Lecture slides of the course Introduction to Big Data

HADOOP FUNDAMENTALS

(Part III)

Lecturer: Dr. Nguyen Ngoc Thao – MSc. Le Ngoc Thanh

Department of Computer Science, FIT, HCMUS

Ho Chi Minh City, September 2018

Outline

- Introduction to Hadoop MapReduce
- Anatomy of a MapReduce job run
- How MapReduce works
    - Map task
    - Reduce task
    - Shuffle and Sort
- A simple example
- Alternatives to MapReduce

Introduction to MapReduce

A programming model for massive data processing in batch mode

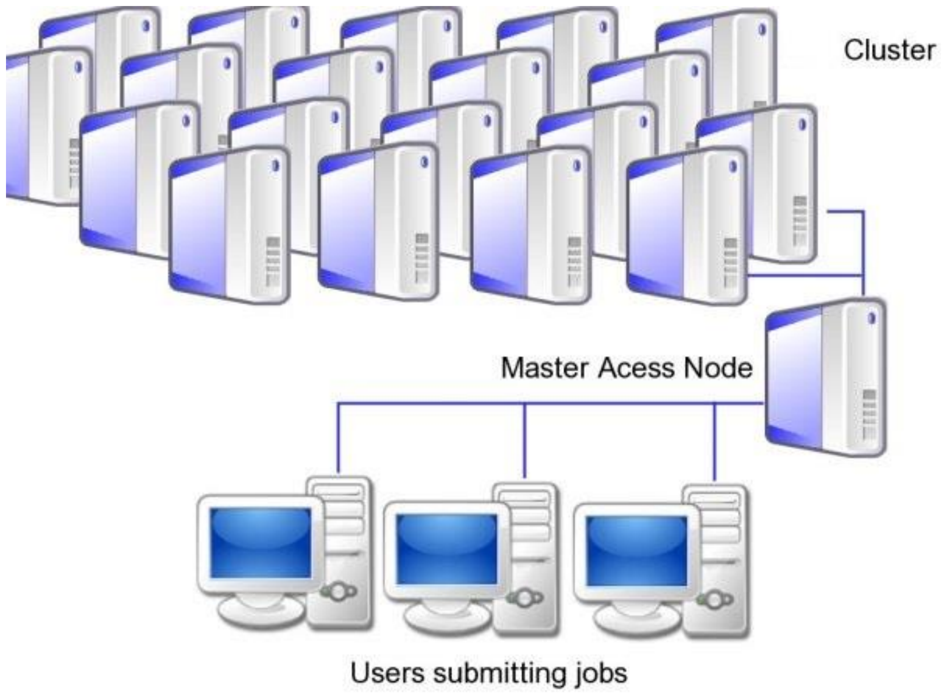
What is MapReduce?

- **MapReduce** is a **distributed** programming model for **massive data processing**.

- How large is an amount of work?
  - Web-scale data on the order of 100s of GBs to TBs or PBs $\rightarrow$ not usually not fit a single computer's hard drive.

Commodity clusters

- The power of MapReduce: the ability to scale to 100s or 1000s of **commodity computers**, each with several cores.
- **Commodity cluster:** small and reasonably priced machines are tied together into a single cost-effective system.

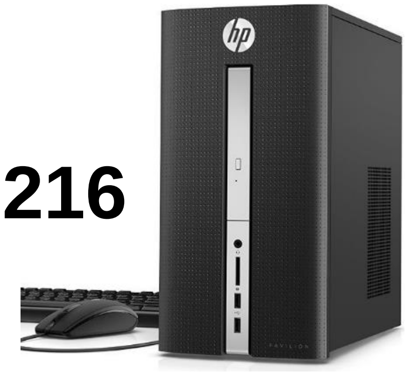


Commodity computer vs. supercomputer

- A theoretical 1000-CPU machine would cost a very large amount of money, far more than 1000 single-CPU or 250 quad-core machines.

1,572,864 cores, 1.5 PiB RAM
250 million US

= 393,216



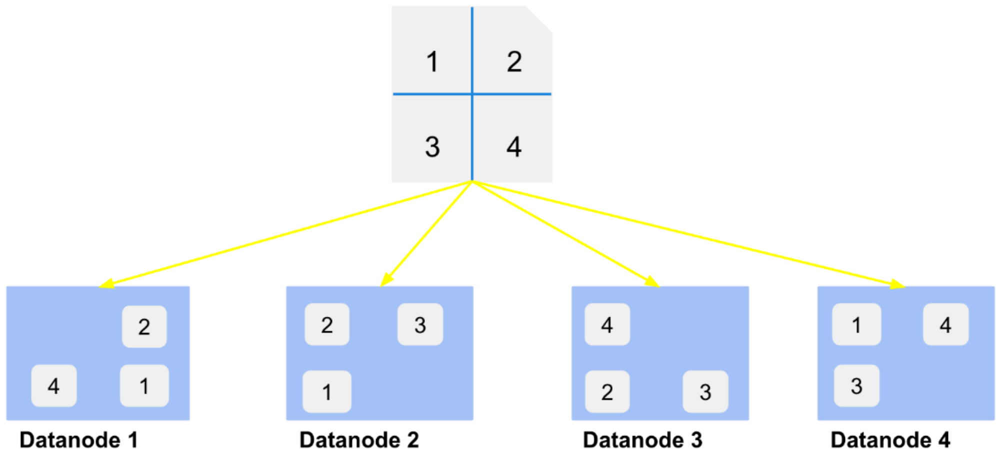
4 cores, 8 GB RAM
500 US

Independent tasks

- MapReduce divides the workload into **multiple independent tasks** and schedule them across cluster nodes.
- The work performed by each task is **done in isolation** from one another.
    - Communication among tasks is mainly limited for scalability reasons.
    - The communication overhead for data synchronization would prevent the model from performing reliably and efficiently at large scale.

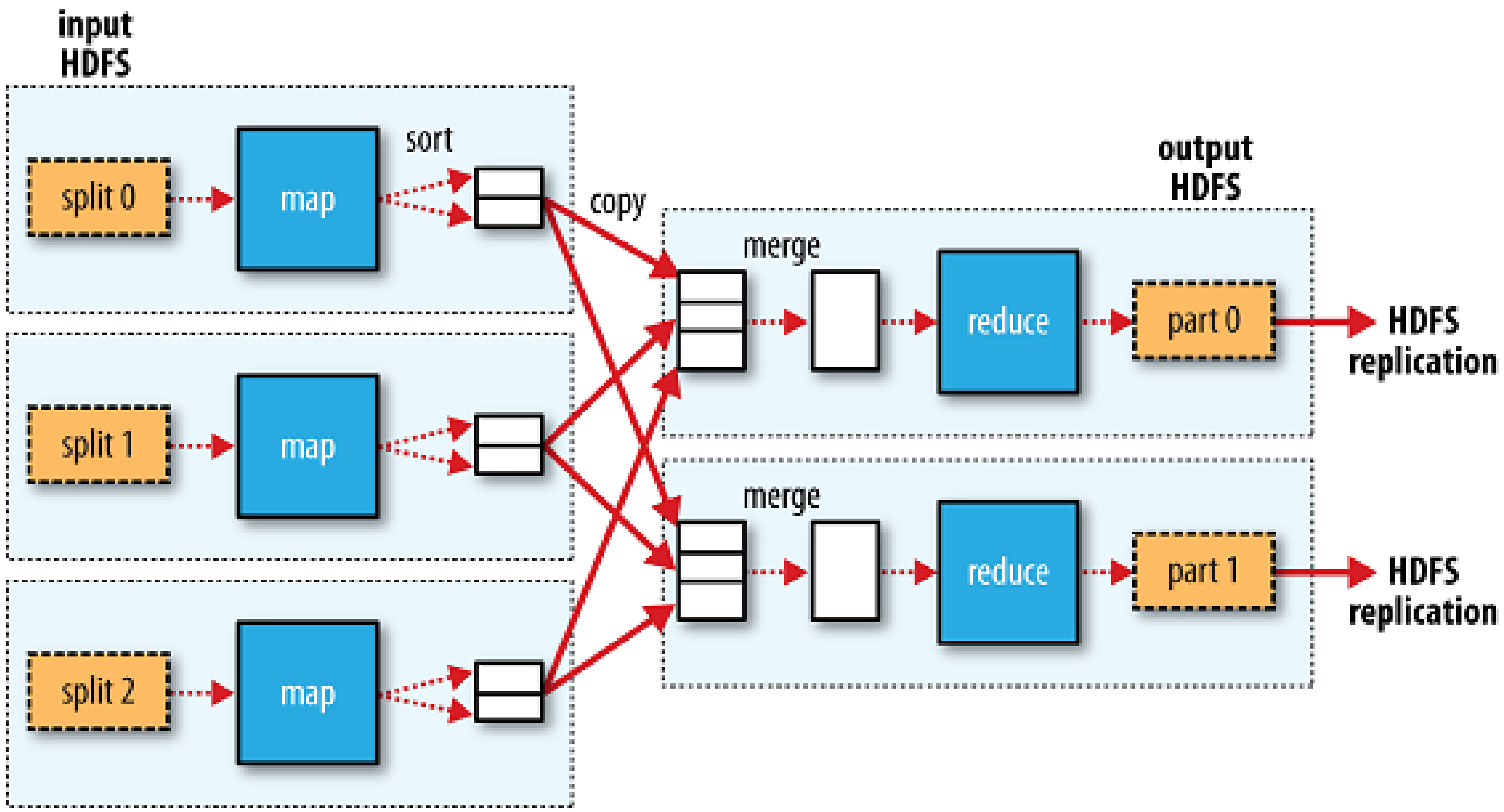
Data distribution

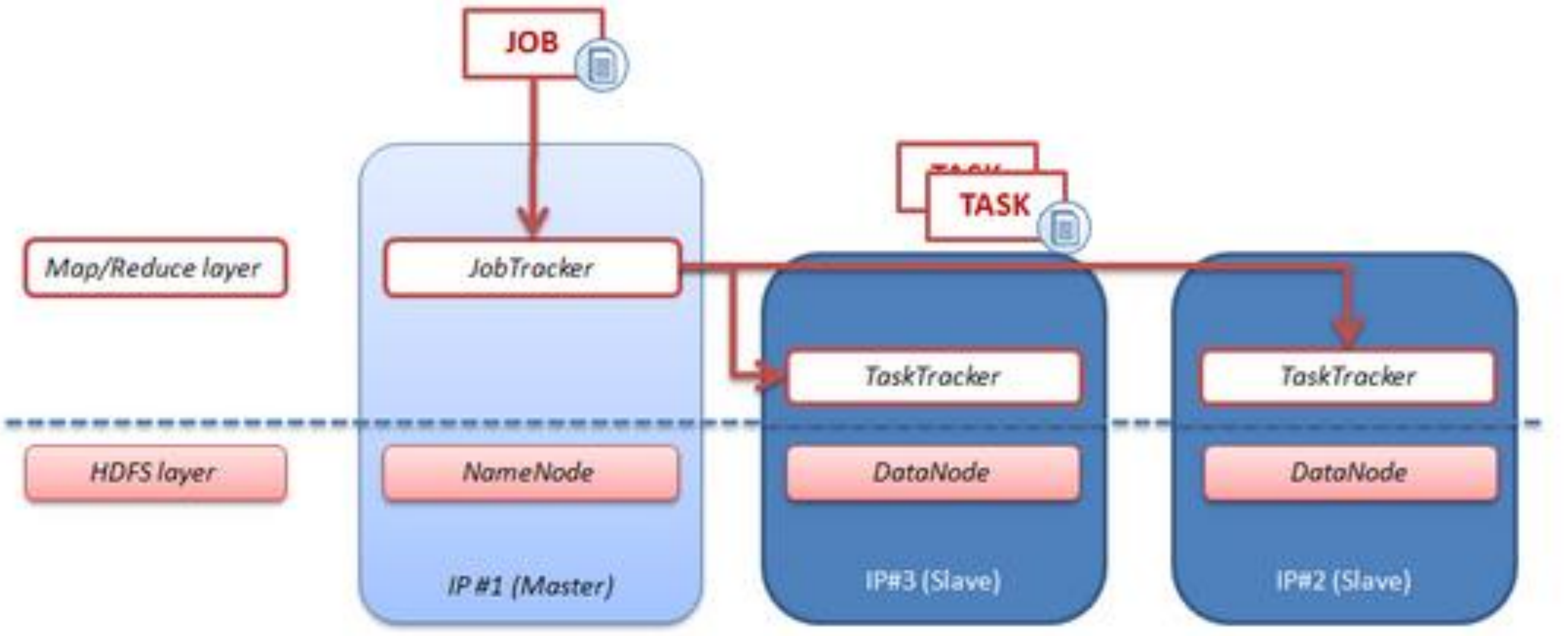
- Data is distributed to cluster nodes as it is being loaded in.
  - An underlying distributed filesystem (e.g., HDFS) splits large data files into chunks which are managed by different nodes in the cluster.



- Several Map tasks can operate on a single file **in parallel**.
  - The performance on huge files can be improved significantly.

A general workflow of MapReduce



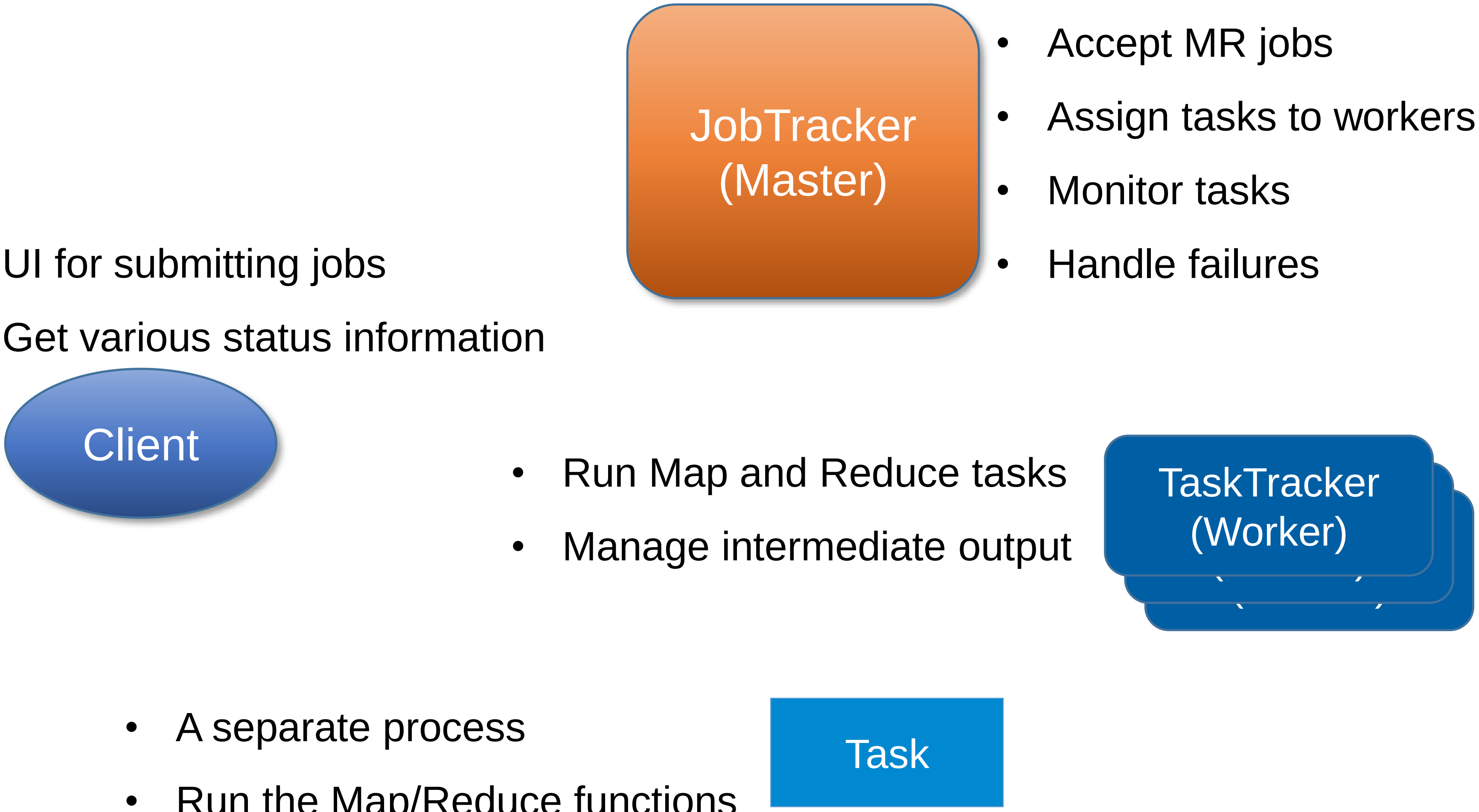


MapReduce job

Anatomy of a MapReduce job run in Hadoop 1.x and 2.x

MapReduce with JobTracker

- Components in a Hadoop MapReduce job run



- Accept MR jobs
- Assign tasks to workers
- Monitor tasks
- Handle failures

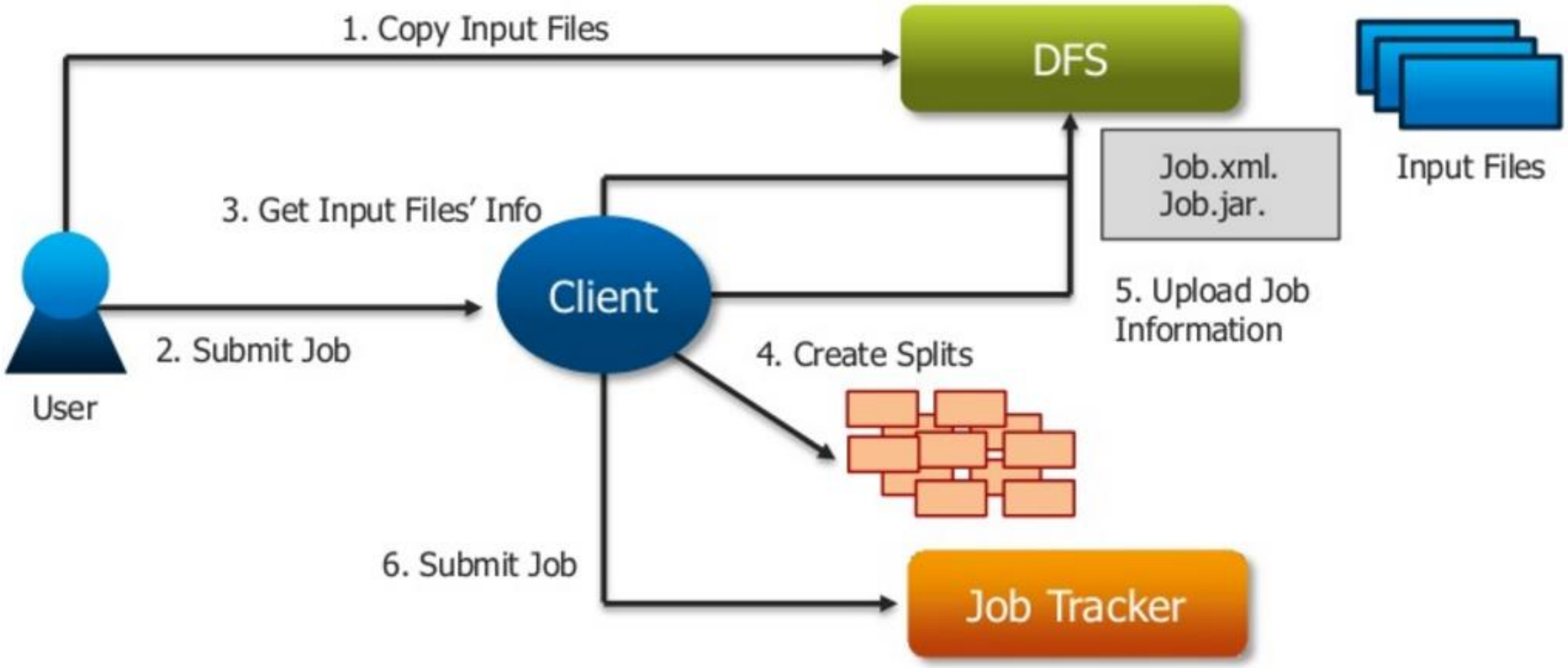
- UI for submitting jobs
- Get various status information

- Run Map and Reduce tasks
- Manage intermediate output

- A separate process
- Run the Map/Reduce functions

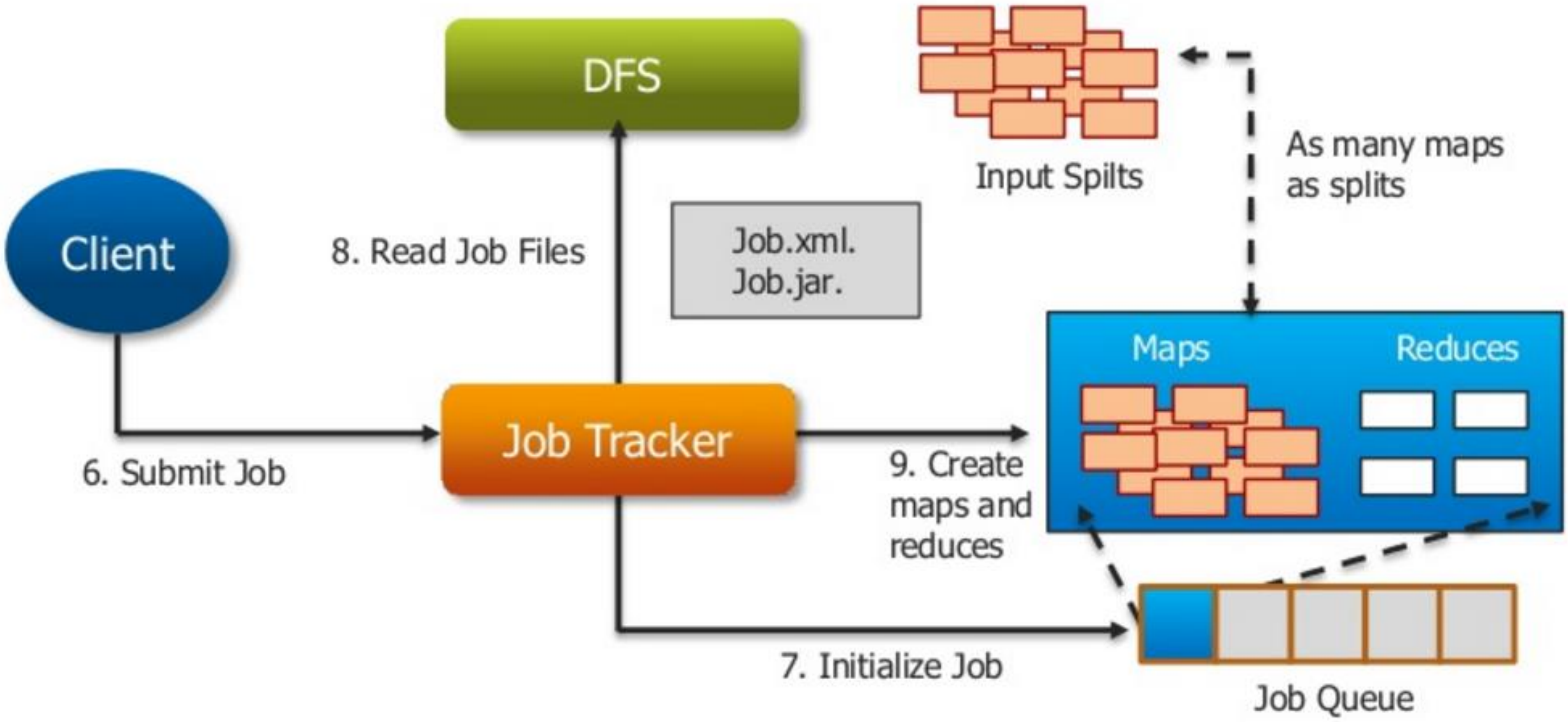
MapReduce with JobTracker

- Job submission



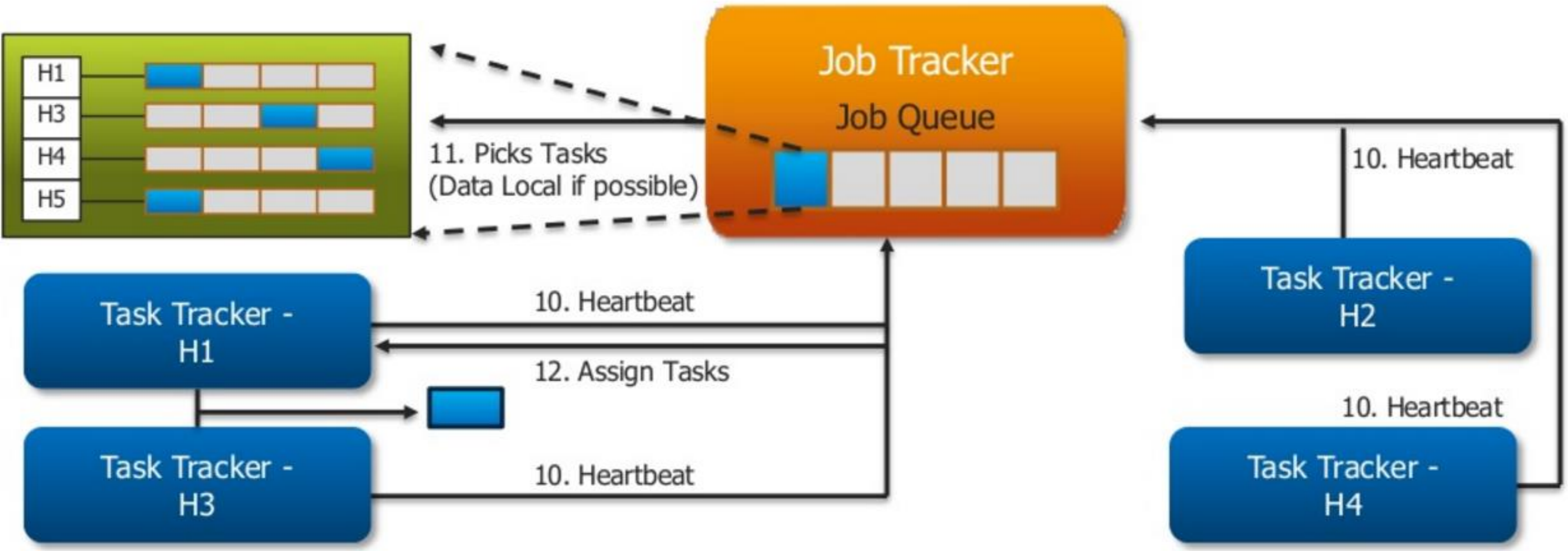
MapReduce with JobTracker

- Job initialization



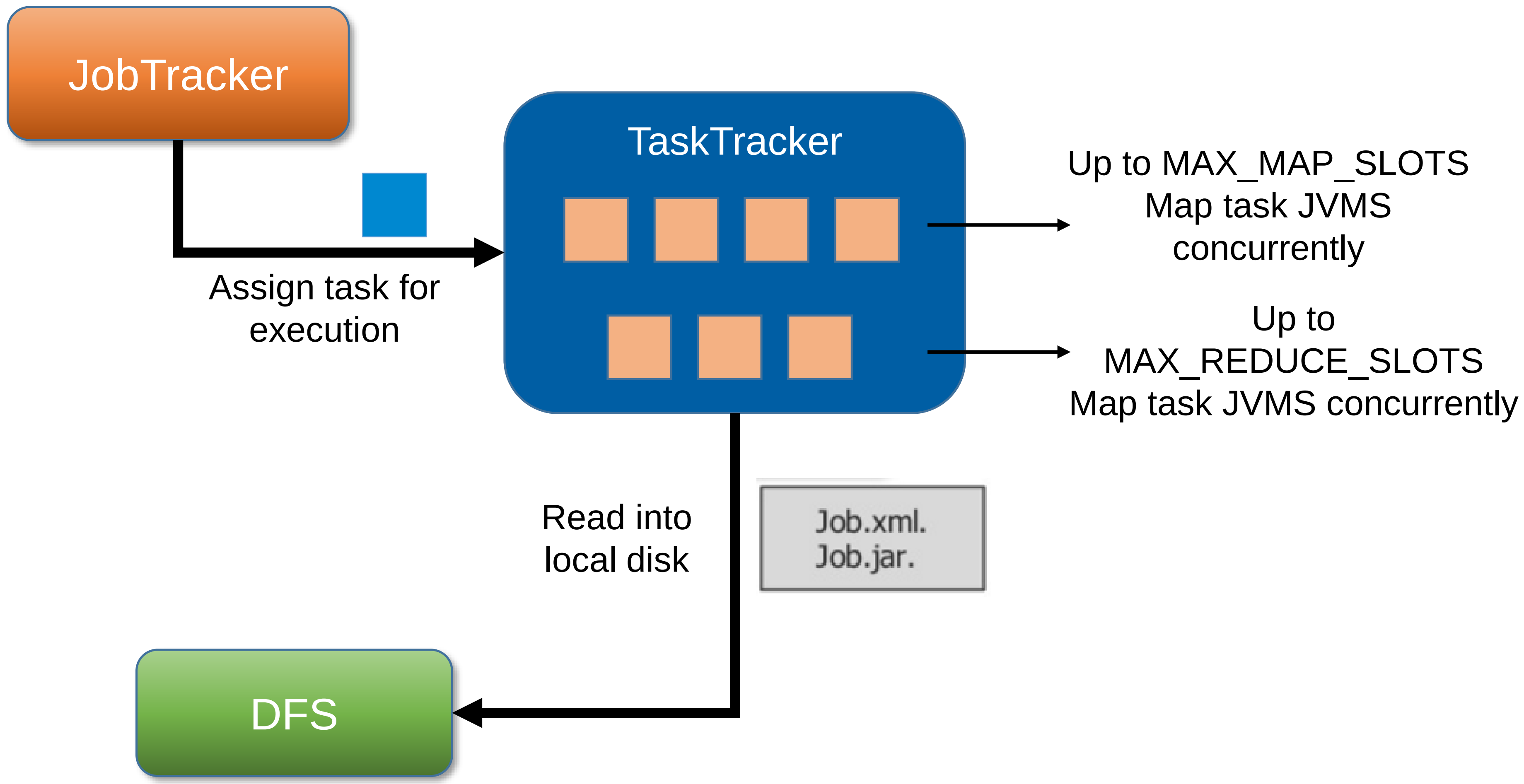
MapReduce with JobTracker

- Job scheduling

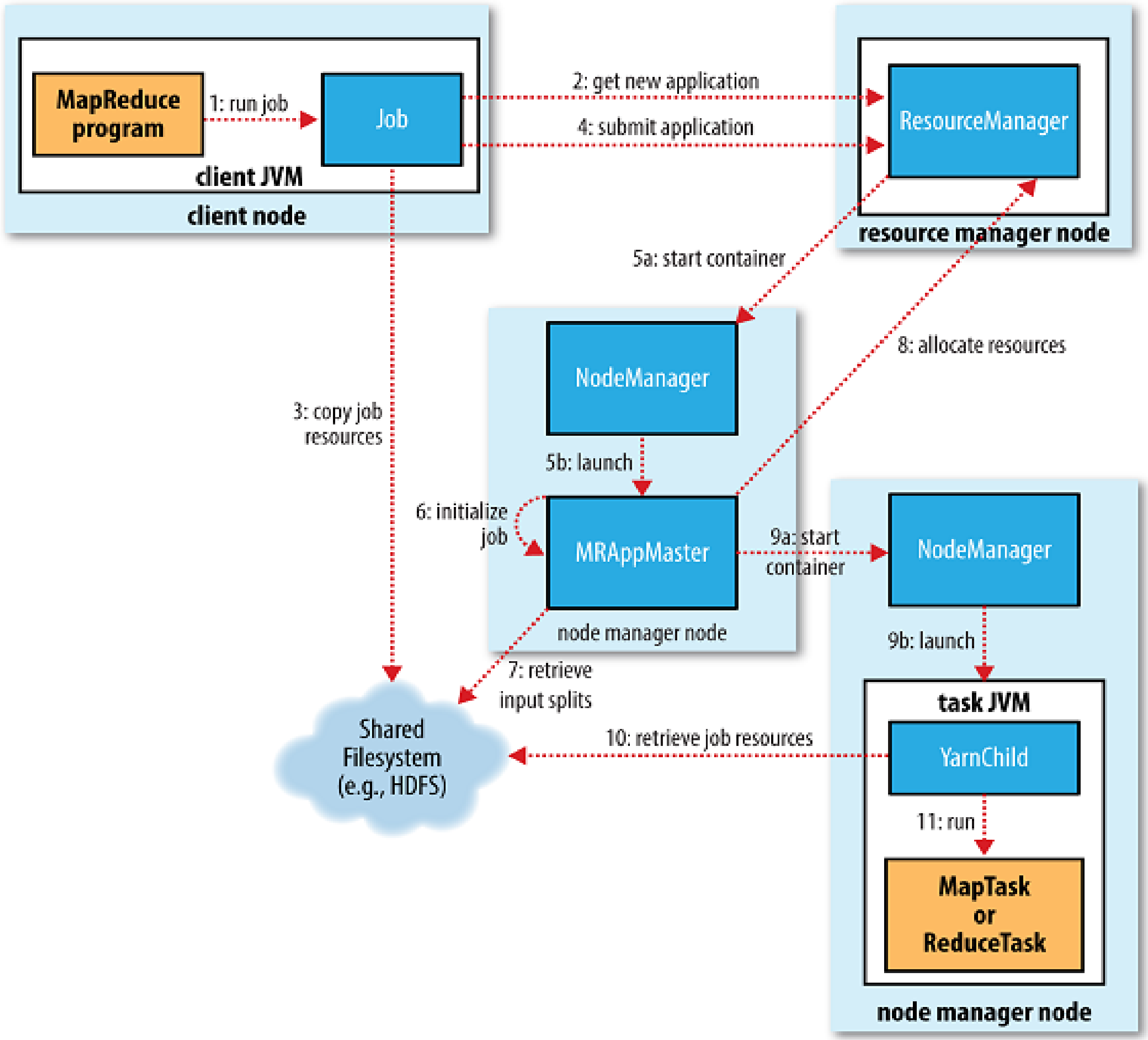


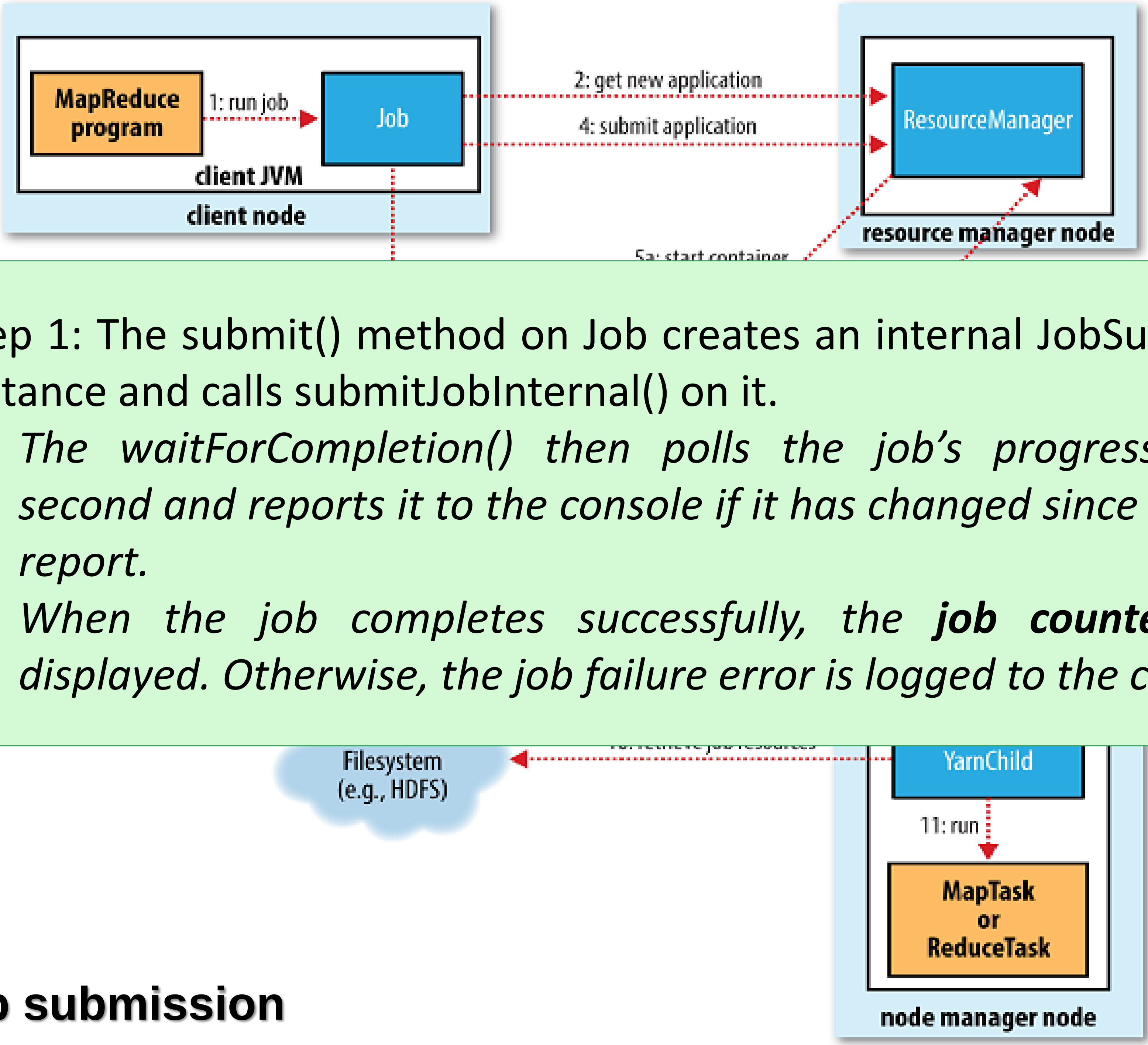
MapReduce with JobTracker

- Job execution


JobTracker
TaskTracker
Assign task for execution
Up to MAX_MAP_SLOTS Map task JVMS concurrently
Up to MAX_REDUCE_SLOTS Map task JVMS concurrently
Read into local disk
Job.xml.
Job.jar.
DFS

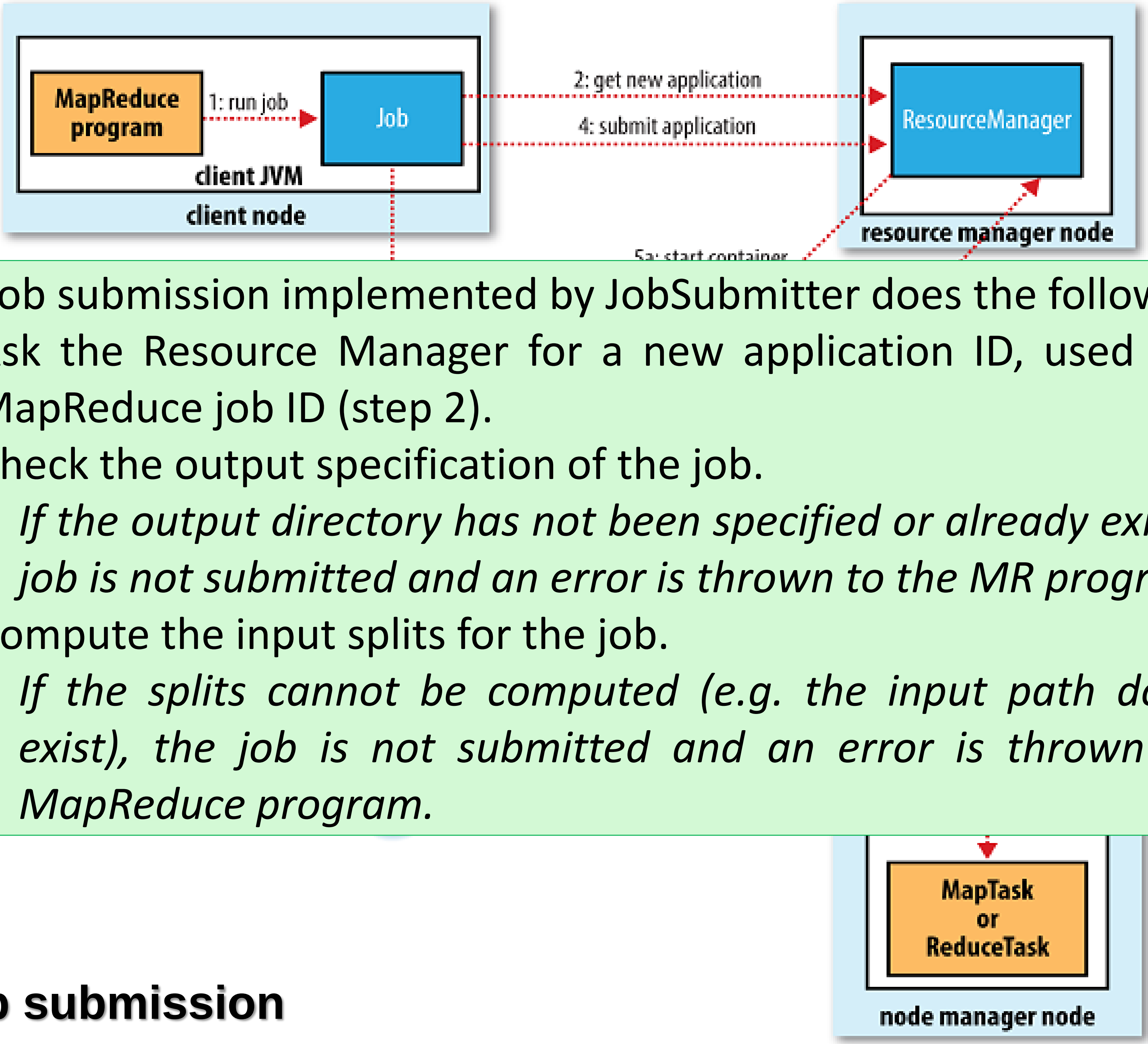
MapReduce with YARN





- Step 1: The submit() method on Job creates an internal JobSubmitter instance and calls submitJobInternal() on it.
    - *The waitForCompletion() then polls the job's progress every second and reports it to the console if it has changed since the last report.*
    - *When the job completes successfully, the **job counters** are displayed. Otherwise, the job failure error is logged to the console.*

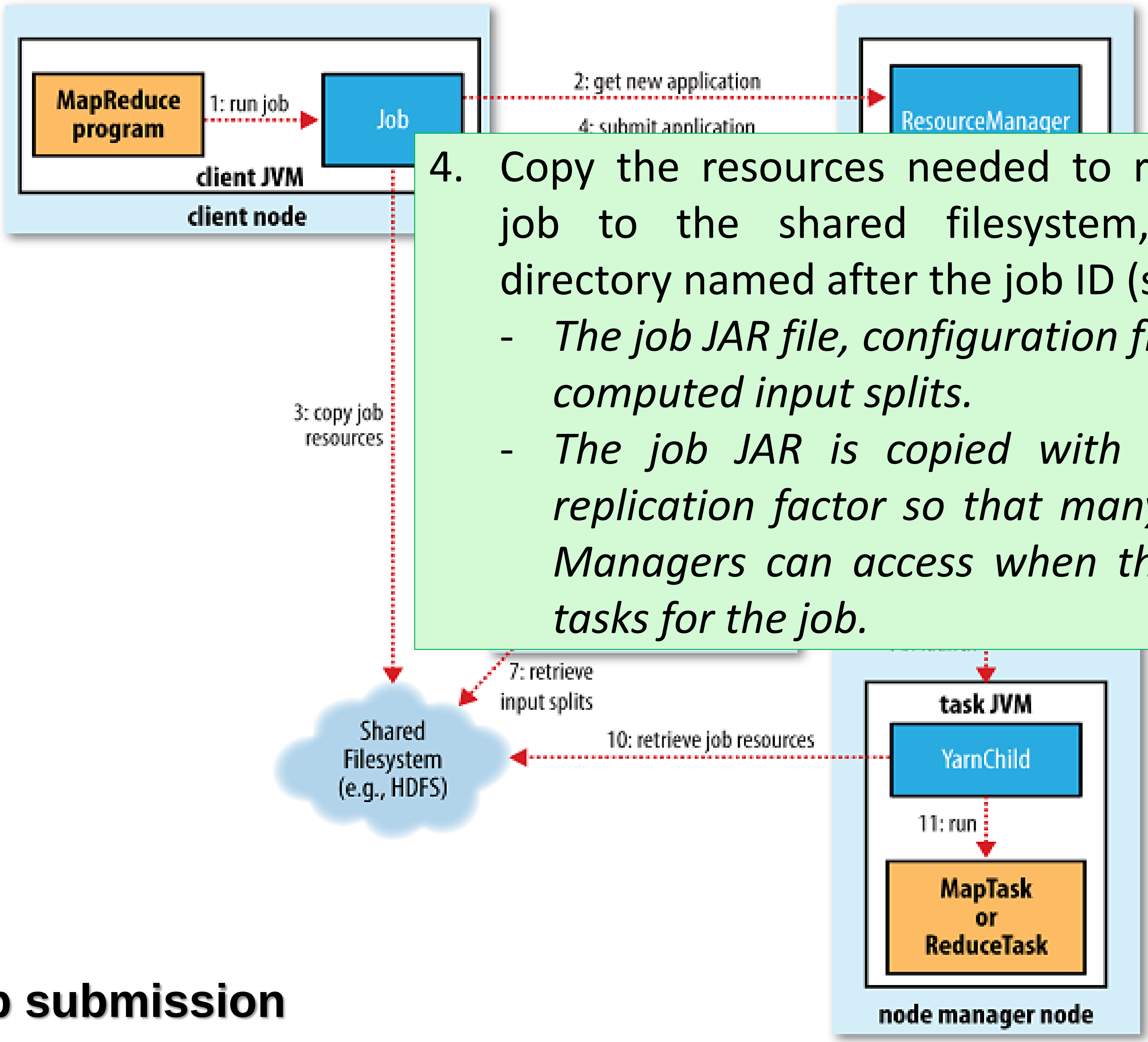
Job submission



The job submission implemented by JobSubmitter does the following:

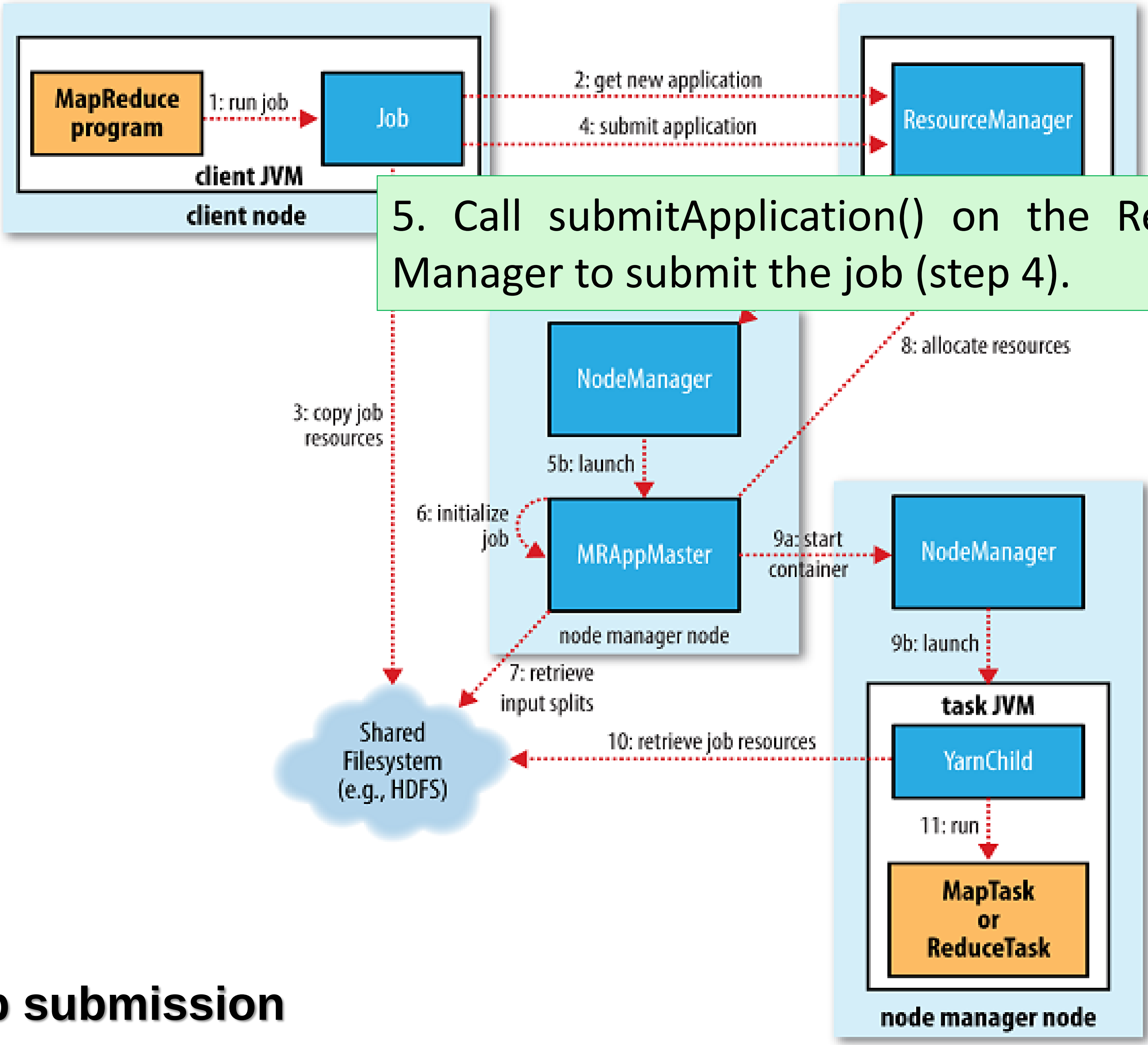
1. Ask the Resource Manager for a new application ID, used for the MapReduce job ID (step 2).
2. Check the output specification of the job.
   - *If the output directory has not been specified or already exists, the job is not submitted and an error is thrown to the MR program.*
3. Compute the input splits for the job.
   - *If the splits cannot be computed (e.g. the input path does not exist), the job is not submitted and an error is thrown to the MapReduce program.*

Job submission



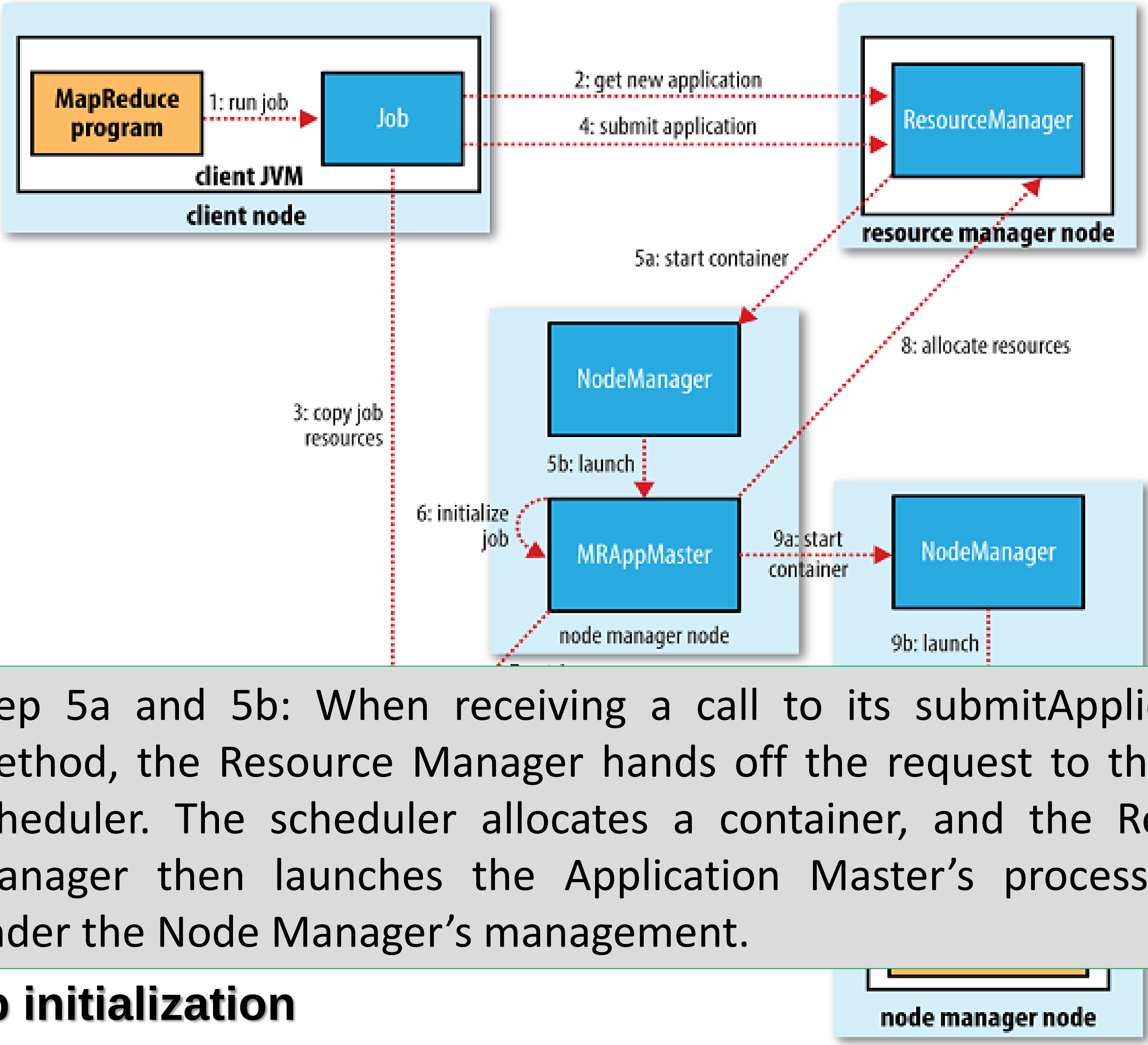
4. Copy the resources needed to run the job to the shared filesystem, in a directory named after the job ID (step 3).
   - *The job JAR file, configuration file, and computed input splits.*
   - *The job JAR is copied with a high replication factor so that many Node Managers can access when they run tasks for the job.*

Job submission



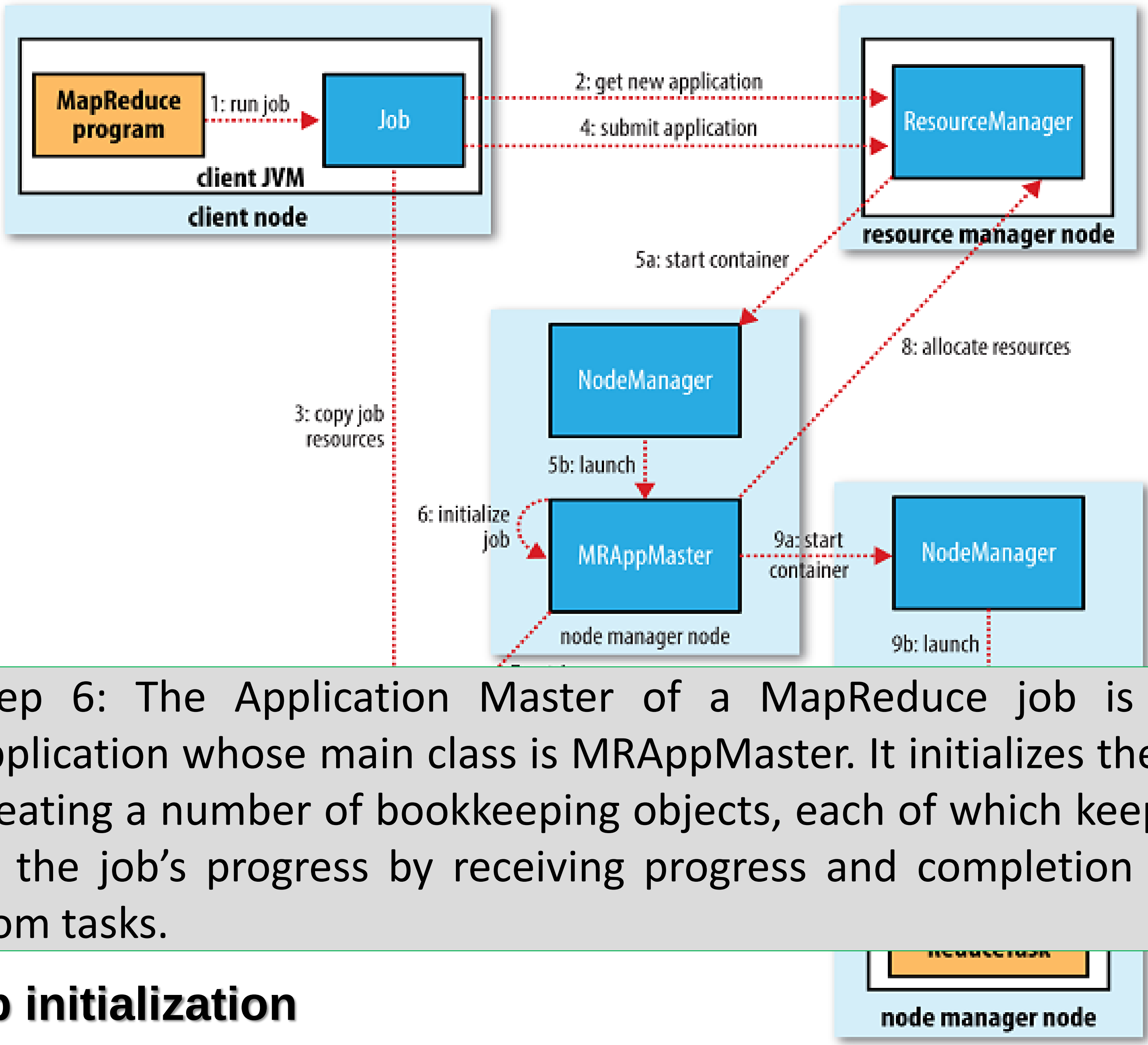
5. Call submitApplication() on the Resource Manager to submit the job (step 4).

Job submission



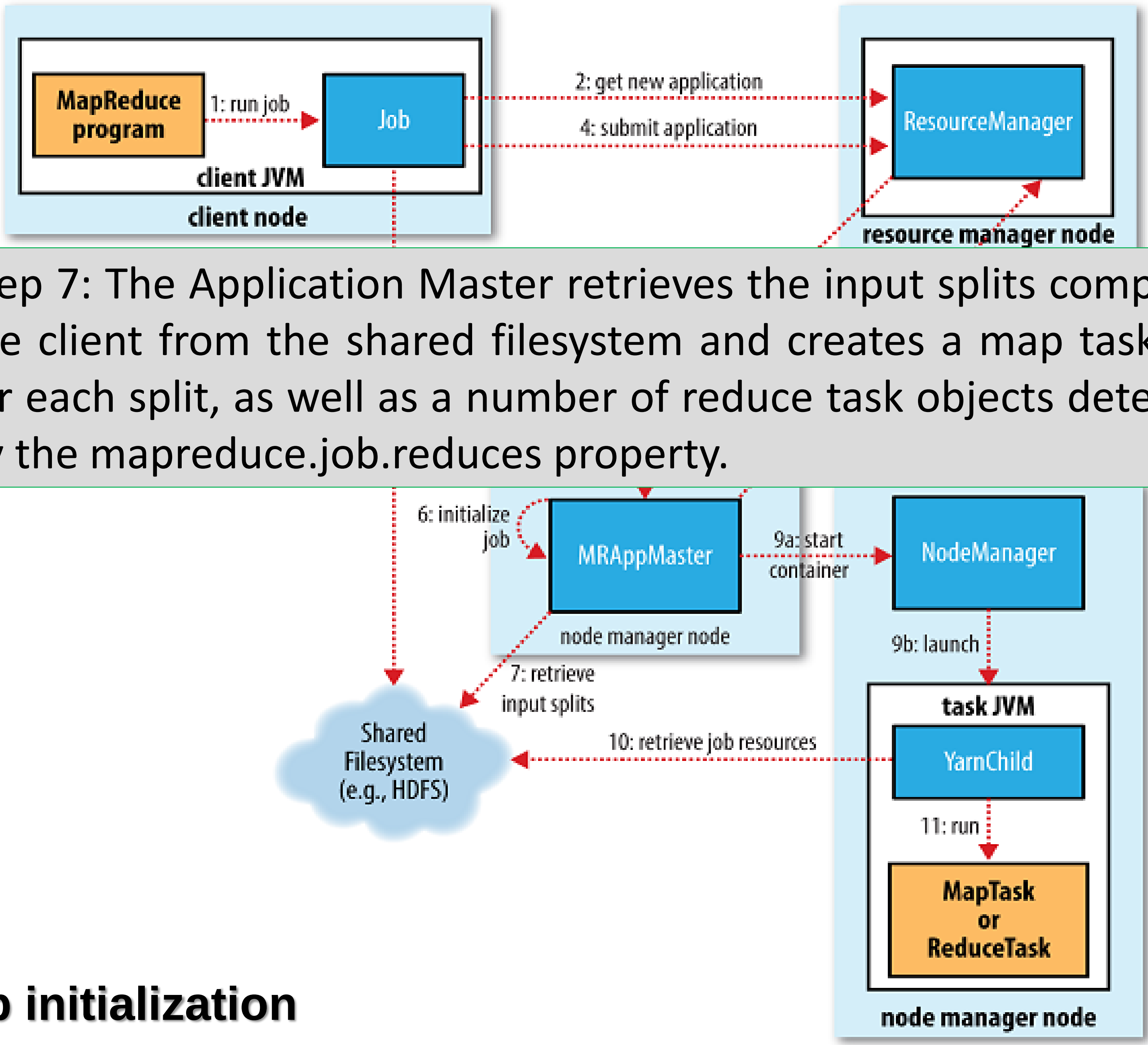
- Step 5a and 5b: When receiving a call to its submitApplication() method, the Resource Manager hands off the request to the YARN scheduler. The scheduler allocates a container, and the Resource Manager then launches the Application Master's process there, under the Node Manager's management.

Job initialization



- Step 6: The Application Master of a MapReduce job is a Java application whose main class is MRAppMaster. It initializes the job by creating a number of bookkeeping objects, each of which keeps track of the job's progress by receiving progress and completion reports from tasks.

Job initialization

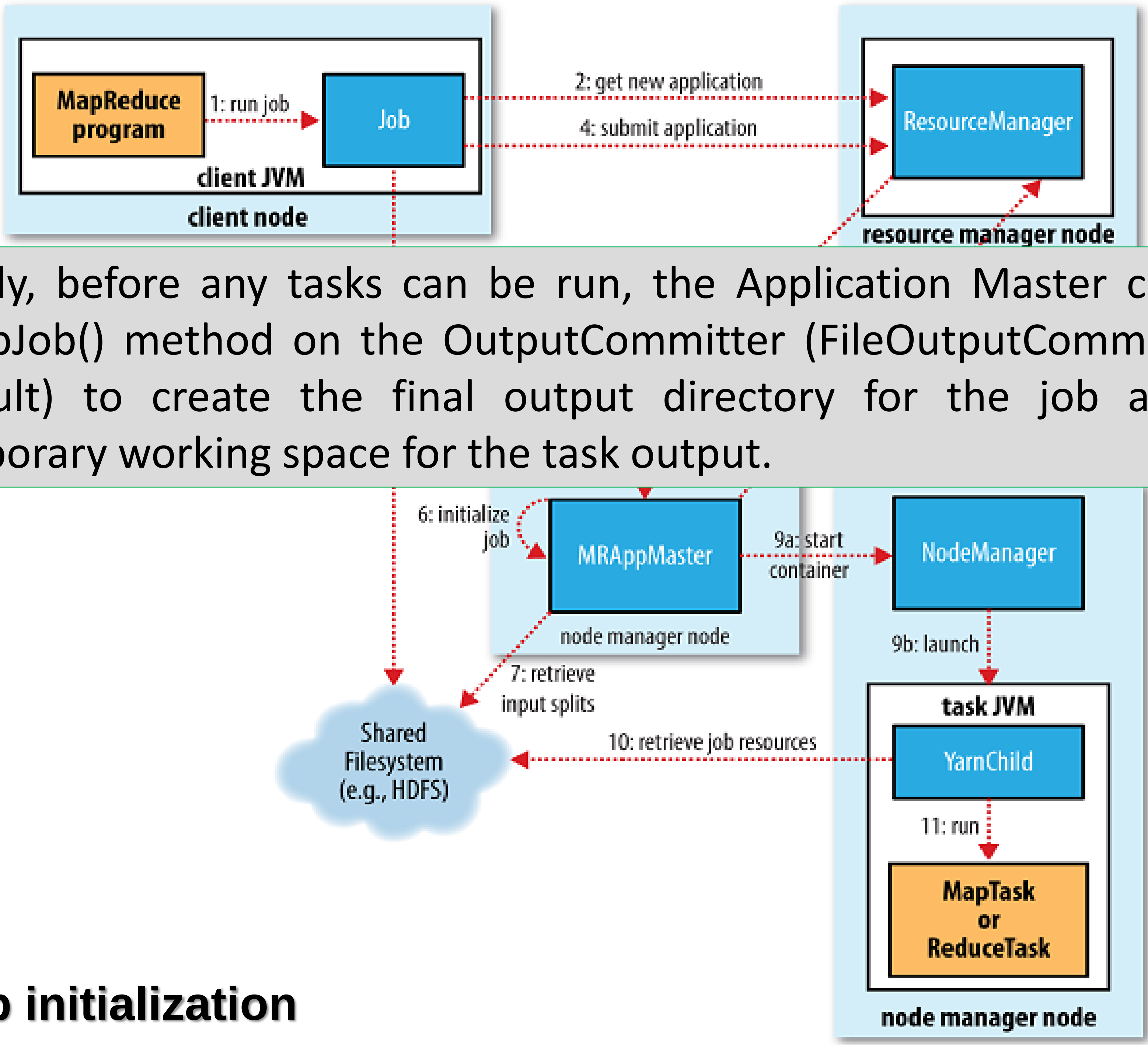


- Step 7: The Application Master retrieves the input splits computed in the client from the shared filesystem and creates a map task object for each split, as well as a number of reduce task objects determined by the mapreduce.job.reduces property.

Job initialization

Uber task

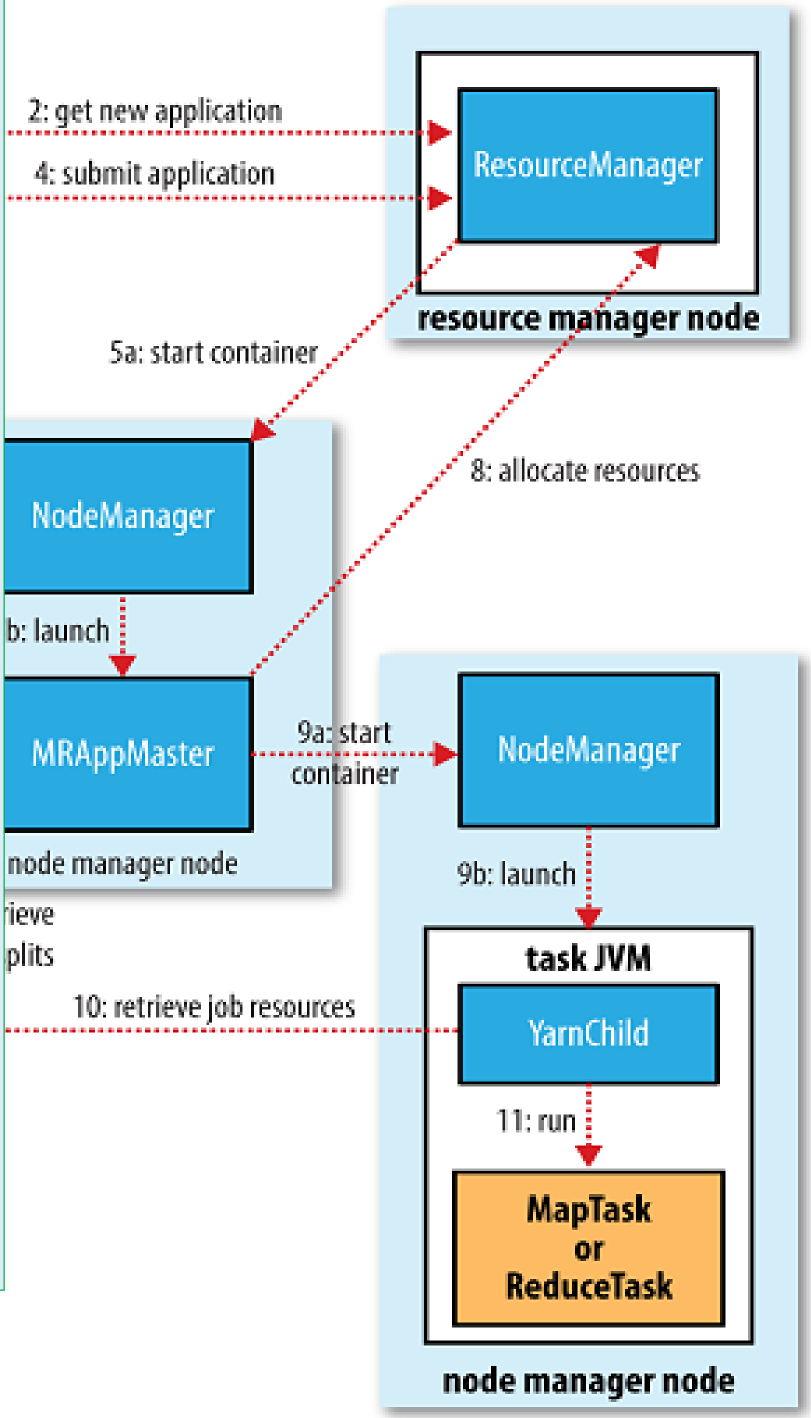
- The Application Master must decide how to run the tasks that make up the MapReduce job.
- For a small job, the Application Master may choose to run the tasks in the same JVM as itself.
  - The overhead of allocating and running tasks in new containers may outweigh the gain to be had in running them in parallel, compared to running them sequentially on one node.
- Such a job is said to be **uberized**, or run as an **uber task**.



Finally, before any tasks can be run, the Application Master calls the setupJob() method on the OutputCommitter (FileOutputCommitter by default) to create the final output directory for the job and the temporary working space for the task output.

Job initialization

- Step 8: If the job does not qualify for uber task, the Application Master requests containers for all the map and reduce tasks from the Resource Manager.
  - *Requests for map tasks are made first with a higher priority than those for reduce tasks, since all the map tasks must complete before the sort phase of the reduce can start.*
  - *Requests for reduce tasks are not made until 5% of map tasks have completed*



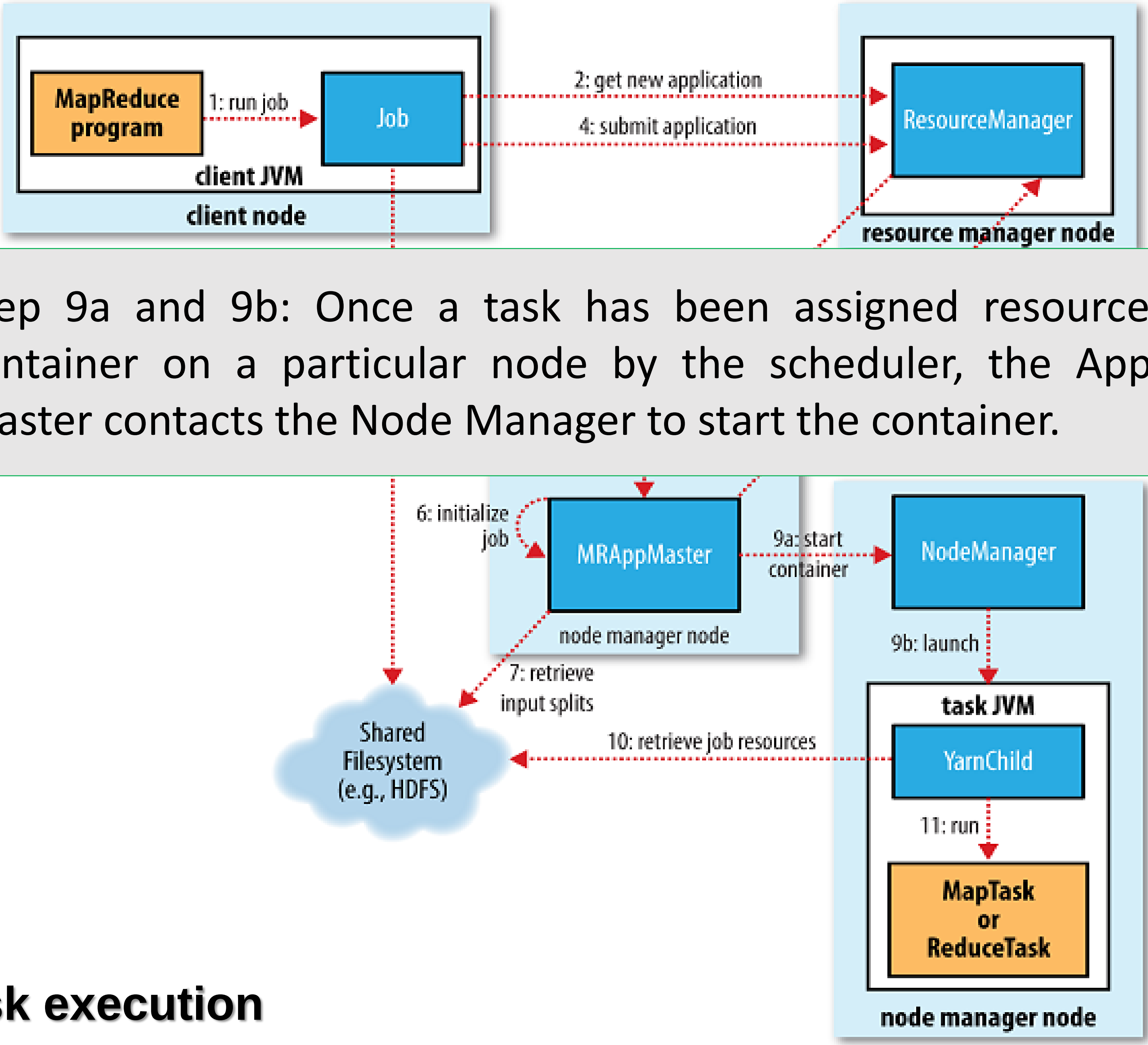
Task assignment

Data locality constraints for Maps

- Reduce tasks can run anywhere in the cluster.
- However, requests for map tasks have **data locality constraints** that the scheduler tries to honor.
    - **Data local (optimal)**: on the same node that the split resides on.
    - **Rack local**: on the same rack, but not the same node, as the split.
    - Some tasks are neither data local nor rack local and retrieve their data from a different rack than the one they are running on.
- For a particular job run, the number of tasks that ran at each locality level can be determined.

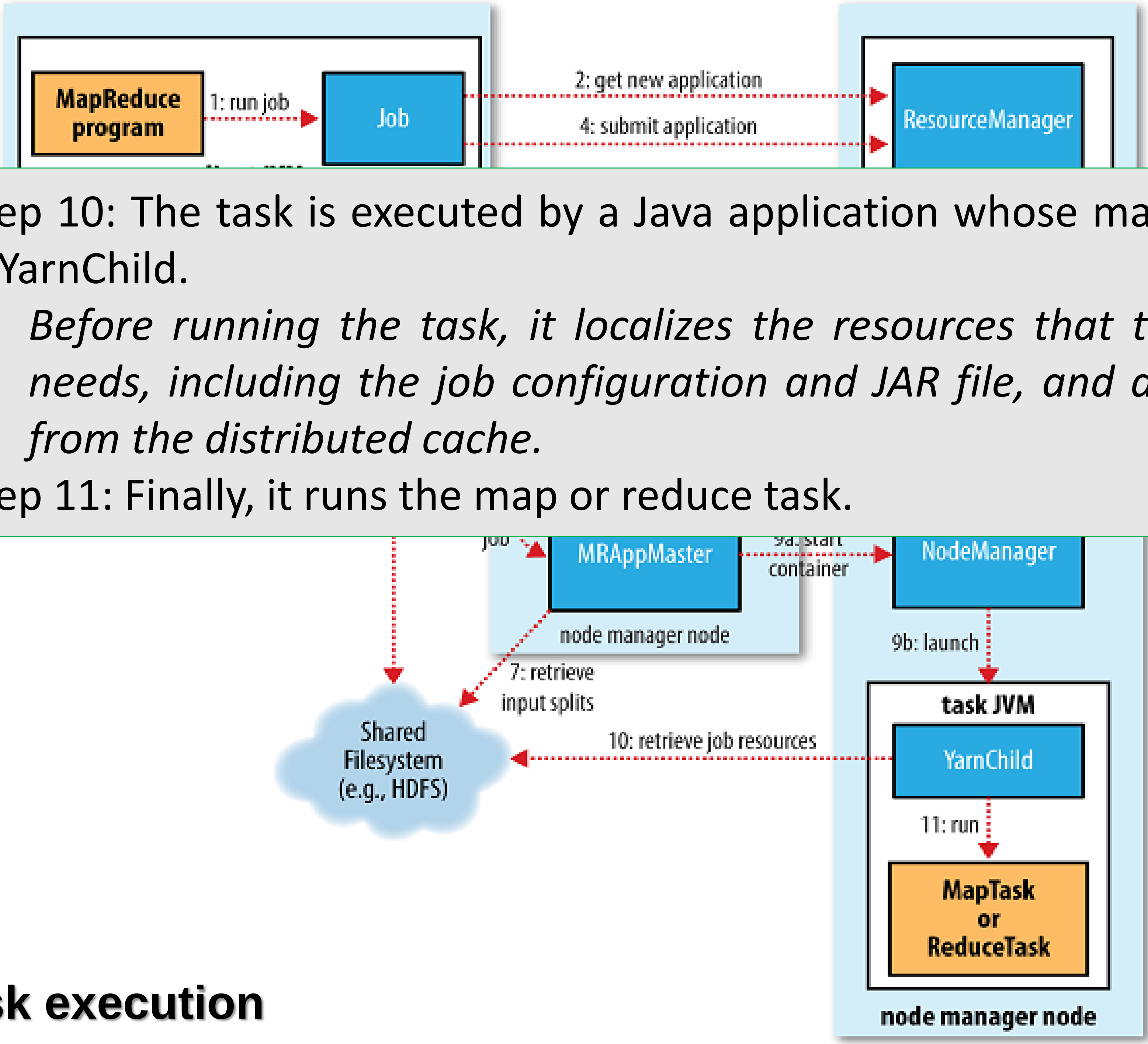
Memory requirements and CPUs

- Requests also specify requirements of memory and CPUs for tasks.
- By default, each map/reduce task is allocated 1,024 MB of memory and one virtual core.
- The values are configurable on a per-job basis.



- Step 9a and 9b: Once a task has been assigned resources for a container on a particular node by the scheduler, the Application Master contacts the Node Manager to start the container.

Task execution

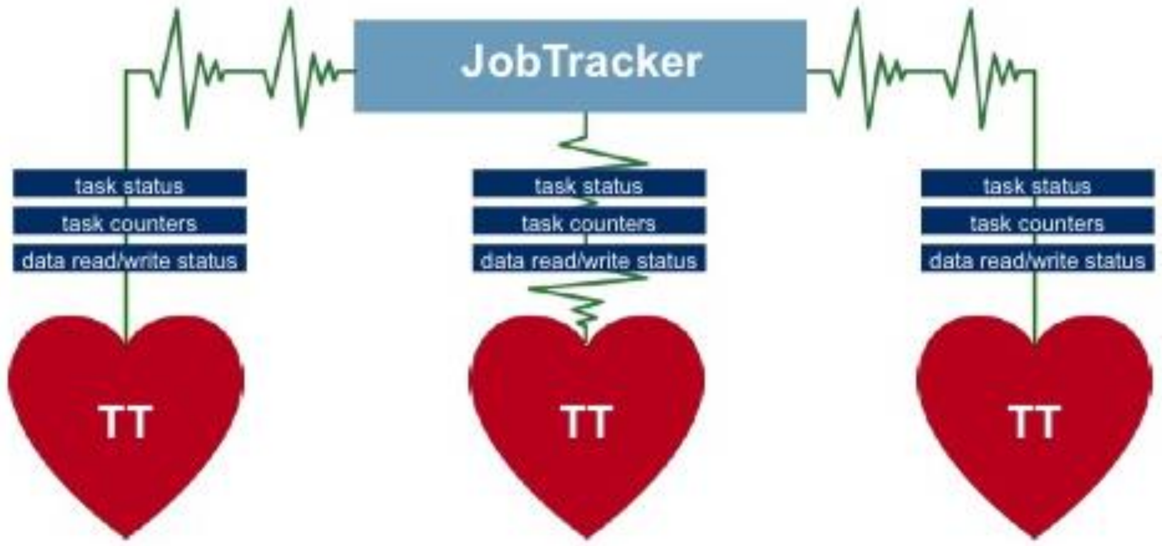


- Step 10: The task is executed by a Java application whose main class is YarnChild.
    - *Before running the task, it localizes the resources that the task needs, including the job configuration and JAR file, and any files from the distributed cache.*
- Step 11: Finally, it runs the map or reduce task.

Task execution

Progress and status updates

- MapReduce jobs are **long-running batch jobs**, taking anything from tens of seconds to hours to run.
- It is important to get feedback on how the job is progressing.



Progress and status updates

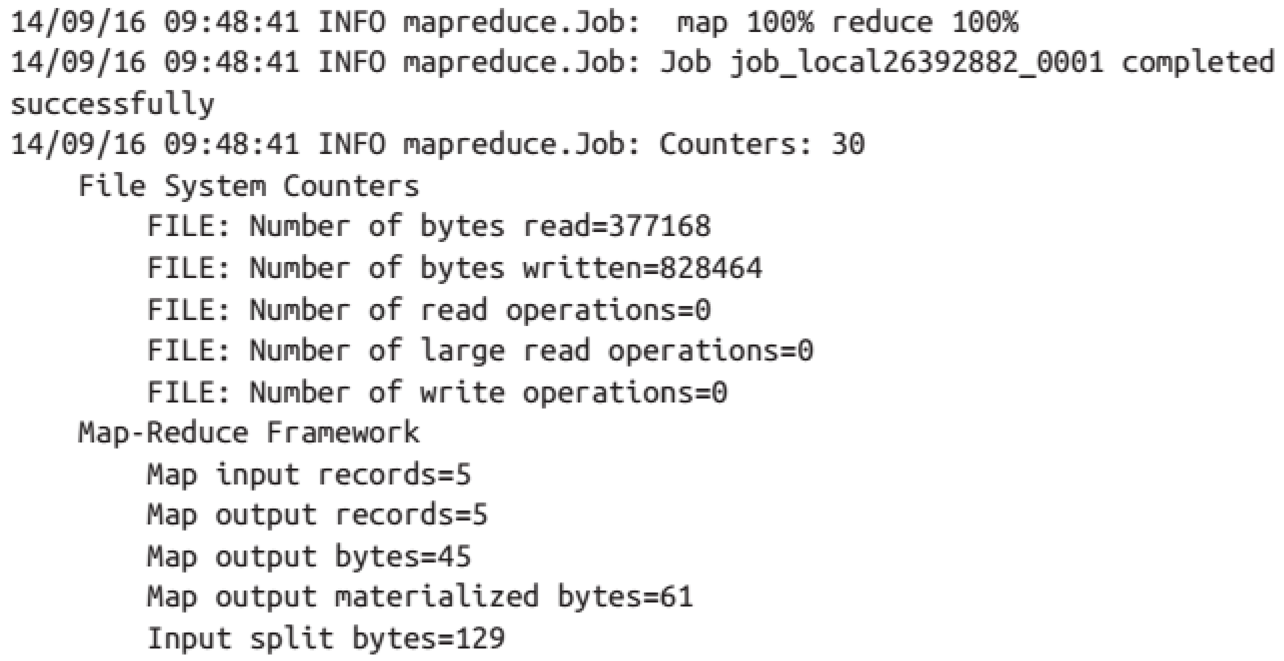
- A job and each of its tasks have a status, which includes
  - The state of job or task (e.g. running, successfully completed, failed).
  - The progress of maps and reduces.
  - The values of the job's counters.
  - A status message or description (which may be set by user code).
- These statuses change over the course of the job.

Progress and status updates

- The progress of a task (i.e. the proportion of the task completed) is tracked while it is running.
- **Map tasks:** the proportion of the input processed.
- **Reduce tasks:** the proportion of the reduce input processed.
    - The total progress is divided into three parts, corresponding to the three phases of the shuffle process.
    - E.g., if the task has run the reducer on half its input, the task’s progress is 5/6, since it has completed the copy and sort phases (1/3 each) and is halfway through the reduce phase (1/6).

Hadoop MapReduce counters

- Tasks also have a set of counters that count various events as the task runs.
- Counters are either built into the framework, such as the number of map output records written, or defined by users.



```
14/09/16 09:48:41 INFO mapreduce.Job:  map 100% reduce 100%
14/09/16 09:48:41 INFO mapreduce.Job: Job job_local26392882_0001 completed
successfully
14/09/16 09:48:41 INFO mapreduce.Job: Counters: 30
    File System Counters
        FILE: Number of bytes read=377168
        FILE: Number of bytes written=828464
        FILE: Number of read operations=0
        FILE: Number of large read operations=0
        FILE: Number of write operations=0
    Map-Reduce Framework
        Map input records=5
        Map output records=5
        Map output bytes=45
        Map output materialized bytes=61
        Input split bytes=129
```

Hadoop MapReduce counters

- MapReduce counters are divided into **Task Counters** and **Job Counters**.
- There are several groups for the built-in counters
  - MapReduceTask Counters
  - Filesystem Counters
  - FileInput-Format Counters
  - FileOutput-Format Counters
  - Job Counters
- See an example in “A test run” on page 27, Hadoop: The Definitive Guide, 4th edition.

MapReduce counters: Task counter

- Gather information about tasks over their execution, and the results are aggregated over all the tasks in a job.
  - Periodically sent from each task to the master units (i.e. TaskTracker $\rightarrow$ JobTracker, or Application Master $\rightarrow$ Resource Manager).
- Counter values are definitive only once a job has successfully completed.
- Some values provide useful diagnostic information as a task is progressing, which can be monitored with a web UI.
  - E.g., PHYSICAL_MEMORY_BYTES, VIRTUAL_MEMORY_BYTES, and COMMITTED_HEAP_BYTES.

MapReduce counters: Job counter

- Measure the job-level statistics
- Maintained by JobTracker or Application Master in YARN
  - E.g., TOTAL_LAUNCHED_MAPS – the number of map tasks launched over the course of a job.

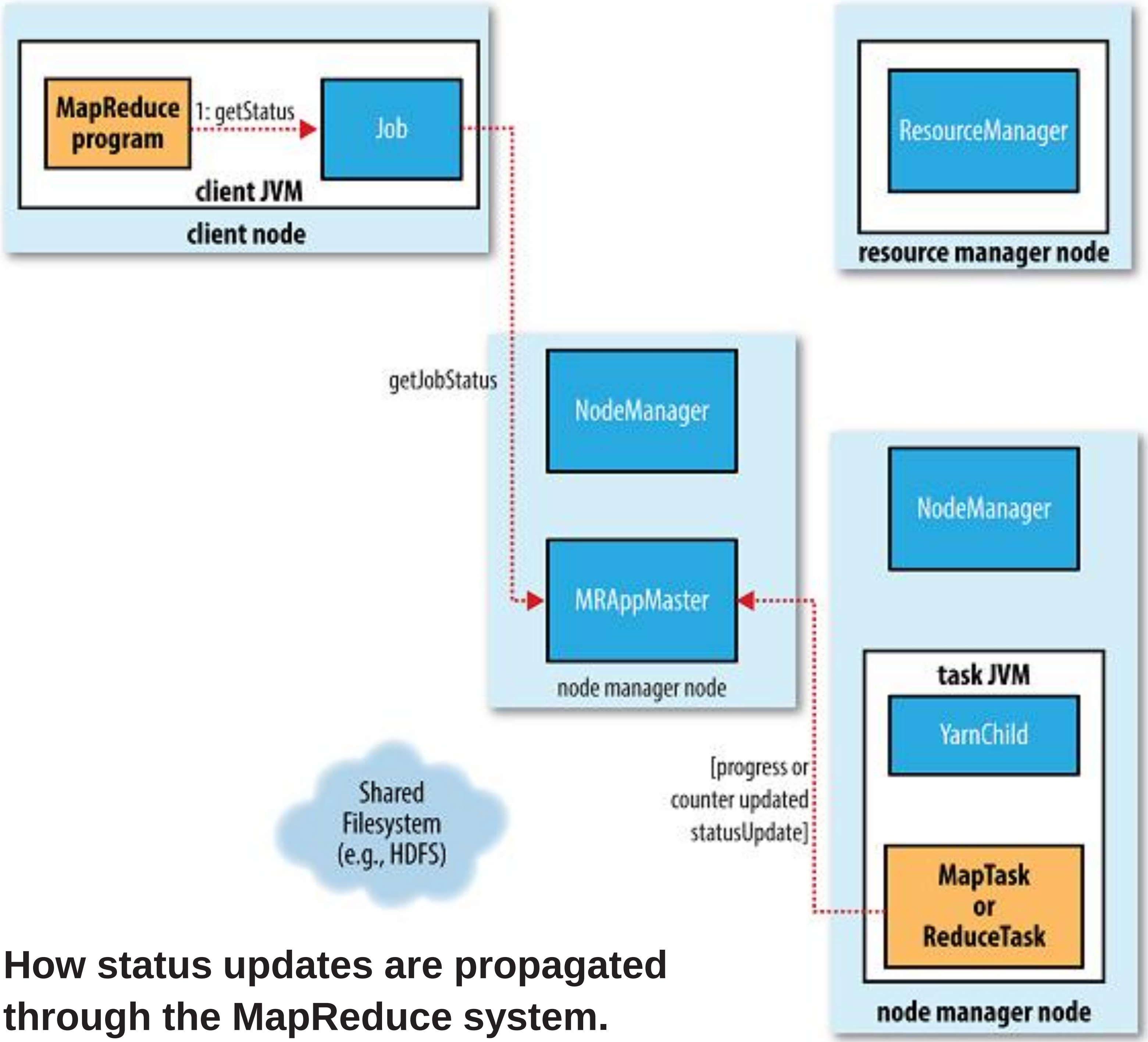
Progress and status updates

- As the map or reduce task runs, the child process communicates with its parent Application Master through the **umbilical interface** every three seconds.
  - The task reports its progress and status (including counters) back to its Application Master, which has an aggregate view of the job

- The Resource Manager webUI displays all the running applications with links to the webUIs of their respective Application Masters.

Progress and status updates

- During the course of the job, the client receives the latest status by polling the Application Master every second.
  - mapreduce.client.progressmonitor.pollinterval
- Clients can also use Job’s getStatus() method to obtain a JobStatus instance, which contains all of the status information for the job.



How status updates are propagated through the MapReduce system.

Job completion

- When the Application Master receives a notification that the last task for a job is complete, it changes the status for the job to **“successful.”**
  - The Application Master also sends an HTTP job notification if it is configured to do so.
- The waitForCompletion() method is returned and a message is printed to the console.
- Job statistics and counters are also shown at this point.

Job completion

- Finally, the Application Master and the task containers clean up their working state (so intermediate output is deleted).
- The OutputCommitter’s commitJob() method is called.

- Job information is archived by the **job history server** to enable later interrogation by users if desired.

Failures

Failure handling

- Buggy user code, processes crash, and machines fail.
- The failure may take place on any of the following entities
  - Task
  - Application Master
  - Node Manager
  - Resource Manager

Task failure

- Runtime exception thrown by user code or sudden exit of the task JVM (JVM bug).
- The Node Manager notices that the process has exited and informs the Application Master to mark the attempt as failed.
- The Application Master then frees up the container so that its resources are available for another task.

Task failure: Hanging tasks

- The Application Master notices the absence of a progress update for a while and proceeds to mark the task as failed.
  - mapreduce.task.timeout (millisecs), 10 mins by default, configured on a per-job basis (or a cluster basis).
- The task JVM process will be killed automatically after that.
- A hanging task will then never free up its container, resulting in a cluster slowdown.

Task failure: Rescheduling

- The Application Master avoids rescheduling tasks on a Node Manager where it has previously failed.
- A task failed several times will not be retried again → the whole job fails
  - mapreduce.map.maxattempts, mapreduce.reduce.maxattempts (by default, 4).
- It is sometimes undesirable to abort the job when a few tasks fail, as it may be possible to use the job’s results.
  - The maximum % tasks allowed to fail without triggering job failure can be set for the job, using mapeduce.map.failures.maxpercent and mapreduce.reduce.failures.maxpercent.

Task failure: Killing task

- A task may also be killed (which is different from it failing), called a **speculative duplicate**.
  - The Node Manager it was running on failed, and the Application Master marked all the task attempts running on it as killed.
- Killed task attempts do not count against the number of attempts to run the task.
- Users may also kill task attempts or job using the webUI or the command line.

Application Master failure

- YARN imposes a limit for the maximum number of attempts for any Application Master running on the cluster.
- Over this number, the Application Master will not be tried again and the job will fail.
    - mapreduce.am.max-attempts property (2, by default).

Application Master failure: Recovery

- The Resource Manager detects failure when there is no heartbeats from the Application Master.
- A new instance running in a new container (managed by a Node Manager) will be started by the Resource Manager.
- The Application Master uses the **job history** to recover the state of the tasks that were already run by the (failed) so as to not rerun these tasks.
    - yarn.app.mapreduce.am.job.recovery.enable (default = true).

Application Master failure: Recovery

- The client will experience a timeout when it issues a status update to the failed Application Master

  $\rightarrow$ ask the Resource Manager for the new Application Master's address.

- This process is transparent to the user.

Node Manager failure

- A failed Node Manager (crashing or running very slowly) will stop sending heartbeats for 10 minutes to the Resource Manager (or send them very infrequently).
  - yarn.resourcemanager.nm.liveness-monitor.expiry-interval-ms
- The Resource Manager remove the failed node from its pool of nodes to schedule containers on.

Node Manager failure: Recovery

- Any task or Application Master running on the failed Node Manager will be recovered using the previous mechanisms.
- In addition, map tasks that were run and completed successfully on the failed Node Manager are **rerun** if they belong to incomplete jobs.
  - Their intermediate output residing on the failed Node Manager's local filesystem may not be accessible to the reduce task.

Node Manager failure

- Node Managers may be blacklisted by the Application Master if the number of failures for application is high, even if the node itself has not failed.
    - mapreduce.job.maxtaskfailures.per.tracker (by default, 3).

The Resource Manager does not do blacklisting across applications *(at the time of writing the book), so tasks from new jobs may be scheduled on bad nodes even if they have been blacklisted by an Application Master running earlier.*

Resource Manager failure

- Failure of the Resource Manager is serious.
- SPOF in the default configuration
- **High availability (HA):** run a pair of Resource Managers in an active-standby configuration.
    - Information about all the running applications is stored in a highly available state store (backed by ZooKeeper or HDFS).

Resource Manager failure: HA

- The new Resource Manager reads the application information from the state store to restart the Application Masters for all the applications running on the cluster.
- This does not count as a failed application attempt.
  - The application did not fail due to an error in the application code, but was forcibly killed by the system.
  - yarn.resourcemanager.am.max-attempts is not affected.

Is the Application Master restart an issue for MapReduce applications?

Resource Manager failure: HA

- The transition of a Resource Manager from standby to active is handled by an embedded **failover controller**
  - E.g. ZooKeeper
- It is also possible to configure manual failover but not recommended.

- Clients and Node Managers must be configured to handle Resource Manager failover.
  - They try connecting to each Resource Manager in a **round-robin fashion** until they find the active one.

Speculative Execution

Straggling task

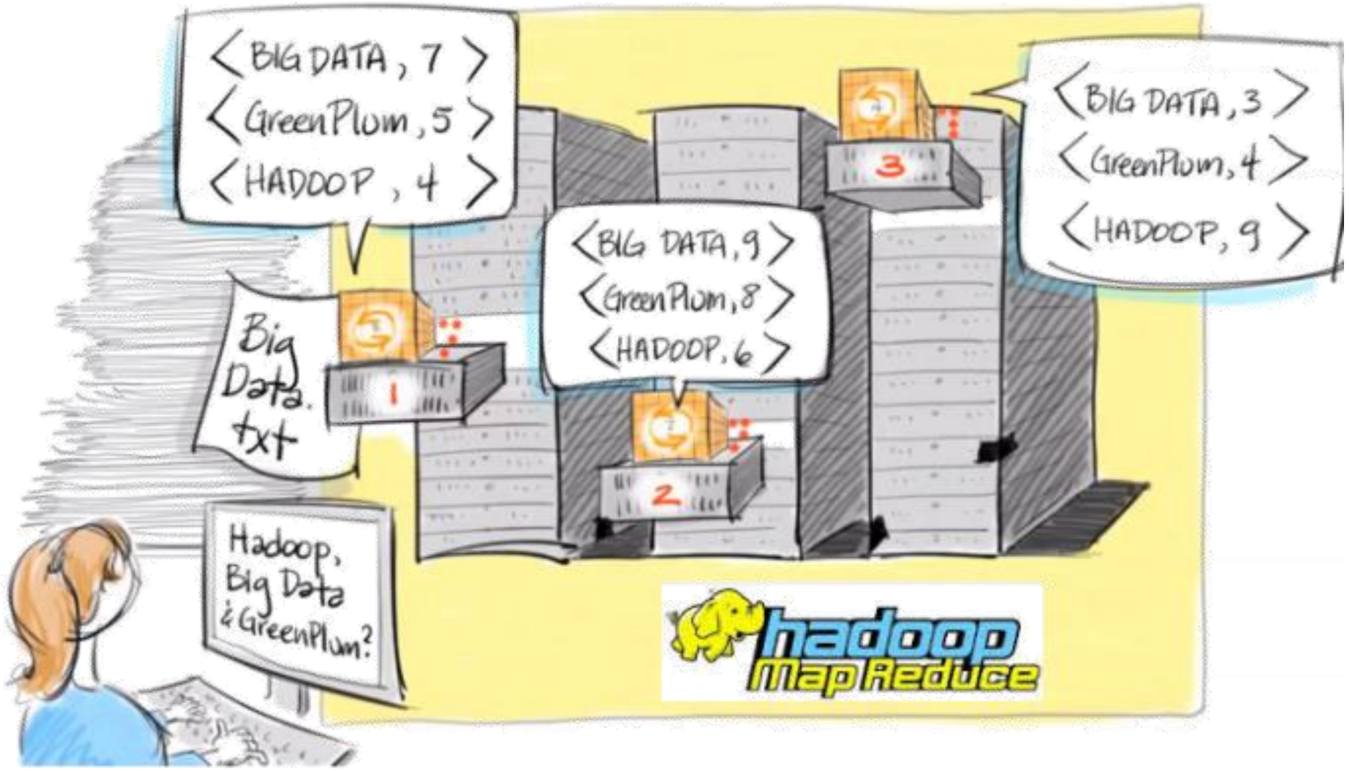
- The job execution time is sensitive to slow-running tasks.
  - E.g., hardware degradation or software misconfiguration, etc.
- Only one slow task will make the whole job take significantly longer than it would have done otherwise.
- When a job consists of hundreds or thousands of tasks, the possibility of a few straggling tasks is very real.
- Hard to detect the cause because the tasks still complete successfully, albeit after a longer time than expected

Speculative execution

- Hadoop does not try to diagnose and fix slow-running tasks
- **Speculative execution:** Hadoop tries to detect when a task is running slower than expected and launches another equivalent task as a backup.
  - The scheduler tracks the progress of all tasks of the same type (map/ reduce) in a job, and launches speculative duplicates only for a small proportion running significantly slower than the average.
  - When a task completes successfully, all duplicate tasks are killed.
- Speculative execution is an optimization, and not a feature to make jobs run more reliably.
  - E.g., problems caused by user-code bugs

Speculative execution

- There are cases that you may want to turn speculative execution off!
- Speculative execution can reduce the overall throughput on a busy cluster.
- Speculative execution for reduce tasks can significantly increase network traffic on the cluster.
- Speculative execution for nonidempotent tasks

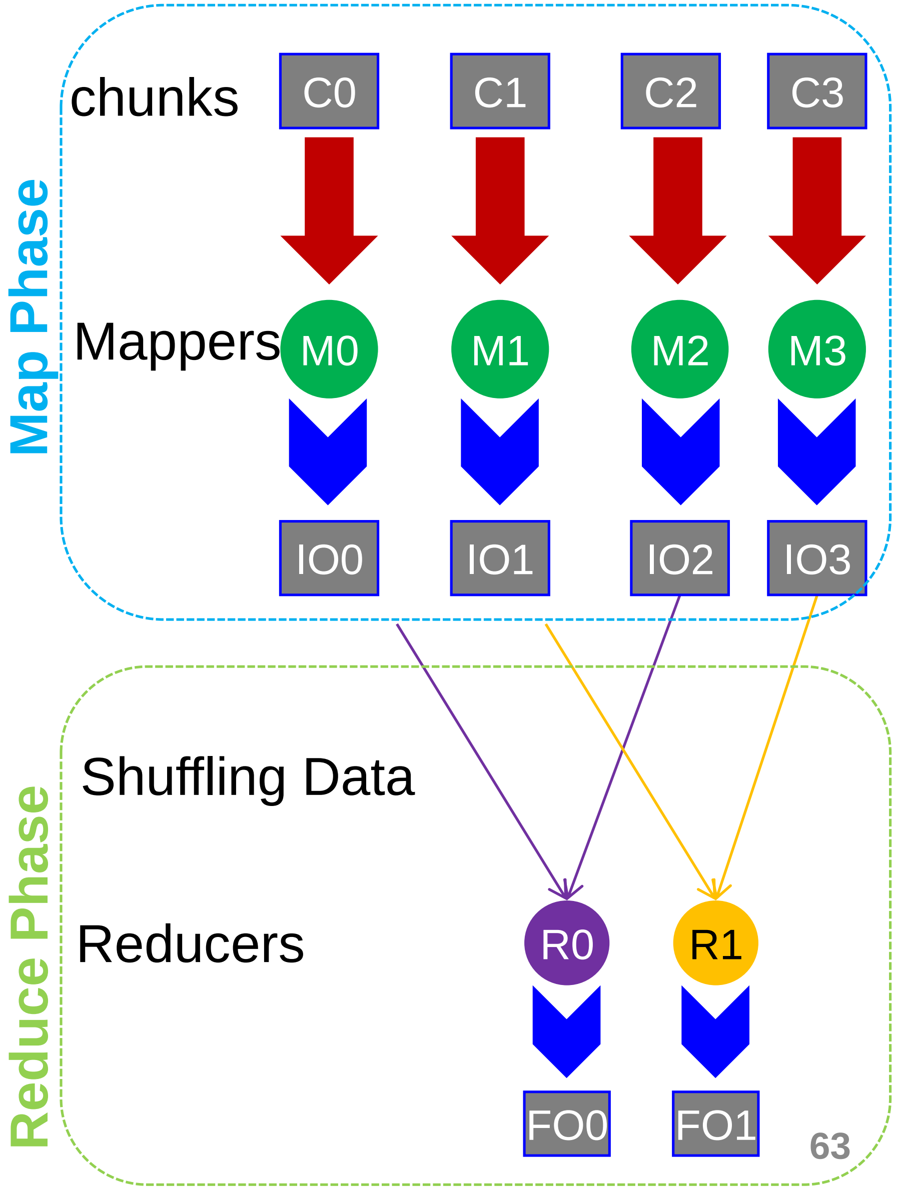


How MapReduce works

Map – Reduce – Shuffle and Sort

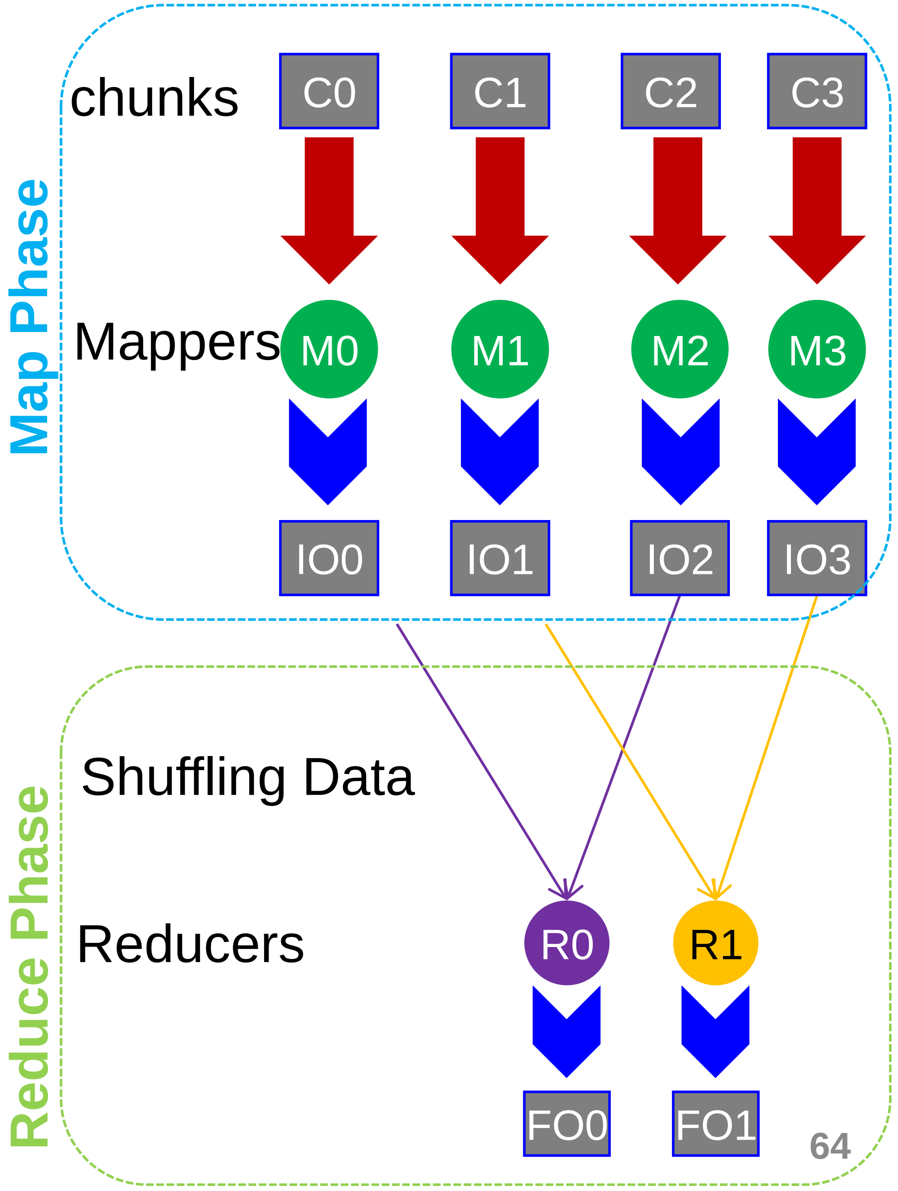
Phases of a MapReduce job

- MapReduce breaks the data flow into two phases, **map phase** and **reduce phase.**
- First, chunks are processed in isolation by **Mappers.**
- The intermediate outputs (IOs) from the Mappers brought into a second set of tasks called **Reducers**.



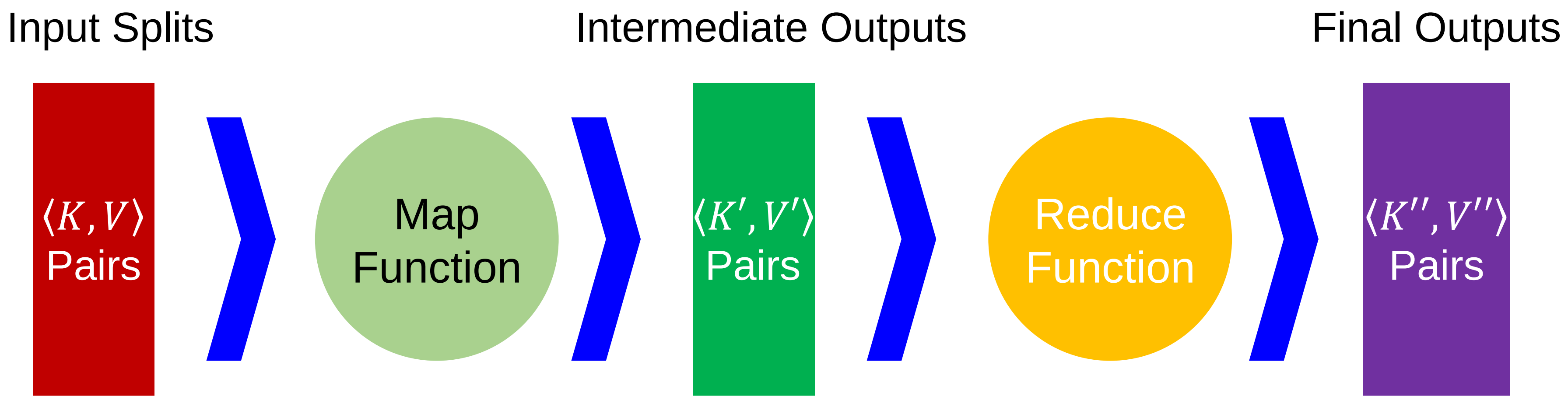
Phases of a MapReduce job

- The process of bringing together IOs into a set of Reducers is known as "**Sort and shuffling process**".
- The Reducers produce the final outputs (FOs).



Keys and Values

- The **map function** and **reduce function** are specified in a MapReduce program to implement Mapper and Reducer.
- MapReduce data elements are key-value $\langle \boldsymbol{K}, \boldsymbol{V} \rangle$ pairs.
- The map and reduce functions receive and emit $\langle K, V \rangle$ pairs.

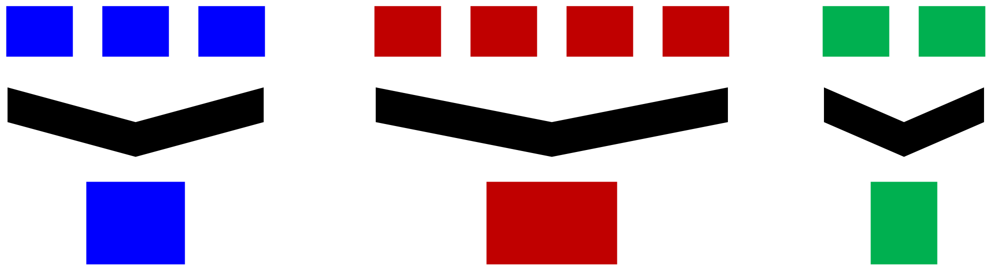

Input Splits
Intermediate Outputs
Final Outputs
$\langle K, V \rangle$ Pairs
Map Function
$\langle K', V' \rangle$ Pairs
Reduce Function
$\langle K'', V'' \rangle$ Pairs

Partitions

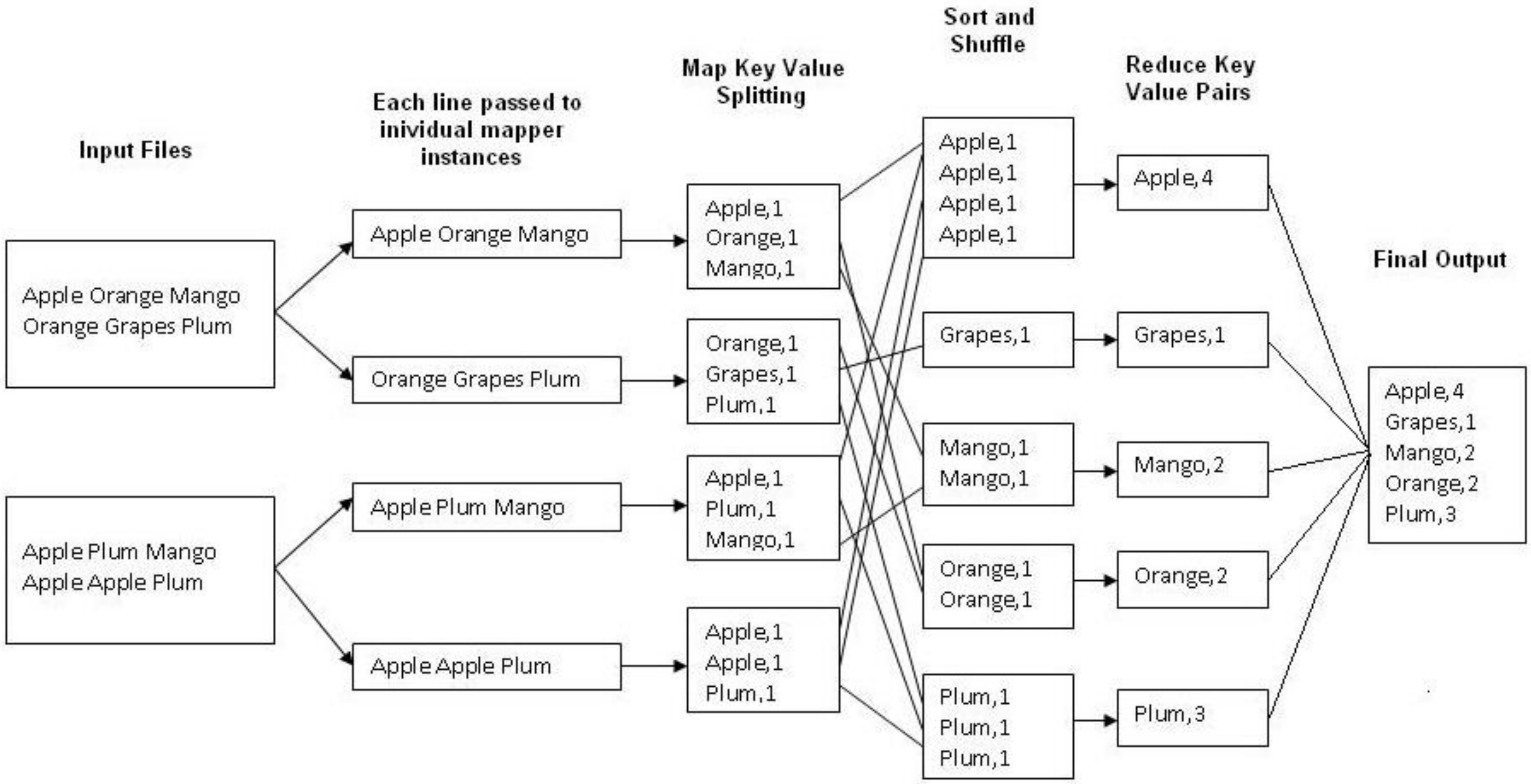
- All values with the same key are presented to a single Reducer together.
- Different subsets of the intermediate key space, which are called **partitions**, are assigned to **different Reducers**.

Different colors represent different keys (potentially) from different Mappers

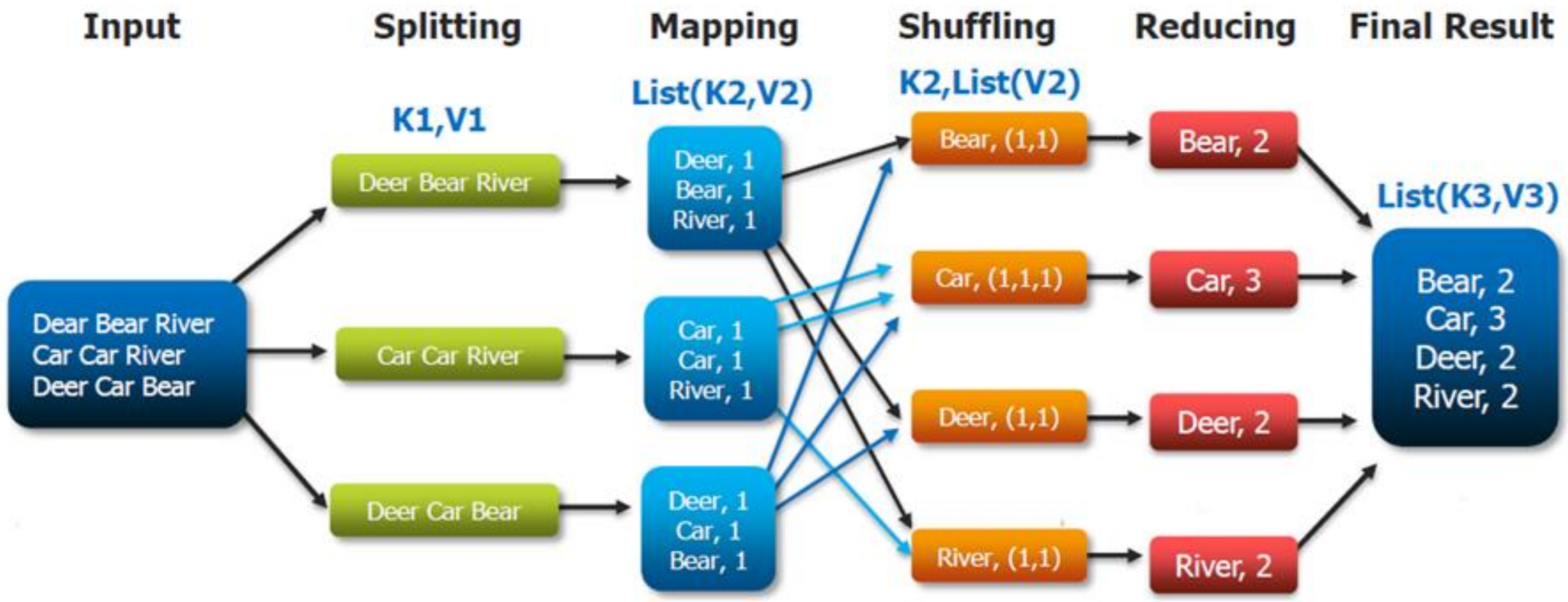
Partitions are the input to Reducers



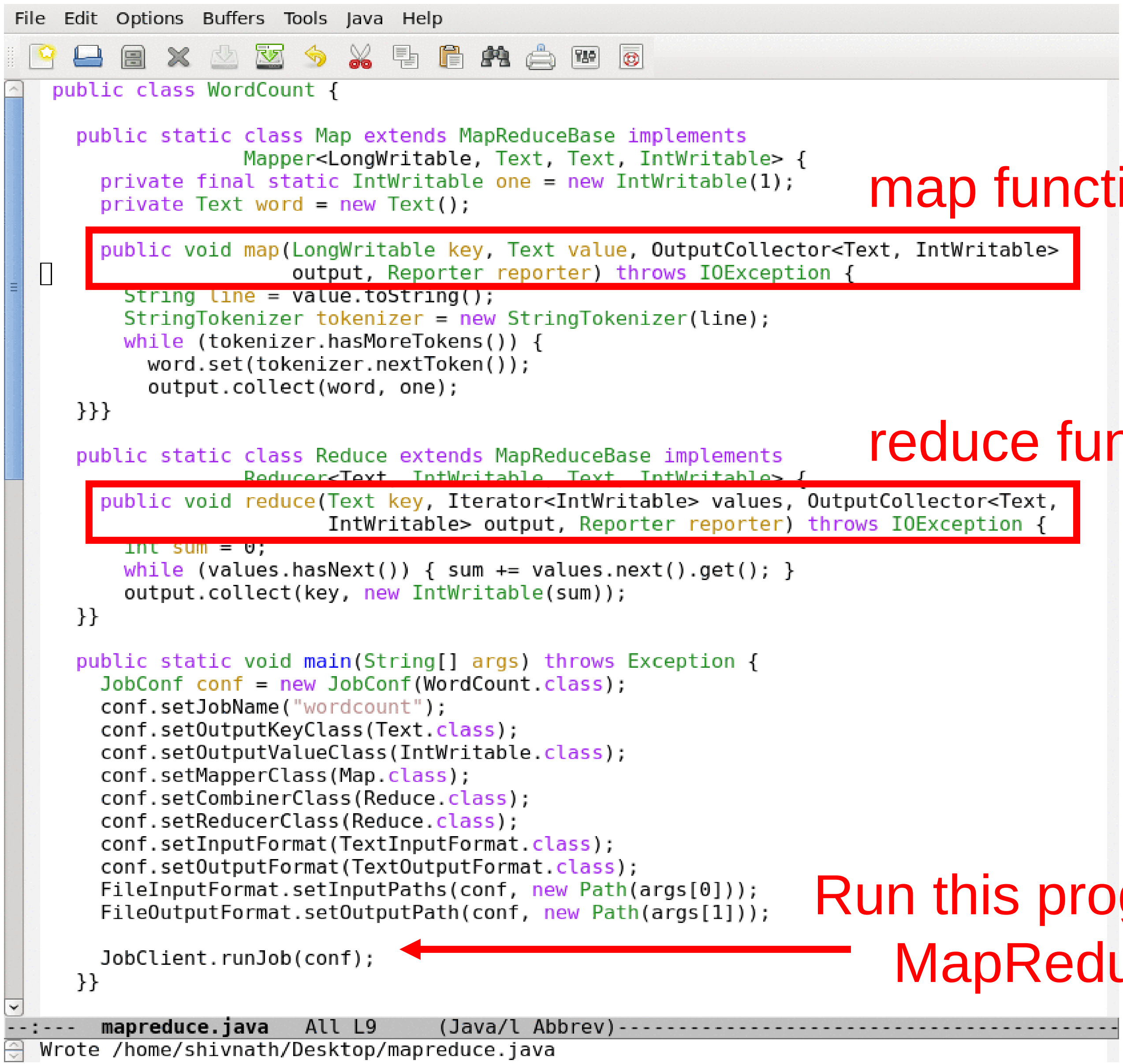
MapReduce example: Word count



MapReduce example: Word count



Lifecycle of a MapReduce job



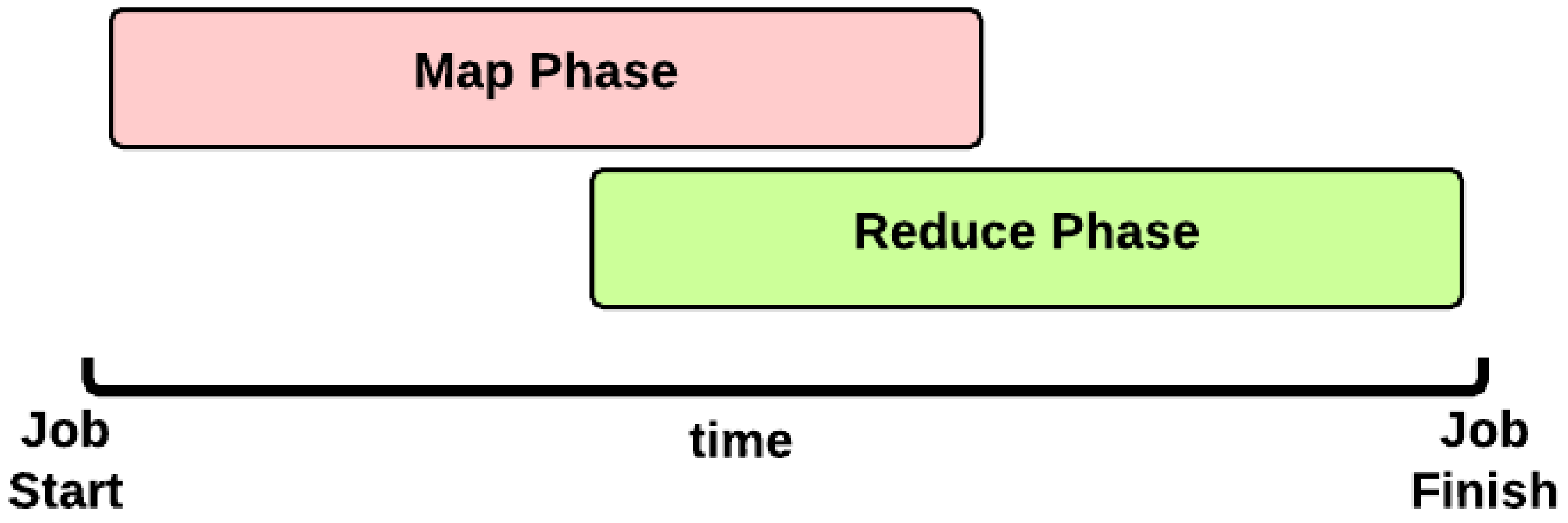
```java
public class WordCount {

  public static class Map extends MapReduceBase implements
                Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable>
                    output, Reporter reporter) throws IOException {
      String line = value.toString();
      StringTokenizer tokenizer = new StringTokenizer(line);
      while (tokenizer.hasMoreTokens()) {
        word.set(tokenizer.nextToken());
        output.collect(word, one);
  }}}

  public static class Reduce extends MapReduceBase implements
                Reducer<Text, IntWritable, Text, IntWritable> {
    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(); }
      output.collect(key, new IntWritable(sum));
  }}

  public static void main(String[] args) throws Exception {
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");
    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);
    conf.setMapperClass(Map.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);
    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);
    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));

    JobClient.runJob(conf);
  }}
--:---  mapreduce.java    All L9     (Java/l Abbrev)------------------------------------
Wrote /home/shivnath/Desktop/mapreduce.java
```

map function

reduce function

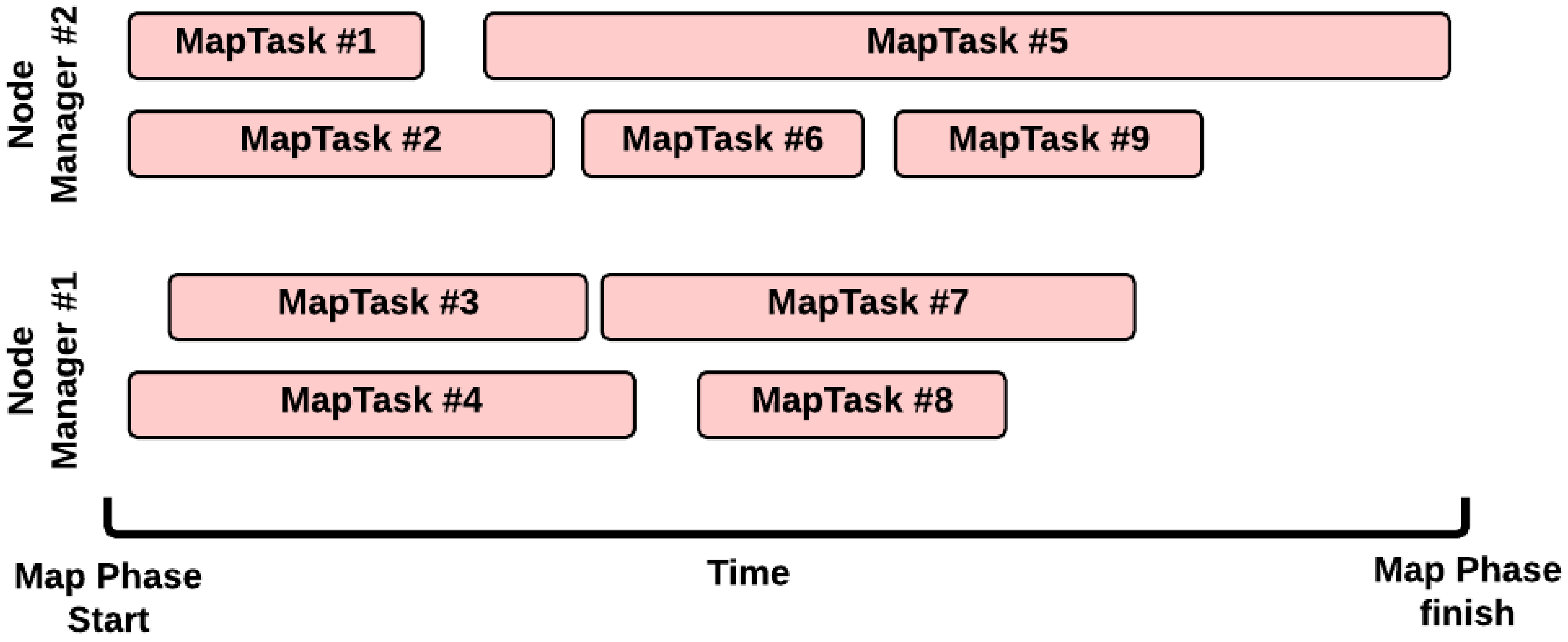
Run this program as a MapReduce job

Lifecycle of a MapReduce job



- **Map phase:** several map tasks are executed.
- **Reduce phase:** several reduce tasks are executed.
- Notice that the reduce phase may start before the end of map phase. Hence, an interleaving between them is possible.

Lifecycle of a MapReduce job



- Many map/reduce tasks can be launched on the same or different nodes.
- How are the number of splits, number of map and reduce tasks, memory allocation to tasks, etc., determined?

Mapper

- Input a pair $\langle K, V\rangle$ and output a pair $\langle K', V'\rangle$.
- For example, count words in user queries (or Tweets/Blogs).
  - The input to the Mapper will be $\langle queryID, QueryText\rangle$

⟨Q1,”The teacher went to the store. The store was closed; the store opens in the morning. The store opens at 9am.”⟩

  - The output from the Mapper would be:

⟨τhe,1⟩ ⟨teacher,1⟩ ⟨went,1⟩ ⟨to,1⟩ ⟨the,1⟩ ⟨store,1⟩ ⟨the,1⟩ ⟨store,1⟩ ⟨was,1⟩ ⟨closed,1⟩ ⟨the,1⟩ ⟨store,1⟩ ⟨opens,1⟩ ⟨in,1⟩ ⟨the,1⟩ ⟨morning,1⟩ ⟨the,1⟩ ⟨store,1⟩ ⟨opens,1⟩ ⟨at,1⟩ ⟨9am,1⟩

Reducer

- Accept the Mapper output and aggregate values on the key.
- In the previous example, the Reducer input would be:

⟨τhe,1⟩ ⟨teacher,1⟩ ⟨went,1⟩ ⟨to,1⟩ ⟨the,1⟩ **⟨store,1⟩** ⟨the,1⟩ **⟨store,1⟩** ⟨was,1⟩ ⟨closed,1⟩ ⟨the,1⟩ **⟨store,1⟩** ⟨opens,1⟩ ⟨in,1⟩ ⟨the,1⟩ ⟨morning,1⟩ ⟨the,1⟩ **⟨store,1⟩** ⟨opens,1⟩ ⟨at,1⟩ ⟨9am,1⟩

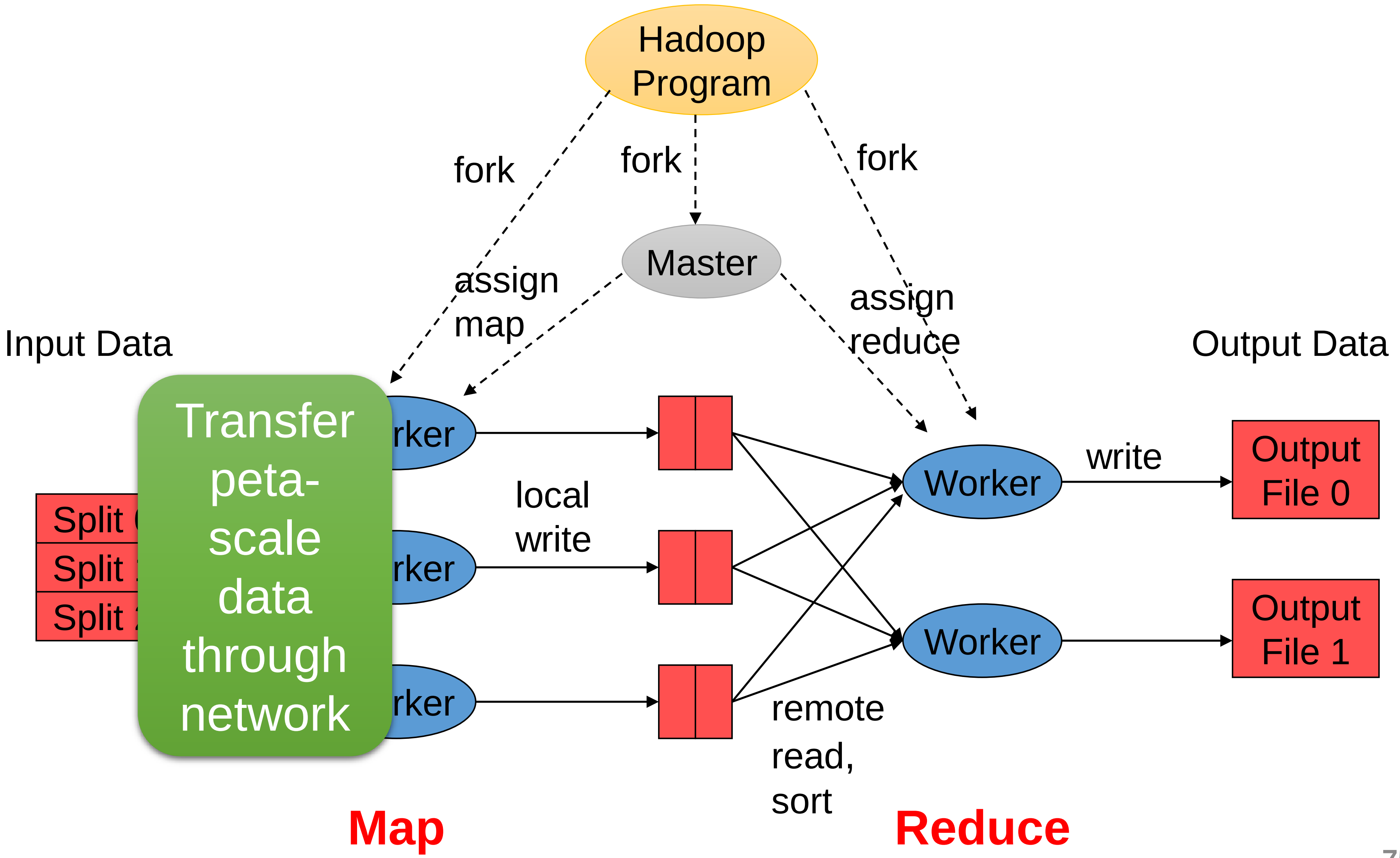
- The output would be:

⟨the,6⟩ ⟨teacher,1⟩ ⟨went,1⟩ ⟨to,1⟩ **⟨store,4⟩** ⟨was,1⟩ ⟨closed,1⟩ ⟨opens,1⟩ ⟨in,1⟩ ⟨ morning,1⟩ ⟨at,1⟩ ⟨9am,1⟩

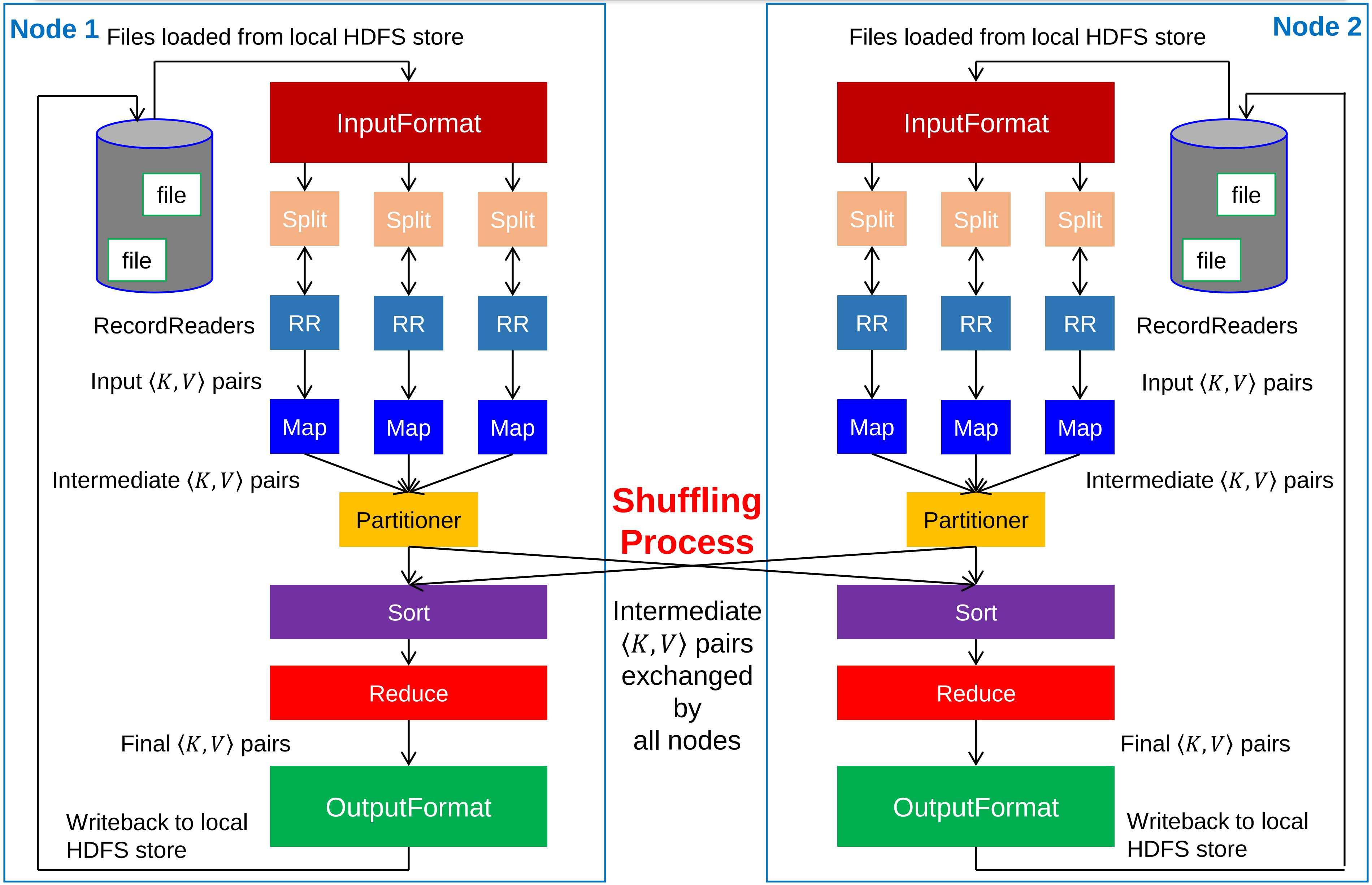
Handling data with MapReduce

- Handle more data? Just add more Mappers/ Reducers!
- No need to handle multithreaded code.
    - Mappers and Reducers are typically single-threaded and deterministic, allowing for restarting of failed jobs.
- Mappers/Reducers run **entirely independent** of each other in separate JVMs.

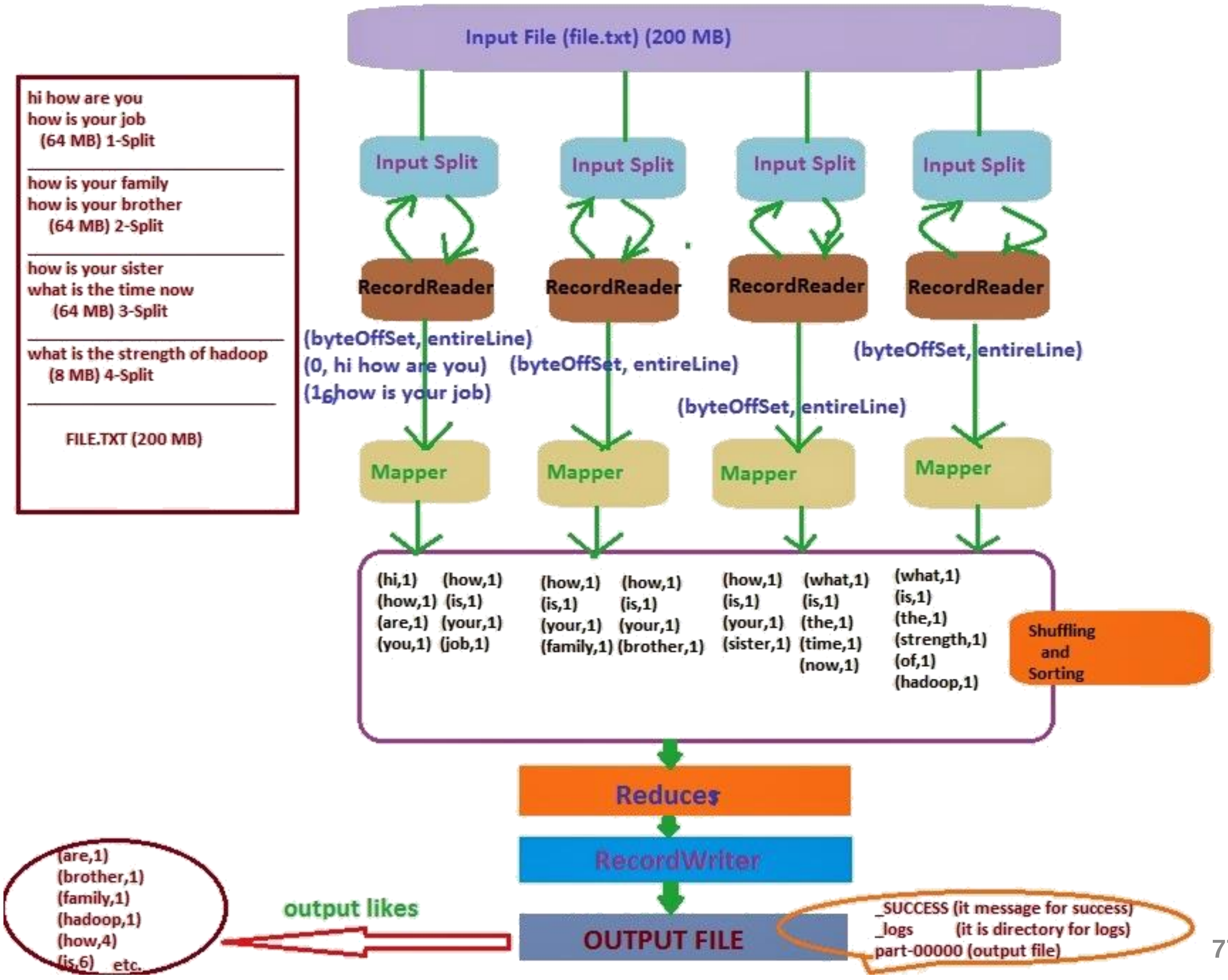
MapReduce workflow



MapReduce workflow



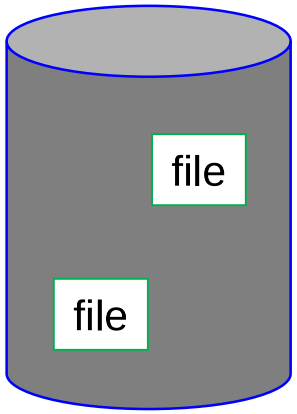
Node 1
Files loaded from local HDFS store
file
file
InputFormat
Split
Split
Split
RecordReaders
RR
RR
RR
Input $\langle K, V \rangle$ pairs
Map
Map
Map
Intermediate $\langle K, V \rangle$ pairs
Partitioner
Sort
Reduce
Final $\langle K, V \rangle$ pairs
OutputFormat
Writeback to local HDFS store
Shuffling Process
Intermediate $\langle K, V \rangle$ pairs exchanged by all nodes
Node 2
Files loaded from local HDFS store
InputFormat
Split
Split
Split
file
file
RR
RR
RR
RecordReaders
Input $\langle K, V \rangle$ pairs
Map
Map
Map
Intermediate $\langle K, V \rangle$ pairs
Partitioner
Sort
Reduce
Final $\langle K, V \rangle$ pairs
OutputFormat
Writeback to local HDFS store



Input File (file.txt) (200 MB)
hi how are you
how is your job
(64 MB) 1-Split
how is your family
how is your brother
(64 MB) 2-Split
how is your sister
what is the time now
(64 MB) 3-Split
what is the strength of hadoop
(8 MB) 4-Split
FILE.TXT (200 MB)
Input Split
Input Split
Input Split
Input Split
RecordReader
RecordReader
RecordReader
RecordReader
(byteOffSet, entireLine)
(0, hi how are you)
(16,how is your job)
(byteOffSet, entireLine)
(byteOffSet, entireLine)
(byteOffSet, entireLine)
Mapper
Mapper
Mapper
Mapper
(hi,1) (how,1) (are,1) (you,1)
(how,1) (is,1) (your,1) (job,1)
(how,1) (is,1) (your,1) (family,1)
(how,1) (is,1) (your,1) (brother,1)
(how,1) (is,1) (your,1) (sister,1)
(what,1) (is,1) (the,1) (time,1) (now,1)
(what,1) (is,1) (the,1) (strength,1) (of,1) (hadoop,1)
Shuffling and Sorting
Reduces
RecordWriter
OUTPUT FILE
output likes
(are,1)
(brother,1)
(family,1)
(hadoop,1)
(how,4)
(is,6) etc.
_SUCCESS (it message for success)
_logs (it is directory for logs)
part-00000 (output file)

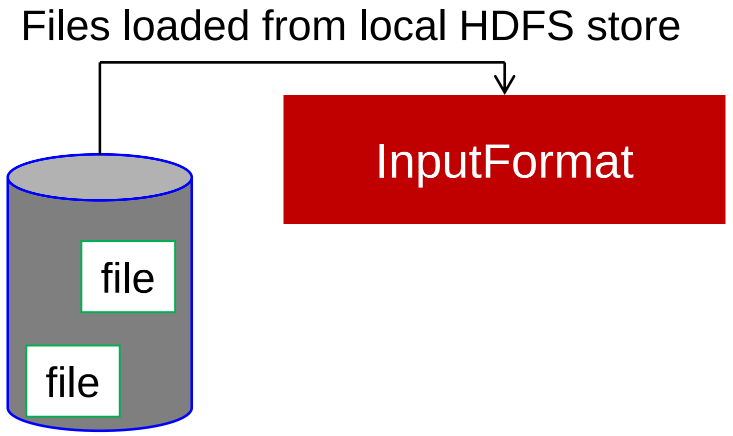
MapReduce workflow: Input files

- **Input files** are where the data for a MapReduce task is initially stored, typically resides in a distributed filesystem.
- The format of input files is arbitrary.
    - Line-based log files, binary files, multi-line input records, or something else entirely.



MapReduce workflow: InputFormat

- How the input files are split up and read is defined by the **InputFormat**, which is a class that does the following:
  - Select the files that should be used for input.
  - Define the InputSplits that break a file.
  - Provide a factory for RecordReader objects that read the file.



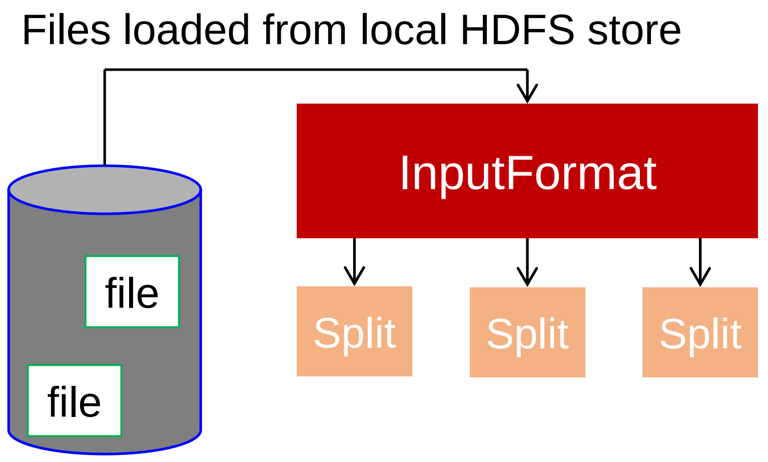
MapReduce workflow: InputFormat

- Several InputFormats are provided with Hadoop.

InputFormat	Description	Key	Value
TextInputFormat	Default format; reads lines of text files	The byte offset of the line	The line contents
KeyValueInputFormat	Parses lines into $\langle K, V \rangle$ pairs	Everything up to the first tab character	The remainder of the line
SequenceFileInputFormat	A Hadoop-specific high-performance binary format	User-defined	User-defined

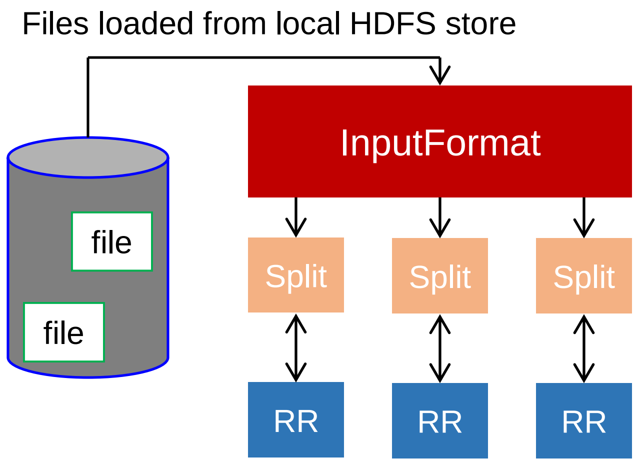
MapReduce workflow: Input splits

- An **input split** describes a unit of work that comprises a single Map task in a MapReduce program.
  - That is, each Map task corresponds to a single input split.
  - By default, the InputFormat breaks a file up into 64MB splits.
- By dividing the file into splits, **several Map tasks** are able to operate on a single file **in parallel**.
  - This can improve the performance on large files significantly.



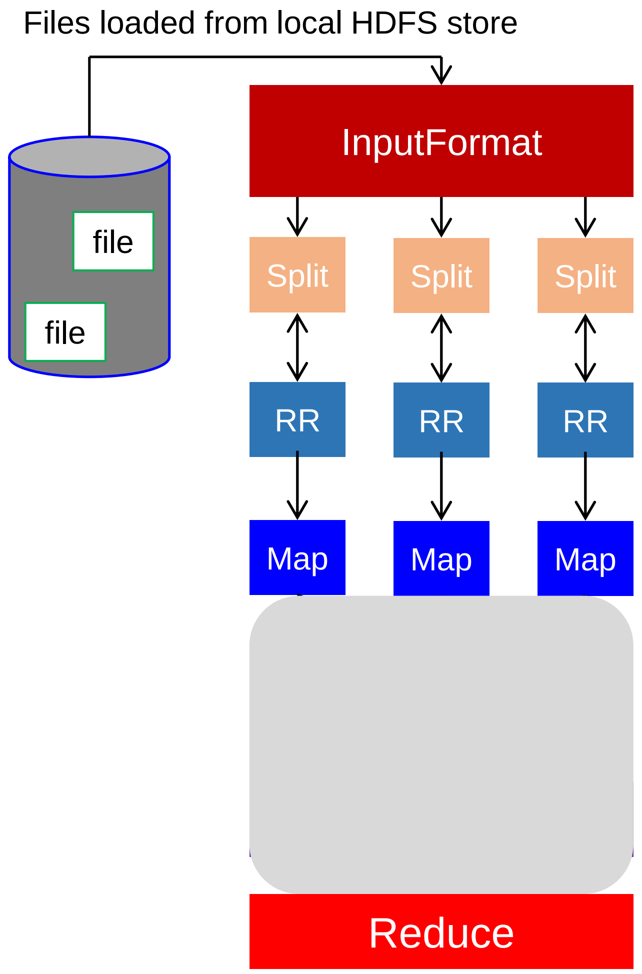
MapReduce workflow: RecordReader

- The input split defines a slice of work but does not describe how to access it.
- The **RecordReader** class actually loads data from its source and converts it into $\langle K, V \rangle$ pairs suitable for Mappers.
    - It is invoked repeatedly on the input until the entire split is consumed.



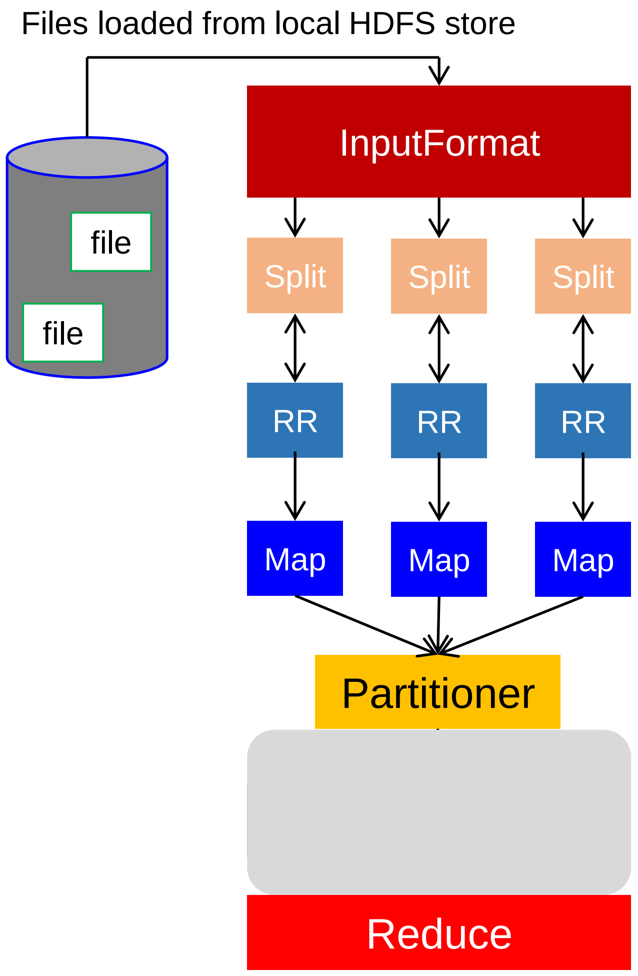
MapReduce workflow: Mapper – Reducer

- The **Mapper** performs the user-defined work of the Map phase.
  - A new Mapper instance is created for each split.
- The **Reducer** performs the user-defined work of the Reduce phase.
  - A new Reducer instance is created for each partition
  - For each key in the partition assigned to a Reducer, the Reducer is called once.



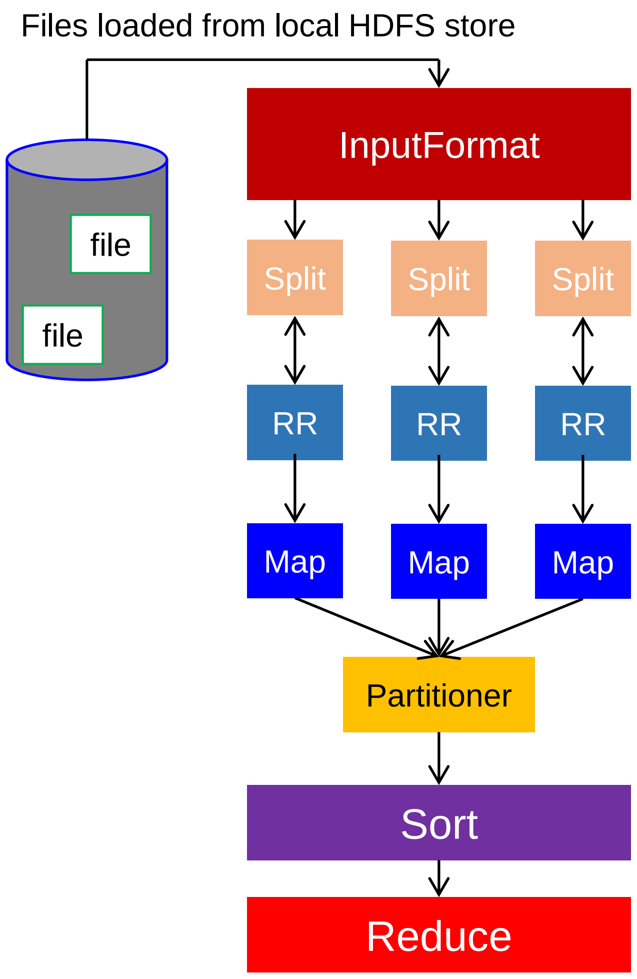
MapReduce workflow: Partitioner

- Each Mapper may emit $\langle K, V \rangle$ pairs to any partition.
  - The map nodes must all agree on where to send different pieces of intermediate data.
- The **partitioner** class determines which partition a given $\langle K, V \rangle$ pair will go to.
- The default HashPartitioner computes a hash value for a given key and assigns it to a partition based on this result.



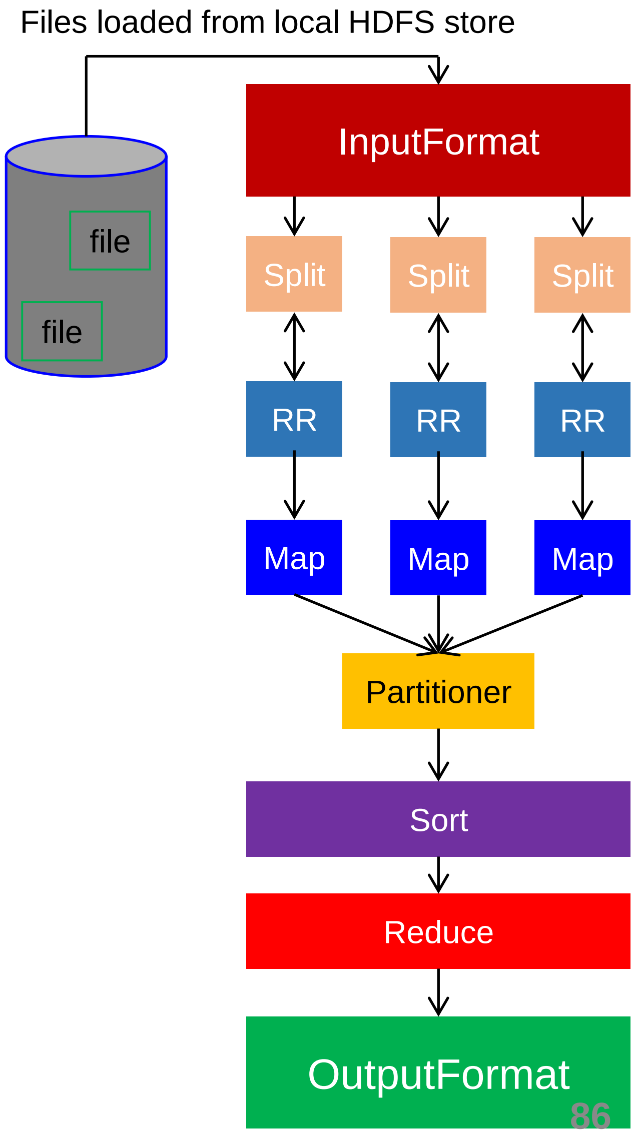
MapReduce workflow: Sort

- The intermediate $\langle K, V \rangle$ pairs can be sorted in both map phase and reduce phase.
- In map phase: during the process of partition, sort and spill to disk.
- In reduce phase: during the process of merging, before the Reducer actually works.



MapReduce workflow: OutputFormat

- The OutputFormat class defines the way $\langle K, V \rangle$ pairs produced by Reducers are written to output files.
- The instances of OutputFormat provided by Hadoop write to files on the local disk or in HDFS.



MapReduce workflow: OutputFormat

- Several OutputFormats are provided by Hadoop.

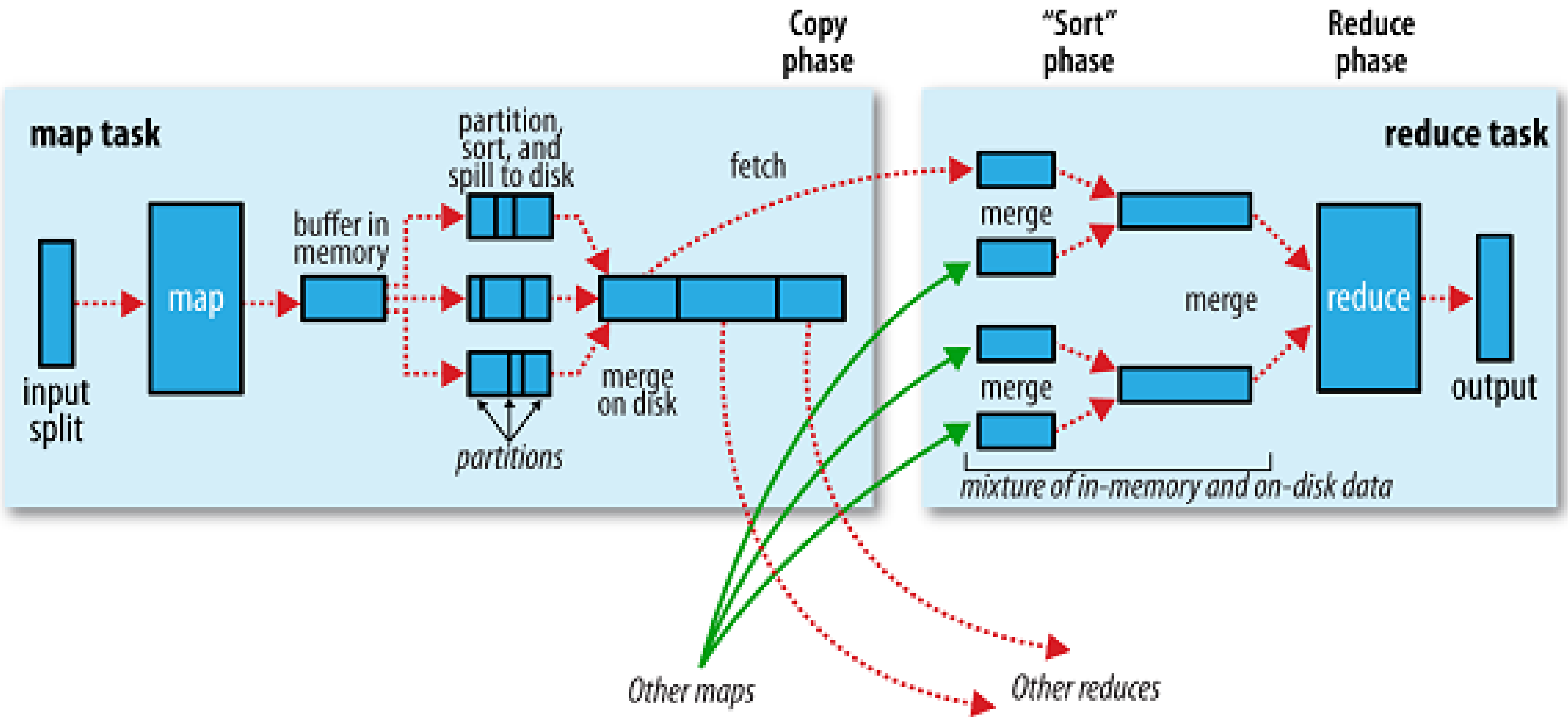
OutputFormat	Description
TextOutputFormat	Default; writes lines in "key \t value" format
SequenceFileOutputFormat	Writes binary files suitable for reading into subsequent MapReduce jobs
NullOutputFormat	Generates no output files

Shuffle and Sort

Shuffle and Sort

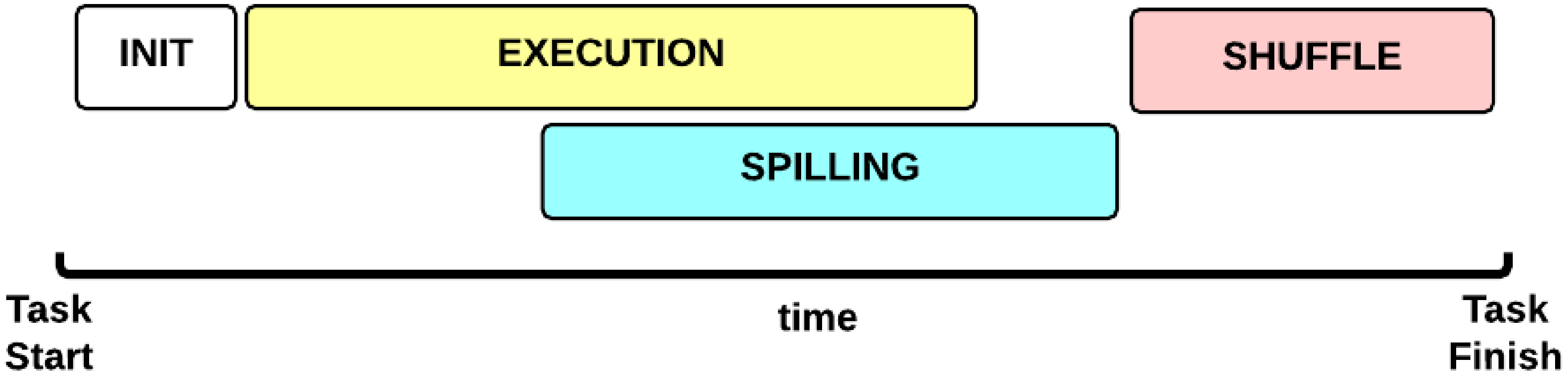
- **Shuffle (and Sort):** the process by which the system performs the sort, and transfers the map outputs to the Reducers as inputs
- It is the heart of MapReduce, allowing to **optimize** a MapReduce program.

Shuffle and sort in MapReduce



Map side

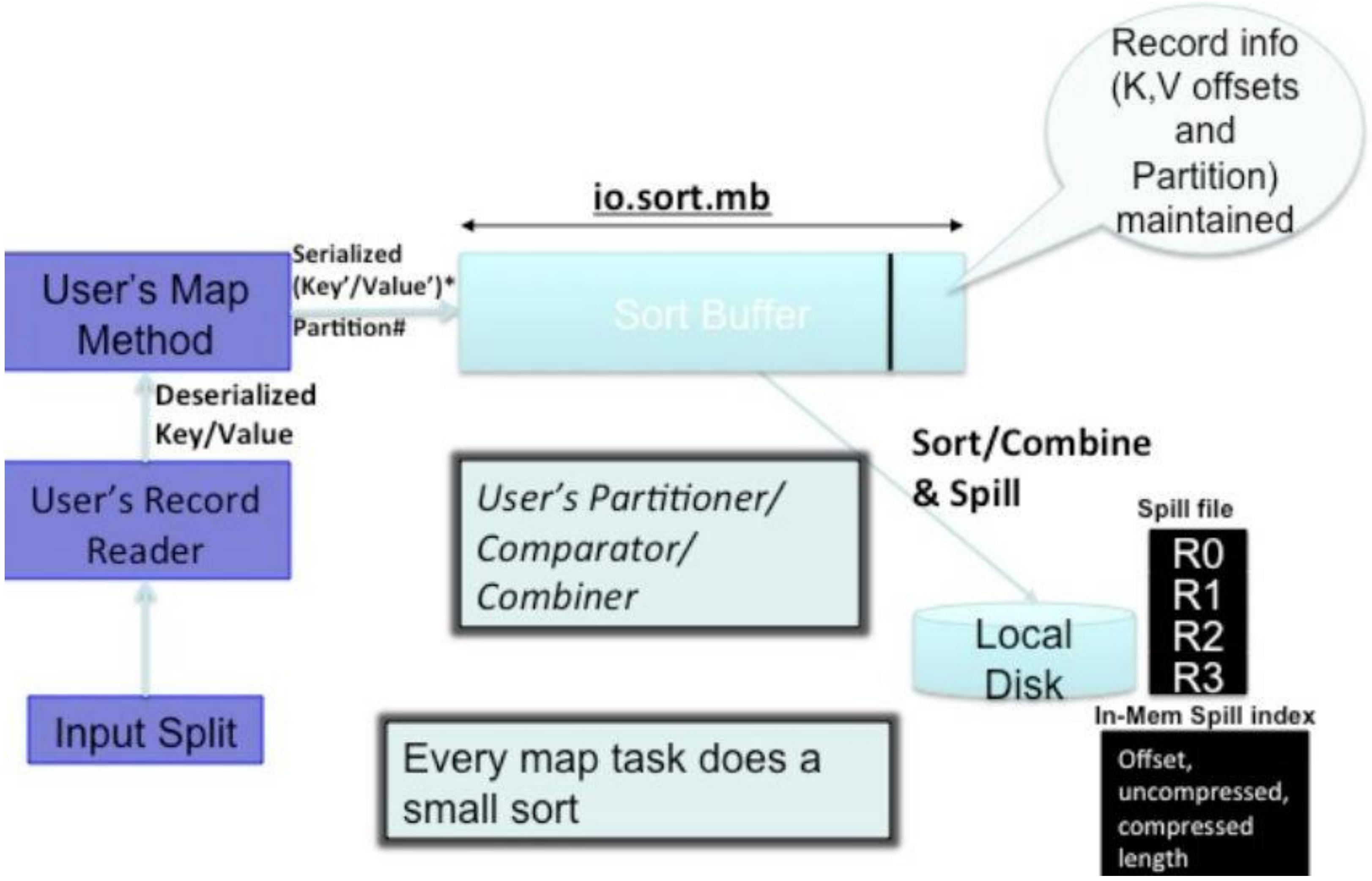
- When the map function starts producing output, it is not simply written to disk.
  - The process takes advantage of buffering writes in memory and doing some presorting for efficiency reasons.



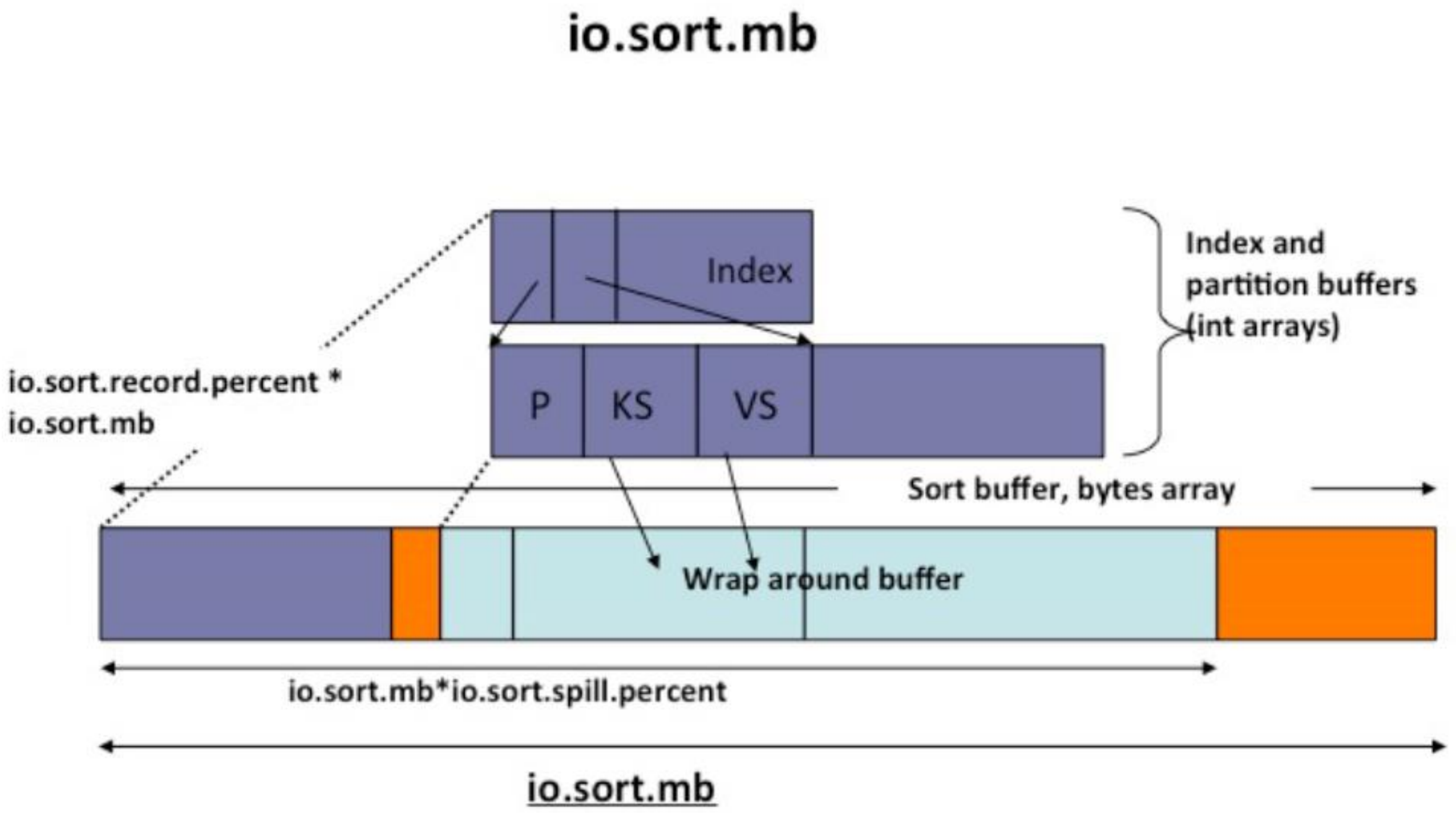
Map side

- Each map task has a **circular memory buffer** (100 MB by default) to write the output to.
- When the contents of the buffer reach a certain threshold size, a background thread will start to spill the data to disk.
- Map outputs continue to be written to the buffer while the spill takes place.
- If the buffer fills up during this time, the map will block until the spill is complete.

Map side: Sort buffer


Record info (K,V offsets and Partition) maintained
io.sort.mb
User's Map Method
Serialized (Key'/Value')*
Partition#
Sort Buffer
Deserialized Key/Value
User's Record Reader
Input Split
User's Partitioner/ Comparator/ Combiner
Sort/Combine & Spill
Spill file
R0
R1
R2
R3
Local Disk
In-Mem Spill index
Offset, uncompressed, compressed length
Every map task does a small sort

Map side: Sort buffer


io.sort.mb
Index
Index and partition buffers (int arrays)
io.sort.record.percent * io.sort.mb
P
KS
VS
Sort buffer, bytes array
Wrap around buffer
io.sort.mb*io.sort.spill.percent
io.sort.mb

Map side

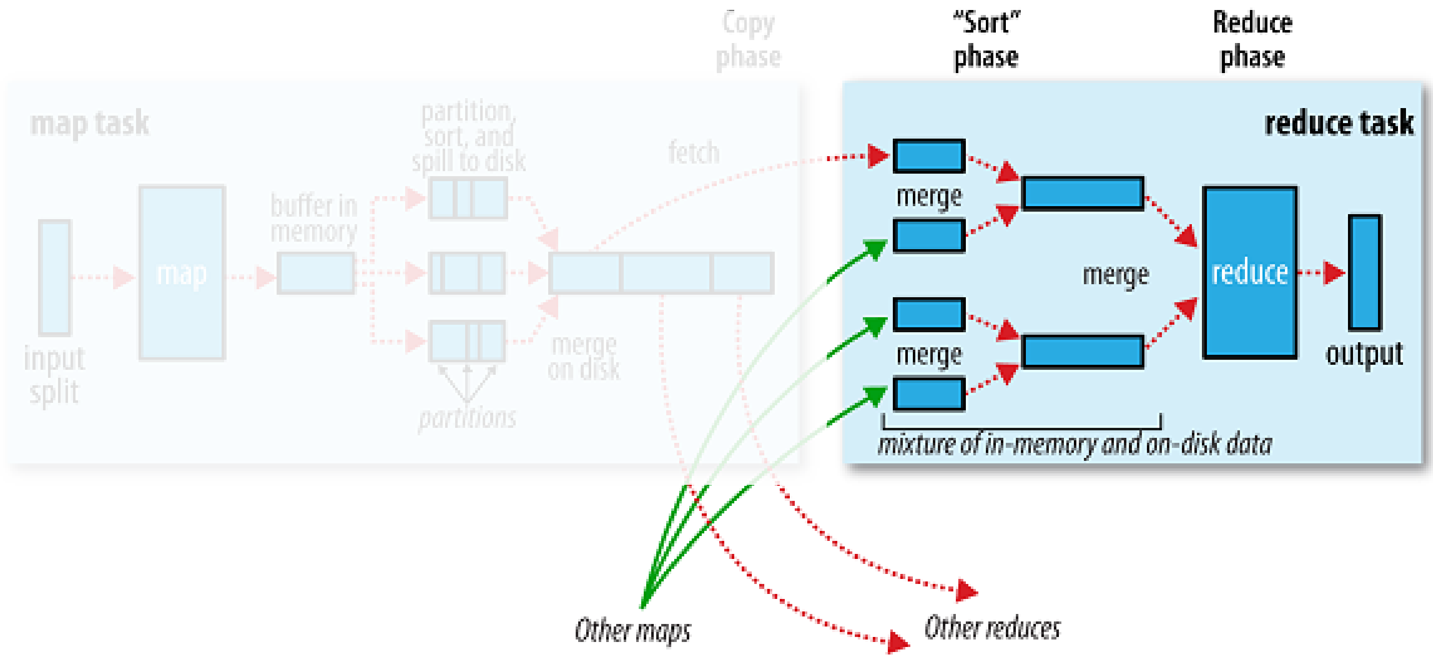
- Before writing to disk, the thread first divides the data into partitions corresponding to the Reducers that they will ultimately be sent to.
- Within each partition, the background thread performs an in-memory sort by key.
  - The map output will be more compact if a Combiner function is run on the output of the sort $\rightarrow$ less data written to local disk and transferred to the Reducer.

Map side

- Each time the memory buffer reaches the spill threshold, a new spill file is created.
  - There could be several spill files after the map task has written its last output record.
- Before the task is finished, the spill files are merged into a single partitioned and sorted output file.
  - If there are at least three spill files, run the Combiner again before the output file is written.
  - Otherwise, the Combiner will not run since the potential reduction in map output size is not worth the overhead in invoking the Combiner.

Reduce side

- The map tasks may finish at different times, so the reduce task starts copying their outputs as soon as each completes.
  - The Reduce task uses a small number of copier threads to fetch map outputs in parallel.
- This is known as the **copy phase** of the reduce task.



Reduce side

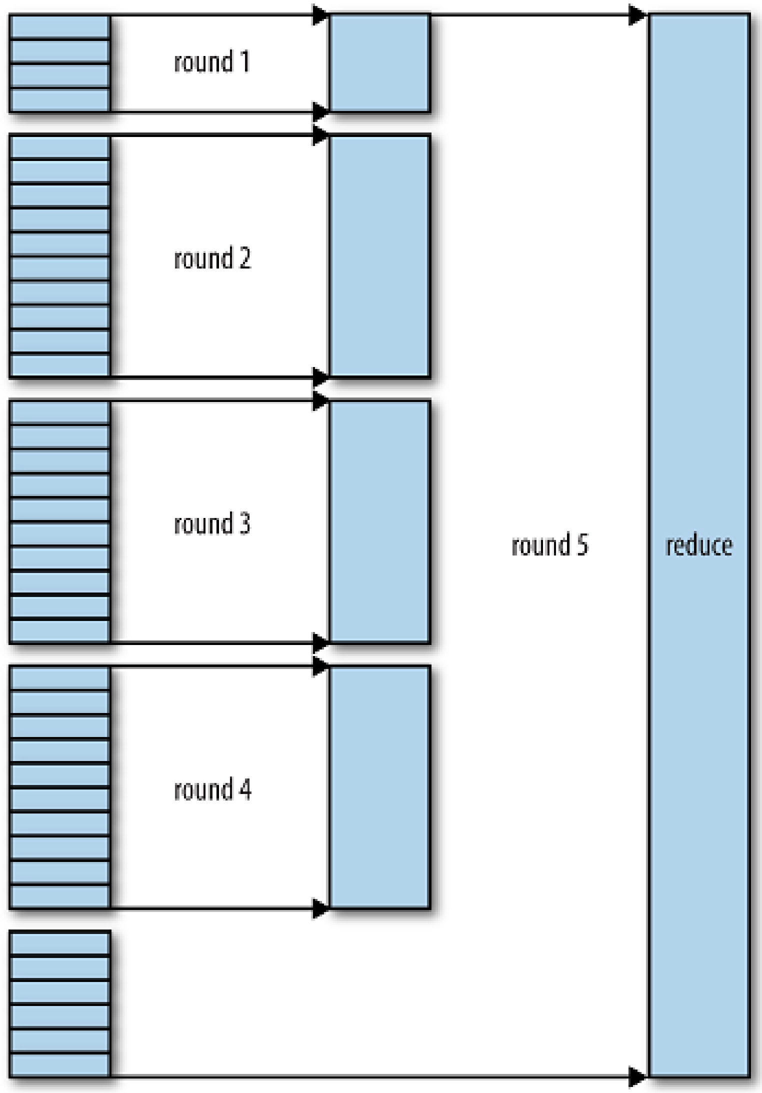
- Map outputs are copied to the Reduce task JVM’s memory if they are small enough; otherwise, they are copied to disk.
- When the in-memory buffer reaches a threshold size or reaches a threshold number of map outputs, it is merged and spilled to disk.
  - If a Combiner is specified, it will be run during the merge to reduce the amount of data written to disk.

Reduce side

- As the copies accumulate on disk, a background thread merges them into larger, sorted files.
    - This saves some time merging later on.
- Note that any map outputs that were compressed (by map task) have to be decompressed in memory in order to perform a merge on them.

Reduce side

- When all the map outputs have been copied, the Reducer moves into the Sort phase, which merge the outputs **in rounds** and maintaining their sort ordering.
- E.g., if there were 50 Map outputs and the merge factor was 10, there would be 5 rounds.
    - Each round would merge 10 files into 1, so at the end there would be 5 intermediate file.



Reduce side

- Rather than have a final round that merges these five files into a single sorted file, the merge directly feed the Reduce function and the last phase, the Reduce phase, starts.
- This final merge can come from a mixture of in-memory and on-disk segments.

A simple example

Take a glance at a simple Java MapReduce program

About the dataset

- Weather sensors collect data every hour at many locations across the globe and gather a large volume of log data.
- National Climatic Data Center, or NCDC.
- The data is stored following the line-oriented ASCII format.
    - Each line is a record.
    - A rich set of meteorological elements, many of which are optional or with variable data lengths.

```
0057
332130   # USAF weather station identifier
99999    # WBAN weather station identifier
19500101 # observation date
0300     # observation time
4
+51317   # latitude (degrees x 1000)
+028783  # longitude (degrees x 1000)
FM-12
+0171    # elevation (meters)
99999
V020
320      # wind direction (degrees)
1        # quality code
N
0072
1
00450    # sky ceiling height (meters)
1        # quality code
C
N
010000   # visibility distance (meters)
1        # quality code
N
9
-0128    # air temperature (degrees Celsius x 10)
1        # quality code
-0139    # dew point temperature (degrees Celsius x 10)
1        # quality code
10268    # atmospheric pressure (hectopascals x 10)
1        # quality code
```

About the dataset

- Datafiles are organized by date and weather station.
  - A directory for each year from 1901 to 2001, each containing a gzipped file for each weather station with its readings for that year.
- E.g., the first entries for 1990

```
% ls raw/1990 | head
010010-99999-1990.gz
010014-99999-1990.gz
010015-99999-1990.gz
010016-99999-1990.gz
010017-99999-1990.gz
010030-99999-1990.gz
010040-99999-1990.gz
```

Analyzing the data

- ***Purpose**: What's the highest recorded global temperature for each year in the dataset?*
- Without using Hadoop:
  - Classic tool for processing line-oriented data is awkward.
  - The complete run for the century took **42 minutes** in one run on a single EC2 High-CPU Extra Large instance.
- With Hadoop:
  - The same program runs, without alteration, on a full dataset.
  - That took **six minutes** on a 10-node EC2 cluster running High-CPU Extra Large instances.

Analyzing the data: Map function

- The only needed fields are **year** and **air temperature**.
- The **map function** is just a data preparation phase
  - Set up the data in such a way that the **reduce function** can find the maximum temperature for each year.
- Drop bad records by filtering out temperatures that are missing, suspect, or erroneous, etc.

Analyzing the data: Map function

- For example, consider the following sample lines of input data

```
0067011990999991950051507004...9999999N9+00001+99999999999...
0043011990999991950051512004...9999999N9+00221+99999999999...
0043011990999991950051518004...9999999N9-00111+99999999999...
0043012650999991949032412004...0500001N9+01111+99999999999...
0043012650999991949032418004...0500001N9+00781+99999999999...
```

- These lines are presented to the Map function as the key-value pairs

```
(0, 0067011990999991950051507004...9999999N9+00001+99999999999...)
(106, 0043011990999991950051512004...9999999N9+00221+99999999999...)
(212, 0043011990999991950051518004...9999999N9-00111+99999999999...)
(318, 0043012650999991949032412004...0500001N9+01111+99999999999...)
(424, 0043012650999991949032418004...0500001N9+00781+99999999999...)
```

Analyzing the data: Map function

- Extract the year and air temperature
- Emit them as output
  - The temperature values have been interpreted as integers.
- For example,

```
(1950, 0)
(1950, 22)
(1950, −11)
(1949, 111)
(1949, 78)
```

Analyzing the data: Map function

- The emitted key-value pairs are sorted and grouped by key, before being sent to the reduce function.
- For example,

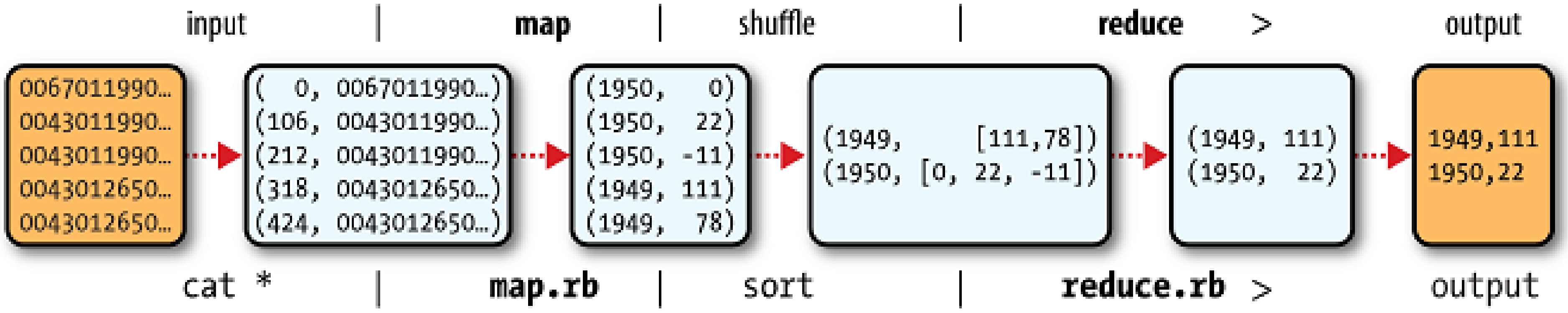
```
(1949, [111, 78])
(1950, [0, 22, −11])
```

Each year appears with a list of all its air temperature readings.

Analyzing the data: Reduce function

- Iterate through the list and pick up the maximum reading
- For example,

```
(1949, 111)

(1950, 22)
```

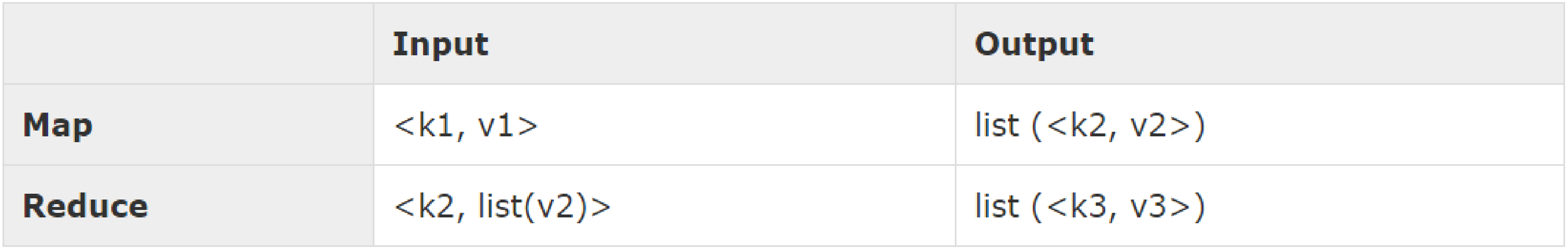


MapReduce logical data flow

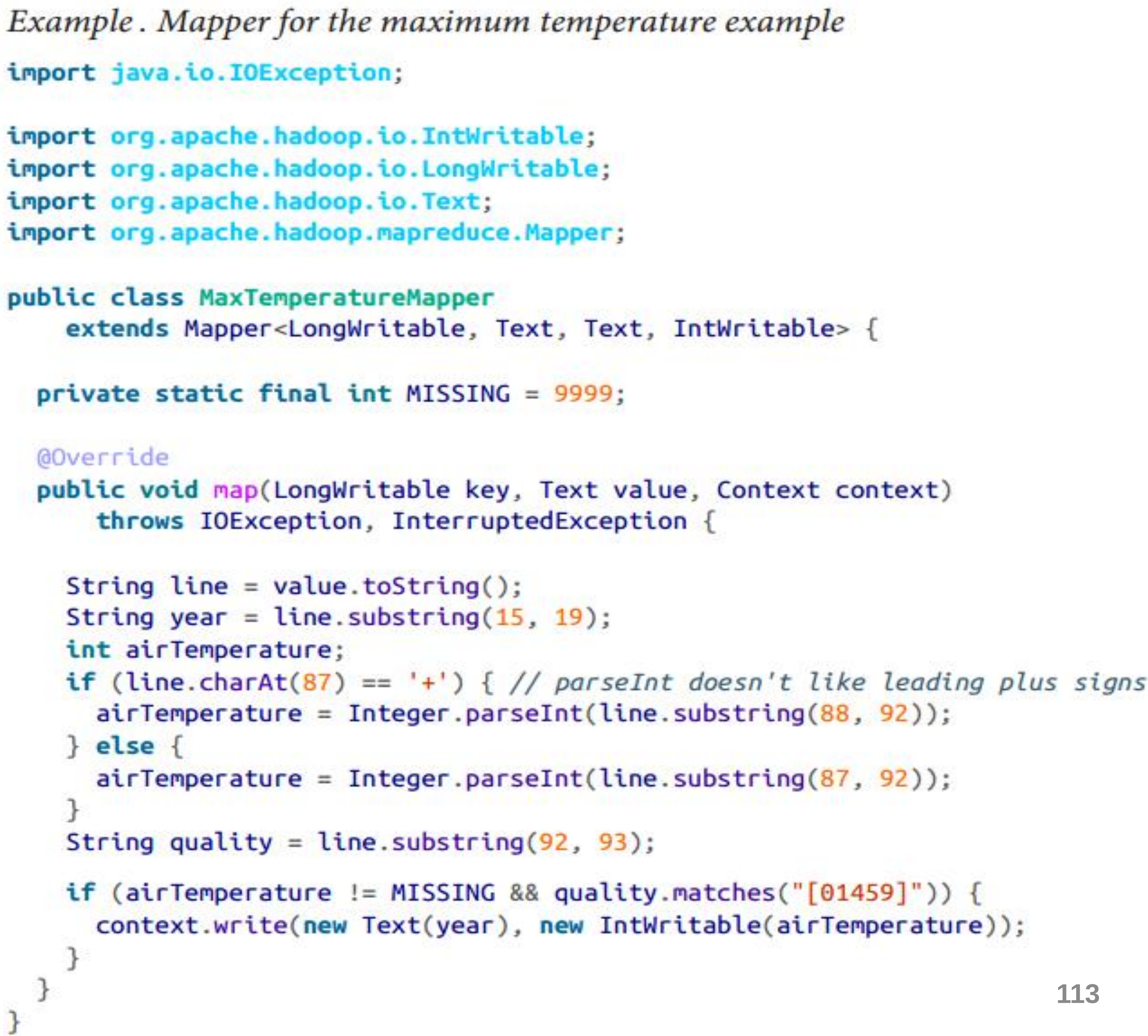
Inputs and Outputs

- The input and output types of a MapReduce job are typically

(Input) $\langle k1, v1 \rangle$ → Map → $\langle k2, v2 \rangle$ → Reduce → $\langle k3, v3 \rangle$ (Output)



	Input	Output
Map	<k1, v1>	list (<k2, v2>)
Reduce	<k2, list(v2)>	list (<k3, v3>)


Example . Mapper for the maximum temperature example

```
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class MaxTemperatureMapper
    extends Mapper<LongWritable, Text, Text, IntWritable> {

  private static final int MISSING = 9999;

  @Override
  public void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {

    String line = value.toString();
    String year = line.substring(15, 19);
    int airTemperature;
    if (line.charAt(87) == '+') { // parseInt doesn't like leading plus signs
      airTemperature = Integer.parseInt(line.substring(88, 92));
    } else {
      airTemperature = Integer.parseInt(line.substring(87, 92));
    }
    String quality = line.substring(92, 93);
    if (airTemperature != MISSING && quality.matches("[01459]")) {
      context.write(new Text(year), new IntWritable(airTemperature));
    }
  }
}
```

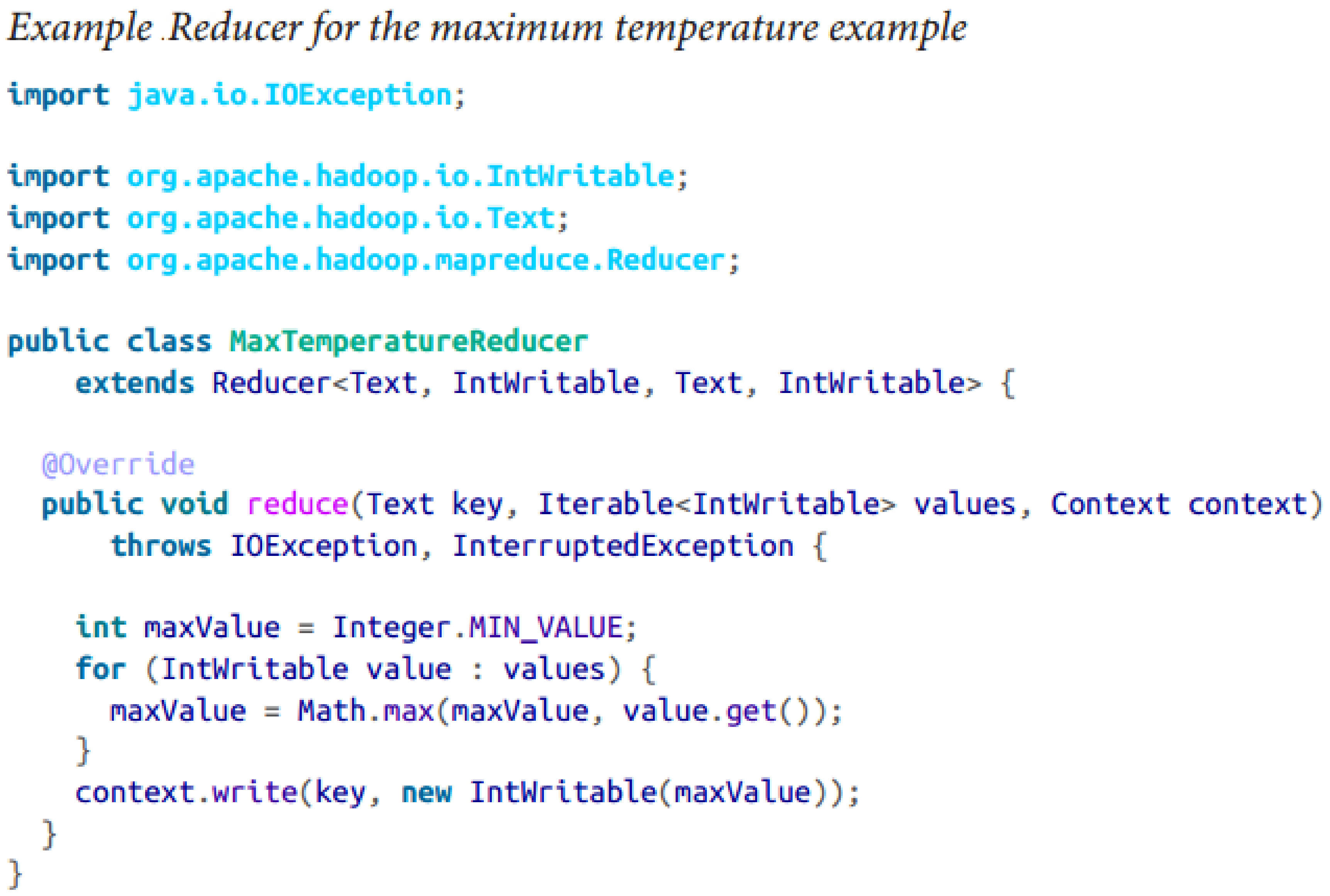
Source Code

Map function

- The generic **Mapper class** has four formal type parameters
  - Input key, input value, output key, and output value types for the Map function.
- The map() method is passed a key and a value.
- It also provides an instance of Context to write the output to.

Analyzing the data: Map function

- Convert the Text value of the input line into a Java String,
- Use the substring() method to extract the interested columns
- Write the year as a Text object (since it is just used as a key), and the temperature is wrapped in an IntWritable
    - An output record is written only if the temperature is present and the quality code indicates the temperature reading is OK.



Example .Reducer for the maximum temperature example

```java
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class MaxTemperatureReducer
    extends Reducer<Text, IntWritable, Text, IntWritable> {

  @Override
  public void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {

    int maxValue = Integer.MIN_VALUE;
    for (IntWritable value : values) {
      maxValue = Math.max(maxValue, value.get());
    }
    context.write(key, new IntWritable(maxValue));
  }
}
```

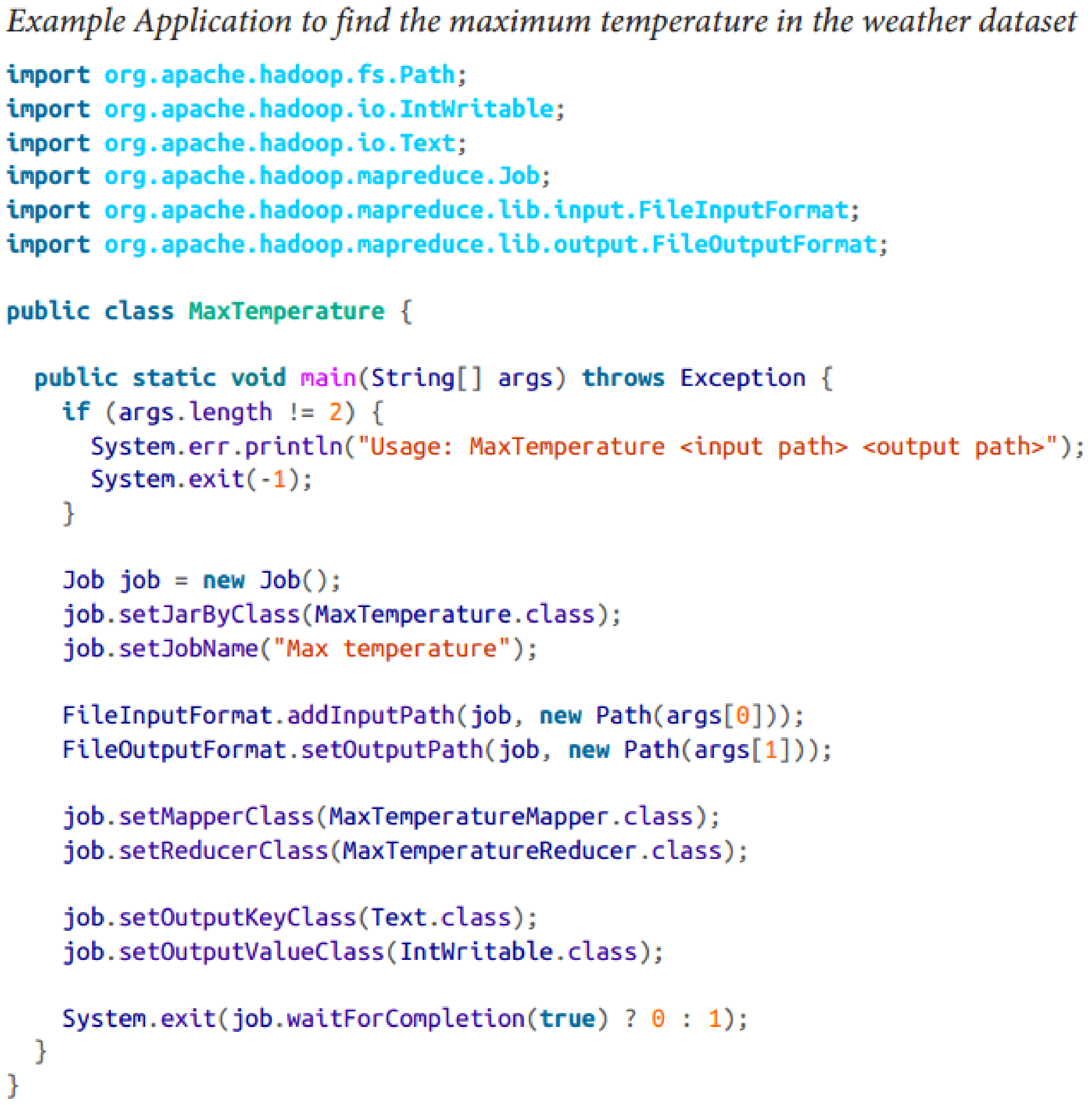
Source Code

Reduce function

- Four formal type parameters are used to specify the input and output types.
- The input types of the reduce function must match the output types of the map function.

Analyzing the data: Reduce function

- Output a record for each year and its maximum temperature
- Input types: Text and IntWritable.
- Output types: Text and IntWritable.
- Iterate through the temperatures and compare each with a record of the highest found so far



Example Application to find the maximum temperature in the weather dataset

```java
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 MaxTemperature {

  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.println("Usage: MaxTemperature <input path> <output path>");
      System.exit(-1);
    }

    Job job = new Job();
    job.setJarByClass(MaxTemperature.class);
    job.setJobName("Max temperature");

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.setMapperClass(MaxTemperatureMapper.class);
    job.setReducerClass(MaxTemperatureReducer.class);

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

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

Source Code

Application source code

- A **Job object** forms the specification of the job and control over how the job is run.
- The code is packaged into a JAR file, which Hadoop will distribute around the cluster.
    - If a class is passed in the Job’s setJarByClass() method, Hadoop locates the relevant JAR file by looking for the JAR file containing this class.

Application source code

- The **output path** (of which there is only one) is specified by the static setOutputPath() method on FileOutputFormat.
  - It specifies a directory where the output files from the reduce function are written.
- The **map** and **reduce** is specified via the setMapperClass() and setReducerClass() methods.

Application source code

- setOutputKeyClass() and setOutputValueClass(): control the output types for the Reduce function
  - It must match what the Reduce class produces.
- The map output types default to the same types as the reducer, so they do not need to be set.
  - The map output types can be set using setMapOutputKeyClass() and setMapOutputValueClass() if different.

Application source code

- The **waitForCompletion()** method on Job submits the job and waits for it to finish.
  - The single flag argument indicates whether information about the progress is printed to the console.
  - The return value is a Boolean indicating success (true) or failure (false), which is translated into the program’s exit code of 0 or 1.

Analyzing the data: Final result

- The output was written to the output directory, which contains one output file per reducer.
- This job had a single reducer, so there is a single file.

```
% cat output/part-r-00000
1949 111
1950 22
```

This can be interpreted as saying that the maximum temperature recorded in 1949 was 11.1°C, and in 1950 it was 2.2°C.

Combiner function

- Many MapReduce jobs are limited by the bandwidth available on the cluster, so it pays to minimize the data transferred between map and reduce tasks.
- The **combiner function** runs on the map outputs and its output forms the input to the reduce function.

Analyzing the data: Combiner

- Suppose that for the maximum temperature readings for the year 1950 were processed by two Mappers (because they were in different splits).
- Imagine the first map and second map produced the output:

```
(1950, 0)
(1950, 20)
(1950, 10)
```

```
(1950, 25)
(1950, 15)
```

- The reduce function received `(1950, [0, 20, 10, 25, 15])` and produced `1950 25`.

Analyzing the data: Combiner

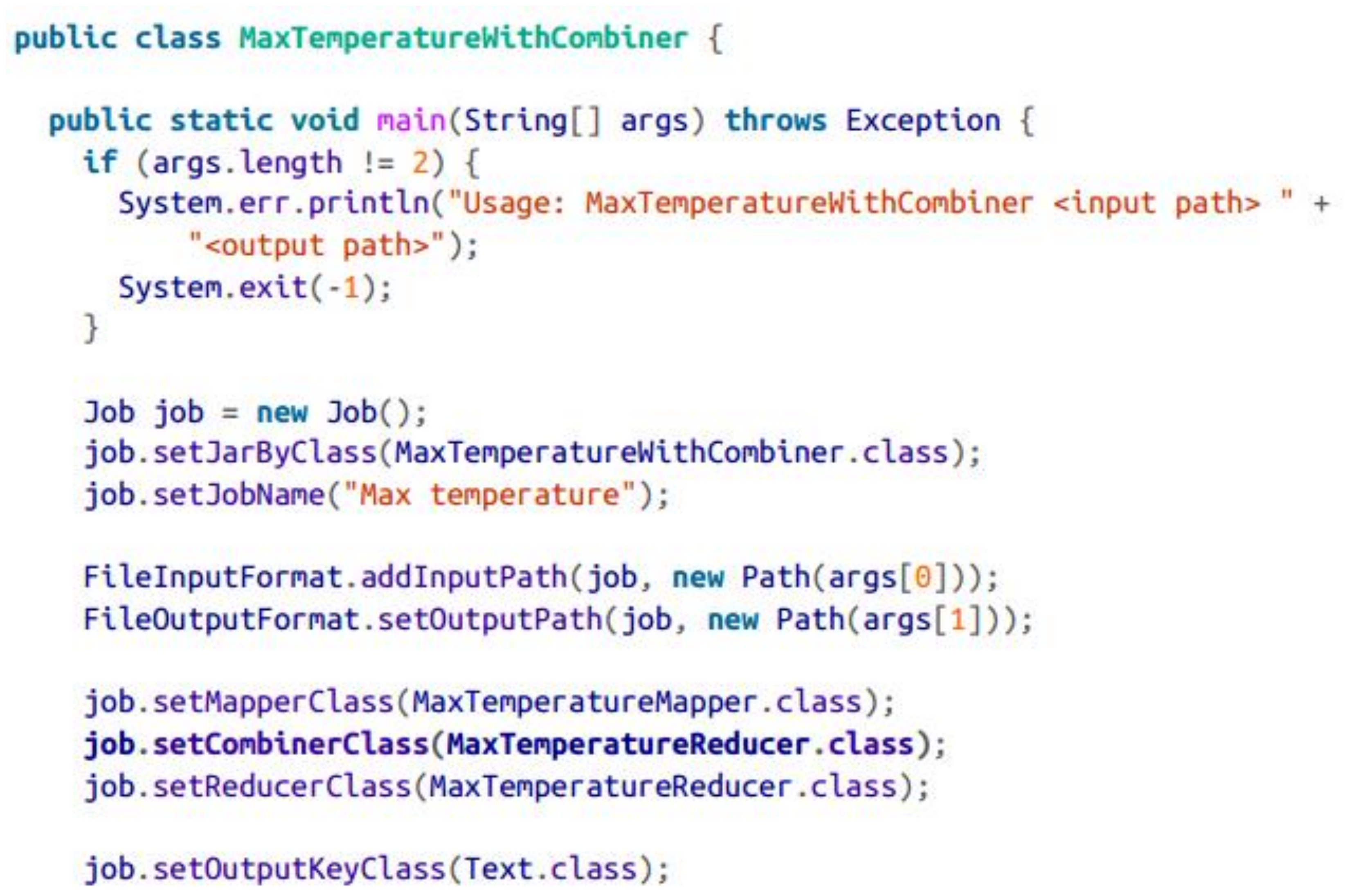
- A combiner function, just like the reduce function, finds the maximum temperature for each map output.
- The reduce function would then be called with `(1950, [20, 25])`.
- More succinctly, the function calls on the temperature values in this case as follows:

```
max(0, 20, 10, 25, 15)
= max(max(0, 20, 10), max(25, 15))
= max(20, 25) = 25
```

Combiner function

- Not all functions possess this property.
  - E.g., calculating mean temperatures, `mean(0, 20, 10, 25, 15) = 14` but `mean(mean(0, 20, 10), mean(25, 15)) = mean(10, 20) = 15`.
- The combiner function does not replace the reduce function.
  - The reduce function is still needed to process records with the same key from different Mappers.
- Yet, it helps cutting down the amount of data shuffled between the Mappers and the Reducers.

Combiner function

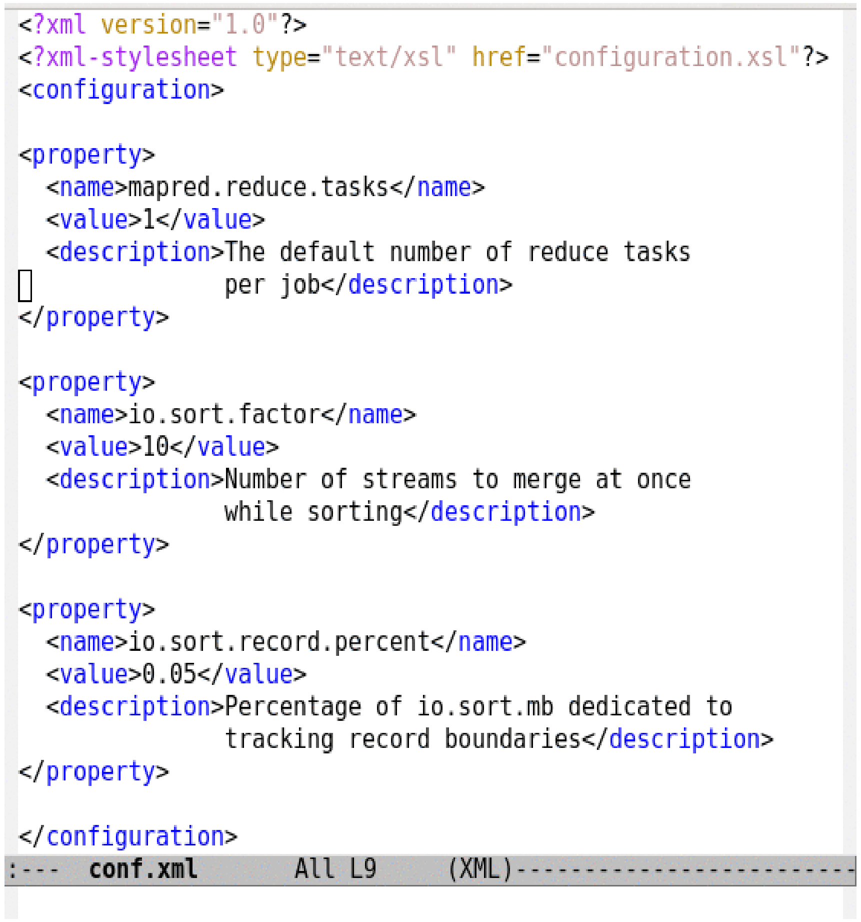


```java
public class MaxTemperatureWithCombiner {

  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.println("Usage: MaxTemperatureWithCombiner <input path> " +
          "<output path>");
      System.exit(-1);
    }

    Job job = new Job();
    job.setJarByClass(MaxTemperatureWithCombiner.class);
    job.setJobName("Max temperature");

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.setMapperClass(MaxTemperatureMapper.class);
    job.setCombinerClass(MaxTemperatureReducer.class);
    job.setReducerClass(MaxTemperatureReducer.class);

    job.setOutputKeyClass(Text.class);
```

Job configuration parameters

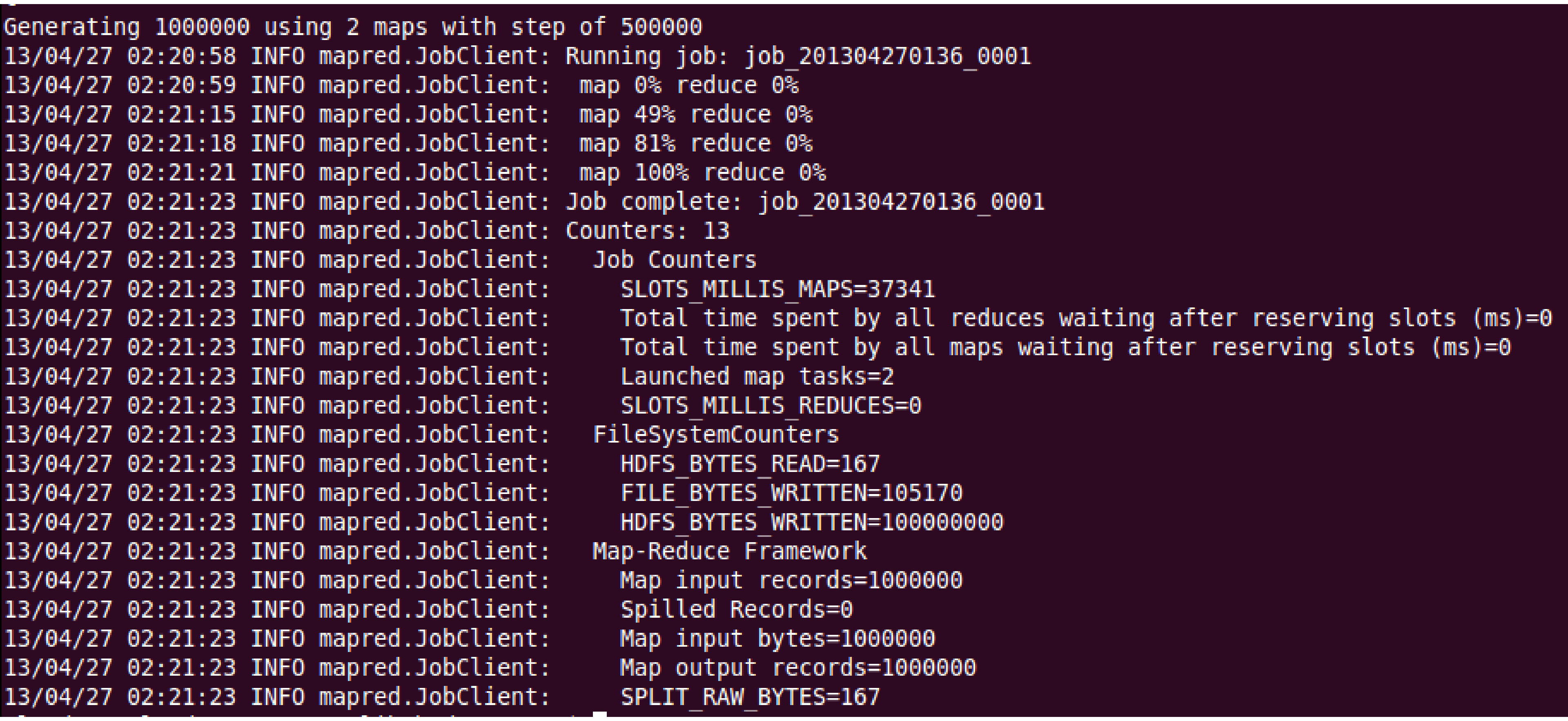
- 190+ parameters in Hadoop.
- Set manually or defaults are used.

- Do the settings impact performance?
- What are ways to set these parameters?
  - Defaults -- are they good enough?
  - Best practices -- the best setting can depend on data, job, and cluster properties
  - Automatic setting



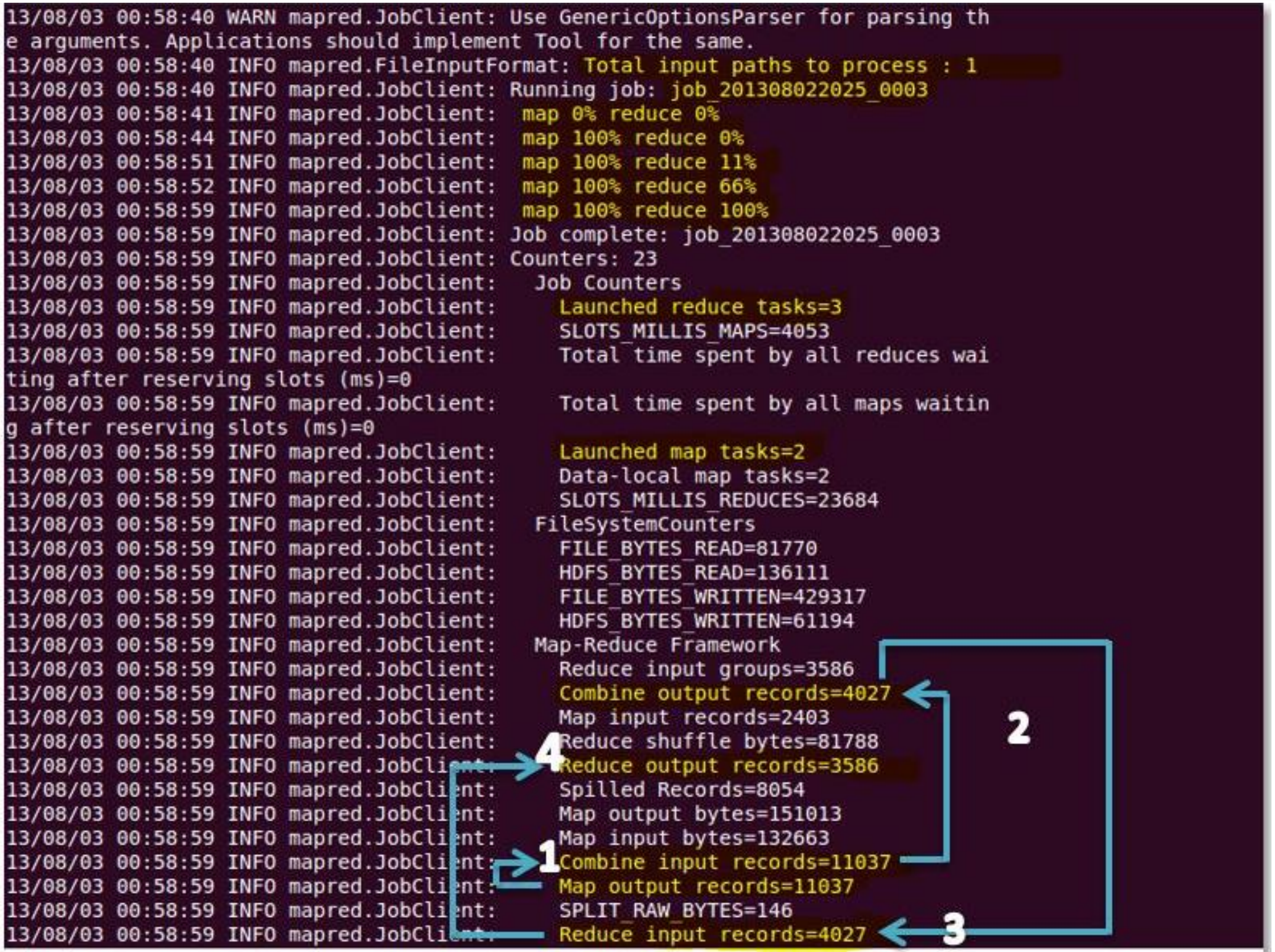
```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>

<property>
  <name>mapred.reduce.tasks</name>
  <value>1</value>
  <description>The default number of reduce tasks
               per job</description>
</property>

<property>
  <name>io.sort.factor</name>
  <value>10</value>
  <description>Number of streams to merge at once
               while sorting</description>
</property>

<property>
  <name>io.sort.record.percent</name>
  <value>0.05</value>
  <description>Percentage of io.sort.mb dedicated to
               tracking record boundaries</description>
</property>

</configuration>
:---  conf.xml       All L9     (XML)------------------------
```

Understanding the execution log



```
Generating 1000000 using 2 maps with step of 500000
13/04/27 02:20:58 INFO mapred.JobClient: Running job: job_201304270136_0001
13/04/27 02:20:59 INFO mapred.JobClient:  map 0% reduce 0%
13/04/27 02:21:15 INFO mapred.JobClient:  map 49% reduce 0%
13/04/27 02:21:18 INFO mapred.JobClient:  map 81% reduce 0%
13/04/27 02:21:21 INFO mapred.JobClient:  map 100% reduce 0%
13/04/27 02:21:23 INFO mapred.JobClient: Job complete: job_201304270136_0001
13/04/27 02:21:23 INFO mapred.JobClient: Counters: 13
13/04/27 02:21:23 INFO mapred.JobClient:   Job Counters
13/04/27 02:21:23 INFO mapred.JobClient:     SLOTS_MILLIS_MAPS=37341
13/04/27 02:21:23 INFO mapred.JobClient:     Total time spent by all reduces waiting after reserving slots (ms)=0
13/04/27 02:21:23 INFO mapred.JobClient:     Total time spent by all maps waiting after reserving slots (ms)=0
13/04/27 02:21:23 INFO mapred.JobClient:     Launched map tasks=2
13/04/27 02:21:23 INFO mapred.JobClient:     SLOTS_MILLIS_REDUCES=0
13/04/27 02:21:23 INFO mapred.JobClient:   FileSystemCounters
13/04/27 02:21:23 INFO mapred.JobClient:     HDFS_BYTES_READ=167
13/04/27 02:21:23 INFO mapred.JobClient:     FILE_BYTES_WRITTEN=105170
13/04/27 02:21:23 INFO mapred.JobClient:     HDFS_BYTES_WRITTEN=100000000
13/04/27 02:21:23 INFO mapred.JobClient:   Map-Reduce Framework
13/04/27 02:21:23 INFO mapred.JobClient:     Map input records=1000000
13/04/27 02:21:23 INFO mapred.JobClient:     Spilled Records=0
13/04/27 02:21:23 INFO mapred.JobClient:     Map input bytes=1000000
13/04/27 02:21:23 INFO mapred.JobClient:     Map output records=1000000
13/04/27 02:21:23 INFO mapred.JobClient:     SPLIT_RAW_BYTES=167
```

Understanding the execution log



```
13/08/03 00:58:40 WARN mapred.JobClient: Use GenericOptionsParser for parsing th
e arguments. Applications should implement Tool for the same.
13/08/03 00:58:40 INFO mapred.FileInputFormat: Total input paths to process : 1
13/08/03 00:58:40 INFO mapred.JobClient: Running job: job_201308022025_0003
13/08/03 00:58:41 INFO mapred.JobClient:  map 0% reduce 0%
13/08/03 00:58:44 INFO mapred.JobClient:  map 100% reduce 0%
13/08/03 00:58:51 INFO mapred.JobClient:  map 100% reduce 11%
13/08/03 00:58:52 INFO mapred.JobClient:  map 100% reduce 66%
13/08/03 00:58:59 INFO mapred.JobClient:  map 100% reduce 100%
13/08/03 00:58:59 INFO mapred.JobClient: Job complete: job_201308022025_0003
13/08/03 00:58:59 INFO mapred.JobClient: Counters: 23
13/08/03 00:58:59 INFO mapred.JobClient:   Job Counters
13/08/03 00:58:59 INFO mapred.JobClient:     Launched reduce tasks=3
13/08/03 00:58:59 INFO mapred.JobClient:     SLOTS_MILLIS_MAPS=4053
13/08/03 00:58:59 INFO mapred.JobClient:     Total time spent by all reduces wai
ting after reserving slots (ms)=0
13/08/03 00:58:59 INFO mapred.JobClient:     Total time spent by all maps waitin
g after reserving slots (ms)=0
13/08/03 00:58:59 INFO mapred.JobClient:     Launched map tasks=2
13/08/03 00:58:59 INFO mapred.JobClient:     Data-local map tasks=2
13/08/03 00:58:59 INFO mapred.JobClient:     SLOTS_MILLIS_REDUCES=23684
13/08/03 00:58:59 INFO mapred.JobClient:   FileSystemCounters
13/08/03 00:58:59 INFO mapred.JobClient:     FILE_BYTES_READ=81770
13/08/03 00:58:59 INFO mapred.JobClient:     HDFS_BYTES_READ=136111
13/08/03 00:58:59 INFO mapred.JobClient:     FILE_BYTES_WRITTEN=429317
13/08/03 00:58:59 INFO mapred.JobClient:     HDFS_BYTES_WRITTEN=61194
13/08/03 00:58:59 INFO mapred.JobClient:   Map-Reduce Framework
13/08/03 00:58:59 INFO mapred.JobClient:     Reduce input groups=3586
13/08/03 00:58:59 INFO mapred.JobClient:     Combine output records=4027
13/08/03 00:58:59 INFO mapred.JobClient:     Map input records=2403
13/08/03 00:58:59 INFO mapred.JobClient:     Reduce shuffle bytes=81788
13/08/03 00:58:59 INFO mapred.JobClient:     Reduce output records=3586
13/08/03 00:58:59 INFO mapred.JobClient:     Spilled Records=8054
13/08/03 00:58:59 INFO mapred.JobClient:     Map output bytes=151013
13/08/03 00:58:59 INFO mapred.JobClient:     Map input bytes=132663
13/08/03 00:58:59 INFO mapred.JobClient:     Combine input records=11037
13/08/03 00:58:59 INFO mapred.JobClient:     Map output records=11037
13/08/03 00:58:59 INFO mapred.JobClient:     SPLIT_RAW_BYTES=146
13/08/03 00:58:59 INFO mapred.JobClient:     Reduce input records=4027
```

Exercise

- Demonstrate the MapReduce data flow in determining how many times different words appear in a set of files.
- For example, given the files:

  foo.txt: Sweet, this is the foo file

  bar.txt: This is the bar file

- The output is expected to be:

```
sweet 1
this  2
is    2
the   2
foo   1
bar   1
file  2
```

Exercise: Answer keys

- Several Mapper instances are created on the different machines, each of which receives a different input file.
- The Mappers output (word, 1) pairs which are then forwarded to the Reducers.
- Several Reducer instances are also instantiated on the different machines, each of which is responsible for processing the list of values associated with a different word.
    - The list of values will be a list of 1's.
    - The reducer sums up those ones into a final count associated with a single word, and then emits the final (word, count) output.

Word Count: Baseline

```
class Mapper
    method Map(docid $a$, doc $d$)
        for all term $t \in$ doc $d$ do
            Emit(term $t$, count 1)

class Reducer
    method Reduce(term $t$, counts $[c_1, c_2, \ldots]$)
        $sum \leftarrow 0$
        for all count $c \in$ counts $[c_1, c_2, \ldots]$ do
            $sum \leftarrow sum + c$
        Emit(term $t$, count $s$)
```

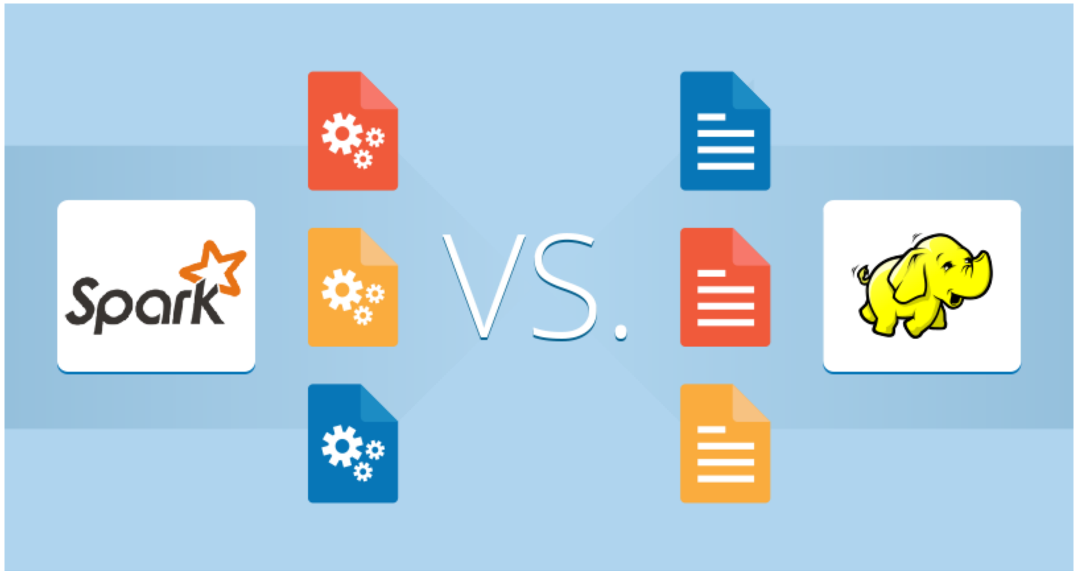
Word Count: Version 1

```
class Mapper
    method Map(docid $a$, doc $d$)
        $H \leftarrow$ new AssociativeArray
        for all term $t \in$ doc $d$ do
            $H\{t\} \leftarrow H\{t\} + 1$                    ▷ Tally counts for entire document
        for all term $t \in H$ do
            Emit(term $t$, count $H\{t\}$)
```

Word Count: Version 2

```
class Mapper
    method Initialize
        $H \leftarrow$ new AssociativeArray
    method Map(docid $a$, doc $d$)
        for all term $t \in$ doc $d$ do
            $H\{t\} \leftarrow H\{t\} + 1$                    ▷ Tally counts *across* documents
    method Close
        for all term $t \in H$ do
            Emit(term $t$, count $H\{t\}$)
```

Key: preserve state across input key-value pairs!



Alternatives to MapReduce

Better ways to handle different types of Big Data

Newer technologies to MapReduce

Batch processing

hadoop MapReduce

TEZ

APACHE Spark™

Flink

APACHE STORM™

samza

Stream processing

THE END