MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

Overview

This paper introduces MapReduce, a programming model for processing large datasets. The essence of this model are the map function that generates intermediate key-value pairs from the input data and the reduce function that merges all the intermediate values with the same key with some associated operation. The paper first explains the need for MapReduce, which is the questions of parallelization and distribution that complicate otherwise straightforward computation. It then briefly describes the programming model for MapReduces and lists some problems and explains how MapReduce can be used to solve these problems.

In the third section, the paper delves into depth about the authors' implementation of MapReduce and the overall execution flow. It also discusses other aspects of the implementation such as the master node, fault tolerance, bandwidth considerations, computational costs and backups. The following section discusses improvements that can be made to the basic MapReduce flow such as adding a partitioning function for the Reduce tasks or adding an intermediate Combiner function in problems where there is a significant repetition of the intermediate keys. In the fifth section, the performance of MapReduce implementations of Grep and Sort are evaluated with normal executions not having backup tasks, and two hundred tasks being killed (for Sort only). The authors go on to discuss their experiences with MapReduce in the Google environment and the associated considerations. In the seventh section, the authors discuss similar and related works to MapReduce before concluding by listing the strengths of MapReduce.

Contributions and Positive Aspects

The MapReduce algorithm achieves its goal of aiming to simplify the problems of distribution and parallelization that accompany problems of working with Big Data. Since the MapReduce framework already has the framework necessary to split and then combine the data through the execution flow, the developer using MapReduce would not need to spend an excessive amount of time puzzling out that aspect of the job. For example, in the word counting example from the paper, the challenge is not so much puzzling out how to count the words as much as working out how to perform the computations on possibly terabytes of data. MapReduce virtually eliminates these considerations. The simplicity and clarity of purpose of this algorithm has helped it become an extremely influential framework, as seen in Hadoop's MapReduce and the underlying foundation for Apache Spark.

Another reason for its enduring popularity is the modular nature of MapReduce. By splitting the execution flow into neatly defined stages and specifying exactly what needs to happen in each phase, it is possible for developers to swap in and out different methods of execution that would better suit their problems. This can even be seen in the paper, when it discusses possible refinements to MapReduce, especially in adding a combiner in case keys are repeated a great amount in the intermediate stage. The ease of modification of MapReduce has helped make it

versatile and applicable to a wide variety of problems, even if the problem cannot be solved completely with MapReduce alone, MapReduce is still a useful tool for specific segments.

Limitations

It is important to note when discussing MapReduce's limitations that since it is so effective at accomplishing what it sets out to do, that the limitations are generally a discussion of problems/scenarios that MapReduce is not the best fit for.

The flip side to MapReduce's effectiveness in simplifying the process of parallelization is that it will not be as effective in problems that are more challenging to parallelize. In general, the key issue here would be the recombination phase. If it is difficult to recombine the data, then MapReduce may not be the ideal solution. A good example of a class of problems which MapReduce would not be able to solve are NP-Hard graph-related problems such as the Travelling Salesman problem, and influence maximization in which if the graph is segmented, recombination would cause the values to lose their meaning.

As the paper touches on, there are also considerations of storage and bandwidth. If the input dataset is large, then the intermediate data will yield data that is at least as large but probably larger and it would be necessary to have a good file system in place to handle this expansion in data. If the intermediate stage contains too much data to store, then MapReduce may not be the best solution to the problem. Additionally, given the expansion of data in the intermediate stages and the considerations of shuffling and then recombining data along with latency issues, it is not unreasonable to expect MapReduce to take a long time to execute.

MapReduce seems to not consider the possibility of dynamic input data, for example streaming data, or if changes were made to the input data after execution begins. It seems to be best suited to batch-processing. MapReduce also appears to work best for linear, 'one-and-done'-type execution flows. For example, in word counting and sorting, there is a clearly-defined beginning, middle and end. However for problems with iterations, such as machine learning or data mining problems with iterations, it would probably not be feasible to repeat MapReduce in a cycle for the specified number of iterations.

Points for Discussion, Comments and Suggestions

Approaching MapReduce from the perspective of invariant-based reasoning as discussed in class, I wonder if invariant-based reasoning can be applied to this algorithm, and if so, what the safety and progress properties of MapReduce would be. Would the safety and progress properties depend on the problem? (e.g. word counting having different properties from sorting) Additionally, when it comes to the master node and replicas being made of backups, it brings to mind the discussions of chain replication. An in-depth discussion of this would be beyond the scope of the paper, but I wonder if there are implementations of MapReduce that use Paxos for the master node or chain replication in any form.

When it comes to the discussion of fault tolerance, in my opinion, the paper brushed off concerns about scenarios in which the master node fails. I find it difficult to believe that modern-day

implementations of MapReduce would simply abort the entire process and restart if the master node were to fail.

In the section discussing Backup Tasks, the paper lists ‘stragglers’ as one of the reasons for execution taking a long time. I wish the authors had included some more causes for the prolongation of execution. Relating to this, in my opinion, the authors should have more clearly-defined the scope of problems that it is possible for MapReduce to solve. The paper says multiple times that Google uses MapReduce in machine learning problems, but never explain how or in what capacity. I know from prior education that inapplicability to machine learning is one of MapReduce’s most well-known drawbacks. The paper only goes into depth about one specific use of MapReduce at Google, which, to me, was not enough information about the practical applications of MapReduce.

In the section where the performance of MapReduce was evaluated, I felt that using only two examples was not enough. Given that the very first example of an application of MapReduce was the word counting problem, its absence in this section was clearly felt by me. It might have also been useful to see more applications of the refinements in the previous section being applied to the problems here, especially the partition and combiner functions.

Conclusion

This paper introduces and explains MapReduce a two-step programming model that first emit pairs of keys and values from the input data and then performs a reduction on these pairs to obtain some final output. It assists developers in working out issues of parallelization with big data and is highly modular and versatile. However, it is also limited by the types of problems it can solve