Big Data Processing, 2014/15

Lecture 6: MapReduce - behind the scenes continued (a very mixed bag)

Claudia Hauff (Web Information Systems)
ti2736b-ewi@tudelft.nl

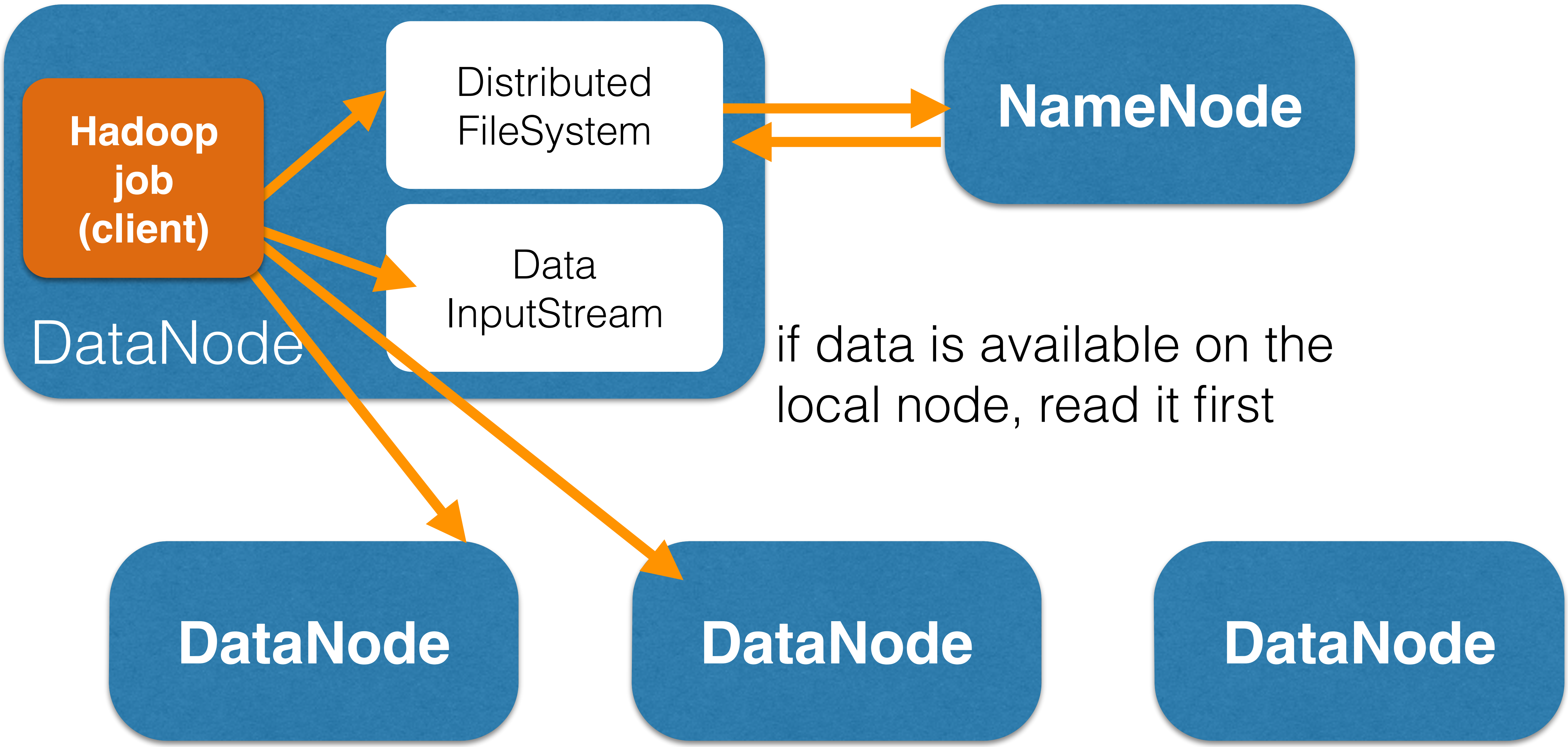
Course content

- Introduction
- Data streams 1 & 2
- The MapReduce paradigm
- **Looking behind the scenes of MapReduce:** HDFS & **Scheduling**
- Algorithm design for MapReduce
- A high-level language for MapReduce: Pig 1 & 2
- MapReduce is not a database, but HBase nearly is
- Lets iterate a bit: Graph algorithms & Giraph
- How does all of this work together? ZooKeeper/Yarn

Learning objectives

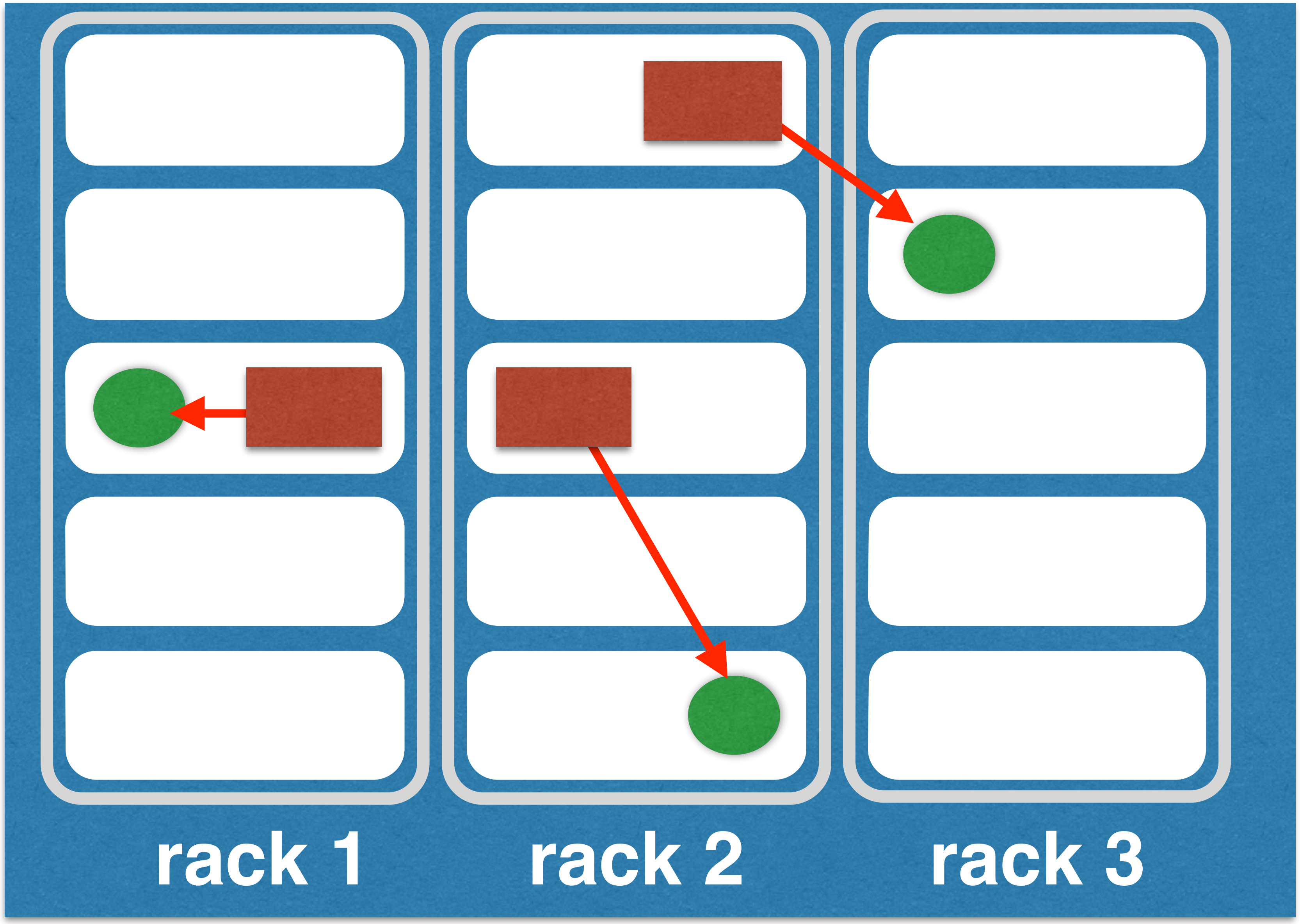
- **Exploit** Hadoop’s Counters and setup/cleanup efficiently
- **Explain** how Hadoop addresses the problem of job scheduling
- **Explain** Hadoop’s shuffle & sort phase and **use** that knowledge to improve your Hadoop code
- **Implement** strategies for efficient data input

Question: what happens in each stage of the “read operation” ?

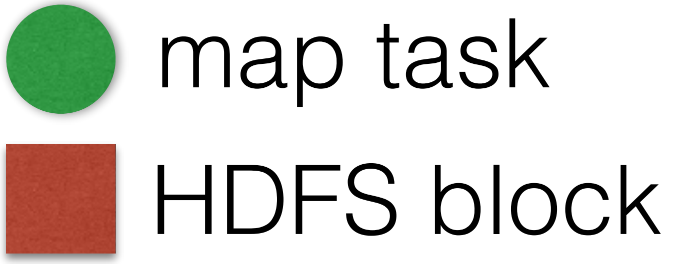


if data is available on the local node, read it first

Question: if the 3 map tasks started at the same time, in what order will they finish?



Data center



Question: which of the following GFS components reside on a chunkserver?

file	block	metadata
chunk	checksum	namespace
master	heartbeat	file permissions

Hadoop Programming Revisited: setup and cleanup

Setup & cleanup

Programmer "hints" the number of mappers to use

- One `MAPPER` object for each map task
  - Associated with a sequence of key/value pairs (the "input split")
  - `map()` is called for each key/value pair by the execution framework

Programmer can set the number of reducers

- One `REDUCER` object for each reduce task
  - `reduce()` is called once per intermediate key
- `MAPPER/REDUCER` are Java objects -> allows side effects
  - Preserving state across multiple inputs
  - Initialise additional resources
  - Emit (intermediate) key/value pairs in one go

Setup

Setup useful for one-off operations:

- opening an SQL connection
- loading a dictionary
- etc.

WordCount - count only valid dictionary terms*

```java
public class MyMapper extends
    Mapper<Text, IntWritable, Text, IntWritable> {

 private Set<String> dictionary;//all valid words

 public void setup(Context context) throws IOException {
      dictionary = Sets.newHashSet();
      loadDictionary();//defin
 }

 public void map(Text key, IntWritable val, Context context)
                    throws IOException, InterruptedException {
      if(!dictionary.contains(key.toString())
        return;
      context.write(key, new IntWritable(1));
 }
}
```

Called once in the life cycle of a Mapper object: before any calls to `map()`

Called once for each key/value pair that appears in the input split

single reducer setting

Cleanup

*WordCount** - how many words start with the same letter?*

```
public class MyReducer extends
  Reducer<PairOfIntString, FloatWritable, NullWritable, Text> {
 private Map<Character, Integer> cache;

 public void setup(Context context) throws IOException {
      cache = Maps.newHashMap();
 }
 public void reduce(PairOfIntString key, Iterable<IntWritable>
                   values, Context context) throws
                   IOException, InterruptedException {
      char c = key.toString().charAt(0);
      for(IntWritable iw : values){
         //add iw to the current value of key c in cache
      }
 }
 public void cleanup(Context context) throws IOException,
                   InterruptedException {
      for (Character c : cache.keySet()) {
         context.write(new Text(c), new IntWritable(cache.get(c));
      }
 }
}
```

single reducer setting

Cleanup

*WordCount** - how many words start with the same letter?*

```
public class MyReducer extends
  Reducer<PairOfIntString, FloatWritable, NullWritable, Text> {
  private Map<Character, Integer> cache

  public void setup(Context context) th
        cache = Maps.newHashMap();
  }
  public void reduce(PairOfIntString key, Iterable<IntWritable>
                     values, Context con
                     IOException, Interr
        char c = key.toString().charAt(0
        for(IntWritable iw : values){
           //add iw to the current value
        }
  }
  public void cleanup(Context context) throws IOException,
                     InterruptedExceptio
        for (Character c : cache.keySet(
           context.write(new Text(c), ne
        }
  }
}
```

Called once in the life cycle of a Reducer object: before any calls to `reduce()`

Called once for each key that was assigned to the reducer

Called once in the life cycle of a Reducer object: after all calls to `reduce()`

Hadoop Programming Revisited: Counters

Counter basics

- **Gathering data about the data** we are analysing, e.g.
  - Number of key/value pairs processed in map
  - Number of empty lines/invalid lines
- Wanted:
  - **Easy** to collect
  - **Viewable during job execution** (stop Hadoop job early at too many invalid key/value pairs)
- What about log messages?
  - Write to the error log when an invalid line occurs
  - Hadoop's logs are huge, you need to know where to look
  - Aggregating stats from the logs requires another pass over it

Counter basics

- **Gathering data about the data** we are analysing, e.g.
  - Number of key/value pairs processed in map
  - Number of empty lines/invalid lines

WordCount example: what if we want to know more?
- How many words are not in the dictionary?
- How many words could not be parsed?
- How many words have less than two characters?

Question: how can you achieve that with your current Hadoop knowledge?

Counter basics

- Counters: Hadoop’s way of **aggregating** statistics
- Counters **count** (increment)
- **Built-in counters** maintain **metrics** of the job
  - MapReduce counters (e.g. #skipped records by all maps)
  - File system counters (e.g. #bytes read from HDFS)
  - Job counters (e.g. #launched map tasks)
- You have already seen them

Question: what are the reasons for the discrepancy in the amount of data read and written?

- Counters: Hadoop's way of **aggregating** statistics

```
Map-Reduce Framework
    Map input records=5903
    Map output records=47102
    Combine input records=47102
    Combine output records=8380
    Reduce output records=5934
File System Counters
    FILE: Number of bytes read=118124
    FILE: Number of bytes written=1075029
    HDFS: Number of bytes read=996209
    HDFS: Number of bytes written=59194
```

Built-in vs. user-defined

- **Built-in counters**: maintained by the JobTracker
- **User-defined Counters** are maintained by the task with which they are associated
  - Periodically sent to the Tasktracker and then the Jobtracker for global aggregation

Counter values are only definite once the job has completed (Counters may go down if a task fails!)

(pre-YARN setup)

Code example

WordCount - count words and chars*

```
enum Records {
    WORDS, CHARS;
};
public class WordCount {
  public static class Map extends MapReduceBase implements
             Mapper<LongWritable, Text, Text, IntWritable> {

    public void map(LongWritable key, Text value,
                        OutputCollector< Text, IntWritable> output,
                        Reporter reporter) throws IOException {
        String[] tokens = value.toString().split(" ");

        for (String s : tokens) {
          output.collect(new Text(s), new IntWritable(1));
          reporter.getCounter(Records.WORDS).increment(1);
          reporter.getCounter(Records.CHARS).increment(s.length());
        }
    }
}
```

several enum's possible: used to group counters

user-defined counters appear automatically in the final status output

Code example

WordCount - count words and chars*

```
enum Records
    WORDS, C
};
public class
  public sta


    public v                                             utput,
                                                         {
        Stri

        for
          ou
          re
          re                                             gth());
        }
    }
}
```

```
Map-Reduce Framework
        Map input records=5903
        Map output records=47102
        Combine input records=47102
        Combine output records=8380
        Reduce output records=5934
…
Records
        CHARS=220986
        WORDS=47102
```

user-defined counters appear automatically in the final status output

Code example II

```
enum Records { MAP_WORDS, REDUCE_WORDS; };

public class WordCount {
   --> MAPPER
  public void map(LongWritable key, Text value, OutputCollector<
                  Text,IntWritable> output, Reporter reporter)
                  throws IOException {

    String[] tokens = value.toString().split(" ");
    for (String s : tokens) {
      output.collect(new Text(s), new IntWritable(1);
      reporter.getCounter(Records.MAP_WORDS).increment(1);
    }
  }
   --> REDUCER
  public void reduce(Text key, Iterator<IntWritable> values,
                     OutputCollector<Text,IntWritable> output,
                     Reporter reporter) throws IOException {
    int sum = 0;
    while (values.hasNext())
      sum += values.next().get();
    reporter.getCounter(Records.REDUCE_WORDS).increment(sum);
  }
}
```

Question: Why does it make more sense in this scenario to define the Counter in the Mapper?

```
enum Records { MAP_WORDS, REDUCE_WORDS; };

public class WordCount {
   --> MAPPER
  public void map(LongWritable key, Text value, OutputCollector<
                   Text,IntWritable> output, Reporter reporter)
                   throws IOException {

    String[] tokens = value.toString().split(" ");
    for (String s : tokens) {
      output.collect(new Text(s), new IntWritable(1));
      reporter.getCounter(Records.MAP_WORDS).increment(1);
    }
  }
   --> REDUCER
  public void reduce(Text key, Iterator<IntWritable> values,
                      OutputCollector<Text,IntWritable> output,
                      Reporter reporter) throws IOException {
    int sum = 0;
    while (values.hasNext())
      sum += values.next().get();
    reporter.getCounter(Records.REDUCE_WORDS).increment(sum);
  }
}
```

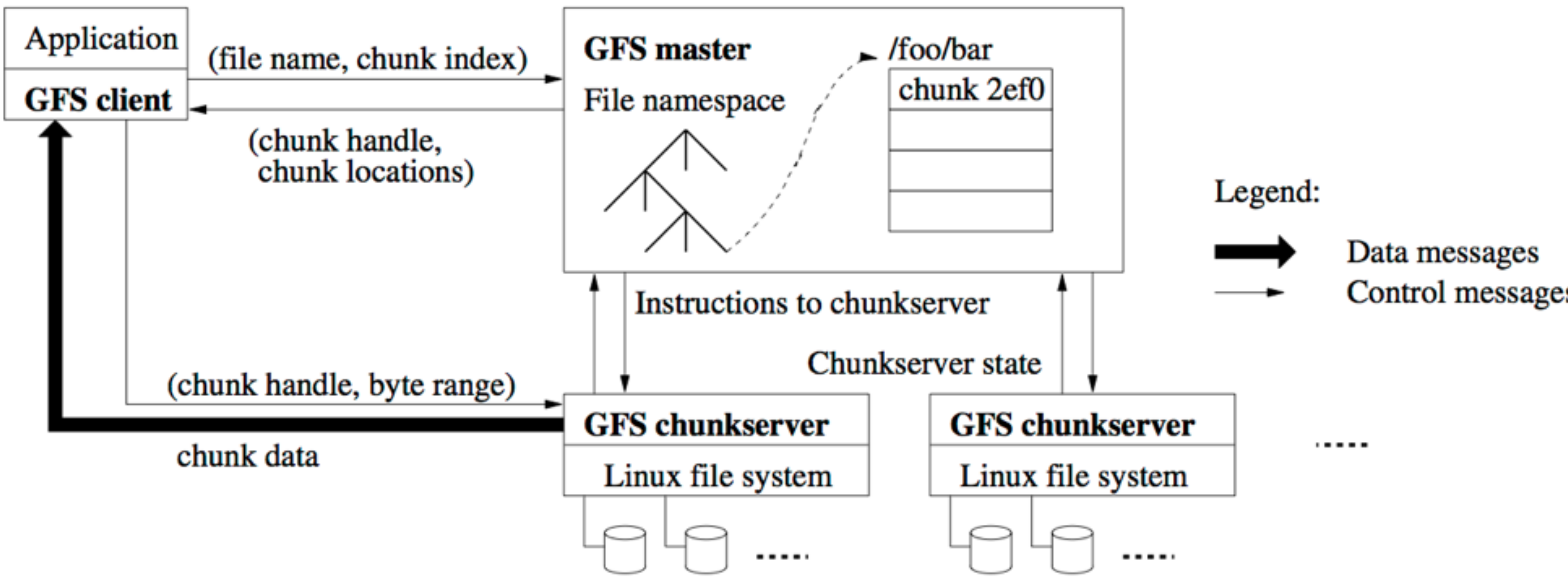
Question: Does this code work as expected?

WordCount across mapper/reducer

```
class InMemoryCounter {
  public int count;
  public InMemoryCounter() {count=0;}
}
public class WordCount {
  public static InMemoryCounter imc;
  static { imc = new InMemoryCounter(); }
  --> MAPPER
  public void map(LongWritable key, Text value, OutputCollector<
      Text, IntWritable> output, Reporter reporter) throws IOException {
      String[] tokens = value.toString().split(“ “);
      for(String w : tokens) {
         if(w.matches("[^a-zA-Z]")==true)  //count non-alphanumeric terms
            imc.count++;
         output.collect(new Text(w), new IntWritable(1));
      }
  }
  --> REDUCER
  public void reduce(Text key, Iterator<IntWritable> values,
    OutputCollector<Text, IntWritable> output, Reporter reporter)
    throws IOException {
      while (values.hasNext()) {
        int v = values.next().get();
        if(key.toString().matches("[^a-zA-Z]")==false)//count the rest
          imc.count+=v;
      }
  }
}
```

Job Scheduling

Last time … GFS/HDFS



distributed file system: file systems that manage the storage across a network of machines.

Source: http://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf

What about the jobs?

- "Hadoop job": unit of work to be performed (by a client)
  - Input data
  - MapReduce program
  - Configuration information
- Hadoop divides input data into **fixed size input splits**
  - One map task per split
  - One map function call for each record in the split
  - Splits are processed in parallel (if enough DataNodes exist)
- Job execution controlled by **JobTracker** and **TaskTrackers** (pre-YARN setup)

What about the jobs?

Question: What is the optimal input split size?

- Configuration information

Question: Can we also exploit data locality in the reducer?

- Splits are processed in parallel (if enough DataNodes exist)

- Job execution controlled by **JobTracker** and **TaskTrackers** (pre-YARN setup)

JobTracker and TaskTracker (Classic MapReduce or MapReduce 1)

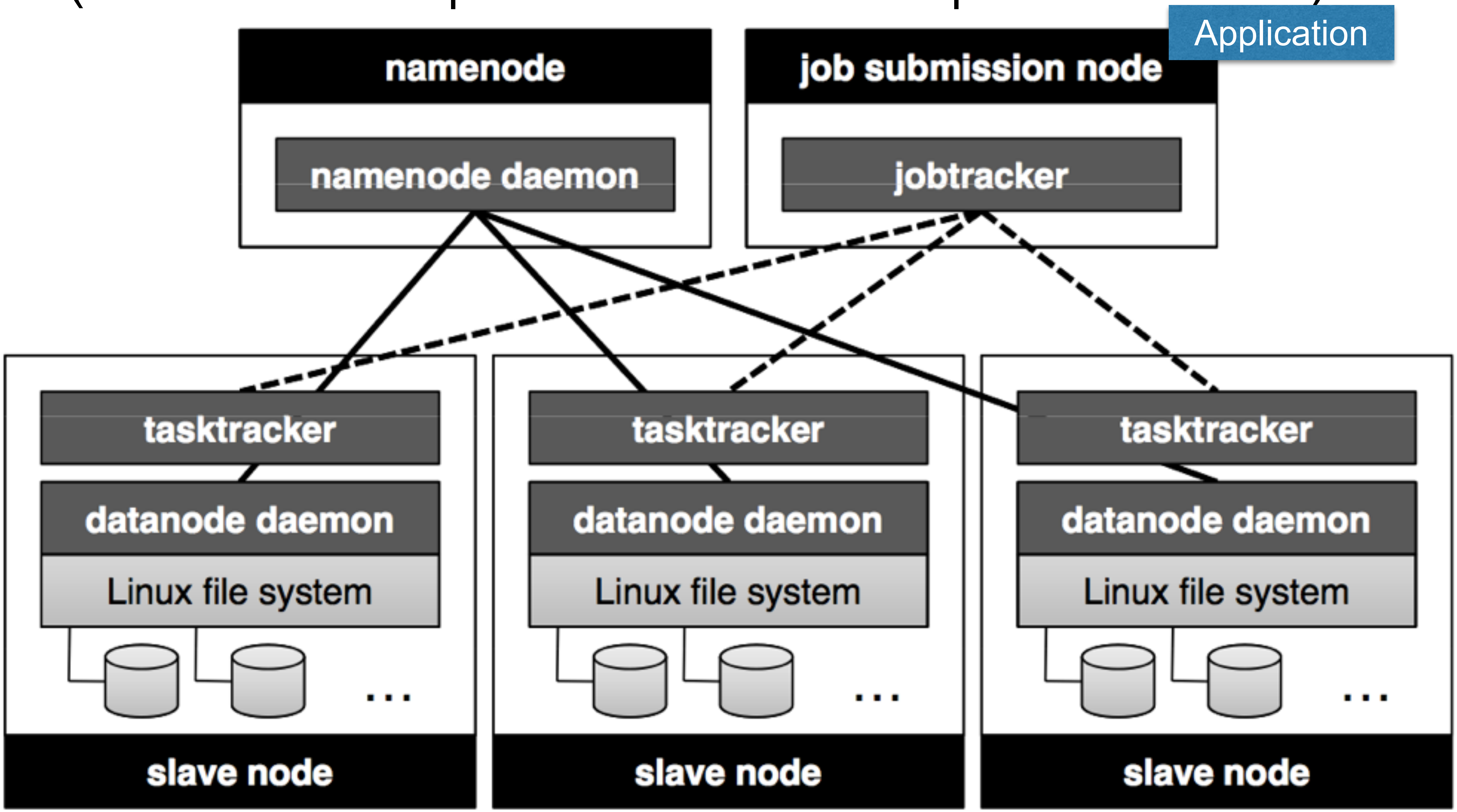


 image source: http://lintool.github.io/MapReduceAlgorithms/

JobTracker and TaskTracker (Classic MapReduce or MapReduce 1)

- JobTracker
  - **One** JobTracker **per Hadoop cluster**
  - **Middleman** between your application and Hadoop (single point of contact)
  - Determines the **execution plan** for the application (files to process, assignment of nodes to tasks, task monitoring)
  - Takes care of (supposed) **task failures**

- TaskTracker
  - **One** TaskTracker **per DataNode**
  - Manages individual tasks
  - **Keeps in touch** with the JobTracker (via HeartBeats) - sends progress report & signals empty task slots

YARN (MapReduce 2)

- JobTracker/TaskTrackers setup becomes a **bottleneck** in clusters with thousands of nodes
- As answer YARN has been developed (**Y**et **A**nother **R**esource **N**egotiator)
- YARN splits the JobTracker's tasks (job scheduling and task progress monitoring) into two daemons:
  - **Resource manager** (RM)
  - **Application master** (negotiates with RM for cluster resources; each Hadoop job has a dedicated master)

Job scheduling

- Thousands of tasks may make up one job
- Number of tasks can exceed number of tasks that can run concurrently
  - Scheduler maintains task queue and tracks progress of running tasks
  - Waiting tasks are assigned nodes as they become available
- "Move code to data"
  - Scheduler starts tasks on node that holds a particular block of data needed by the task if possible

Job scheduling

- Early on: **FIFO scheduler**
  - Job occupies the whole cluster while the rest waits
  - Not feasible in larger clusters

- Improvement: different job priorities VERY_HIGH, HIGH, NORMAL, LOW, or VERY_LOW
  - Next job is the one with the highest priority
  - No pre-emption: if a low priority job is occupying the cluster, the high priority job still has to wait

- Now: Fair Scheduler & Capacity Scheduler

Fair Scheduler

- Goal: every user receives a **fair share** of the cluster capacity over time
- If a single job runs, it uses the entire cluster
  - As more jobs are submitted, free task slots are given away such that each user receives a “fair share”
  - Short jobs complete in reasonable time, long jobs keep progressing
- A user who submits more jobs than a second user will not get more cluster resources on average

Fair Scheduler

- Jobs are placed in pools, default: one pool per user

- **Pre-emption**: if a pool has not received its fair share for a certain period of time, the scheduler will kill tasks in pools running over capacity to give more slots to the pool running under capacity
  - **Task kill != Job kill**
  - Scheduler needs to keep track of all users, resources used

Capacity Scheduler

- Cluster is made up of a number of queues (similar to the Fair Scheduler pools)
- Each queue has an allocated capacity
- Within each queue, jobs are scheduled using FIFO with priorities
- Idea: users (defined using queues) simulate a separate MapReduce cluster with FIFO scheduling for each user

Speculative execution

- Map phase is only as fast as slowest MAPPER
- Reduce phase is only as fast as slowest REDUCER
- Hadoop job is sensitive to stragglers (tasks that take unusually long to complete)
- Idea: identical copy of task executed on a second node; the output of whichever node finishes first is used (improvements up to 40%)
  - running task is killed
- Can be done for both MAPPER/REDUCER
- Strategy does not help if straggler due to skewed data distribution

Speculative execution

Question: Why is speculative execution in practice mostly restricted to map tasks?

- Hadoop job is sensitive to stragglers (tasks that take unusually long to complete)

Question: Can we use the Partitioner to avoid a skewed distribution (e.g. on WordCount)?

  - running task is killed

- Can be done for both MAPPER/REDUCER

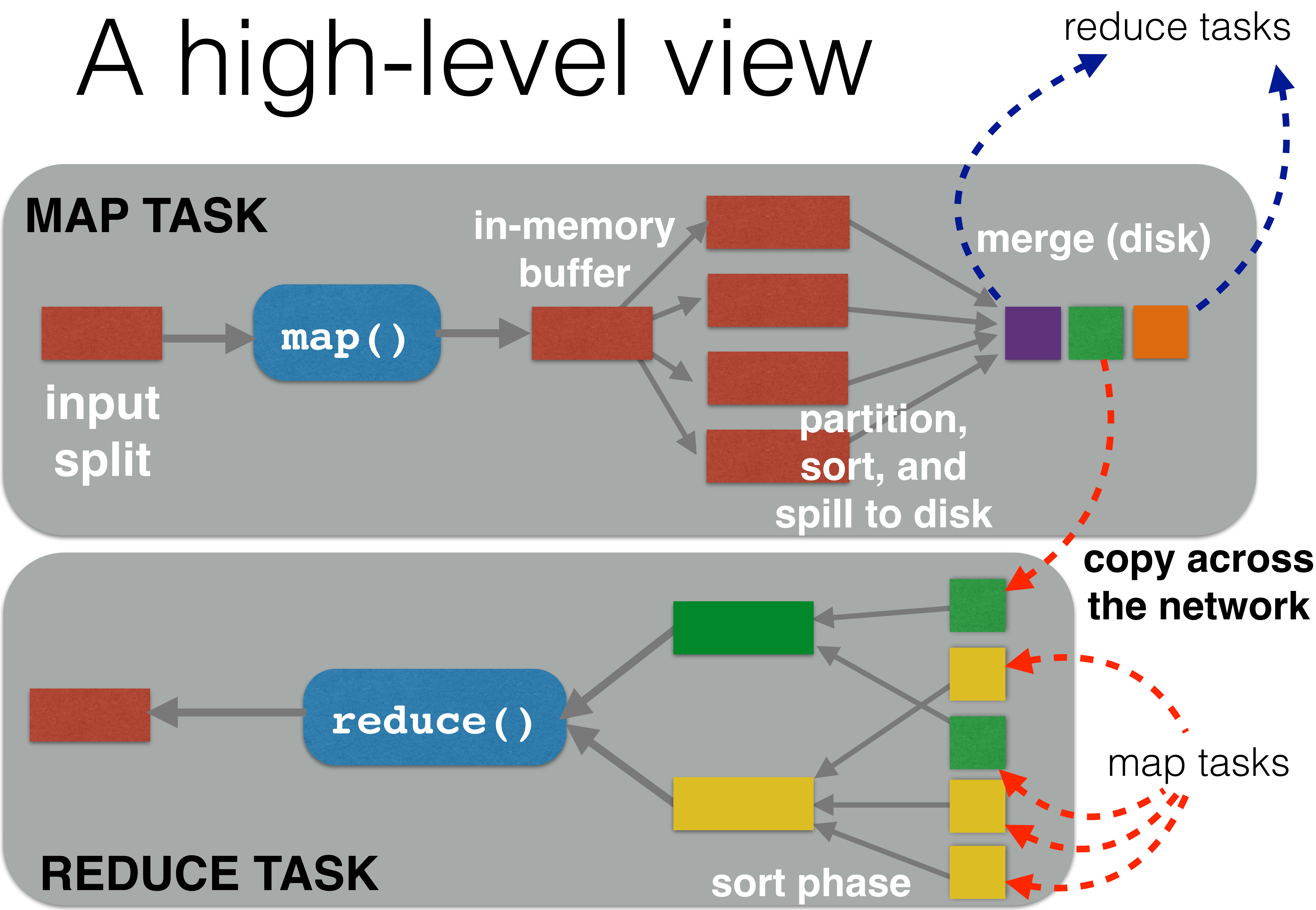
- Strategy does not help if straggler due to skewed data distribution

Shuffle & Sort

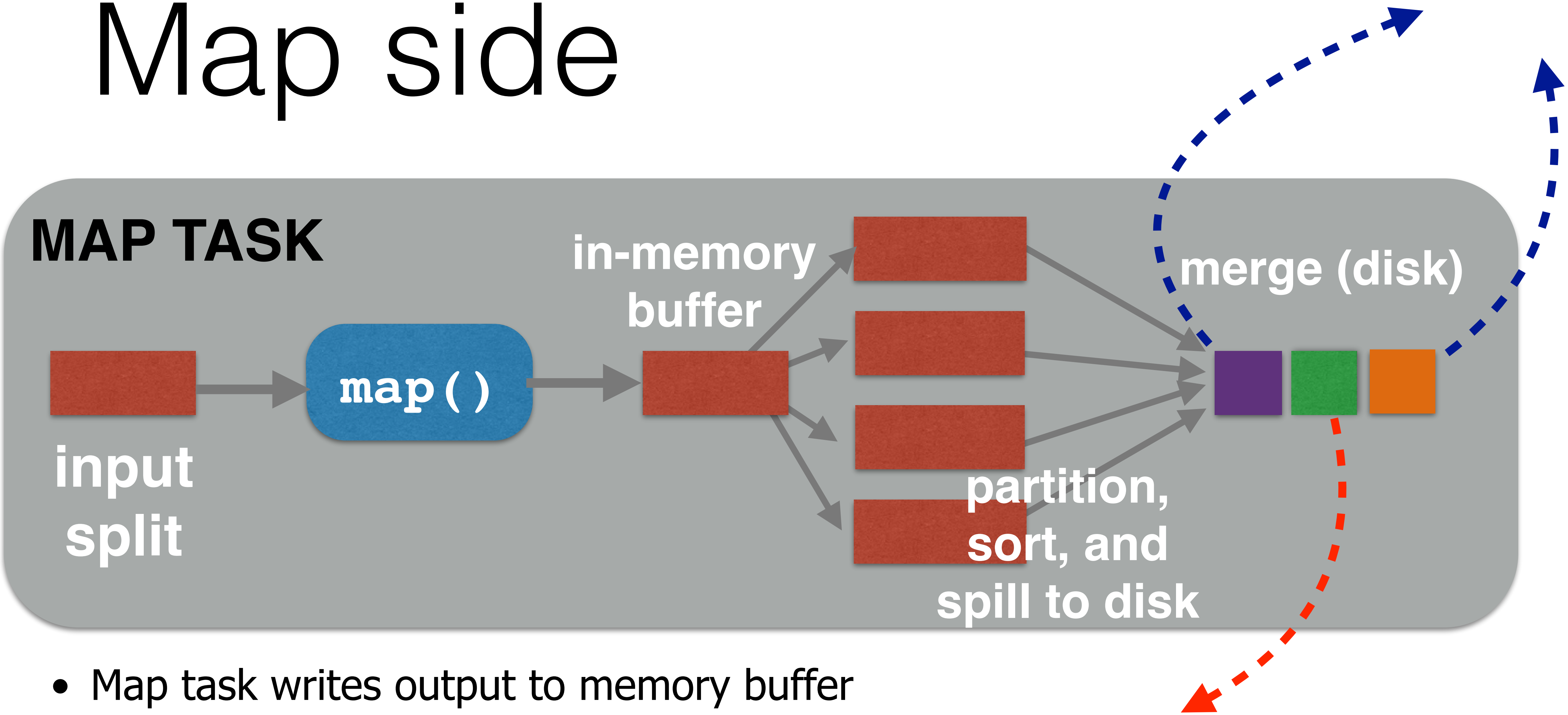
Shuffle & sort phase

- **Hadoop guarantee**: the input to every reducer is sorted by key
- **Shuffle**: sorting of **intermediate key/value pairs** and transferring them to the reducers (as input)
- "Shuffle is the heart of MapReduce"
- Understanding shuffle & sort is vital to recognise job bottlenecks
- Disclaimer: constantly evolving (*again*), description most valid for Hadoop 0.2X

A high-level view



Map side

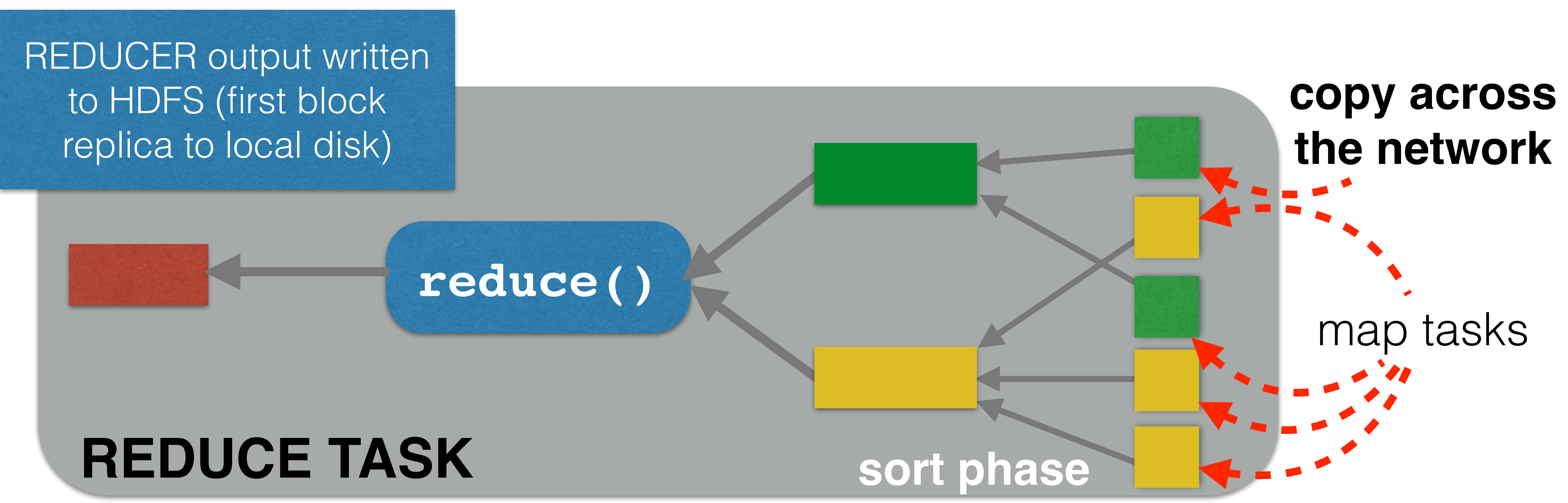


- Map task writes output to memory buffer
- Once the buffer is full, a background thread spills the content to disk (spill file)
  - Data is partitioned corresponding to reducers they will be send to
  - Within partition, in-memory sort by key [combiner runs on the output of sort]

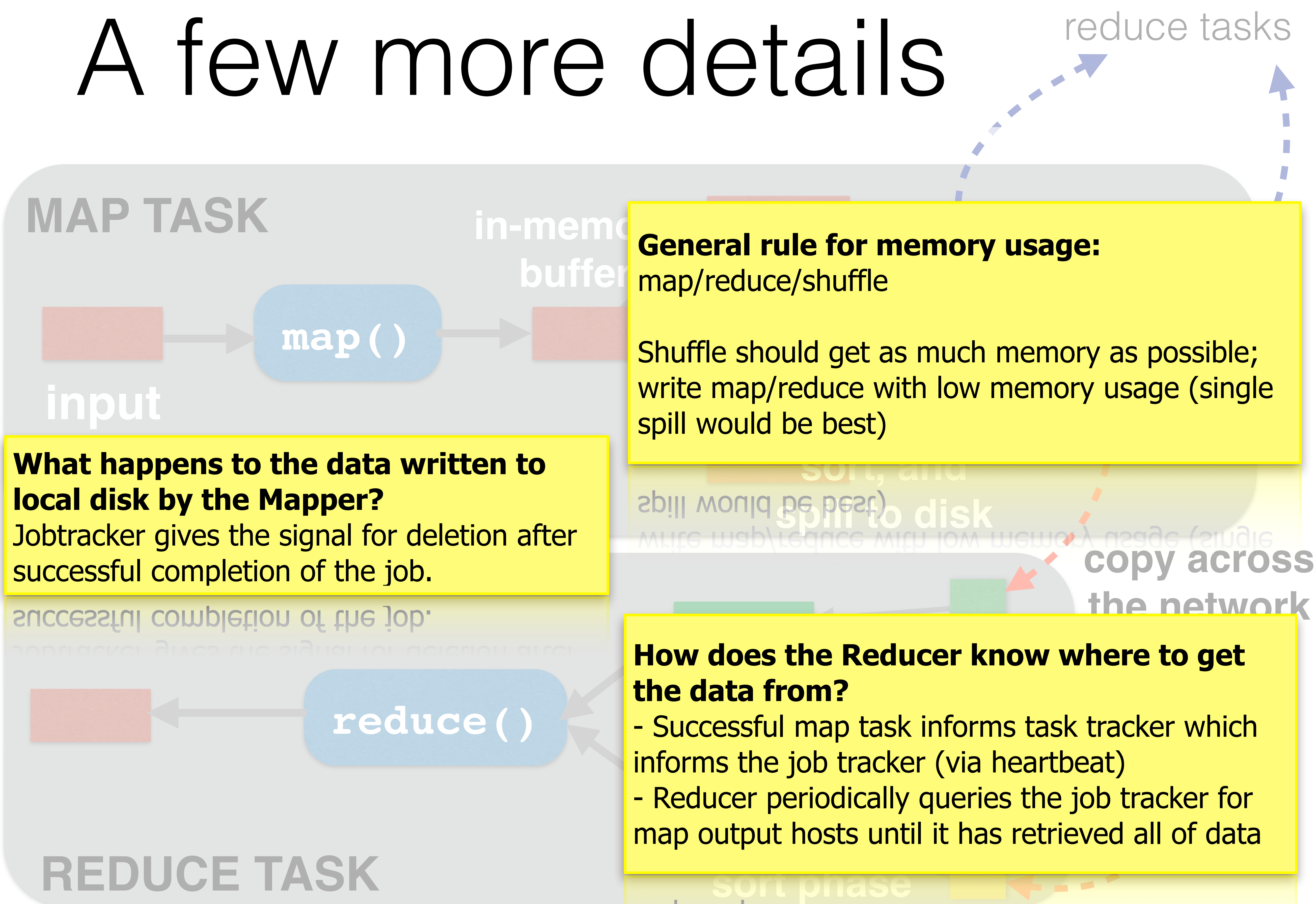
- After last `map()` call, the spill files are merged [combiner may run again]

Reduce side

- Reducer requires the map output for its partition from **all map tasks of the cluster**
- Reducer starts copying data as soon as a map task completes ("copy phase")
- Direct copy to reducer's memory if the output is small, otherwise copy to disk
- In-memory buffer is merged and spilled to disk once it grows too large
- **Combiner may run again**
- Once **all** intermediate keys are copied the "sort phase" begins: merge of map outputs, maintaining their sort ordering



A few more details



General rule for memory usage:
map/reduce/shuffle

Shuffle should get as much memory as possible; write map/reduce with low memory usage (single spill would be best)

What happens to the data written to local disk by the Mapper?
Jobtracker gives the signal for deletion after successful completion of the job.

How does the Reducer know where to get the data from?
- Successful map task informs task tracker which informs the job tracker (via heartbeat)
- Reducer periodically queries the job tracker for map output hosts until it has retrieved all of data

Sort phase recap

- Involves all nodes that executed map tasks and will execute reduce tasks
  - Job with m mappers and r reducers involves up to mr distinct copy operations
- Reducers can only start calling `reduce()` after all mappers are finished
  - **Key/value guarantee**: one key has all values "attached"
- Copying can start earlier for intermediate keys

Summary

- Hadoop Counters, setup/cleanup
- Job scheduling
- Shuffle & sort

THE END