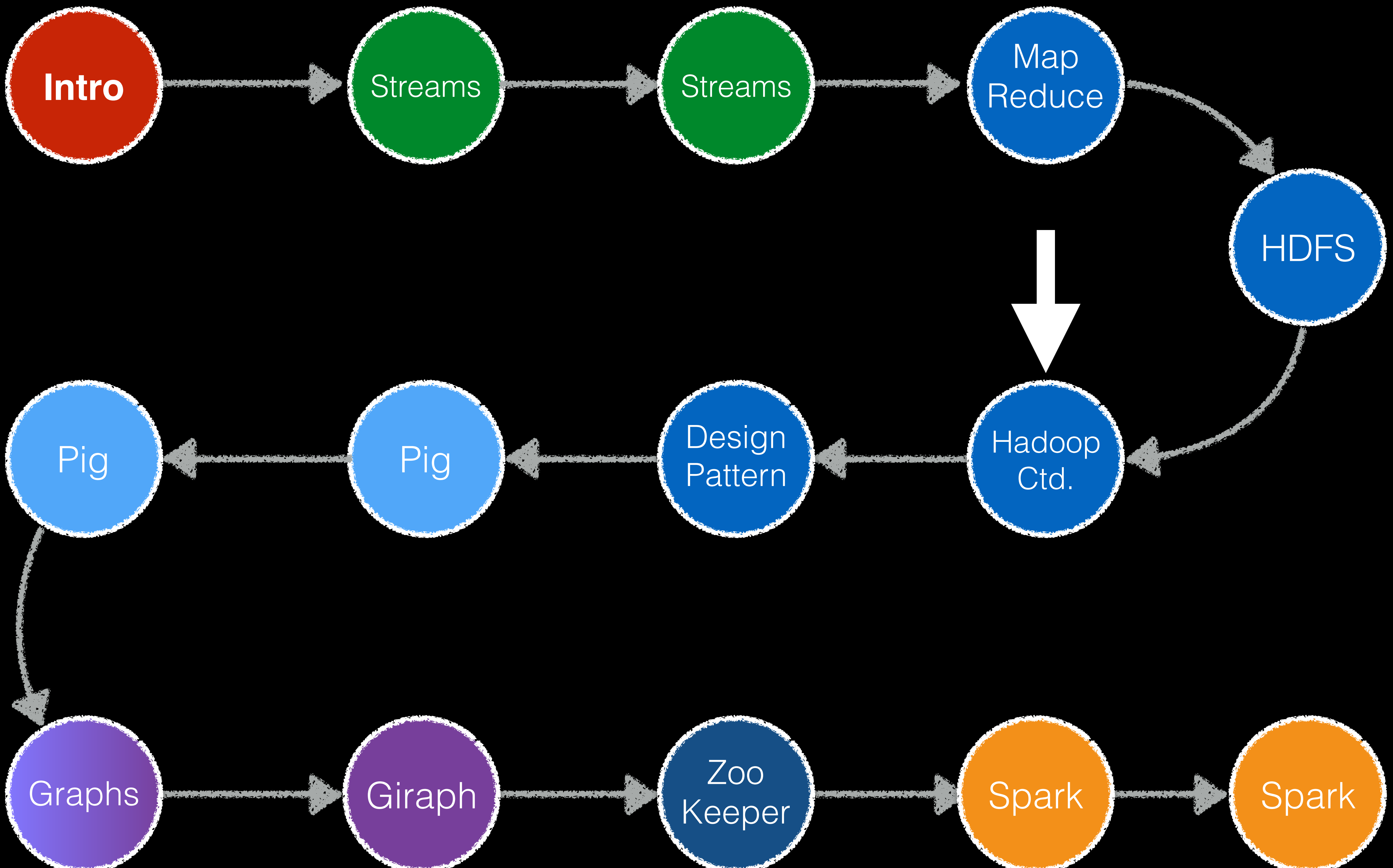
TI2736-B

Big Data Processing

Claudia Hauff
ti2736b-ewi@tudelft.nl

TUDelft


Intro
Streams
Streams
Map Reduce
HDFS
Hadoop Ctd.
Design Pattern
Pig
Pig
Graphs
Giraph
Zoo Keeper
Spark
Spark

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

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();//defined elsewhere, reads file from HDFS
 }

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

Setup

Setup useful for one-off operations:

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

WordCount - count only valid dictionary terms*

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

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

 public void setup(Context context) throws
      dictionary = Sets.newHashSet();
      loadDictionary();//defined elsewhere
 }

 public void map(Text key, IntWritable val
                     throws IOException, I
      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?

```java
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) thr
      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
  - **Estimates are viewable during job execution** (e.g. to stop a Hadoop job early at too many invalid key/value pairs)
- Why not use log messages instead?
  - 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
- 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

- 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

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

Counter basics

- 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**: exist for each Hadoop job
- **User-defined Counters** are maintained by the application they are associated with
  - Periodically sent to the Tasktracker and then the Jobtracker for global aggregation (pre-YARN setup)
  - Aggregated per job by the ResourceManager (YARN)

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

Code example

WordCount* - count words and chars

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

    public void map(LongWritable key, Text value,
                          Context context) throws IOException {
        String[] tokens = value.toString().split(" ");

        for (String s : tokens) {
          context.write(new Text(s), new IntWritable(1));
          context.getCounter(Records.WORDS).increment(1);
          context.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, CHARS;
};
public class WordC
  public static cl
               Mapper

    public void ma

          String[] t

          for (Strin
            context.
            context.
            context.
          }
    }
}
```

```
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

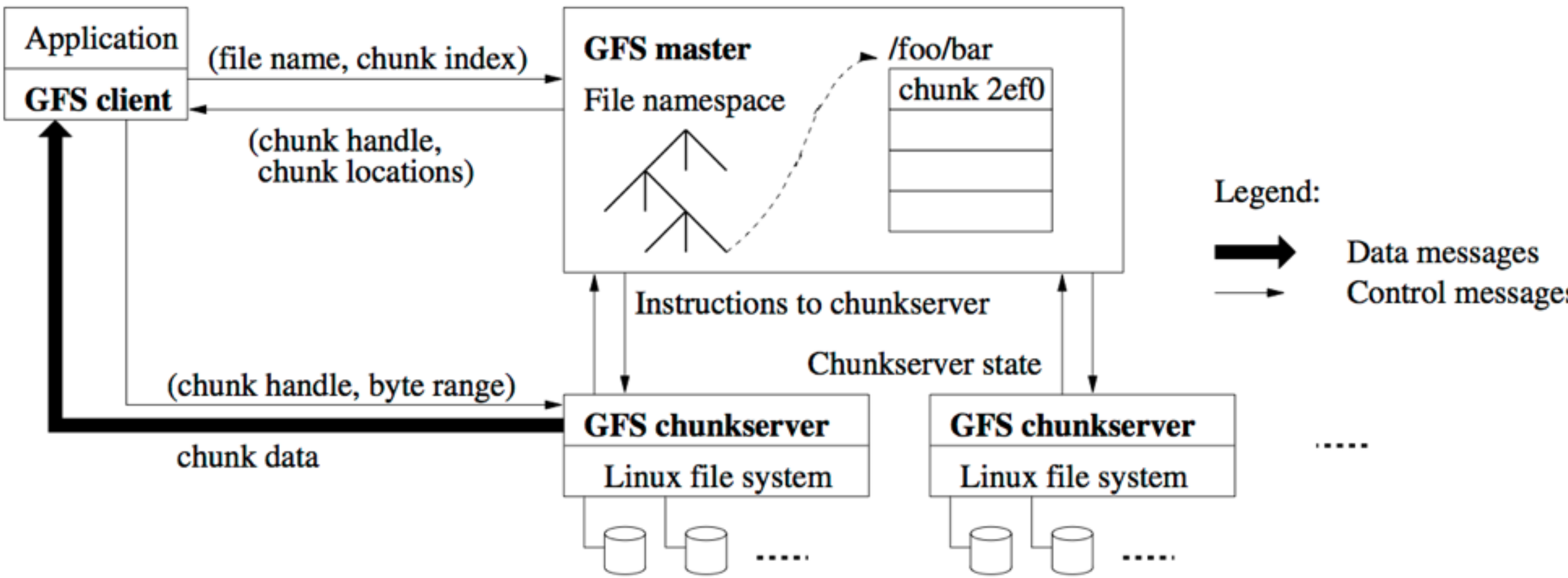
Code example II

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

public class WordCount {
  --> MAPPER
  public void map(LongWritable key, Text value,
                  Context context)
                  throws IOException {

    String[] tokens = value.toString()
    for (String s : tokens) {
      context.write(new Text(s), new I
      context.getCounter(Records.MAP_W
    }
  }
   --> REDUCER (Combiner is a copy of the Reducer)
  public void reduce(Text key, Iterator<IntWritable> values,
                     Context context) throws IOException {
    int sum = 0;
    while (values.hasNext())
      sum += values.next().get();
    context.getCounter(Records.REDUCE_WORDS).increment(sum);
  }
}
```

```
...
Records
     MAP_WORDS=47102
     REDUCE_WORDS=47102
```

Job Scheduling

Last time … GFS/HDFS



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

Image 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
  - 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)

JobTracker and TaskTracker

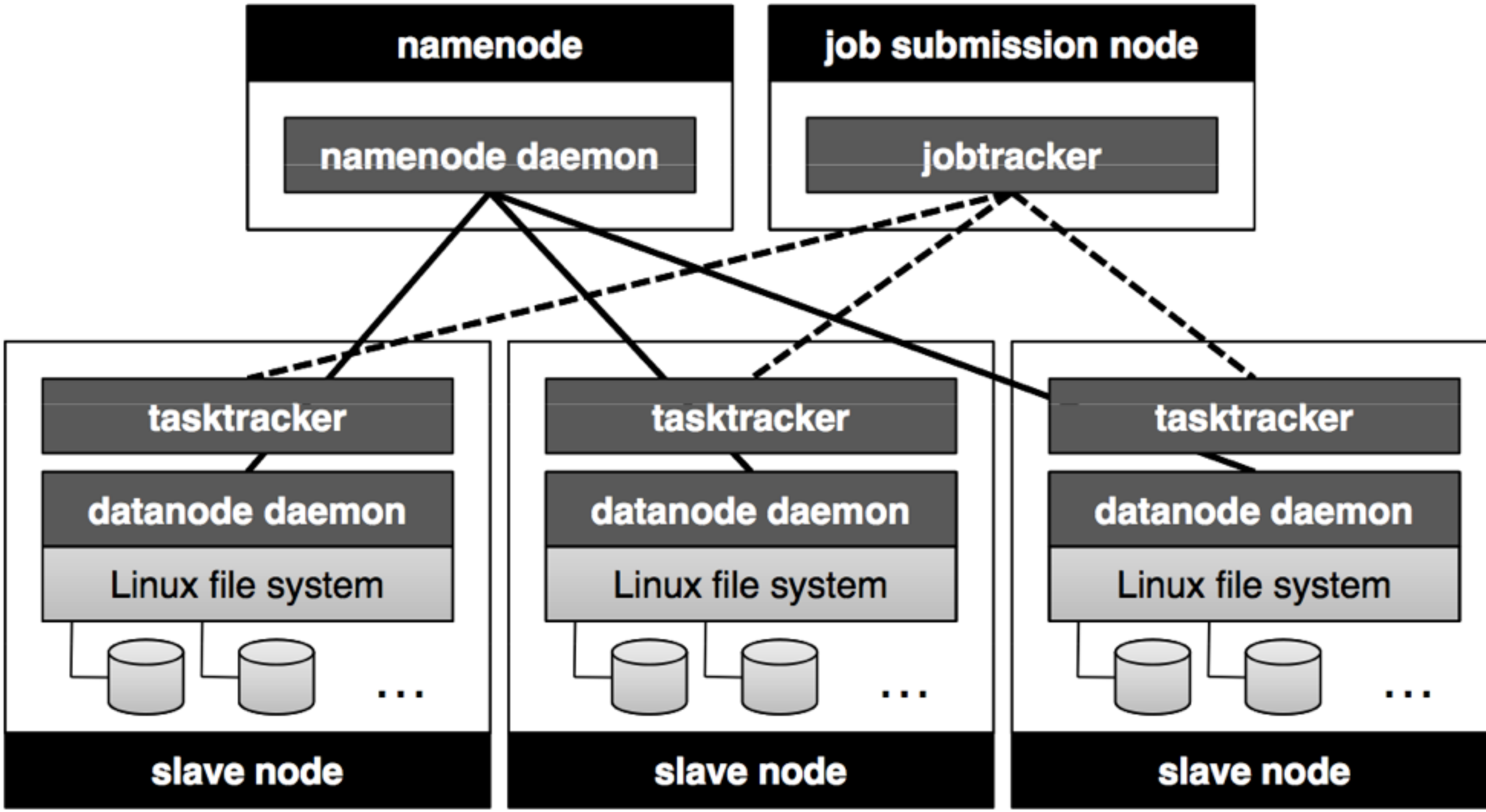


Image source: http://lintool.github.io/MapReduceAlgorithms/

Hadoop in practice: Yahoo! (2010)

"MapReduce 1"

- **40 nodes/rack** sharing one IP switch
- **16GB RAM** per cluster node, 1-gigabit Ethernet
- 70% of disk space allocated to HDFS
  - Remainder: operating system, data emitted by Mappers (not in HDFS)
- **NameNode**: up to **64GB RAM**
- **Total storage**: 9.8PB -> **3.3PB** net storage (replication: 3)
- **60 million files**, 63 million blocks
- 54,000 blocks hosted per DataNode
- **1-2 nodes lost per day**
- **Time for cluster to re-replicate lost blocks: 2 minutes**

HDFS cluster with 3,500 nodes

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

- FIFO scheduler
- Priority scheduler
- Fair scheduler
- Capacity scheduler

Basic schedulers

- 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

Fair Scheduler I

- 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 II

- Jobs are placed in **pools**, default: one pool per user
- **Pre-emption**: if a pool has not received its fair share for a 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

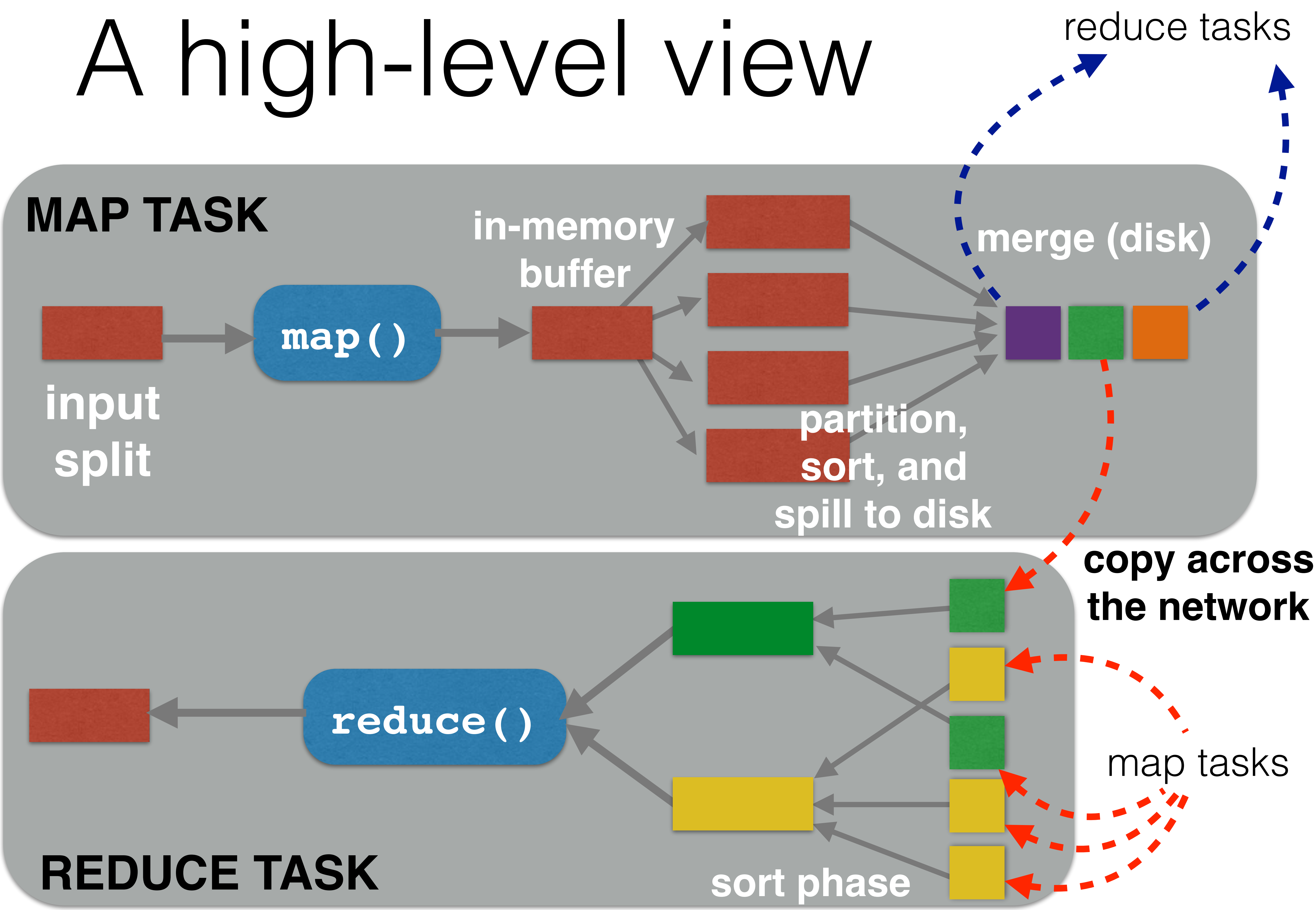
- **Map phase is only as fast as slowest MAPPER**
- **Reduce phase is only as fast as slowest REDUCER**
- Hadoop job is sensitive to **stragglers** (tasks that take unusually long to complete)
- Idea: **identical copy** of task executed on a second node; the output of whichever node finishes first is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

Shuffle & Sort

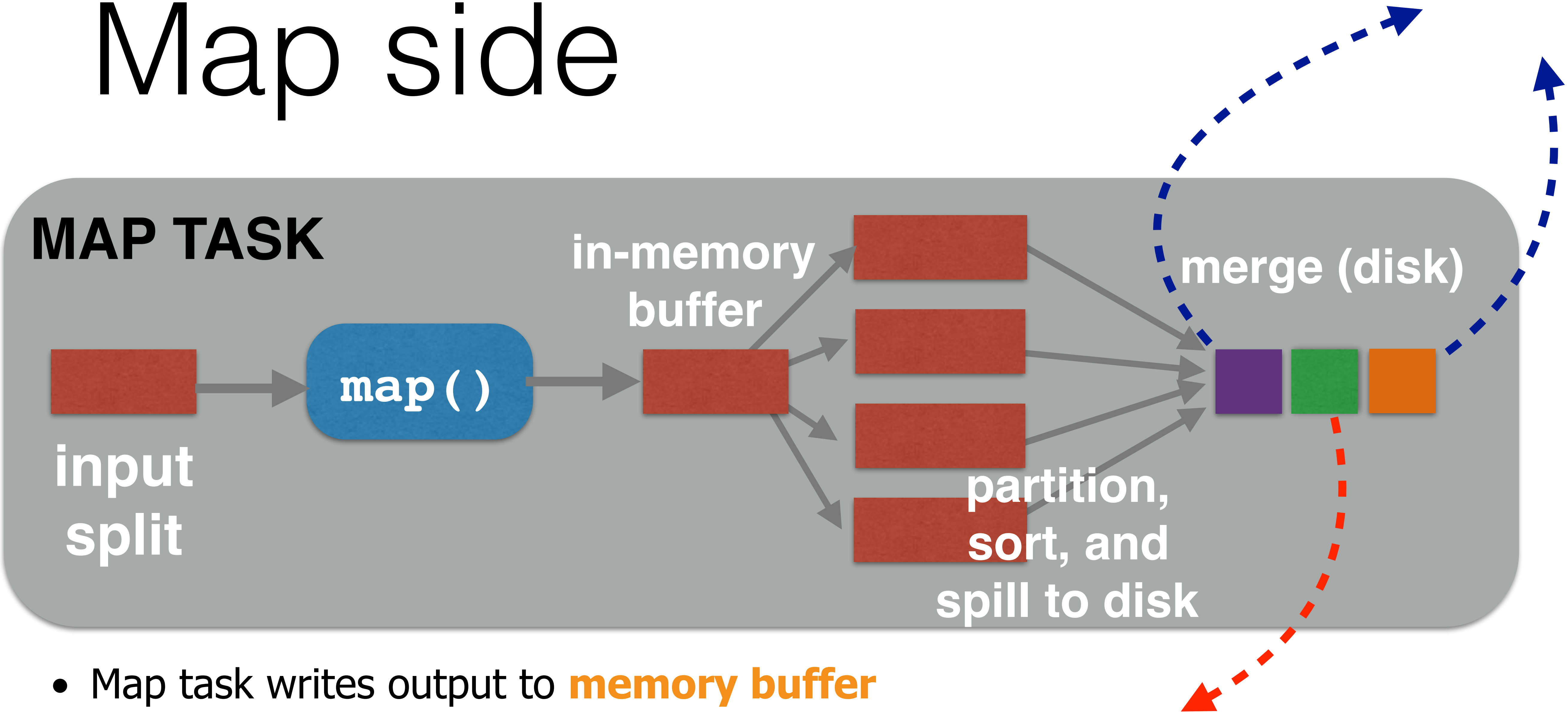
Shuffle & sort phase

- **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*)

A high-level view



Map side

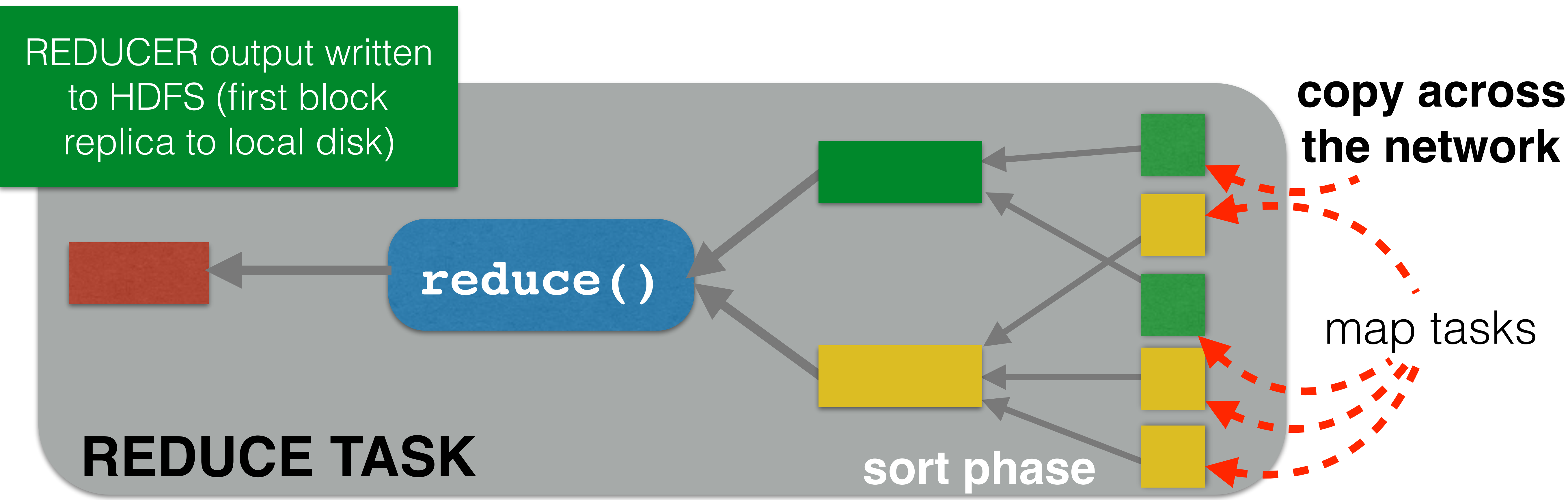


- Map task writes output to **memory buffer**
- Once the buffer is full, a background thread **spills** the content to **disk** (spill file)
  - Data is partitioned corresponding to reducers they will be send to
  - Within partition, in-memory sort by key [combiner runs on the output of sort]

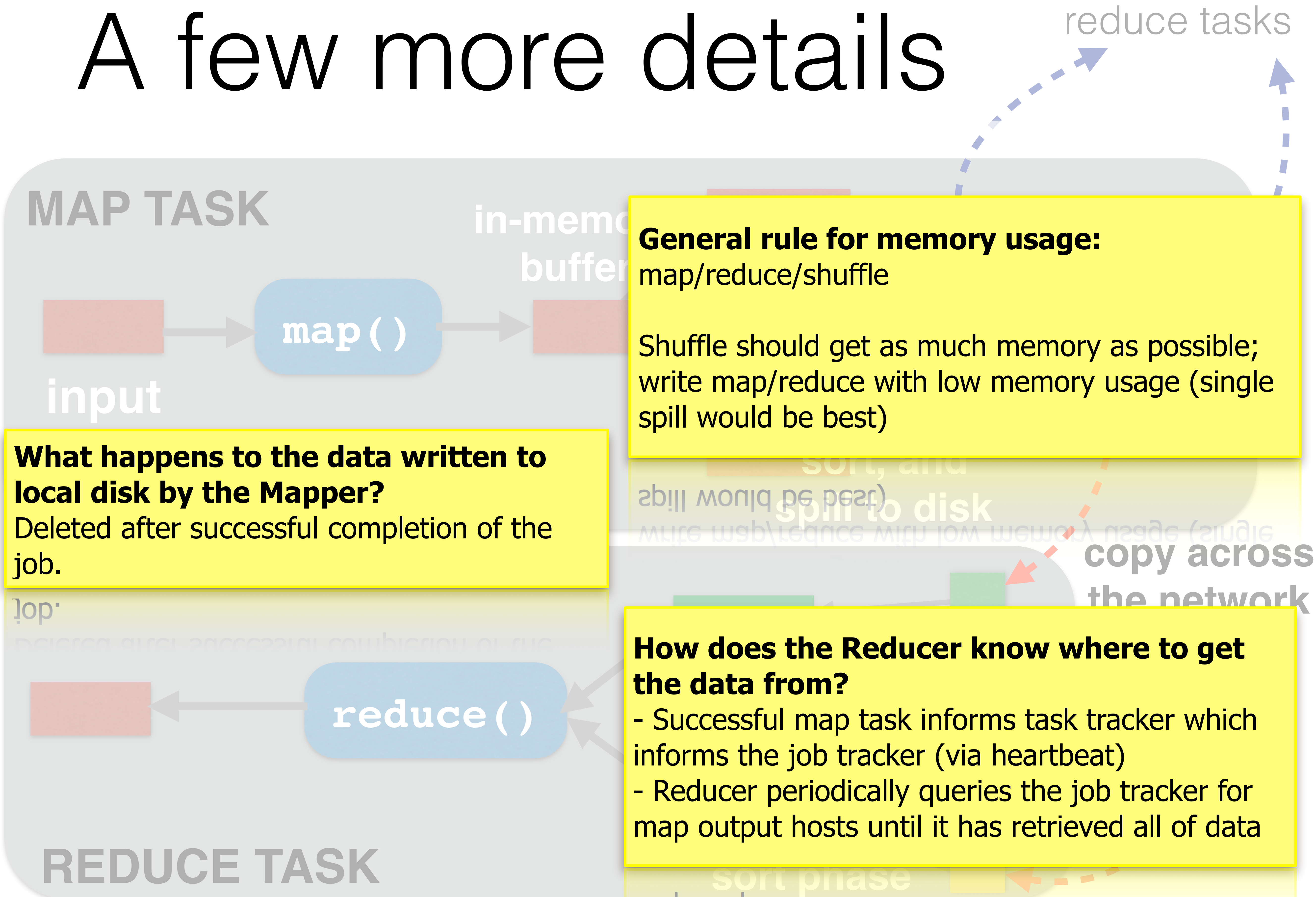
- After last `map()` call, the spill files are merged [combiner may run again]

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?
Deleted 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 $m*r$ 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

Data input

Input splits and logical bounds

- 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

Input split	record
(part of) a text file	line of text
(range of) database table rows	a single row
(part of) an XML file	XML element
(part of) a video stream	keyframe

Input split

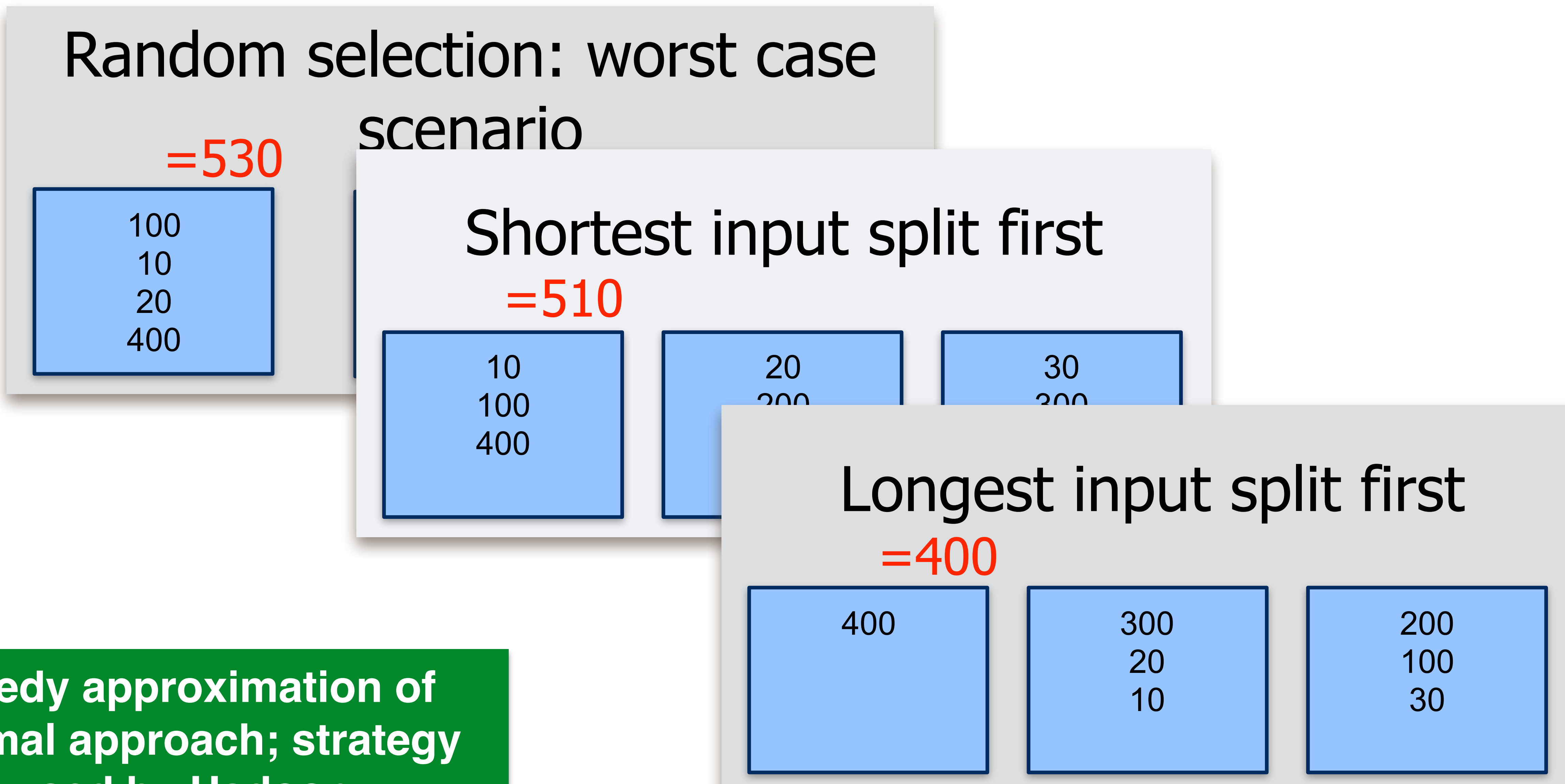
```java
public abstract class InputSplit {

  public abstract long getLength() throws
                          IOException, InterruptedException;

  public abstract String[] getLocations() throws
                          IOException, InterruptedException;
}
```

Scenario: There are less free map slots than input splits.

Questions: Given the input splits and their sizes,
what are possible strategies of how to pick the next input split to process by a map task?
Given 3 free map slots and 7 input splits of sizes
{10, 20, 30, 100, 200, 300, 400}, which strategy works best?

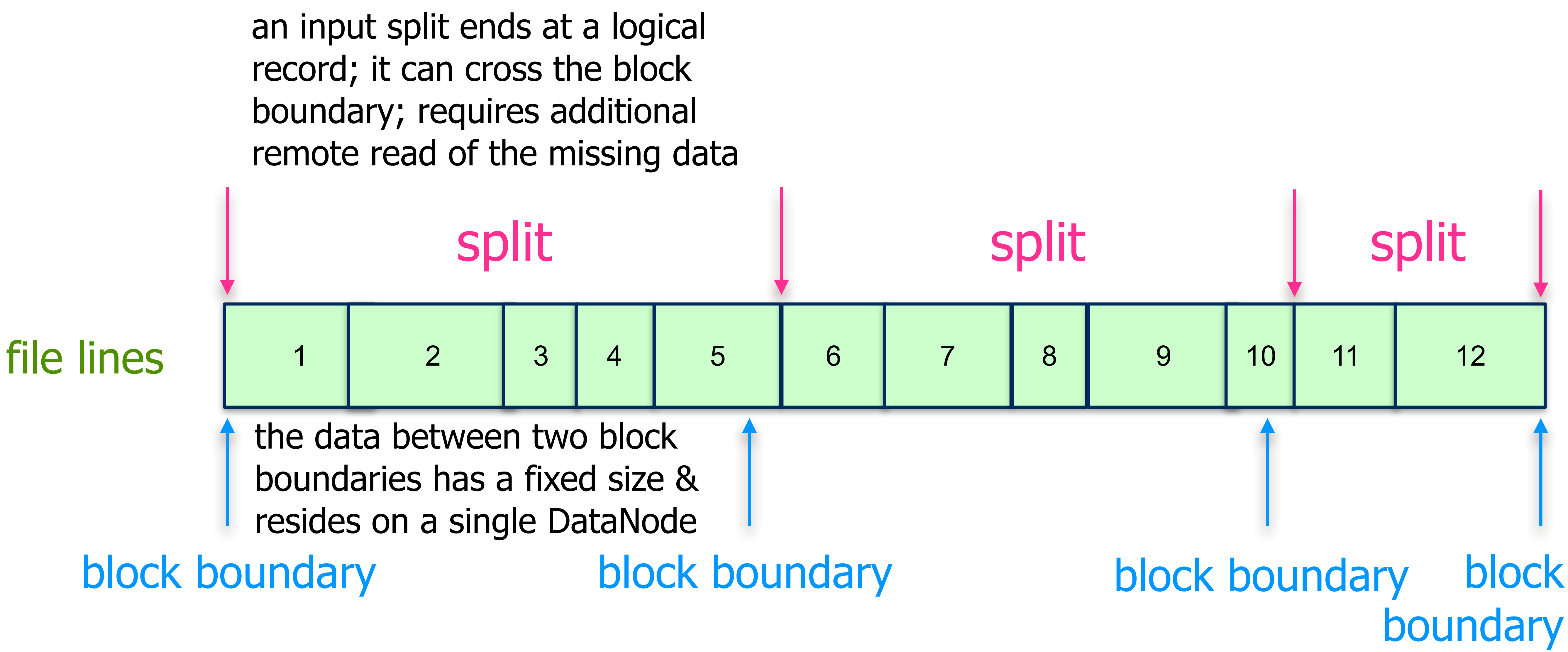
Input splits and logical bounds

3 free map slots and 7 input splits of sizes **{10, 20, 30, 100, 200, 300, 400}**



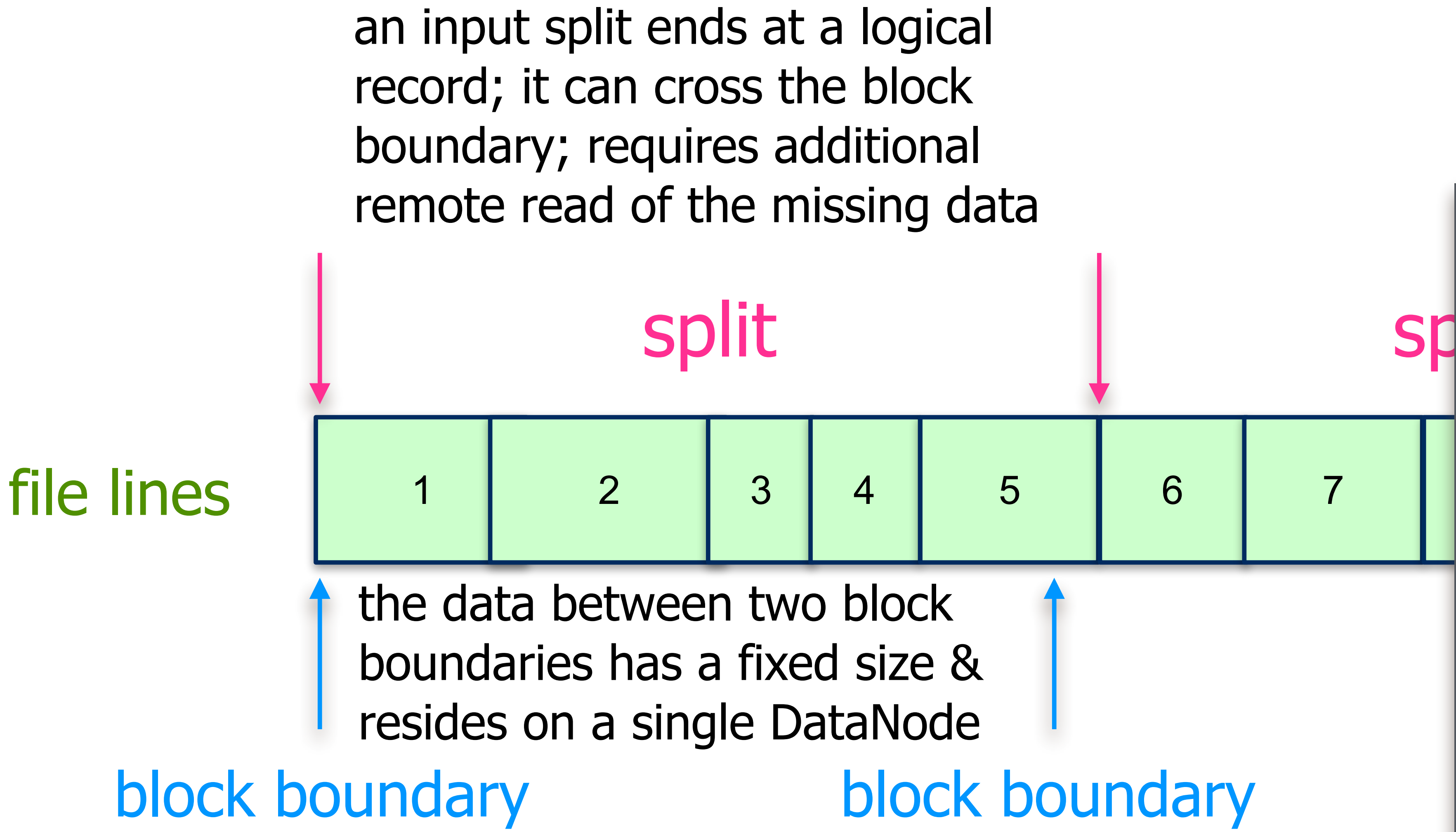
greedy approximation of optimal approach; strategy used by Hadoop

Input splits vs. HDFS blocks



`TextInputFormat`

Input splits vs. HDFS blocks



What about:

`<wikipedia>`
`<page></page>`
. . . .
`10GB later`
. . . .
`<page></page>`
`</wikipedia>`

`TextInputFormat`

HDFS: Compression and Small Files

Splittable compression

Compression format	Tool	Algorithm	Filename extension	Splittable?
DEFLATE[a]	N/A	DEFLATE	*.deflate*	No
gzip	*gzip*	DEFLATE	*.gz*	No
bzip2	*bzip2*	bzip2	*.bz2*	Yes
LZO	*lzop*	LZO	*.lzo*	No[b]
LZ4	N/A	LZ4	*.lz4*	No
Snappy	N/A	Snappy	*.snappy*	No

space/time tradeoff: faster (de)compression means less space savings

Source: Tom White's Hadoop: The Definite Guide

Splittable compression

Compression format	Tool	Algorithm	Filename extension	Splittable?
DEFLATE[a]	N/A	DEFLATE	*.deflate*	No
gzip	*gzip*	middle ground		No
bzip2	*bzip2*	better compression		Yes
LZO	*lzop*	optimized for speed, less effective compression		No[b]
LZ4	N/A			No
Snappy	N/A			No

space/time tradeoff: faster (de)compression means less space savings

Source: Tom White's Hadoop: The Definite Guide

Splittable is an important attribute

- 1GB uncompressed file
  - Stored within 16 blocks on HDFS (block size 64MB)
  - Hadoop job creates 16 input splits, **each processed by one map task**

- 1GB gzip-compressed file
  - Stored within 16 blocks on HDFS
  - Hadoop job cannot create 16 input splits (reading at an arbitrary point does not work)
  - A **single map task** will process the 16 HDFS blocks

Hadoop Archives

- Storing a large number of small files is inefficient
  - **But**: not all files can be **easily** converted to blocks (e.g. millions of images)

- Files and blocks **occupy namespace** which is limited by the physical memory in the NameNode
  - Small files take up large portion of namespace but not the disk space
  - Rule of thumb: 150 bytes per file/directory/block (1 million files of one block each: 300MB of memory)

- Hadoop Archive (`*.har`) is a solution

small=substantially less than the block size (64MB/128MB)

A Web special: WARC

- **Web ARCHive format**: aggregates digital resources in an archive and keeps track of related information
  - Per resource: text header and arbitrary data
- Extension of the **Internet Archive's ARC format**
- Commonly used to store **Web crawls**

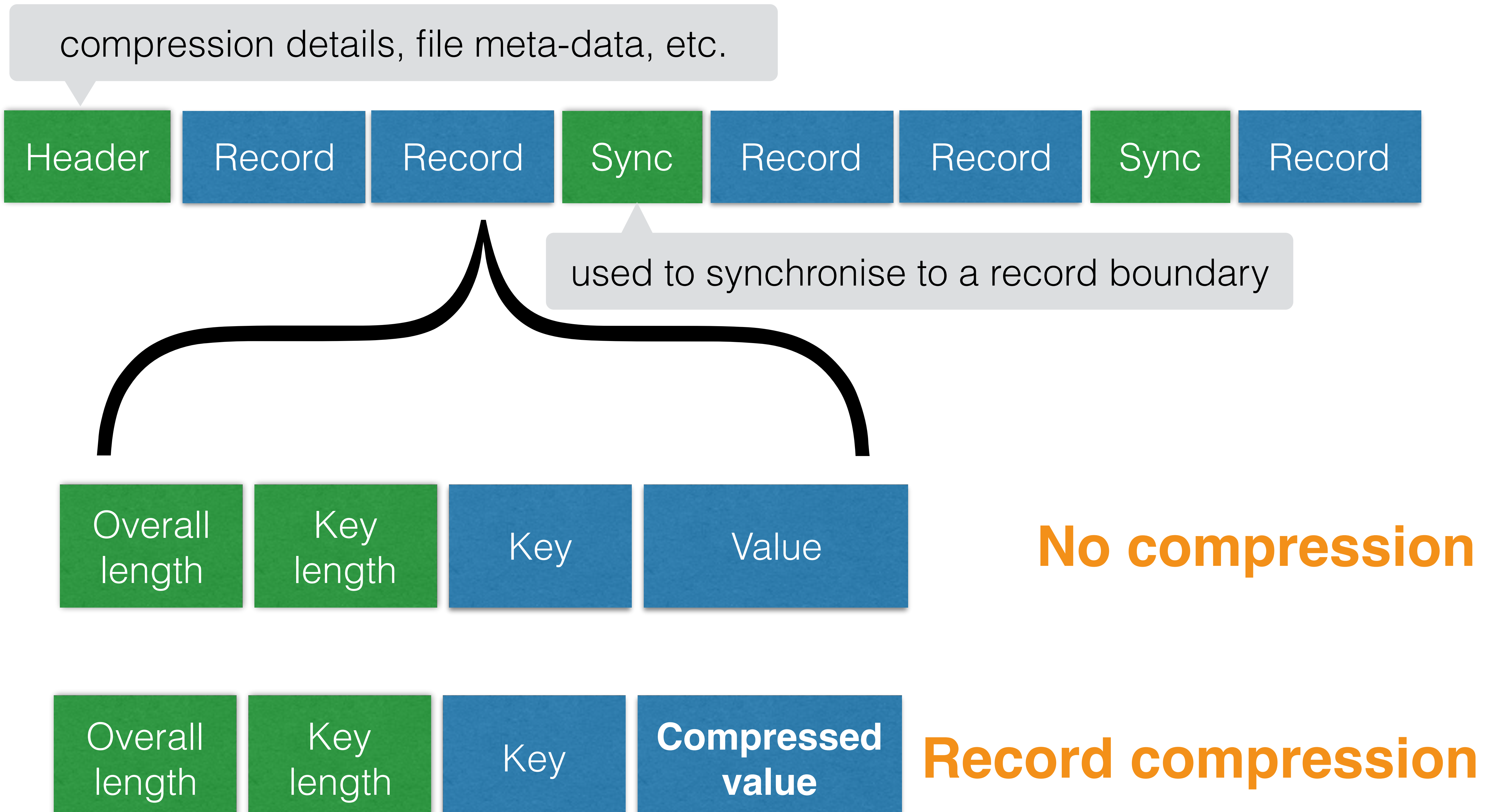
A Web special: WARC

```
WARC/0.17
WARC-Type: response
WARC-Target-URI: http://www.archive.org/robots.txt
WARC-Date: 2008-04-30T20:48:25Z
WARC-Payload-Digest: sha1:SUCGMUVXDKVB5CS2NL4R4JABNX7K466U
WARC-IP-Address: 207.241.229.39
WARC-Record-ID: <urn:uuid:e7c9eff8-f5bc-4aeb-b3d2-9d3df99afb30>
Content-Type: application/http; msgtype=response
Content-Length: 782

HTTP/1.1 200 OK
Date: Wed, 30 Apr 2008 20:48:24 GMT
Server: Apache/2.0.54 (Ubuntu) PHP/5.0.5-2ubuntu1.4 mod_ssl/2.0.54 OpenSSL/
0.9.7g
Last-Modified: Sat, 02 Feb 2008 19:40:44 GMT
ETag: "47c3-1d3-11134700"
Accept-Ranges: bytes
Content-Length: 467
Connection: close
Content-Type: text/plain; charset=UTF-8

##############################################
# Welcome to the Archive!
```

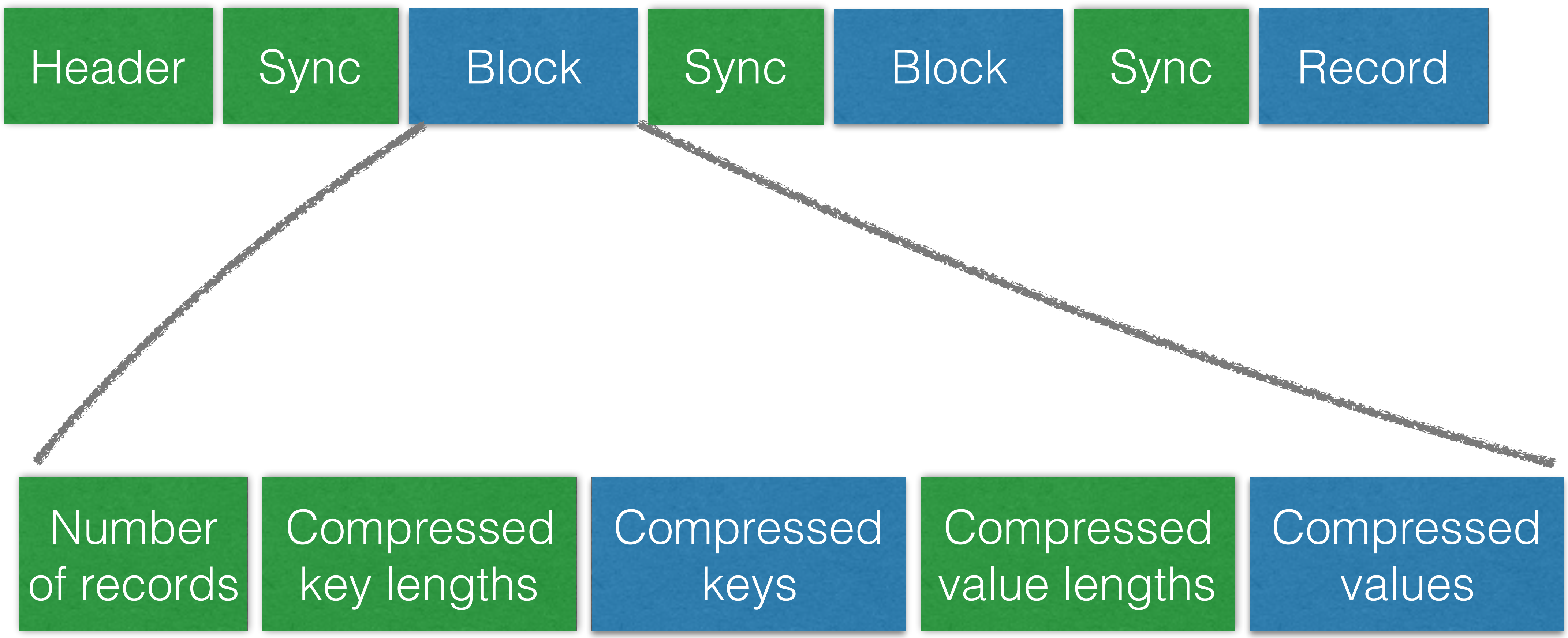
Hadoop's SequenceFile format

- Main usage: **intermediate** output of Mappers written in this format
- Flat file, consisting of **binary key-value pairs**
- Defines a `Reader`, `Writer` and `Sorter`
- **Three types**:
  - Uncompressed key-value pairs
  - Compressed values ("record compressed")
  - Keys and values compressed ("block compressed")
- From small files to SequenceFile: `(some_key,file_content)`

Hadoop's SequenceFile format



Hadoop's SequenceFile format



Block compression (allows most compression)

HDFS: the rest

HDFS is just one possible implementation ….

Filesystem	URI scheme	Java implementation (all under org.apache.hadoop)	Description
Local	`file`	`fs.LocalFileSystem`	A filesystem for a locally connected disk with client-side checksums. Use `RawLocalFileSystem` for a local filesystem [illegible]ns. See LocalFileSystem.
HDFS	`hdfs`	`hdfs.DistributedFileSystem`	Hadoop's distributed filesystem. HDFS is designed to work efficiently in conjunction with MapReduce.
FTP	`ftp`	`fs.ftp.FTPFileSystem`	A filesystem backed by an FTP server.
S3 (native)	`s3n`	`fs.s3native.NativeS3FileSystem`	A filesystem backed by Amazon S3. See http://wiki.apache.org/hadoop/AmazonS3.
S3 (block-based)	`s3`	`fs.s3.S3FileSystem`	A filesystem backed by Amazon S3, which stores files in blocks (much like HDFS) to overcome S3's 5 GB file size limit.
HAR	`har`	`fs.HarFileSystem`	A filesystem layered on another filesystem for archiving files. Hadoop Archives are typically used for archiving files in HDFS to reduce the namenode's memory usage. See Hadoop Archives.

we can also use the local filesystem

Source: Tom White's Hadoop: The Definite Guide

Summary

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

THE END