Combiner Function

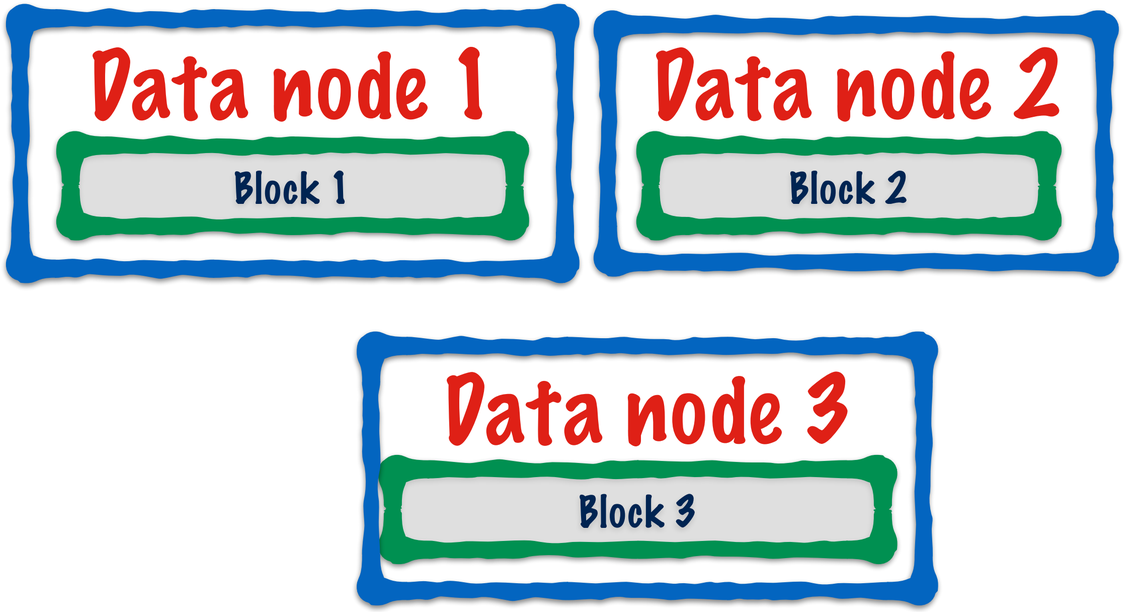
Let's go back to our Word Count example

Objective: Create a **Frequency Distribution** of words in a text file

Word	Count
because	1
each	4
figure	9
..	..

RECAP

The text file has been divided into blocks and stored in HDFS

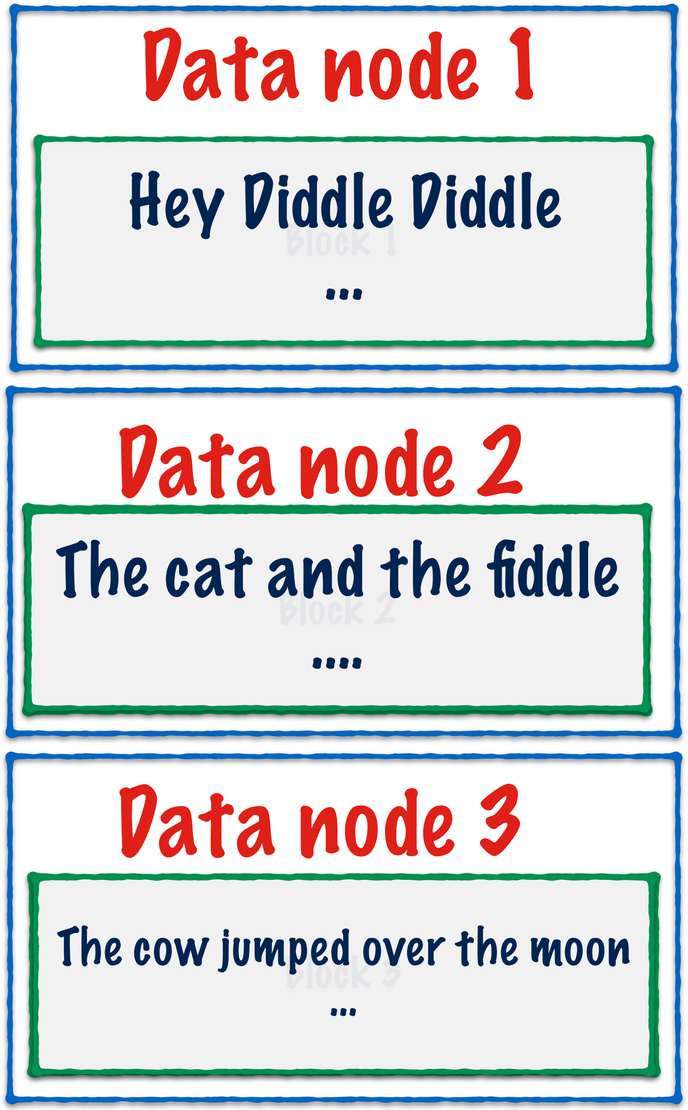


Name node

The name node stores metadata

Here is how the data flows in a MapReduce job

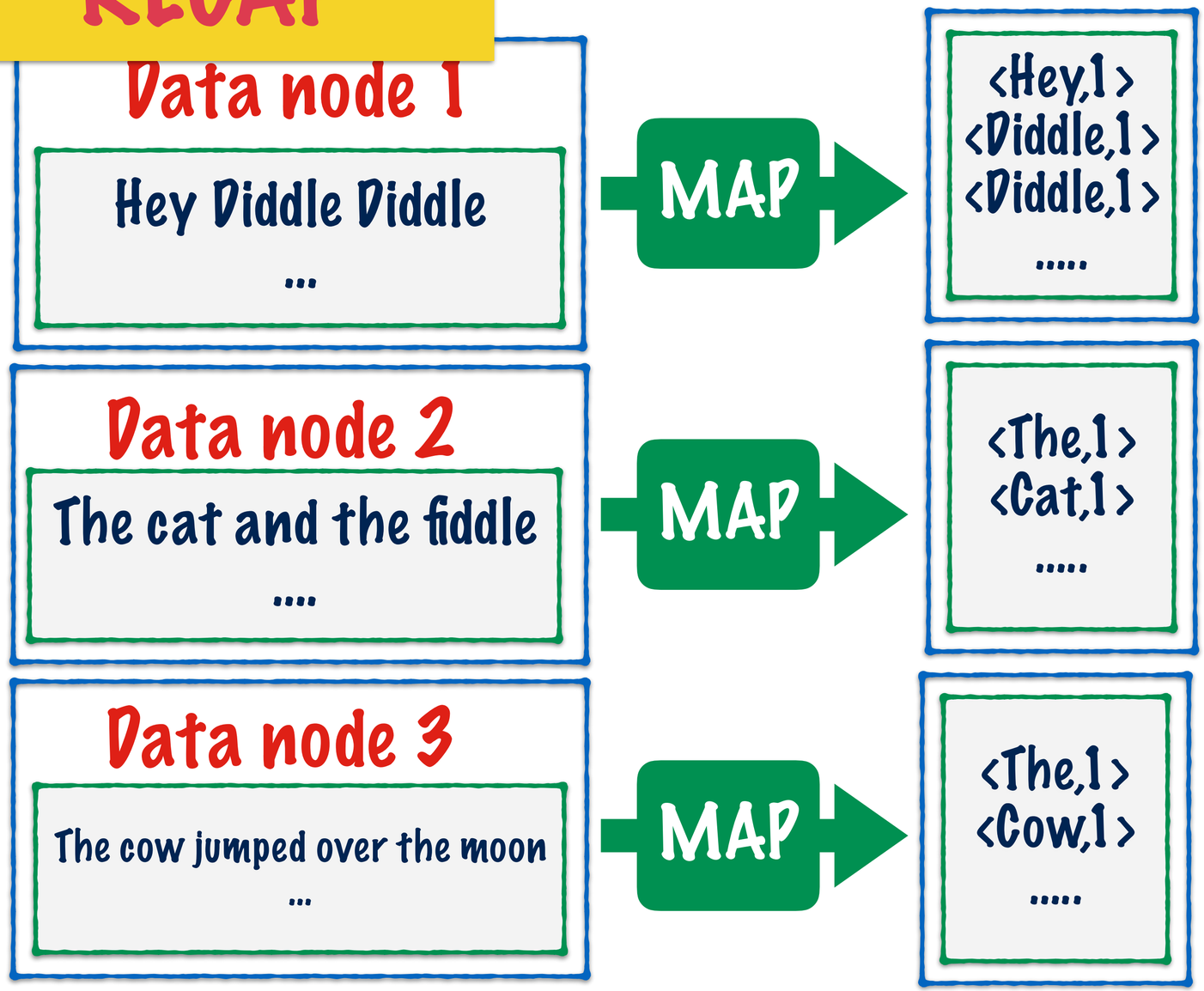
RECAP



Each block here would represent a part of the text file

Here is a snapshot of some text from each block

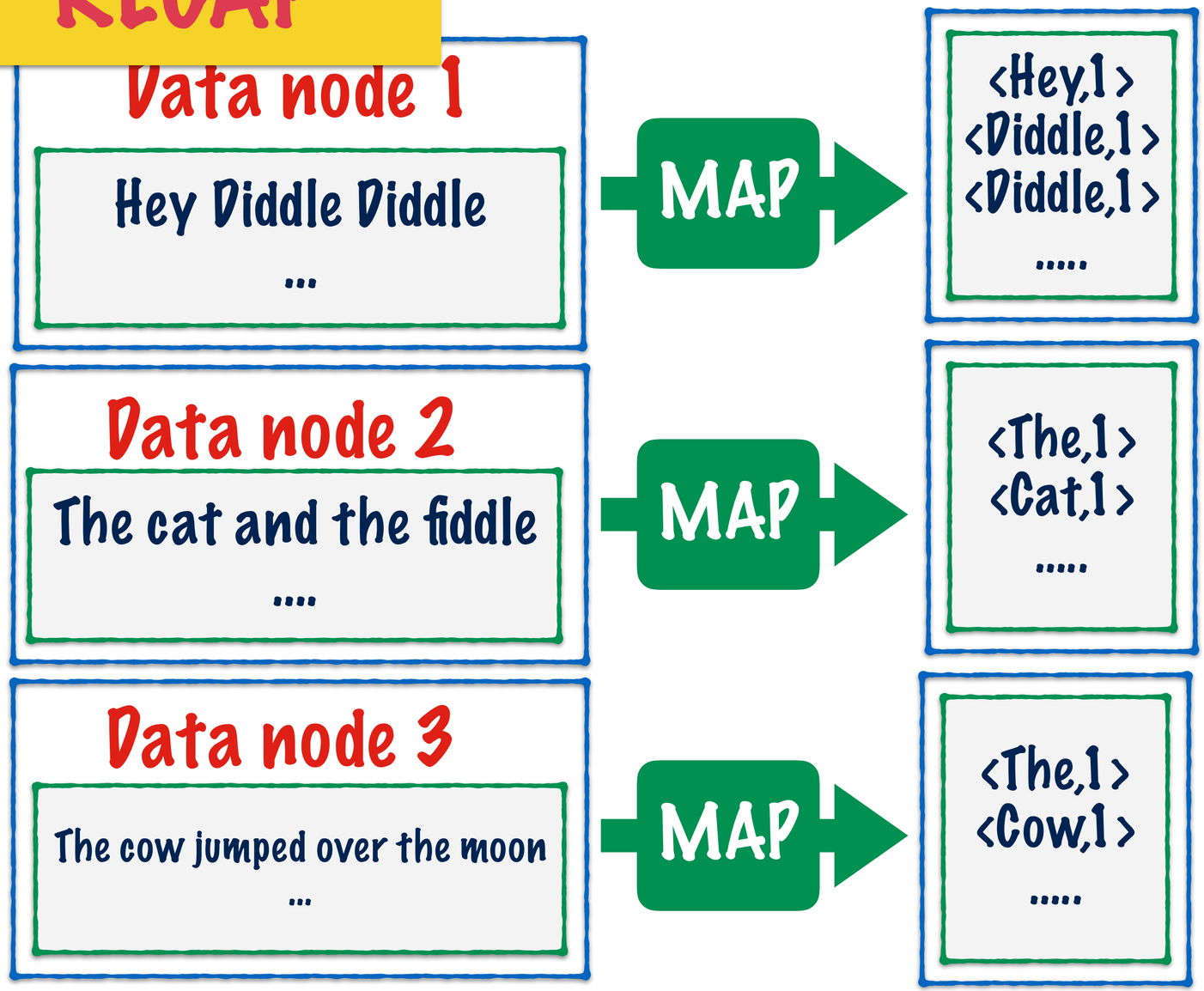
RECAP



The map step will generate a list of key-value pairs on each node

From now on all the inputs and outputs are formatted as <key,value> pairs

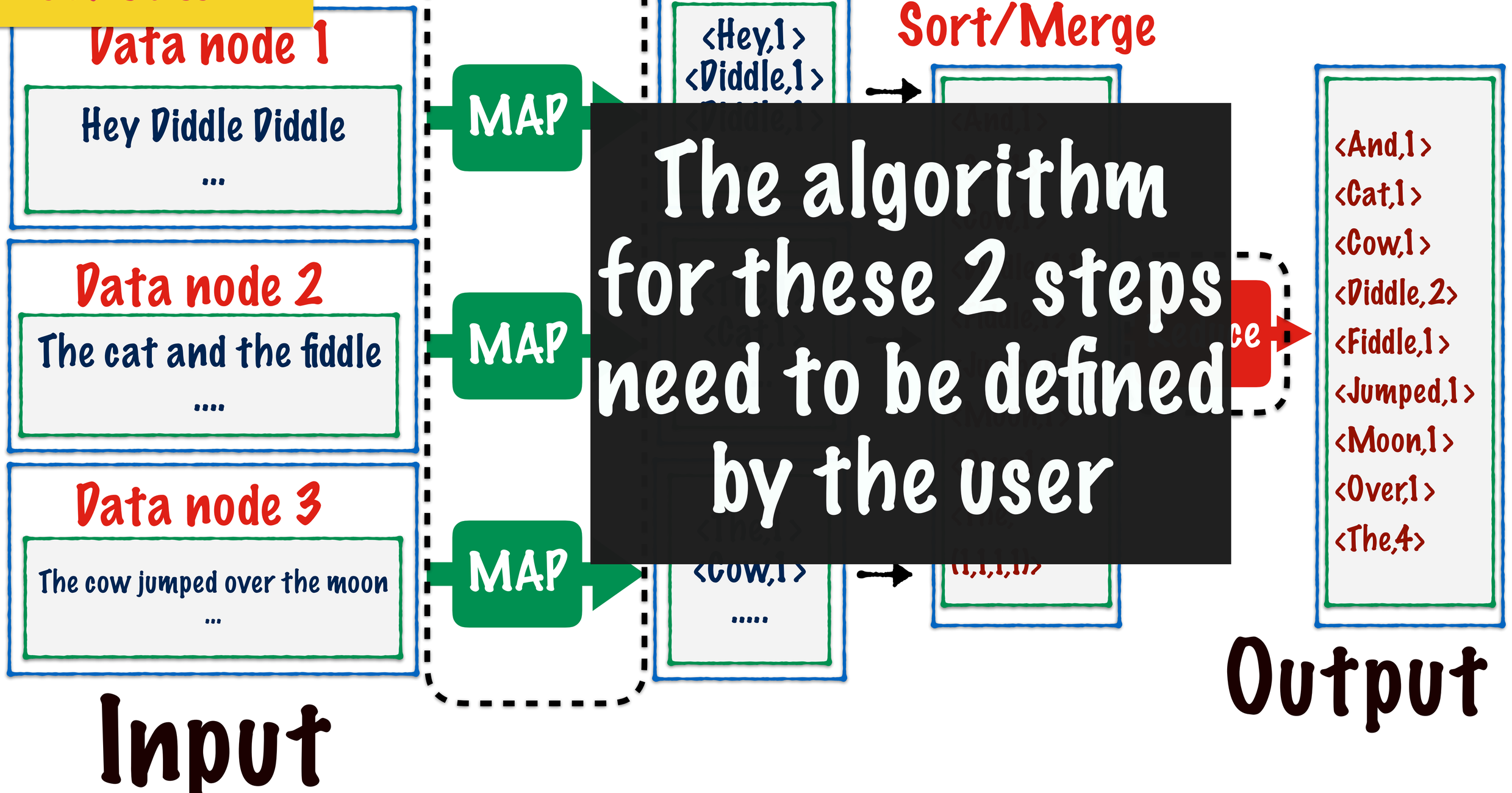
RECAP

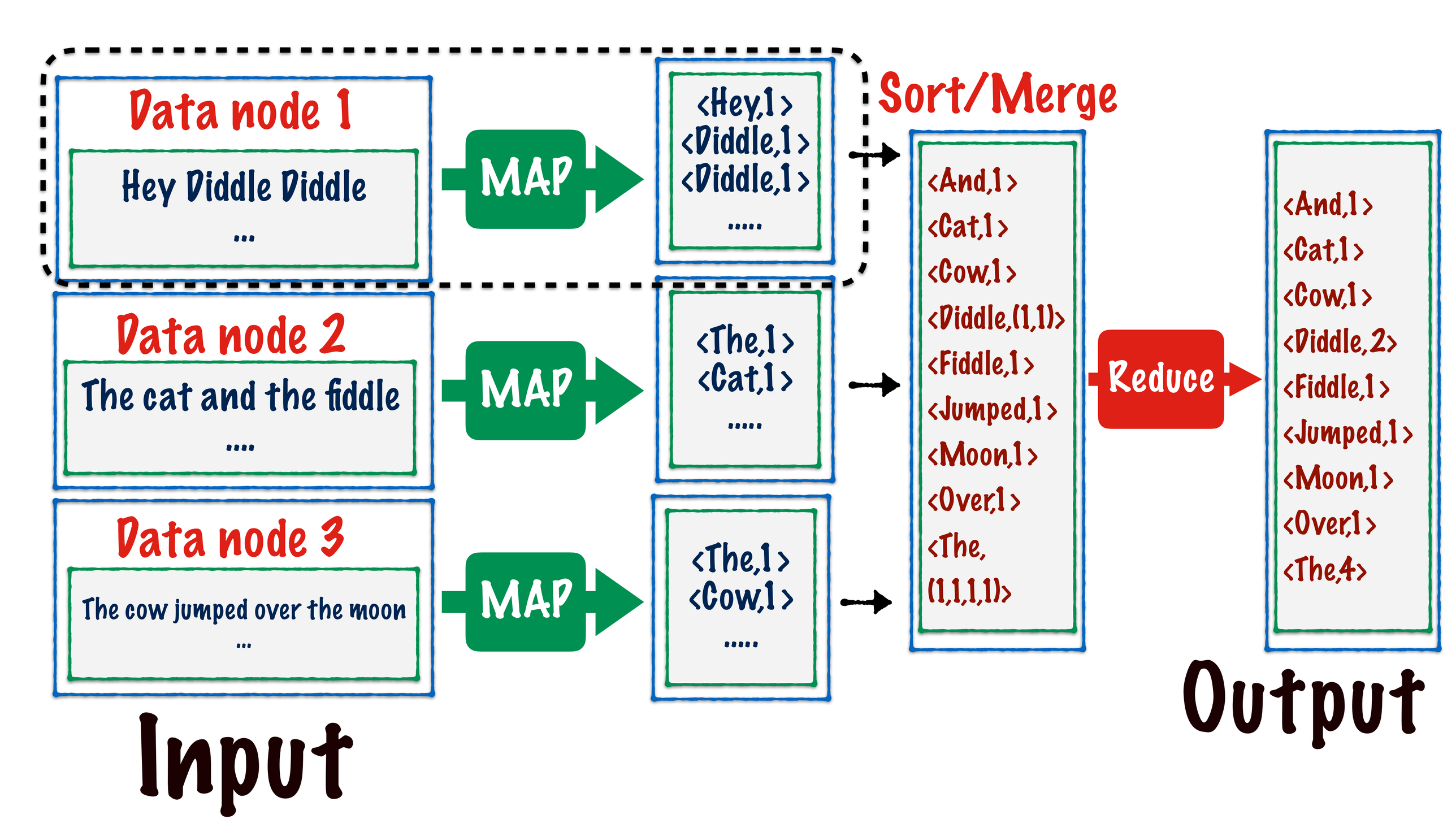


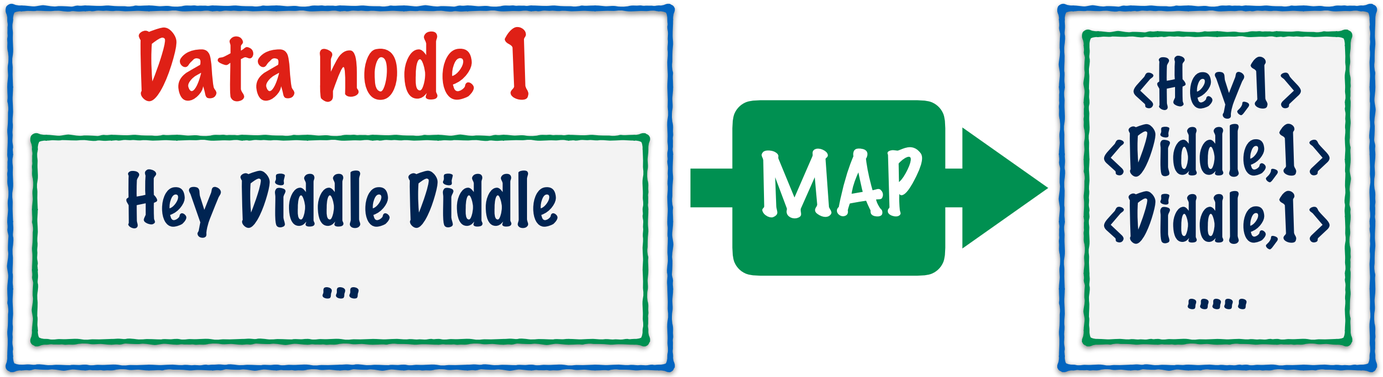
These are all copied over to one single node

On that node an operation called Sort/Merge occurs

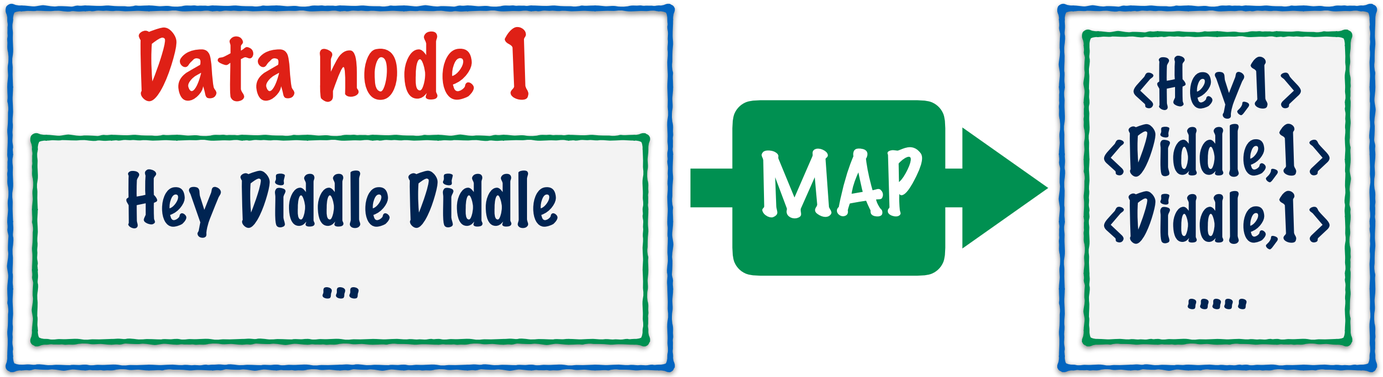
RECAP


Data node 1
Hey Diddle Diddle
...
Data node 2
The cat and the fiddle
....
Data node 3
The cow jumped over the moon
...
MAP
MAP
MAP
<Hey,1>
<Diddle,1>
<Cow,1>
.....
Sort/Merge
<And,1>
<Cat,1>
<Cow,1>
<Diddle,2>
<Fiddle,1>
<Jumped,1>
<Moon,1>
<Over,1>
<The,4>
The algorithm for these 2 steps need to be defined by the user
Input
Output


Data node 1
Hey Diddle Diddle
...
Data node 2
The cat and the fiddle
....
Data node 3
The cow jumped over the moon
...
MAP
MAP
MAP
<Hey,1>
<Diddle,1>
<Diddle,1>
.....
<The,1>
<Cat,1>
.....
<The,1>
<Cow,1>
.....
Sort/Merge
<And,1>
<Cat,1>
<Cow,1>
<Diddle,(1,1)>
<Fiddle,1>
<Jumped,1>
<Moon,1>
<Over,1>
<The,
(1,1,1,1)>
Reduce
<And,1>
<Cat,1>
<Cow,1>
<Diddle,2>
<Fiddle,1>
<Jumped,1>
<Moon,1>
<Over,1>
<The,4>
Input
Output

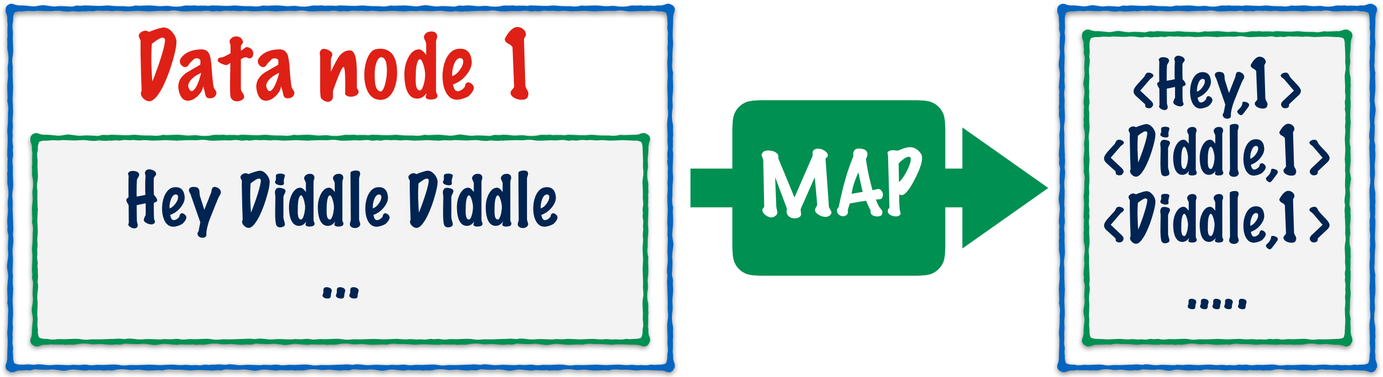


Let's concentrate on the operation in each data node



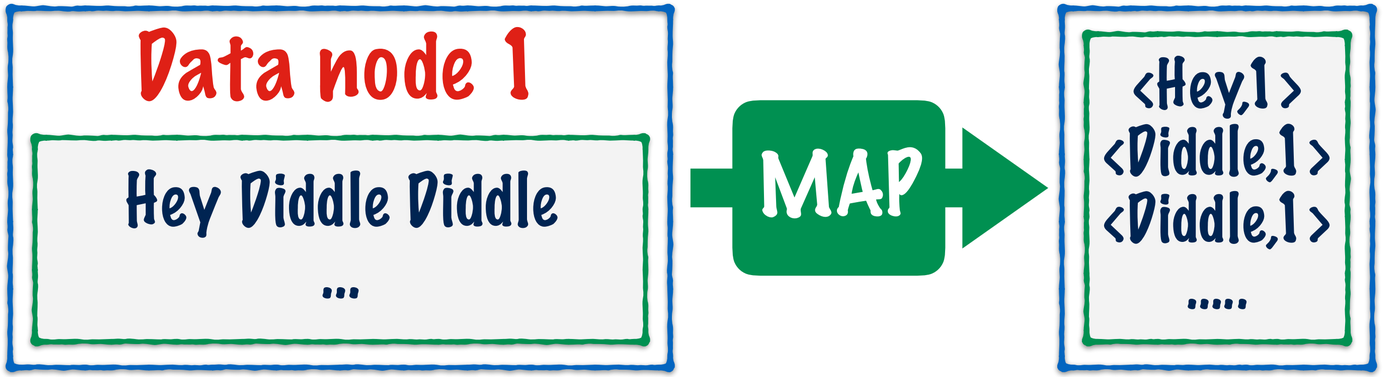
Map operation
produces a list of key-value pairs

Reduce operation
combines pairs with the same key

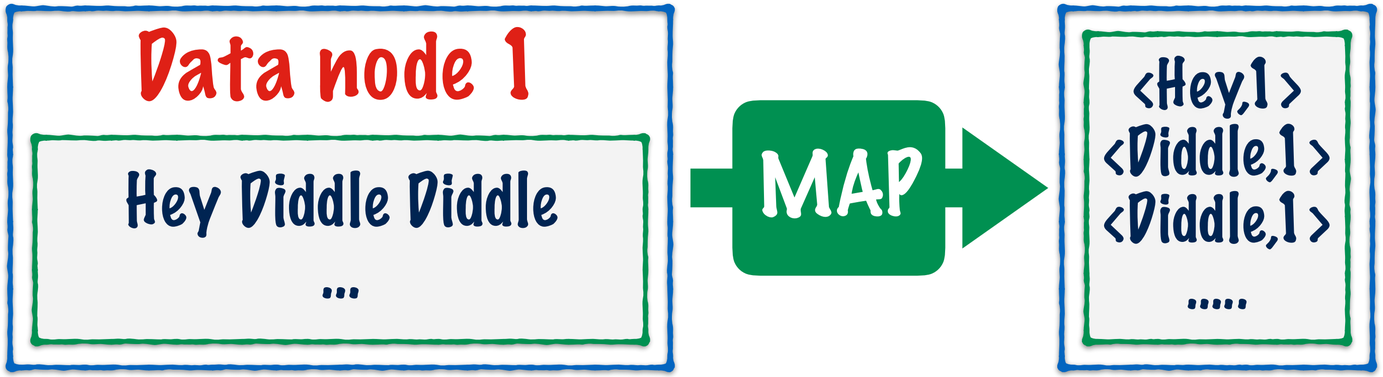


Map operation is highly parallelized

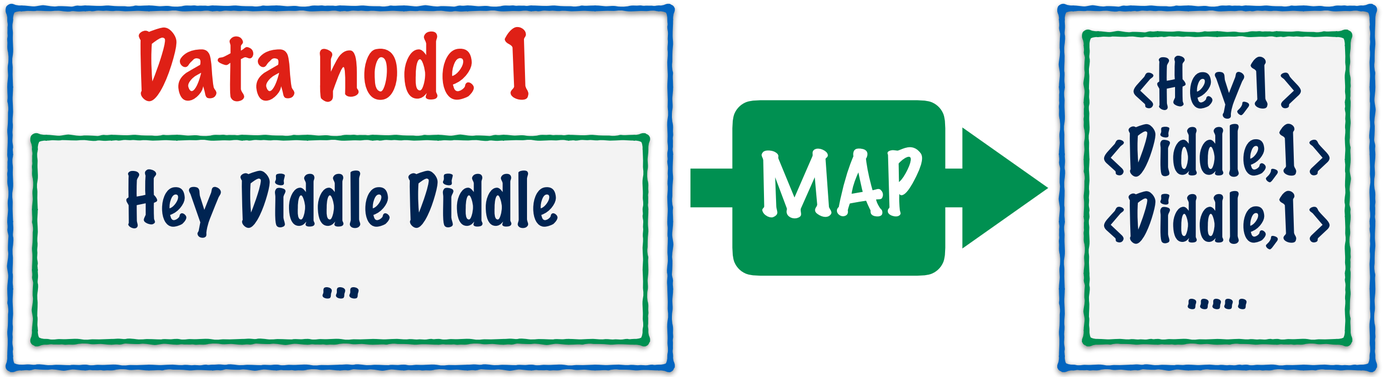
Reduce operation currently has no parallelization



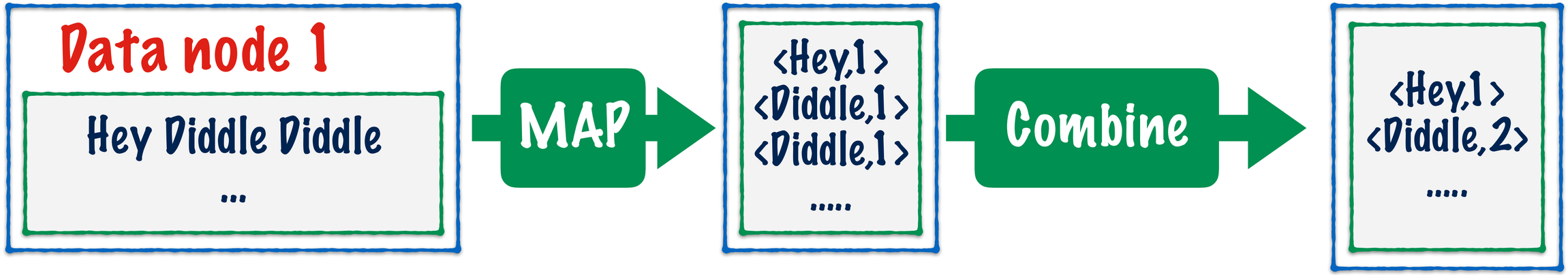
What if some part of the Reduce operation could happen on each data node ie. in parallel?



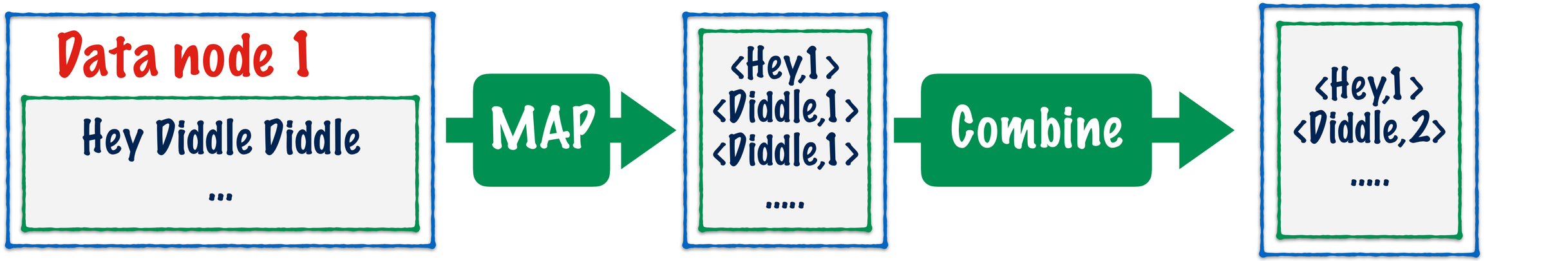
Essentially, we want to combine key-value pairs with same key - before they are copied over for reduce



Use a combiner function



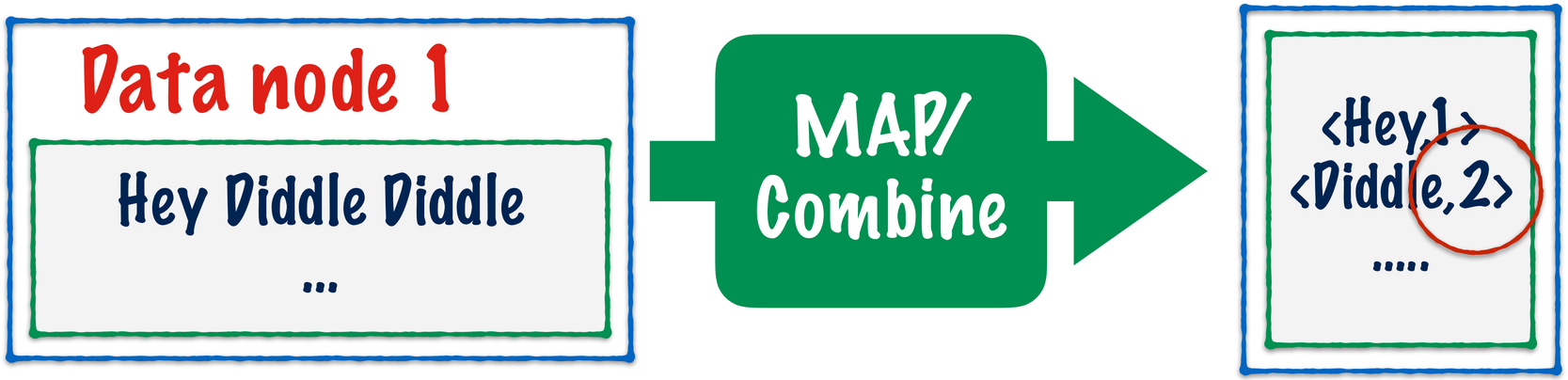
Use a combiner function



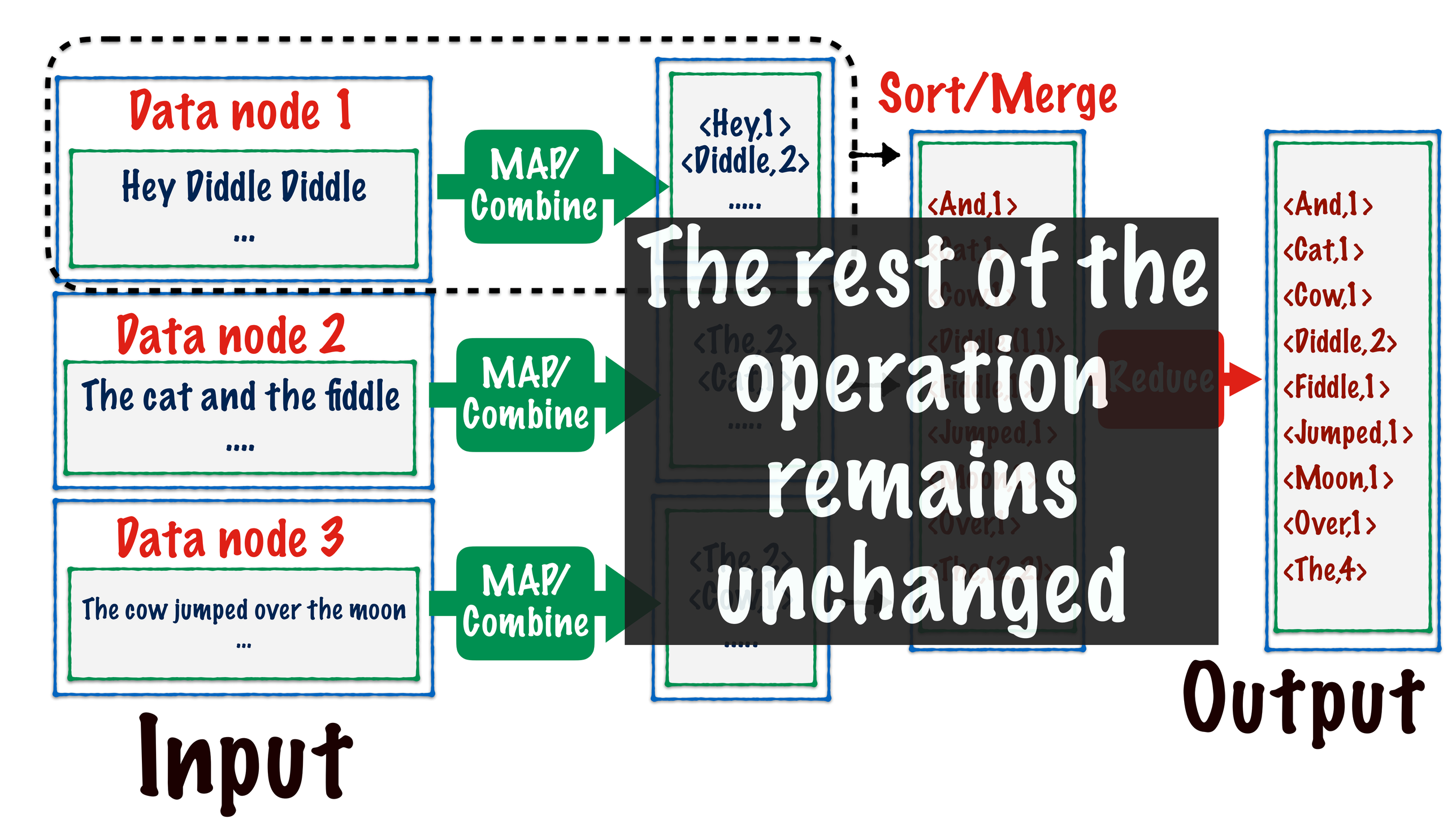
<Word, 1>

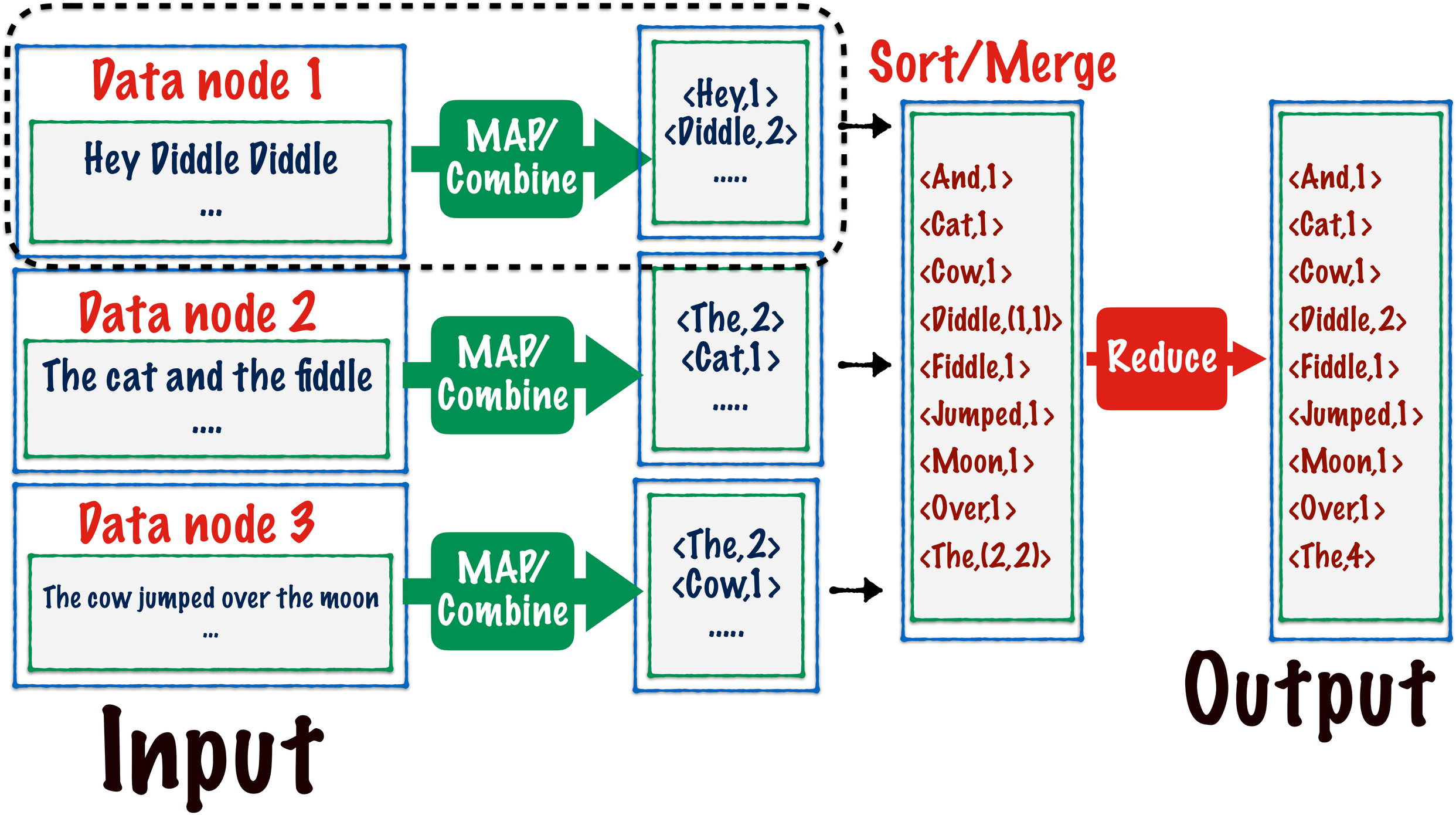
<Word, Count>

This function has combined key-value pairs with the same key



This function has
combined key-value
pairs with the same key


Data node 1
Hey Diddle Diddle
...
Data node 2
The cat and the fiddle
....
Data node 3
The cow jumped over the moon
...
MAP/ Combine
MAP/ Combine
MAP/ Combine
<Hey,1>
<Diddle,2>
.....
Sort/Merge
<And,1>
Reduce
<And,1>
<Cat,1>
<Cow,1>
<Diddle,2>
<Fiddle,1>
<Jumped,1>
<Moon,1>
<Over,1>
<The,4>
The rest of the operation remains unchanged
Input
Output


Data node 1
Hey Diddle Diddle
...
Data node 2
The cat and the fiddle
....
Data node 3
The cow jumped over the moon
...
MAP/ Combine
MAP/ Combine
MAP/ Combine
<Hey,1>
<Diddle,2>
.....
<The,2>
<Cat,1>
.....
<The,2>
<Cow,1>
.....
Sort/Merge
<And,1>
<Cat,1>
<Cow,1>
<Diddle,(1,1)>
<Fiddle,1>
<Jumped,1>
<Moon,1>
<Over,1>
<The,(2,2)>
Reduce
<And,1>
<Cat,1>
<Cow,1>
<Diddle,2>
<Fiddle,1>
<Jumped,1>
<Moon,1>
<Over,1>
<The,4>
Input
Output

Implementing a Combiner in MapReduce

We already wrote code for the following 3 steps

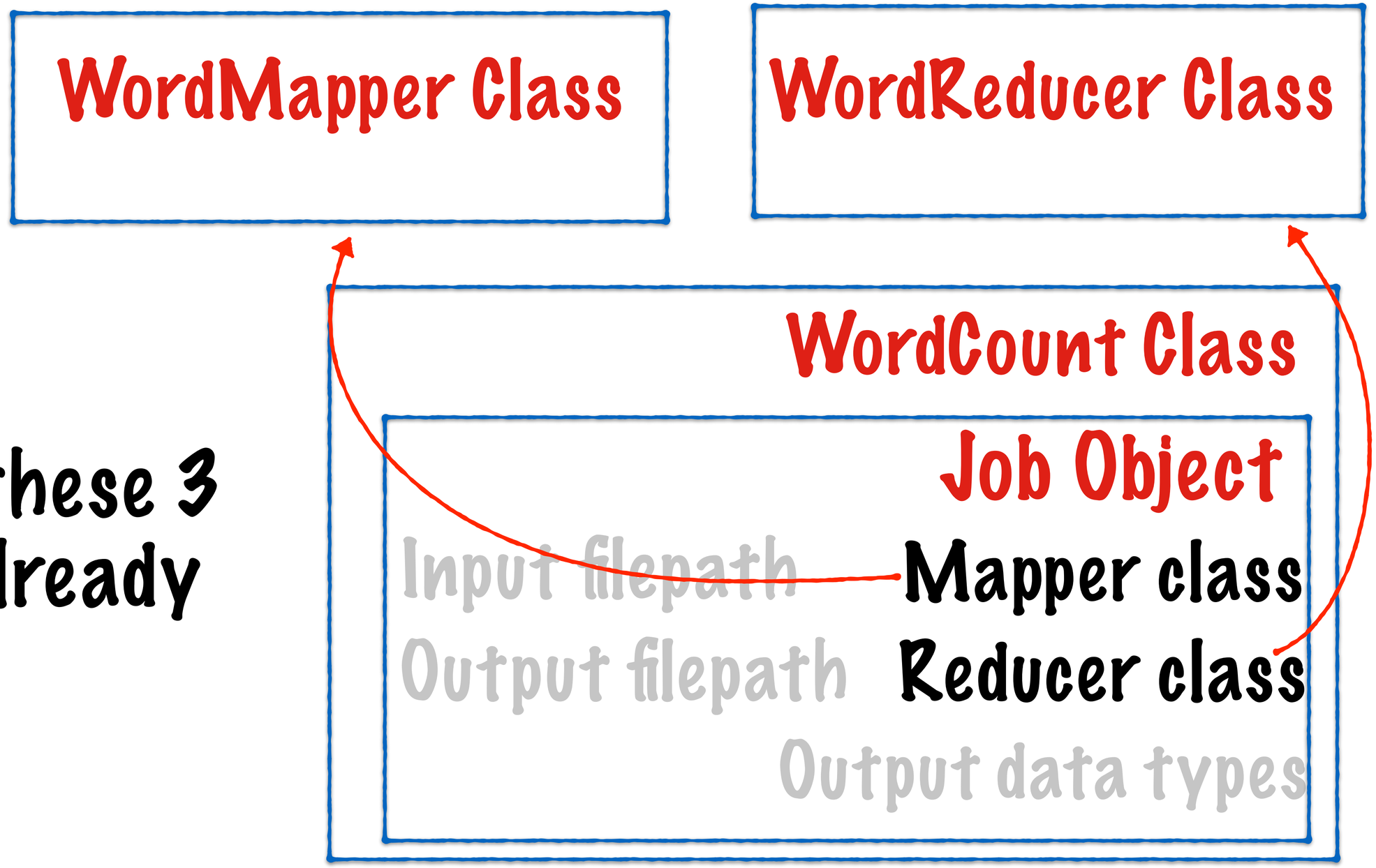
Step 1: Write a map() function

Step 1a: Write a combiner() function

Step 2: Write a reduce() function

Step 3: Setup a driver that points to our map and reduce implementations

We have these 3 classes already



WordMapper Class

WordReducer Class

The Combiner function is implemented by extending the Reducer class

The same class that WordReducer inherits from

WordCount Class

Job Object

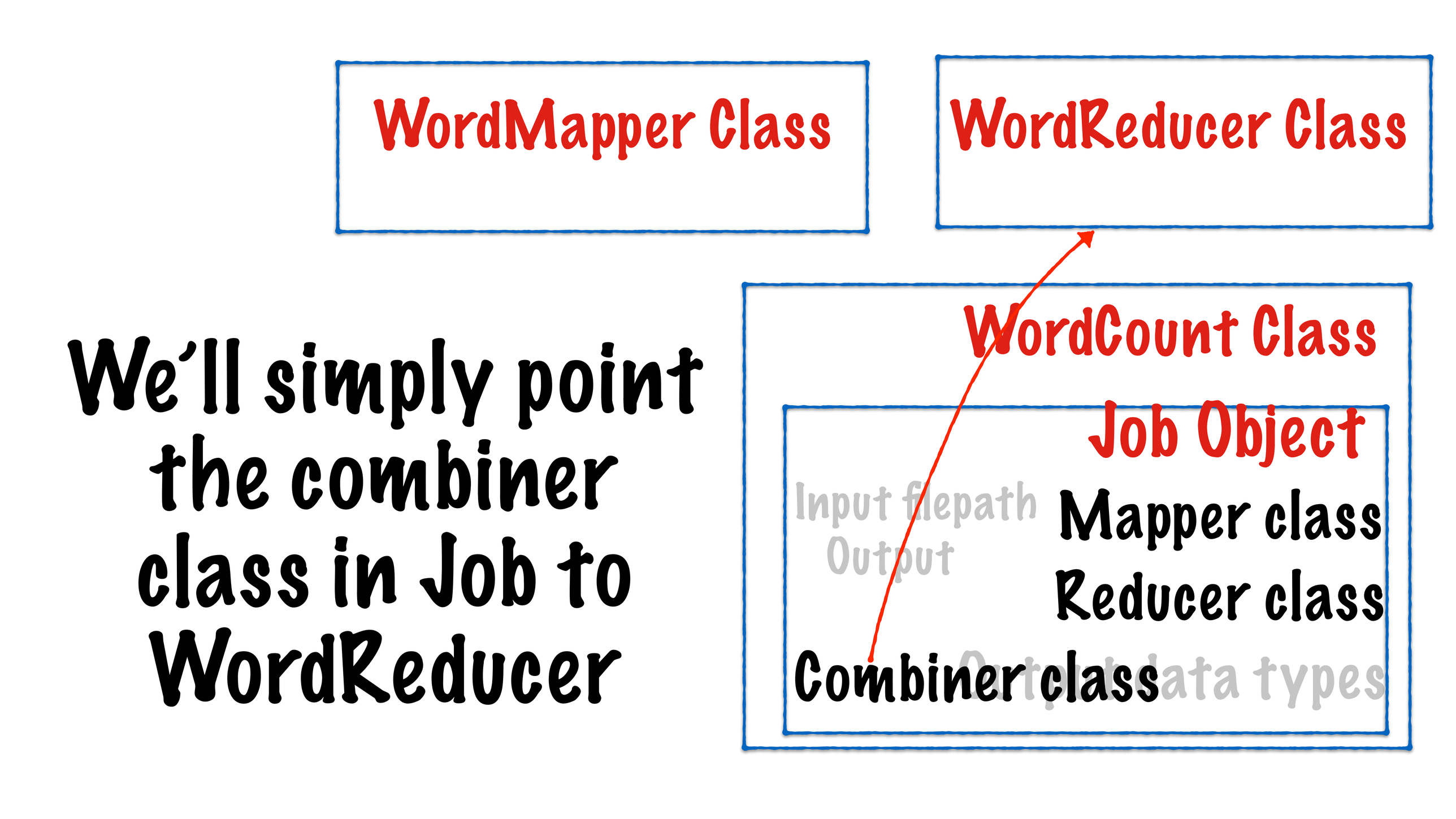
Input filepath

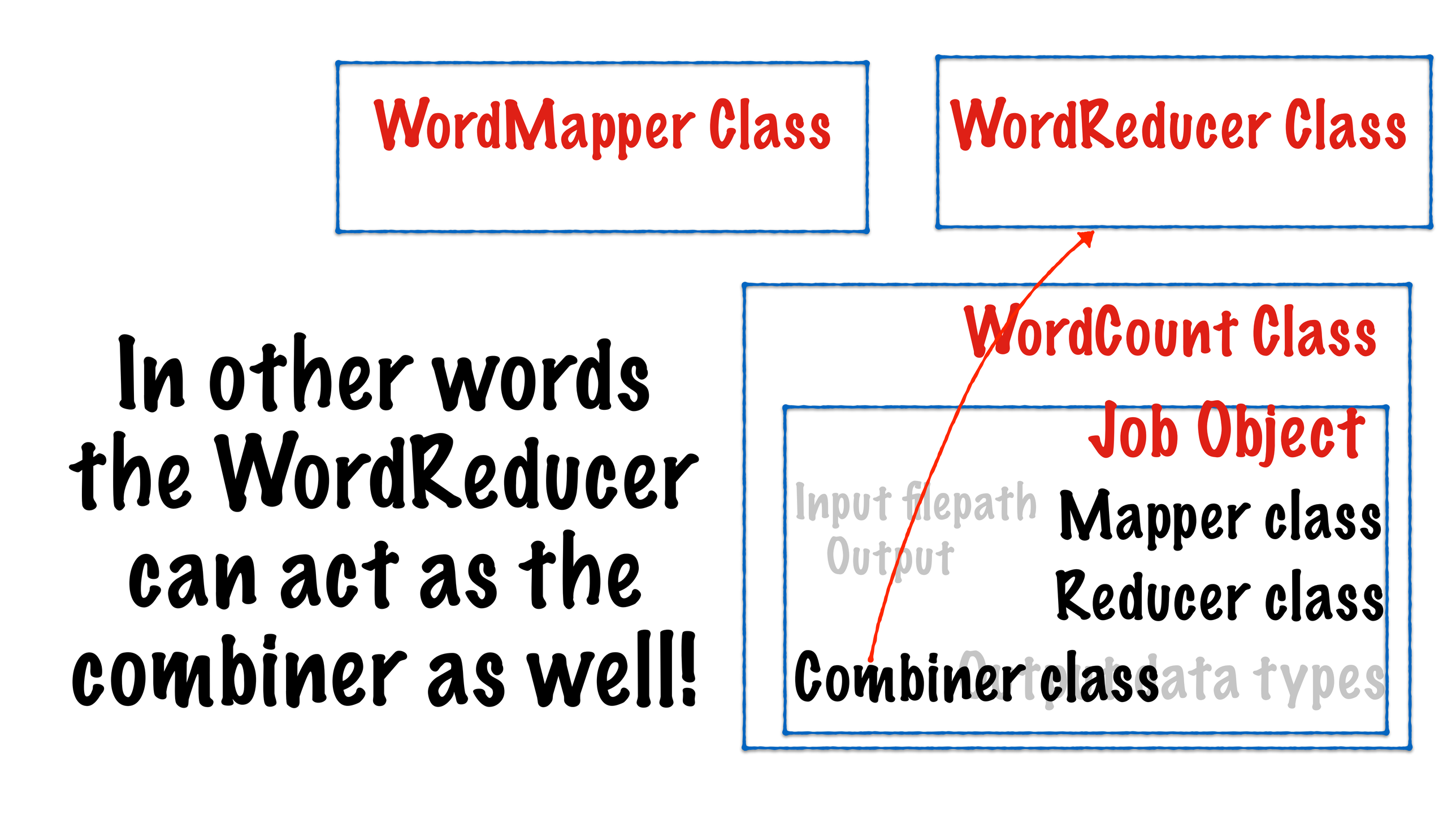
Mapper class

Output

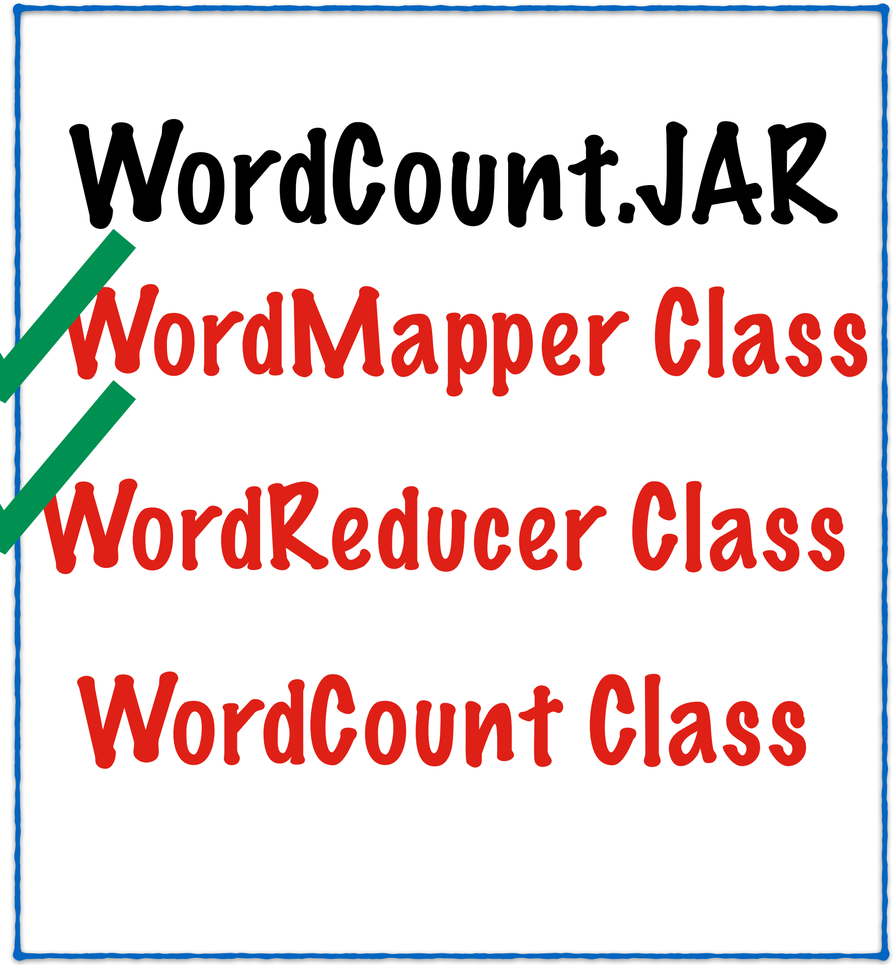
Reducer class

Output data types


WordMapper Class
WordReducer Class
We'll simply point the combiner class in Job to WordReducer
WordCount Class
Job Object
Input filepath
Output
Mapper class
Reducer class
Combiner class
Output data types


WordMapper Class
WordReducer Class
WordCount Class
Job Object
Input filepath
Output
Mapper class
Reducer class
Combiner class
Output data types
In other words
the WordReducer
can act as the
combiner as well!

We don't need to change these 2 classes



WordCount Class

```java
package com.company;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

    public static void main(String[] args) throws Exception {
        if(args.length !=2){
            System.err.println("Invalid Command");
            System.err.println("Usage: WordCount <input path> <output path>");
            System.exit(0);
        }
        Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

        job.setJarByClass(WordCount.class);



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

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

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

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

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

Here we just need to make 1 change

WordCount Class

```
package com.company;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

    public static void main(String[] args) throws Exception {
        if(args.length !=2){
            System.err.println("Invalid Command");
            System.err.println("Usage: WordCount <input path> <output path>");
            System.exit(0);
        }
        Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

        job.setJarByClass(WordCount.class);

        job.setJobName("Word Count");

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

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

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

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

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

We set the Mapper class and Reducer class to the classes we've written

WordCount Class

Set the combiner class also

```
package com.company;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

    public static void main(String[] args) throws Exception {
        if(args.length !=2){
            System.err.println("Invalid Command");
            System.err.println("Usage: WordCount <input path> <output path>");
            System.exit(0);
        }
        Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

        job.setJarByClass(WordCount.class);

        job.setJobName("Word Count");

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

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

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);
        job.setCombinerClass(WordCountReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

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

Constraints on the combiner function

The combiner function makes the MapReduce task faster

1. The reduce operation is partially parallelized

2. Less data needs to be transferred across the network

The combiner function should have no impact on the final result itself

Whether it is used or not, the results should remain the same

In fact, the decision of whether to use the combiner or not ultimately rests with Hadoop

Even if you specify a combiner function,
Hadoop might decide not to call it

So, the outcome of MapReduce should
be independent of the combiner

Let's see an example of a combiner
that does not satisfy this constraint

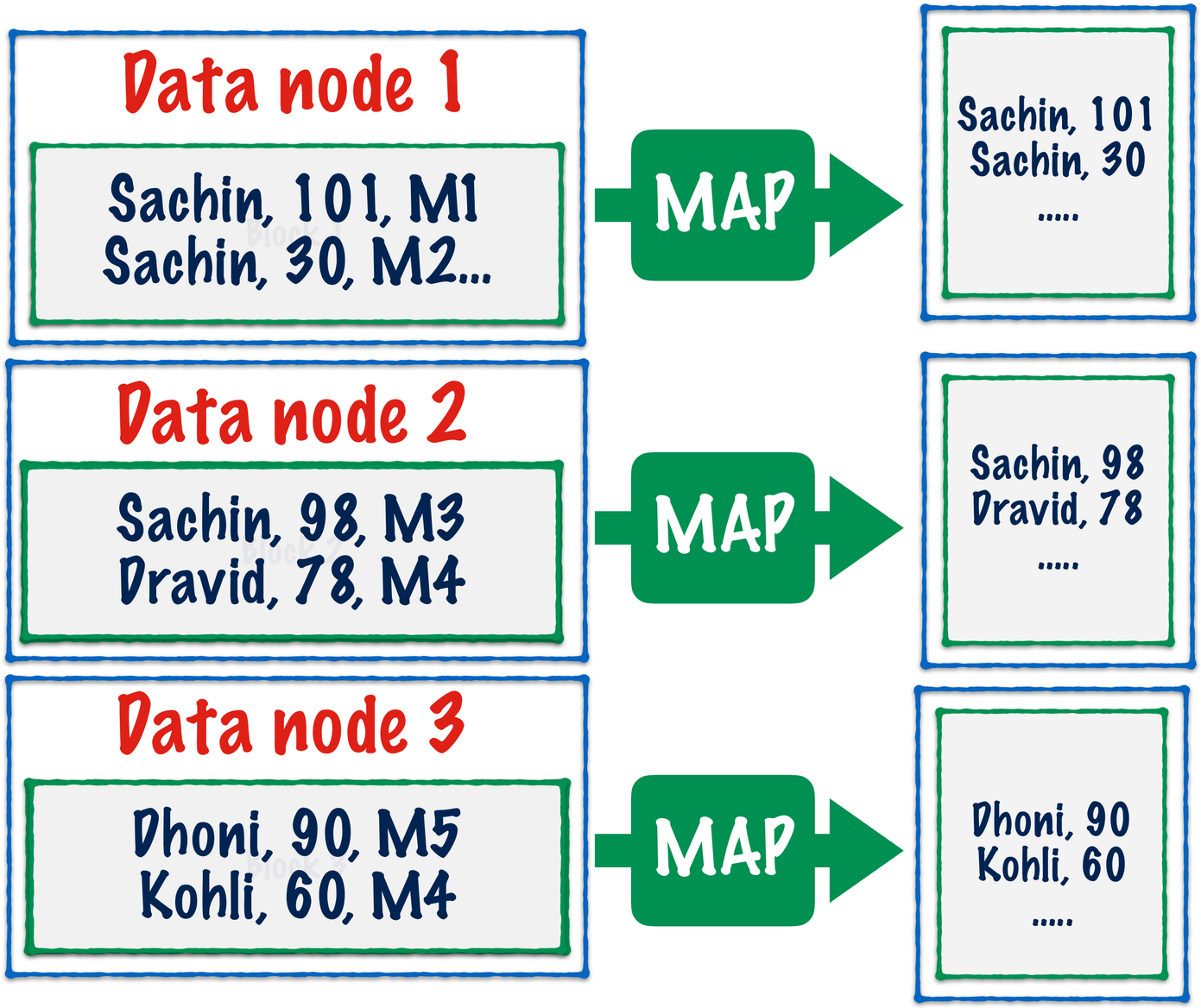
Example: A file contains 3 columns

Player, Runs scored, Match

Objective: Compute the batting average for each player

Can we use average() as the combiner function?

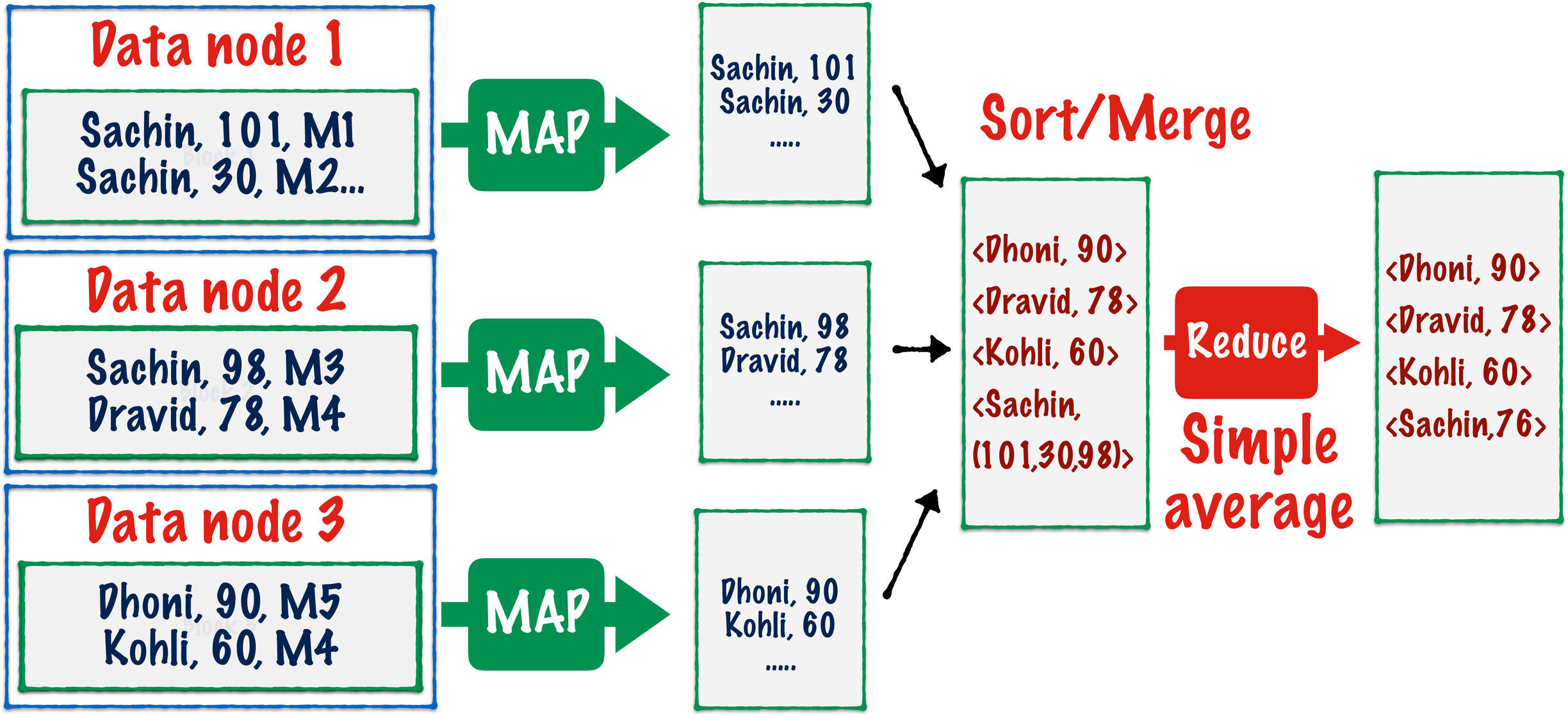
Objective: Compute the batting average for each player



Let's first see this without a combiner

Map step just picks the player name and score

Objective: Compute the batting average for each player

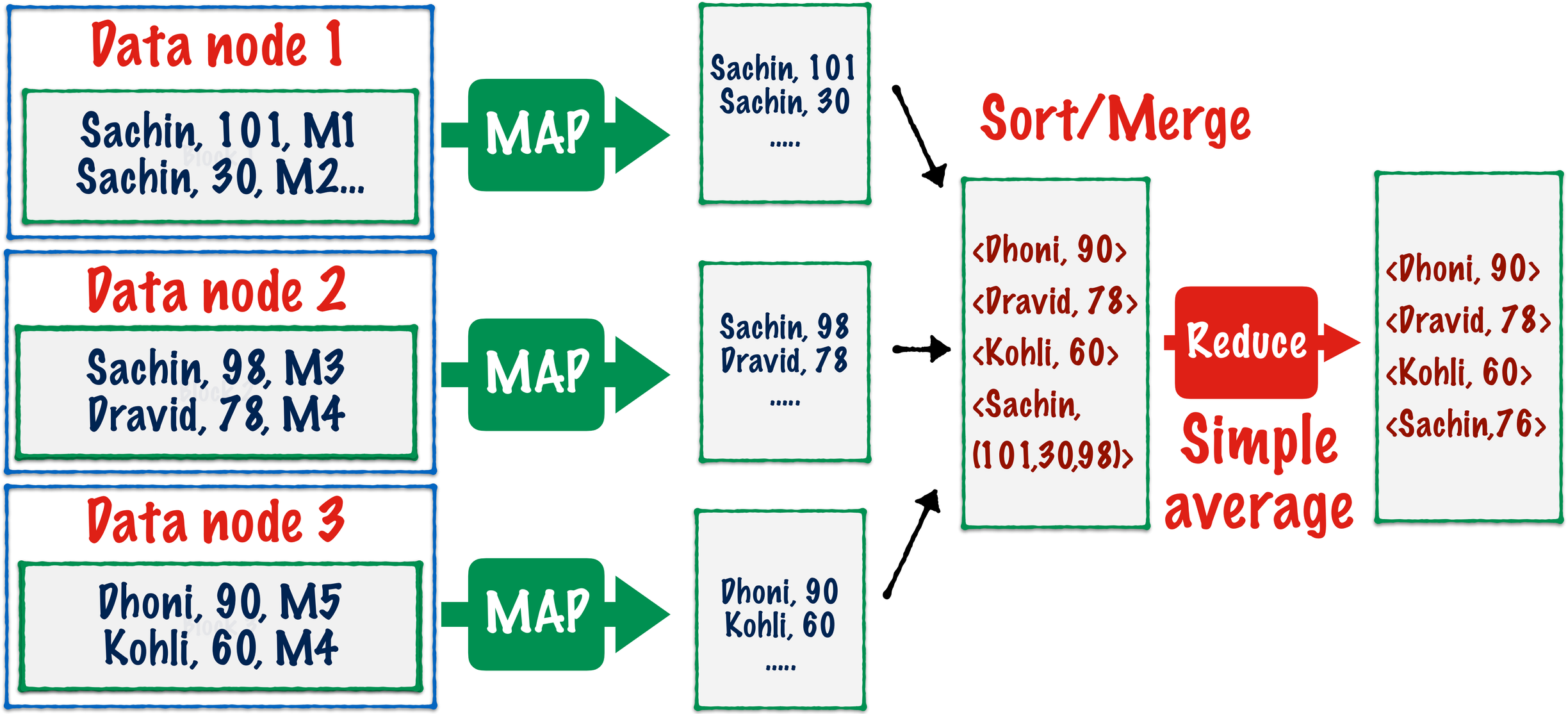


Objective: Compute the **batting average** for each player

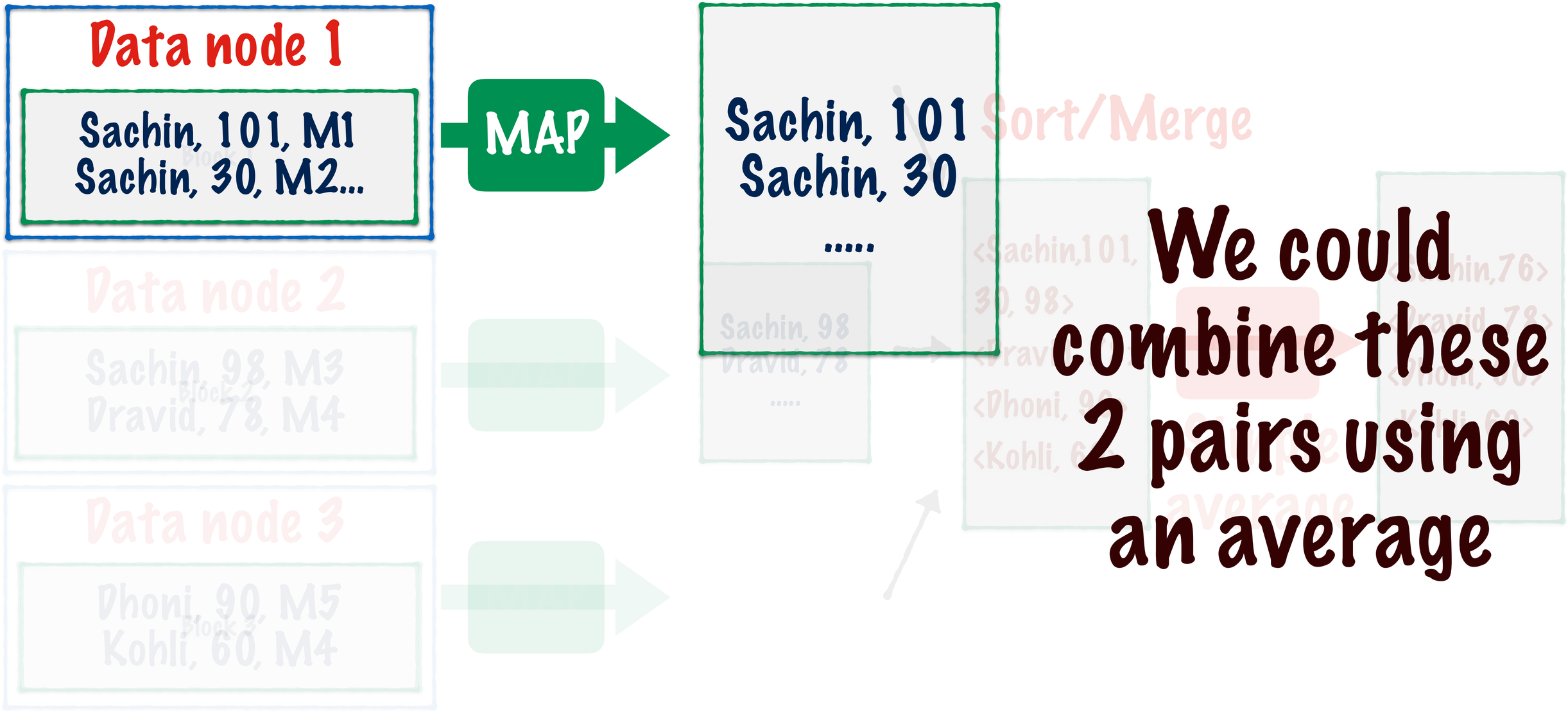
Let's see what's different with average() as the combiner function

<Dhoni, 90>
<Dravid, 78>
<Kohli, 60>
<Sachin,76>

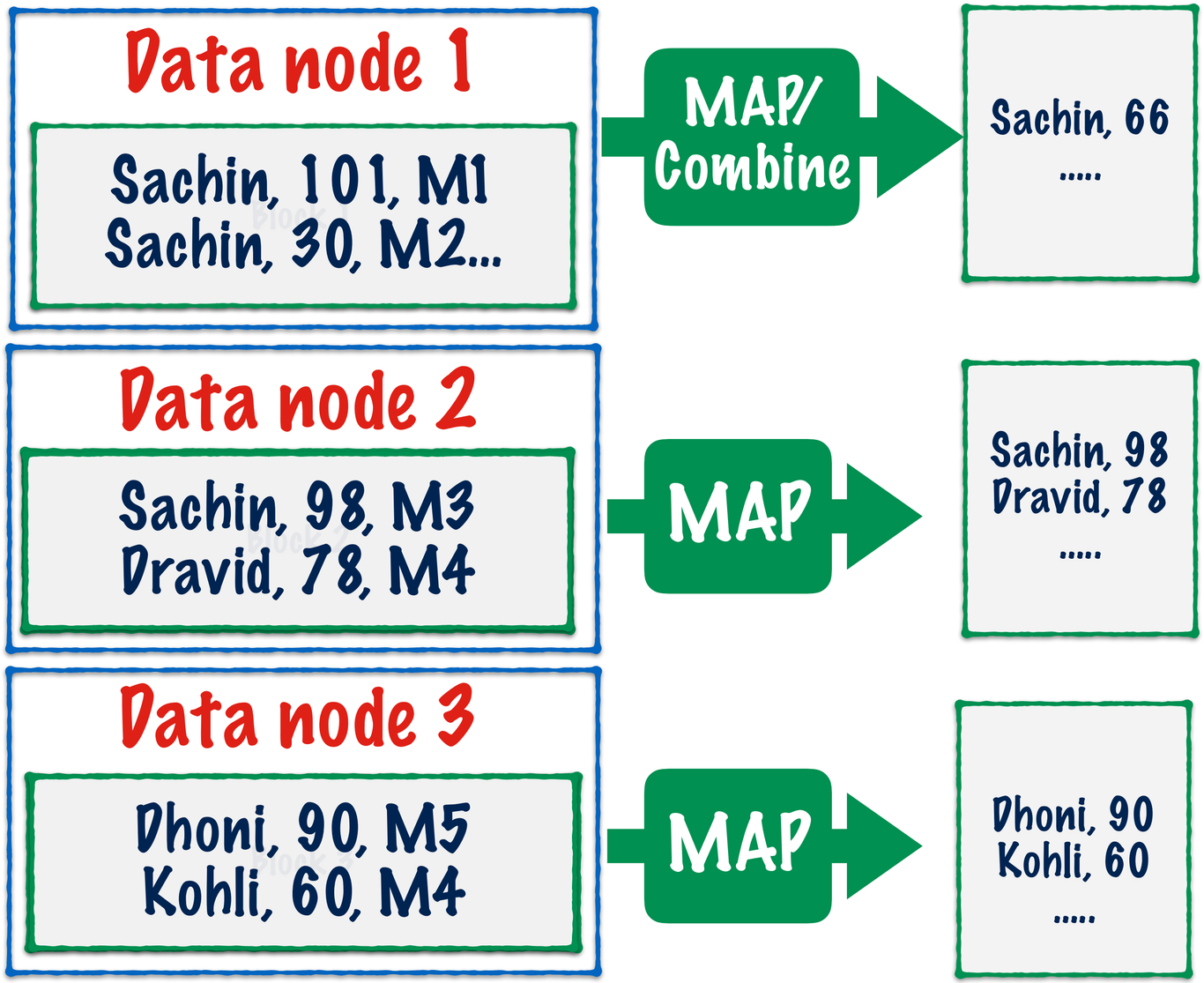
Objective: Compute the batting average for each player



Objective: Compute the batting average for each player

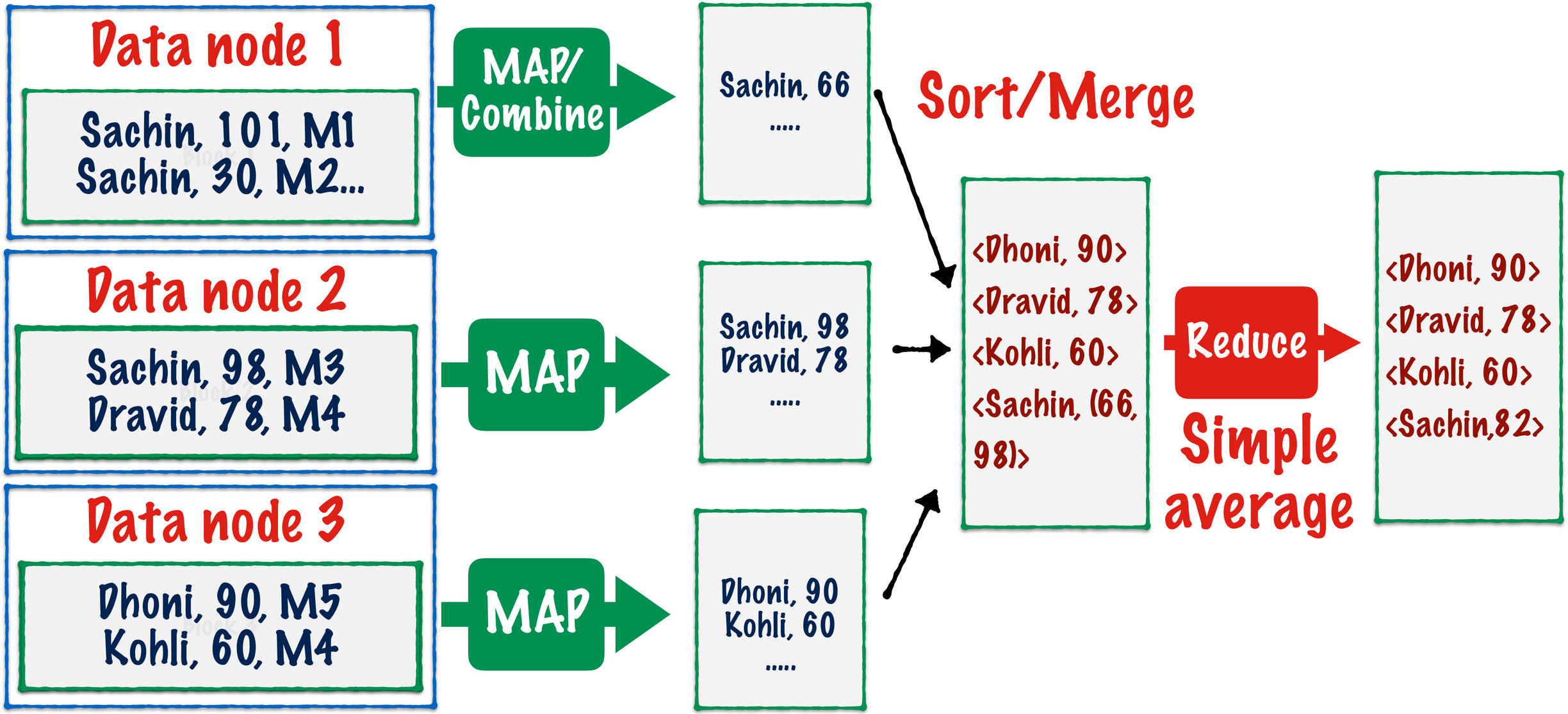


Objective: Compute the batting average for each player



Combiner won't have any impact on the other 2 nodes

Objective: Compute the batting average for each player



Objective: Compute the batting average for each player

Without Combiner	With Combiner
<Dhoni, 90>	<Dhoni, 90>
<Dravid, 78>	<Dravid, 78>
<Kohli, 60>	<Kohli, 60>
<Sachin,76>	<Sachin,82>

The result is different with and without the combiner

Objective: Compute the batting average for each player

This is because

Average of a set of numbers <> Average (Averages of subsets)

Objective: Compute the batting average for each player

In our example

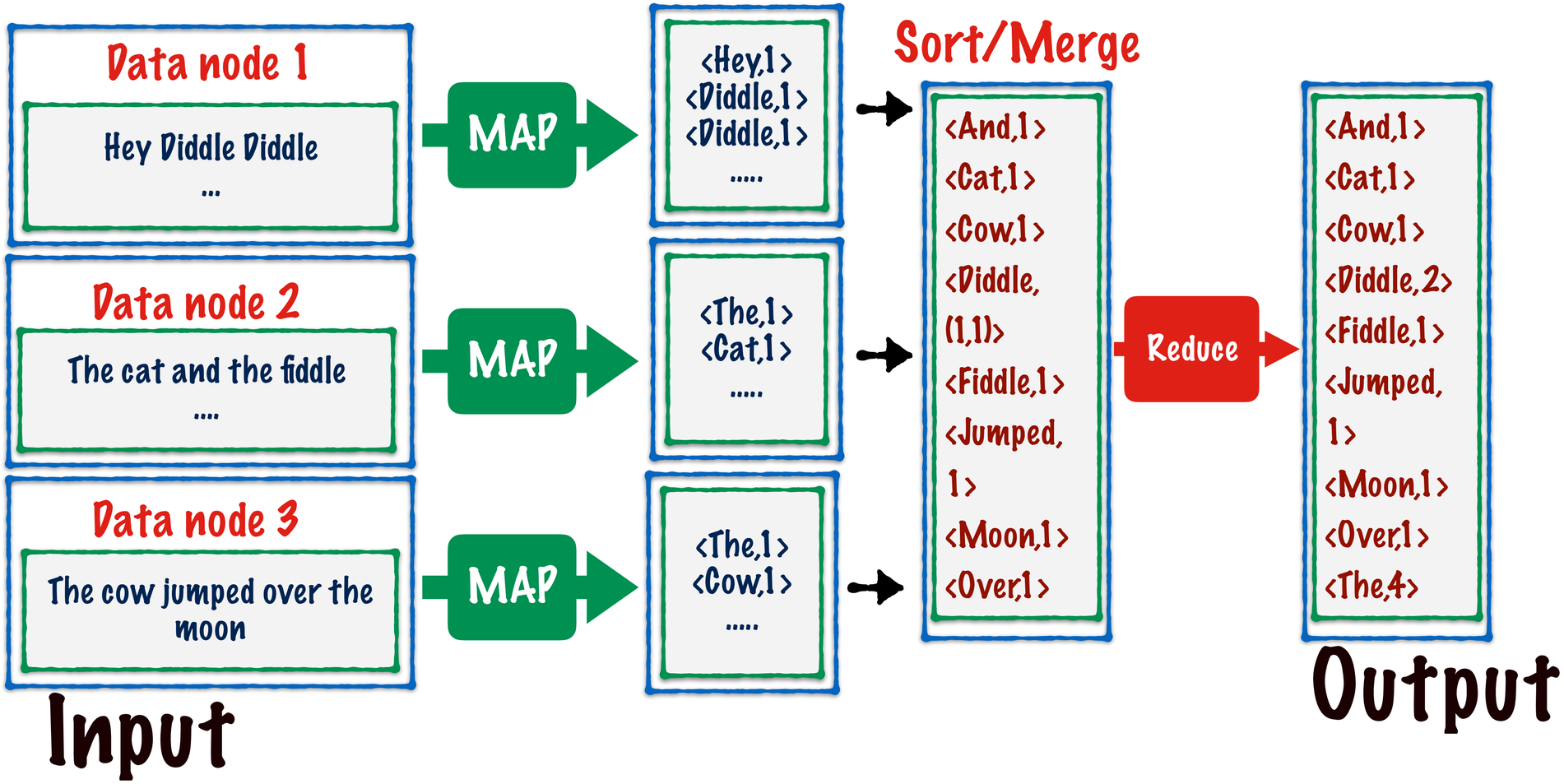
Average (101, 30, 98) <> Average (Avg(101, 30), 98)

We can't use average() as
the combiner function

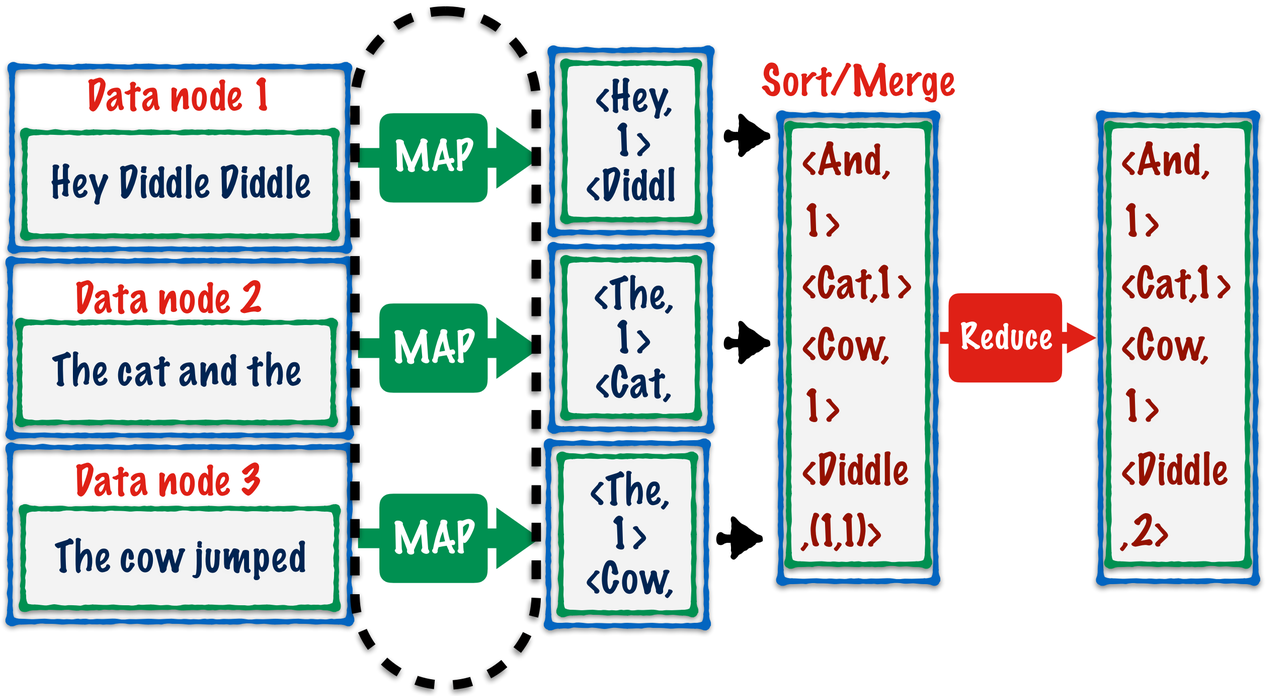
Shuffle and Sort
Using Multiple Reducers

Shuffle and Sort

Here is a MapReduce representation that we are familiar with



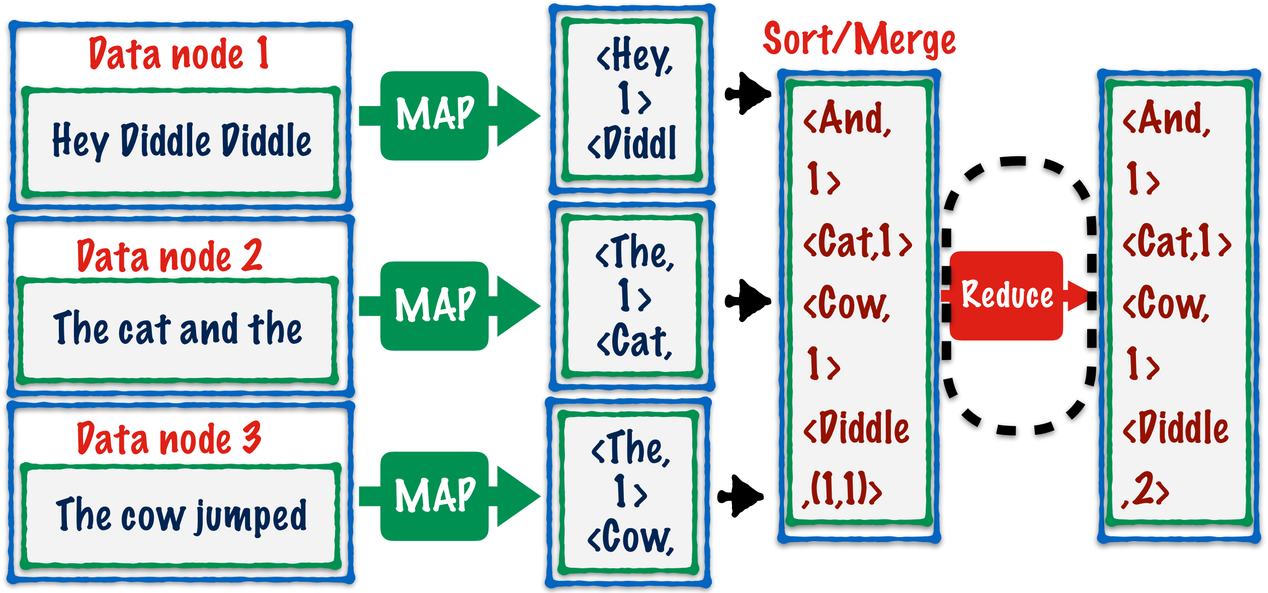
Shuffle and Sort



These represent a bunch of **map tasks** that run **in parallel**

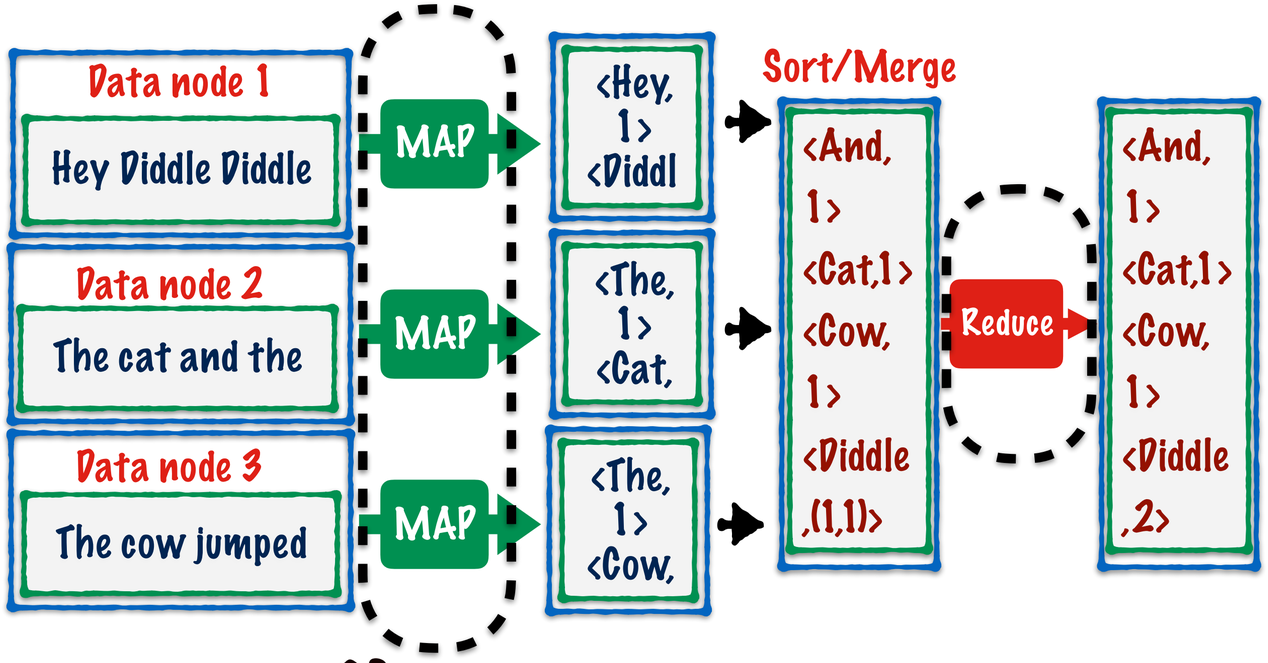
Each of these map tasks is executed by a **"process"** on a node in the cluster

Shuffle and Sort



Similarly, the reduce task is also executed by a "process"

Shuffle and Sort



All these processes are launched and managed by YARN

The map processes are called mappers

The reduce processes are called reducers

Shuffle and Sort

mappers

reducers

How many mappers and reducers does a MapReduce job require?

Shuffle and Sort

mappers

reducers

The number of mappers is usually controlled by YARN

Shuffle and Sort

mappers

reducers

Each mapper operates on one HDFS block

Shuffle and Sort

mappers

It depends on the number of "splits" i.e. the number of blocks your input file is broken into

reducers

Shuffle and Sort

mappers

= number of "splits"

reducers

This effectively means multiple mappers can be assigned to 1 Data node

Shuffle and Sort

mappers

= number of "splits"

reducers

The number of mappers **can only be controlled to a limited extent** by the user

Shuffle and Sort

mappers

= number of "splits"

reducers

The number of reducers can be controlled by the user

Shuffle and Sort

mappers

= number of "splits"

reducers

By default, there is only 1 reducer

which is what we have seen so far

Shuffle and Sort

mappers

= number of "splits"

reducers

With multiple reducers, the reduce task can also be parallelized

Shuffle and Sort

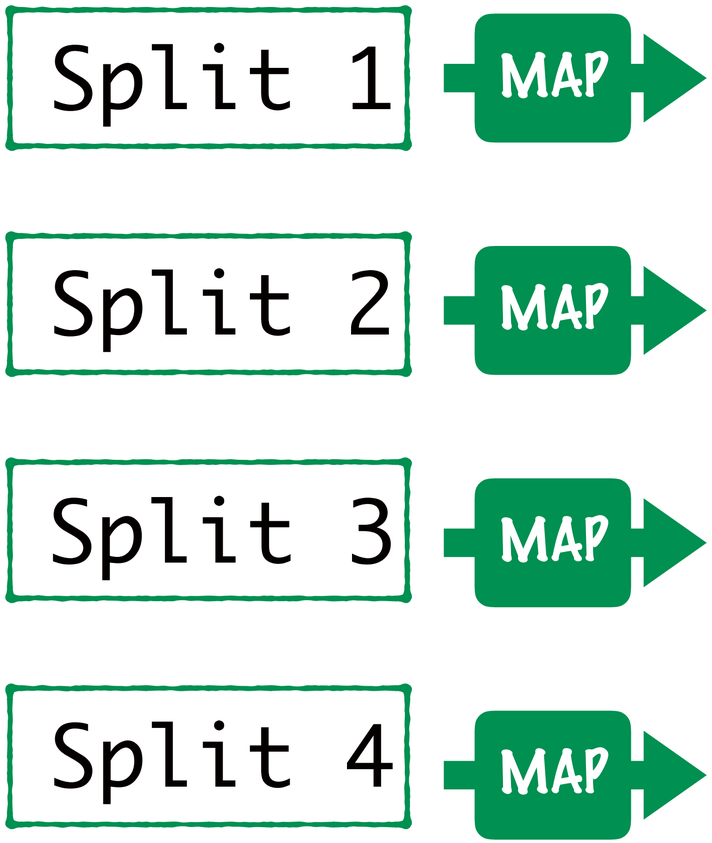
reducers

This is how you normally run a MapReduce job

```
$ hadoop jar <JARpath> <mainclass> <inputFile> <outputDir>
                                    -D mapred.reduce.tasks=2
```

The number of reducers can be specified when you run the job

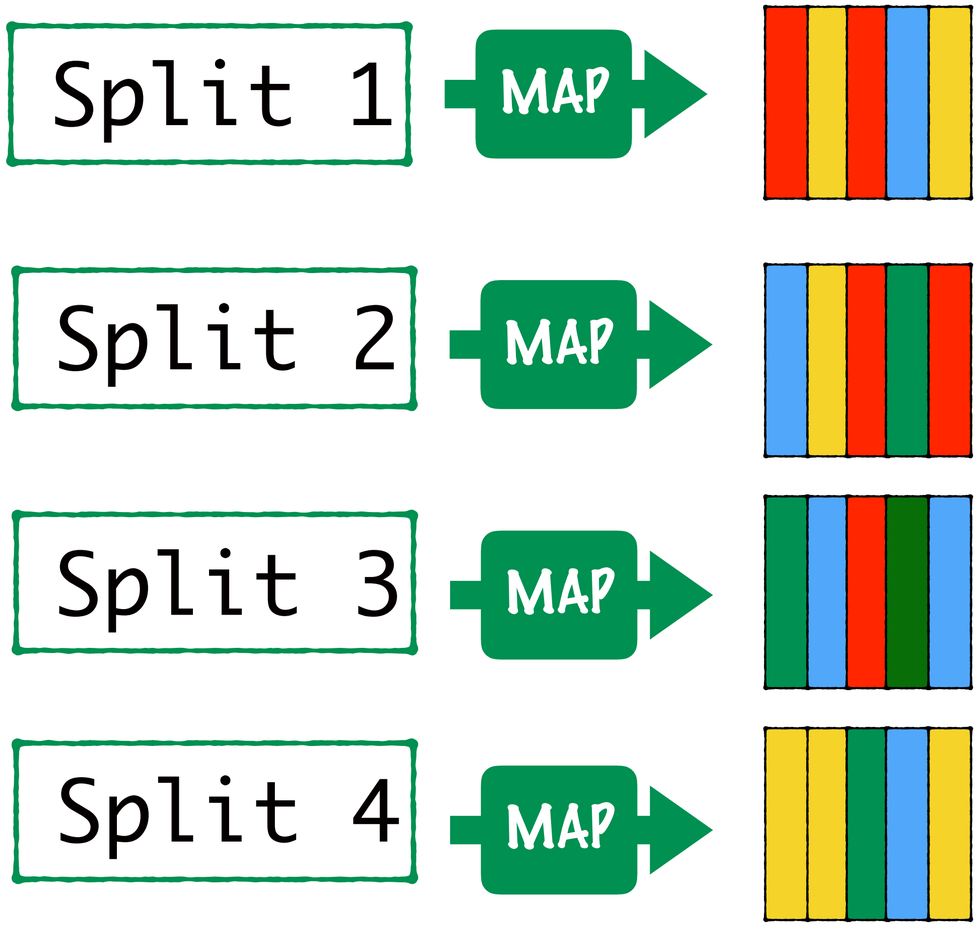
Shuffle and Sort



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

There will 4 mappers

Shuffle and Sort

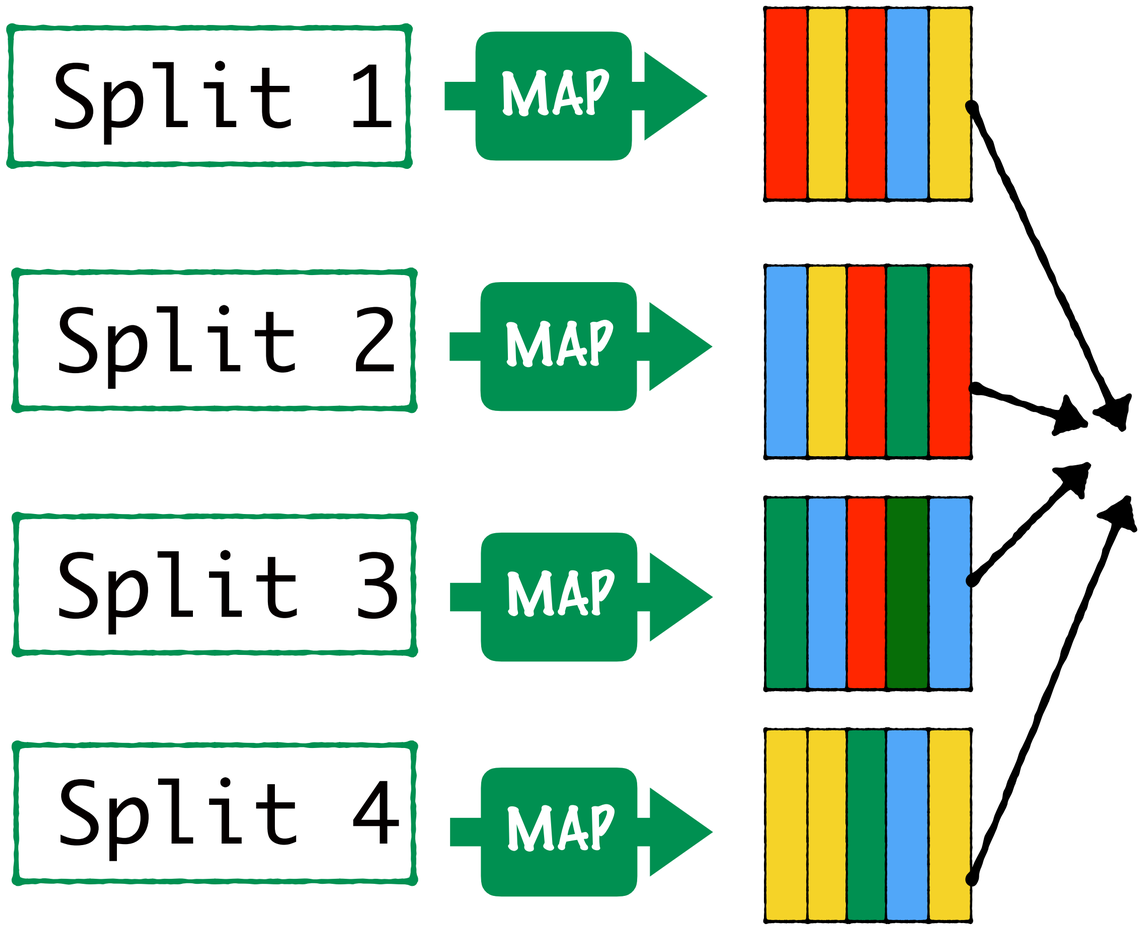


The mappers output <Key, Value> pairs

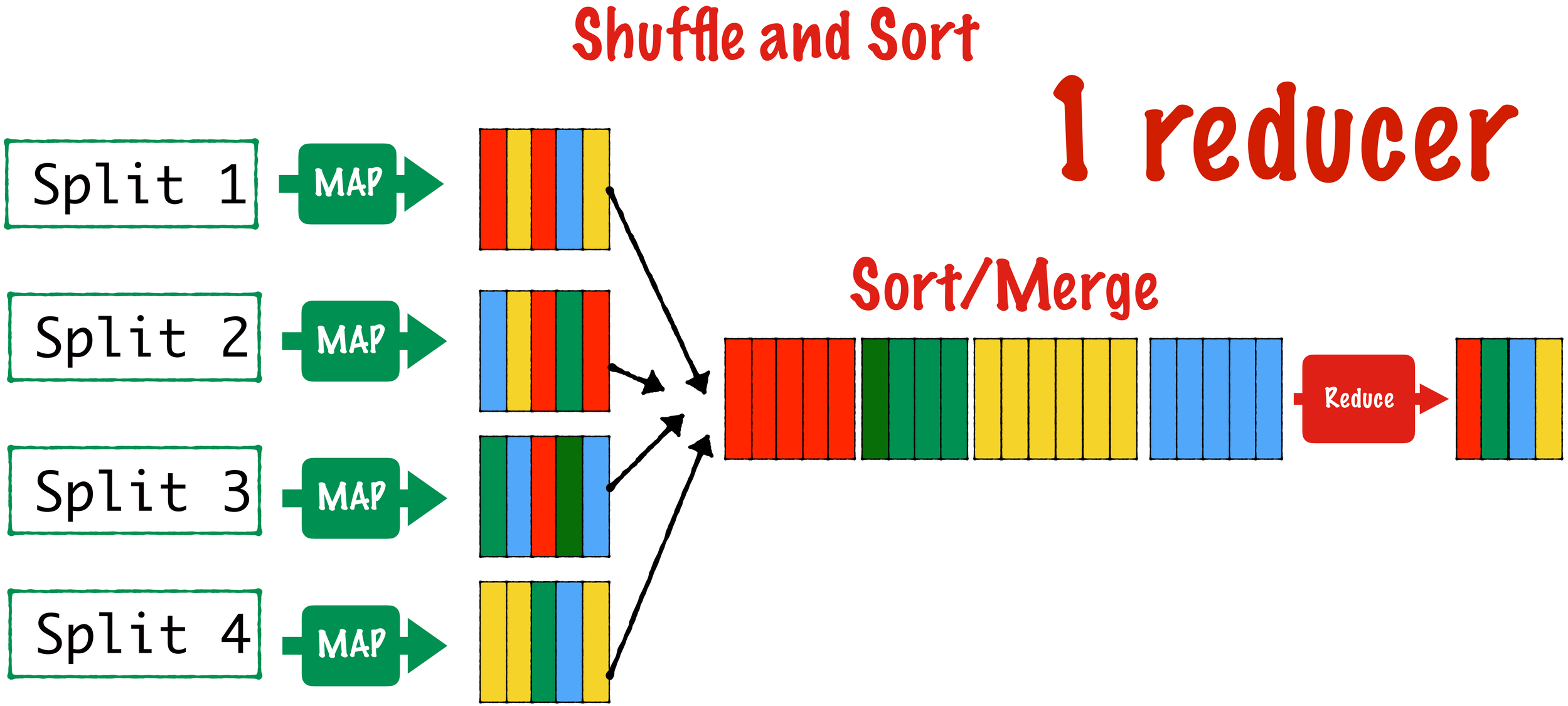
Each block here represents a <Key, Value> pair

The color represents the key

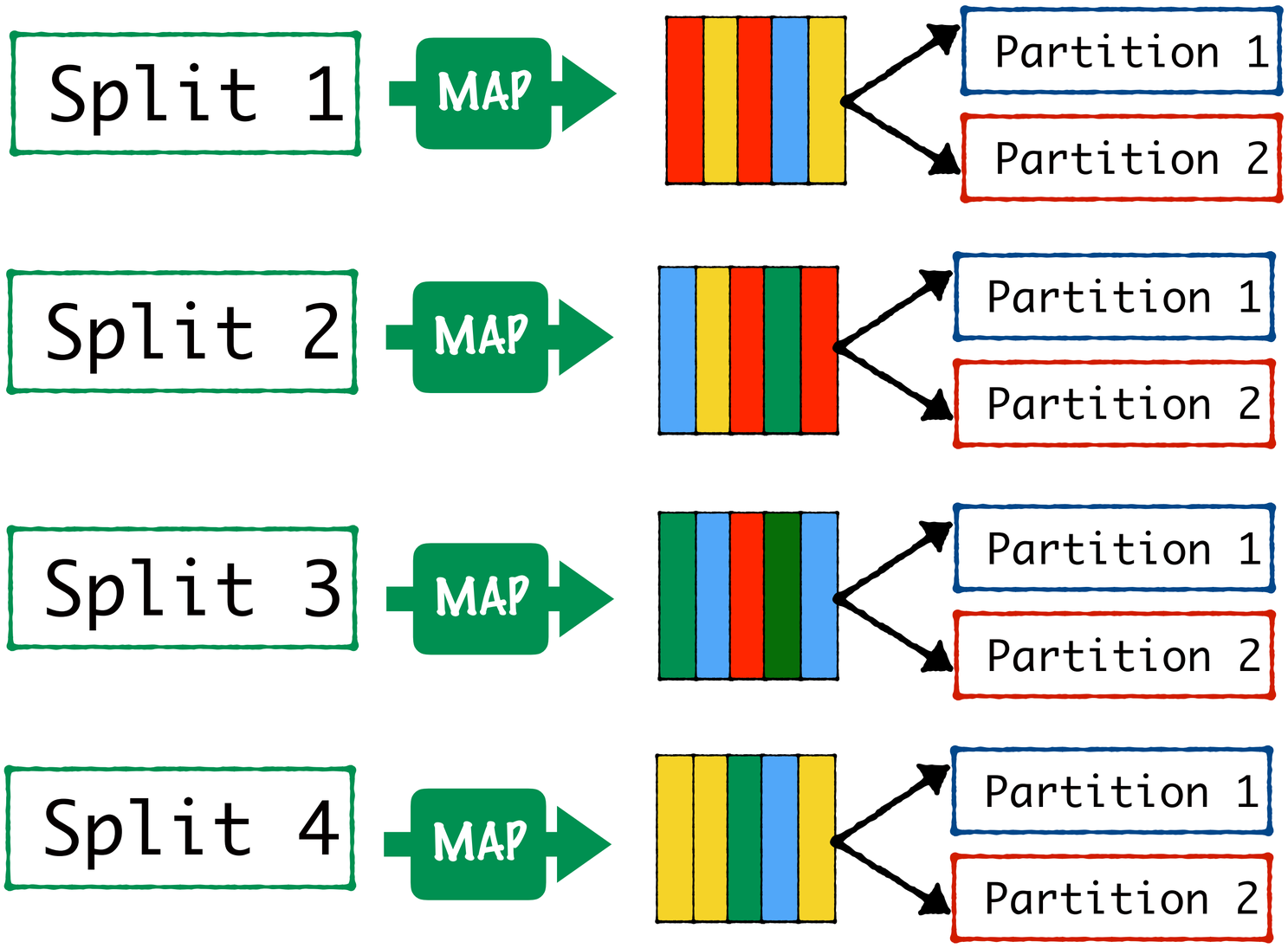
Shuffle and Sort



When there is only 1 reducer, these are simply sorted by key and then reduced

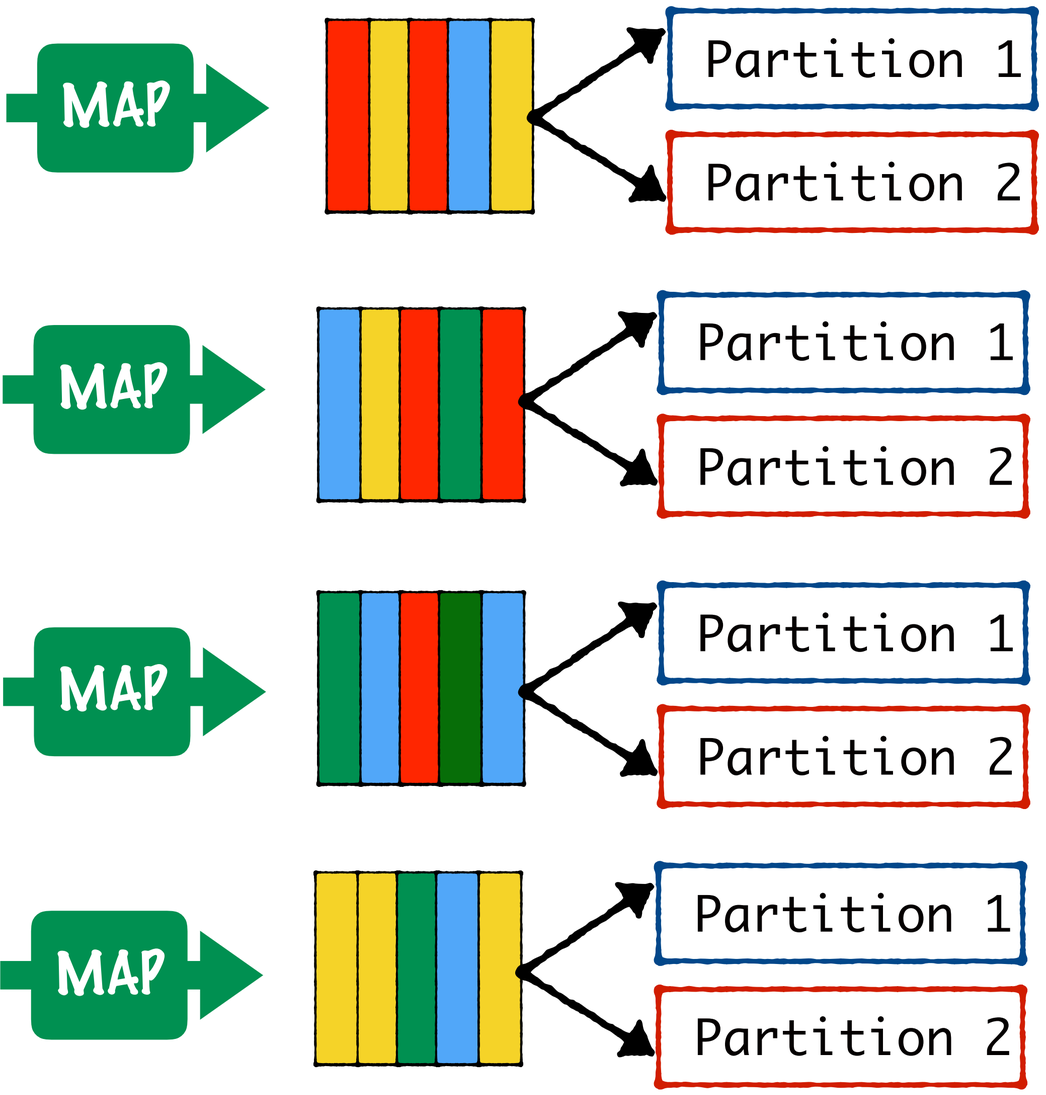

Shuffle and Sort
1 reducer
Split 1
Split 2
Split 3
Split 4
MAP
MAP
MAP
MAP
Sort/Merge
Reduce

Shuffle and Sort



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

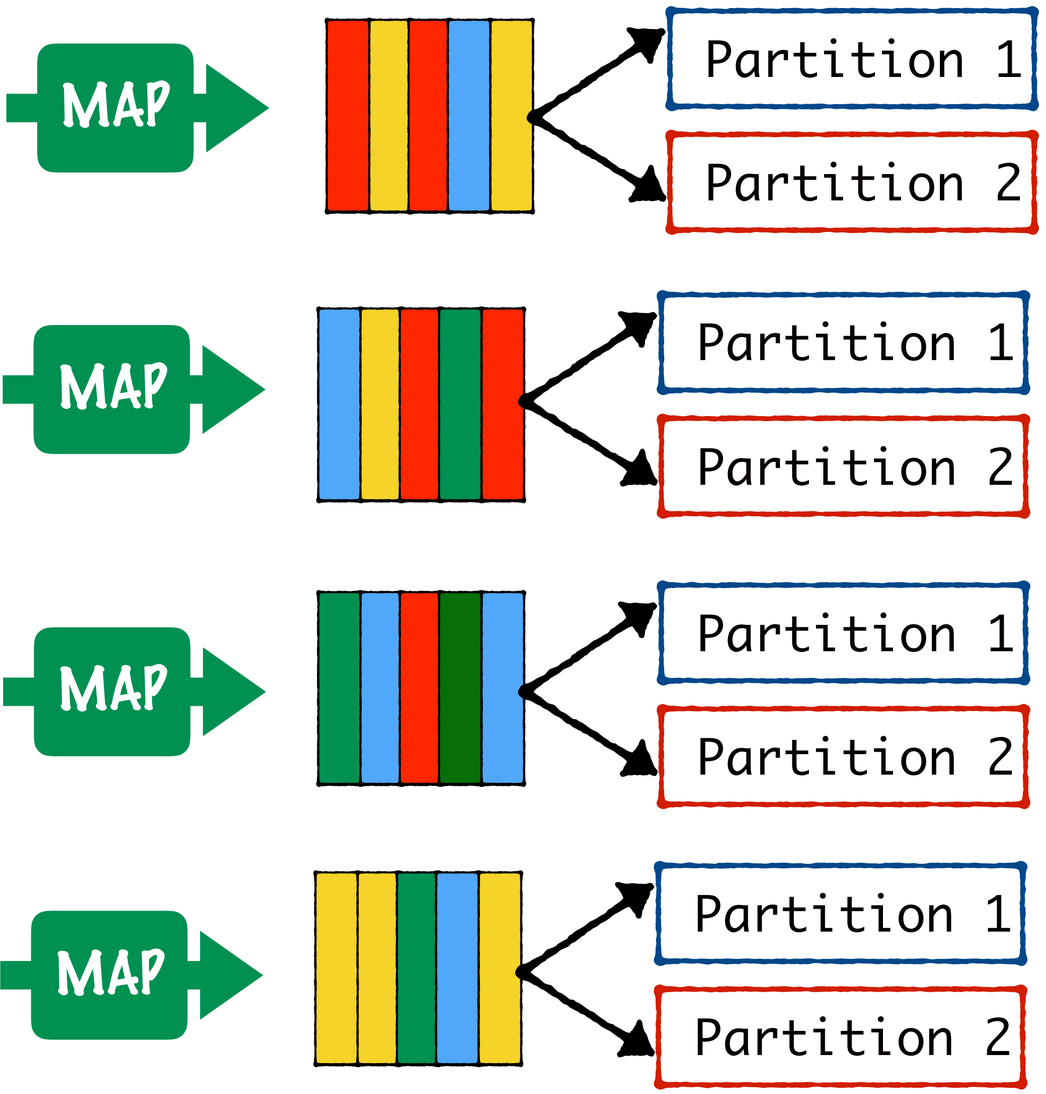
Shuffle and Sort



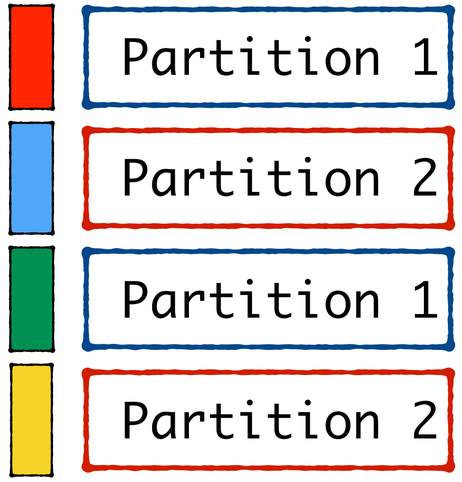
The map output is assigned to a partition based on the key

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

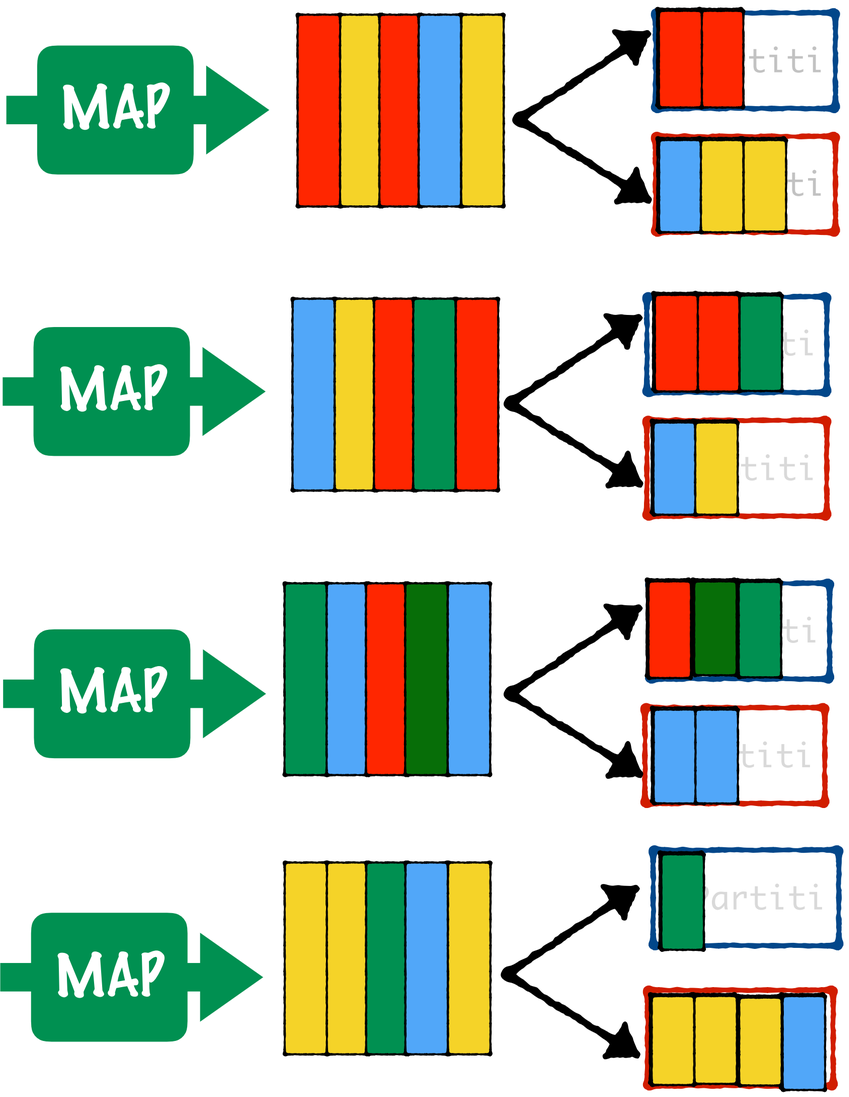
Shuffle and Sort



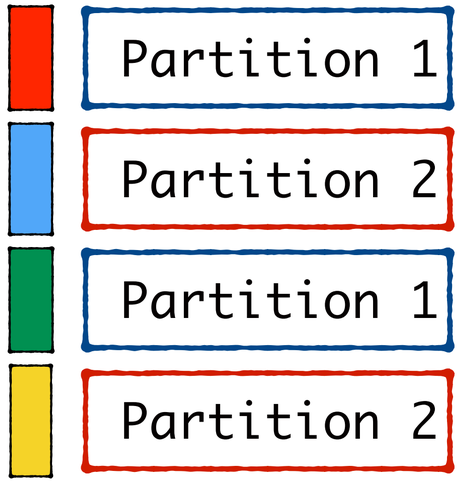
All values with the same key must be assigned to the same partition

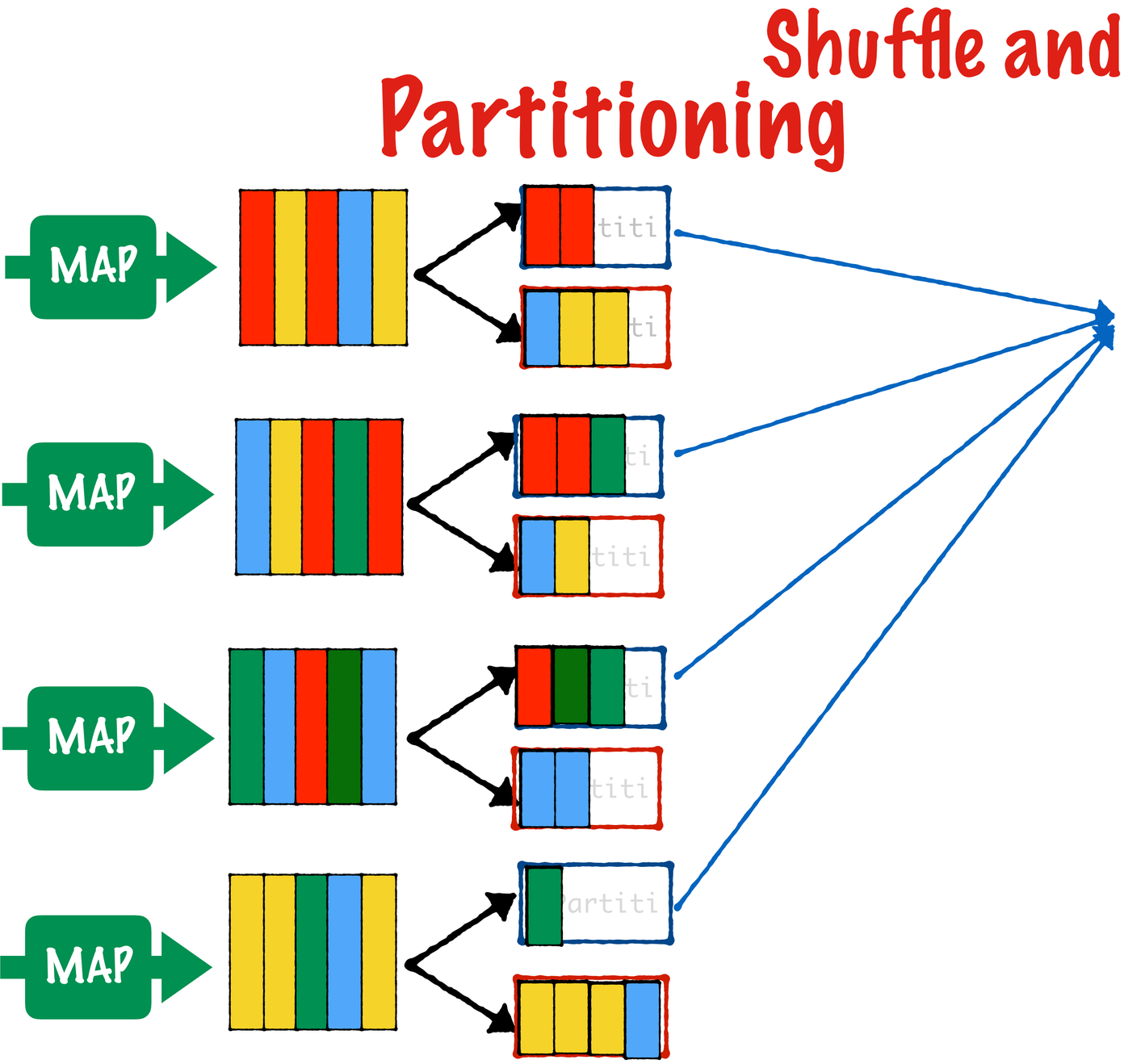


Shuffle and Sort

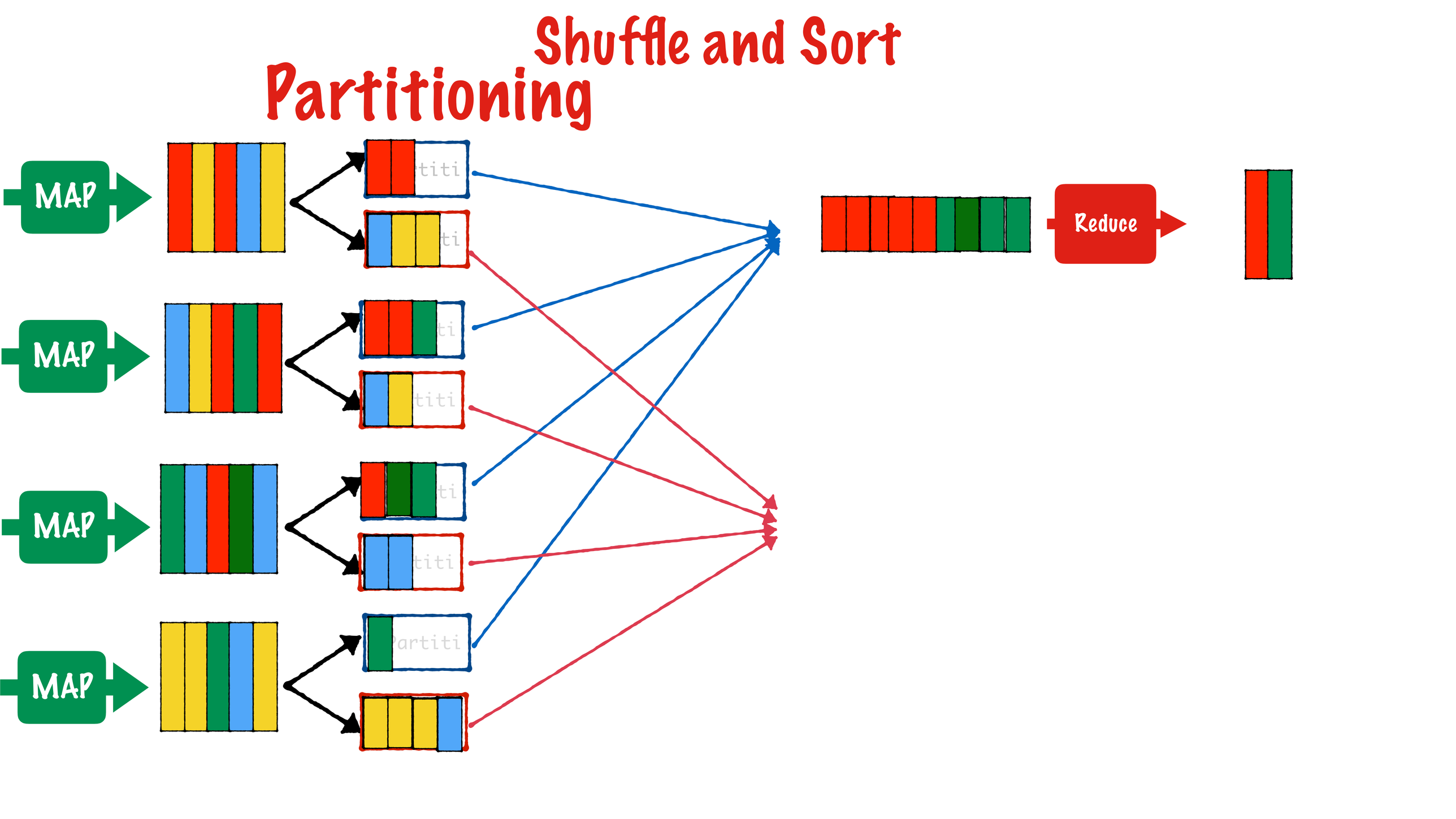


All values with the same key must be assigned to the same partition

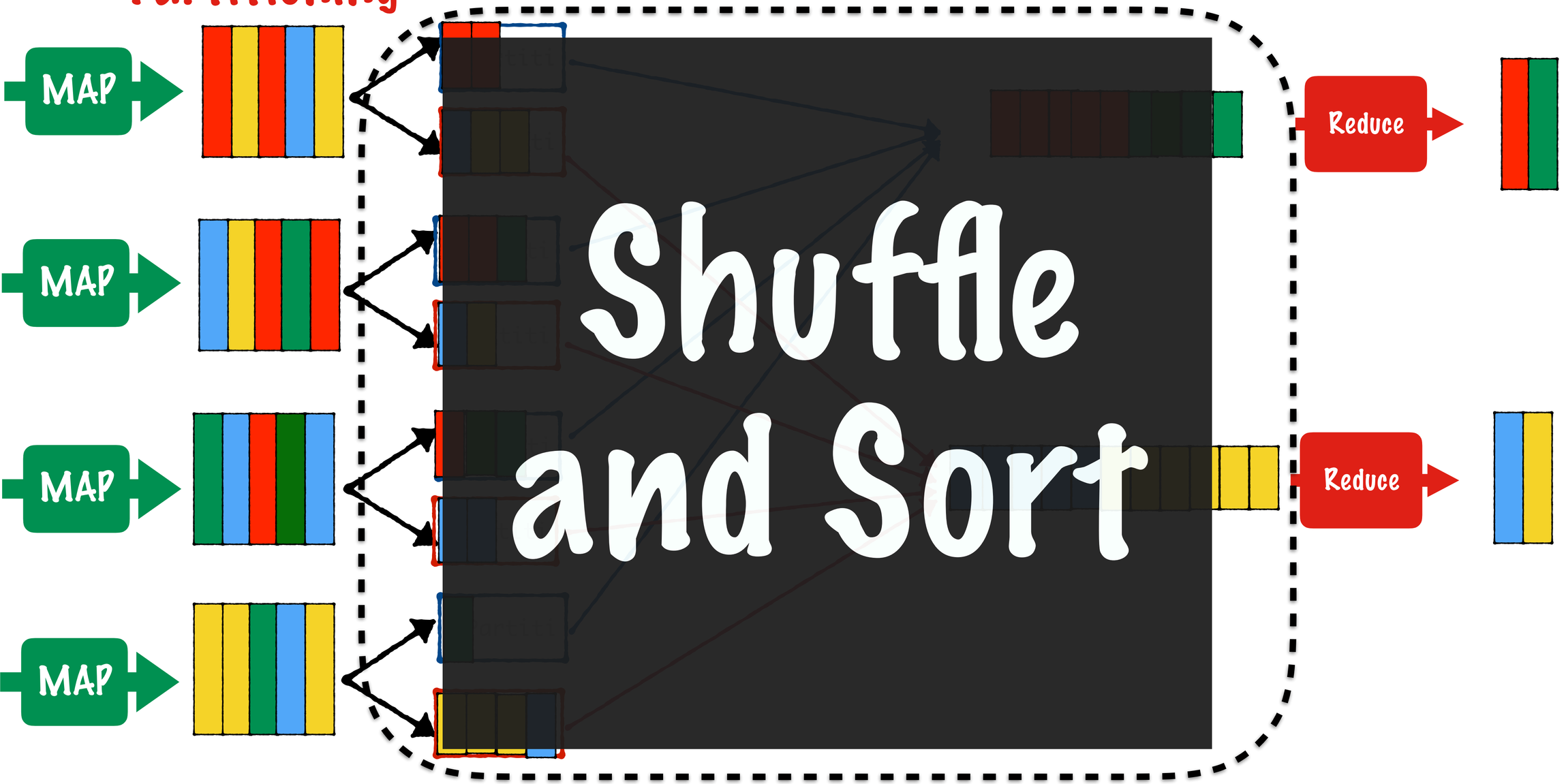


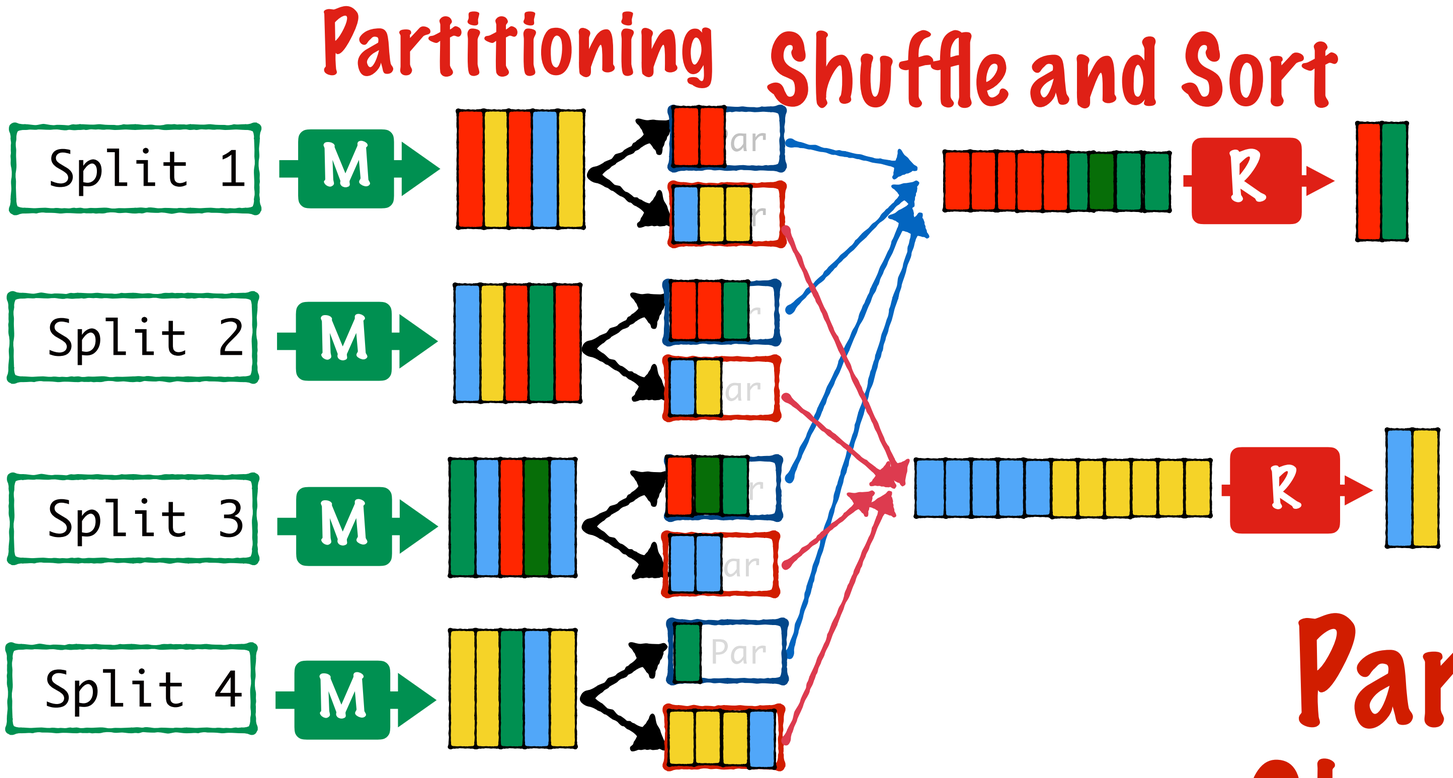


The data in each partition is sent to 1 reducer

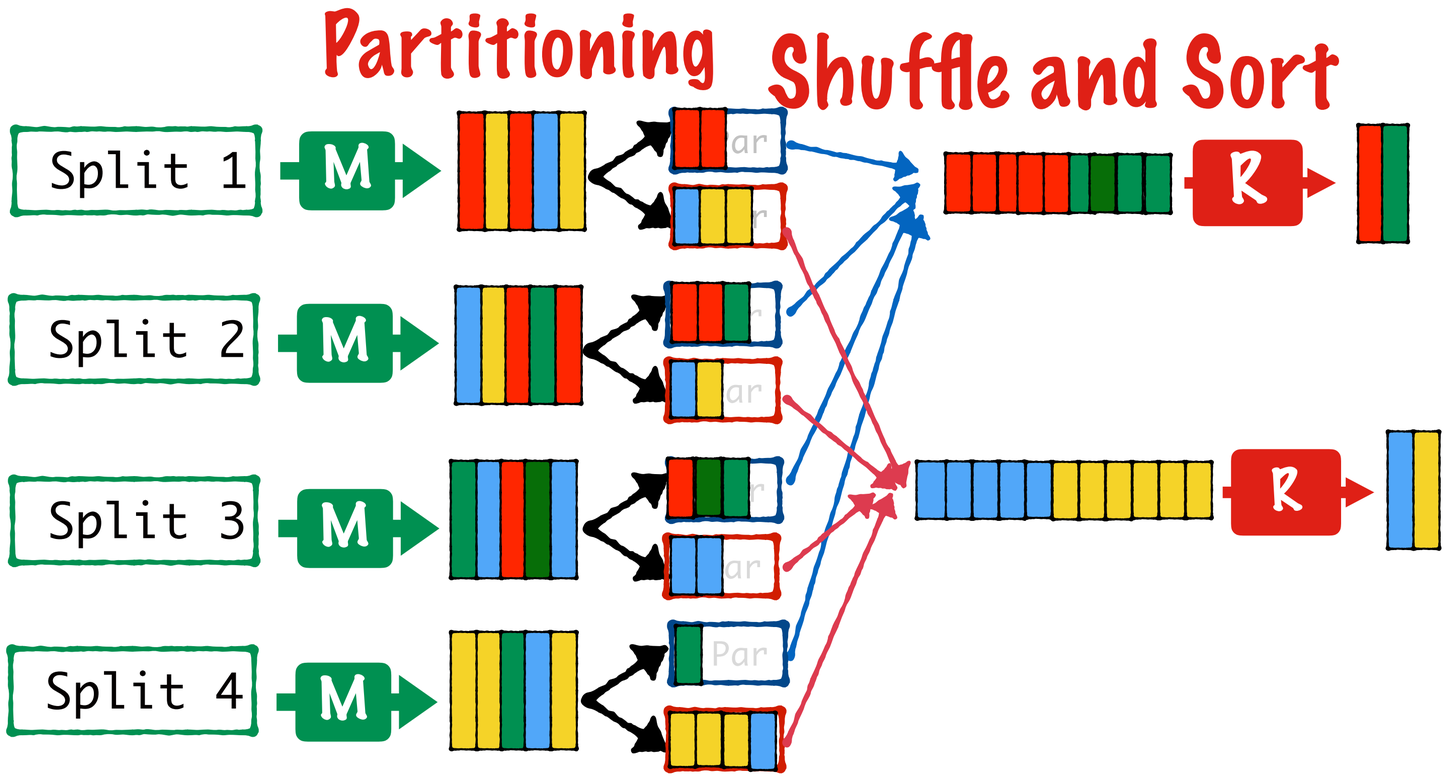

Shuffle and Sort
Partitioning
MAP
MAP
MAP
MAP
Reduce

Shuffle and Sort

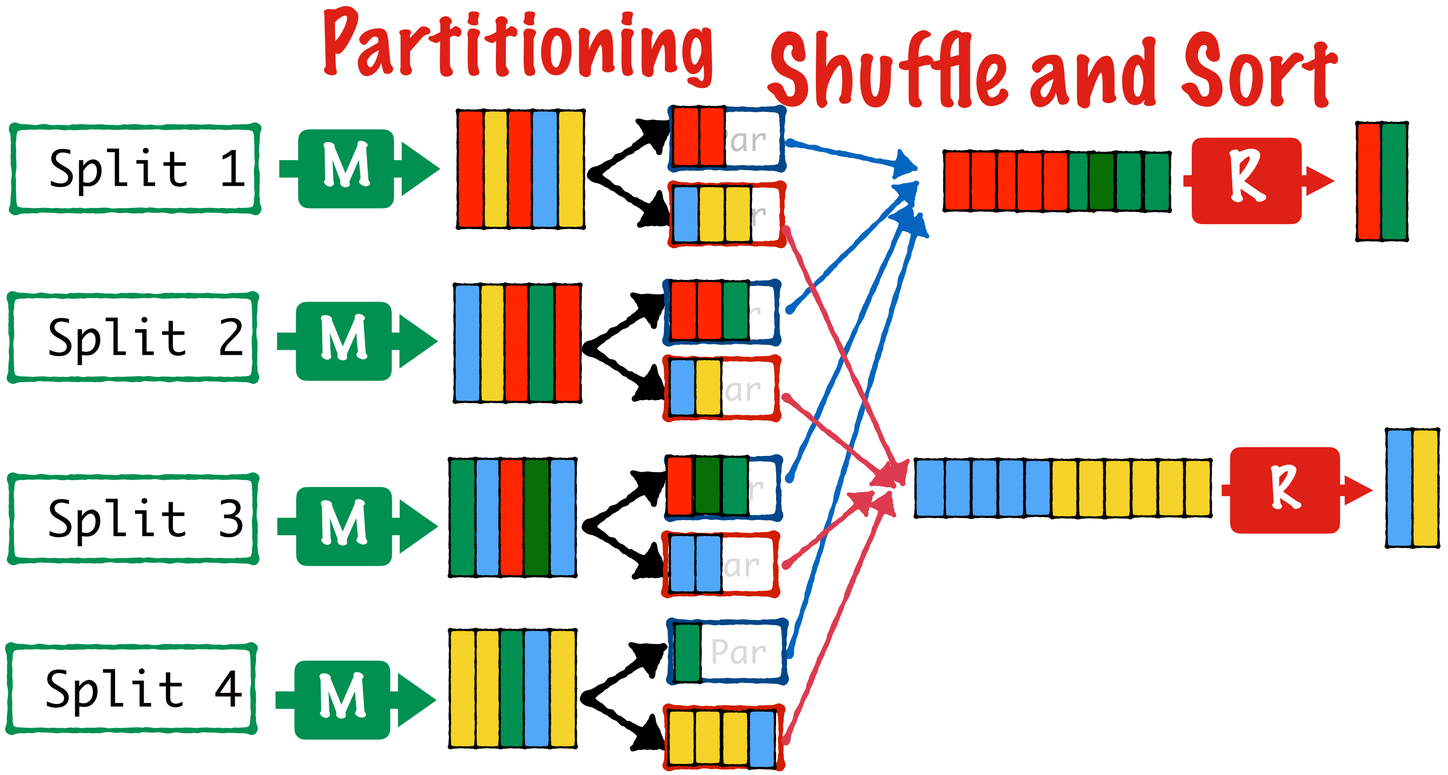




Partitioning and **Shuffle/Sort** are 2 extra operations that happen for MR jobs with multiple reducers

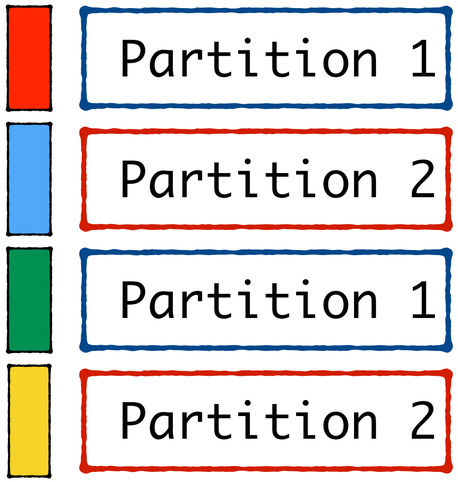


Number of partitions = Number of reducers



Hadoop takes care of mapping keys to partitions

Hadoop takes care of mapping keys to partitions



The partition number of a key depends on its hash value

The partition number of a key depends on its hash value

The default partitioning logic is implemented in the getPartition() method of HashPartitioner

```
public class HashPartitioner<K, V> extends Partitioner<K, V> {
    public HashPartitioner() {
    }

    public int getPartition(K key, V value, int numReduceTasks)
{
        return (key.hashCode() & 2147483647) % numReduceTasks;
    }
}
```

The default partitioning logic is implemented in the getPartition() method of HashPartitioner

```
public class HashPartitioner<K, V> extends Partitioner<K, V> {
    public HashPartitioner() {

    }
public int getPartition(K key, V value, int numReduceTasks)
{
    return (key.hashCode() & 2147483647) % numReduceTasks;
    }
}
```

Each key is assigned a unique Partition Id based on this logic

```
public class HashPartitioner<K, V>
extends Partitioner<K, V> {

    public HashPartitioner() {
    }

    public int getPartition(K key, V value, int numReduceTasks) {
        return (key.hashCode() & 2147483647) % numReduceTasks;
    }
}
```

You can write your own Partitioning logic by extending the Partitioner Class and implementing the getPartition() method

Partitioning logic should be such that keys are evenly distributed among the partitions

HashPartitioner does a pretty good job of this

MapReduce in Python with the Hadoop Streaming API

Hadoop Streaming API

Since Hadoop is written in Java, it's natural to write MapReduce programs in Java

Hadoop Streaming API

But, you can implement the map() and reduce() functions in ANY LANGUAGE

Hadoop Streaming API

Hadoop provides the Streaming API for exactly this purpose

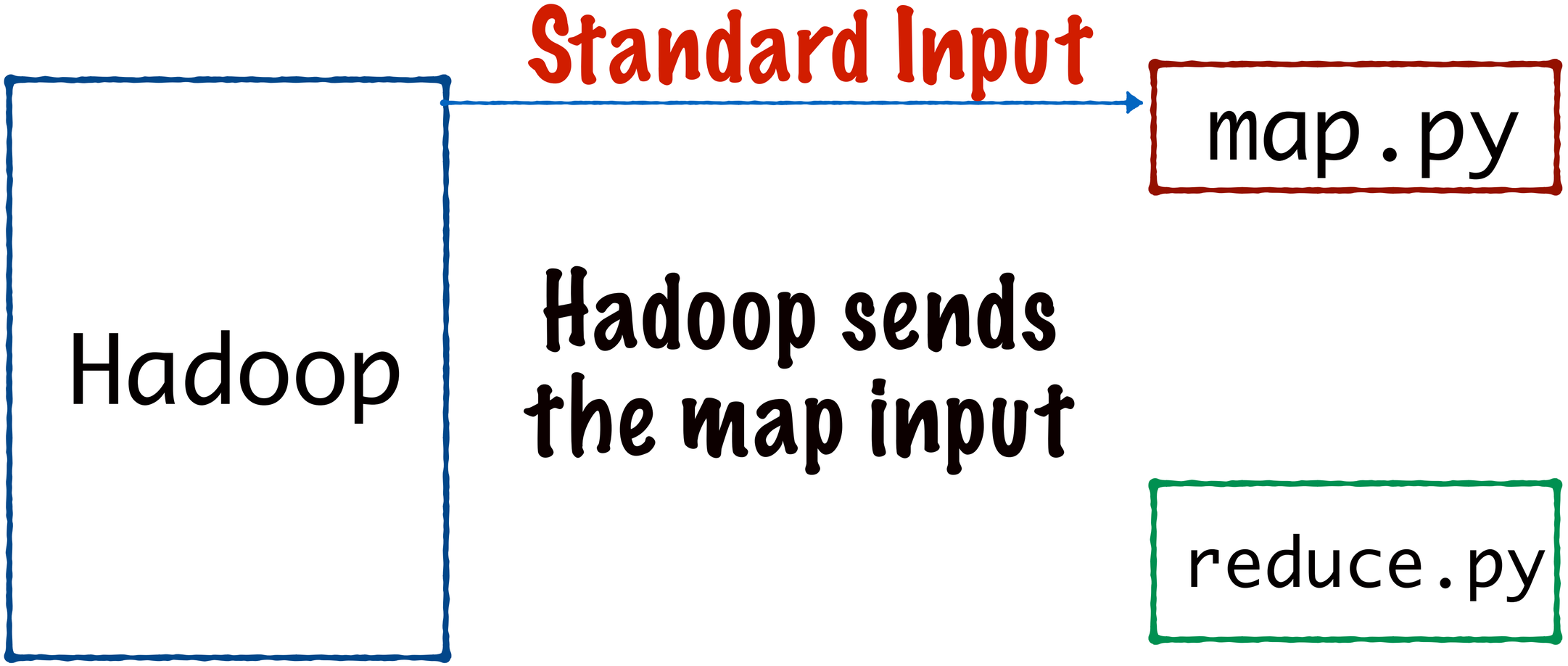
Hadoop Streaming API

The Streaming API uses Standard Input/Output to communicate with your program

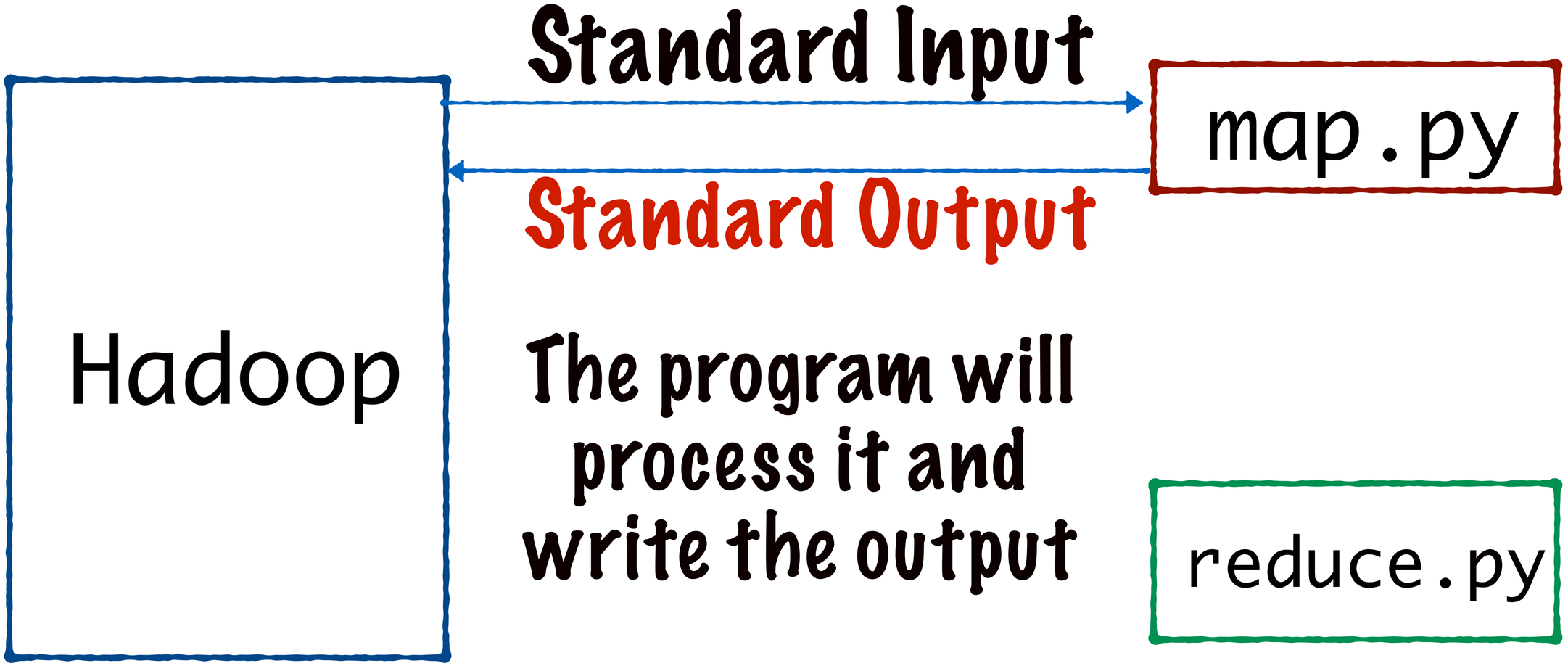
Hadoop Streaming API

Let's say we implemented the map and reduce functions in Python

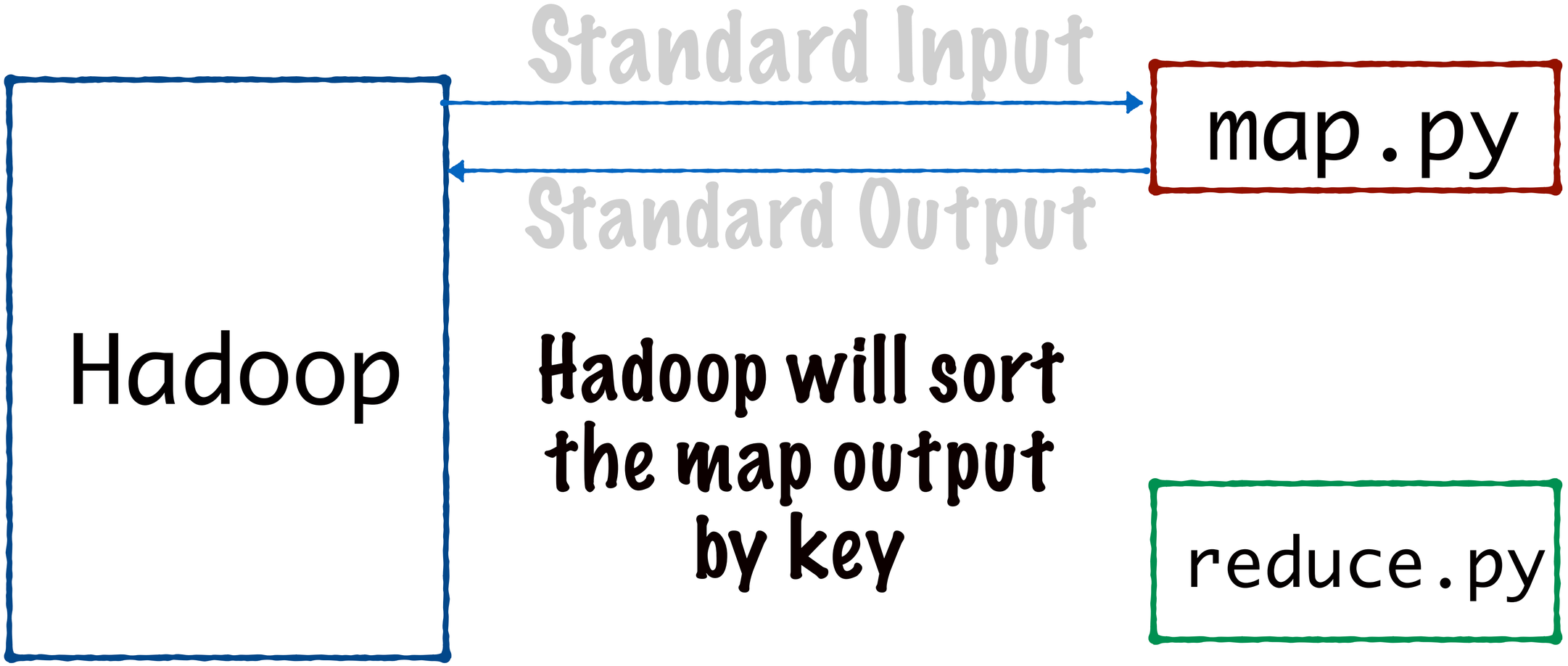
Hadoop Streaming API



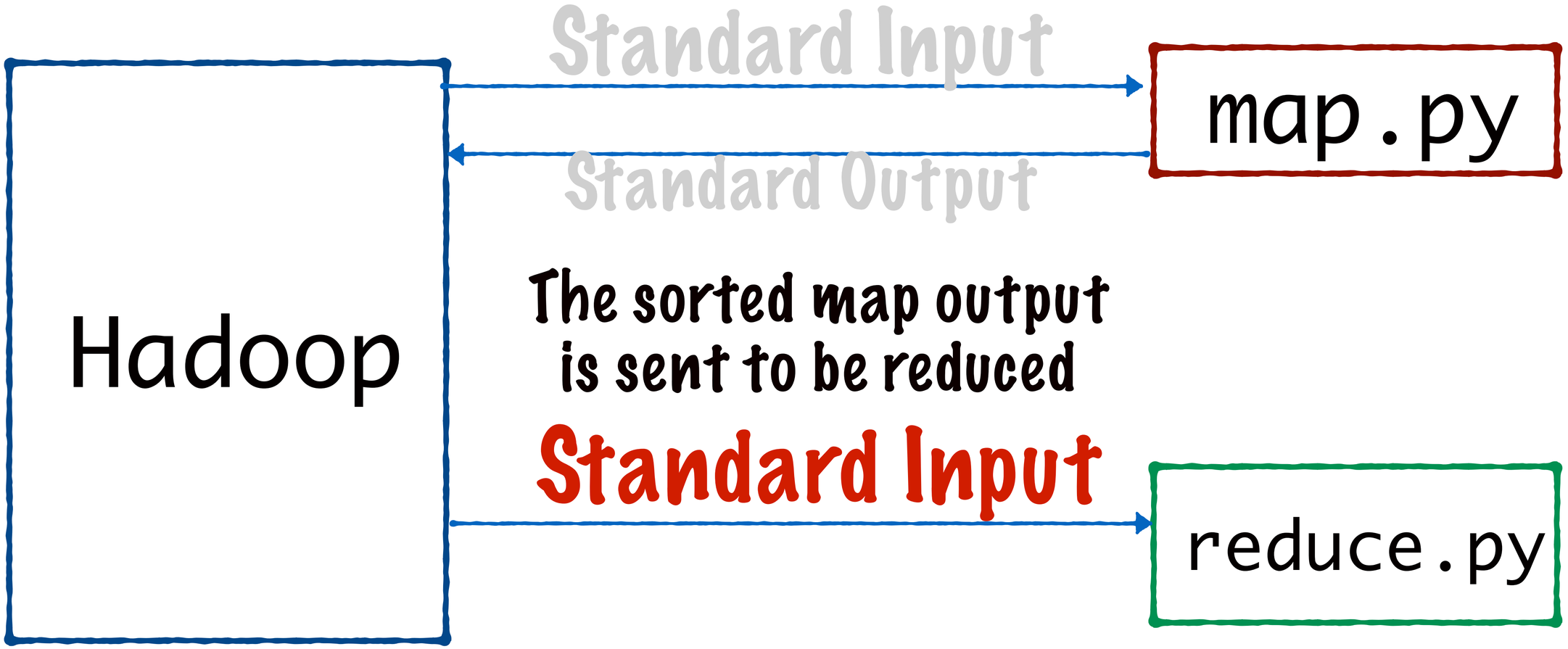
Hadoop Streaming API



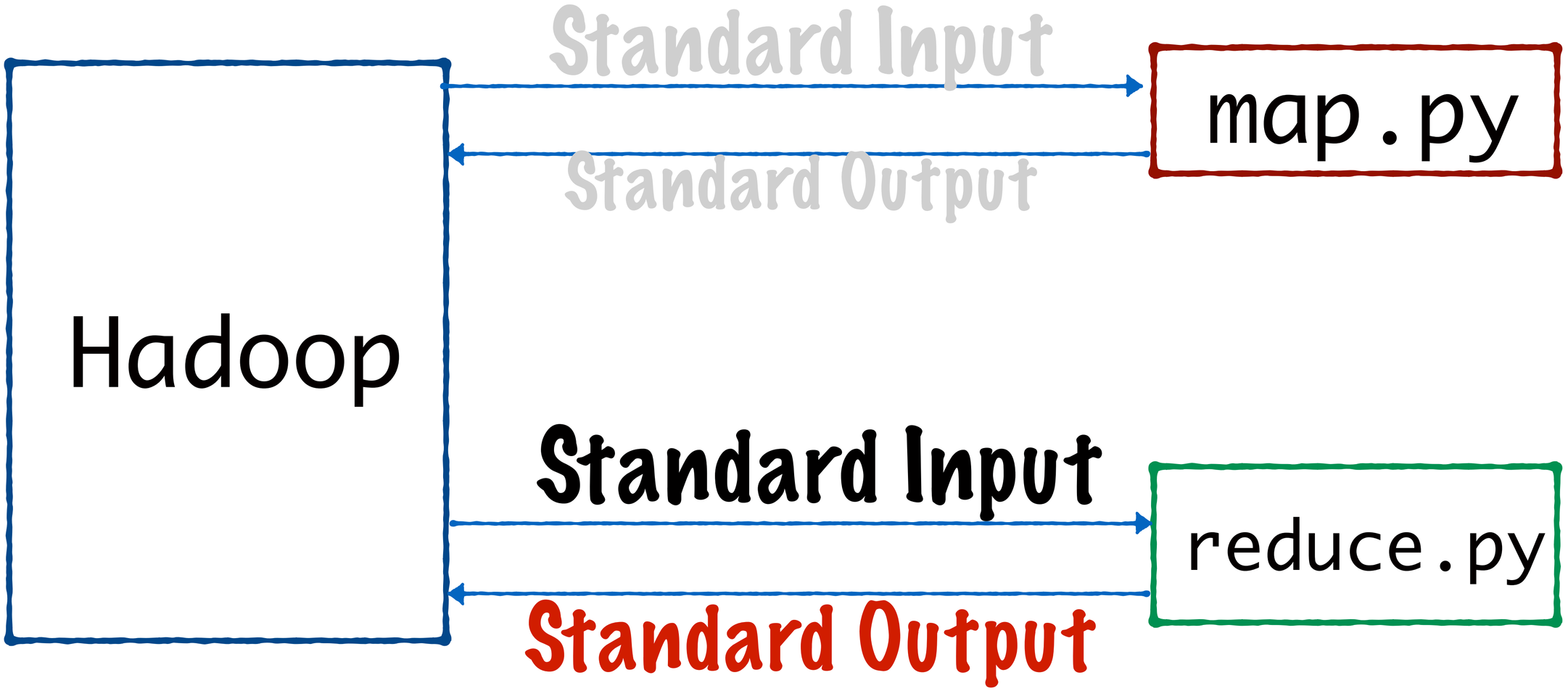
Hadoop Streaming API



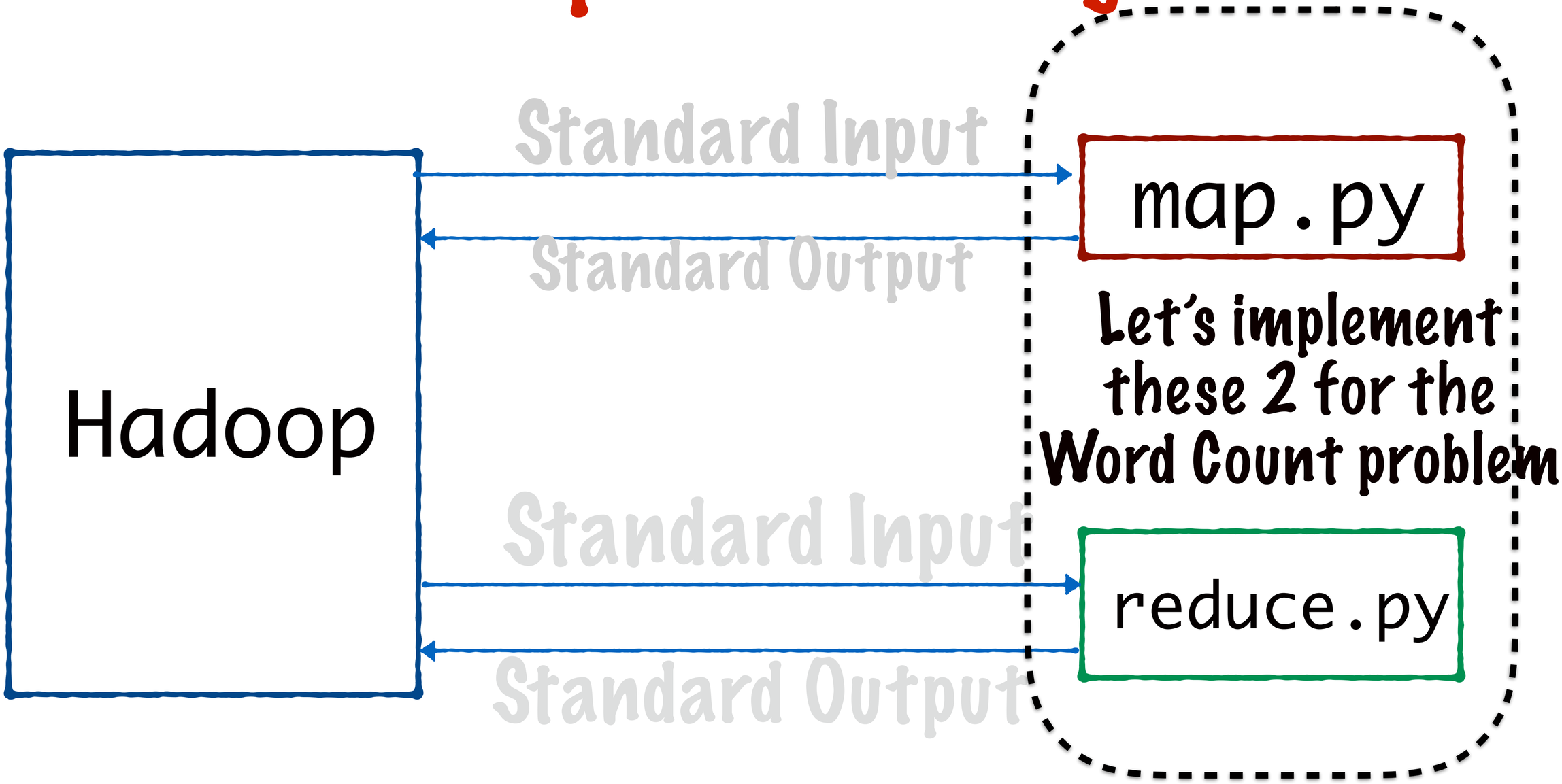
Hadoop Streaming API



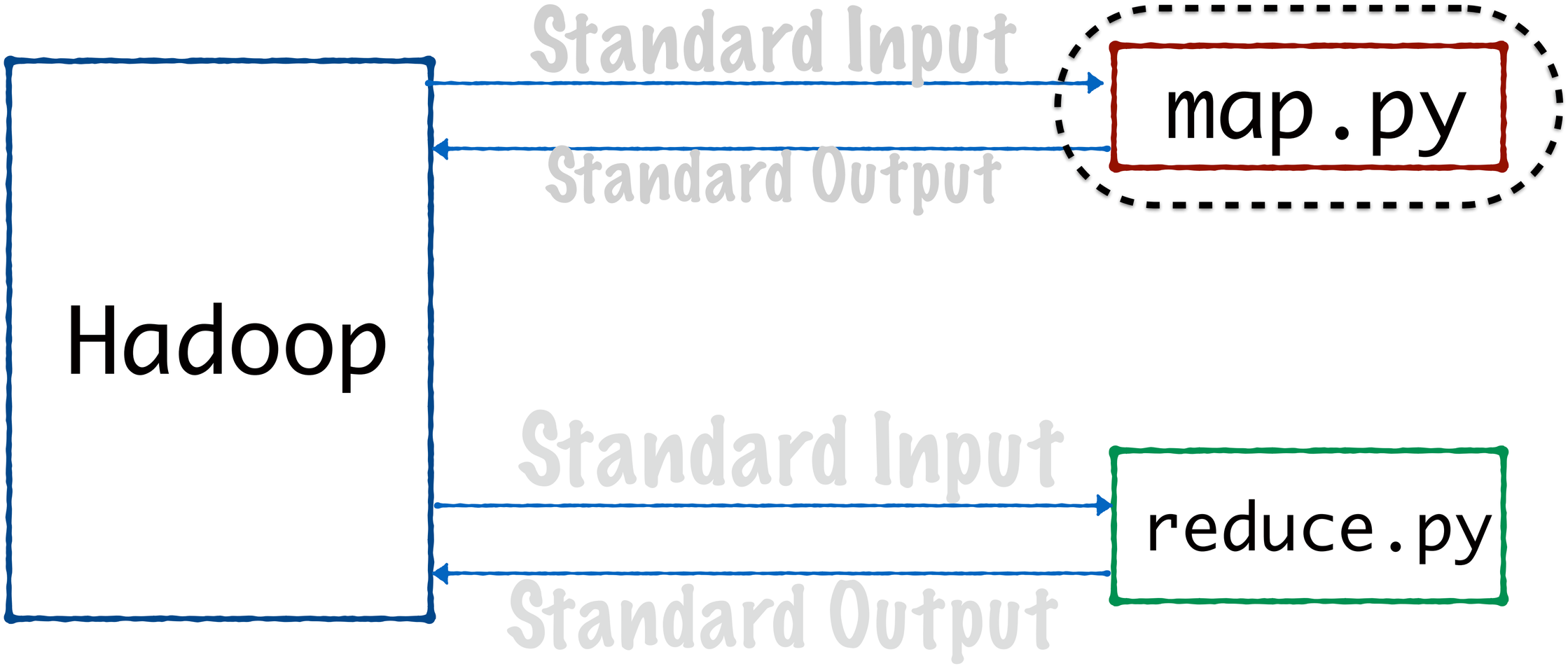
Hadoop Streaming API



Hadoop Streaming API



Hadoop Streaming API



Hadoop Streaming API

map.py

```
import sys
for line in sys.stdin:
    for word in line.split():
        print "%s\t%s" % (word,1)
```

Hadoop Streaming API

map.py

This represents the code for the step

```
import
for
```

<1,Hey diddle diddle> → map() →

<Hey,1>
<Diddle,1>
<Diddle,1>

<Linenum, Line of text>

<Word, 1>

Hadoop Streaming API

map.py

The lines of the text file are sent from Hadoop over Standard Input

```
import sys
for line in sys.stdin:
        for word in line.split():
        print "%s\t%s" % (word,1)
```

Hadoop Streaming API

map.py

For each word, we print (word,1) to Standard Output

```
import sys
for line in sys.stdin:
        for word in line.split():
            print "%s\t%s" % (word,1)
```

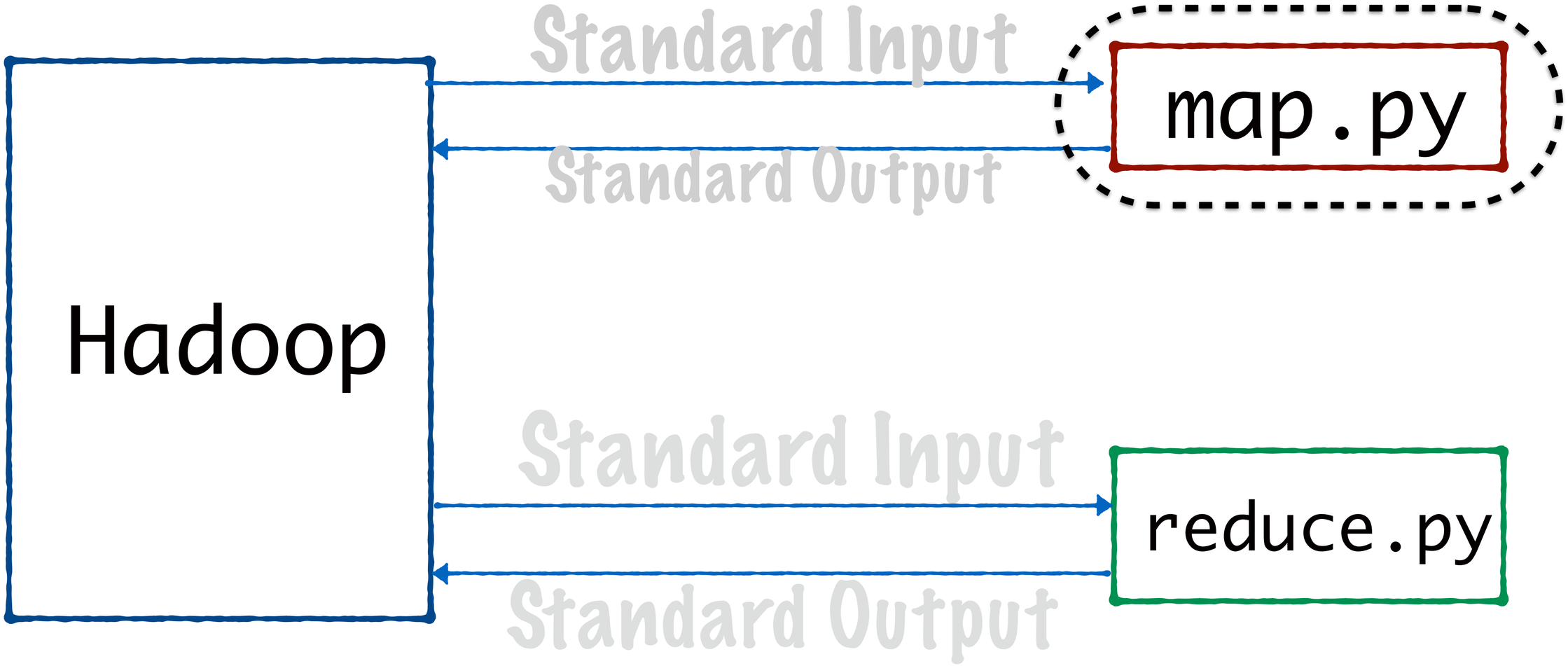
Hadoop Streaming API

map.py

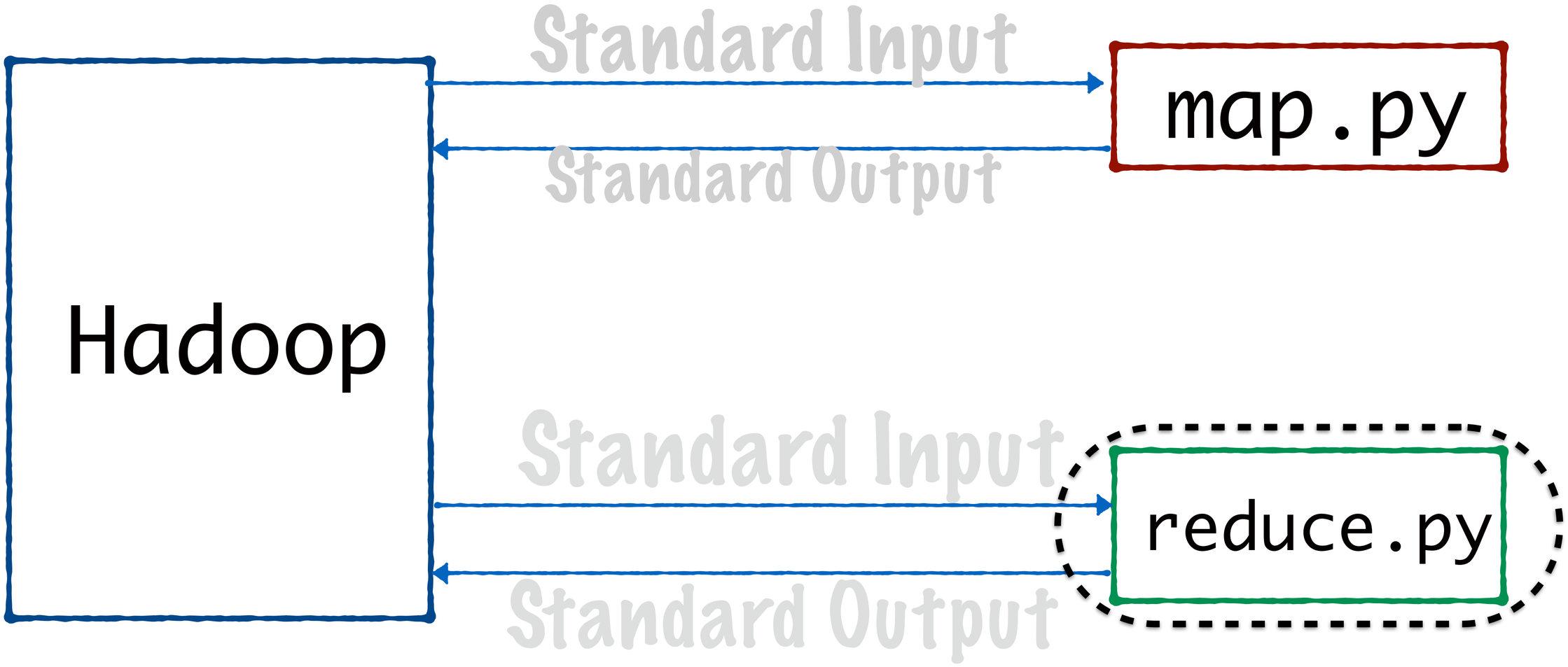
Hadoop expects the key and value to be separated by a tab

```
import sys
for line in sys.stdin:
        for word in line.split():
            print "%s\t%s" % (word,1)
```

Hadoop Streaming API



Hadoop Streaming API



Hadoop Streaming API

reduce.py

The map output is sorted and sent to the reduce function

Unlike in Java, the sorted output is not merged

Hadoop Streaming API

reduce.py

```
import sys

(last_key,count) = (None, 0)
for line in sys.stdin:
    (key, value) = line.strip().split("\t")
    if last_key and last_key!=key:
        print "%s\t%s" % (last_key,count)
        (last_key,count) = (key,int(value))
    else:
        (last_key,count) = (key, count + int(value))
```

Hadoop Streaming API

reduce.py

This function represents the code for the step

<Diddle,1>
<Diddle,1>
<The,1>
..
.

<Diddle,2>

<word, 1>

<word, count>

Hadoop Streaming API

reduce.py

Since the map output is coming in like a stream, we need to keep track of what the last key was

```
import sys

(last_key,count) = (None, 0)
for line in sys.stdin:
    (key, value) = line.strip().split("\t")
    if last_key and last_key!=key:
        print "%s\t%s" % (last_key,count)
        (last_key,count) = (key,int(value))
    else:
        (last_key,count) = (key, count + int(value))
```

Hadoop Streaming API

reduce.py

As long as we see the same key, we accumulate the count, else we start over

```
import sys

(last_key,count) = (None, 0)
for line in sys.stdin:
    (key, value) = line.strip().split("\t")
    if last_key and last_key!=key:
        print "%s\t%s" % (last_key,count)
        (last_key,count) = (key,int(value))
    else:
        (last_key,count) = (key, count + int(value))
```

Hadoop Streaming API

reduce.py

In the beginning, we initialize a value for the last key

```
import sys

(last_key,count) = (None, 0)
for line in sys.stdin:
    (key, value) = line.strip().split("\t")
    if last_key and last_key!=key:
        print "%s\t%s" % (last_key,count)
        (last_key,count) = (key,int(value))
    else:
        (last_key,count) = (key, count + int(value))
```

Hadoop Streaming API

reduce.py

Each pair is in 1 line of the Standard Input, key and value separated by tab

```
import sys

(last_key,count) = (None, 0)
for line in sys.stdin:
    (key, value) = line.strip().split("\t")
    if last_key and last_key!=key:
        print "%s\t%s" % (last_key,count)
        (last_key,count) = (key,int(value))
    else:
        (last_key,count) = (key, count + int(value))
```

Hadoop Streaming API

reduce.py

If we see a new key, we print the count for the last key and start over

```
import sys

(last_key,count) = (None, 0)
for line in sys.stdin:
    (key, value) = line.strip().split("\t")
    if last_key and last_key!=key:
        print "%s\t%s" % (last_key,count)
        (last_key,count) = (key,int(value))
    else:
        (last_key,count) = (key, count + int(value))
```

Hadoop Streaming API

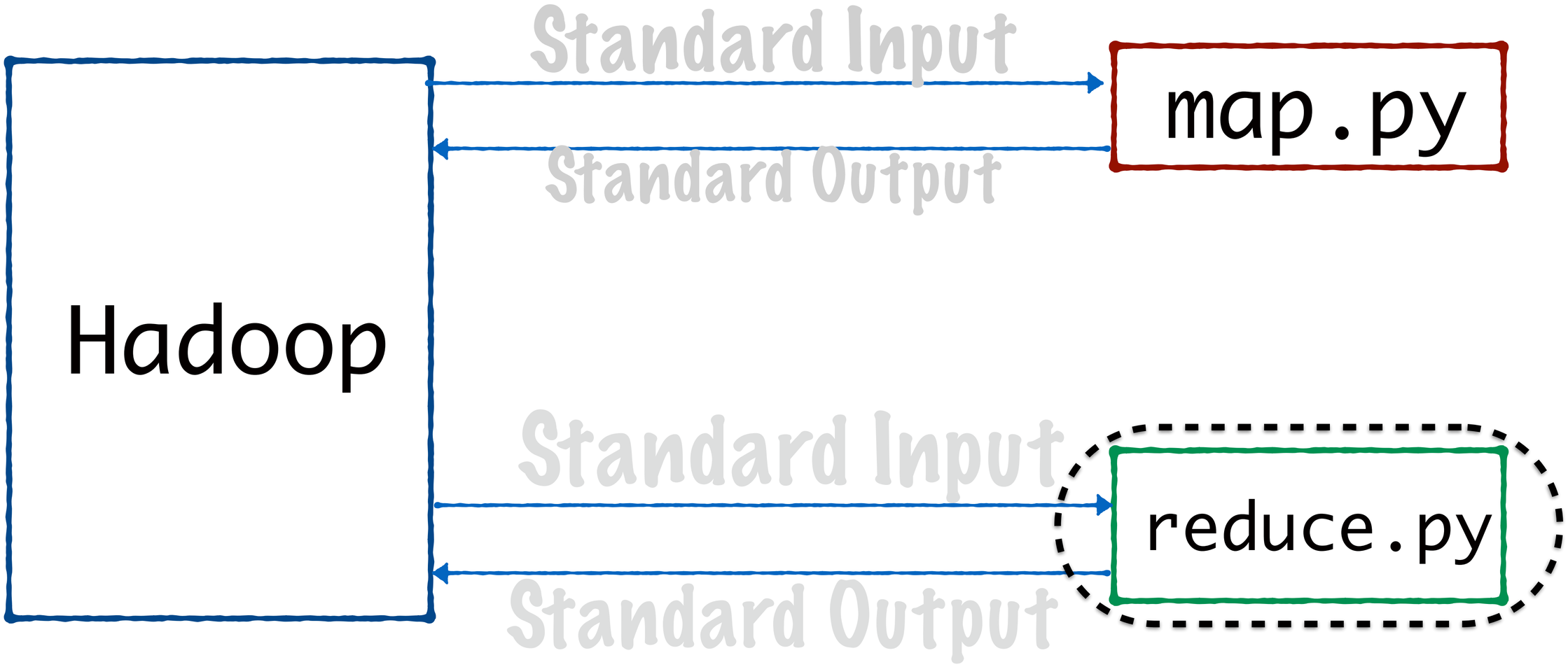
reduce.py

If it's still the same key, then we add to the word count

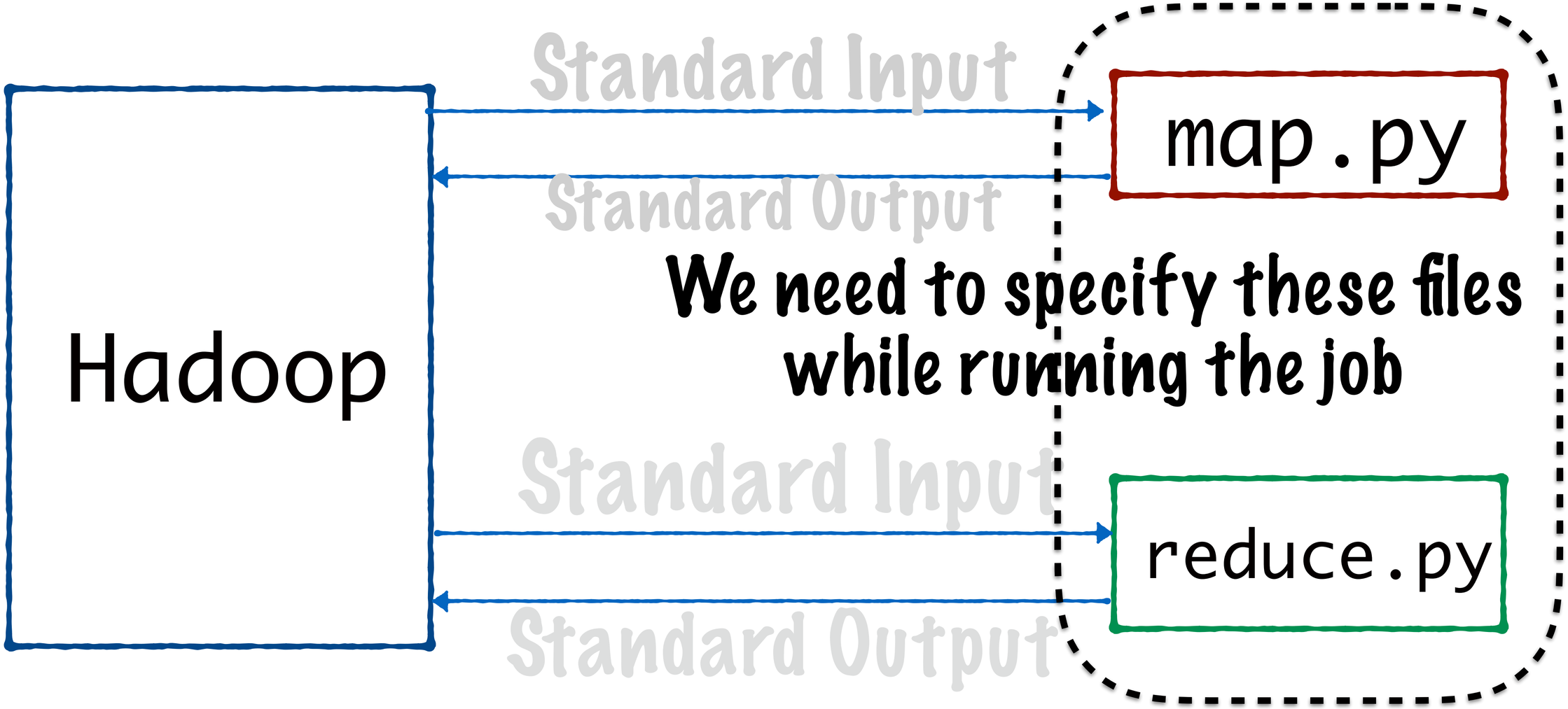
```
import sys

(last_key,count) = (None, 0)
for line in sys.stdin:
    (key, value) = line.strip().split("\t")
    if last_key and last_key!=key:
        print "%s\t%s" % (last_key,count)
        (last_key,count) = (key,int(value))
    else:
        (last_key,count) = (key, count + int(value))
```

Hadoop Streaming API



Hadoop Streaming API



Hadoop Streaming API

Running a job with Hadoop Streaming

```
$ hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar \
-files <mapCodePath>,<reduceCodePath> \
-input <inputFilePath> \
-output <outputFilePath> \
-mapper "python <mapCodePath>" \
-combiner "python <reduceCodePath>" \
-reducer "python <reduceCodePath>"
```

The jars for Hadoop Streaming

Hadoop Streaming API

Running a job with Hadoop Streaming

```
$ hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar \
-files <mapCodePath>,<reduceCodePath> \
-input <inputFilePath> \
-output <outputFilePath> \
-mapper "python <mapCodePath>" \
-combiner "python <reduceCodePath>" \
-reducer "python <reduceCodePath>"
```

The file paths for map.py and reduce.py

These files will be copied over to all the nodes where computation is happening

Hadoop Streaming API

Running a job with Hadoop Streaming

```
$ hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar \
-files <mapCodePath>,<reduceCodePath> \
-input <inputFilePath> \
-output <outputFilePath> \
-mapper "python <mapCodePath>" \
-combiner "python <reduceCodePath>" \
-reducer "python <reduceCodePath>"
```

The input text file path and output directory

Hadoop Streaming API

Running a job with Hadoop Streaming

```
$ hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar \
-files <mapCodePath>,<reduceCodePath> \
-input <inputFilePath> \
-output <outputFilePath> \
-mapper "python <mapCodePath>" \
-combiner "python <reduceCodePath>" \
-reducer "python <reduceCodePath>"
```

We specify the file in which the mapper is implemented

Hadoop Streaming API

Running a job with Hadoop Streaming

```
$ hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar \
-files <mapCodePath>,<reduceCodePath> \
-input <inputFilePath> \
-output <outputFilePath> \
-mapper "python <mapCodePath>" \
-combiner "python <reduceCodePath>" \
-reducer "python <reduceCodePath>"
```

We specify the path for the files where combiner and reducer functions are implemented

Hadoop Streaming API

Running a job with Hadoop Streaming

```
$ hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar \
-files <mapCodePath>,<reduceCodePath> \
-input <inputFilePath> \
-output <outputFilePath> \
-mapper "python <mapCodePath>" \
-combiner "python <reduceCodePath>" \
-reducer "python <reduceCodePath>"
```

For Word count problem, the combiner is the same as the reducer