MapReduce

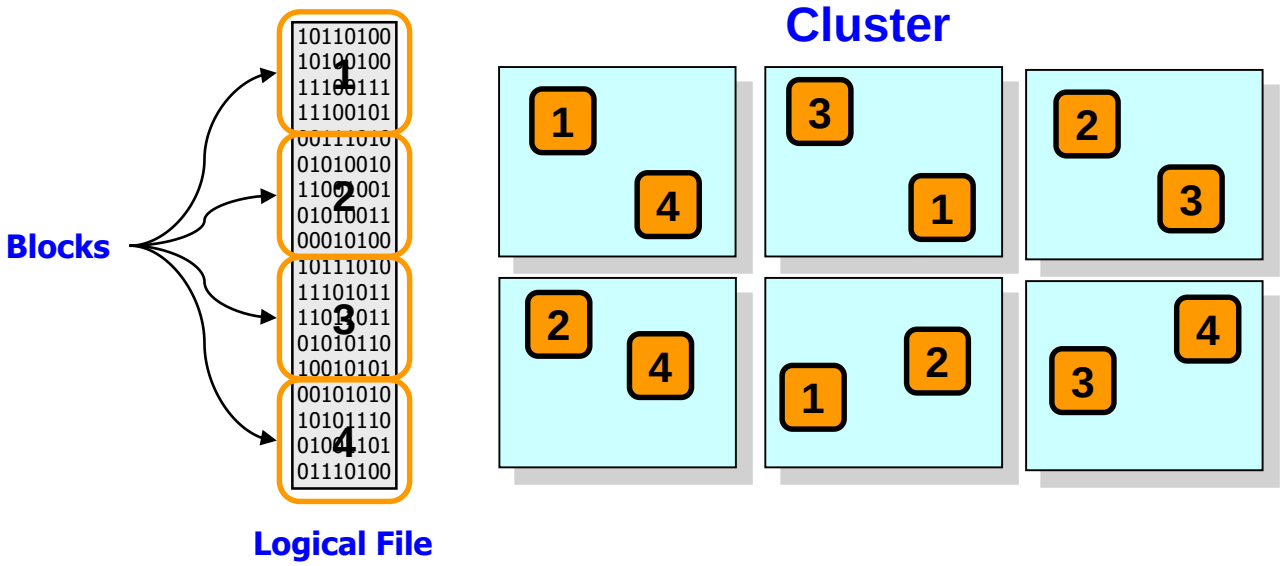
Agenda

- MapReduce Introduction
- MapReduce Tasks
- WordCount Example
- Splits
- Execution
- Scheduling

Introduction to MapReduce

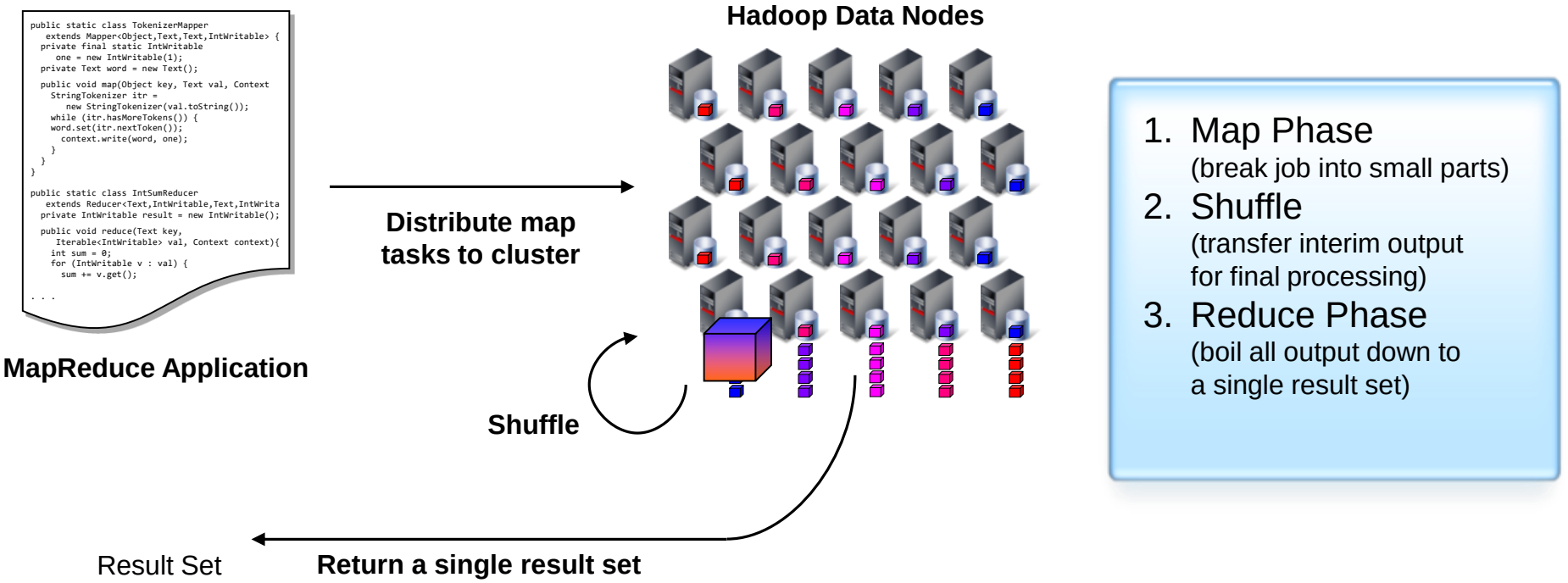
- **Driving principals**
  - Data is stored across the entire cluster
  - Programs are brought to the data, not the data to the program

- **Data is stored across the entire cluster (the DFS)**
  - The entire cluster participates in the file system
  - Blocks of a single file are distributed across the cluster
  - A given block is typically replicated as well for resiliency



MapReduce Explained

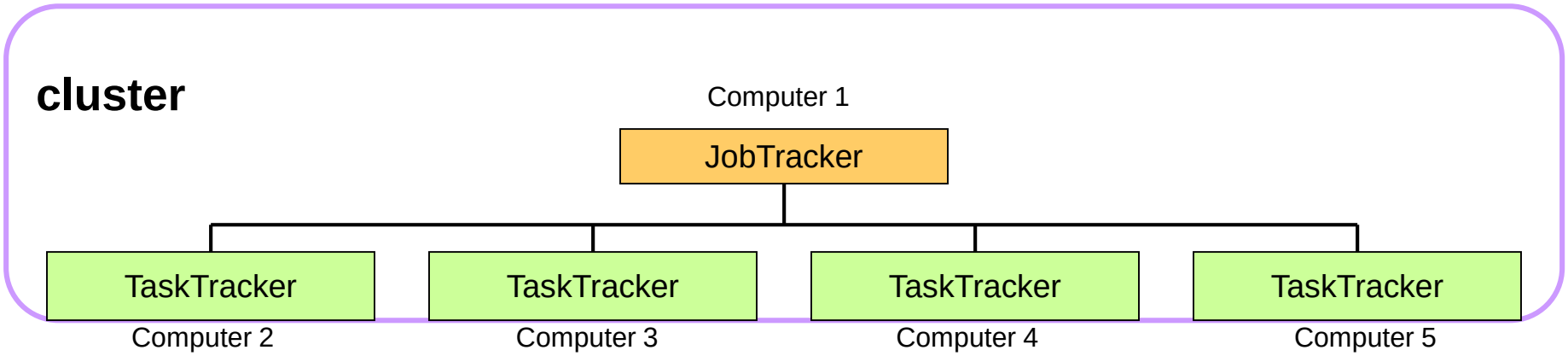
- Hadoop computation model
  - Data stored in a distributed file system spanning many inexpensive computers
  - Bring function to the data
  - Distribute application to the compute resources where the data is stored
- Scalable to thousands of nodes and petabytes of data



1. Map Phase
   (break job into small parts)
2. Shuffle
   (transfer interim output for final processing)
3. Reduce Phase
   (boil all output down to a single result set)

MapReduce Engine

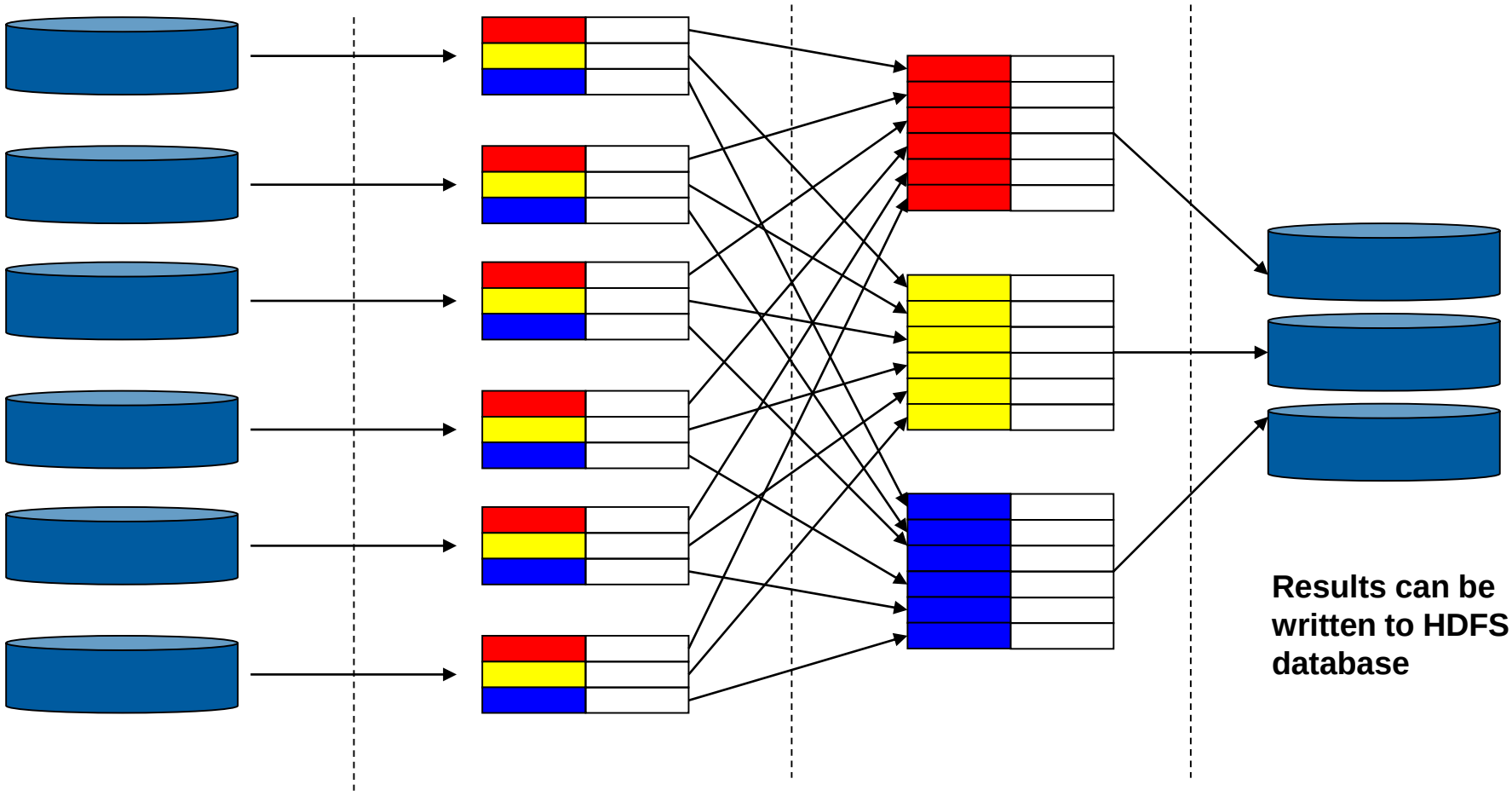
- Master / Slave architecture
  - Single master (JobTracker) controls job execution on multiple slaves (TaskTrackers).
- **JobTracker**
  - Accepts MapReduce jobs submitted by clients
  - Pushes *map* and *reduce* tasks out to TaskTracker nodes
  - Keeps the work as physically close to data as possible
  - Monitors tasks and TaskTracker status
- **TaskTracker**
  - Runs map and reduce tasks
  - Reports status to JobTracker
  - Manages storage and transmission of intermediate output



The MapReduce Programming Model

- **"Map" step:**
  - Input split into pieces
  - Worker nodes process individual pieces in parallel (under global control of the Job Tracker node)
  - Each worker node stores its result in its local file system where a reducer is able to access it
- **"Reduce" step:**
  - Data is aggregated (‘reduced” from the map steps) by worker nodes (under control of the Job Tracker)
  - Multiple reduce tasks can parallelize the aggregation

MapReduce Overview

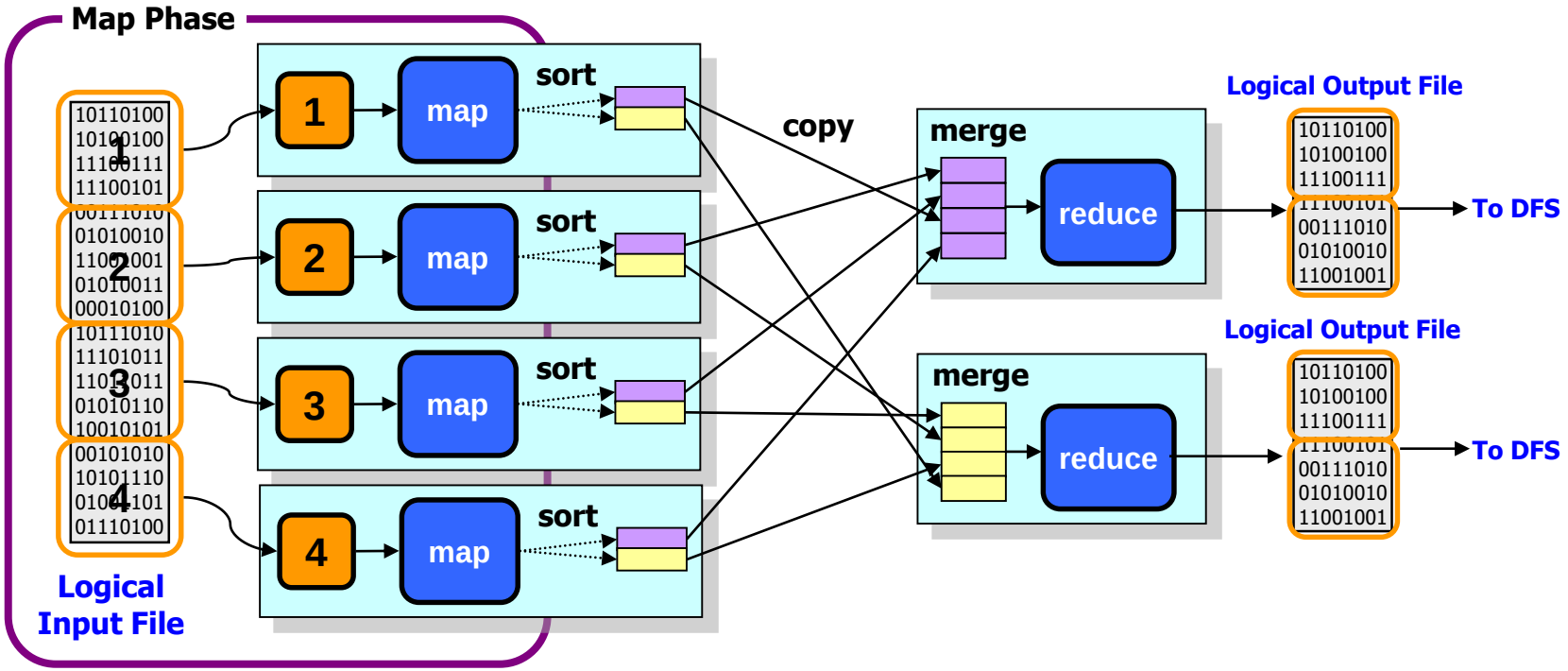


Agenda

- MapReduce Introduction
- MapReduce Tasks
  - Map
  - Shuffle
  - Reduce
  - Combiner
- WordCount Example
- Splits
- Execution
- Scheduling

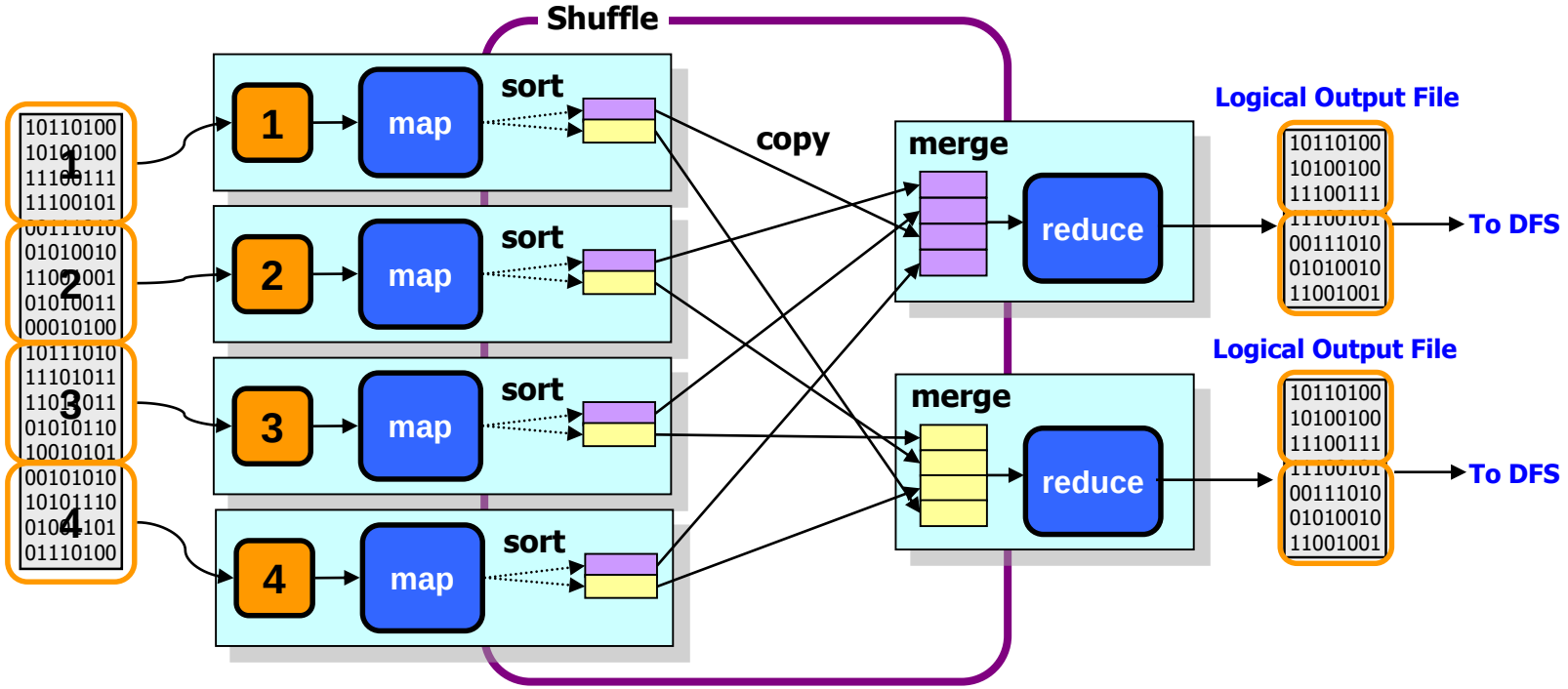
MapReduce – Map Phase

- **Mappers**
  - Small program (typically), distributed across the cluster, local to data
  - Handed a *portion* of the input data (called a split)
  - Each mapper parses, filters, or transforms its input
  - Produces grouped `<key,value>` pairs



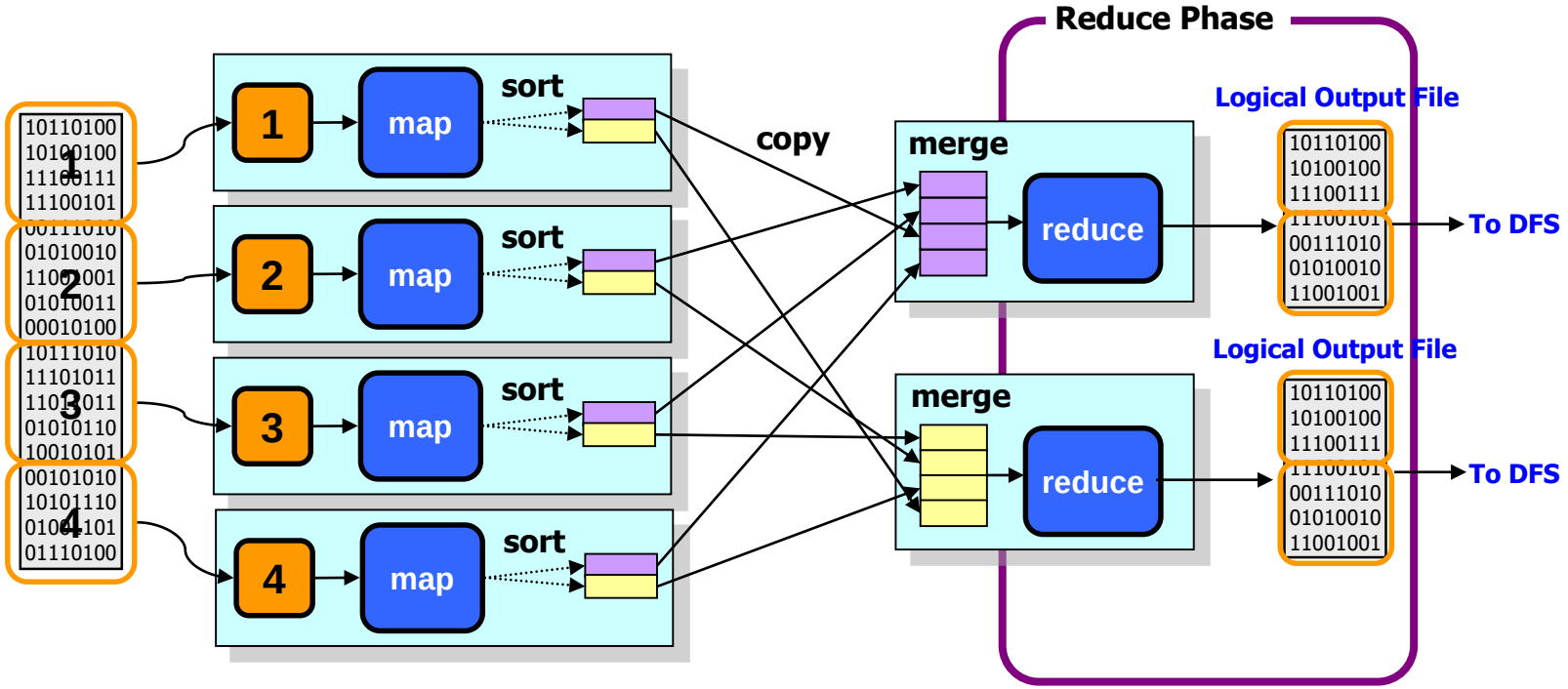
MapReduce – The Shuffle

- The output of each mapper is locally grouped together by `key`
- One node is chosen to process data for each unique `key`
- All of this movement (shuffle) of data is transparently orchestrated by MapReduce



MapReduce – Reduce Phase

- **Reducers**
  - Small programs (typically) that aggregate all of the `values` for the `key` that they are responsible for
  - Each reducer writes output to its own file



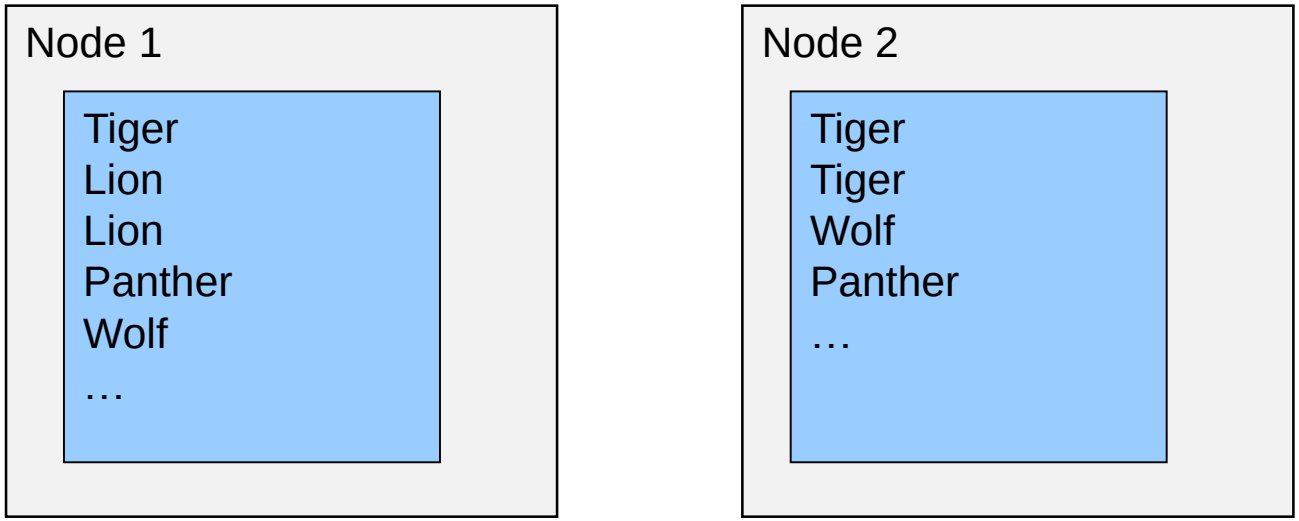
Agenda

- MapReduce Introduction
- MapReduce Tasks
- WordCount Example
- Splits
- Execution
- Scheduling

Word Count Example

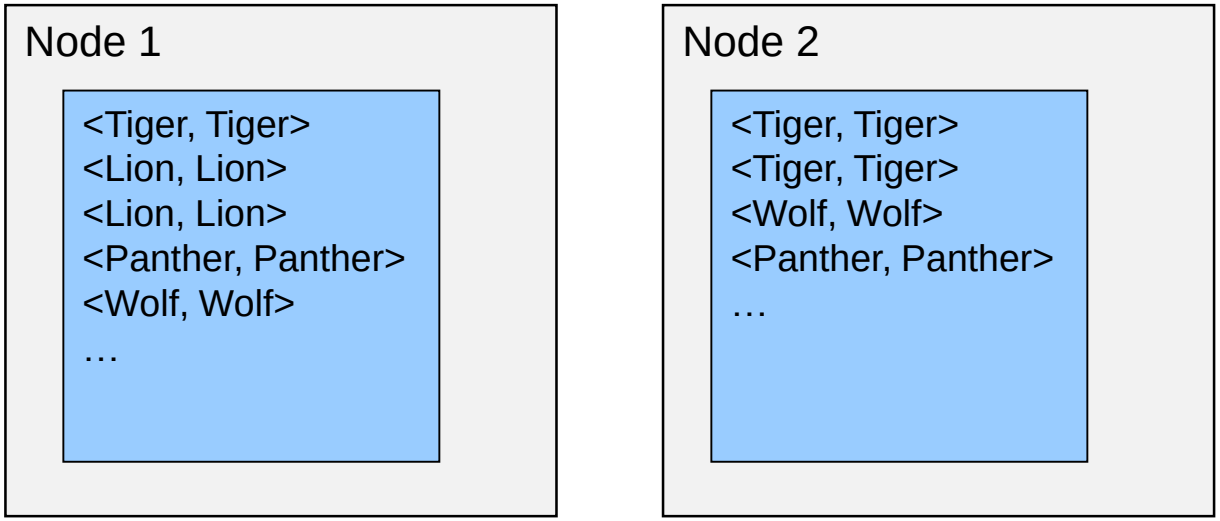
- In this example we have a list of animal names
  - MapReduce can automatically split files on line breaks
  - Our file has been split into two blocks on two nodes
- We want to count how often each big cat is mentioned. In SQL that would be:

```
SELECT COUNT(NAME) FROM ANIMALS
       WHERE NAME IN (Tiger, Lion …)
       GROUP BY NAME;
```



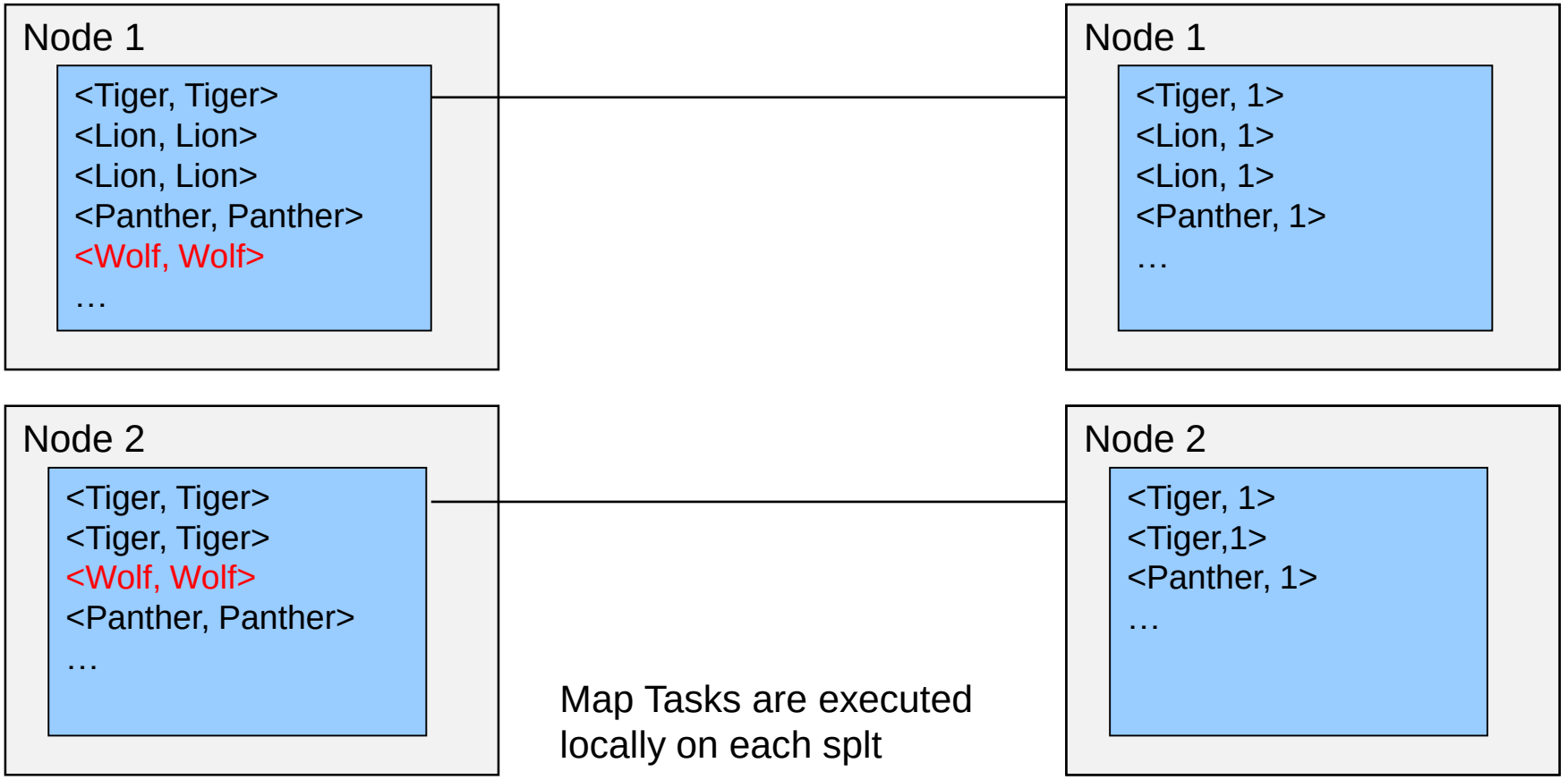
Map Input

- Map Tasks need Key and Value pairs as input
- If no key is available it needs to be fabricated
- The mapping from input (files, web link, …) to
  <key, value> pairs is done in the InputFormat class



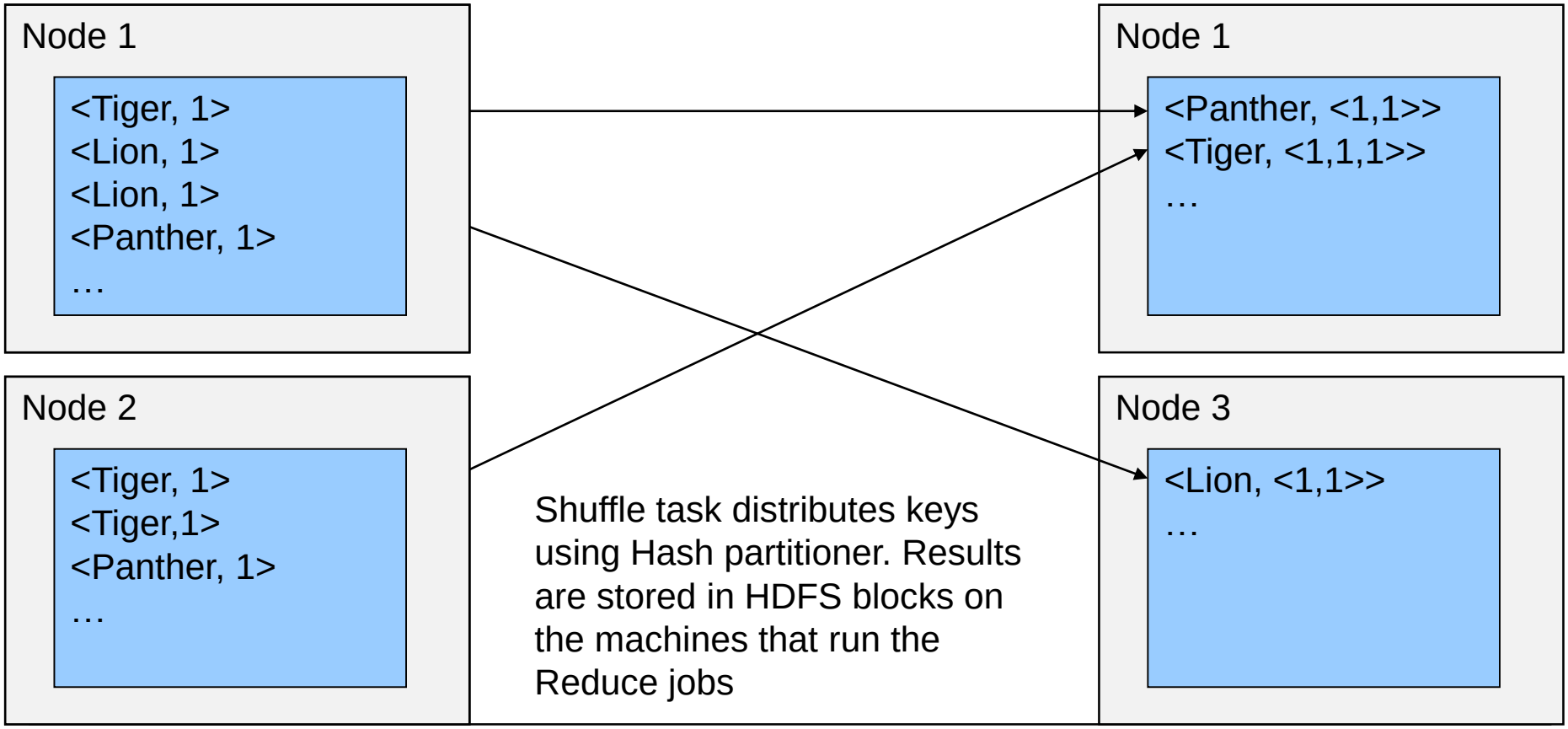
Map Task

- We have two tasks in our map task
  - Filter non big cat rows
  - Prepare count by transforming to
    - <Text(name), Integer(1)>



Shuffle

- Shuffle moves all values of one key to the same target node
- Distribution is done through a Partitioner Class (normally hash distribution)
- Reduce Tasks can run on arbitrary nodes, in our example Node 1 and 3
  - The number of Map and Reduce tasks do not need to be identical
  - Differences are handled by hash partitioner



Persisting Data in Shuffle

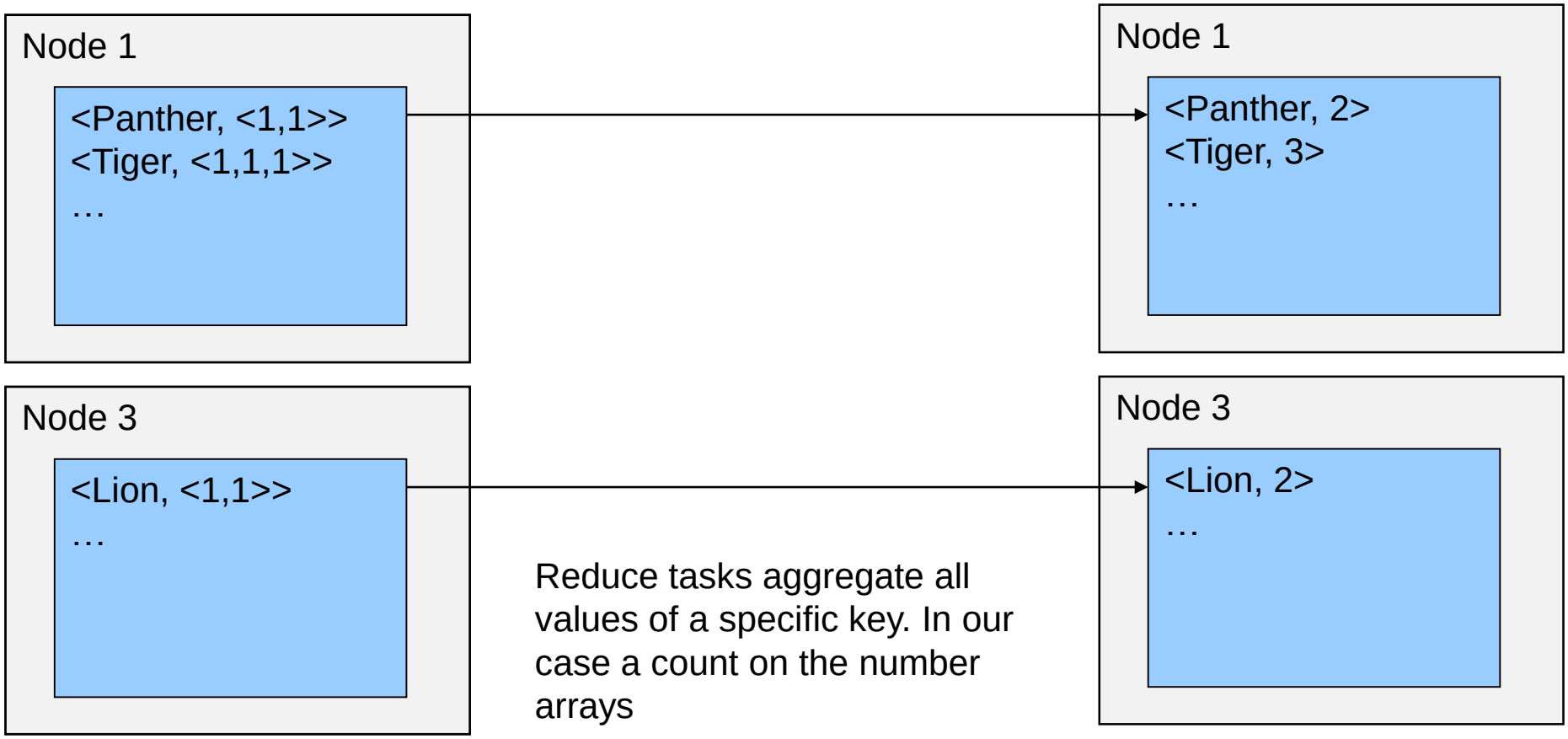
- Data is persisted in DFS in Shuffle phase
  - Allows Map and Reduce tasks to be restarted

- Multiple Map outputs need to be merged into a single Reducer Input

- Merging is done in Memory
  - Regularly persisted in DFS
  - Sort memory for merge can be configured
  - Using a merge sort operation of the various input files

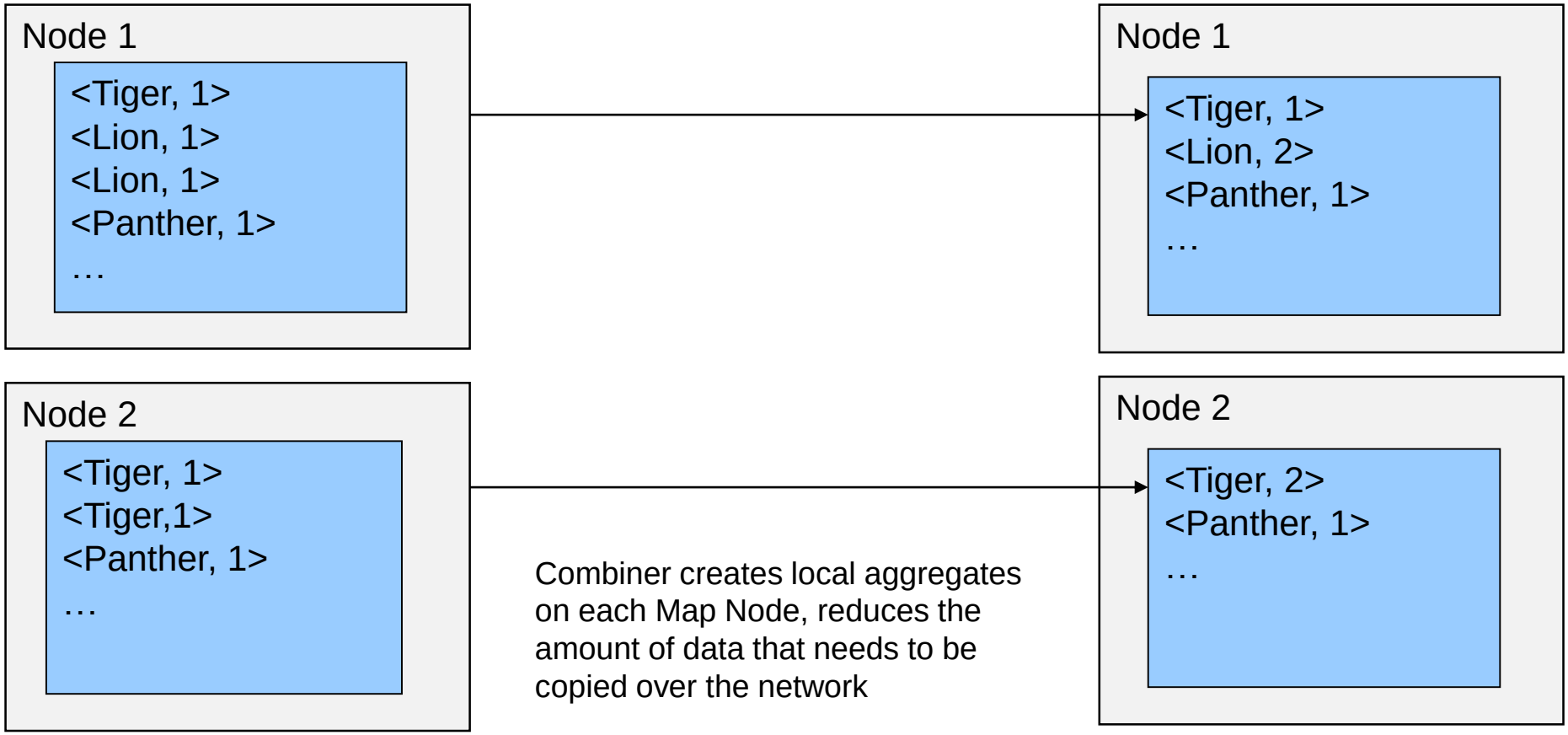
Reduce

- The reduce task computes aggregated values for each key
  - Normally the output is written to DFS
  - Per default one output part file per Reduce task



Optional: Combiner

- For performance reasons a local Aggregate in the Map task can be helpful
- Reduces the amount of data that needs to be copied over the network
  - Also reduces Merge effort
- After Map task and before Shuffle



Map/Reduce tasks

- **Local Execution**
  - Hadoop will attempt to execute splits locally
  - If no local Map slot is available split will be moved to the Map task
- **Number Map Tasks**
  - It is possible to configure the number of Map and Reduce tasks
  - If file is not splittable there will only be a single Map task
- **Number Reduce Tasks**
  - Normally there are less Reduce tasks than Map tasks
  - Reduce output is written locally to HDFS
  - If you need a single output task use one Reduce task
- **Redundant Execution**
  - It is possible to configure redundant execution, i.e. 2 or more Map tasks are started for each split
    - The first Map task for a split that finishes wins.
    - In systems with large numbers of machines and cheap machines this may increase performance
    - In systems with smaller number of nodes or high quality hardware it can decrease overall performance.

Agenda

- MapReduce Introduction
- MapReduce Tasks
- WordCount Example
- Splits
- Execution
- Scheduling

Splits

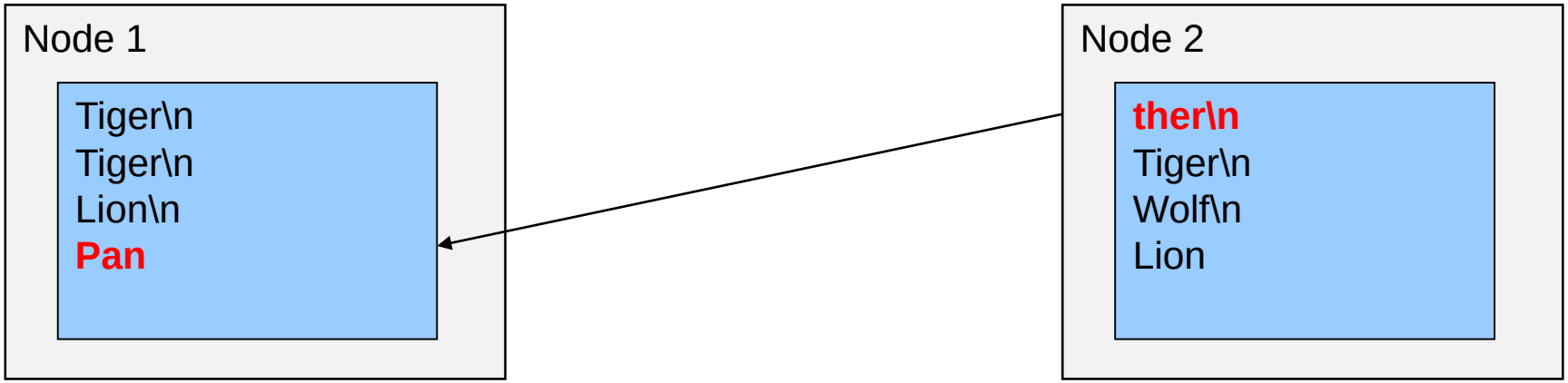
- Files in MapReduce are stored in Blocks (128 MB)
- MapReduce divides data into fragments or **splits**.
  - One map task is executed on each split
- Most files have records with defined **split points**
  - Most common is the end of line character
- The **`InputSplitter`** class is responsible for taking a HDFS file and transforming it into splits.
  - Aim is to process as much data as possible locally

Classes

- There are three main classes reading data in MapReduce:
  - **`InputSplitter`**, dividing a File into Splits
    - normally the block sizes but depends on number of requested Map tasks etc.
  - **`RecordReader`**, takes a split and reads the files into records
    - for example one record per line (**`LineRecordReader`**)
  - **`InputFormat`**, takes each record and transforms it into a <key, value> pair that is then forwarded to the Map task
- Lots of additional helper classes handling compression etc.
  - IBM provides additional compression handlers for lzo etc.

RecordReader

- Most of the time a Split will not happen at a block end
- Files are read into Records by the RecordReader class
  - Normally the RecordReader will start and stop at the split points.
- **`LineRecordReader`** will read over the end of the split till the line end.
  - HDFS will send the missing piece of the last record over the network
- Likewise the LineRecordReader of Block 2 will disregard the first incomplete line

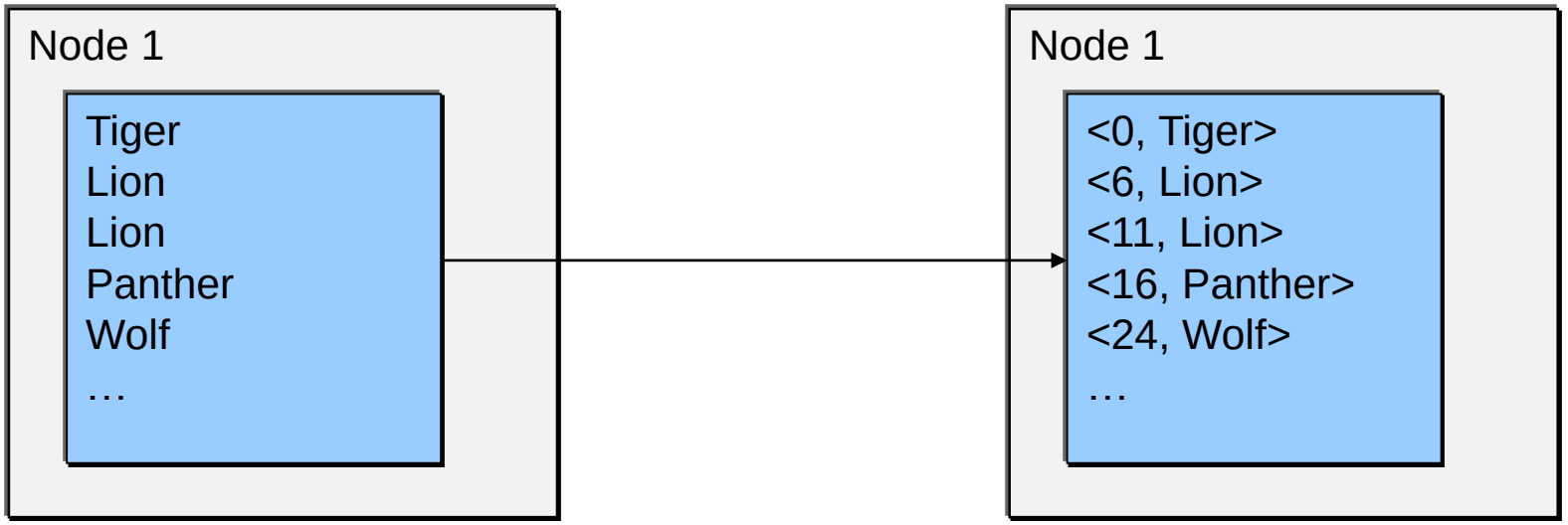


In our example RecordReader1 will not stop at “Pan” but will read on till the end of the line. Likewise RecordReader2 will ignore the first line

InputFormat

- MapReduce Tasks read files by defining an InputFormat class
  - Map tasks expect <key, value> pairs

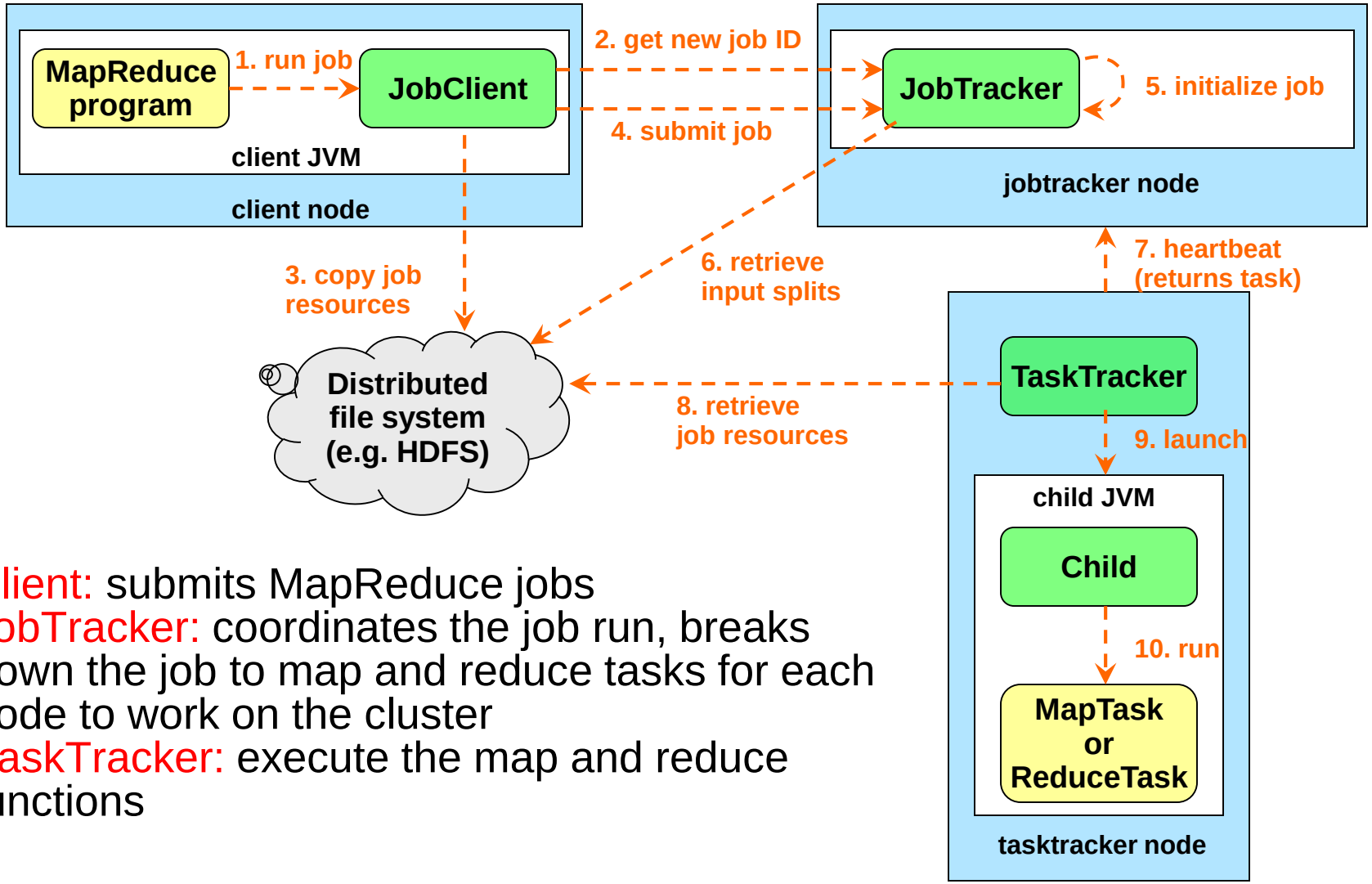
- To read line-delimited text files Hadoop provides the TextInputFormat class
  - It returns one key, value pair per line in the text
  - The value is the content of the line
  - The key is the offset to the new line character



Agenda

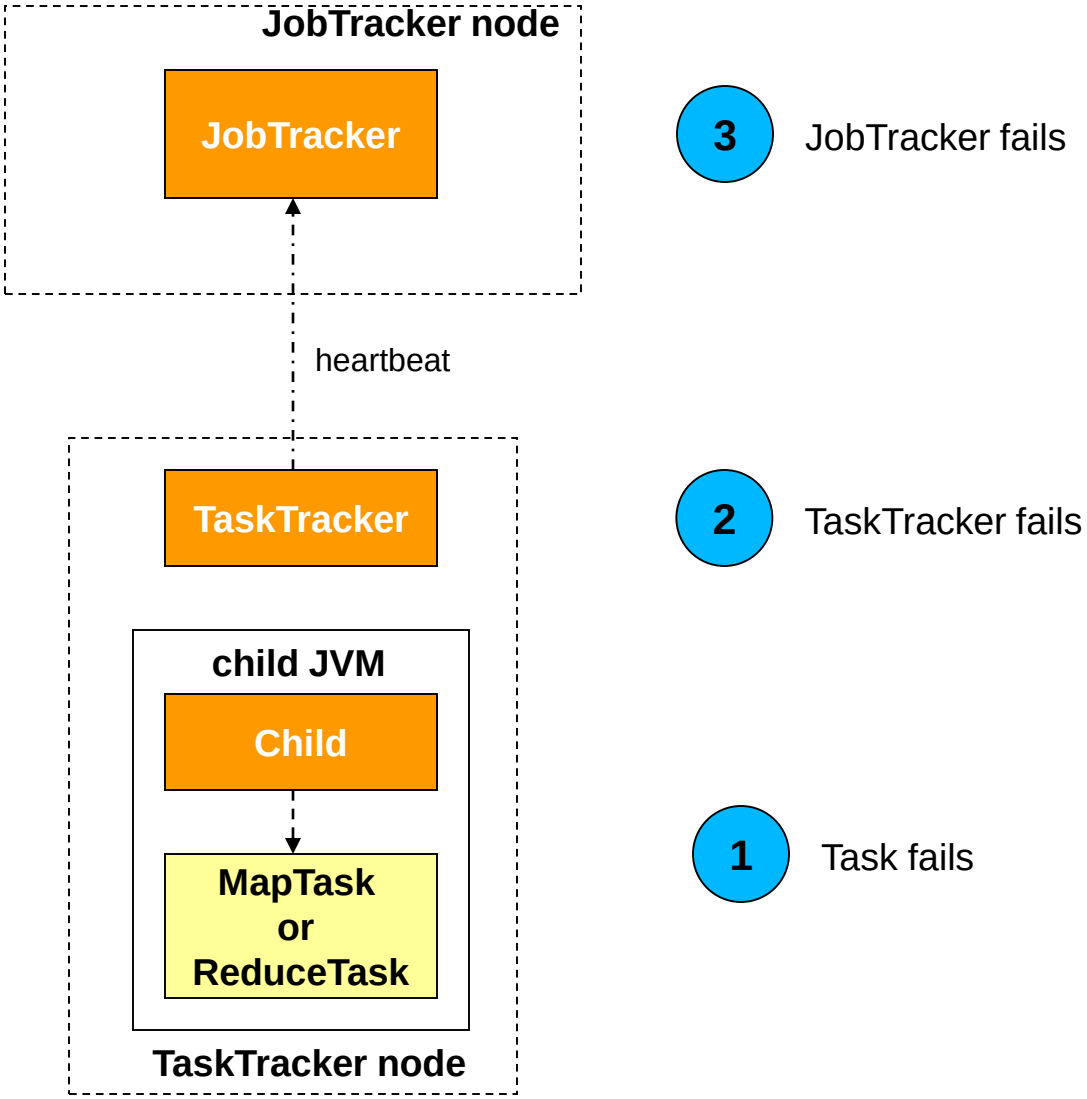
- MapReduce Introduction
- MapReduce Tasks
- WordCount Example
- Splits
- Execution
- Scheduling

How does Hadoop run MapReduce jobs?



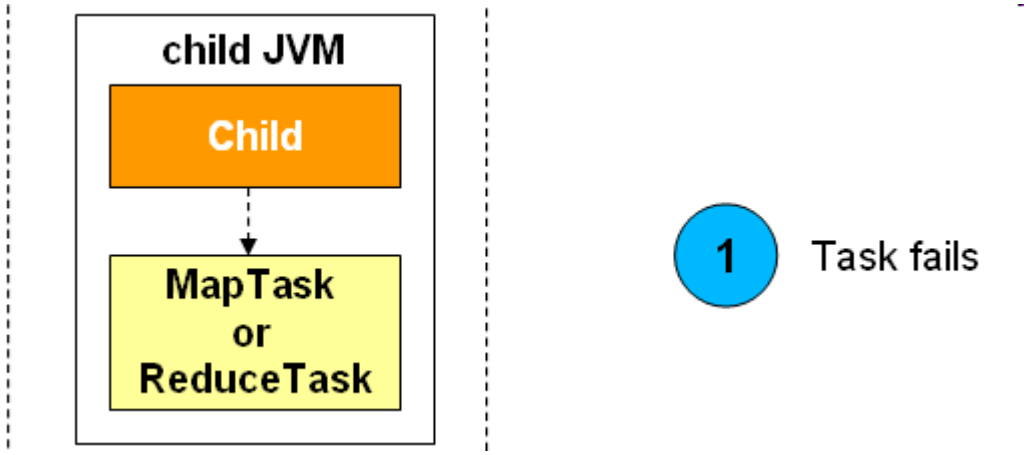
- Client: submits MapReduce jobs
- JobTracker: coordinates the job run, breaks down the job to map and reduce tasks for each node to work on the cluster
- TaskTracker: execute the map and reduce functions

Fault tolerance



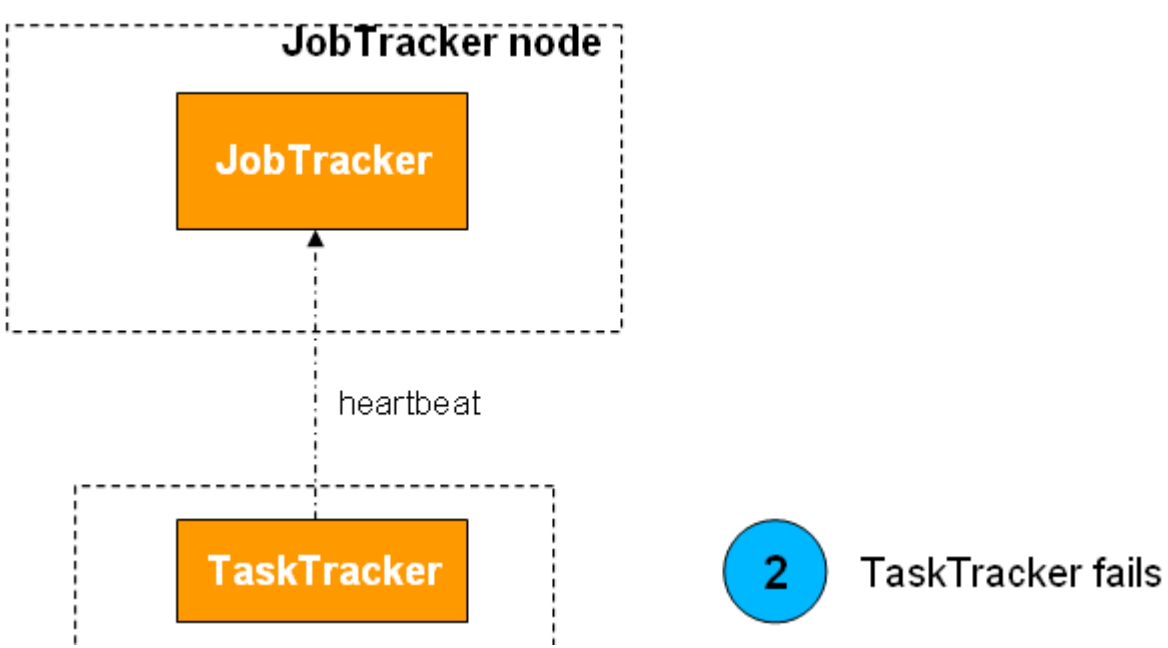
Fault tolerance

- **Task Failure**
  - If a child task fails, the child JVM reports to the TaskTracker before it exits. Attempt is marked failed, freeing up slot for another task.
  - If the child task hangs, it is killed. JobTracker reschedules the task on another machine.
  - If task continues to fail, job is failed.



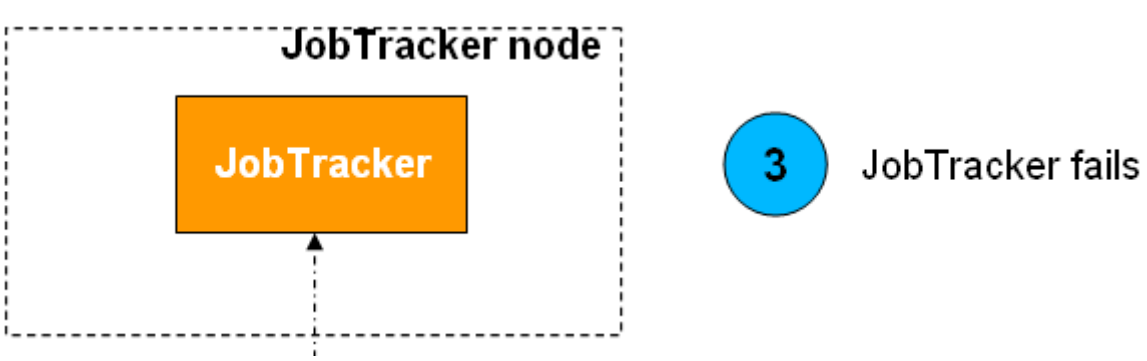
Fault tolerance

- **TaskTracker Failure**
  - JobTracker receives no heartbeat
  - Removes TaskTracker from pool of TaskTrackers to schedule tasks on.



Fault tolerance

- **JobTracker Failure**
  - Singe point of failure. Job fails



Configuration

- **`mapred-site.xml`**
  - Configuration settings for MapReduce (prefixed with mapred.)
  - These settings are global, some of the settings can be changed per Map Task

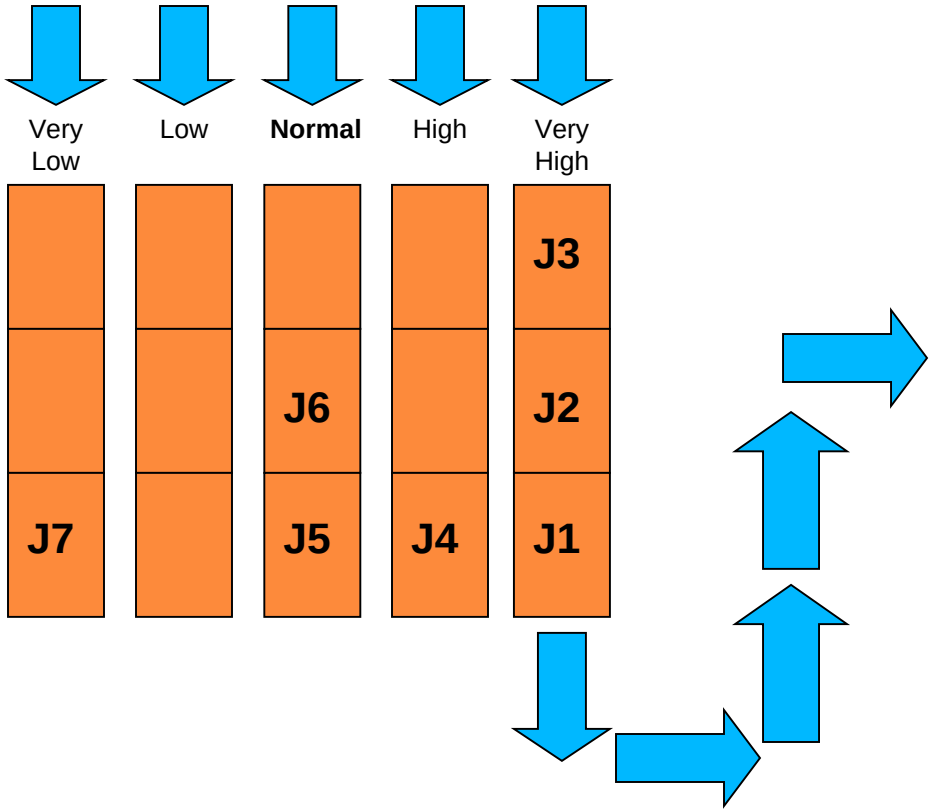
Parameter	Description
`jobtracker.taskScheduler`	**Scheduler used by JobTracker. In BigInsights changed to:** `com.ibm.biginsights.scheduler.WorkflowScheduler`
`tasktracker.map.tasks.maximum` `tasktracker.reduce.tasks.maximum`	**Maximum Number of Map/Reduce Tasks per task tracker. Set according to memory and CPUs in System**
`child.java.opts`	**Max. memory of JVM for each task**
`map.tasks.speculative.execution` `reduce.tasks.speculative.execution`	**Starts redundant tasks**
`io.sort.mb` `io.sort.factor`	**Parameters for merging map output in reducer (memory cache and number of files to merge at a time)**

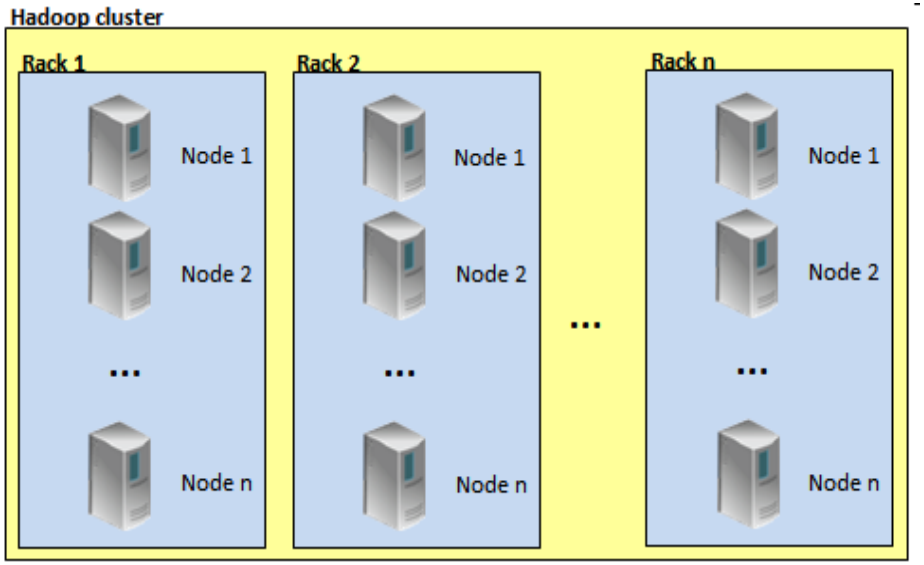
Agenda

- MapReduce Introduction
- MapReduce Tasks
- WordCount Example
- Splits
- Execution
- Scheduling
  - FIFO scheduler (with priorities)
  - Fair scheduler

Scheduling – FIFO scheduler (with priorities)

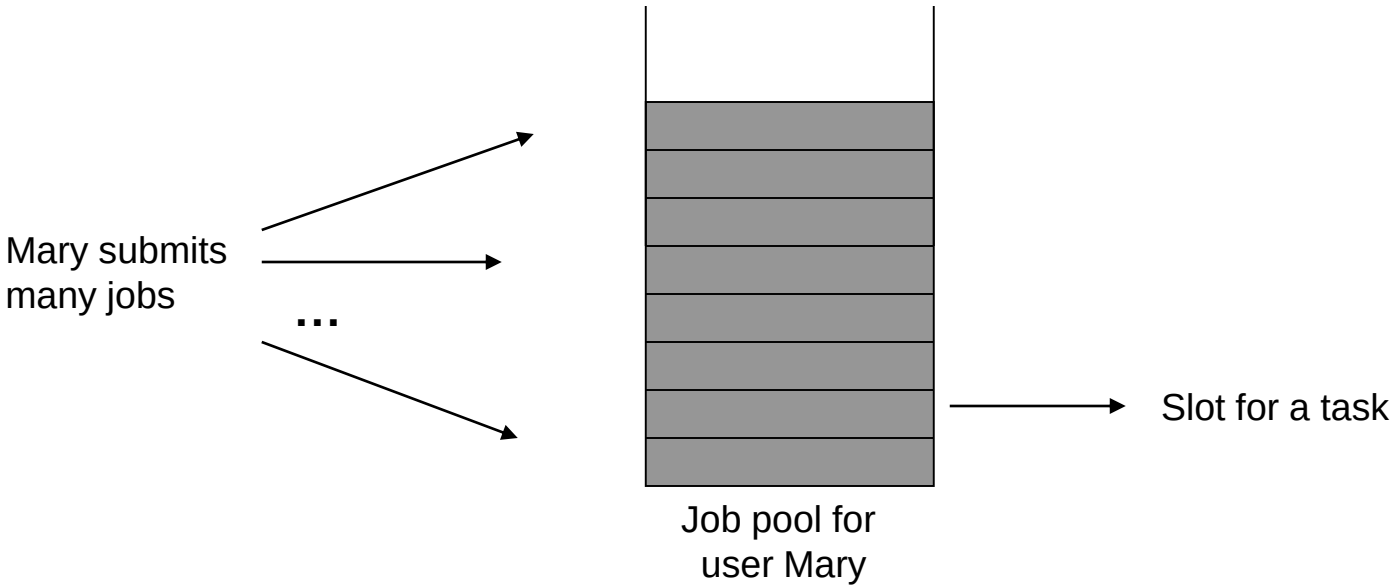
- Each job uses the whole cluster, so jobs wait their turn.
- Can set priorities for the jobs in the queue (5 queues with priorities)
- Preemption is not supported





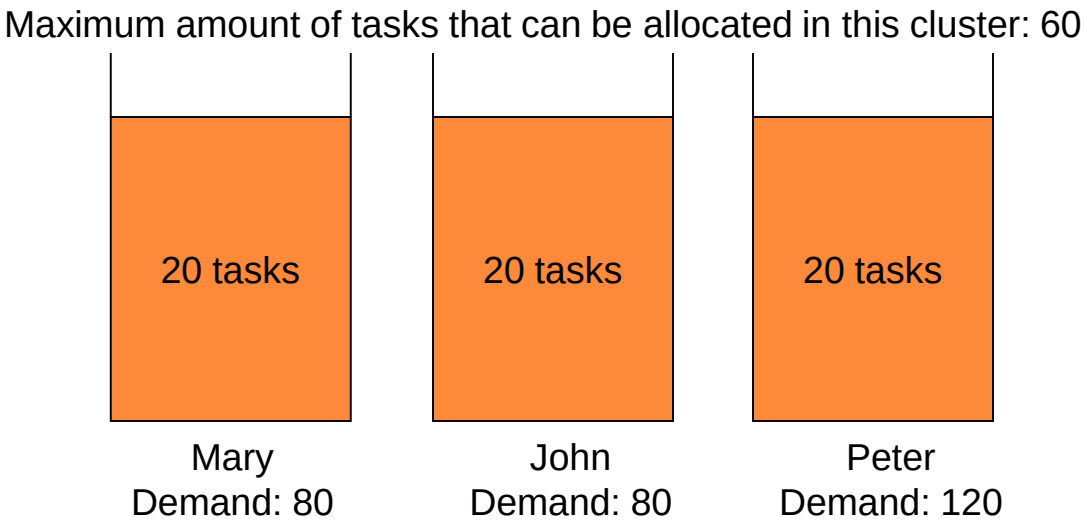
Scheduling – Fair scheduler

- Jobs are assigned to pools (1 pool per user by default)
- A user job pool is a number of slots assigned for tasks for that user
- Each pool gets the same number of task slots by default



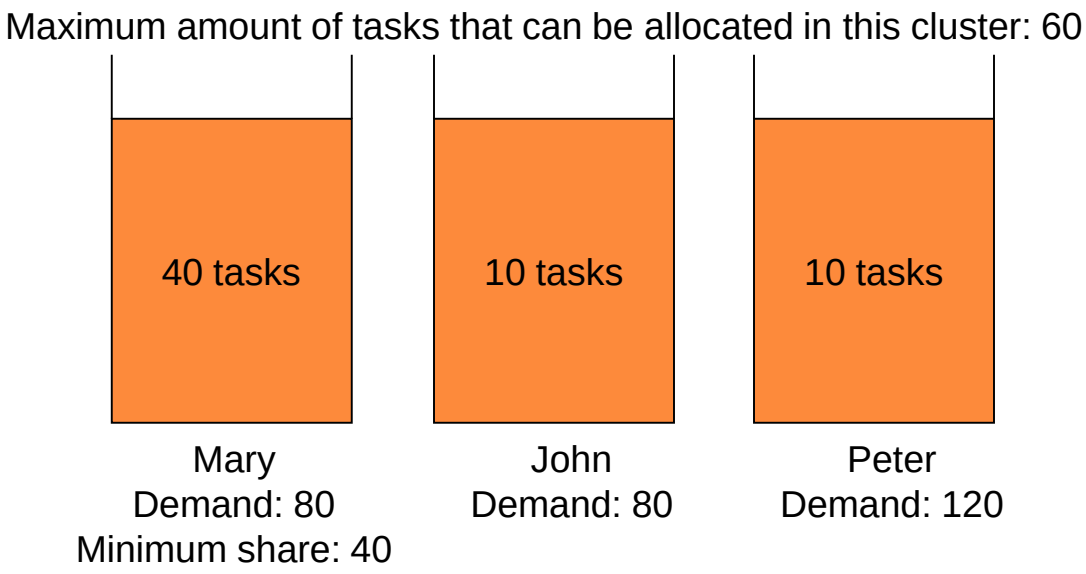
Scheduling – Fair scheduler

- Multiple users can run jobs on the cluster at the same time
- Example:
  - Mary, John, Peter submit jobs that demand 80, 80, and 120 tasks respectively
  - Say the cluster has a limit to allocate 60 tasks at most
  - Default behavior: Distribute task fairly among 3 users (each get 20)



Scheduling – Fair scheduler

- Minimum share can be set for a pool
- In the previous example, say Mary has a minimum share of 40
- Mary would be allocated 40, then the rest is distributed evenly to other pools



Questions?