Designing Good Algorithms for MapReduce and Beyond

Foto N. Afrati  
Google  
afrati@gmail.com

Magdalena Balazinska  
University of Washington  
magda@cs.washington.edu

Anish Das Sarma  
Google  
anish@google.com

Bill Howe  
University of Washington  
billhowe@cs.washington.edu

Semih Salihoglu  
Stanford University  
semih@stanford.edu

Jeffrey D. Ullman  
Stanford University  
ullman@gmail.com

Categories and Subject Descriptors
H.0 [Information Systems]: General

General Terms
Algorithms, Design, Management, Performance

Keywords
map-reduce algorithms, distributed computing, communication parallelism tradeoff

Tutorial Summary
As MapReduce/Hadoop grows in importance, we find more exotic applications being written this way. Not every program written for this platform performs as well as we might wish. There are several reasons why a MapReduce program can underperform expectations. One is the need to balance the communication cost of transporting data from the mappers to the reducers against the computation done at the mappers and reducers themselves. A second important issue is selecting the number of rounds of MapReduce. A third issue is that of skew. If wall-clock time is important, then using many different reduce-keys and many compute nodes may minimize the time to finish the job. Yet if the data is uncooperative, and no provision is made to distribute the data evenly, much of the work is done by a single node.

Communication Cost and Computation Cost
We start by explaining the tradeoff between communication cost and the computation cost of the reducers: the finer we partition the work of the reducers so that more parallelism can be extracted, the greater will be the total communication between mappers and reducers. We introduce a model of problems that can be solved in a single round of MapReduce computation, and use it to demonstrate the tradeoff [4]. We then use the model to compute lower bounds and present algorithms that meet these bounds for a number of problems described below.

Hamming Distance 1: We demonstrate how to use the model from[4] on the problem of finding pairs of bit strings of length $b$ that are at Hamming distance 1. For this problem, we show that any MapReduce algorithm that assigns at most $q$ inputs to one reducer, has to send each input to at least $\frac{b}{\log q}$ reducers on average. Therefore as the computation cost of the reducers decreases (i.e as $q$ decreases), the communication cost increases as each input has to be sent to more reducers on average. We then show several algorithms that match this lower bound exactly at different levels of $q$.

Joins and Multiway Joins: The join is one of the most common operations on data. Hash joins implemented in MapReduce are quite simple unless there are skew issues. When we need to join more than two relations, however, it is not clear which is more efficient: a sequence of MapReduce jobs each performing a two-way join or one MapReduce job that performs a multiway join. We will discuss this issue and show how to perform a multiway join by minimizing the communication cost between the mappers and the reducers [5]. Multiway joins can be used to find all occurrences of a graph pattern, such as triangles, in a larger data graph. We will present these techniques ([24], [2]).
**Fuzzy/Similarity Joins:** We shall present several single-round MapReduce algorithms for finding all pairs of elements from an input set that meet a similarity threshold [3]. Algorithms are presented first in terms of Hamming distance, but extensions to edit distance and Jaccard distance will also be discussed. There are many different approaches to the similarity-join problem using MapReduce, and none dominates the others when both communication and reducer costs are considered. Our cost analyses enable applications to pick the optimal algorithm based on their communication, memory, and cluster requirements.

**Multiround MapReduce Algorithms**

We present algorithms that can be implemented in two rounds of MapReduce for various problems, including:

(a) Analytic queries, which often involve joining large relations, and usually end by aggregating the output.
(b) Enumerating triangle patterns in a data graph by using two rounds of MapReduce instead of one.
(c) For matrix multiplication, there is an elementary one-round MapReduce algorithm. We show how to implement this algorithm optimally, but then explore a two-round algorithm. We demonstrate that the two-round approach has communication cost significantly less than the cost for the one-round family of algorithms [4]. Moreover, the computation cost of both approaches is the same.

**Problems and Solutions with Load Imbalances**

We define and illustrate the problem of skew in a MapReduce job. We study its causes, its prevalence in today’s Hadoop clusters, and its impact on job execution times [22, 17]. We present different methods for mitigating skew in MapReduce applications. We first discuss the MapReduce approach to handling slow tasks (aka stragglers) automatically during the execution of a job by using speculative tasks [10]. We present the approach, study how well it works in practice [22], and discuss the types of skew that it can handle. We then discuss various options that developers have for mitigating other types of skew starting from tuning the number of tasks in their jobs to implementing skew-resilient algorithms (eg [23]) and other best practices [17]. Finally, we present recent advances aimed at extending the MapReduce paradigm to mitigate skew automatically in a manner that is transparent to developers and users. We present techniques that improve the original MapReduce speculative execution [7, 28], various skew-resilient join implementations [21, 1, 27], static skew-avoidance techniques [13, 14, 15, 16], and dynamic skew-mitigation techniques [26, 18].

**Iterative Algorithms**

Many algorithms using MapReduce, including PageRank, text analysis, recursive relational queries, clustering, neural-network analysis, social network analysis, and network traffic analysis share a common trait: data is processed iteratively until the computation satisfies a convergence or stopping condition. The MapReduce framework does not directly support these iterative data analysis applications. Instead, programmers must implement iterative programs by manually issuing multiple MapReduce jobs and orchestrating their execution using a driver program.

We present algorithms and best practices for implementing this “external” iteration over vanilla MapReduce, and then present more sophisticated techniques for transforming an iterative algorithm into a noniterative one with applications to graph query, clustering, and machine learning [8, 19]. We will then present “internal” extensions to MapReduce that directly support iteration and show the significant effect they have on performance [8]. We use this platform to demonstrate a scalable Datalog system [20]. Finally, we will consider a variety of optimization strategies for recursive queries in this environment, with some emphasis on the “endgame problem”: jobs where only a few iterations are required to compute most of the result, but many more less productive iterations are required to complete the work [6].

**Nonblocking Processing**

We discuss the MapReduce design decision to extensively rely on synchronization barriers during processing [10]. We discuss techniques for parallel data processing without blocking both in the context of directed acyclic graphs of operations [11, 9] and in the context of iterative processing [12]. We also discuss interactions between synchronization barriers and fault-tolerance [25].

1. REFERENCES


