

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

© Massachusetts Institute of Technology 2016. All rights reserved.

Author
Department of Electrical Engineering and Computer Science
May 20, 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 May 20, 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, performance 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 Manjula Mahajan for their enduring support and love throughout my time at MIT.

I would like to thank Professor Matei Zaharia for his guidance, patience, and support while advising me throughout this project. I learned a lot throughout this project and am extremely grateful for the support.

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 and all of the memories that I have created.

Contents

1	Introduction	13
1.1	Distributed Systems	13
1.2	Shuffle	14
1.2.1	Shuffle Introduction	14
1.2.2	Shuffle Analysis	14
1.3	Adaptive Scheduling of Joins	16
1.3.1	Join Basics	16
1.3.2	Shuffle Join	16
1.3.3	Broadcast Join	17
2	Implementation	23
2.1	Spark	23
2.2	ShuffledRDD	23
2.3	Joins	24
2.3.1	ShuffleReader Changes	24
2.3.2	ShuffleJoinRDD and BroadcastJoin RDD	24
2.3.3	Joins in Spark SQL	25
3	Experiment	29
3.1	Setup	29
3.2	Regular Shuffle	29
3.3	Broadcast and ShuffleJoinRDD	29
3.4	Spark SQL join	29

4	Future Research and Conclusion	31
4.1	Future Research	31
4.1.1	Extension of Shuffle	31
4.1.2	Extension to Join	31
4.2	Conclusion	32

List of Figures

1-1	Shuffle for Letter Count in Map Reduce	15
1-2	Unbalanced shuffle of partitions	18
1-3	Balanced shuffle of partitions.	19
1-4	Typical Shuffle Join.	20
1-5	Broadcast Join	21
2-1	Plan for Sort Merge Join	27

List of Tables

1.1	Table for Dataset 1	16
1.2	Table for dataset 2	16
1.3	Table of Joined Data	16

Chapter 1

Introduction

1.1 Spark and MapReduce

New data processing systems such as Spark and MapReduce have been designed to help process the increasing amount of data. Instead of relying on just one powerful computer, these systems use many computers due to lower costs and increased scalability and fault tolerance. Because these systems are distributed in nature, they have stages (shuffle stages) where they transfer information between jobs. .

1.2 Shuffle

We will use MapReduce to explain the shuffle in more detail, but the main concepts still apply to Spark.

1.2.1 Shuffle Introduction

In the first stage of MapReduce, the map phase, the data is loaded onto different computers and computation is performed on it that results in a group of key value pairs. The final phase of MapReduce, the reduce phase, assumes that all key-value pairs with the same key are grouped together onto the same machine. We call this property the shuffle guarantee. Thus, the shuffle phase, an intermediate phase that the system handles internally, transfers

key value pairs to satisfy the shuffle guarantee.

Figure 1-1 display the inner workings of the shuffle phase in MapReduce. For instance, a programmer may want to count the number of letters in a distributed file. The mappers will each load part of the distributed file and count the number of letters in it. 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.

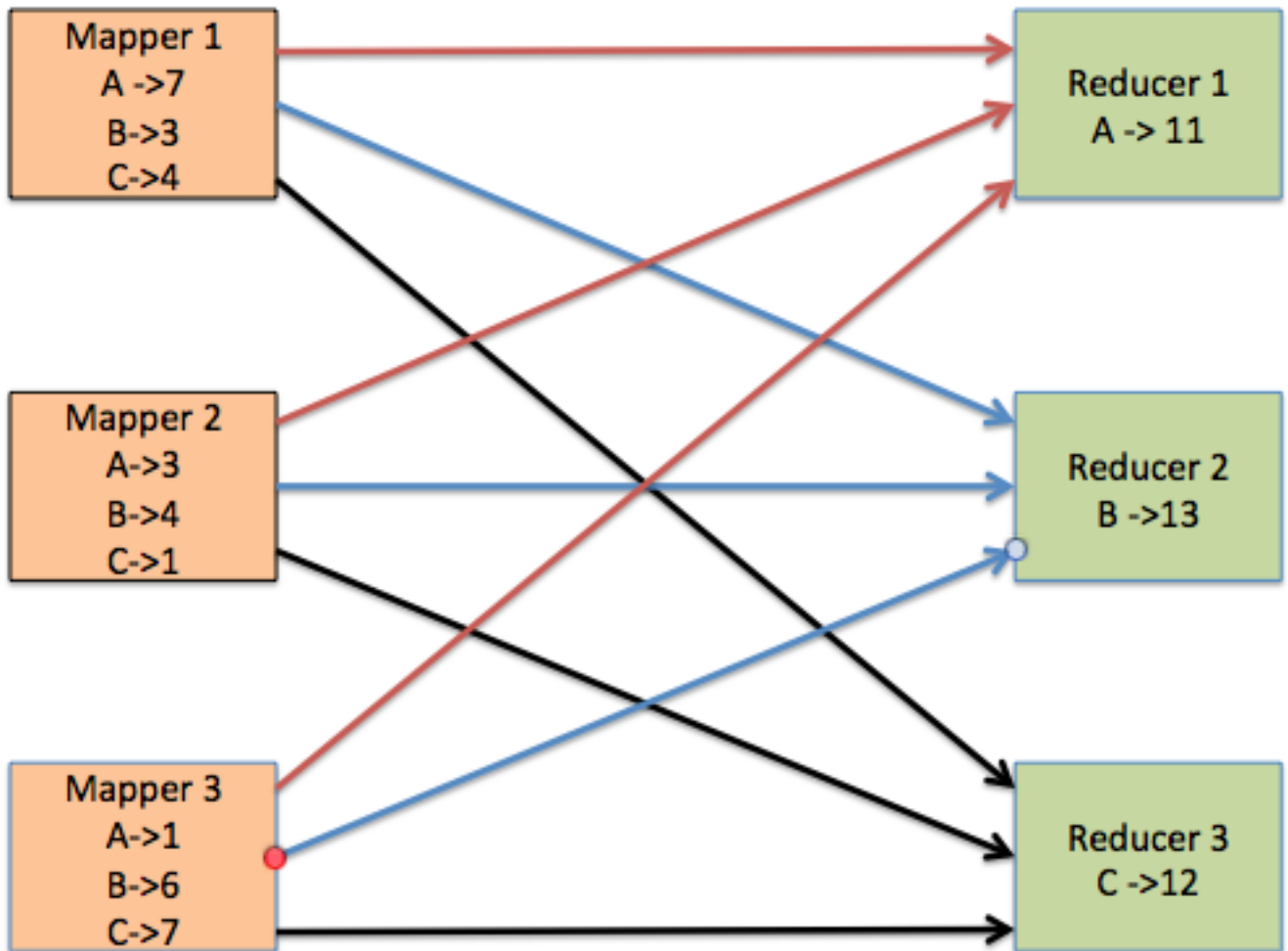


Figure 1-1: Shuffle for Letter Count in Map Reduce

Due of the huge amounts of keys, these systems do not transfer data on the granularity

of keys. Instead, they have the concept of partitions, which key value pairs with different keys. Programmers can pick different partitioning functions such as hash partitioning and range partitioning to map keys to partitions. Two keys that are identical are guaranteed to be mapped to the same partition. As long as all the mappers partition their data in the same way and send each the corresponding partitions to the same reducer, the system satisfies the shuffle guarantee.

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. A more balanced workload for reducers reduces network latency for transferring the data and also reduces the execution time for the slowest worker. Figure ??, depicts a shuffle scenario that results in unbalanced partitions. Each mapper partition gets sent to the reducer id equal to the partition id mod the number of reducers. This protocol in theory should result in pretty balanced reducers but is not guaranteed to. As depicted, Reducer 3 receives 40MB of data but Reducer 1 receives 90MB of data. However, if we knew the size of each partition after the mappers have run, we could more intelligently balance the reducers. As seen in Figure ??, with the same map output partitions, the system could attain complete balance of 60MB for each reducer.

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 tables by finding intersections between keys in respective columns. For instance, if we have Table1.1 and Table1.2 that we are trying to join based on the intersection of key1 and key2, the resulting output is Table1.3

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 actual implementation of joins in MapReduce is very similar to the shuffle scenario presented above. Instead of one dataset participating in the shuffle, two datasets participate in the shuffle and ensure that their corresponding partitions are both sent to the same reducer. Figure 1-4 details a shuffle join. For both datasets, all of the keys that mapped to partition 1 were sent to the reducer 1 and this happens respectively for the rest of the partitions.

1.3.3 Broadcast Join

The diagram above may seem to imply that mappers and reducers are different machines. However, this distinction is artificial and there are no separate machines for mappers and reducers. Therefore, not all data in the shuffle stage is transferred over the network. In Figure 1-1, if Mapper 1 and Reducer 1 were the same machine, the key value pair A=7 would be read locally and not have to be sent over the network.

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 broadcast join. Notice how the amount of data being transferred would be in the kilobytes 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, then 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.

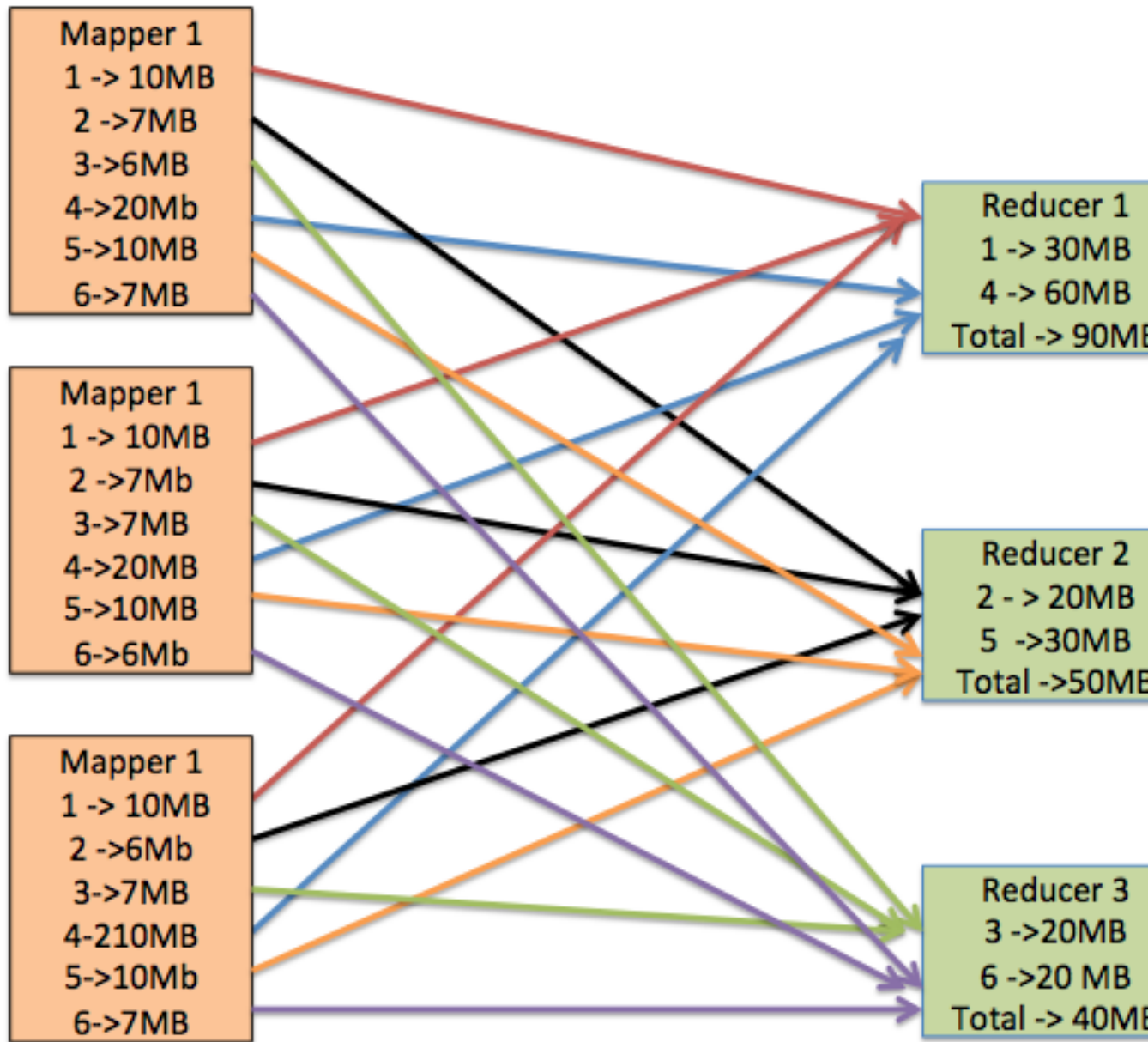


Figure 1-2: Unbalanced shuffle of partitions

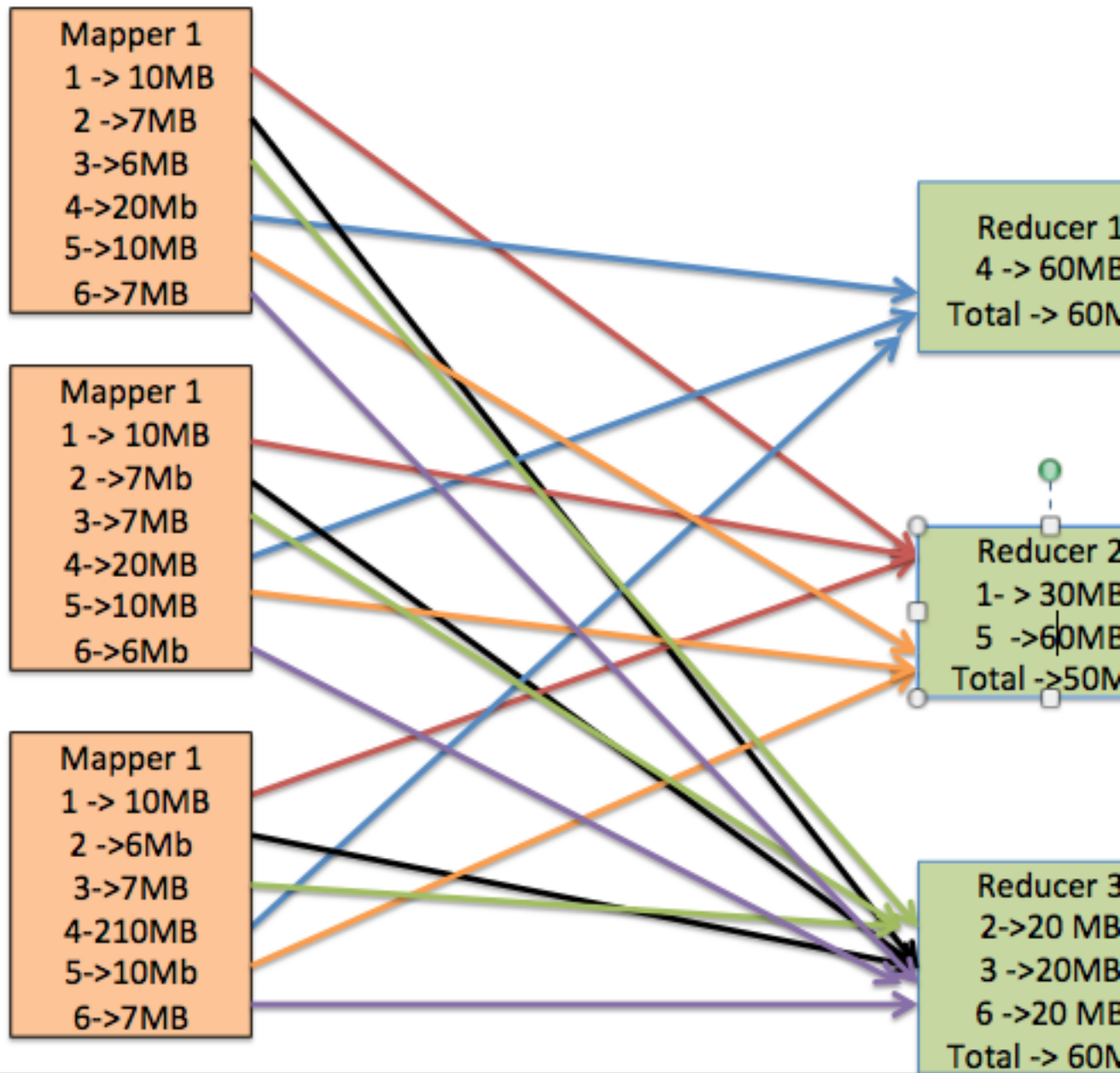


Figure 1-3: Balanced shuffle of partitions.

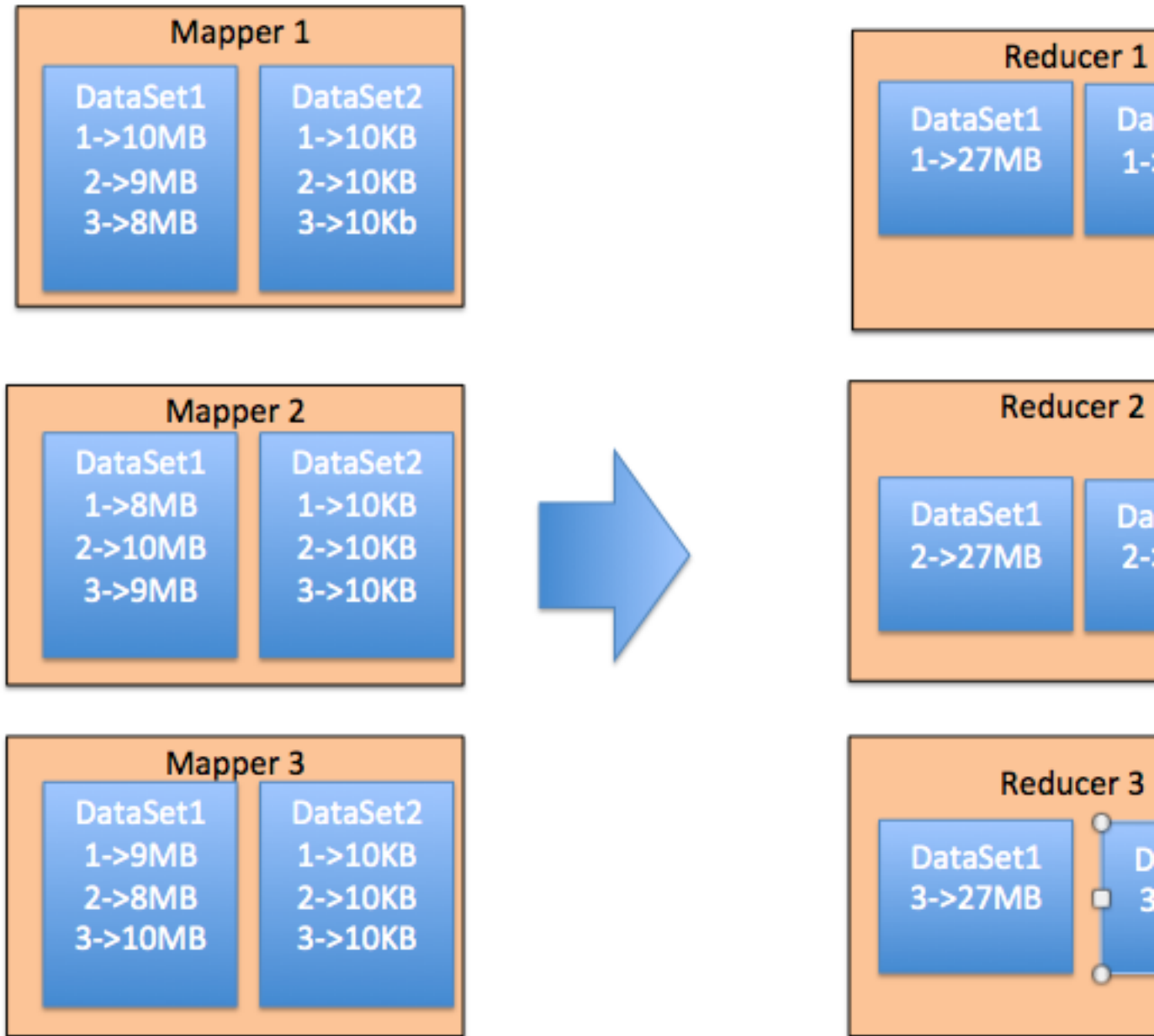


Figure 1-4: Typical Shuffle Join

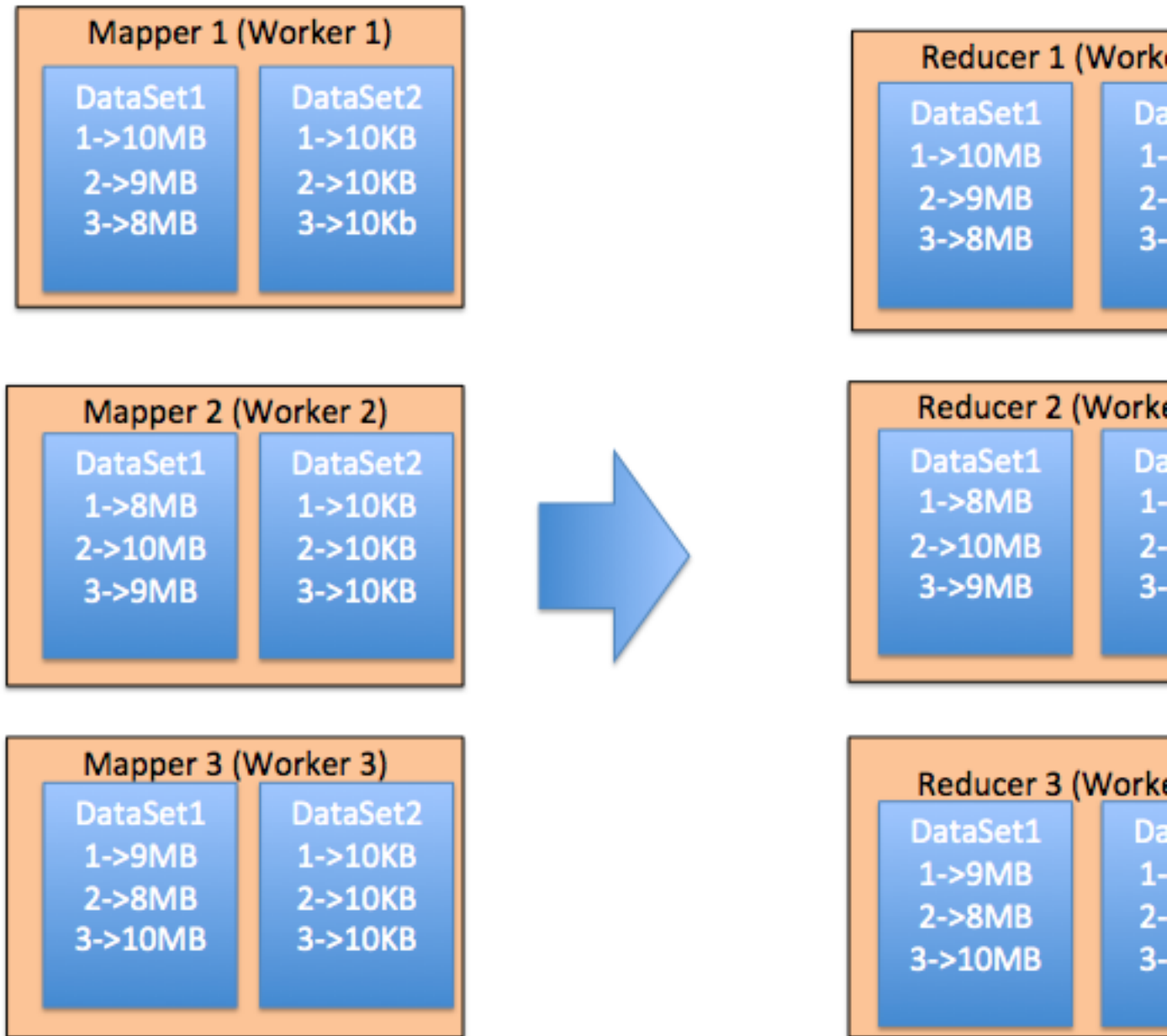


Figure 1-5: Broadcast Join

Chapter 2

Implementation

2.1 Spark

All of the code was being implemented in Spark. As mentioned before, Spark is the next generation of map reduce and has received a huge amount of support. Although the code was implemented in Spark, it could also be implemented in map reduce to achieve similar gains. Matei Zaharia added code that allowed the tracking of sizes of map output files.

2.2 ShuffledRDD

As mentioned before, the RDD is the main interface to interact with data in Spark. The RDD we developed we developed basically is a new version of ShuffledRDD. Our version of ShuffledRDD2 is basically a new version of the RDD. It takes in a shuffle dependency, which is basically a bunch of partitions that have finished the map output stage. As mentioned before, in a general ShuffledRDD just does basic logic, and does not do balancing. Our dependency basically takes in a number of reducers and then best tries to balance the partitions of the reducer. As this is more a proof of concept, our version of ShuffledRowRDD will only output consecutive partitions to one reducer. In other words, it is impossible for a reducer to have partitions 1 and 3, without having partition 2. In the future, we hope to expand the RDD so that it best balances the partitions.

2.3 Joins

2.3.1 ShuffleReader Changes

We added some flexibility to how shuffles are fetched. As mentioned in the shuffle section, we would like the bigger RDD to stay in place. The current interface allows a reducer to pick a specific partition from the mapper and autofetched it from all of the mappers. In other words, you could only request partition 1 from all the mappers. This would be extremely problematic for the broadcast join as we want none of the bigger RDD to move at all. The key part of the RDD interface is basically how a new reducer partition is computed from the old map partition. Thus, each of the reducer partitions is computed by requesting the necessary amount of partitions from the mappers as described above.

2.3.2 ShuffleJoinRDD and BroadcastJoin RDD

We implement two different type of RDD's, the ShuffleJoinRDD and the BroadcastJoinRDD. Both of these RDD's take two shuffle dependencies, which remember are basically the outputs of map stages, partitioned in a certain way. These dependencies must be partitioned in the same way. Otherwise, we have no way of ensuring that the two keys are in the same partition.

The ShuffleJoinRDD is implemented in a manner similar to the way to a regular shuffledRDD is. The shuffle dependency basically fetches a number of partitions of the shuffle dependency. It ensures that the partitions that it fetches are the same, i.e. reducer partition 1 will fetch dataset 1 partition 1 and dataset2 partition 1 from all of the workers. Once these partitions are fetched, we basically form a map with the smaller partition present and iterate through the bigger partition, seeing if there are keys present in this map. If so, we add this to ourput.

The BroadcastJoinRDD implements the broadcast shuffle that was explained above. For each reduce partition, it requests one local partition from the mapper from the bigRDD. To be clear, we use the new api and are just requesting one partition of the bigRDD from one

mapper. We then request all of the partitions from the smaller RDD, thus giving us all of the smaller RDD. We then use the same strategy to actually join them as the shuffle join rdd.

2.3.3 Joins in Spark SQL

Although the RDD interface can be used, many programmers and data analysts prefer not to use this interface and are more familiar with the spark interface. Thus, Spark offers a sql like interface. Within, this interface they offer a join operator and functions just like a join that was explained above. Although this code is written in a sql like statement, the code is still converted and executed by using RDD's.

Because we are not just using the RDD interface, the exact way we implement our code changes is a little more complicated. We only implement our code is a specific technique for sort merge join. One key change change is that we add a flag indicating whether our join is a sort merge join. Because the query planner determines the type of join, we only implement our join type for sort merge join.

Although the exact semantics for how a sort merge join can be found here, the sort merge join like the shuffle requires everything that needs to be compared together to be on the machine. The main difference is that when it finds intersections, instead of putting one of the datasets in a map like interface, it sorts both partitions, and then iterates through them and advances the partitions.

The following example in Figure 2-1 details the inner workings of how the query plan works. The first part of the plan is called the exchanges. The exchanges are where pretty much all of my changes happened. We create ShuffledRowRDDs which are pretty much equivalent to regular shuffle RDD's, but are slightly different as they represent rows as we are dealing with tables in this spark sql configuration. The program will compare partition 1 of our ShuffledRowRDD to partition 1 of the other ShuffledRowRDD. Therefore, we need to ensure that they match up accordingly. This is very similar to the key concepts

between the broadcast join and shuffled RDD. Within the shuffledRowRDD, we look at the total size of the partitions of each dataset. If one is smaller than a particular threshold and the other is not, then we do a broadcast join. Both the threshold and the ability to do the sort merge join are set through a flag that can be triggered in the conf field.

In the broadcast join, the shuffledRowRDD for the big RDD produced is basically the same as the initial RDD. However, for the smaller shuffledRowRDD, each partition contains all of the partitions from the data. We set the number of partitions it has to be equal to the number of partitions the smaller one has. The shuffle join is what happens by default without our code being implemented. It sets, the number of partitions for each ShuffledRowRDD to be some number that is set in the conf. It then ensures that partition 1 of the ShuffleRowRDD, i.e. will have the corresponding partition for partition one in shuffledRowRDD 1 and shuffledRowRDD 2.

The rest of the query plan is basically used for the sort merge join. The sort feature sorts the RDD. The ZippedPartitions part is used as we iterate through the two sorted lists for merging them.

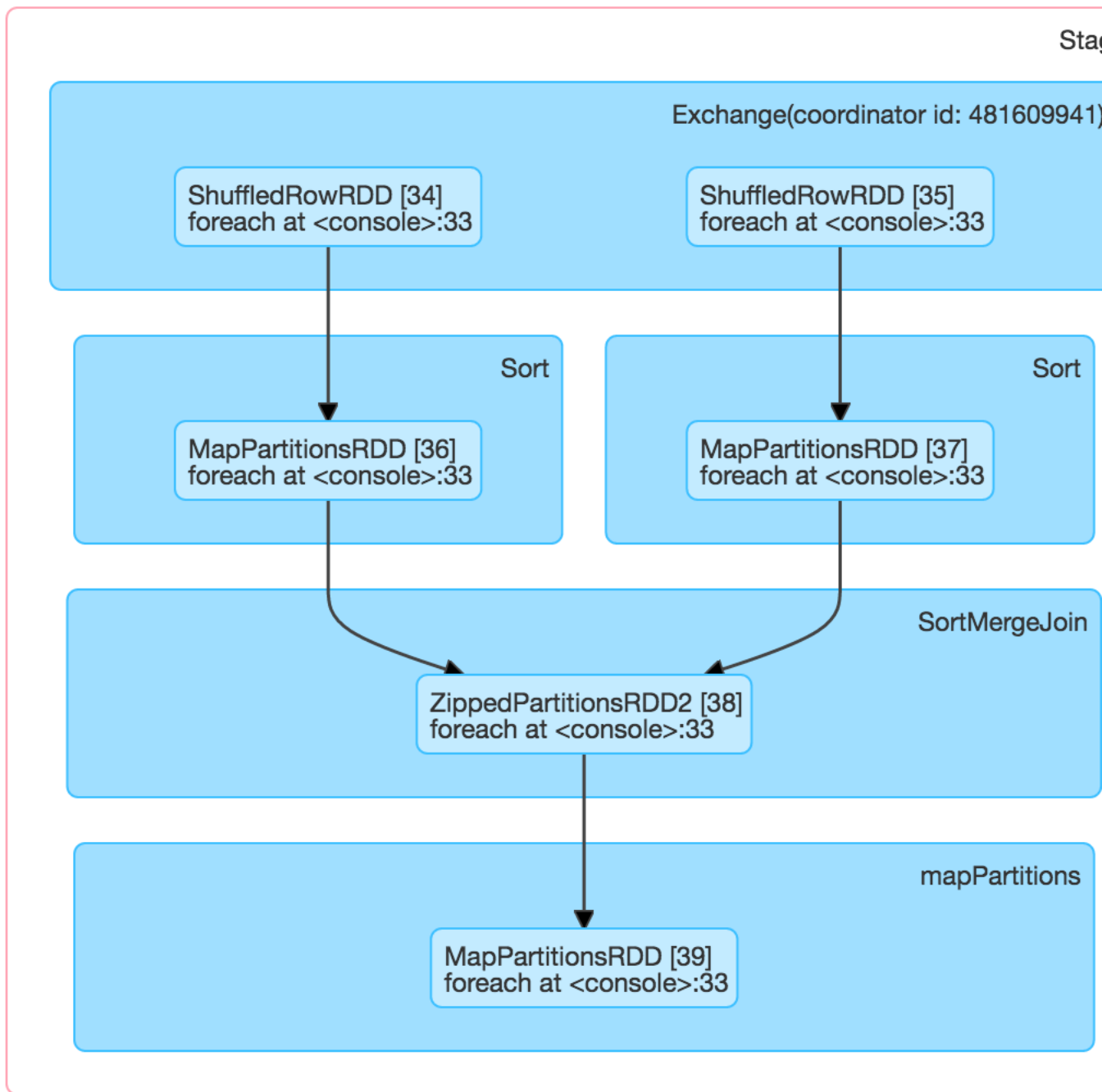


Figure 2-1: Plan for Sort Merge Join

Chapter 3

Experiment

3.1 Setup

All jobs were run using the spark/ec2 launch scripts. They were run on four aws m1.large machines. They were run ten times, with the last times being average.

3.2 Regular Shuffle

3.3 Broadcast and ShuffleJoinRDD

3.4 Spark SQL join

Chapter 4

Future Research and Conclusion

4.1 Future Research

4.1.1 Extension of Shuffle

The new version of the shuffle rdd is limited in a couple ways. First, each reducer can only fetch partitions consecutively, so allowing it to pick non-consecutive partitions could potentially improve performance.

Second, the current version only supports providing the number of reducers, but other formats such as specifying the maximum bytes a reducer can have, and then system automatically determines the number of reducers.

4.1.2 Extension to Join

First, we hijack the exchange framework to make the quickest possible change to allow our optimization, but we could probably do this in a cleaner fashion.

Second, the users has to statically pass in thresholds that determine when to switch between broadcast and shuffle joins. It would be useful if the system could determine this based on factors such as the size of the rdd s as well as additional info such as the ram of each machine as well as the network bandwidth.

Third, we either broadcast an entire RDD or default to a shuffle pattern. However, if RDD1 has a big partition 1 and a small partition2 and RDD2 has a small partition 1 and big

partition 2, we still do a shuffle. However, we could possibly save time by having RDD1 broadcast its partition 1 and RDD2 broadcasting its partition 2.

4.2 Conclusion

In conclusion, we show that improvements can be made to shuffle stage of Spark. Instead of predetermining our shuffle strategy, we can adapt it based on the output of the mappers. We show that we can use this for improvements in the regular shuffle, in joins with rdds, and in joins using in Spark Sql. Although we have shown gains, the work can be extended to show further possible gains.

Bibliography