Adaptive Scheduling in Spark

by

Rohan Mahajan

Submitted to the Department of Electrical Engineering and Computer
Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the Massachusetts Institute of Technology

June 2016

Author . . . .
Department of Electrical Engineering and Computer Science
June 4, 2016

Certified by . . . .
Prof. Matei Zaharia
Thesis Supervisor

Accepted by . . . .
Dr. Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Adaptive Scheduling in Spark

by

Rohan Mahajan

Submitted to the Department of Electrical Engineering and Computer Science
on June 4, 2016, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Because most data processing systems are distributed in nature, data must be transferred between these machines. Currently, Spark data processing systems predetermines the strategies for how this data is to be shuffled before the job has started, but in certain situations, perfomance may be improved by not performing the typical strategy. We add functionality to track metrics about the data during the job and adapt our strategy during the job. We use this strategy to improve regular shuffle performance, joins using Spark's RDD interface, and joins in Spark SQl.

Acknowledgments

First, I would like to thank my parents Umesh Mahajan and Manjulan for their enduring support and love throughout my time at MIT.

I would like to thank Professor Matei Zaharia for his guidance and support while advising me throughout this project.

At MIT, my work would never have been completed if not for the support of my friends. I would like to thank them for all the lessons that I have learned from them, the questions that they answered, and all of the memories that I have created. I would also like to give special thanks to James Thomas who answered numerous questions throughout the project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Distributed Systems

With the rise of big data, new data processing systems have been designed to help process it. Instead of relying on using just onemore powerful computers, these systems use many computers and are thus distributed in nature due to economic costs,scalability, and fault tolerance. Because programming in these distributed environments is challenging, data processing systems try to abstract this from the user and provide a simple interface for them. One of the most popular systems is Map Reduce, invented by Google, which provides a simple map and reduce operation to the user. Another of these systems is Spark, which provides a slightly more expressive api then map reduce and also has different modes of fault tolerance than spark along with caching. One key and slow stage that both of these systems have, is that because they are distributed in nature, they have a stage where data must be transferred between machines, called the shuffle stage. This shuffle stage can be the bottleneck and be the reason for low performance of jobs.

1.2   Shuffle

1.2.1   Shuffle Introduction

In map reduce, data is loaded onto different computers and an computation is performed on it (the map phase) that results in a group of key value pairs. The final phase of map reduce,the reduce phase assumes that all key-value pairs with the same key are grouped onto the same machine. Thus, an intermediate phase that these systems handle themselves is the shuffle phase where data is transferred so that all key value pairs with the same keys result to be on the same machine.

The following example in Figure 1-1 details the inner workings of what happens in a shuffle for map reducer. Suppose we want to count the number of letters in a distributed file. The mappers will count the number of letters in their individual file. However, we need to aggregate this and thus all the counts for letter a will be sent to worker1, letter b will be sent to worker 2, letter c will be sent to worker c. These reducers will then promptly aggregate the counts that they receive from the mappers.

Because of the huge amounts of keys, these systems do not deal with on the granurality of keys. nstead, they deal with the concepts of partitions. These systems have different partitioning functions that map key-value pairs to different partitions. Some popular partition schemes include range and hash partitioning. As long as all the mappers agree to partition their data in the same way and send each partition that is the same to the same reducer, we are ensured that any two keys that are the same will be in the same partition.

1.2.2   Shuffle Analysis

Throughout the shuffle process, we want to balance the amount of data being sent to the reducers. These systems are constrained by the slowest worker, so generally we want to minimize the latencies of the slowest worker. By balancing the data being sent, we can first reduce the latency for network transfer. Second, because the data each node has to process is more balanced, we can reduce the execution time for the slowest node. We illustrate this
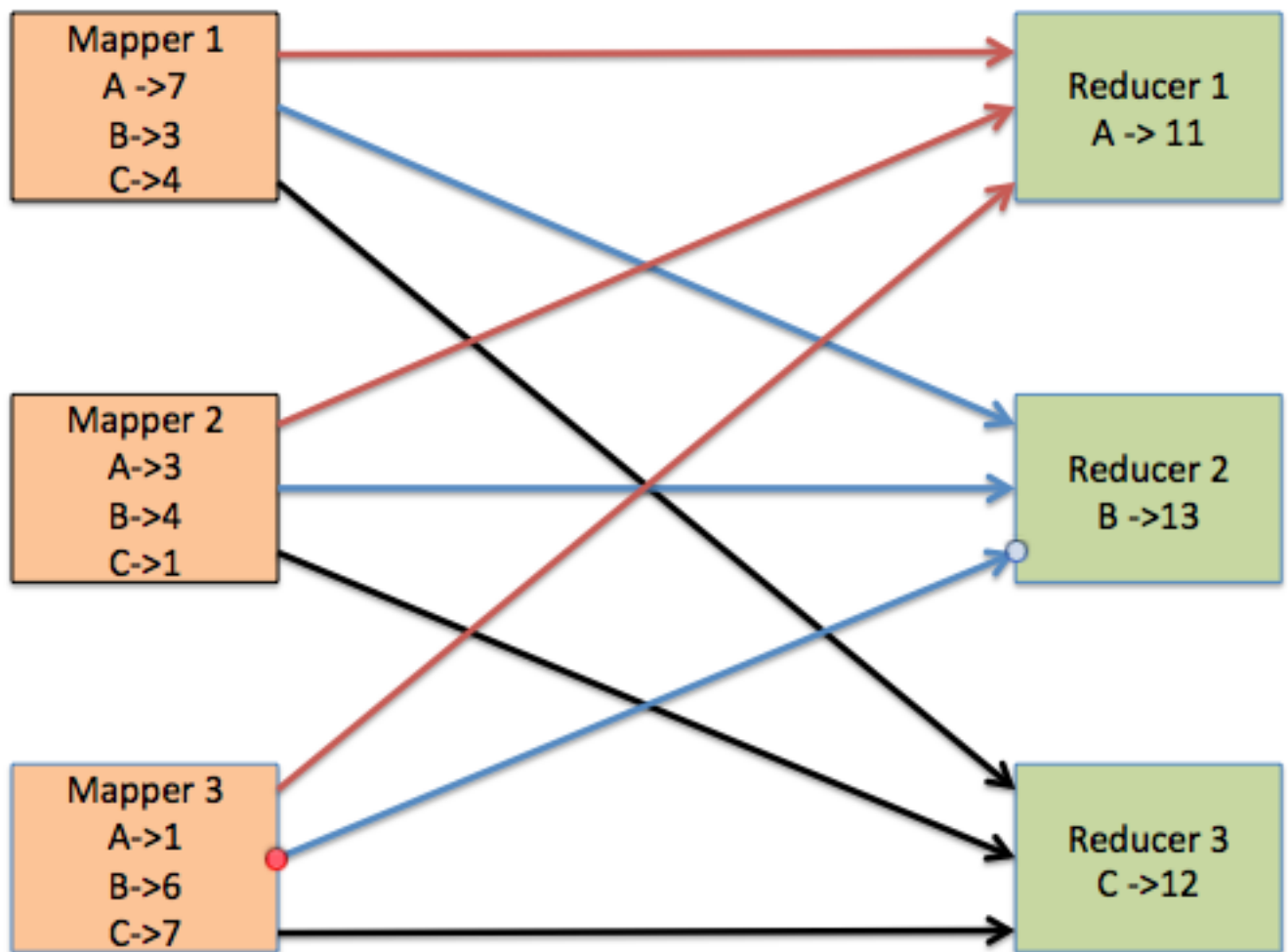
Figure 1-1: Shuffle for Letter Count in Map Reduce

in the following situation. In Figure **??**, we demonstrate a common scenario that happens in these shuffle scenarios. Let us say we have a simple partitioning scheme, where the reducer a partition lands up is determined by partition id mod the number of works. This seems reasonable as typically we do not have extra information about the size of the partitions. This leads to situations where we one reducer receiving 50MB of data while another reducer gets 90MB of data. However, if we knew the size of each partition after the mappers have run, we could more intelligently partition the data. As seen in Figure **??**, we see that by intelligently distributing the partitions in the same situation we can lead to each partition for 60MB.

1.3 Adaptive Scheduling of Joins

1.3.1 Join Basics

A common operation in these data processing environments is a join. A join basically combines two datasets by finding intersections between the keys between the two different datasets. An illustrative example is seen below.For instance, we have data in 1.1 and corresponding data in 1.2 that we are trying to join based on the intersection of key1 and key2. The resulting output is what we will find in 1.3 We will see a twice as we will see the interesection with other a in table 2, b is not present in the output table because is not found in table 2, while c will be present because its output is found in table 3.

Key1	Value1
a	1
a	1
b	3
c	4

Table 1.1: Table for Dataset 1

Key2	Value2
a	5
c	7

Table 1.2: Table for dataset 2

Key1	Value1	Value2
a	1	5
a	2	5
c	4	7

Table 1.3: Table of Joined Data

1.3.2 Shuffle Join

The above description describes how a join works in a simple non-distributed case. However, we are dealing with joins in a distributed fashion. Joins are very similar to the shuffle

but with two sets of partitions that we have to merge. For instance, take a look at the figure 1-4 As depicted, basically a shuffle takes places twice, with both datasets sending all their corresponding to the same reducer. For instance, for both datasets, all of the keys that mapped to partition 1 was sent to the reducer 1 and this happens respectively for the rest of the partitions.

1.3.3   Broadcast Join

One thing to notice in the diagram is that it seems that mappers and reducers are different machines. However, this distinction is imaginary and wholly based to help logic decision. In fact, they do run on the same machines. Because of this not all data it transferred over the network. If reducer 1 corresponds to mapper 1, then it actually does not need to send the network of 10MB corresponding to partition 1 already on the mapper.

In certain situations, it might make sense to keep all of dataset1 in place and transmit all of dataset 2 to each machine in dataset 1. This method still provides correctness even though dataset 1 stays in place, because all partitions of dataset 2 are sent. The advantages are that if dataset 2 is super small and dataset 1 can just stay in place, the network latency can be reduced. The following diagram 1-5 demonstrates how data would be shuffled using the broadcat join. Notice how the amount of data being transferred would be in the kiloybytes instead of the megabytes.

Broadcast Join is not always the optimal strategy. Because the entirety of dataset2 is sent to all partitions, the amount of computation increases. Additionally, if the datasets are sufficiently the same size, than the amount of data transferred over the network will actually increase because dataset 2 is sent in its entirety to every machine.

Thus, when choosing join strategies it becomes clear that certain strategies are good in certain situations. Thus, it becomes imperative to be able to pick the strategy after the mappers have run and we know the sizes of them.
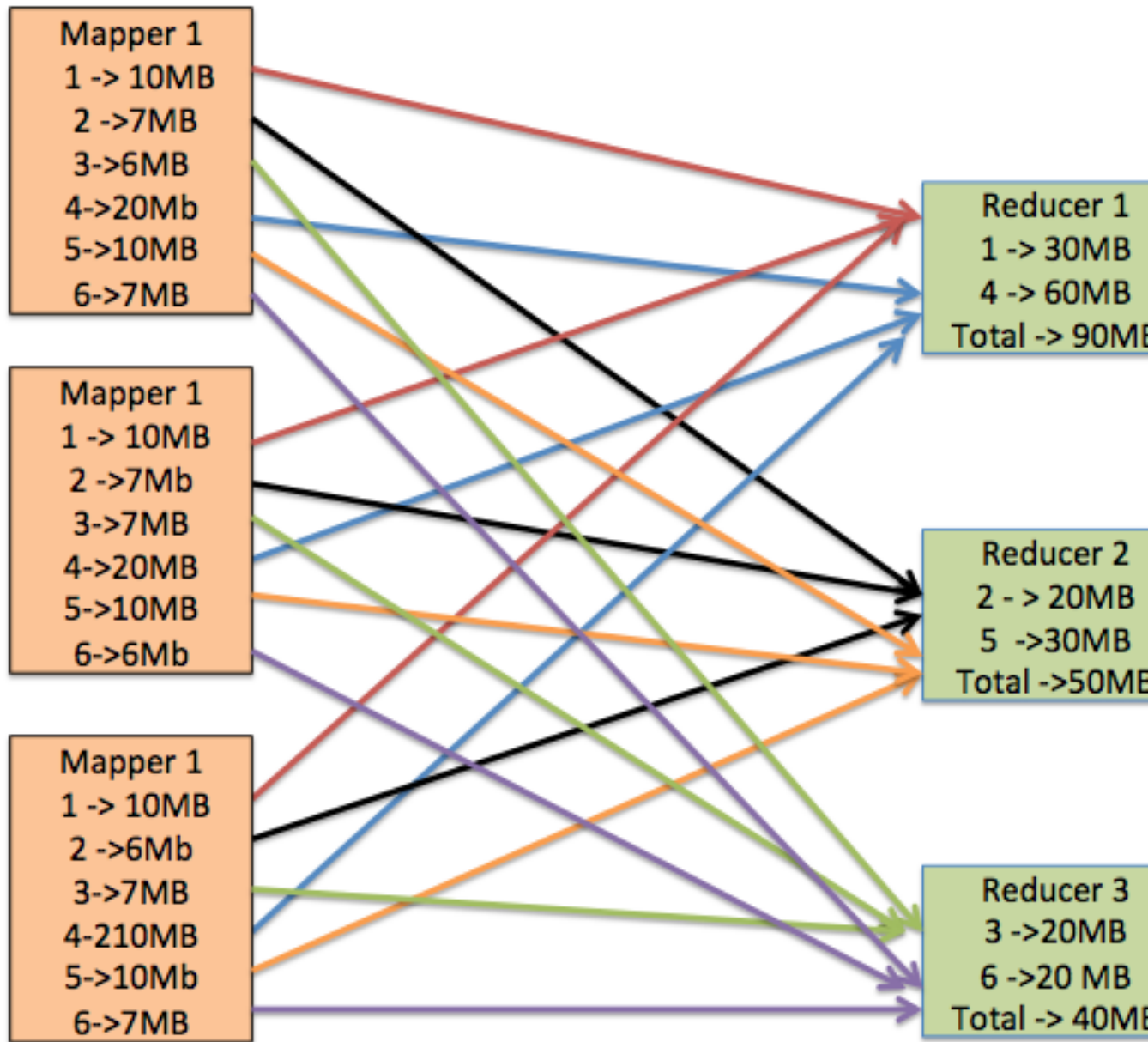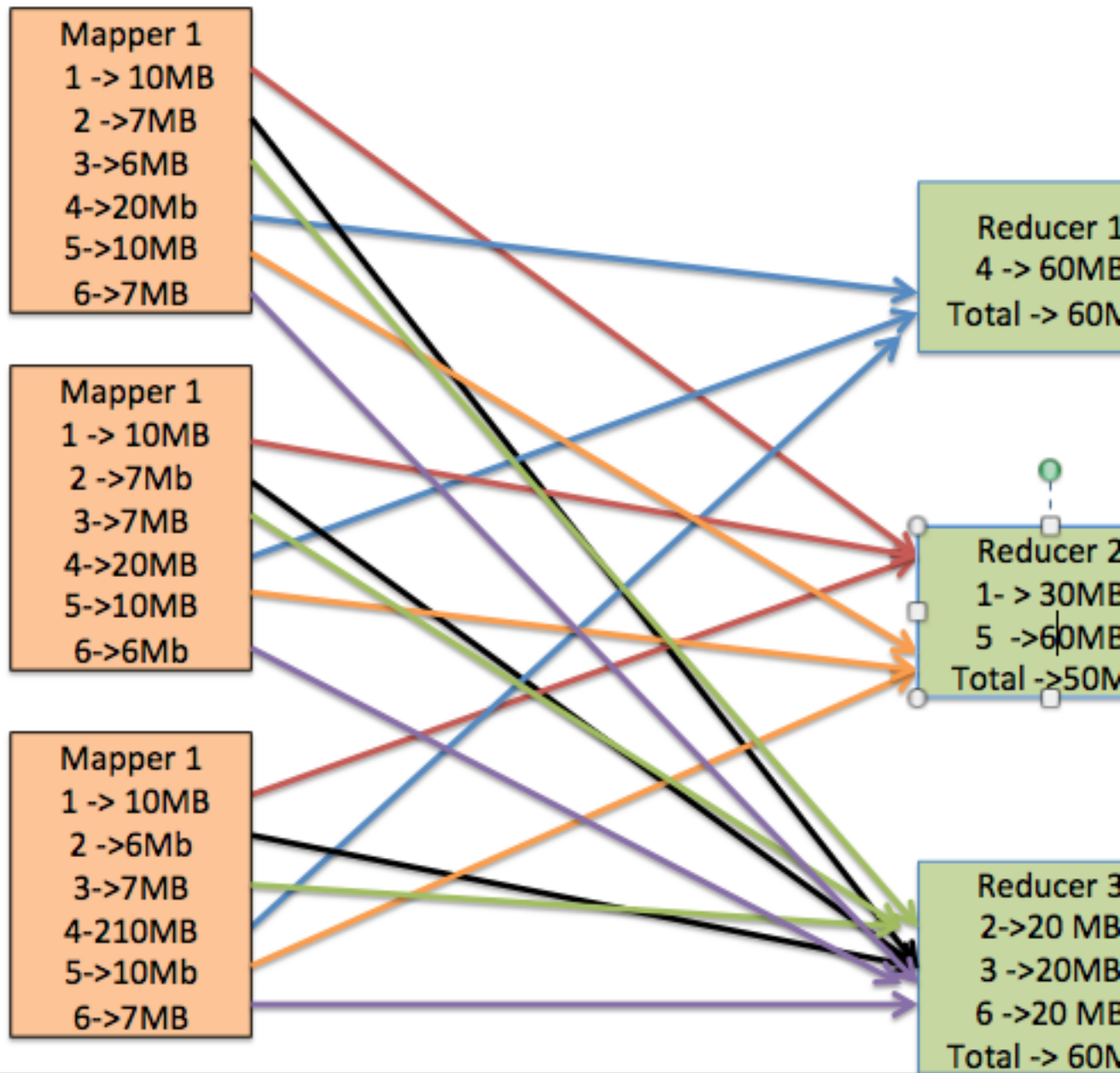
Figure 1-2: Unbalanced shuffle of partitions

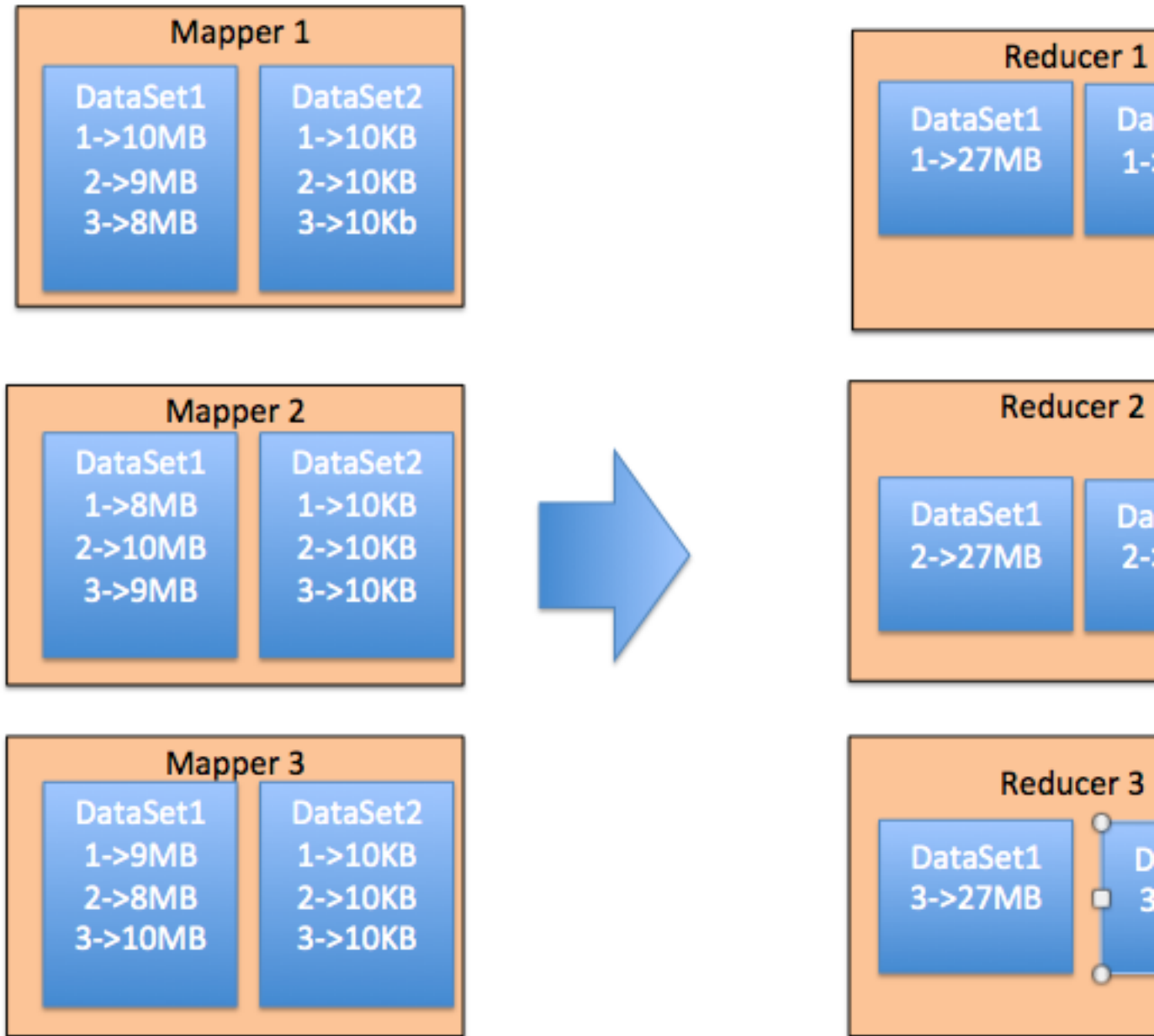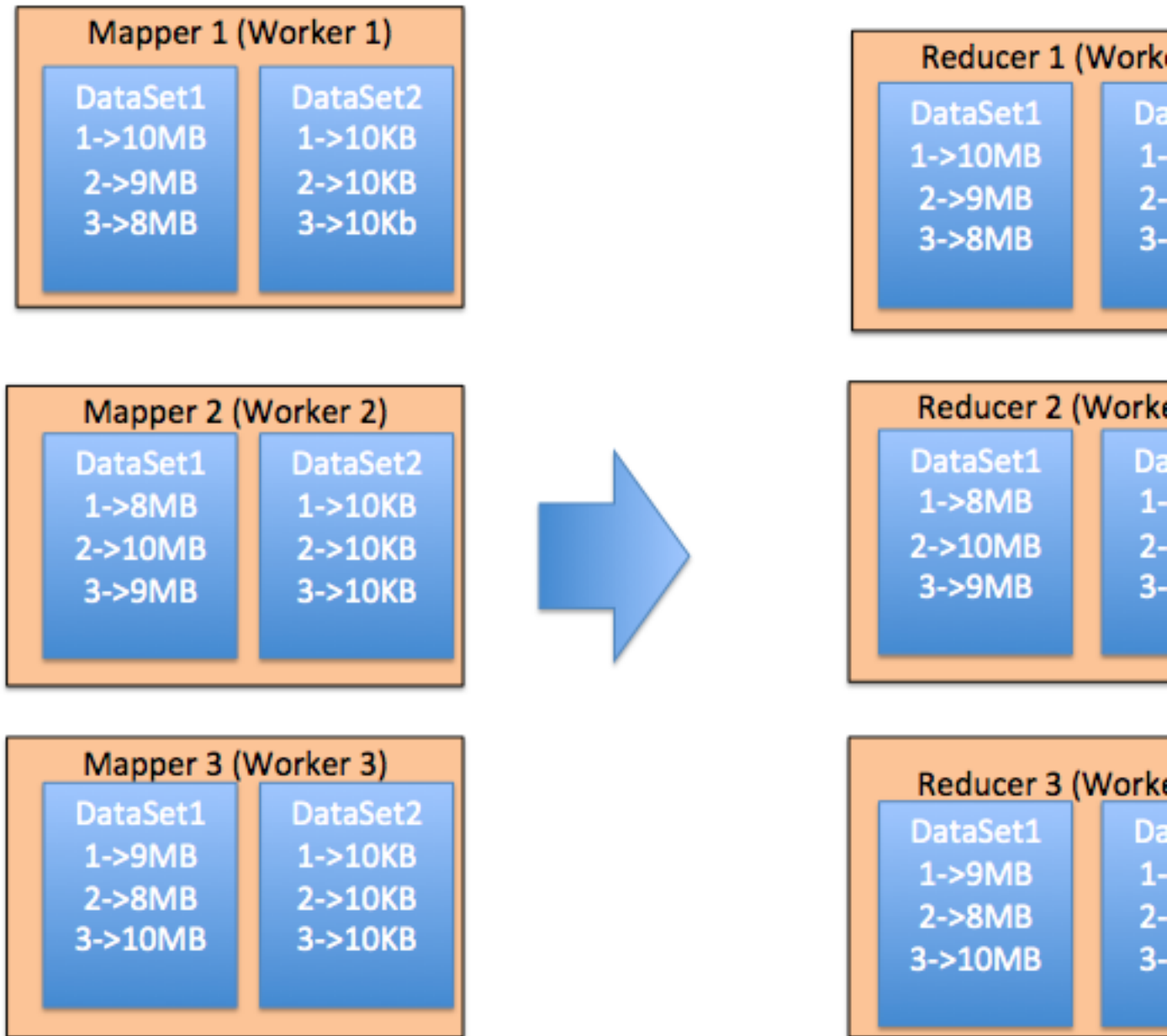**Figure 1-3:** Balanced shuffle of partitions.

Figure 1-4: Typical Shuffle Join.

Figure 1-5: Broadcast Join

Bibliography