Hadoop/HDFS (2)

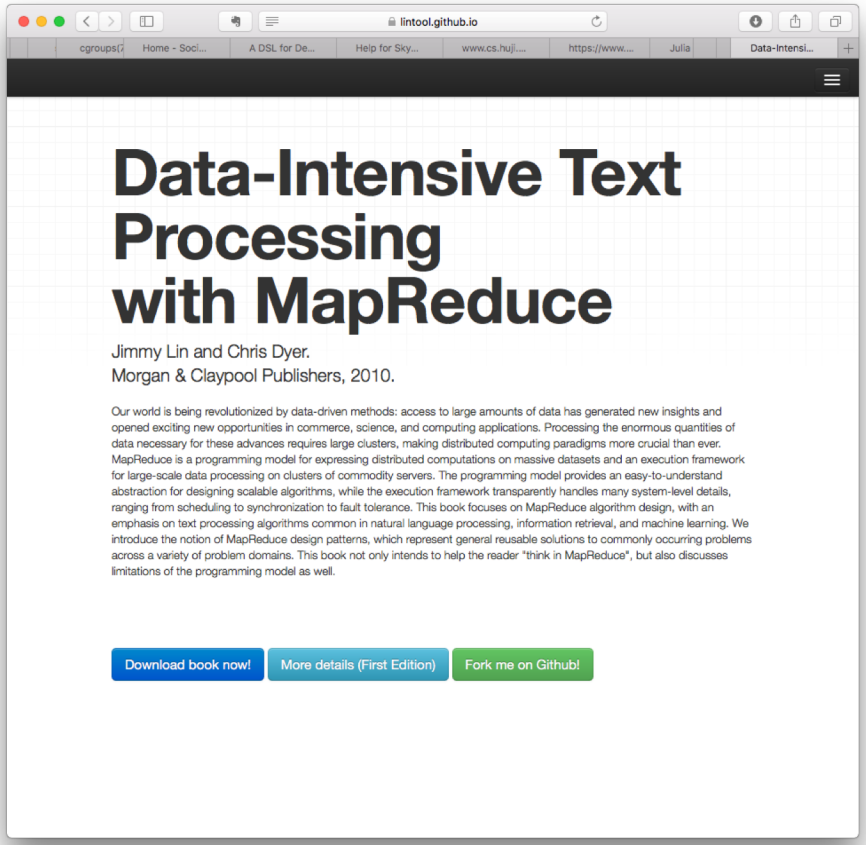
School of Computer Science, UCD

Scoil na Ríomheolaíochta, UCD

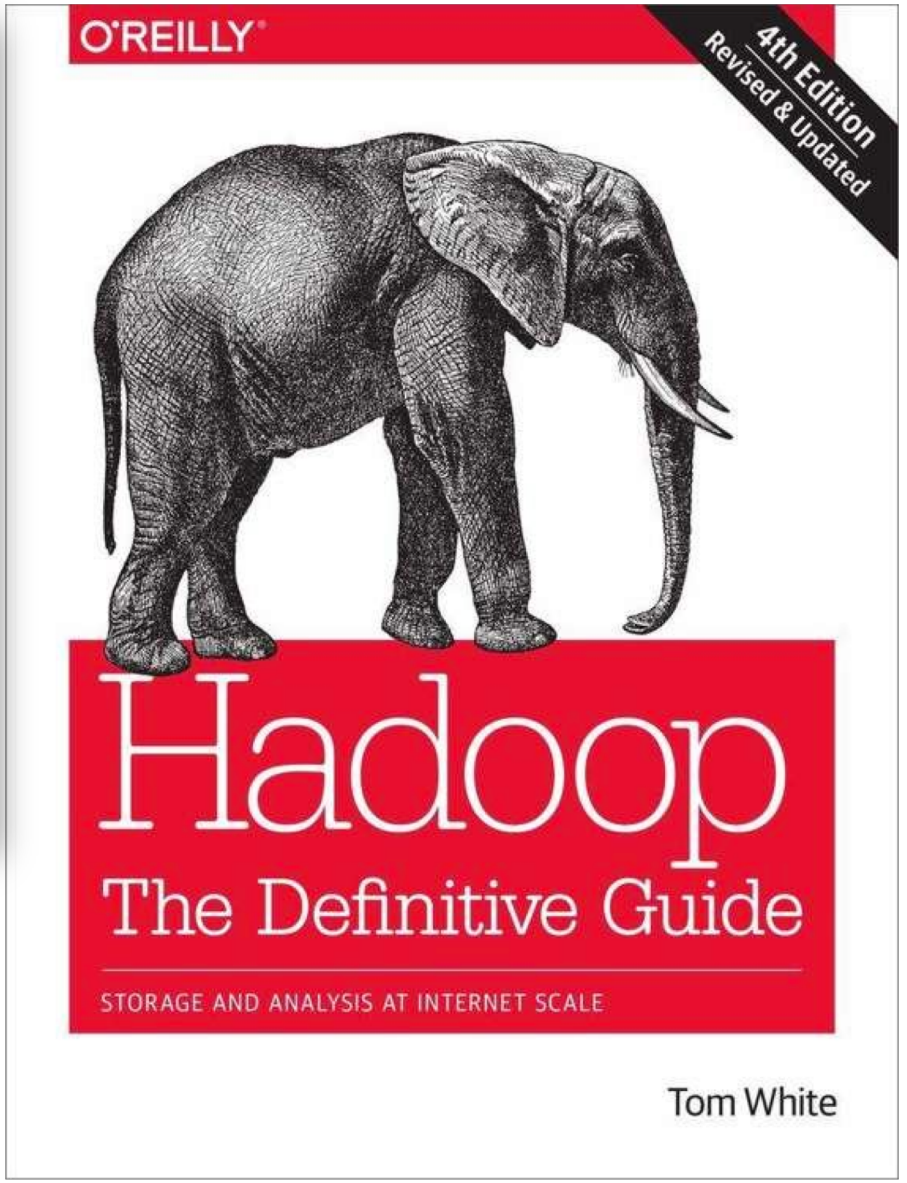
Outline

- (GFS/)HDFS
- Partitioner
- Shuffle and Sort
- Set up and Clean
- Counters
- Job Scheduling

Recommended Reading

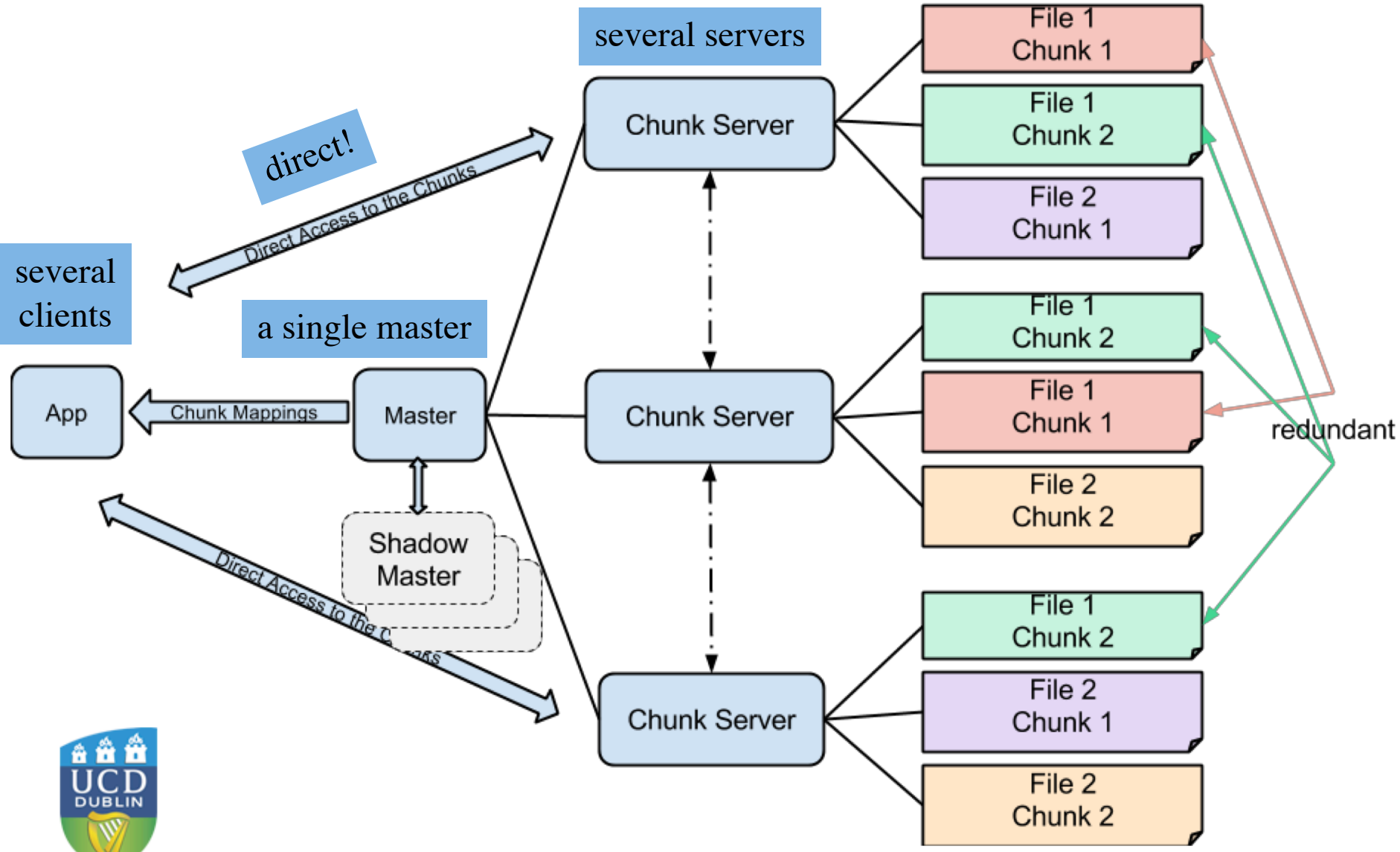


https://lintool.github.io/MapReduceAlgorithms/

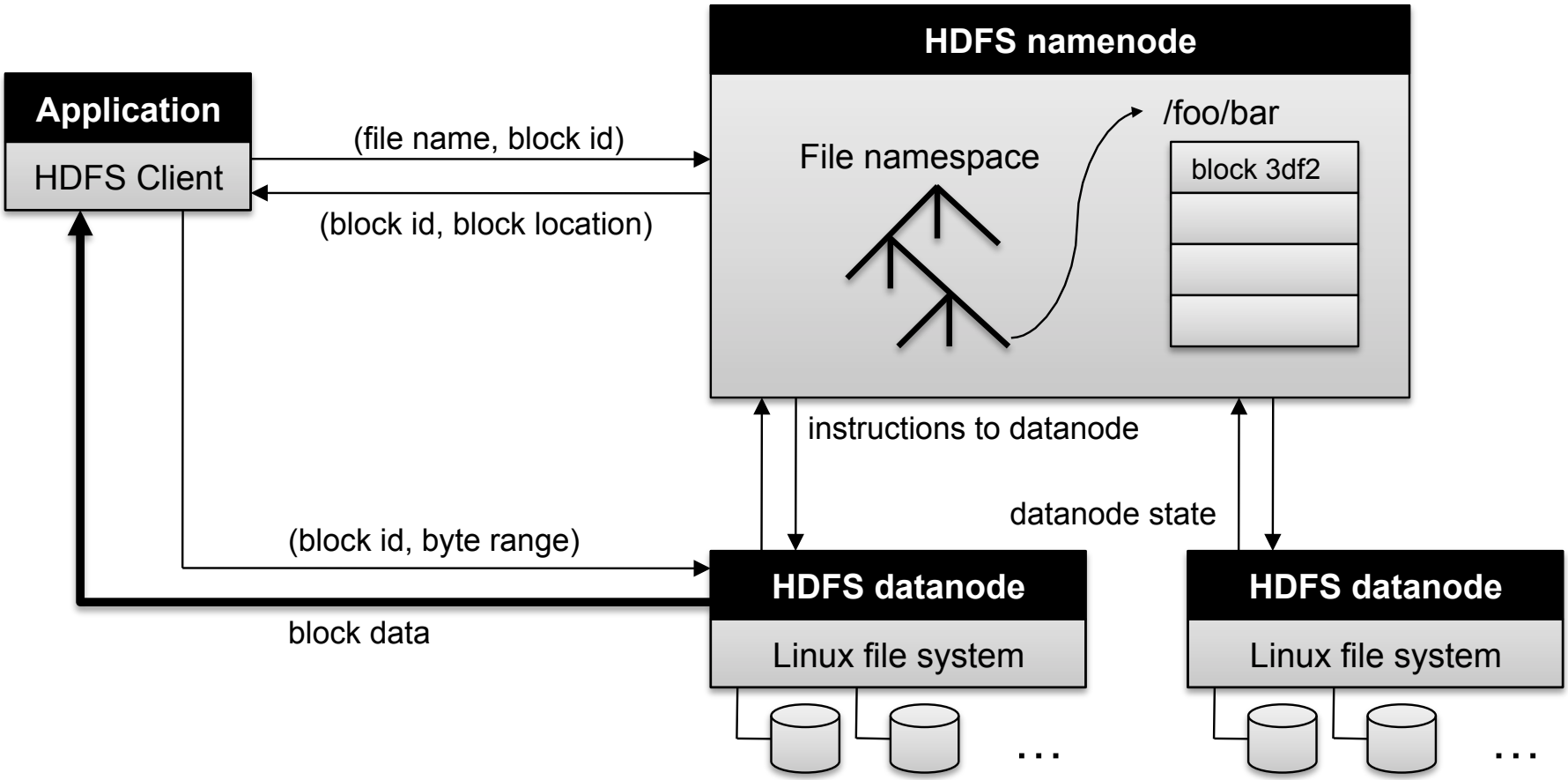


UCD DUBLIN

GFS Architecture

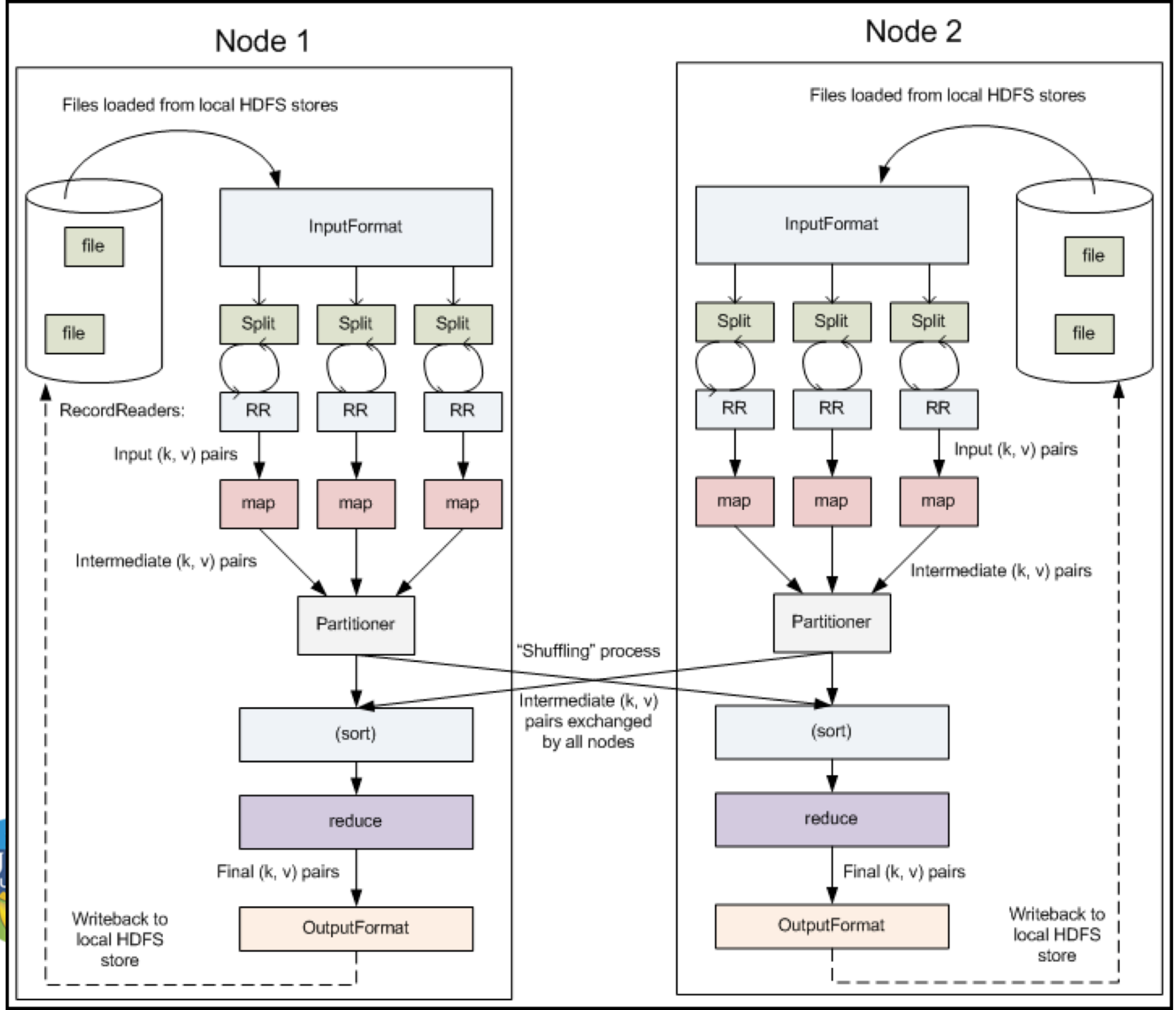


HDFS Architecture



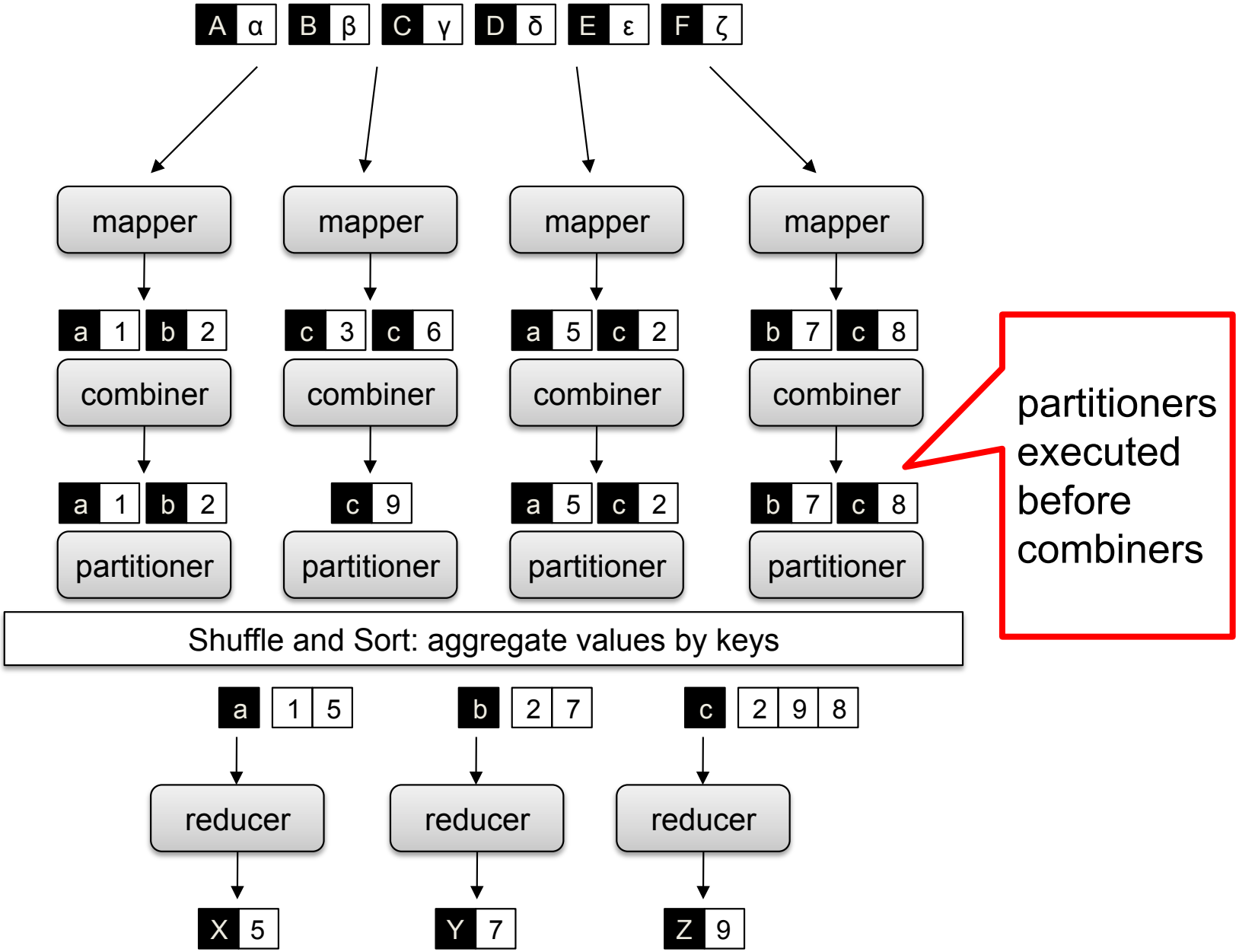
GFS vs. HDFS

GFS	HDFS
Master	NameNode
chunkserver	DataNode
operation log	journal, edit log
chunk	block
random file writes possible	***only append is possible***
multiple writer, multiple reader model	single writer, multiple reader model
chunk: 64KB data and 32bit checksum pieces	per HDFS block, two files created on a DataNode: data file & metadata file (checksums, timestamp)
default block size: 64MB	default block size: 128MB


Node 1
Files loaded from local HDFS stores
file
file
InputFormat
Split
Split
Split
RecordReaders:
RR
RR
RR
Input (k, v) pairs
map
map
map
Intermediate (k, v) pairs
Partitioner
"Shuffling" process
Intermediate (k, v) pairs exchanged by all nodes
(sort)
reduce
Final (k, v) pairs
Writeback to local HDFS store
OutputFormat
Node 2
Files loaded from local HDFS stores
InputFormat
file
file
Split
Split
Split
RR
RR
RR
Input (k, v) pairs
map
map
map
Intermediate (k, v) pairs
Partitioner
(sort)
reduce
Final (k, v) pairs
OutputFormat
Writeback to local HDFS store

Partitioner

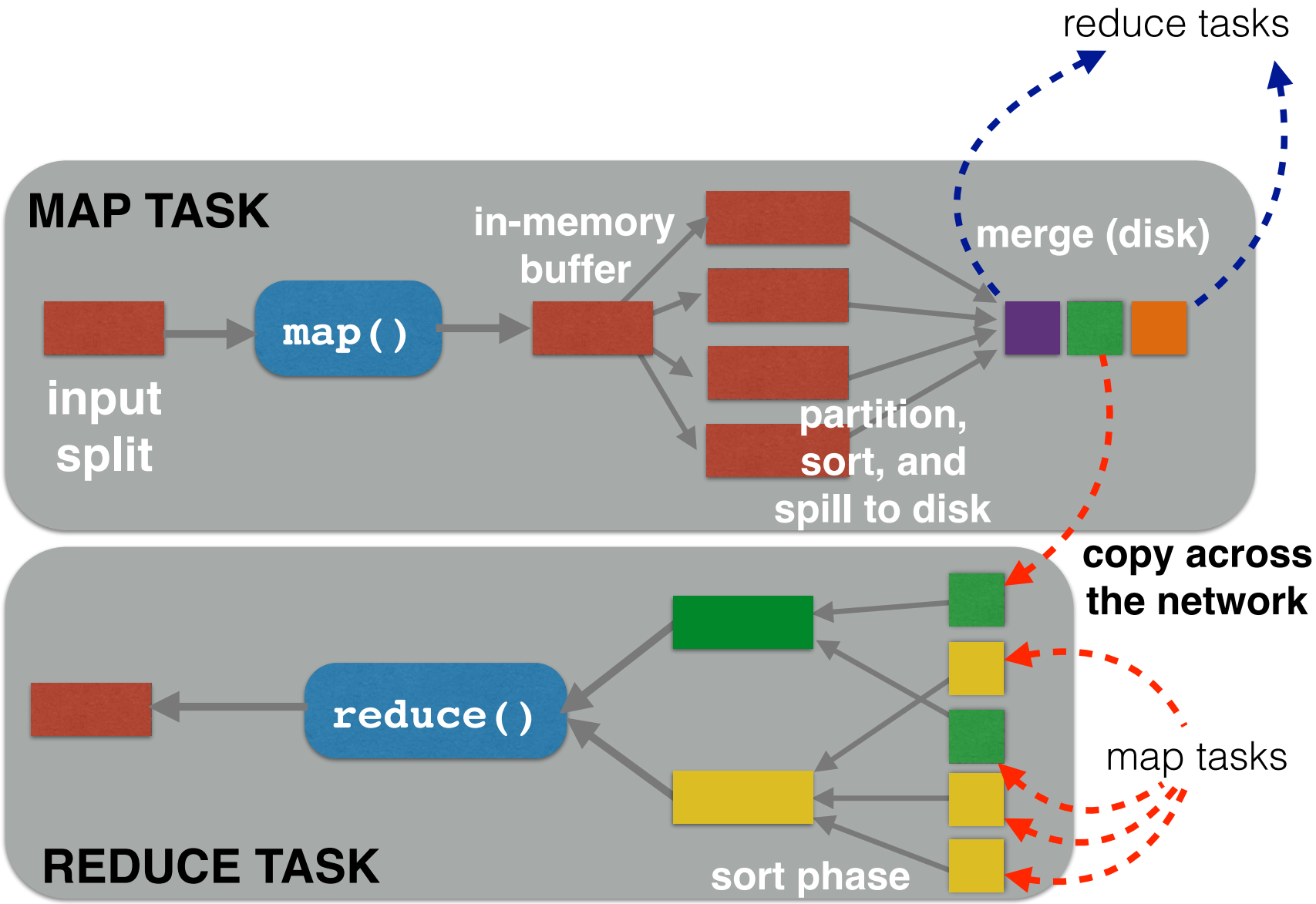
- Responsible for ***dividing*** the ***intermediate key space*** and assigning intermediate key/value pairs to reducers
- Within each ***reducer***, keys are processed in sorted order (i.e. several keys can be assigned to a reducer)
  - All values associated with a ***single key*** are processed in a ***single reduce()*** call


A α
B β
C γ
D δ
E ε
F ζ
mapper
mapper
mapper
mapper
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
combiner
combiner
combiner
combiner
a 1 b 2
c 9
a 5 c 2
b 7 c 8
partitioner
partitioner
partitioner
partitioner
partitioners executed before combiners
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 9 8
reducer
reducer
reducer
X 5
Y 7
Z 9

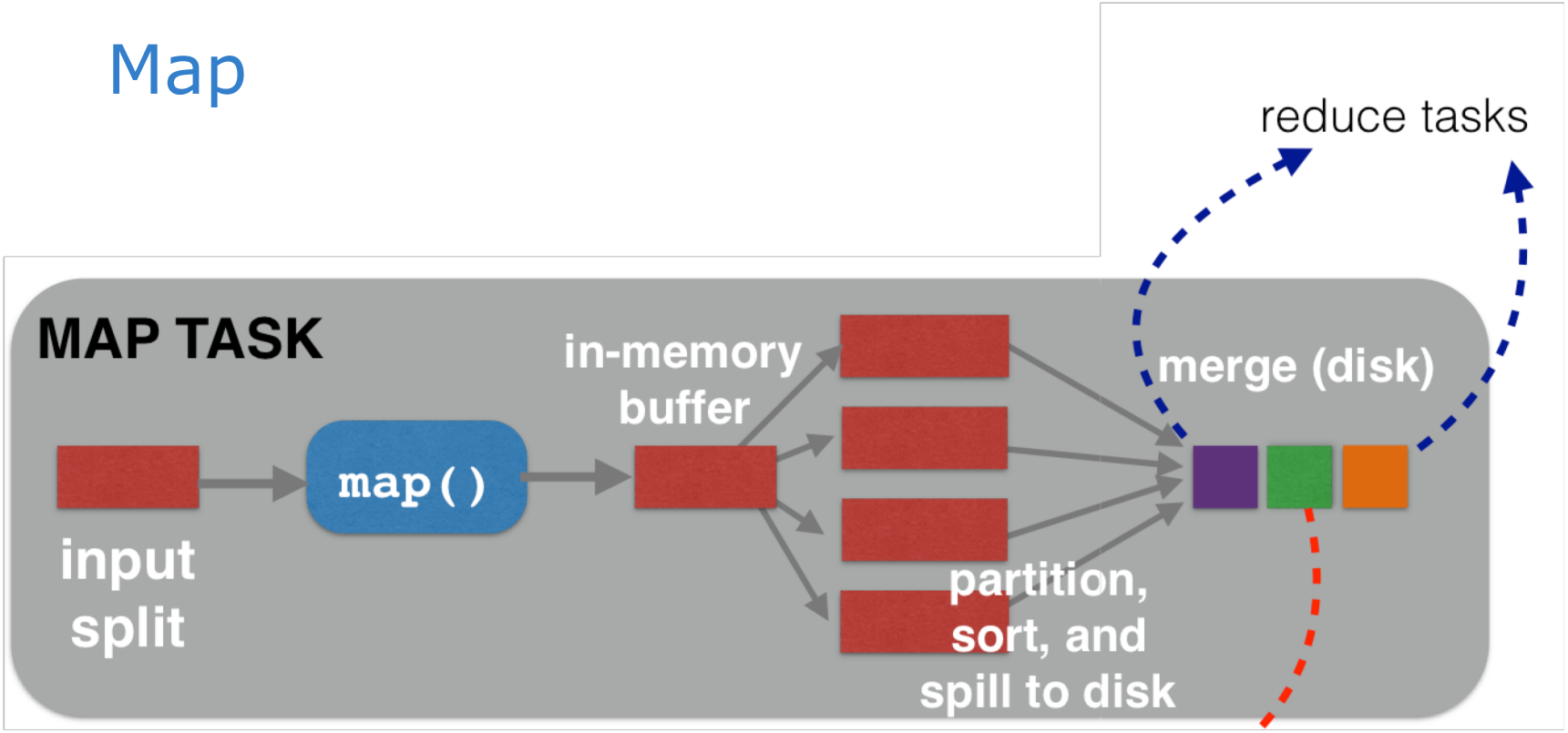
Shuffle and Sort

- ***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

A High Level View



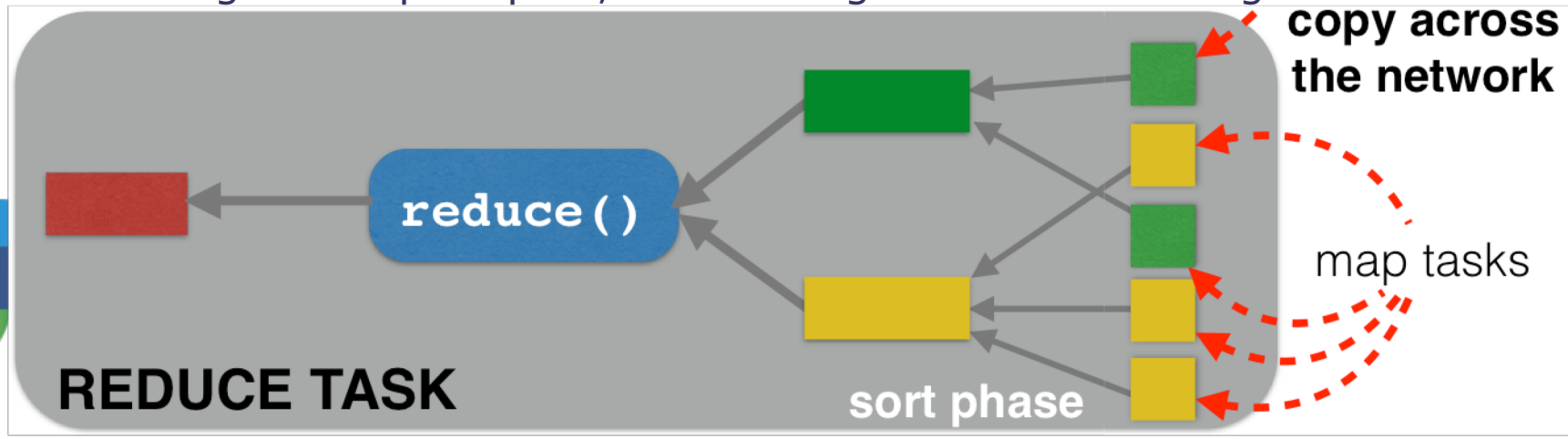
Map



- 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

- 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
- Once **all** intermediate keys are copied the "sort phase" begins: merge of map outputs, maintaining their sort ordering



Sort Phase

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

Setup & Clean

- 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
- 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: Count only "Valid" Terms

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

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

    public void setup(Context context) […]{
         dictionary = new Hashset()
         //load the dictionary in some way
    }

    public void map(Object key, Text value, Context context) […] {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
         String tmpKey = itr.nextToken();
         if(!dictionary.contains(tmpKey) {
                  word.set(tmpKey);
                  context.write(word, one);
         }
      }
}
```

called once: before any call to map()

Cleanup: How Many Words Start with the Same Letter?

```
public class MyReducer extends Reducer<…> {
        privateMap<Character,Integer> cache;
        public void setup(Contextcontext)[…]{
                cache = new HashMap();
        }
        public void reduce(PairOfIntString key,
                        Iterable<IntWritable> values, Context context) […] {
                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) […] {
                for (Character c : cache.keySet()) {
                        context.write(new Text(c), new
IntWritable(cache.get(c));
                }
        }
}
```

called once: before any call to reduce()

called once: after all calls to reduce()

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

- 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)

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
  - Aggregated per job by the ResourceManager (Yarn)

Example: Word Count

```
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());
      }
    }
  }
}
```

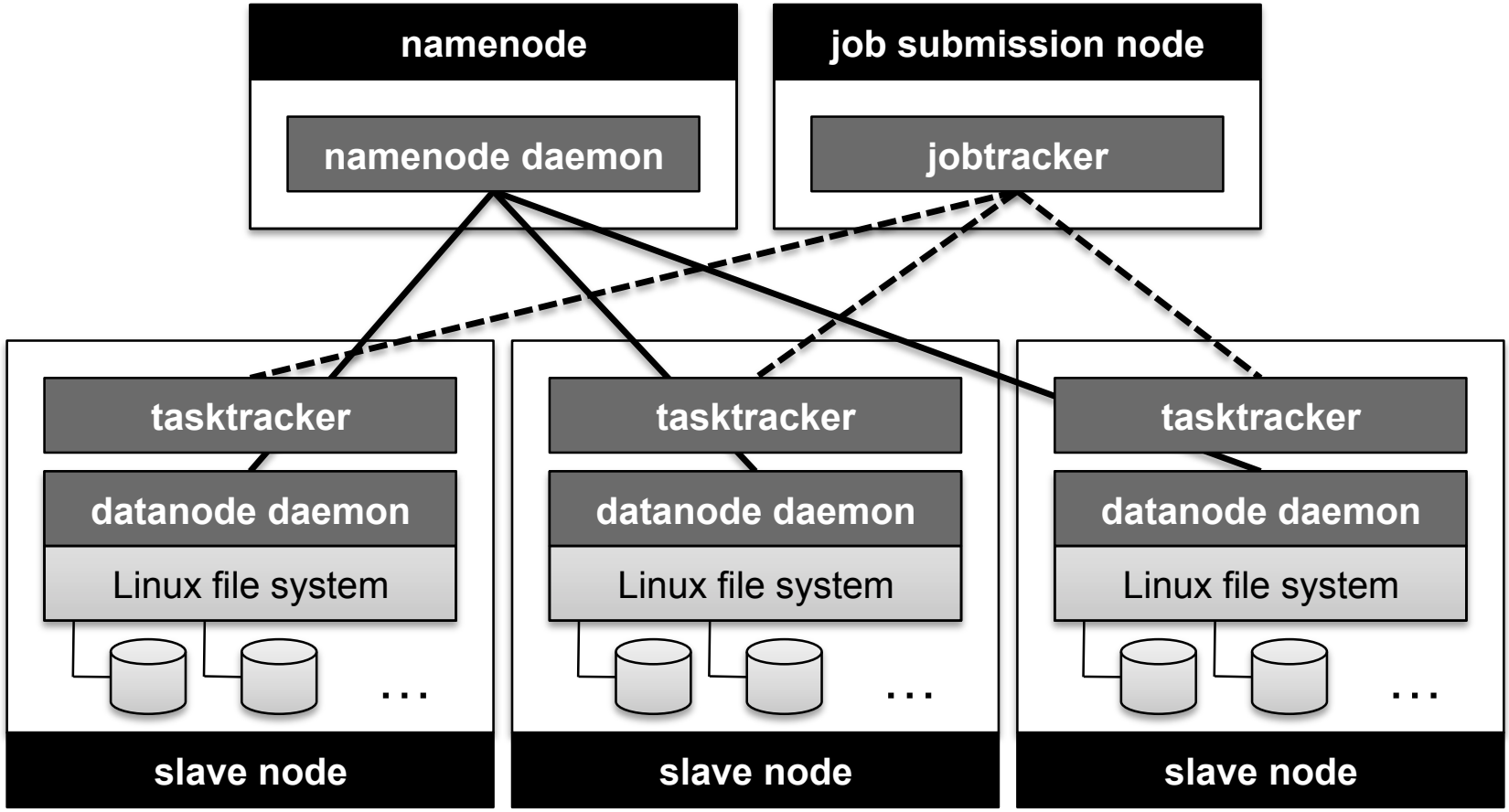
Output

```
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
```

Hadoop 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



Hadoop in Practice (Yahoo! 2010)

- **3,500 nodes** in the cluster
- **40 nodes/rack** sharing one IP switch
- **16GB RAM** per cluster node, 1-gigabit Ethernet
- **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**

Shvachko, K., Kuang, H., Radia, S. and Chansler, R., 2010, May. The hadoop distributed file system. In *Mass storage systems and technologies (MSST)* pp. 1-10.

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

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

- 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 resource than 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 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***