorts the intermediate keys so that the key-value pair with the same key is grouped together in the sorted file. If the file is too large to fit in the memory, then external sort is used.
- The reducer performs reduce by iterating through the sorted files, and the reduce is performed for all pairs with the same key. This can be done by scanning through the data because now the pair with the same key are grouped together.

Handle Worker Failure

- The master pings each worker independently. If no response is received from a worker in a certain amount of time, the master marks the worker as failed.
- Any map completed or is being processed by a failed worker is reset back to the initial idle state. The completed map job is lost because it is stored in the local storage of the failed worker. In contrast, if a reduce worker fails, the job it has completed does not need to be restarted because the result of reduce job is stored in a global file system.
- When a worker fails, all reducers using the data from the failed worker will be notified and they will be told to read the data from a new worker.
- The system does not handle the master failure: if master fails, the whole job is restarted. This is not too much problem because the number of master machine is small.
- In order to achieve deterministic behavior despite potential failure, the system relies on the atomic commits of map and reduce tasks.
- The map task completion is atomic because it saves the result in its own storage. After it sends the message of job completion to master, it atomically switches from in-progress to completed.
- The reduce task achieves atomicity by first saving the result in its own temporary storage. After it completes, it atomically rename the temporary file to the final file name. By relying on the atomic renaming files of the underlying file system, the atomicity of reducer job completion is achieved.

Locality Consideration

- Locality should be an important consideration because network bandwidth is scarce.
- The master tries to schedule the mapping job on a machine that already contains a replica of the data it should use. If that fails, it tries to schedule the job on a machine that is close (e.g. on the same rack) to the machine that contains a replica.

Task Granularity

- It is more ideal that the number of mapping and reducing jobs is much larger than the number of workers. In that case, each worker will perform multiple mapping and reducing job, and this helps with the dynamical balancing of workload, and also helps with recovery of failed job because now other machines have to redo its job and more machines can help with the redo when the number of job is large.

Backup Tasks

- The progress of the whole job can be delayed by a few machine that takes unusually long to finish their jobs.
- This problem can be alleviated by scheduling additional job that tries to finish the same job that has not finished when most of the jobs have already finished. Then if either the original or the newly scheduled job finishes, the whole map-reduce finishes.
- This method can greatly increase the performance by only increasing the burden by a few percent.

Skipping Bad Records

- Sometimes the data can contain bad records that causes crash of the jobs, and we can safely ignore the bad records e.g. when doing big data analysis.
- Each job has a signal handler. If the job fails due to certain record, it sends an UDP package to the master indicating which record causes it to crash.
- When the master receives multiple message about the failure of the same records, this record is considered bad and is skipped for all jobs.

Sanity Checking and Debugging

- It is helpful to include a local mode of execution that sequentially simulate the map-reduce framework. This local mode is useful for debugging.
- It is helpful to include a counter facility that can count the number of occurance of certain events, such as the number of words starting with letter “A”. This can be used to check if the behavior of the system follows user expectation.