Input/Output formats in Hadoop

RECAP

MR Job Configuration

Job object

For every MR Job, a Job object is setup in the **Main Class**

The Job Object has a bunch of **Configurable Options**

RECAP

MR Job Configuration

Job object

All of these have a default configuration

The user can control them by setting them to a specific class

Configurable Options

```
setInputFormatClass()
setMapOutputKeyClass()
setMapOutputValueClass()
setOutputKeyClass()
setOutputValueClass()
setMapperClass()
setCombinerClass()
setPartitionerClass()
setSortComparatorClass()
setGroupingComparatorClass()
setReducerClass()
setOutputFormatClass()
```

RECAP

MR Job Configuration

Job object

Input and Output formats
are by default set to

TextInputFormat.Class

TextOutputFormat.Class

Configurable Options

`setInputFormatClass()`

`setMapOutputKeyClass()`
`setMapOutputValueClass()`
`setOutputKeyClass()`
`setOutputValueClass()`
`setMapperClass()`
`setCombinerClass()`
`setPartitionerClass()`
`setSortComparatorClass()`
`setGroupingComparatorClass()`
`setReducerClass()`

`setOutputFormatClass()`

RECAP

MR Job Configuration

Job object

With this option input is expected to be in the form of text files

Outputs will also be text files

Configurable Options

`setInputFormatClass()`

TextInputFormat.Class

`setOutputFormatClass()`

TextOutputFormat.Class

RECAP

MR Job Configuration

Configurable Options

Job object

You can set this to any class that extends the InputFormat class

`setInputFormatClass()`

TextInputFormat.Class

RECAP

MR Job Configuration

Job object

This class decides how to break up the input files into InputSplits

Each InputSplit will be processed by 1 mapper

Configurable Options

`setInputFormatClass()`

InputFormat class

RECAP

MR Job Configuration

Job object

This class will also read each record in the input file and convert it to a <key,value> pair for the mapper

Configurable Options

`setInputFormatClass()`

InputFormat class

RECAP

MR Job Configuration

Job object

Similarly, for the output you can use any class that extends the OutputFormat class

Configurable Options

`setInputFormatClass()`

InputFormat class

`setOutputFormatClass()`

OutputFormat class

InputFormat class

OutputFormat class

Let's talk about the different classes available in Hadoop that implement these interfaces

InputFormat class

We'll go through the different InputFormat classes

There are corresponding OutputFormat classes for each of them

InputFormat class

The FileInputFormat class
extends InputFormat

InputFormat class

FileInputFormat class

All InputFormat classes that deal with files extend this class

Ex: TextInputFormat extends FileInputFormat and is used for text files

FileInputFormat class

This class has a few methods that are used to set the input file path

```
public static void addInputPath(Job job, Path path)
```

This the method we call to add a single input path that will be used to read the input files

FileInputFormat class

```
public static void addInputPath(Job job, Path path)
```

This adds a Path to the list of Paths that the MR job will look through

FileInputFormat class

```
public static void addInputPaths(Job job, String commaSeparatedPaths)
```

You can add multiple Paths at once using this method

FileInputFormat class

```
public static void setInputPaths(Job job, Path... inputPaths)

public static void setInputPaths(Job job, String commaSeparatedPaths)
```

Using these methods you can set the complete list of Paths at once

These will override any Paths you have set before

FileInputFormat class

Once you've set a list of input paths, the MR job will take all the files in those paths as input

FileInputFormat class

```
public static void setInputDirRecursive(Job job, boolean inputDirRecursive)
```

Setting this option to true will make the MR job look through all the files in the subdirectories in the specified paths as well

FileInputFormat class

The class also specifies how Input Splits are created

1 InputSplit is created for each block in each file

FileInputFormat class

MR jobs with a large number of small files will not be very efficient

FileInputFormat class

Each mapper would only run on a very small amount of data

FileInputFormat class

Large files are much more efficient for storage and processing in Hadoop

FileInputFormat class

The InputSplits that are created have methods to access the file's information

```
getPath()

getStart()

getLength()
```

You can use the getInputSplit() method of the Mapper to get the current split

FileInputFormat class

The InputSplits that are created have methods to access the file's information

```
getPath()
getStart()
getLength()
```

Then call one of these methods to get the file info

FileInputFormat class

`getPath()`	The Path of the file
`getStart()`	The byte offset of the split from the start of the file
`getLength()`	The size of the split in bytes

FileInputFormat class

Here are a few commonly used FileInputFormat classes

TextInputFormat

SequenceFileInputFormat

TextInputFormat

This is the default InputFormat class

TextInputFormat

It's used for reading text files

Each line is treated as a record

For each record a <lineNum, line> pair is generated for the Mapper

TextInputFormat

The input type parameters for the Mapper should be set to

LongWritable, Text

FileInputFormat class

Here are a few commonly used FileInputFormat classes

TextInputFormat

SequenceFileInputFormat

FileInputFormat class

Here are a few commonly used FileInputFormat classes

TextInputFormat

SequenceFileInputFormat

SequenceFileInputFormat

Sequence Files are binary files

They are used to store key, value pairs in a binary format

SequenceFileInputFormat

Here is the general format of
1 record in a Sequence file

Record Length	Key Length	Key	Value

SequenceFileInputFormat

We mentioned that it's preferable to store/process large files in Hadoop

SequenceFileInputFormat

That way each input split would not have too little data to be processed

SequenceFileInputFormat

Sequence files can be used as containers for small files

SequenceFileInputFormat

Record Length	Key Length	Key	Value

The key could be file metadata (name etc)

The value contains the entire file contents

SequenceFileInputFormat

The key and value types of a Sequence File should be consistent with the Mappers Input types

MultipleInputs Class

MultipleInputs Class

What if you want to use multiple input formats, and use different mappers for each of them?

MultipleInputs Class

Ex: A text file processed with Map1 class,
A sequence file with Map2 class

MultipleInputs Class

A text file -> Map1 class,
A sequence file -> Map2 class

Use MultipleInputs instead of setInputFormatClass()

MultipleInputs Class

A text file -> Map1 class,

A sequence file -> Map2 class

```
MultipleInputs.addInputPath(job, new Path(args[0]),
TextInputFormat.class, Map1.class);
```

MultipleInputs Class

A text file -> Map1 class,

A sequence file -> Map2 class

```
MultipleInputs.addInputPath(job, new Path(args[0]), TextInputFormat.class,
Map1.class);

MultipleInputs.addInputPath(job, new Path(args[1]),
SequenceFileInputFormat.class, Map2.class);
```

InputFormat class

OutputFormat class

Let's quickly walk through
the different
OutputFormat classes

OutputFormat class

TextOutputFormat is the default

OutputFormat class

TextOutputFormat

The output will be a text file with a Key Value pair in each line

OutputFormat class

TextOutputFormat

By default the Key and Value are separated by tab

OutputFormat class

TextOutputFormat

You can set your own separator while configuring the Job

```
job.getConfiguration().set("mapreduce.output.textoutputformat.separator", " | ");
```

OutputFormat class

SequenceFileOutputFormat is used to write out binary files

Customized Partitioning

Customized Partitioning

Let's say you have the output from a Word Count job

```
index    14
indexing         12
into     10
inverter         5
is       24
key      6
key-value        6
large    7
list     6
machine 5
machines         8
map      6
master   5
not      7
of       45
on       8
one      8
or       8
phase    5
reduce   8
segment 11
shown    5
term     14
terms    6
that     17
the      61
this     8
to       33
we       12
```

Customized Partitioning

You'd like to separate this output based on the Count

```
index    14
indexing         12
into     10
inverter         5
is       24
key      6
key-value        6
large    7
list     6
machine 5
machines         8
map      6
master   5
not      7
of       45
on       8
one      8
or       8
phase    5
reduce   8
segment 11
shown    5
term     14
terms    6
that     17
the      61
this     8
to       33
we       12
```

Customized Partitioning

Objective: Collect all words with Count <10 in 1 file and >10 in another

index	14
indexing	12
into	10
inverter	5
is	24
key	6
key-value	6
large	7
list	6
machine	5
machines	8
map	6
master	5
not	7
of	45
on	8
one	8
or	8
phase	5
reduce	8
segment	11
shown	5
term	14
terms	6
that	17
the	61
this	8
to	33
we	12

Customized Partitioning

Write a MapReduce which takes in this file as the input

```
index    14
indexing         12
into     10
inverter         5
is       24
key      6
key-value        6
large    7
list     6
machine 5
machines         8
map      6
master   5
not      7
of       45
on       8
one      8
or       8
phase    5
reduce   8
segment 11
shown    5
term     14
terms    6
that     17
the      61
this     8
to       33
we       12
```

Customized Partitioning

Use the Mapper to convert
Word, Count -> Count, Word

```
index    14
indexing         12
into     10
inverter         5
is       24
key      6
key-value        6
large    7
list     6
machine 5
machines         8
map      6
master   5
not      7
of       45
on       8
one      8
or       8
phase    5
reduce   8
segment 11
shown    5
term     14
terms    6
that     17
the      61
this     8
to       33
we       12
```

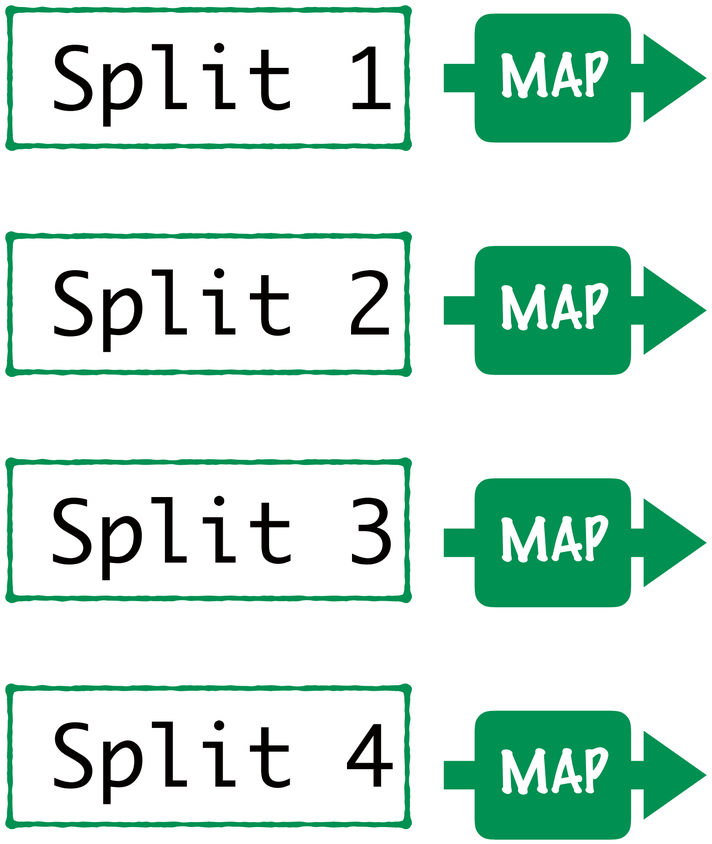
Customized Partitioning

Then you write your own Partitioner class to partition based on Count

```
index    14
indexing         12
into     10
inverter         5
is       24
key      6
key-value        6
large    7
list     6
machine  5
machines         8
map      6
master   5
not      7
of       45
on       8
one      8
or       8
phase    5
reduce   8
segment  11
shown    5
term     14
terms    6
that     17
the      61
this     8
to       33
we       12
```

RECAP

Shuffle and Sort

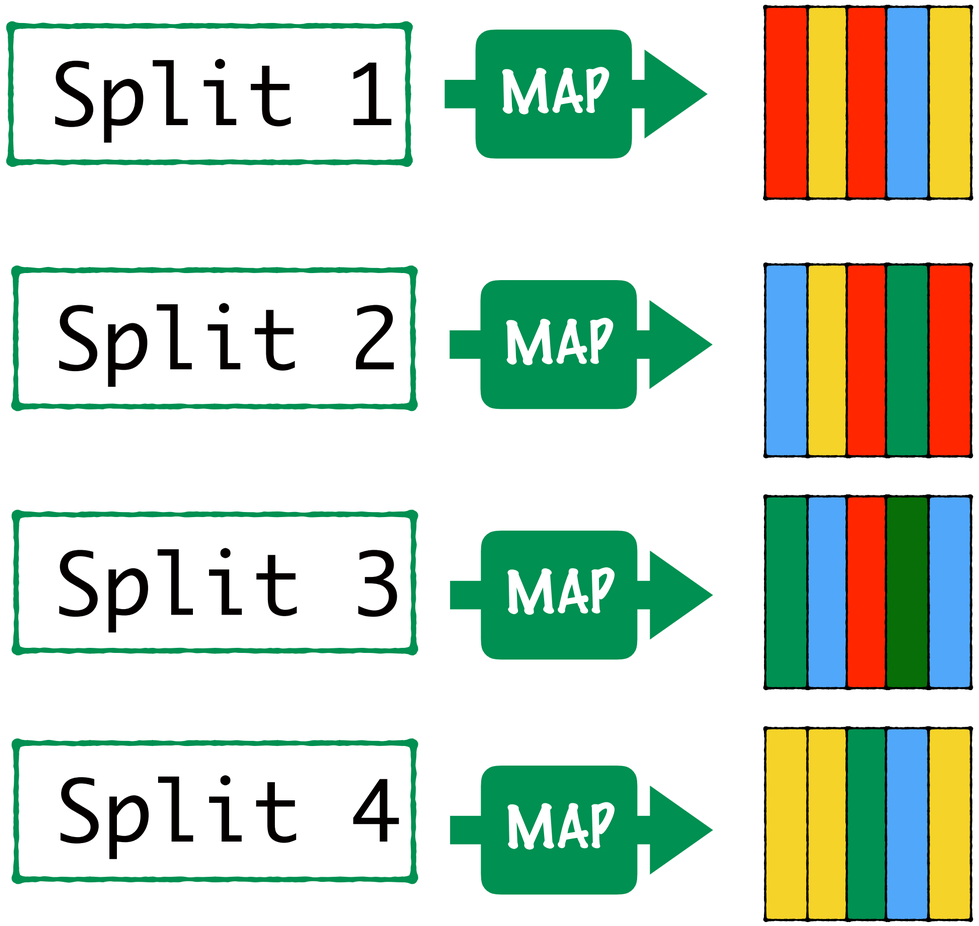


Let's say our input file had been split into 4 blocks by HDFS

There will 4 mappers

RECAP

Shuffle and Sort



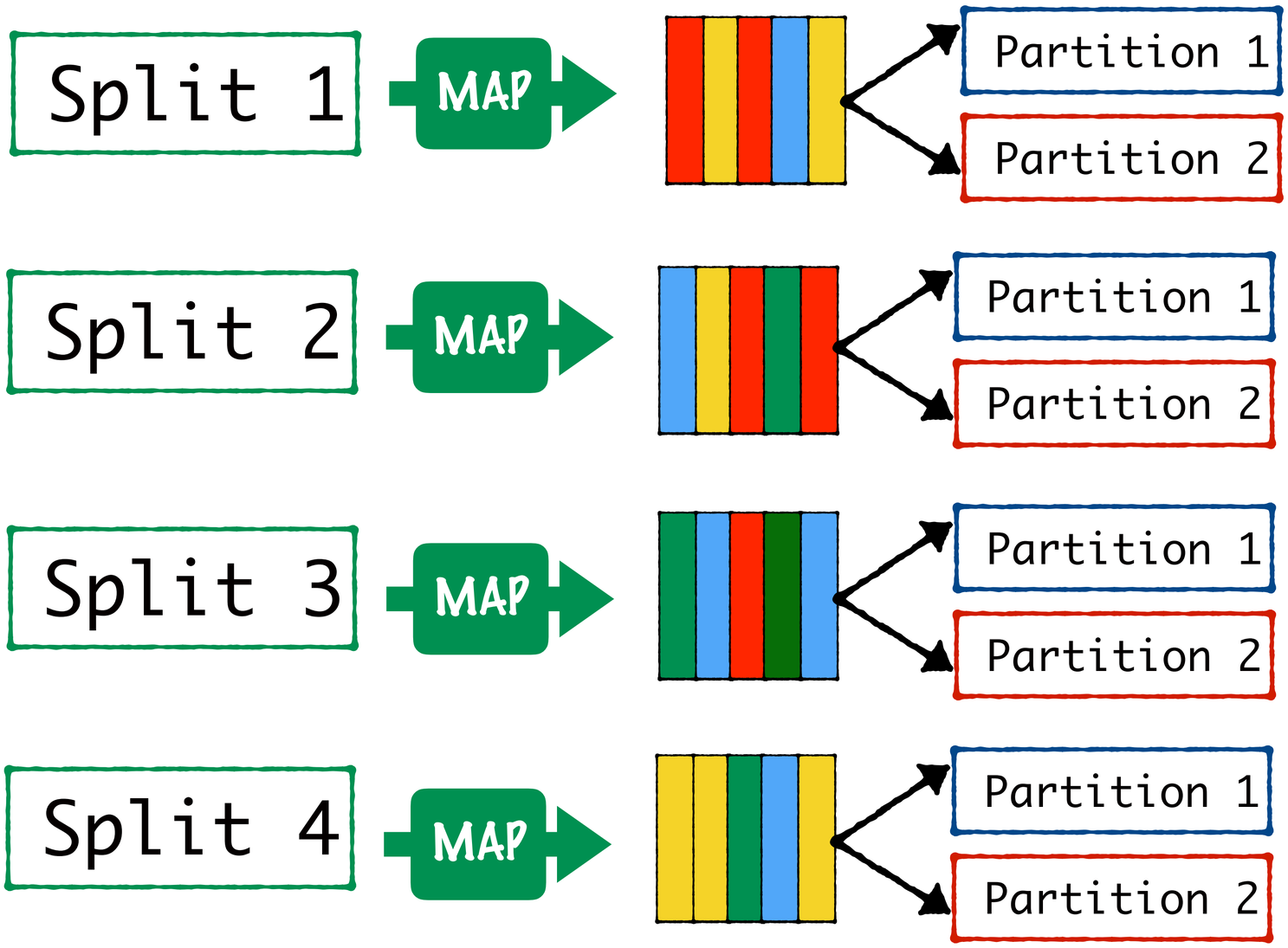
The mappers output <Key, Value> pairs

Each block here represents a <Key, Value> pair

The color represents the key

RECAP

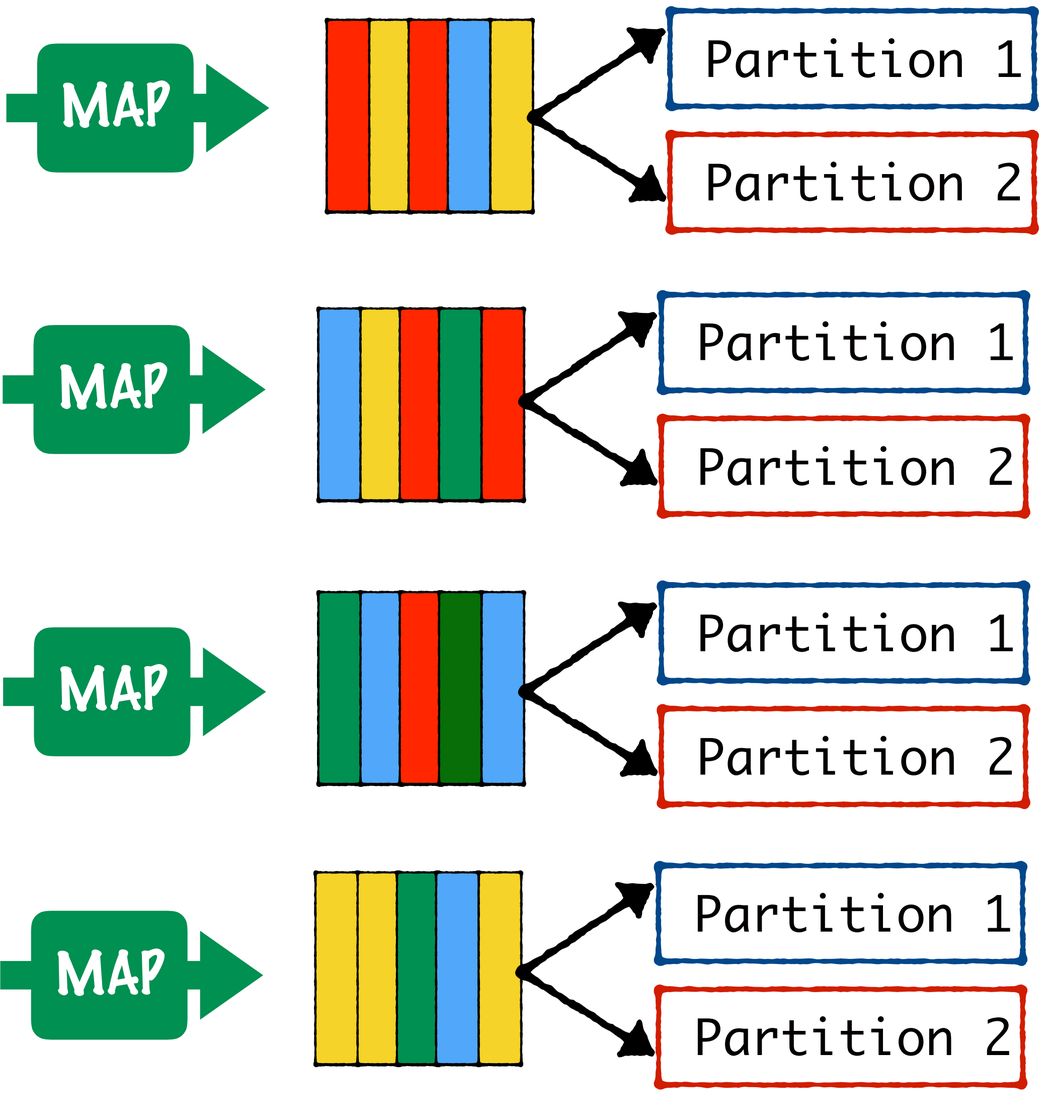
Shuffle and Sort



When there are 2 reducers, the output of each map is first partitioned into 2 partitions

RECAP

Shuffle and Sort

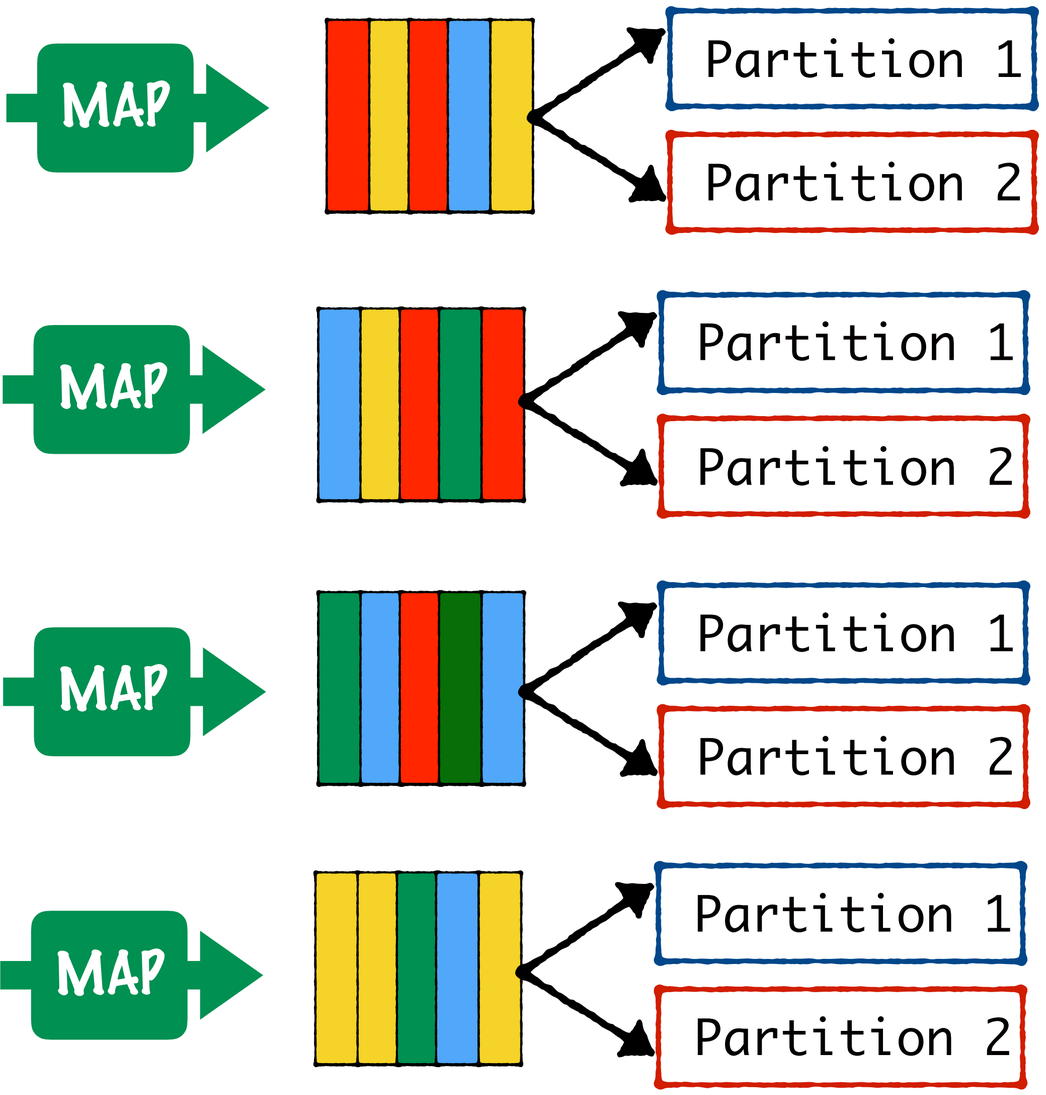


The map output is assigned to a partition based on the key

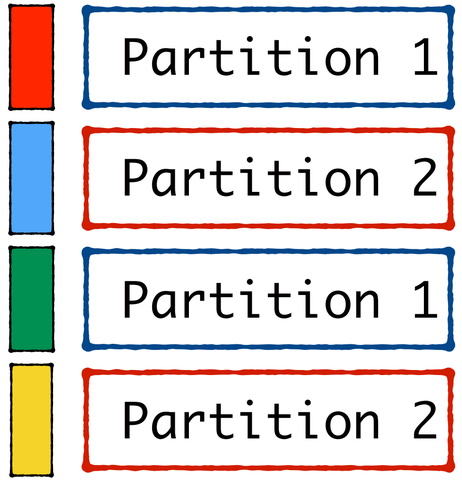
All values with the same key must be assigned to the same partition

RECAP

Shuffle and Sort

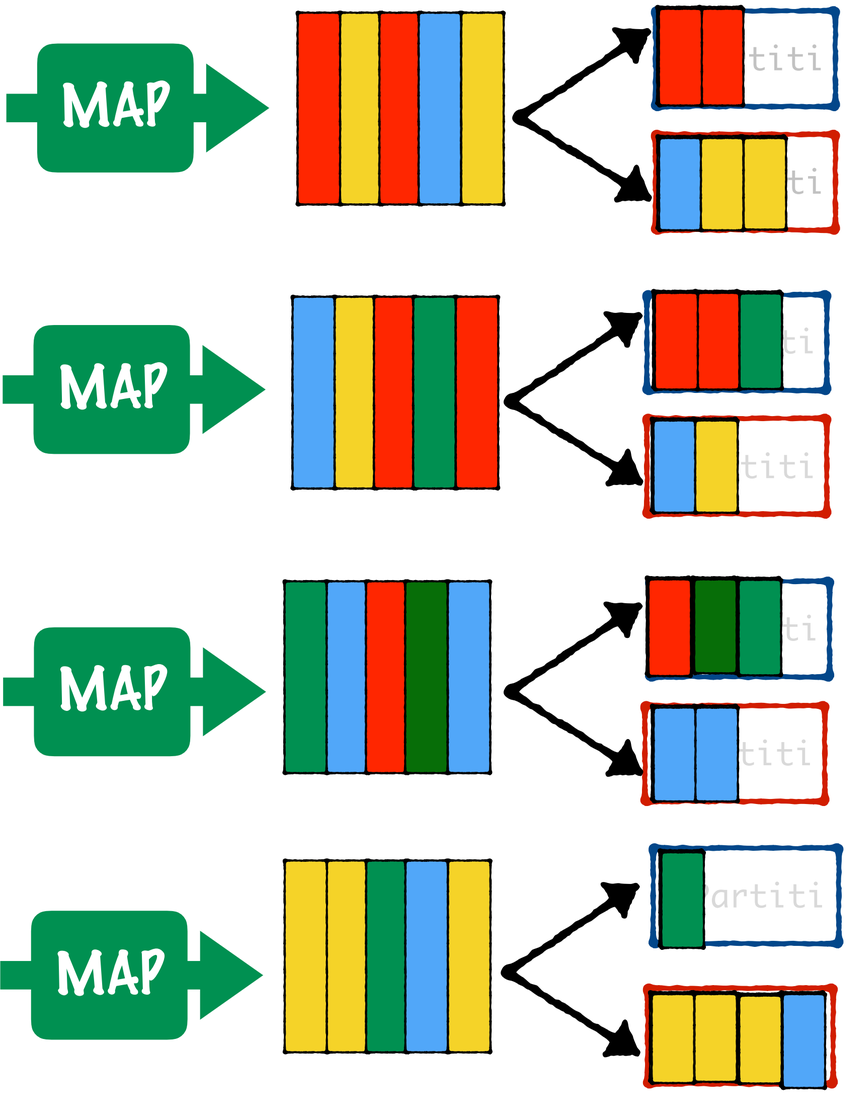


All values with the same key must be assigned to the same partition

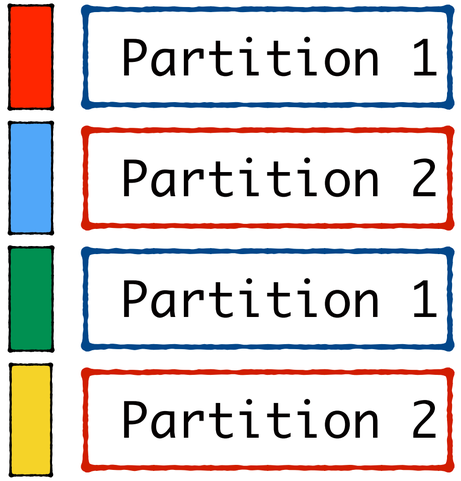


RECAP

Shuffle and Sort



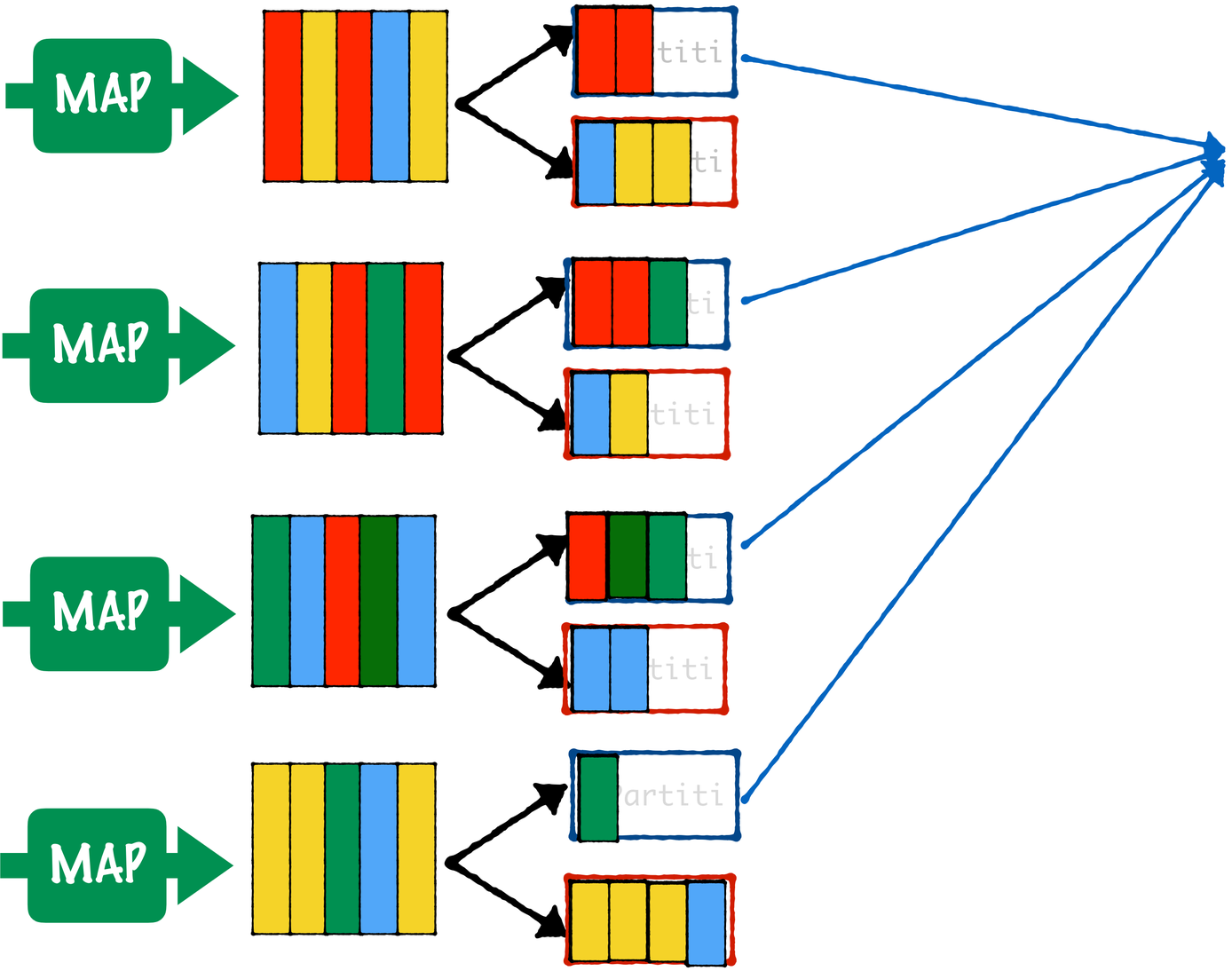
All values with the same key must be assigned to the same partition



RECAP

Shuffle and Sort

Partitioning

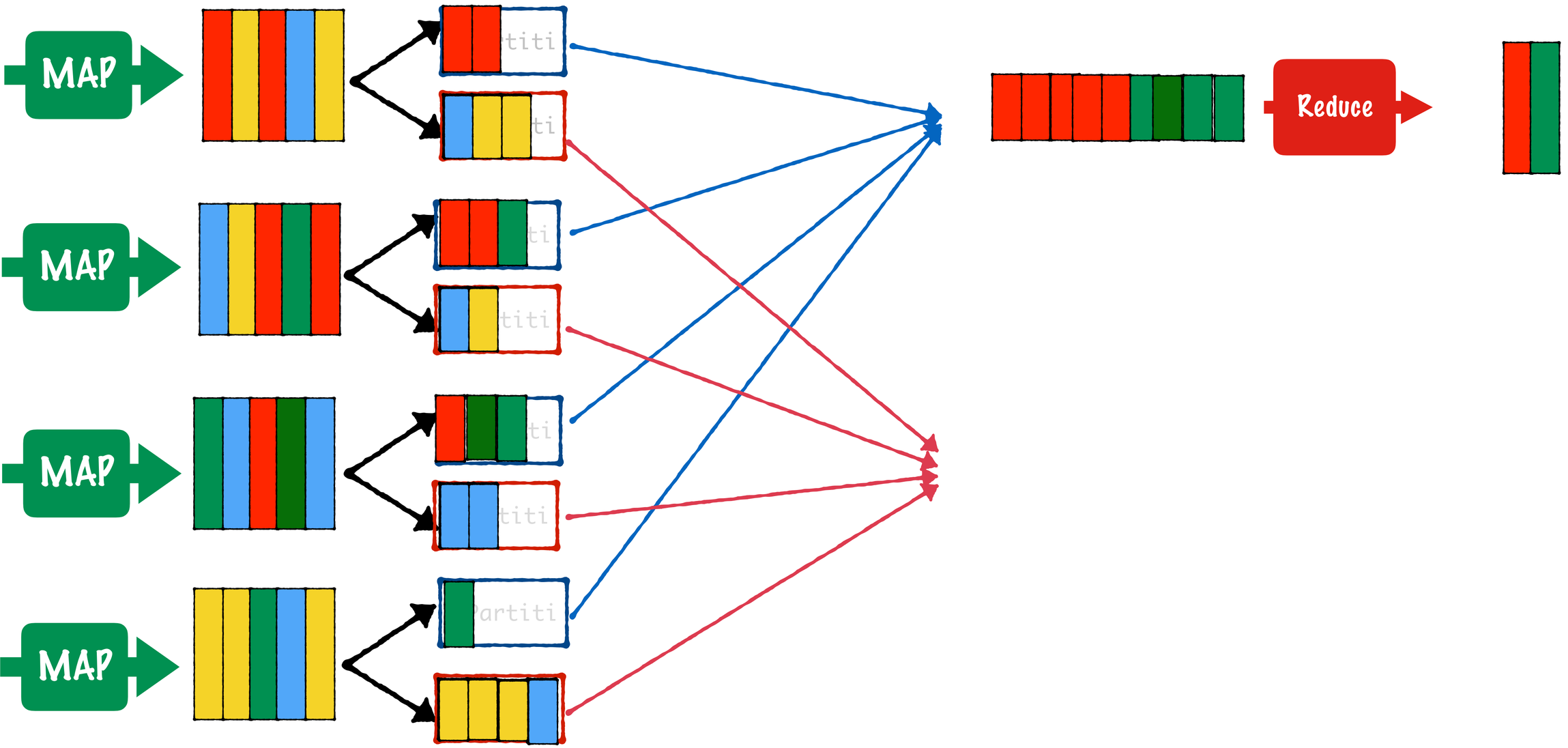


The data in each partition is sent to 1 reducer

RECAP

Shuffle and Sort

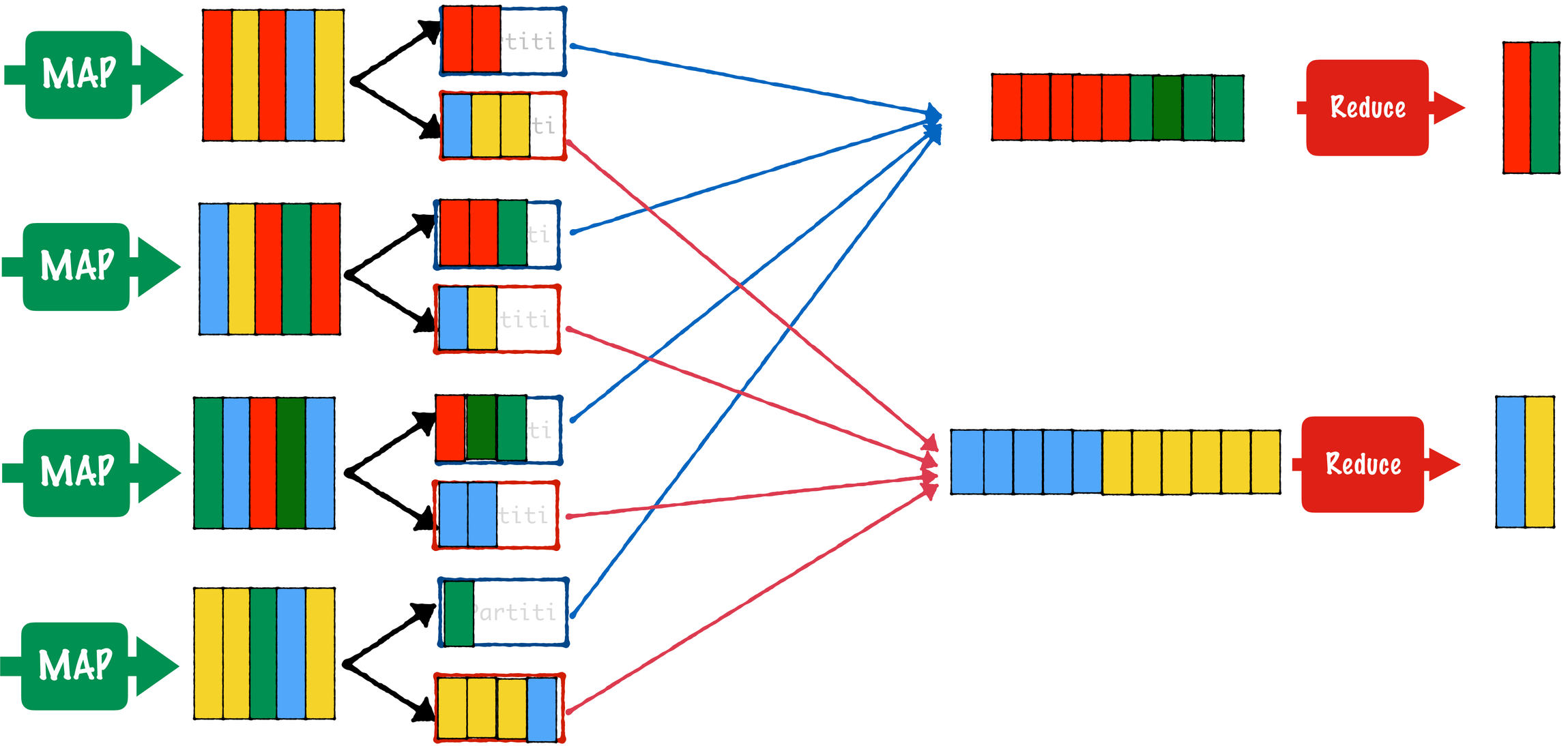
Partitioning

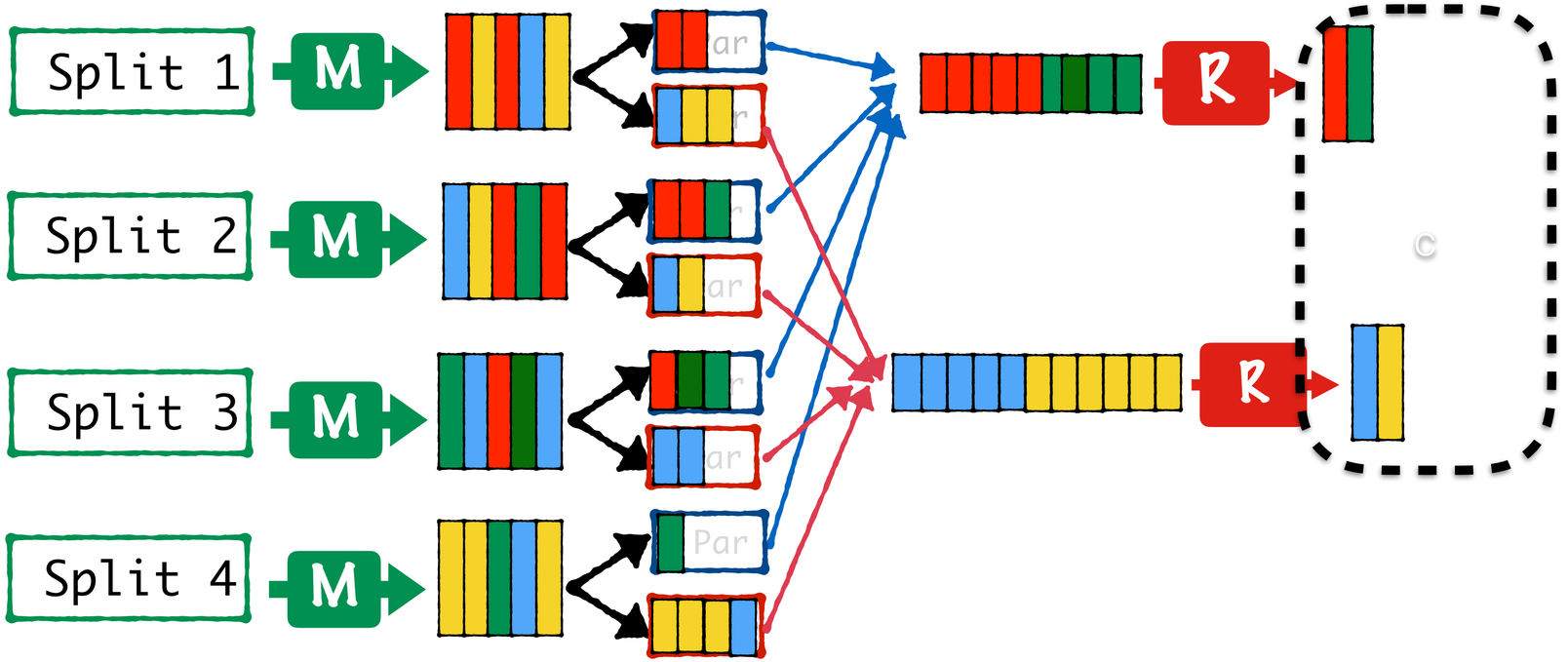


RECAP

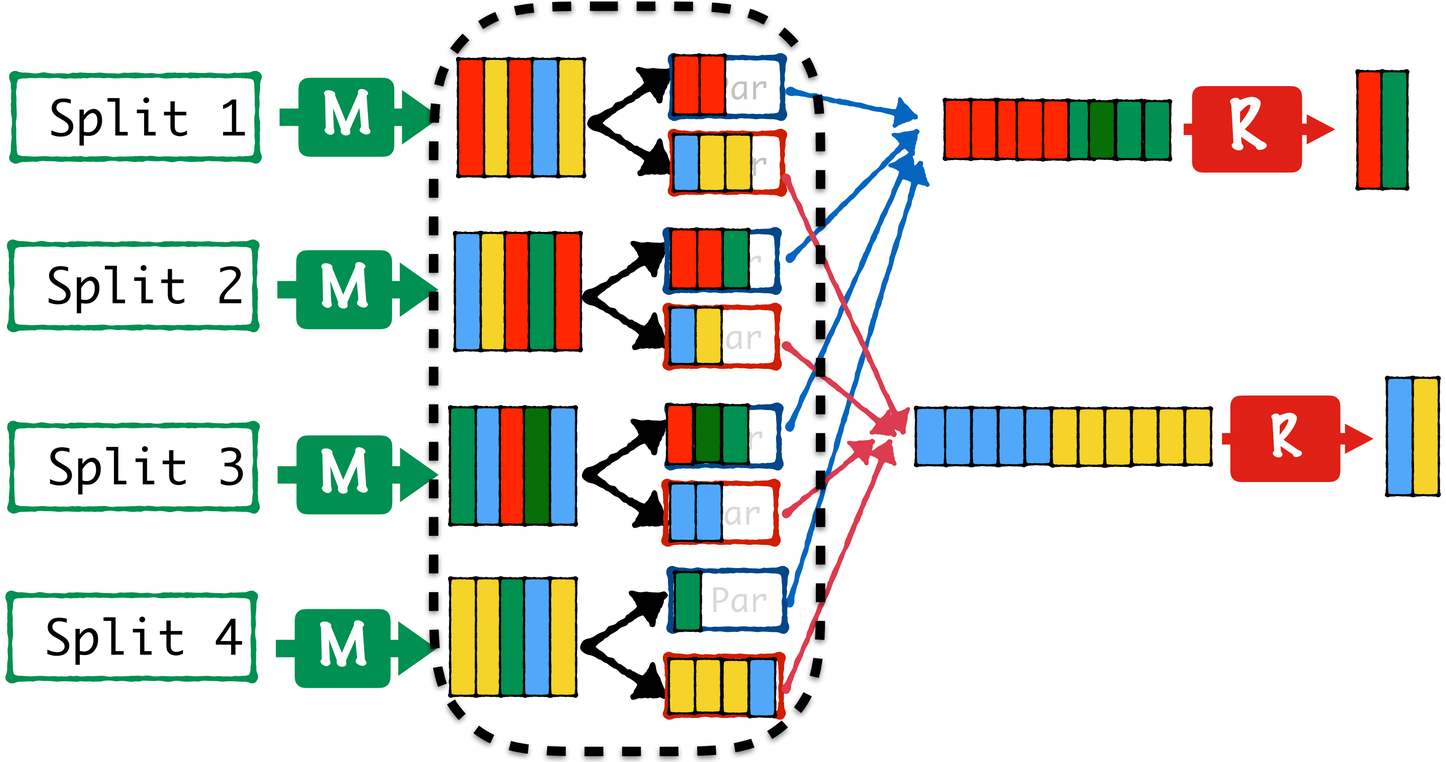
Shuffle and Sort

Partitioning

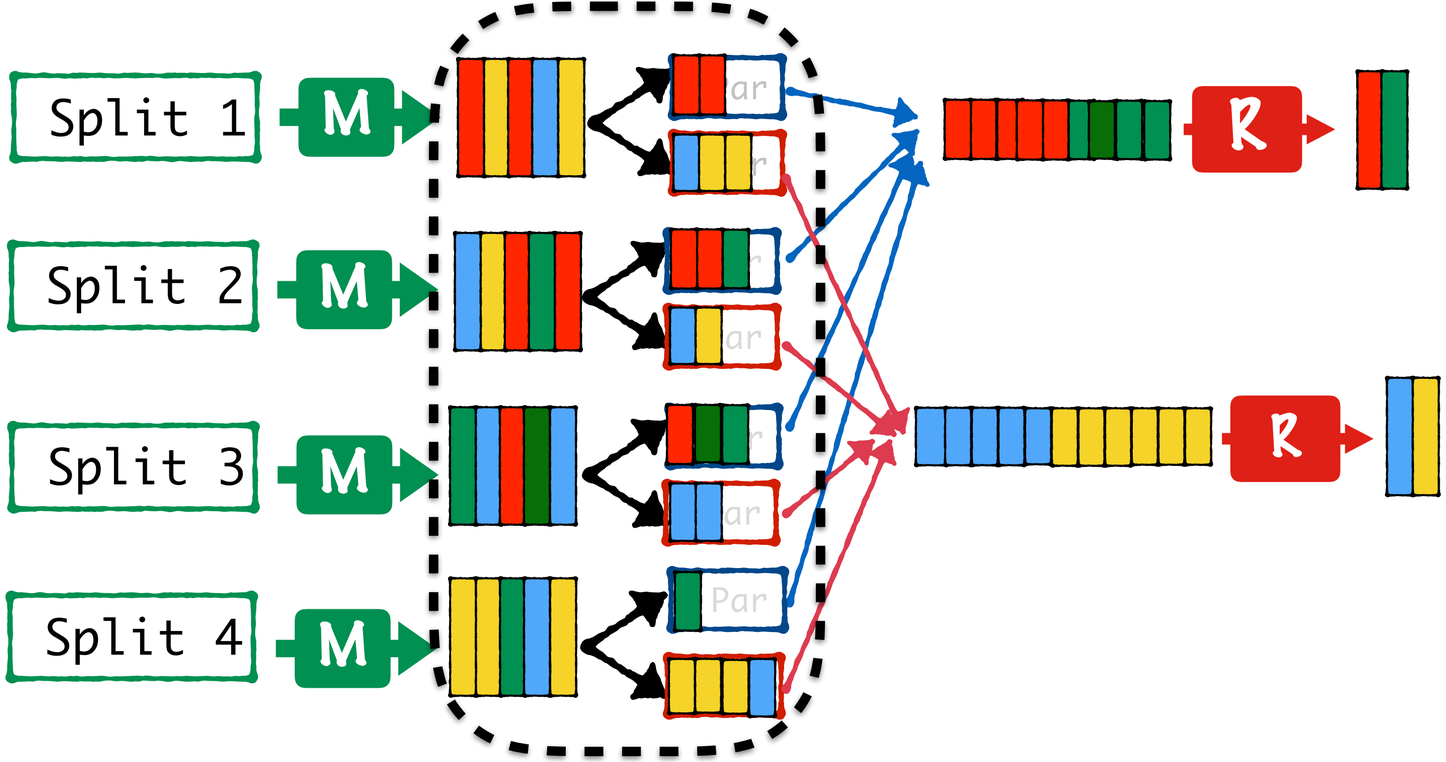




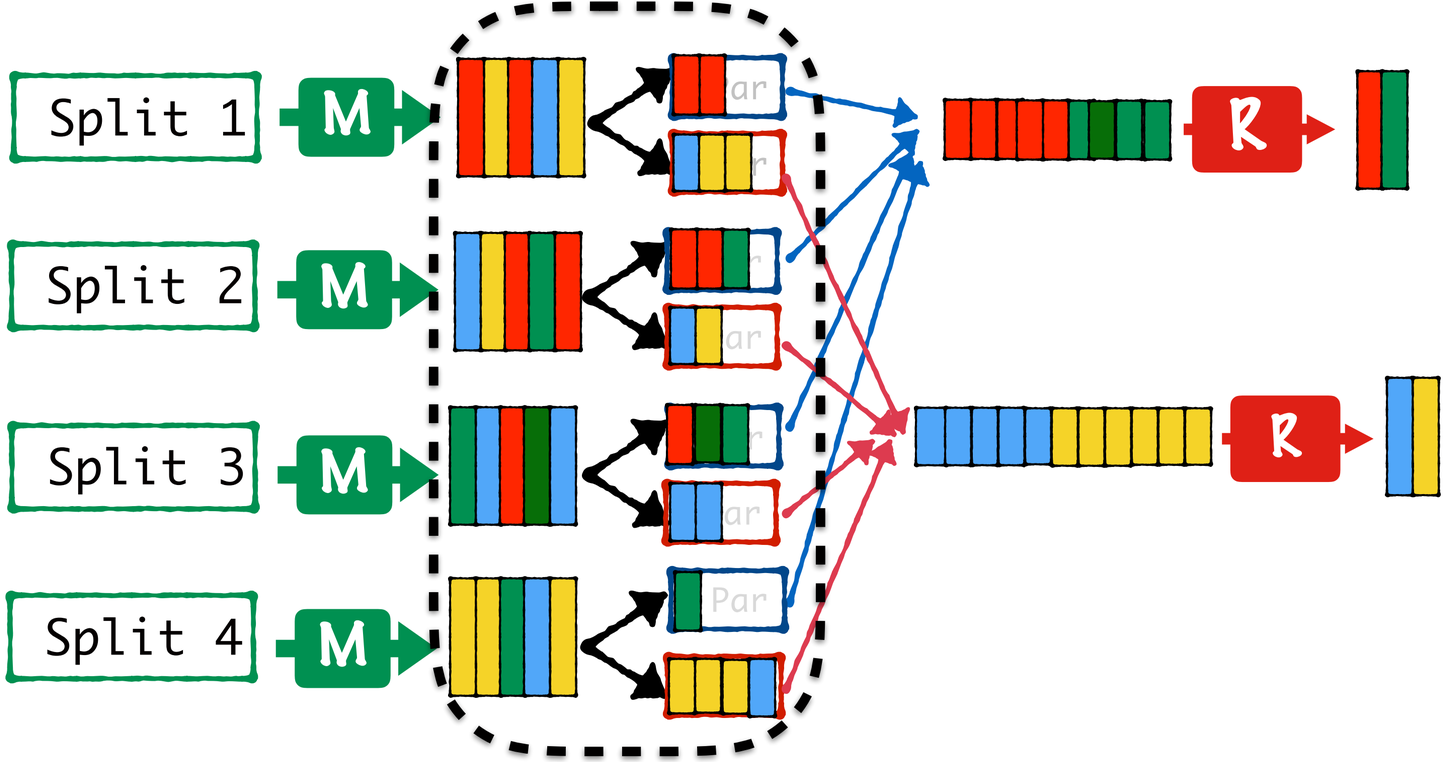
Each reducer will write to a different output file



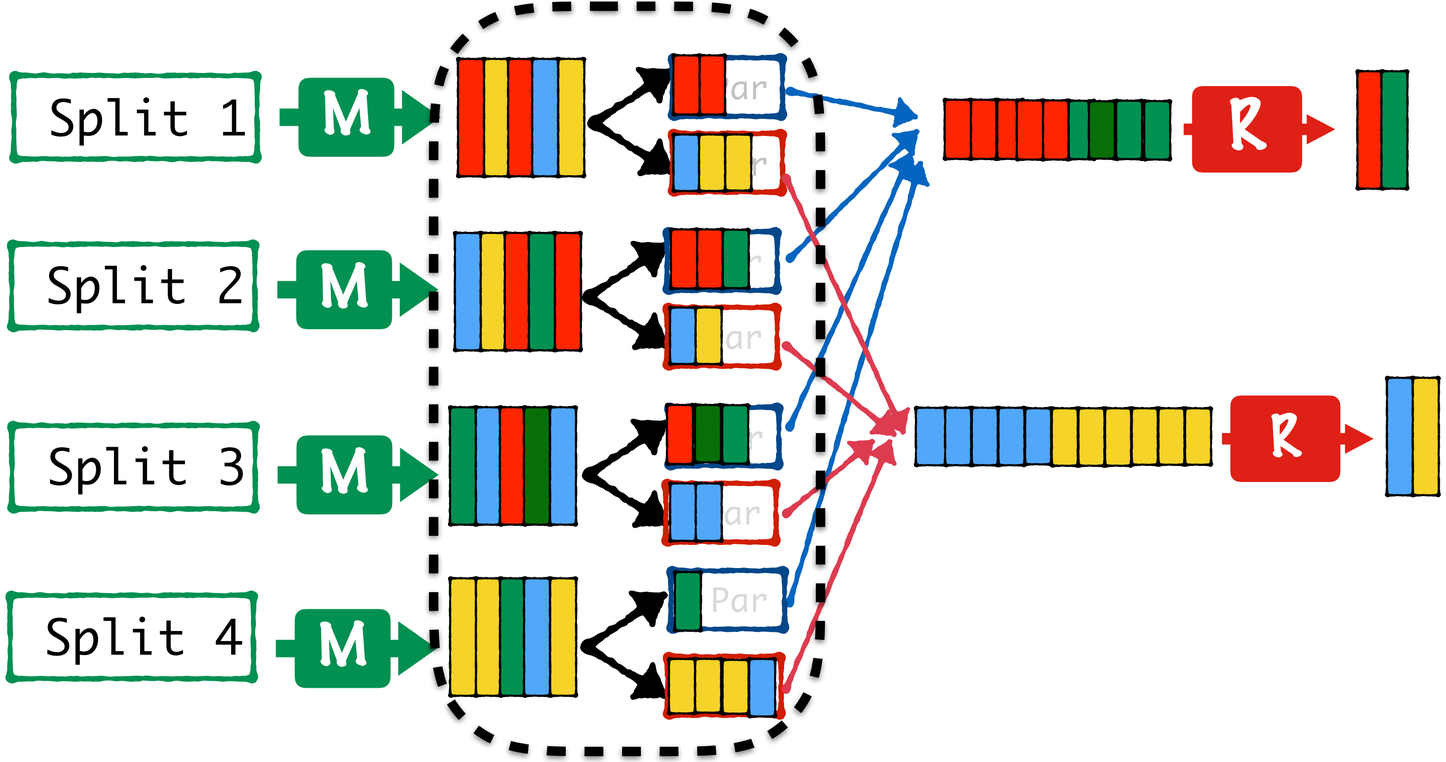
The default Partitioning logic uses the hash value of the key



But you can use your own Partitioner which uses the actual value of the key



In our case the Key = Count



Custom Partitioner Logic

If Count < 10 assign to first Partition else to second Partition

Customized Partitioning

If Count < 10 assign to first Partition else to second Partition

```
public class countPartitioner extends
Partitioner<IntWritable,Text> {
    @Override
    public int getPartition(IntWritable key, Text value, int numReduceTasks) {
        int partition;
        if (key.get()<=10){ partition = 0;}
        else{partition = 1;}
        return partition ;
    }
}
```

You can implement this logic in a Partitioner class

Customized Partitioning

```
public class countPartitioner extends
Partitioner<IntWritable,Text> {
    @Override
    public int getPartition(IntWritable key, Text value, int numReduceTasks) {
        int partition;
        if (key.get()<=10){ partition = 0;}
        else{partition = 1;}
        return partition ;
    }
}
```

Partitioner is a generic whose type parameters should be consistent with the Map Output

Customized Partitioning

```
public class countPartitioner extends
Partitioner<IntWritable,Text> {
    @Override
    public int getPartition(IntWritable key, Text value, int numReduceTasks) {
        int partition;
        if (key.get()<=10){ partition = 0;}
        else{partition = 1;}
        return partition ;
    }
}
```

In our case the Map will just take the <Word, Count> from each line and return <Count, Word>

Customized Partitioning

```
public class countPartitioner extends Partitioner<IntWritable,Text> {
    @Override
      public int getPartition(IntWritable key, Text value, int
numReduceTasks) {
          int partition;
          if (key.get()<=10){ partition = 0;}
          else{partition = 1;}
          return partition ;
    }
}
```

We have to implement the getPartition() method of the Partitioner

Customized Partitioning

```
public class countPartitioner extends Partitioner<IntWritable,Text> {
   @Override
public int getPartition(IntWritable key, Text value,
int numReduceTasks) {
        int partition;
        if (key.get()<=10){ partition = 0;}
        else{partition = 1;}
        return partition;
   }
}
```

This method needs to return an integer which specifies the partition number

Customized Partitioning

```
public class countPartitioner extends Partitioner<IntWritable,Text> {
   @Override
public int getPartition(IntWritable key, Text value,
int numReduceTasks) {
        int partition;
        if (key.get()<=10){ partition = 0;}
        else{partition = 1;}
        return partition ;
   }
}
```

The partition numbers should start from 0

Customized Partitioning

```
public class countPartitioner extends Partitioner<IntWritable,Text> {
    @Override
public int getPartition(IntWritable key, Text value,
int numReduceTasks) {
        int partition;
        if (key.get()<=10){ partition = 0;}
        else{partition = 1;}
        return partition ;
    }
}
```

This method will normally use the number of specified reducers

Customized Partitioning

```
public class countPartitioner extends Partitioner<IntWritable,Text> {
    @Override
public int getPartition(IntWritable key, Text value,
int numReduceTasks) {
        int partition;
        if (key.get()<=10){ partition = 0;}
        else{partition = 1;}
        return partition ;
    }
}
```

In our implementation, we are ignoring the number of reducers

Customized Partitioning

```
public class countPartitioner extends Partitioner<IntWritable,Text> {
   @Override
public int getPartition(IntWritable key, Text value,
int numReduceTasks) {
        int partition;
        if (key.get()<=10){ partition = 0;}
        else{partition = 1;}
        return partition ;
   }
}
```

We have hardcoded the total number of partitions to be 2

Customized Partitioning

```
public class countPartitioner extends Partitioner<IntWritable,Text> {
    @Override
public int getPartition(IntWritable key, Text value,
int numReduceTasks) {
        int partition;
        if (key.get()<=10){ partition = 0;}
        else{partition = 1;}
        return partition ;
    }
}
```

By doing this, we are forced to set the number of reducers to always be 2

Customized Partitioning

```
public class countPartitioner extends Partitioner<IntWritable,Text> {
    @Override
public int getPartition(IntWritable key, Text value,
int numReduceTasks) {
        int partition;
        if (key.get()<=10){ partition = 0;}
        else{partition = 1;}
        return partition ;
    }
}
```

Even if we use more than 2 reducers, except for the first 2, all others will not get any input to process

Customized Partitioning

```
public class countPartitioner extends Partitioner<IntWritable,Text> {
   @Override
public int getPartition(IntWritable key, Text value,
int numReduceTasks) {
        int partition;
        if (key.get()<=10){ partition = 0;}
        else{partition = 1;}
        return partition ;
   }
}
```

This is not very good practice, but it works for the Objective we need to achieve

Customized Partitioning

```
public class countPartitioner extends Partitioner<IntWritable,Text> {
   @Override
public int getPartition(IntWritable key, Text value,
int numReduceTasks) {
        int partition;
        if (key.get()<=10){ partition = 0;}
        else{partition = 1;}
        return partition ;
   }
}
```

Let's quickly see the Map, Reduce and Main classes

Customized Partitioning

Map Class

```
public class Map extends Mapper<LongWritable,Text,IntWritable,Text> {

    @Override
    public void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException
    {
        String line = value.toString();
        String[] data = line.split("\t");

        IntWritable keyOut = new IntWritable(Integer.parseInt(data[1]));
        Text valueOut = new Text(data[0]);

        context.write(keyOut, valueOut);
    }
}
```

Customized Partitioning

Map Class

```
public class Map extends Mapper<LongWritable,Text,IntWritable,Text> {

    @Override
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException
    {
        String line = value.toString();
        String[] data = line.split("\t");


        IntWritable keyOut = new IntWritable(Integer.parseInt(data[1]));
        Text valueOut = new Text(data[0]);

        context.write(keyOut, valueOut);
    }
}
```

Here we parse the line containing Word \t Count

Customized Partitioning

Map Class

```
public class Map extends Mapper<LongWritable,Text,IntWritable,Text> {

    @Override
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException
    {
        String line = value.toString();
        String[] data = line.split("\t");


        IntWritable keyOut = new IntWritable(Integer.parseInt(data[1]));
        Text valueOut = new Text(data[0]);

        context.write(keyOut, valueOut);
    }
}
```

We write out Count, Word

Customized Partitioning

Reduce Class

```
public class Reduce extends Reducer<IntWritable,Text,IntWritable,Text> {
    @Override
    public void reduce(final IntWritable key, final Iterable<Text> values,
                       final Context context) throws IOException, InterruptedException {

        for (Text value : values) {
            context.write(key, value);
        }

    }
}
```

Our Reducer is simply an identity function

Customized Partitioning

Reduce Class

```
public class Reduce extends Reducer<IntWritable,Text,IntWritable,Text> {
    @Override
    public void reduce(final IntWritable key, final Iterable<Text> values,
                       final Context context) throws IOException, InterruptedException {

        for (Text value : values) {
            context.write(key, value);
        }

    }
}
```

We only want the data partitioned, we don't want to change it in any way!

Customized Partitioning

Main Class

```
public class countPartition extends Configured implements Tool{
    @Override
    public int run(String[] args) throws Exception{

        if(args.length !=2){
            System.err.println("Invalid Command");
            System.err.println("Usage: <input path> <output path>");
            return -1;
        }

        Job job = Job.getInstance(getConf());
        job.setJobName("countPartition");
        job.setJarByClass(countPartition.class);

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

        job.setMapperClass(Map.class);
        job.setReducerClass(Reduce.class);
        job.setPartitionerClass(countPartitioner.class);
        job.setNumReduceTasks(2);

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

        return job.waitForCompletion(true) ? 0 : 1;
    }

    public static void main(String[] args) throws Exception {
        int exitCode = ToolRunner.run(new countPartition(), args);
        System.exit(exitCode);
    }
}
```

In our Main Class we need to set the Partitioner Class

Customized Partitioning

Main Class

```
public class countPartition extends Configured implements Tool{
    @Override
    public int run(String[] args) throws Exception{

        if(args.length !=2){
            System.err.println("Invalid Command");
            System.err.println("Usage: <input path> <output path>");
            return -1;
        }

        Job job = Job.getInstance(getConf());
        job.setJobName("countPartition");
        job.setJarByClass(countPartition.class);

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

        job.setMapperClass(Map.class);
        job.setReducerClass(Reduce.class);

        job.setPartitionerClass(countPartitioner.class);

        job.setNumReduceTasks(2);

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

        return job.waitForCompletion(true) ? 0 : 1;
    }

    public static void main(String[] args) throws Exception {
        int exitCode = ToolRunner.run(new countPartition(), args);
        System.exit(exitCode);
    }
}
```

Since we need the number of reduce tasks to be fixed to 2, we can do that here too

Customized Partitioning

Main Class

```
public class countPartition extends Configured implements Tool{
    @Override
    public int run(String[] args) throws Exception{

        if(args.length !=2){
            System.err.println("Invalid Command");
            System.err.println("Usage: <input path> <output path>");
            return -1;
        }

        Job job = Job.getInstance(getConf());
        job.setJobName("countPartition");
        job.setJarByClass(countPartition.class);

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

        job.setMapperClass(Map.class);
        job.setReducerClass(Reduce.class);

        job.setPartitionerClass(countPartitioner.class);
        job.setNumReduceTasks(2);

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

        return job.waitForCompletion(true) ? 0 : 1;
    }

    public static void main(String[] args) throws Exception {
        int exitCode = ToolRunner.run(new countPartition(), args);
        System.exit(exitCode);
    }
}
```

Since we need the number of reduce tasks to be fixed to 2, we can do that here too

Customized Partitioning

Main Class

```
public class countPartition extends Configured implements Tool{
    @Override
    public int run(String[] args) throws Exception{

        if(args.length !=2){
            System.err.println("Invalid Command");
            System.err.println("Usage: <input path> <output path>");
            return -1;
        }

        Job job = Job.getInstance(getConf());
        job.setJobName("countPartition");
        job.setJarByClass(countPartition.class);

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

        job.setMapperClass(Map.class);
        job.setReducerClass(Reduce.class);

        job.setPartitionerClass(countPartitioner.class);
        job.setNumReduceTasks(2);

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

        return job.waitForCompletion(true) ? 0 : 1;
    }

    public static void main(String[] args) throws Exception {
        int exitCode = ToolRunner.run(new countPartition(), args);
        System.exit(exitCode);
    }
}
```

This is another way of setting the number of reduce tasks (normally you can do so at the command line)

Customized Partitioning

When you run this code, in the output directory you will see 2 files

```
/test/partition5/part-r-00000
/test/partition5/part-r-00001
```

Each file corresponds to the output from 1 reducer

Customized Partitioning

/test/partition5/part-r-00000
/test/partition5/part-r-00001

The first reducer output only has counts till 10

6	map
	each
	This
7	not
7	larg
8	mach
8	on
8	an
8	one
8	or
8	redu
8	be
8	this
8	cons
9	In
9	Figu
10	into
10	can

Customized Partitioning

```
/test/partition5/part-r-00000
/test/partition5/part-r-00001
```

The second reducer output only has counts > 10

```
11      segment
12      we
12      indexing
12      MapReduce
14      index
14      term
14      The
14      distributed
16      are
17      that
21      for
24      is
25      and
33      to
35      in
38      a
45      of
61      the
```

Total Sort

Total Sort

Say you have a file with a list of names

Janani

Vitthal

Swetha

Navdeep

Shreya

Pradeep

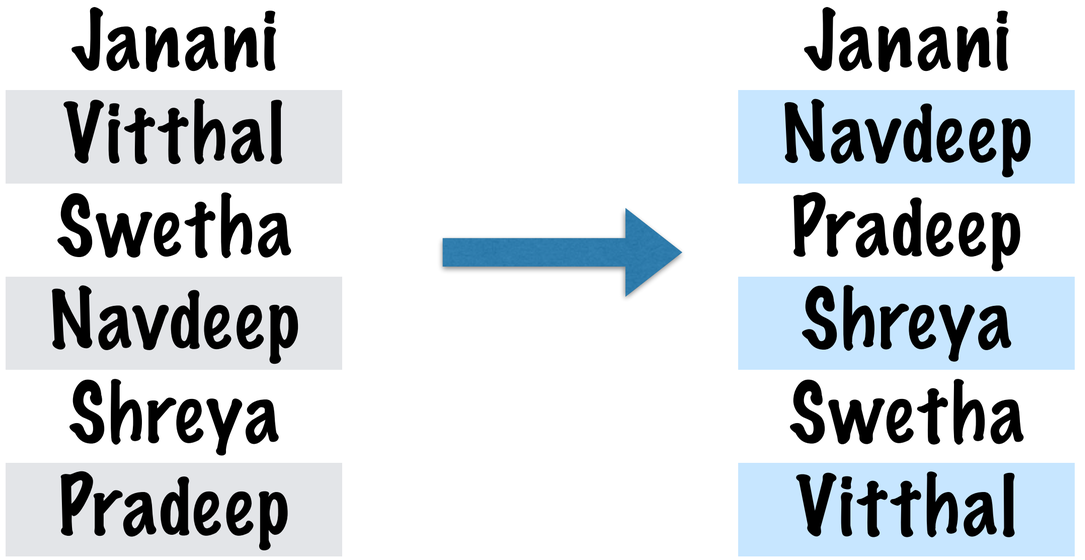
Total Sort

Objective: Sort this list of names

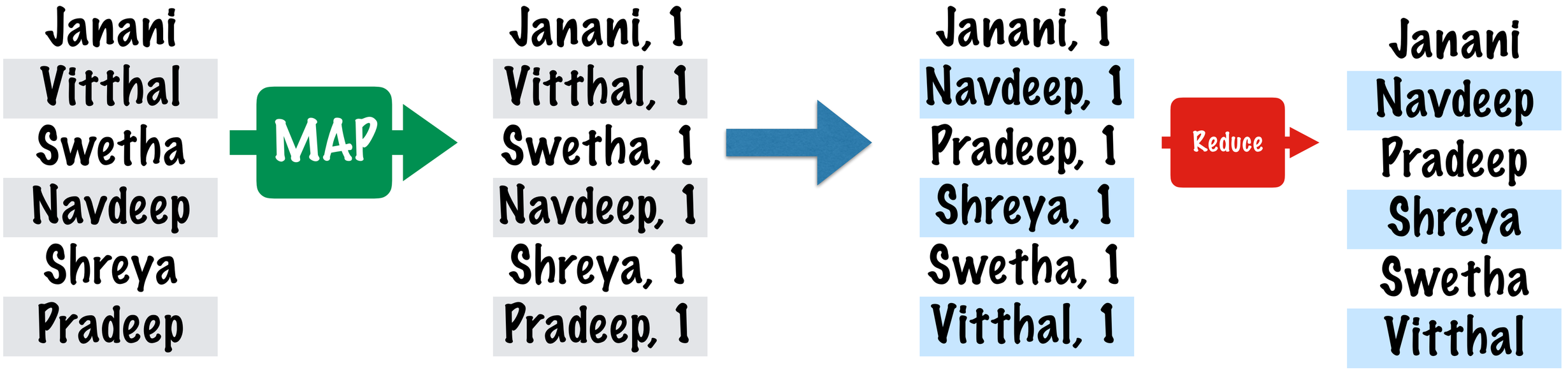
Janani	Janani
Vitthal	Navdeep
Swetha	Pradeep
Navdeep	Shreya
Shreya	Swetha
Pradeep	Vitthal

Total Sort

This would be pretty straight forward using MapReduce

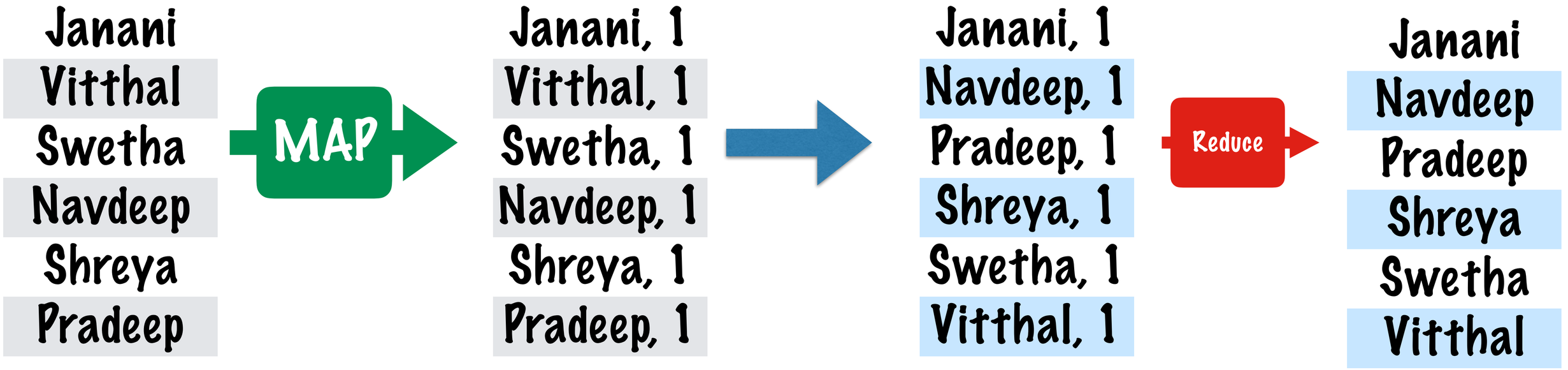


Total Sort



Hadoop will perform a Sort/Merge

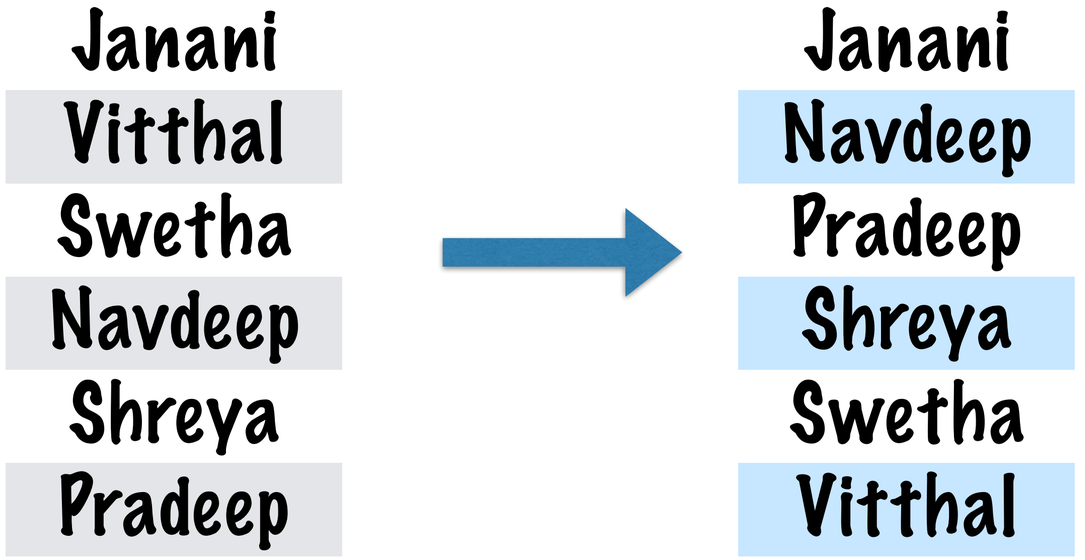
Total Sort



In the Reduce, just write out Key, Null

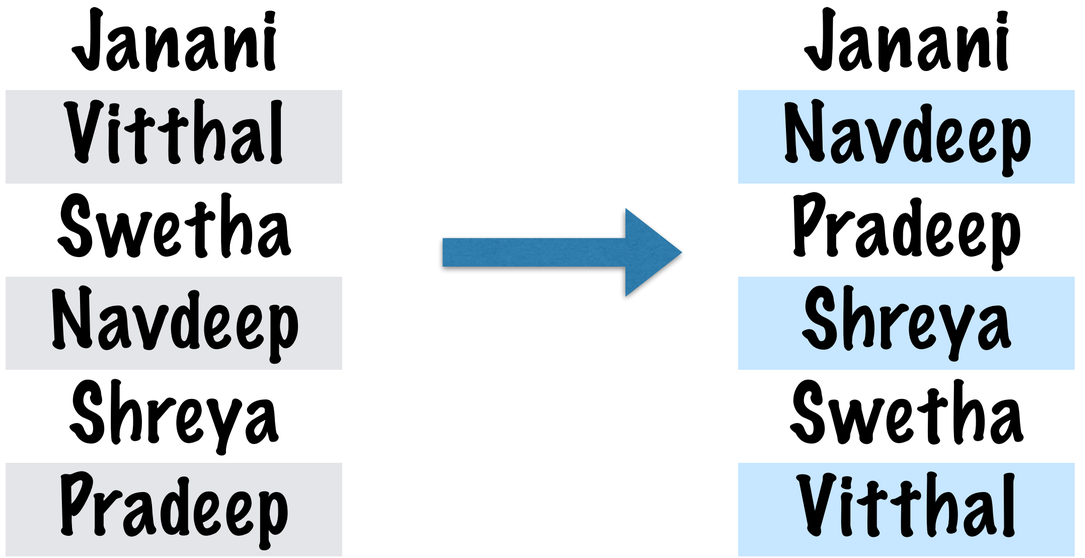
Total Sort

The output will be the same as the input, except that it is sorted



Total Sort

In order to speed up this job, you might want to use multiple reducers



Total Sort

The output is sorted within each Reducer but not at a global level

Janani
Swetha
Shreya

Vitthal
Navdeep
Pradeep

Total Sort

In order to solve this problem Hadoop provides a TotalOrderPartitioner class

Janani
Swetha
Shreya

Vitthal
Navdeep
Pradeep

Total Sort

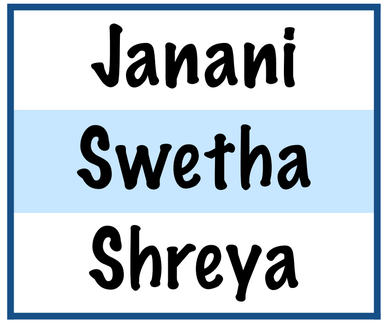
The TotalOrderPartitionar partitions the keys in such a fashion that the keys are globally sorted across all reducers

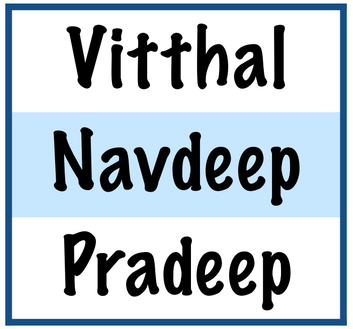
Janani
Swetha
Shreya

Vitthal
Navdeep
Pradeep

Total Sort

This is a huge - you can have your keys distributed to partitions in such a way that even though the reducer sorts only a portion of the data - the entire dataset is sorted!





Total Sort

Janani
Swetha
Shreya

TotalOrderPartitionar is kind of like magic

Vitthal
Navdeep
Pradeep

Total Sort

If you have 99 keys and 3 partitions:

The first 33 in the final sort order will go to the first partition

The second 33 in the final sort order will go to the second partition

The third 33 in the final sort order will go to the third partition

Janani
Swetha
Shreya

Vitthal
Navdeep
Pradeep

TotalOrderPartitioner

The idea is that by sampling the records in the input, we can get a good idea of where the keys are

TotalOrderPartitioner

sampling the records in the input

Once that's known, create a file which maps key -> Partition to evenly distribute the keys

TotalOrderPartitioner

sampling the records in the input

key -> Partition mapping

Remember the keys have to be distributed in such a manner that

TotalOrderPartitioner

sampling the records in the input

key -> Partition mapping

This file is then distributed to all the mappers

TotalOrderPartitioner

sampling the records in the input

key -> Partition mapping

distributed to mappers

This file is used by TotalOrderPartitioner to assign partitions

TotalOrderPartitioner

sampling the records in the input

key -> Partition mapping

distributed to mappers

TotalOrderPartitioner

This file is used by TotalOrderPartitioner to assign partitions

TotalOrderPartitioner

sampling the records in the input

key -> Partition mapping

distributed to mappers

TotalOrderPartitioner

TotalOrderPartitioner

InputSampler Class

distributed to mappers

TotalOrderPartitioner

TotalOrderPartitioner

InputSampler Class

Distributed Cache

TotalOrderPartitioner

TotalOrderPartitioner

InputSampler Class

An InputSampler class takes a job and generates a mapping of key -> Partition

TotalOrderPartitioner

InputSampler Class

It does this by sampling the records in the Input data

TotalOrderPartitioner

InputSampler Class

There are 3 types of InputSamplers available

RandomSampler

SplitSampler

IntervalSampler

TotalOrderPartitioner

InputSampler Class

RandomSampler	Samples records randomly based on some parameters given by the user
SplitSampler	Samples the first n records in each split
IntervalSampler	Samples records at regular intervals

TotalOrderPartitioner

InputSampler Class

After sampling, InputSampler will write a partition file which maps keys -> Partitions

TotalOrderPartitioner

InputSampler Class

Distributed Cache

TotalOrderPartitioner

TotalOrderPartitioner

InputSampler Class

Distributed Cache

TotalOrderPartitioner

TotalOrderPartitioner

Distributed Cache

DistributedCache is a facility provided by the Map-Reduce framework to cache files (text, archives, jars etc.) needed by applications.

TotalOrderPartitioner

Distributed Cache

When you have some data in a small file that needs to be used by all the mappers/reducers, use the Distributed Cache

TotalOrderPartitioner

Distributed Cache

You can use the cache to copy and distribute any files to all nodes in the cluster

TotalOrderPartitioner

InputSampler Class

Distributed Cache

TotalOrderPartitioner

TotalOrderPartitioner

All of this is set up in the Main Class

InputSampler Class

Distributed Cache

TotalOrderPartitioner

TotalOrderPartitioner

```
job.setPartitionerClass(TotalOrderPartitioner.class);

InputSampler.Sampler<Text,IntWritable> sampler = new
InputSampler.RandomSampler<Text,IntWritable>(0.1,10000,10);
InputSampler.writePartitionFile(job, sampler);

// Add to Distributed Cache
Configuration conf = job.getConfiguration();
String partitionFile = TotalOrderPartitioner.getPartitionFile(conf);
URI partitionUri = new URI(partitionFile);
job.addCacheFile(partitionUri);
```

You can set the Partitioner class to TotalOrderPartitioner

TotalOrderPartitioner

```
job.setPartitionerClass(TotalOrderPartitioner.class);

InputSampler.Sampler<Text,IntWritable> sampler = new
InputSampler.RandomSampler<Text,IntWritable>(0.1,10000,10);
InputSampler.writePartitionFile(job, sampler);

// Add to Distributed Cache
Configuration conf = job.getConfiguration();
String partitionFile = TotalOrderPartitioner.getPartitionFile(conf);
URI partitionUri = new URI(partitionFile);
job.addCacheFile(partitionUri);
```

You can set the Partitioner class to TotalOrderPartitioner

TotalOrderPartitioner

```
job.setPartitionerClass(TotalOrderPartitioner.class);

InputSampler.Sampler<Text,IntWritable> sampler = new
InputSampler.RandomSampler<Text,IntWritable>(0.1,10000,10);
InputSampler.writePartitionFile(job, sampler);

// Add to Distributed Cache
Configuration conf = job.getConfiguration();
String partitionFile = TotalOrderPartitioner.getPartitionFile(conf);
URI partitionUri = new URI(partitionFile);
job.addCacheFile(partitionUri);
```

Here we are using a RandomSampler as the InputSampler

TotalOrderPartitioner

```
job.setPartitionerClass(TotalOrderPartitioner.class);

InputSampler.Sampler<Text,IntWritable> sampler = new
InputSampler.RandomSampler<Text,IntWritable>(0.1,10000,10);

InputSampler.writePartitionFile(job, sampler);

// Add to Distributed Cache
Configuration conf = job.getConfiguration();
String partitionFile = TotalOrderPartitioner.getPartitionFile(conf);
URI partitionUri = new URI(partitionFile);
job.addCacheFile(partitionUri);
```

It's a generic whose type parameters should be consistent with the Map output

TotalOrderPartitioner

```
job.setPartitionerClass(TotalOrderPartitioner.class);

InputSampler.Sampler<Text,IntWritable> sampler = new
InputSampler.RandomSampler<Text,IntWritable>(0.1,10000,10);

InputSampler.writePartitionFile(job, sampler);

// Add to Distributed Cache
Configuration conf = job.getConfiguration();
String partitionFile = TotalOrderPartitioner.getPartitionFile(conf);
URI partitionUri = new URI(partitionFile);
job.addCacheFile(partitionUri);
```

This is the % of records to be sampled in a split (chosen at random)

TotalOrderPartitioner

```
job.setPartitionerClass(TotalOrderPartitioner.class);

InputSampler.Sampler<Text,IntWritable> sampler = new
InputSampler.RandomSampler<Text,IntWritable>(0.1,10000,10);

InputSampler.writePartitionFile(job, sampler);

// Add to Distributed Cache
Configuration conf = job.getConfiguration();
String partitionFile = TotalOrderPartitioner.getPartitionFile(conf);
URI partitionUri = new URI(partitionFile);
job.addCacheFile(partitionUri);
```

The max number of records to sample

TotalOrderPartitioner

```
job.setPartitionerClass(TotalOrderPartitioner.class);

InputSampler.Sampler<Text,IntWritable> sampler = new
InputSampler.RandomSampler<Text,IntWritable>(0.1,10000,10);

InputSampler.writePartitionFile(job, sampler);

// Add to Distributed Cache
Configuration conf = job.getConfiguration();
String partitionFile = TotalOrderPartitioner.getPartitionFile(conf);
URI partitionUri = new URI(partitionFile);
job.addCacheFile(partitionUri);
```

The max number of splits to sample

TotalOrderPartitioner

```
job.setPartitionerClass(TotalOrderPartitioner.class);

InputSampler.Sampler<Text,IntWritable> sampler = new

InputSampler.RandomSampler<Text,IntWritable>(0.1,10000,10);
InputSampler.writePartitionFile(job, sampler);

// Add to Distributed Cache
Configuration conf = job.getConfiguration();
String partitionFile = TotalOrderPartitioner.getPartitionFile(conf);
URI partitionUri = new URI(partitionFile);
job.addCacheFile(partitionUri);
```

The sampler will then construct key-> partition mapping and write it to a file

TotalOrderPartitioner

```
job.setPartitionerClass(TotalOrderPartitioner.class);

InputSampler.Sampler<Text,IntWritable> sampler = new InputSampler.RandomSampler<Text,IntWritable>(0.1,100
InputSampler.writePartitionFile(job, sampler);

// Add to Distributed Cache
Configuration conf = job.getConfiguration();
String partitionFile = TotalOrderPartitioner.getPartitionFile(conf);
URI partitionUri = new URI(partitionFile);
job.addCacheFile(partitionUri);
```

The partition file's URI can be accessed using the Job's configuration

TotalOrderPartitioner

```
job.setPartitionerClass(TotalOrderPartitioner.class);

InputSampler.Sampler<Text,IntWritable> sampler = new InputSampler.RandomSampler<Text,IntWritable>(0.1,100
InputSampler.writePartitionFile(job, sampler);

// Add to Distributed Cache
Configuration conf = job.getConfiguration();
String partitionFile = TotalOrderPartitioner.getPartitionFile(conf);
URI partitionUri = new URI(partitionFile);
job.addCacheFile(partitionUri);
```

The file is added to the cache to be copied to all the nodes

Secondary Sort

Secondary Sort

MR job outputs are generally sorted by key

Secondary Sort

For each key, the reducer will see a group of values

Secondary Sort

There is no guarantee that those values are sorted

Secondary Sort

What if you wanted to sort those values?

Secondary Sort

We have a Map Output which looks like this

Key	Value
Delhi	20
Bangalore	10
Delhi	33
Bangalore	34
Delhi	12
Bangalore	16

Key	Value
Bangalore	10
Bangalore	34
Bangalore	16
Delhi	20
Delhi	33
Delhi	12

Normally after Sort/Merge

Secondary Sort

The Values here are not sorted

Key	Value
Bangalore	10
Bangalore	34
Bangalore	16
Delhi	20
Delhi	33
Delhi	12

Secondary Sort

The Objective is to sort the values as well

Key	Value
Bangalore	10
Bangalore	34
Bangalore	16
Delhi	20
Delhi	33
Delhi	12

Secondary Sort

This is the output we want after Sort/Merge

Key	Value
Bangalore	10
Bangalore	16
Bangalore	34
Delhi	12
Delhi	20
Delhi	33

Secondary Sort

There is a standard strategy in Hadoop to sort by value

Key	Value
Bangalore	10
Bangalore	16
Bangalore	34
Delhi	12
Delhi	20
Delhi	33

Secondary Sort

Make the map output

<Key,Value>,Null>

Secondary Sort

<<Key,Value>,Null>

There are 3 operations in Sort/Merge

Partitioning

Sort by Key

Group values by key

Secondary Sort

<<Key,Value>,Null>

Partitioning

`setPartitionerClass()`

Sort by Key

`setSortComparatorClass()`

Group values by key

`setGrouping ComparatorClass()`

Secondary Sort

<<Key,Value>,Null>

Partitioning

`setPartitionerClass()`

Sort by Key

Group values by key

`setSortComparatorClass()`

`setGroupingComparatorClass()`

Partitioner should only consider the first member

Secondary Sort

<Key,Value>,Null>

Partitioning

setPartitionerClass()

Sort by Key

setSortComparatorClass()

Sort using both the members

Group using by key

setGrouping ComparatorClass()

Secondary Sort

<<Key,Value>,Null>

Partitioning

`setPartitionerClass()`

Sort by Key

`setSortComparatorClass()`

Group values by key

`setGrouping ComparatorClass()`

Secondary Sort

<<Key,Value>,Null>

Partitioning

setPartitionerClass()

Group the values only by the first member

Sort by Key

setSortComparatorClass()

Group values by key

setGrouping ComparatorClass()

Secondary Sort

```
setPartitionerClass()

setSortComparatorClass()

setGroupingComparatorClass()
```

Let's write the classes for each of these options

Secondary Sort

setPartitionerClass()

```
public class FirstPartitioner extends Partitioner<TextIntPair,NullWritable>{
    @Override
    public int getPartition(TextIntPair key, NullWritable value, int
numReduceTasks){
     return (key.getFirst().hashCode()*Integer.MAX_VALUE) % numReduceTasks;
    }
}
```

FirstPartitioner extends the Partitioner class

Secondary Sort

setPartitionerClass()

```
public class FirstPartitioner extends
Partitioner<TextIntPair,NullWritable>{
    @Override
    public int getPartition(TextIntPair key, NullWritable value, int numReduceTasks){
     return (key.getFirst().hashCode()*Integer.MAX_VALUE) % numReduceTasks;
    }
}
```

Key	Value
Bangalore	10

The type parameters are consistent with our <<Key,Value>, Null>

Secondary Sort

setPartitionerClass()

```
public class FirstPartitioner extends
Partitioner<TextIntPair,NullWritable>{
    @Override
    public int getPartition(TextIntPair key, NullWritable value, int numReduceTasks){
     return (key.getFirst().hashCode()*Integer.MAX_VALUE) % numReduceTasks;
    }
}
```

Key	Value
Bangalore	10

TextIntPair is a custom Writable with 2 members, the first is Text and the second is IntWritable

Secondary Sort

setPartitionerClass()

```
public class FirstPartitioner extends Partitioner<TextIntPair,NullWritable>{
    @Override
     public int getPartition(TextIntPair key, NullWritable value, int
numReduceTasks){
      return (key.getFirst().hashCode()*Integer.MAX_VALUE) % numReduceTasks;
     }
}
```

The getPartition() method uses only the first member to generate a partition id

Secondary Sort

setPartitionerClass()

setSortComparatorClass()

setGroupingComparatorClass()

Secondary Sort

```
setPartitionerClass()

setSortComparatorClass()

setGroupingComparatorClass()
```

Secondary Sort

setSortComparatorClass()

```
public class KeyComparator extends WritableComparator {
    protected  KeyComparator(){
        super(TextIntPair.class, true);
    }

    @Override
    public int compare(WritableComparable w1, WritableComparable w2){

      TextIntPair p1 = (TextIntPair) w1;
        TextIntPair p2 = (TextIntPair) w2;

        int cmp = p1.getFirst().compareTo(p1.getFirst());
        if (cmp!=0){
            return cmp;
        }
       return p1.getSecond().compareTo(p2.getSecond());
    }

}
```

This class needs to extend WritableComparator

Secondary Sort

setSortComparatorClass()

```
public class KeyComparator extends WritableComparator {
    protected  KeyComparator(){
        super(TextIntPair.class, true);
    }

    @Override
    public int compare(WritableComparable w1,
WritableComparable w2){

      TextIntPair p1 = (TextIntPair) w1;
        TextIntPair p2 = (TextIntPair) w2;

        int cmp = p1.getFirst().compareTo(p1.getFirst());
        if (cmp!=0){
            return cmp;
        }
       return p1.getSecond().compareTo(p2.getSecond());
    }

}
```

This class has a compare method we need to implement

Secondary Sort

setSortComparatorClass()

```
public class KeyComparator extends WritableComparator {
    protected  KeyComparator(){
        super(TextIntPair.class, true);
    }

    @Override
    public int compare(WritableComparable w1, WritableComparable w2){

        TextIntPair p1 = (TextIntPair) w1;
        TextIntPair p2 = (TextIntPair) w2;

        int cmp = p1.getFirst().compareTo(p1.getFirst());
        if (cmp!=0){
            return cmp;
        }
       return p1.getSecond().compareTo(p2.getSecond());
    }

}
```

Cast the arguments to TextIntPair

Secondary Sort

setSortComparatorClass()

```
public class KeyComparator extends WritableComparator {
    protected  KeyComparator(){
        super(TextIntPair.class, true);
    }

    @Override
    public int compare(WritableComparable w1, WritableComparable w2){
  TextIntPair p1 = (TextIntPair) w1;

  TextIntPair p2 = (TextIntPair) w2;

        int cmp = p1.getFirst().compareTo(p1.getFirst());
            if (cmp!=0){
                return cmp;
            }
        return p1.getSecond().compareTo(p2.getSecond());
    }

}
```

Compare the first members

Secondary Sort

setSortComparatorClass()

```
public class KeyComparator extends WritableComparator {
    protected  KeyComparator(){
        super(TextIntPair.class, true);
    }

    @Override
    public int compare(WritableComparable w1, WritableComparable w2){
  TextIntPair p1 = (TextIntPair) w1;

  TextIntPair p2 = (TextIntPair) w2;


          int cmp = p1.getFirst().compareTo(p1.getFirst());
       if (cmp!=0){
           return cmp;

       }
          return p1.getSecond().compareTo(p2.getSecond());
     }

}
```

If they are the same compare the second

Secondary Sort

```
setPartitionerClass()

setSortComparatorClass()

setGroupingComparatorClass()
```

Secondary Sort

```
setPartitionerClass()

setSortComparatorClass()

setGroupingComparatorClass()
```

Secondary Sort

setGroupingComparatorClass()

```
public class GroupComparator extends WritableComparator {
    protected  GroupComparator(){
        super(TextIntPair.class, true);
    }

    @Override
    public int compare(WritableComparable w1, WritableComparable w2){

        TextIntPair p1 = (TextIntPair) w1;
        TextIntPair p2 = (TextIntPair) w2;

      return p1.getFirst().compareTo(p1.getFirst());
    }

}
```

This class also needs to extend WritableComparator

Secondary Sort

setGroupingComparatorClass()

```
public class GroupComparator extends WritableComparator {
    protected  GroupComparator(){
        super(TextIntPair.class, true);
    }

    @Override
    public int compare(WritableComparable w1,
WritableComparable w2){

        TextIntPair p1 = (TextIntPair) w1;
        TextIntPair p2 = (TextIntPair) w2;

      return p1.getFirst().compareTo(p1.getFirst());
    }

}
```

Once again we'll implement the compare method

Secondary Sort

setGroupingComparatorClass()

```
public class GroupComparator extends WritableComparator {
    protected  GroupComparator(){
        super(TextIntPair.class, true);
    }

    @Override

    public int compare(WritableComparable w1, WritableComparable w2){

            TextIntPair p1 = (TextIntPair) w1;
            TextIntPair p2 = (TextIntPair) w2;

        return p1.getFirst().compareTo(p1.getFirst());
     }

}
```

But this time we only compare the First members

Secondary Sort

```
setPartitionerClass()

setSortComparatorClass()

setGroupingComparatorClass()
```

Secondary Sort

```
setPartitionerClass()
setSortComparatorClass()
setGroupingComparatorClass()
```

Set these options in the Main class

Secondary Sort

```
setPartitionerClass(FirstPartitioner.class)

setSortComparatorClass(KeyComparator.class)

setGroupingComparatorClass(GroupComparator.class)
```

Set these options in the Main class

Secondary Sort

`setPartitionerClass(FirstPartitioner.class)`

We partition only by the key

`setSortComparatorClass(KeyComparator.class)`

We sort by key and then value

`setGroupingComparatorClass(GroupComparator.class)`

We group/merge only by key