the Availability Digest

How Does Google Do It?

February 2008

Search for "availability" on Google. How does Google give you 674,000,000 web references in 150 milliseconds? Through an extremely efficient, large-scale index of petabytes (that's millions of gigabytes) of web data.

A major problem Google faces is how to process this massive amount of worldwide data in a time that makes the indices useful. Google has solved this problem by building an equally massive parallel computing facility driven by its *MapReduce* function. MapReduce distributes applications crunching terabytes of data across hundreds or thousands of commodity PC-class machines to obtain results in minutes.

With MapReduce, Google programmers without any parallel and distributed systems experience can easily utilize the resources of these massive clusters. MapReduce takes care of partitioning the input data, scheduling the program's execution across the machines, handling balky and failed machines, balancing the load, and managing intermachine communications.

Google fellows Jeffrey Dean and Sanjay Ghemawat describe the MapReduce function in their paper entitled MapReduce: Simplified Data Processing on Large Clusters.[1] This Availability Digest article summarizes their description.

The MapReduce Programming Model

MapReduce is focused on solving problems in which it is desired to derive a set of *key/value pairs* from a massively large collection of data. "Key/value pairs" may sound a bit abstract, but some examples will illustrate their importance with regard to services such as those provided by Google.

A key/value pair is simply that – it is a key followed by a value. For instance, a simple example is one in which the key is a word and its value is the number of occurrences of that word in the data set.

A value can be a single number, a list of items, or a vector of items.

Other more meaningful examples include:

- For a particular URL (the key), a list of all web sites that point to that URL (the value).
- For a particular word (the key), a list of all documents that contain that word (the value).

[1] Jeffrey Dean, Sanjay Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, *Communications of the ACM*; January 8, 2008.
This paper can also be found at http://209.85.163.132/papers/mapreduce-osdi04.pdf.

- For a particular document (the key), a list of all words and their frequencies contained in the document (a vector value).

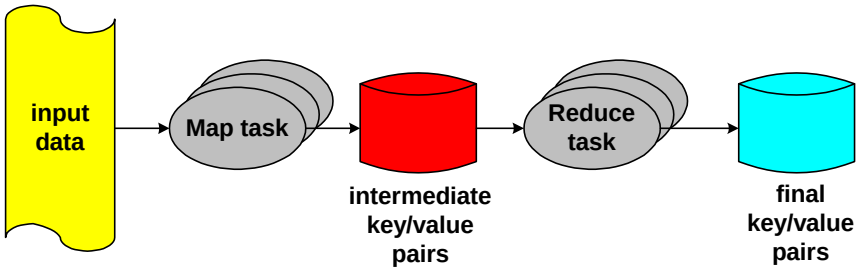
MapReduce operations can be strung together to create more complex data summaries.

The MapReduce programming model requires that the programmer write two fairly simple programs:

- A Map function that generates a set of intermediate key/value pairs from the input data set.

- A Reduce function that merges intermediate key/value pairs associated with the same intermediate key.

MapReduce automatically parallelizes these tasks and executes them on a large cluster of commodity machines. Many Map tasks are created, each working on a portion of the input data set. The Map tasks parse the input data and create *intermediate* key/value pairs. These are passed to the Reduce tasks, which merge and collate the results of the various Map tasks to generate the *final* key/value pairs.



The MapReduce Function

For instance, the simple word count example given above determines the number of instances of each word in the input data set. In this case, each Map function generates a key/value pair for each word in its portion of the input data set. This key/value pair is simply the word (the key) followed by the number “1” (the value). Each key/value pair is written to an intermediate key/value store.

The Reduce function reads the intermediate key/value store and merges by key all of the intermediate key/value pairs that have been generated by the Map tasks. In this simple case, the Reduce function will simply add up the number of key/value pairs for each word and will store the final count in the final key/value store. In the above example, an intermediate key/value might be “horse/1.” The final key/value might be “horse/356.”

In many applications, the Map task can do some initial summarization itself before passing the intermediate file regions to the Reducers, thus offloading some of the Reducer processing. This is done by creating a *Combiner* function in the Map task. For instance, in the above simple example, the Map task might aggregate the count for each key before writing it to the intermediate key/value store.

MapReduce automatically parallelizes and executes the Map and Reduce tasks. It partitions the input data, schedules the task executions across a set of machines, balances the load across these machines, handles balky and faulty machines, and manages intermachine communications. The complexities of the distributed system are completely hidden to the programmer. Programmers without any experience in parallel and distributed processing can easily utilize the resources of a large distributed system.

The MapReduce Implementation

The Cluster Hardware

The very large Google clusters used for running MapReduce applications can each contain thousands of commodity PC-class machines. Each machine is typically a dual-processor x86 machine running Linux. The machines are configured with two to four gigabytes of memory and with inexpensive, directly-attached IDE disks. The Google File System (GFS) manages the files on the directly-attached disks.

The machines in a cluster are connected by 100 megabit/sec. or 1 gigabit/sec. LANs.

The Program

The user program comprises the Map and Reduce tasks. They are typically fairly short and simple programs.

In addition to writing the Map and Reduce tasks, the user must specify a set of runtime parameters for MapReduce. These parameters include:

- the number of *splits*, *M*, into which the input data is to be split. One Map task will be created for each split. Thus, there are *M* Map tasks.

- the number of *intermediate file regions*, *R*, into which the intermediate key/value pairs are to be partitioned. The association of an intermediate key/value with an intermediate file region is done via a hashing algorithm [hash(key)mod *R*]. A standard MapReduce hashing algorithm can be used, or the user can specify his own algorithm. There will be one Reduce task created for each intermediate file region for a total of *R* reduce tasks.

- the number of PCs to use. The set of *M* plus *R* tasks is usually much greater than the number of specified PCs.

- the amount of memory that a task can use in a machine.

The MapReduce Infrastructure

Users submit their MapReduce jobs to a cluster scheduler. The scheduler creates *M* Map tasks and *R* Reduce tasks and assigns them to the user-specified number of machines in the cluster. Each machine can have multiple tasks assigned to it. These are the *worker* machines.

One special task is started on one machine in the cluster. It is the *Master* task. The Master task:

- stores the state (idle, in-progress, completed, failed) of each Map and Reduce task and the machine to which it is assigned.

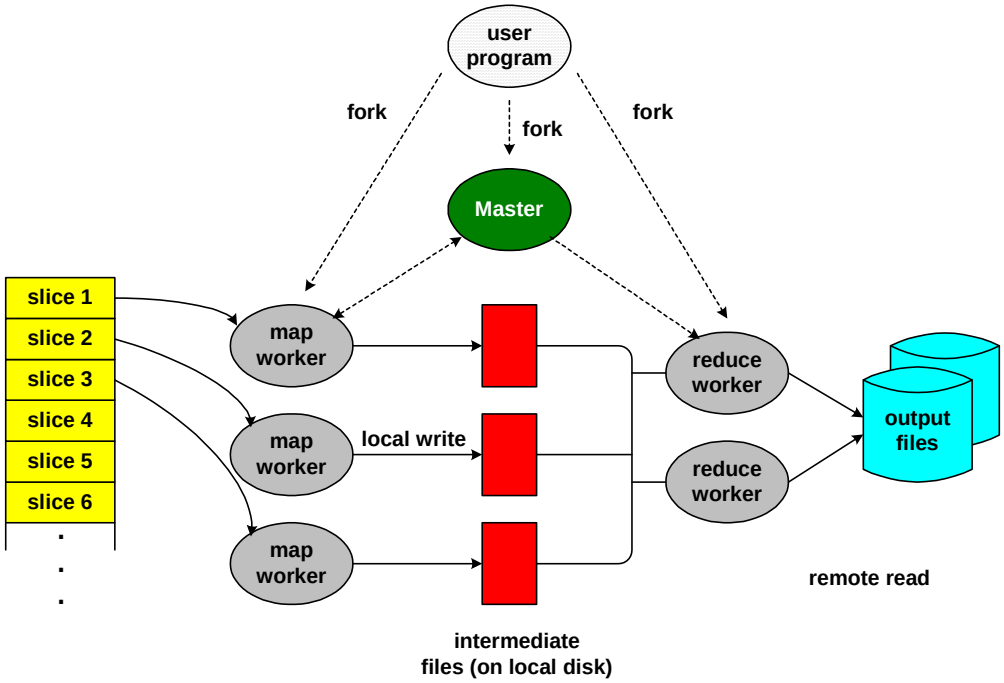
- acts as the conduit through which the locations of the intermediate file regions are propagated from Map tasks to Reduce tasks:

    - For each completed Map task, the Master stores the locations and the sizes of the *R* intermediate file regions produced by those Map tasks.

    - It pushes the location and size of the intermediate file regions to the Reduce functions responsible for those regions.

- provides additional support for performance, statistical reporting, and fault tolerance, as described later.

Once the scheduler has assigned the Map and Reduce tasks to specific machines, the MapReduce function proceeds as follows:



MapReduce Implementation

- The input data is split into the *M* specified splits, each typically 16 megabytes to 64 megabytes in size.

- The Map, Reduce, and Master tasks are started in their assigned machines. There are usually multiple Map and Reduce tasks assigned to each worker machine.

- The Master picks idle worker tasks and assigns them each a job. Each of the *M* Map worker tasks is assigned a slice from the input data. Each of the *R* Reduce worker tasks is assigned an intermediate file region.

- A Map worker task reads its assigned input slice and, using the user-supplied Map function, parses key/value pairs from it. The intermediate key/value pairs that it produces are buffered in its memory.

- Periodically, the buffered key/value pairs are organized into the *R* intermediate file regions by the Map worker task and are written to local disk. The disk locations of these regions are passed back to the Master.

- The Master notifies a Reduce worker task when an intermediate file region assigned to it becomes available and provides that worker task with the machine ID and the file location of the Map worker's machine.

- A Reduce worker uses remote procedure calls to read the newly-created intermediate file regions for its assigned region from the PCs at which they are resident. It then sorts the key/value pairs by key.

- The Reduce worker passes each key/value pair to the user-supplied Reduce function. The results of the Reduce function are appended to a global output file for this Reduce partition.

- When all Map tasks and Reduce tasks have completed, MapReduce returns control to the user program.

The result is a set of *R* output key/value files written to global storage. Often, these output files do not need to be combined into a single file. Rather, they are used as input to another MapReduce program for further processing.

Fault Tolerance

A typical MapReduce cluster contains hundreds to thousands of commodity machines. As a consequence, machine failure is common. MapReduce uses reexecution as the primary mechanism for recovery from a machine failure.

Worker Failure

The Master continually pings its worker machines. If a worker machine fails to respond, its state is marked as failed.

Map Worker

If the failed machine is a Map worker, any of its completed or in-progress tasks are reset to idle and its results ignored. This is necessary because the intermediate key regions which it has produced are stored on its local disk and may not yet have been read by the Reduce workers.

The Map tasks that had been running in the failed machine are restarted on other machines, and each reprocesses its assigned input split. The Reduce workers are notified of the reexecution and switch to the new Map worker to get their assigned intermediate file regions. Duplicates are ignored.

Reduce Worker

If the failed machine is a Reduce worker, its reduce tasks are reassigned to other machines. The Reduce tasks do not have to reexecute work already processed since their results are contained in global storage.

Load Balancing

To maintain a balance of load across the cluster, should a machine fail, its tasks may be allocated across several other worker machines.

Master Failure

Since the Master task is running on a separate machine, it will hardly ever fail. Rather than trying to continue on from a checkpointed state, all currently executing MapReduce programs are simply aborted and must be resubmitted.

Bad Records

Another common failure mode is process aborts due to bad records. A bad record that cannot be processed can effectively abort the entire MapReduce program.

If a Map or a Reduce task generates an exception, it sends a “last gasp” message to the Master before aborting. The message informs the Master of the record location and identification. If the Master receives two such messages for a given record, it notifies the new worker task that this record should be skipped on the task’s reexecution.

Stragglers

A PC can be slowed down significantly by an erratic disk that is generating a series of errors or by being overloaded by other tasks assigned to it by the cluster manager. The MapReduce program cannot complete until this *straggler* machine completes its MapReduce functions, consequently slowing down the entire execution.

MapReduce compensates for stragglers as follows. After most worker tasks have completed, MapReduce will start a backup copy of each incomplete worker task on another machine in the cluster. The results of the first worker task to complete are used by the Master, and the other task is aborted. In this way, stragglers have a limited effect on program execution. Google has found that without backup tasks, processing times can increase by as much as 50%.

Locality

Google has found that the limiting factor in these large computations is network capacity. It is imperative to minimize the amount of traffic flowing across the network due to intertask communications.

Network traffic is greatly minimized by ensuring that the Map workers do most of their work on their local disks. To start with, the input data is spread across the local disks of all of the Map worker machines. In fact, the data is written in three copies, with each split written to three different machines.

The Master attempts to assign Map tasks to splits that are on the same machine. If this cannot be done, a worker machine close to one of the machines holding the split is assigned to the split.

To the extent that Map tasks have local access to their splits, no network traffic is generated by a Map task since data read locally consumes no network bandwidth. Only the reading of the intermediate file regions by the Reduce tasks and the writing of completed regions to the global store consumes network capacity. Consequently, network capacity is not generally a limiting factor in the execution of MapReduce tasks.

Task Granularity

From a performance viewpoint, it is important to specify a fine granularity with respect to the Map and Reduce tasks. It is optimum to have *M* and *R*, the number of worker tasks, be much greater than the number of machines.

This fine granularity has several benefits. For one, it is easier to balance the MapReduce load across many machines. In addition, the reexecution time following a machine failure can be minimized by spreading its worker tasks among several surviving machines.

A typical configuration would have 200,000 Map tasks and 5,000 Reduce tasks spread among 2,000 worker machines.

Scalability

MapReduce is massively scalable. It is limited only by the capacity of the Master machine. The Master task must track *M* + *R* tasks and *M* x *R* intermediate file regions.

For the typical configuration described above, the Master must track 205,000 + (200,000 x 5,000) items, or more than a billion items.

Statistics

MapReduce maintains several counters, such as the number of tasks that are in progress or have completed, the input key/value pairs processed, and the number of intermediate keys generated. In addition, users can generate their own counters of execution activity.

The values of these counters are returned to the Master on each heartbeat ping. There they are organized into a web page so that users can monitor the progress of their runs. These statistics are also used to create a final report for the run. The report is returned to the user program.

Debugging

Debugging a program that is running across a thousand machines can be a daunting task. To remove this complexity, MapReduce can be configured to run on one machine only. Programmers can then use standard debugging tools to get their programs to run properly.

Performance

Google has measured some impressive performance metrics for MapReduce. For instance, a ten-terabyte search to count the number of a rare three-character sequence completes in 80 seconds (plus 70 seconds of overhead to start up the tasks).

A ten-terabyte sort of 100-byte records takes about fifteen minutes.

Summary

MapReduce is being used by Google for many applications requiring data reduction of massive data sets. Thousands of MapReduce jobs are run each day.

Perhaps the best known application is Google's production indexing system that responds so quickly to user searches. The MapReduce application that creates these indices parses a large set of documents retrieved by Google's crawling system – the Google spiders. Typically, 20 terabytes of data are processed at a time. A sequence of five to ten MapReduce operations is used to generate these indices.

The success of MapReduce is attributed to several factors:

- The model is easy to use even by programmers inexperienced in parallel and distributed processing systems. It hides the complexities of parallelization, fault tolerance, locality optimization, and load balancing.

- A large variety of problems are easily expressible as MapReduce computations. Such problems include indexing, sorting, data mining, and machine learning.

- MapReduce scales to large clusters containing thousands of machines. The implementation makes efficient use of these machine resources