MapReduce: Google case study about data processing on Large Clusters

- Mohammed Sarshaar

Business Problem:

The developers at Google have created a number of specialized calculations over many years to handle massive datasets, including web request logs and crawled pages, and produce derived data, such as inverted indices and summaries of web activity. Even though these calculations are conceptually straightforward, they must be distributed among hundreds or thousands of machines in order to be finished on time. The first duties are frequently made more difficult by the difficulties of managing failures, spreading data, and parallelizing computations, which results in lengthy and intricate code.

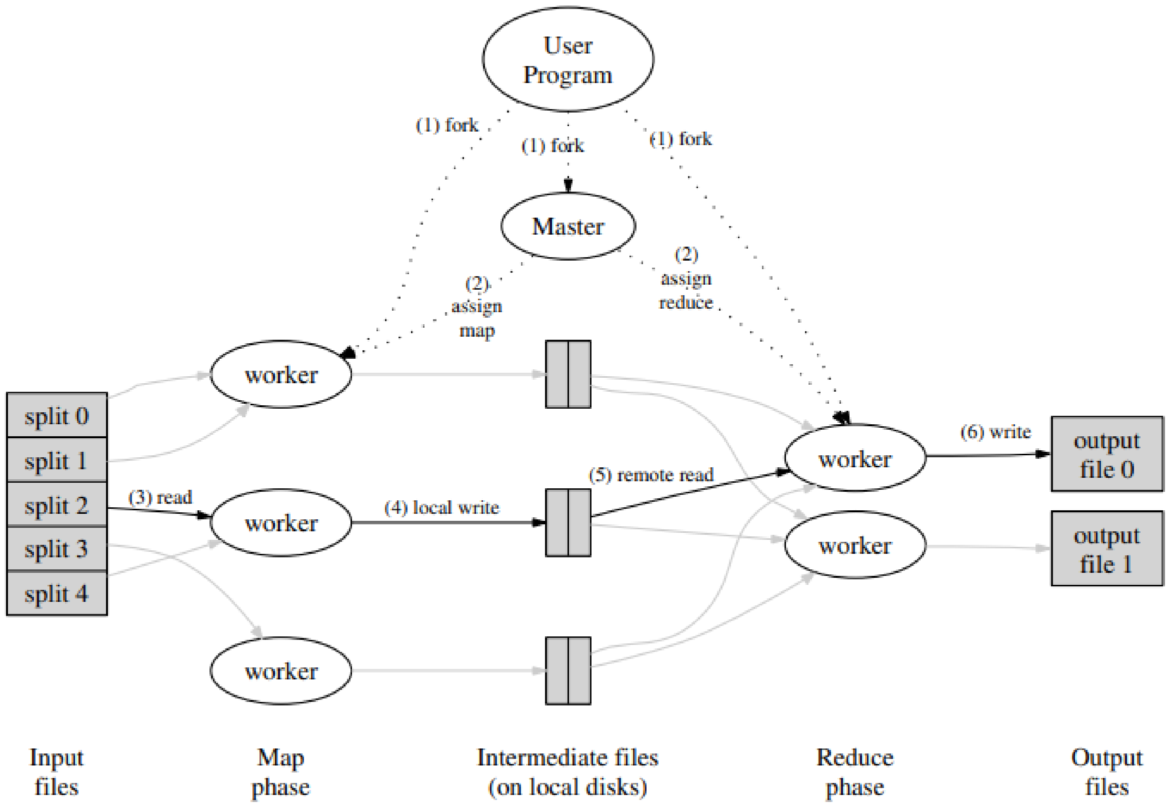
In order to overcome this complexity, the authors developed a novel abstraction that hides the finer points of data distribution, fault tolerance, load balancing, and parallelization inside a library while enabling the expression of simple calculations. The map and reduce functions in Lisp and other functional languages serve as the foundation for this concept. They discovered that the majority of calculations required mapping each input record to produce intermediate key/value pairs, then combining the results by decreasing those pairs based on shared keys. This functional architecture focuses on re-execution for fault tolerance and makes parallelization simple using user-defined map and reduce operations.

This work's primary contributions are a high-performance implementation appropriate for sizable clusters of conventional PCs and an intuitive user interface that facilitates automated parallelization and distribution of large-scale computations.

Programming Model

A set of input key/value pairs is used in the computation, and a set of output key/value pairs is produced. The calculation is expressed by the MapReduce library user as two functions, Map and Reduce. A user-written map generates a series of intermediate key/value pairs from an input pair. All intermediate values linked to the same intermediate key I are grouped together by the MapReduce framework and passed to the Reduce function.

An intermediate key I and a set of values for that key are accepted by the Reduce function, which is also user-written. These values are combined to create a potentially smaller collection of values. Usually, each Reduce invocation yields either zero or a single output value. An iterator provides the intermediate values to the user's reduction function. This enables us to manage value lists that are too big to store in memory.



Implementation

The Map invocations are distributed across multiple machines by automatically partitioning the input data into a set of M splits. The input splits can be processed in parallel by different machines. Reduce invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function (e.g., hash(key) mod R). The number of partitions (R) and the partitioning function are specified by the user.

The steps listed below take place when the user program invokes the MapReduce function:

1. The user program's MapReduce library first divides the input files into M chunks, each of which is normally 16-64 megabytes (MB) in size (a user-controllable parameter). On a cluster of machines, it then launches several copies of the program.

2. The master copy of the program is one of the unique copies. The remaining employees are those to whom the boss has delegated tasks. M map tasks and R reduce tasks need to be assigned. The master selects inactive employees and gives them either a reduce or a map duty.

3. When given a map job, a worker examines the contents of the matching input split. Key/value pairs are extracted from the input data and sent to the user-defined Map function. The Map function creates intermediate key/value pairs, which are then buffered in memory.

4. The buffered pairs, which have been divided into R areas by the partitioning algorithm, are periodically written to the local disk. The master receives the locations of these buffered pairs on the local disk and is in charge of sending them on to the reduce workers.

5. A reduction worker reads the buffered data from the map workers' local disks using remote procedure calls after receiving notification from the master about their locations. After reading all of the intermediate data, a reduce worker groups all instances of the same key together by sorting the data according to the intermediate keys

6. The reduce worker iterates through the sorted intermediate data, passing the key and matching set of intermediate values to the user's reduction function for each unique intermediate key it encounters. For this reduction partition, a final output file is appended with the reduction function's output.

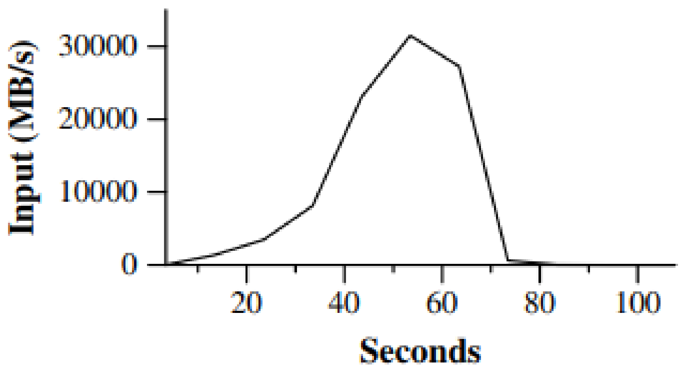
7. The user program is woken up by the master once all map and reduce tasks have been finished. At this stage, the user program's MapReduce call goes back to the user code.

Performance

In this part, we evaluate MapReduce's performance on two calculations that are executed on a sizable cluster of computers. A single calculation looks for a certain pattern in around one terabyte of data. The other computation sorts approximately one terabyte of data.

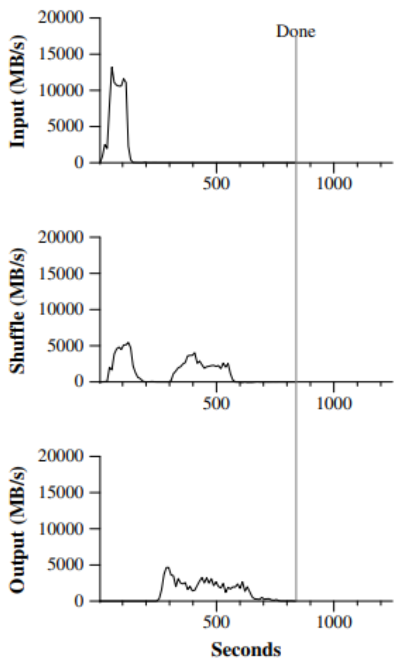
1. Grep

The Grep program scans through 10^{10} 100-byte records, searching for a relatively rare three-character pattern (the pattern occurs in 92,337 records). The input is split into approximately 64MB pieces (M = 15000), and the entire output is placed in one file (R = 1).

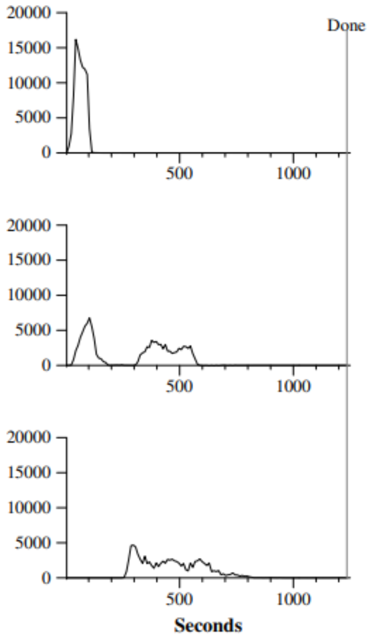


2. Sort

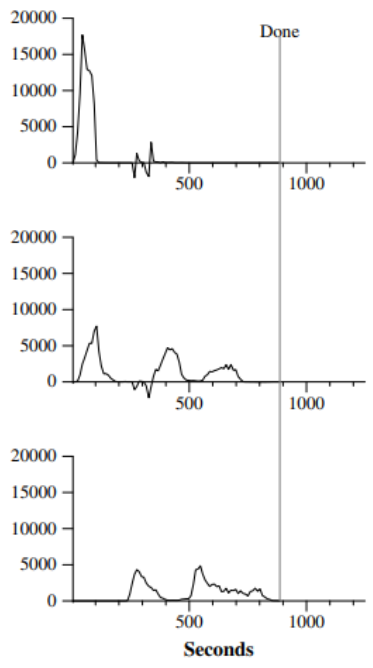
The sort program sorts 1010 100-byte records (approximately 1 terabyte of data). This program is modeled after the TeraSort benchmark. When a 10-byte sorting key is extracted from a text line using a three-line Map function, the key and the original text line are output as the intermediate key/value pair. As the Reduce operator, we made use of an integrated Identity function. As the output key/value pair, this function delivers the intermediate key/value pair unaltered.



(a) Normal execution



(b) No backup tasks



(c) 200 tasks killed

Conclusions

At Google, the MapReduce programming model has been effectively applied to a wide range of tasks. This achievement can be attributed to a number of factors. Because it conceals the specifics of parallelization, fault-tolerance, locality optimization, and load balancing, the paradigm is first and foremost simple to use, even for programmers who are unfamiliar with distributed and parallel systems. Second, a wide range of issues can be simply expressed as MapReduce calculations. Third, we have created a MapReduce solution that can scale to massive machine clusters of thousands of machines. The approach is appropriate for use on many of the major computational issues that Google faces since it effectively utilizes these machine resources.

References

1. Jeffrey Dean and Sanjay Ghemawat. Google article : Simplified Data Processing on Large Clusters
2. Guy E. Blelloch. IEEE Transactions on Computers
3. Luiz A. Barroso, Jeffrey Dean, and Urs Holzle. Web search for a planet: The Google cluster architecture
4. Arash Baratloo, Mehmet Karaul, Zvi Kedem, and Peter Wyckoff. Charlotte: Metacomputing on the web.
5. Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system.
6. Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The Condor experience.