CSx55: Distributed Systems [Hadoop]

Trying to have your cake and eat it too

Each phase pines for tasks with locality and their numbers on a tether

Alas within a phase, you get one, but not the other

Who gets what?

Stay tuned to find out

Shrideep Pallickara
Computer Science
Colorado State University

Frequently asked questions from the previous class survey

- Can you rebuild MapReduce computations solely from the reducers, if for some reason the mappers are failing continually?
- Is the combiner solely for optimization?
- Why do map and reduce in Hadoop throw `InterruptedException`?

Topics covered in today's lecture

- Hadoop

MapReduce tasks & Split strategies

Hadoop divides the input to a MapReduce job into fixed-sized pieces

- These are called **input-splits** or just splits
- Creates **one map task per split**
  - Runs *user-defined map function* for each **record** in the split

Split strategy: Having many splits

- Time taken to process split is small compared to processing the whole input
- Quality of **load balancing** increases as splits become *fine-grained*
  - Faster machines process proportionally more splits than slower machines
  - Even if machines are identical, this feature is desirable
    - Failed tasks get relaunched, and there are other jobs executing concurrently

Split strategy: If the splits are too small

- **Overheads** for managing splits and map task creation dominates total job execution time

- Good split size tends to be an HDFS **block**
  - This could be changed for a cluster or specified when each file is created

Scheduling map tasks

- Hadoop does its best to run a map task on the *node where input data resides* in HDFS
  - **Data locality**
- What if all three nodes holding the HDFS block replicas are busy?
  - Find free map slot on node in the same rack
  - Only when this is not possible, is an off-rack node utilized
    - Inter-rack network transfer

Why the optimal split size is the same as the block size ...

- Largest size of input that can be stored on a single node
- If split size spanned two blocks?
  - Unlikely that any HDFS node has stored *both* blocks
  - Some of the split *will have to be transferred* across the network to node running the map task
    - Less efficient than operating on local data without the network movement

Managing Outputs

Map task outputs

- Stored on the local disk
  - Not HDFS

- Once the job is complete, **intermediate map outputs are thrown away**
  - Storing in HDFS with replication is an overkill

Reduce tasks do not have the advantage of data locality

- Input to a single reduce task
  - Output from **all the mappers**
  - Sorted map outputs transferred over the network to node where reduce task is running
    - **Merged and then passed** to the reduce function
- Output of reduce task stored on HDFS
  - One replica of block is stored on local node, other replicas are stored on off-rack nodes

Number of reduce tasks

- Not governed by the size of the input
- Specified independently

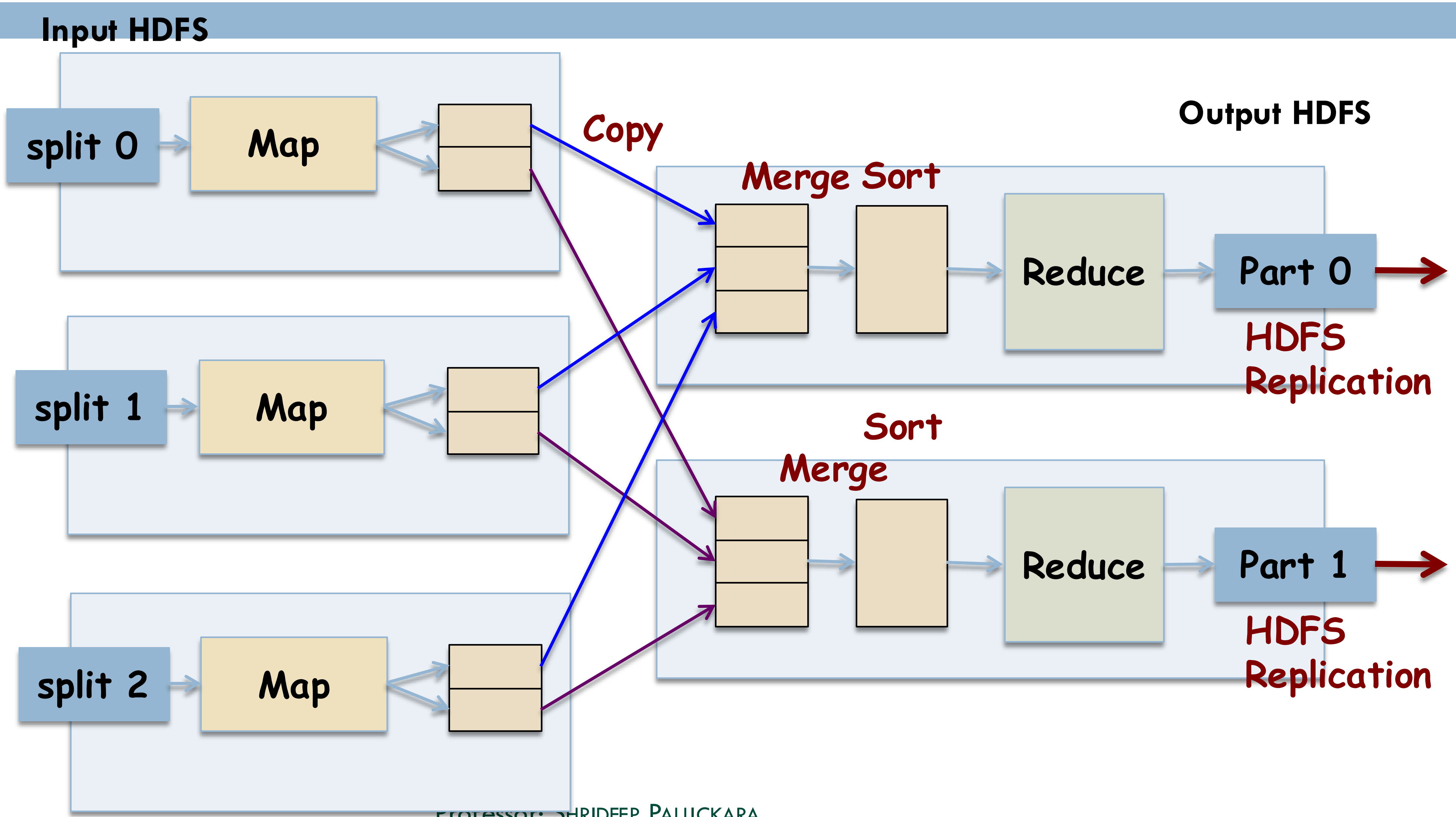
When there are multiple reducers

- Maps **partition** their outputs
  - One partition for ***each*** reduce task
  - There can be many keys in each partition
  - Records for a given key are all in the same partition

- Partitioning controlled with a *partitioning function*
  - Default uses a hash function to bucket the key space

Data Flow

MapReduce Dataflow



In Hadoop a Map task has 4 phases

- Record reader
- Mapper
- Combiner
- Partitioner

Map task phases: **Record Reader**

- **Translates** input splits into records
- Parse data into records, but does not parse the record itself
- Passes the data to the mapper in the form of a key/value pair
  - **key** in this context is *positional information*
  - **value** is the chunk of data that comprises a record

Map task phases: **Map**

- **User-provided code** is executed on each key/value pair from the record reader
- This user-code produces *zero or more* new key/value pairs, called the **intermediate pairs**
  - *key* is what the data will be grouped on and value is the information pertinent to the analysis in the **reducer**
  - Choice of key/value pairs is critical and not arbitrary

Map task phases: **Combiner**

- Can **group data** in the map phase
- Takes the intermediate keys from the mapper and applies a user-provided method to *aggregate values* in the small scope of that one mapper
- *Significantly reduces the amount of data* that has to move over the network
  - Sending ("hello", 3) requires fewer bytes than sending ("hello", 1) three times over the network

Map task phases: **Partitioner** [1/2]

- Takes the intermediate key/value pairs from the mapper (or combiner) and splits them up into **shards**, one shard per reducer
- Default: key.hashCode() % (number of reducers)
  - Randomly distributes the keyspace *evenly* over the reducers
  - But still ensures that keys with the same value in different mappers end up at the same reducer

Map task phases: **Partitioner** [2/2]

- Partitioner can be customized (e.g., for sorting)
  - Changing the partitioner is rarely necessary

- The partitioned data is written to the local file system for each map and waits to be **pulled** by its respective reducer

In Hadoop a Reduce task has 4 phases

- Shuffle
- Sort
- Reducer
- Output format

Reduce task phases: **Shuffle and sort**

- Shuffle
  - Takes the output files written by all of the partitioners and downloads them to the local machine in which the reducer is running
- Sort
  - Individual data pieces are then **sorted by key** into one larger data list
  - **Groups equivalent keys together** so that their values can be iterated over easily in the reduce task

Reduce task phases: **Shuffle and sort**

- This phase is **not customizable** and the framework handles everything automatically
- The only control a developer has is how the keys are sorted and grouped by specifying a custom `Comparator` object

Reduce task phases: **Reducer**

- Takes the grouped data as input and runs a reduce function **once per key grouping**
- The function is passed the key and an iterator/iterable over all of the values associated with that key
  - A wide range of processing can happen in this function: data can be aggregated, filtered, and combined etc.
- Once the reduce function is done, it sends zero or more key/value pairs to the final step, the output format
- N.B.: map & reduce functions will change from job to job

Reduce task phases: **Output format**

- Translates the final key/value pair from the reduce function and writes it out to a file using a record writer
- By default:
  - Separate the key and value with a tab
  - Separates records with a newline character
- Can typically be customized to provide richer output formats
  - But in the end, the data is written out to HDFS, regardless of format

COMBINER FUNCTIONS

Combiner functions

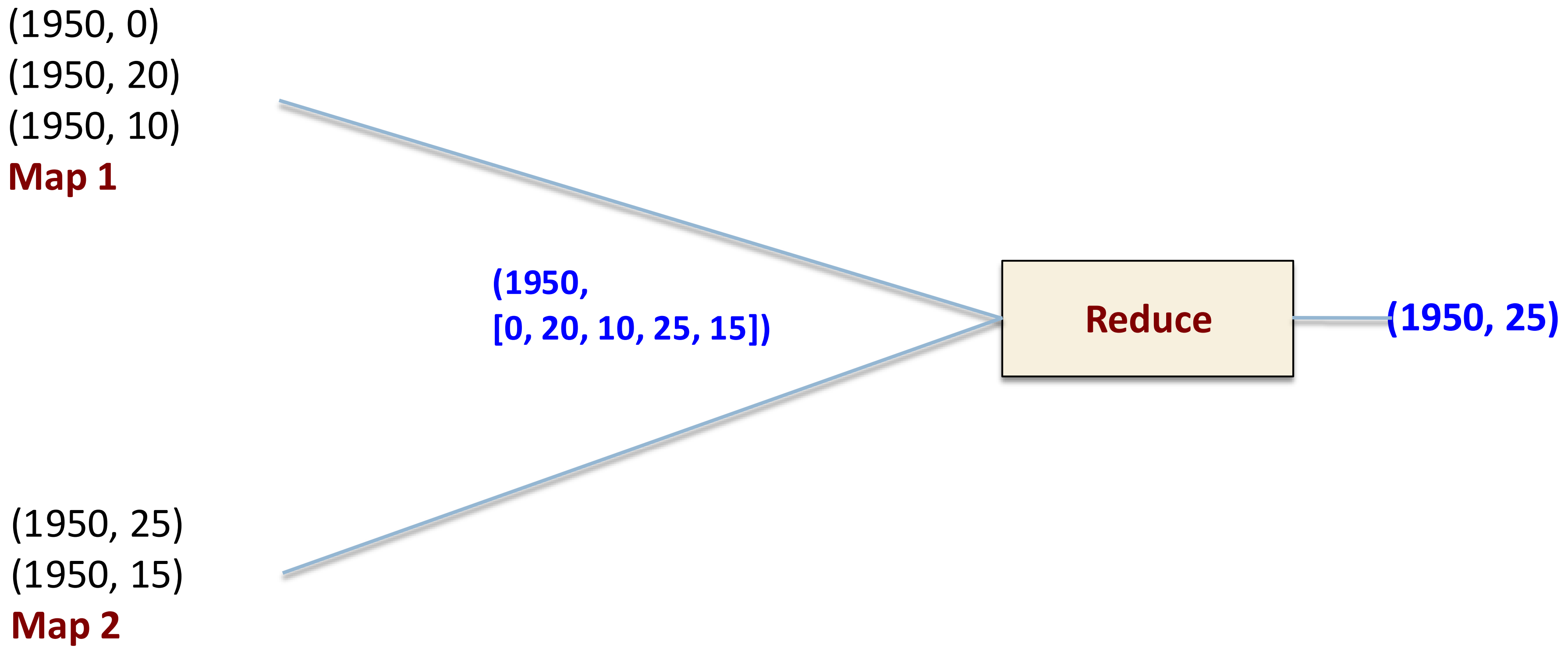
- Many MapReduce jobs are limited by the available network bandwidth
  - Framework has mechanisms to *minimize the data transferred* between map and reduce tasks

- A **combiner** function is run <u>on the map output</u>
  - Combiner output fed to the reduce task

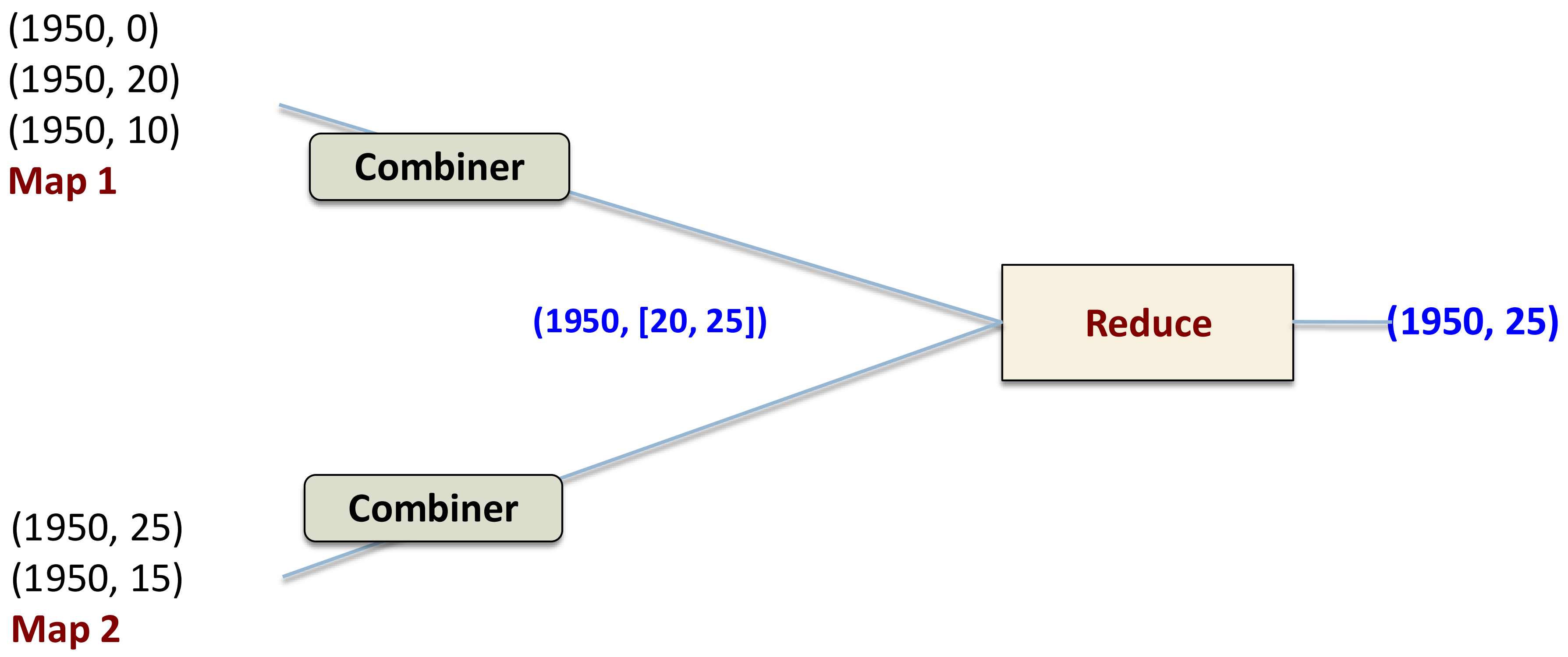
Combiner function

- No guarantees on *how many times* Hadoop will call this on a map output record
  - The combiner should, however, result in the same output from the reducer
- **Contract** for the combiner **constrains the type of function** that can be used

Combiner function: Let's look at the maximum temperature example [1/2]



Combiner function: Let's look at the maximum temperature example [2/2]



A closer look at the function calls

- max(0, 20, 10, 25, 15) =

  max (max(0, 20, 10), max(25, 15)) =

  max (20, 25) = 25

- Functions with this property are called **commutative** *and* **associative**
  - Commutative: Order of operands (5+2) = (2 + 5)
    - Division and subtraction are not commutative
  - Associative: Order of operators 5 x (5x3) = (5x5)x3
    - Vector cross products are not

Not all functions posses the commutative and associative properties

- What if we were computing the mean temperatures?
- We can cannot use mean as our combiner function

mean(0, 20, 10, 25, 15) = 14

BUT

mean(mean(0, 20, 10), mean(25, 15)) =

mean(10, 20) = 15

Combiner: Summary

- The combiner **does not replace** the reduce function
  - Reduce *is still needed* to process records from different maps

- But it is useful for **cutting down traffic** from maps to the reducer

Specifying a combiner function

```
public class MaxTemperatureWithCombiner {

  public static main(String[] args) throws Exception {
    Job job = Job.getInstance();
    job.setJarByClass(MaxTemperature.class);
    job.setJobName("Max temperature");

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.setMapperClass(MaxTemperatureMapper.class);
    job.setCombinerClass(MaxTemperatureReducer.class);
    job.setReducerClass(MaxTemperatureReducer.class);

    job.setOutputKey(Text.class);
    job.setOutputValueClass(IntWritable.class);

    System.exit(job.waitForCompletion(true) ? 0: 1);

  }
}
```

ANOTHER EXAMPLE (COMBINER)

Another example with StackOverflow [1/2]

- Given a list of user's comment determine the average comment length per-hour

- To calculate average we need two things:
  - Sum values that we want to average
  - Number of values that went into the sum

Another example with StackOverflow [2/2]

- Reducer can do this very easily by iterating through each value in the set and adding to a running sum while keeping count
- But if you do this you cannot use the reducer as your combiner!
  - Calculating an average is not an associative operation
    - You cannot change the order of the operators
    - mean(0, 20, 10, 25, 15) = 14 BUT ..
    - mean(mean(0, 20, 10), mean(25, 15)) = mean(10, 20) = 15

Approach to ensuring code reuse at the combiner

- Mapper will output two columns of data
  - Count and average
- Reducer will multiply "count" field by the "average" field to add to a running count *and* add "count" to the running count
  - Then divide the running sum with running count
    - Output the count with the calculated average

Mapper code

```
public static class AverageMapper extends
    Mapper < Object, Text, IntWritable, CountAverageTuple > {

   private CountAverageTuple outCountAverage = new CountAverageTuple();
   public void map( Object key, Text value, Context context)
      throws IOException, InterruptedException {
     Map < String, String > parsed =
            MRDPUtils.transformXmlToMap( value.toString());
     String strDate = parsed.get(" CreationDate");
     String text = parsed.get(" Text");
     // get the hour this comment was posted in
     Date creationDate = frmt.parse( strDate);
    outHour.set( creationDate.getHours());

    outCountAverage.setCount( 1);
    outCountAverage.setAverage( text.length());

     // write out the hour with the comment length
     context.write( outHour, outCountAverage);
   }
}
```

Reducer code

```
public class AverageReducer extends Reducer < IntWritable,
CountAverageTuple, IntWritable, CountAverageTuple > {
    private CountAverageTuple result = new CountAverageTuple();

    public void
    reduce(IntWritable key, Iterable < CountAverageTuple > values,
        Context context) throws IOException, InterruptedException {
       float sum = 0; float count = 0;

       // Iterate through all input values for this key
       for (CountAverageTuple val : values) {
          sum + = val.getCount() * val.getAverage();
          count + = val.getCount();
       }
       result.setCount( count);
       result.setAverage( sum / count);
       context.write( key, result);
     }
}
```

Data flow for the average example

Input key	Input Value	
Hour	**Count**	**Average**
4	1	10
4	1	8
4	1	21
3	1	1
3	1	19
9	1	7
9	1	12

Group 1

Group 2

<u>Setting</u>:
Combiner executes over Groups 1 and 2
DOES NOT execute on the last two rows

Combiner Output/ Reducer Input

Output key	Output Value	
Hour	**Count**	**Average**
3	2	10
4	3	13
9	1	7
9	1	12

The contents of this slide set are based on the following references

- *Tom White. Hadoop: The Definitive Guide. 3rd Edition. Early Access Release. O'Reilly Press. ISBN: 978-1-449-31152-0.* Chapters [2 and 3].
- *MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems.* 1st Edition. Donald Miner and Adam Shook. O'Reilly Media ISBN: 978-1449327170. [Chapter 1-3]