6.033 Critique 2: MapReduce

March 10, 2016

1 Introduction

MapReduce, introduced by Google in the eponymous paper (Dean, Ghemawat 2004), is a distributed programming model and implementation used for automatically parallelizing code to be run over large datasets. It restricts the permissible set of programs to those implementing *map* and *reduce* functions which respectively generate and merge a set of intermediate key/value pairs. [Abstract]

The system designers seek to reduce the knowledge overhead and boilerplate code required to easily and effectively design a distributed programming operation by exposing a simple and flexible interface to programmers. Its design is platform-independent and highly scalable. We shall focus this critique on Google's in-network implementation, which runs on a large cluster of computers hosted by the company. [3]

2 Background

As increases in processor speed have diminished, programmers have sought alternatives to keep up with ever-growing requirements in complex code and big data. One promising field is that of distributed computing: programs are executed in parallel over many machines, taking advantage of the computing power and memory of the entire cluster.

Distributed programming, however, is very difficult to do manually. In the best case, the mental overhead of utilizing multiple computers requires extensive developer time. In the worst cases, parallelizing a program introduces race conditions and unpredictable errors resulting from network latency, machine failure, and variances in runtime across different machines. As such, many developers have historically hesitated to write distributed programs, for fear of incorrectness.

Other systems have addressed this problem by, similarly to MapReduce, restricting the set of programming options and automating the distributed code. These include Bulk Synchronous Programming, River, and the Charlotte system, all of which had features which inspired components of MapReduce. However, MapReduce features a more complex implementation than many of these, allowing for a simpler developer interface and easier distributed programming. [7]

3 System Design

The design of MapReduce, and specifically Google's implementation, relies heavily on modularity to allow the automatic distribution of work. The **user program** is the first module to interface with the system. Assuming proper implementation, the developer specifies the *map* and *reduce* functions, the data source, and defines how the data is to be split.

Then, the MapReduce system automatically forks this process and copies the code to the **map worker** and **reduce worker** submodules, which respectively process the code for the *map* and *reduce* functions. These workers communicate via

accesses to local memory on worker machines. Finally, a single **master** process is forked onto a machine. This process communicates with and manages task distribution across all of the workers, and indicates to the user when the program is complete. This modularity permits the flexibility and scalability intrinsic to the MapReduce system.

3.1 Simplicity

The MapReduce system heavily emphasizes simplicity in its design, seeking to allow the "conceptually straightforward" tasks performed by many distributed programs to be implemented without concerning developers with the difficult aspects of distributed programming. [1] This is primarily accomplished by defining the simple abstraction of Map and Reduce, which take in predefined input values and collectively convert these to a specified output via an intermediate key. [2] This has been proven to be effective; as of one year after the original deployment of MapReduce, several hundred implementing programs had been written. [Abstract]

In addition, MapReduce supplies developers with an extensive set of debugging tools. For one, a feature is included allowing all of the MapReduce code to be run locally and sequentially. [4.7] Additionally, the master collects status information about the running code and workers and displays this on an HTTP page. [4.8] These features allow developers to keep track of their code as it runs and easily diagnose bugs, improving simplicity.

3.2 Scalability

By nature of its status as a distributed computing platform, MapReduce is scalable. In general, this is accomplished by subdividing the tasks into small chunks, numbering many more than the count of worker machines. This allows task scheduling to occur dynamically as some machines run more quickly or encounter faster subtasks. [3.5] This was shown to be successful, as demonstrated in the performance metrics in Section 5.

One limiting factor to scalability is the heavy network usage required by MapReduce. The ethernet-connected worker and master machines by design pass substantial amounts of data to and from each-other. The MapReduce system somewhat mitigates this by attempting to schedule *map* tasks to workers already holding the given data, and by having the map workers write their output to local memory. [3.4] This has the downside of reduced fault tolerance, however, as failures in workers can cause stored calculations to be lost. In general, these alterations seemed to sufficiently respond to the concerns of bandwidth resource usage, though I question how fully this is addressed.

3.3 Fault-Tolerance

The last major design goal of the MapReduce system is fault-tolerance. The master process, successfully handles the monitoring and response to failed workers. By regularly pinging workers, the master can detect when they fail and reassign their tasks accordingly. [3.3] In addition, the atomic nature of workers' final steps for any given task prevents failures from propagating.

On the other hand, master failures are **not** accounted for in MapReduce. Instead, a failure in the master simply returns an error to the client. It is assumed that this is rare enough to be nonproblematic, though this tradeoff seems to be fairly inconvenient for long-running programs. [3.3]

Finally, MapReduce responds to faults in user input by permitting users to instruct MapReduce to skip bad records. This may be useful in cases where small numbers of failures do not affect final results, such as in statistical analysis. [4.6]

4 Conclusions

The MapReduce model attempts to provide a simple but flexible developer interface for writing effective distributed programs over large datasets. It does this by establishing *map* and *reduce* abstractions which are automatically split across worker machines, managed by a master machine. The system focuses on developer simplicity, scalability, and fault-tolerance as primary goals. MapReduce has been shown to be successful via its impressive benchmarks across common tasks, and the extent of its use in Google's production code.