Map Reduce Notes

MapReduce is a programming model and system for processing large data sets.

Overview

- The model is easy to use as it hides the details of parallelization, fault tolerance, data distribution, locality optimization, and load balancing. Programmers do not need experience with parallel or distributed systems to use MapReduce.
- A typical MapReduce computation processes many terabytes of data. Computations are often performed with 200,000 map tasks and 5,000 reduce tasks using 2,000 worker machines. Each task is about 16 MB to 64 MB of data.
- Machines are typically dual-processor x86 processors running Linux. Each machine has about 2 GB to 4 GB of memory.

Programming Model

- **Map.** Takes an input and produces a set of intermediate key/value pairs.
- **Reduce.** Accepts intermediate pairs and combines shared keys together to produce an output.

Design

Master

For each map and reduce task, the master stores the state (Idle, In-Progress, or Completed) and worker machine identity. The master assigns tasks to the workers. The master propagates intermediate file locations from map tasks to reduce tasks.

Execution Order

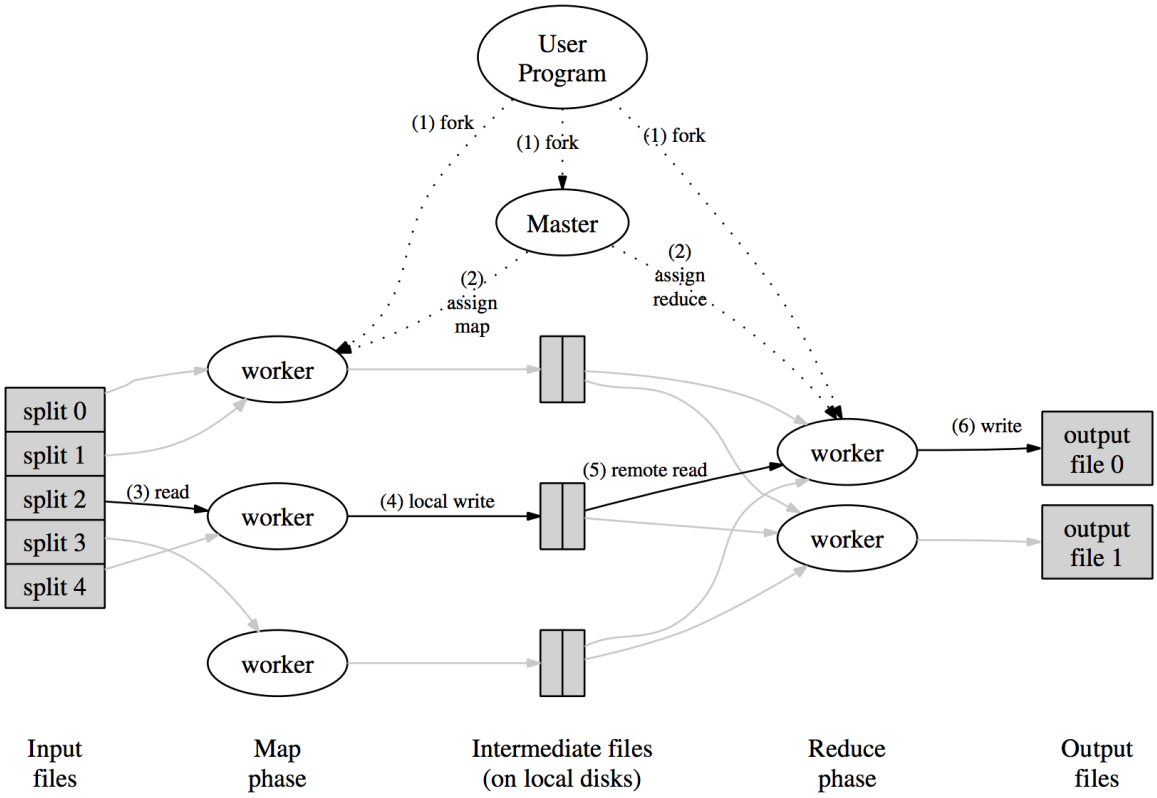

Figure 1: Execution overview

1. MapReduce library in the user program splits input files into *M* tasks. Copies of the program are started on a cluster of machines.
2. One of the program copies becomes the master. The rest become workers. The master assigns *M* map tasks and *R* reduce tasks to the workers.
3. Map workers read the contents split up from the input. It parses the input to generate input key/value pairs. Next, it passes each input pair into the user-defined Map function. This will output intermediate key/value pairs that are buffered into memory.
4. Periodically, a map worker writes buffered intermediate pairs to local disk across *R* partitions. The partitioning function is defaulted to `hash(key) mod R`. The map worker passes the disk locations of these pairs to the master. The master will forward these locations to the reduce workers.
5. Once a reduce worker is notified by the master of the locations for the buffered intermediate pairs, the reduce worker will use an RPC to read the data from the map workers' local disk. Once all intermediate data has been read, it sorts the data by intermediate keys. This will group all identical keys together.

6. The reduce worker passes each unique intermediate key and the corresponding set of values to the user-defined Reduce function. The output is appended to a final output file for this reduce partition.
7. When all map and reduce tasks have completed, the master returns the result to the user program. The output is available in 'R' output files, one per reduce task. Each of the file names were specified by the user prior to running the program.

Fault Tolerance

Worker Failure

- The master periodically pings every worker. When a worker fails to respond in time:
    - If the tasks have an In-Progress state, the master changes the worker's tasks to Idle. Next, the master reschedules the tasks onto other workers.
    - For the map tasks in a Completed state, the tasks need to be rescheduled because the output is on the failed machine's local disk--inaccessible.
    - For the reduce tasks in a Completed state, the tasks do not need to be rescheduled as the output is stored on GFS (Google File System).
- MapReduce is resilient to a large number of worker failures. In one example, network maintenance caused 80 machines to become unreachable. The MapReduce master rescheduled the tasks onto other machines and was able to successfully complete the MapReduce operation.

Master Failure

The master periodically writes checkpoints for its data structures. If the master dies, a new master will be started from the last checkpointed state. If this fails, the MapReduce computation is aborted and the client will have to retry the MapReduce operation.

Atomic Commits

- Write outputs by map and reduce tasks are atomic commits. In-Progress tasks write output to private temporary files.
- When a map task completes, the worker sends a list of the *R* temporary files to the master. Map tasks produce *R* output files, one per reduce task. The master records the *R* file names into its data structure.

- When a reduce task completes, the reduce worker atomically renames its temporary output file to the user-defined final output file. Reduce tasks produce one file.

Locality

- The MapReduce master gets input data location information from GFS. The master will attempt to schedule map tasks on a machine that contain the corresponding input data. If the attempt fails, the master will retry on another replica (GFS maintains data copies on at least 3 replicas).
- GFS's file chunk size of 64 MB correlates nicely with MapReduce's task size of 16 MB to 64 MB. All MapReduce tasks can be performed on one machine.
- A majority of input data is read locally and consumes no network bandwidth.

Task Granularity

The number of *M* and *R* tasks should be much larger than the number of worker machines. This allows for each worker to perform many different tasks. This improves dynamic load balancing and speeds up failure recovery.

Backup Tasks

- A *straggler* is a machine that takes an unusually long time to complete one of its last map or reduce tasks in a computation. Stragglers are the most common cause for significant delays to a MapReduce operation.
- When a MapReduce operation is near completion, the master schedules backup tasks that are copies of the remaining In-Progress tasks. The task is marked as Completed once either the primary or backup task completes.
- In a sort program example, MapReduce operations took 44% longer to complete without backup tasks enabled.

Examples

- **Number of Occurrences.** The map function processes documents and outputs `{word: 1}` pairs for each `word` found. The reduce function merges the pairs by `word`, sums up the values, and outputs a `{word: total count}` pair.
- **Count of URL Access Frequency.** The map function processes web page request logs and outputs `{url: 1}` pairs. The reduce function merges the pairs by `url`, sums up the values, and outputs a `{url, total count}` pair.
- **Reverse Web-Link Graph.** The map function parses each page and outputs `{target: source}` pairs for each `target` URL found in the page

named `source`. The reduce function merges the pairs by `target`, concatenates the list of `source` pages, and outputs a `{target: List(source)}` pair.
- **Inverted Index.** The map function parses each document and outputs a sequence of `{word: document ID}` pairs. The reduce function merges the pairs by `word`, sorts the corresponding `document ID`, and outputs a `{word: List(document ID)}` pair.

Example: Google Web Search Indexing System

The indexing system needs to regularly process more than 20 terabytes of raw data retrieved by Google's crawling system. Using MapReduce has provided several benefits:

- The indexing code became simplified and easier to understand as it doesn't need to support distribution, parallelization, or fault tolerance. One phase of the computation dropped from about 3,800 lines of code to about 700 lines of code.
- Time to implement changes to the indexing code went down. One change that took a few months in the old system only took a few days in the new system.
- The system became easier to operate as it didn't need to deal with problems associated with machine failures, slow machines, and networking issues.