8.1 Limitations of MapReduce

As we have seen throughout this book, solutions to many interesting problems in text processing do not require global synchronization. As a result, they can be expressed naturally in MapReduce, since map and reduce tasks run independently and in isolation. However, there are many examples of algorithms that depend crucially on the existence of shared global state during processing, making them difficult to implement in MapReduce (since the single opportunity for global synchronization in MapReduce is the barrier between the map and reduce phases of processing).

The first example is *online learning.* Recall from Chapter 7 the concept of learning as the setting of parameters in a statistical model. Both EM and the gradient-based learning algorithms we described are instances of what are known as *batch* learning algorithms. This simply means that the full "batch" of training data is processed before any updates to the model parameters are made. On one hand, this is quite reasonable: updates are not made until the full evidence of the training data has been weighed against the model. An earlier update would seem, in some sense, to be hasty. However, it is generally the case that more frequent updates can lead to *more* rapid convergence of the model (in terms of number of training instances processed), even if those updates are made by considering *less* data [24]. Thinking in terms of gradient optimization (see Section 7.5), online learning algorithms can be understood as computing an approximation of the true gradient, using only a few training instances. Although only an approximation, the gradient computed from a small subset of training instances is often quite reasonable, and the aggregate behavior of multiple updates tends to even out errors that are made. In the limit, updates can be made after *every* training instance.

Unfortunately, implementing online learning algorithms in MapReduce is problematic. The model parameters in a learning algorithm can be viewed as shared global state, which must be updated as the model is evaluated against training data. All processes performing the evaluation (presumably the mappers) must have access to this state. In a batch learner, where updates occur in one or more reducers (or, alternatively, in the driver code), synchronization of this resource is enforced by the MapReduce framework. However, with online learning, these updates must occur after processing smaller numbers of instances. This means that the framework must be altered to support faster processing of smaller datasets, which goes against the design choices of most existing MapReduce implementations. Since MapReduce was specifically optimized for batch operations over large amounts of data, such a style of computation would likely result in inefficient use of resources. In Hadoop, for example, map and reduce tasks have considerable startup costs. This is acceptable because in most circumstances, this cost is amortized over the processing of many key-value pairs. However, for small datasets, these high startup costs become intolerable. An alternative is to abandon shared global state and run independent instances of the training algorithm in parallel (on different portions of the data). A final solution is then arrived at by merging individual results. Experiments, however, show that the merged solution is inferior to the output of running the training algorithm on the entire dataset [52].

A related difficulty occurs when running what are called *Monte Carlo simulations*, which are used to perform inference in probabilistic models where evaluating or representing the model exactly is impossible. The basic idea is quite simple: samples are drawn from the random variables in the model to simulate its behavior, and then simple frequency statistics are computed over the samples. This sort of inference is particularly useful when dealing with socalled *nonparametric models*, which are models whose structure is not specified in advance, but is rather inferred from training data. For an illustration, imagine learning a hidden Markov model, but inferring the number of states, rather than having them specified. Being able to parallelize Monte Carlo simulations would be tremendously valuable, particularly for unsupervised learning applications where they have been found to be far more effective than EM-based learning (which requires specifying the model). Although recent work [10] has shown that the delays in synchronizing sample statistics due to parallel implementations do not necessarily damage the inference, MapReduce offers no natural mechanism for managing the global shared state that would be required for such an implementation.

The problem of global state is sufficiently pervasive that there has been substantial work on solutions. One approach is to build a distributed datastore capable of maintaining the global state. However, such a system would need to be highly scalable to be used in conjunction with MapReduce. Google's BigTable [34], which is a sparse, distributed, persistent multidimensional sorted map built on top of GFS, fits the bill, and has been used in exactly this manner. Amazon's Dynamo [48], which is a distributed key-value store (with a very different architecture), might also be useful in this respect, although it wasn't originally designed with such an application in mind. Unfortunately, it is unclear if the open-source implementations of these two systems (HBase and Cassandra, respectively) are sufficiently mature to handle the low-latency and high-throughput demands of maintaining global state in the context of massively distributed processing (but recent benchmarks are encouraging [40]).