Introduction to Big Data

HADOOP MAPREDUCE

Le Ngoc Thanh – Nguyen Ngoc Thao
{lnthanh, nnthao}@fit.hcmus.edu.vn

Outline

- Hadoop MapReduce
    - A general MapReduce job run
- How MapReduce works in phases
- Handling failures in MapReduce
- A simple example of MapReduce

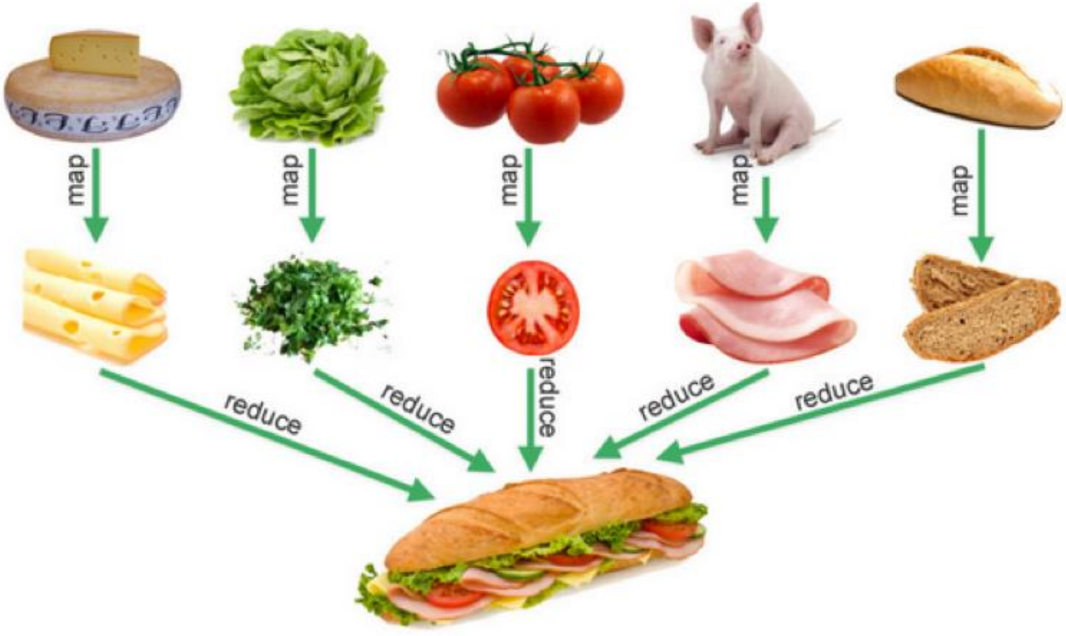
hadoop

Map Reduce

What is MapReduce (MR)?

- MapReduce is a programming model that enables distributed computations on massive data in batch mode.
- The workload is divided into multiple independent tasks, each performs the work separately from one another.

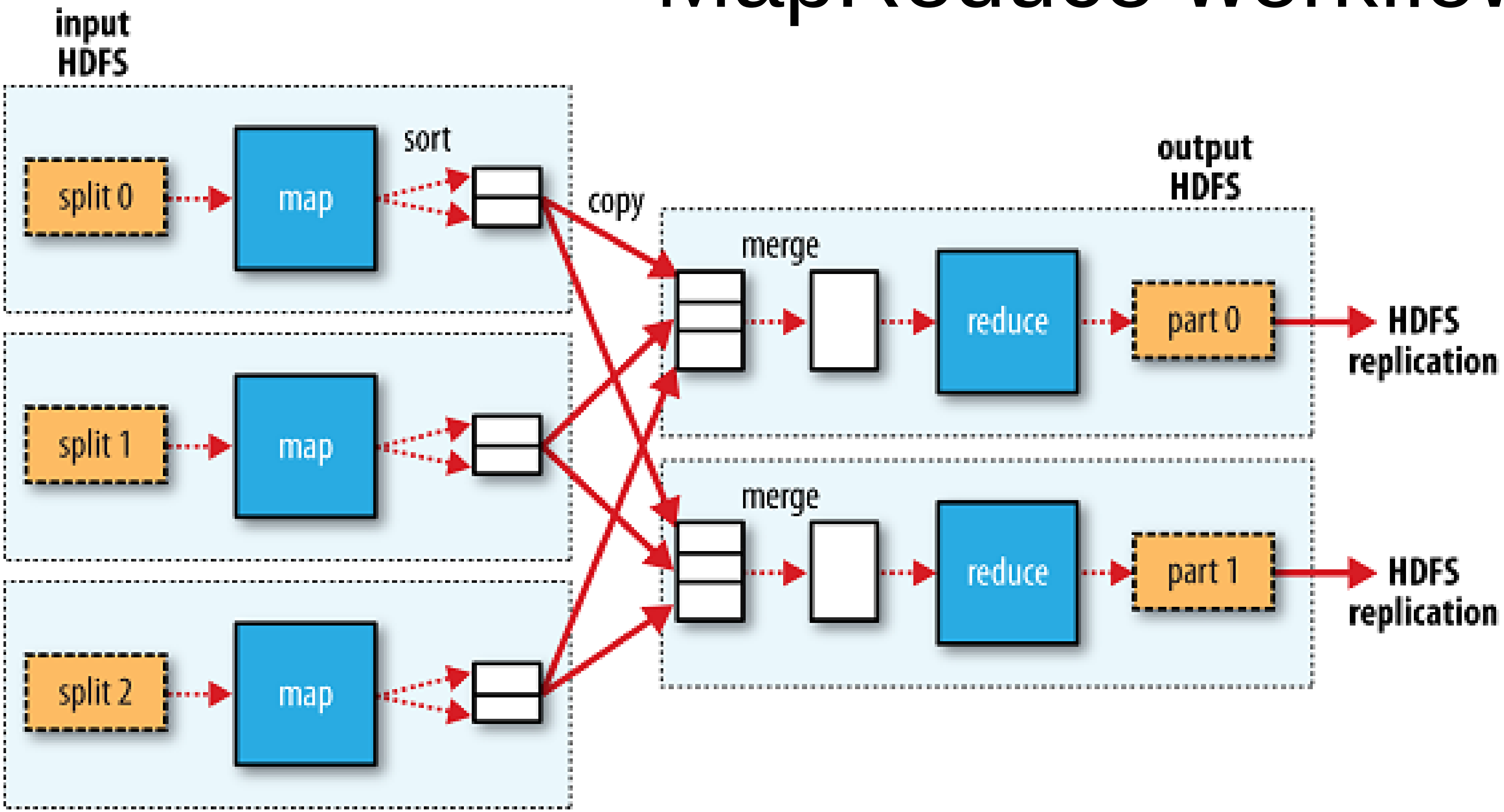
Making a "banh mi" is akin to an MR job, sourcing ingredients independently and assembling them in the final step.



Independent tasks: Key enabler

- Data is distributed to cluster nodes (e.g., HDFS) and MR schedules the tasks across these nodes.
- Map/Reduce tasks are single-threaded and deterministic in separate JVMs, allowing for restarting of failed jobs.
  - Users are free from handling multi-threaded code.
- Communication among tasks is limited for scalability.
  - The synchronization overheads would prevent the model from performing reliably and efficiently at large scale.

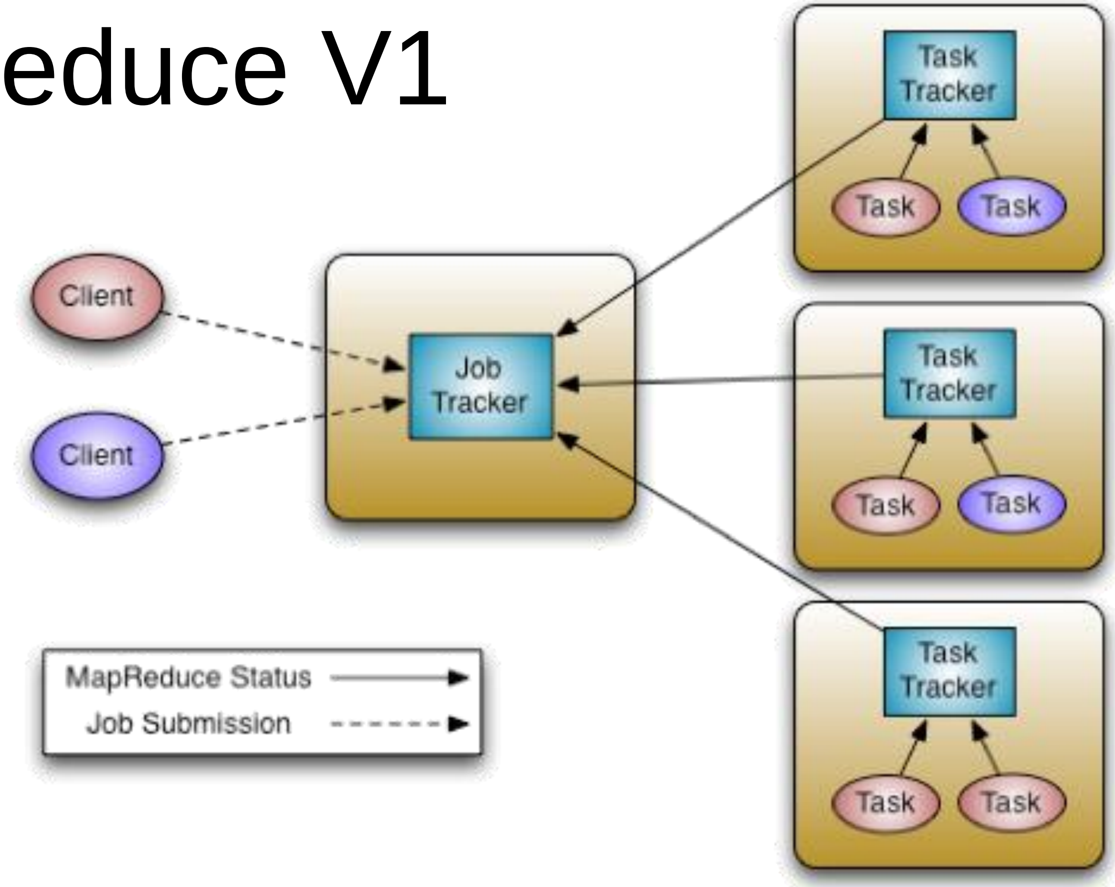
MapReduce workflow



- Several Map tasks can operate on a single huge data file in parallel → performance improved significantly.

MapReduce job run

Hadoop MapReduce V1



Task

- A separate process
- Run the MR functions

TaskTracker (Worker)

- Run MR tasks
- Manage intermediate output

JobTracker (Master)

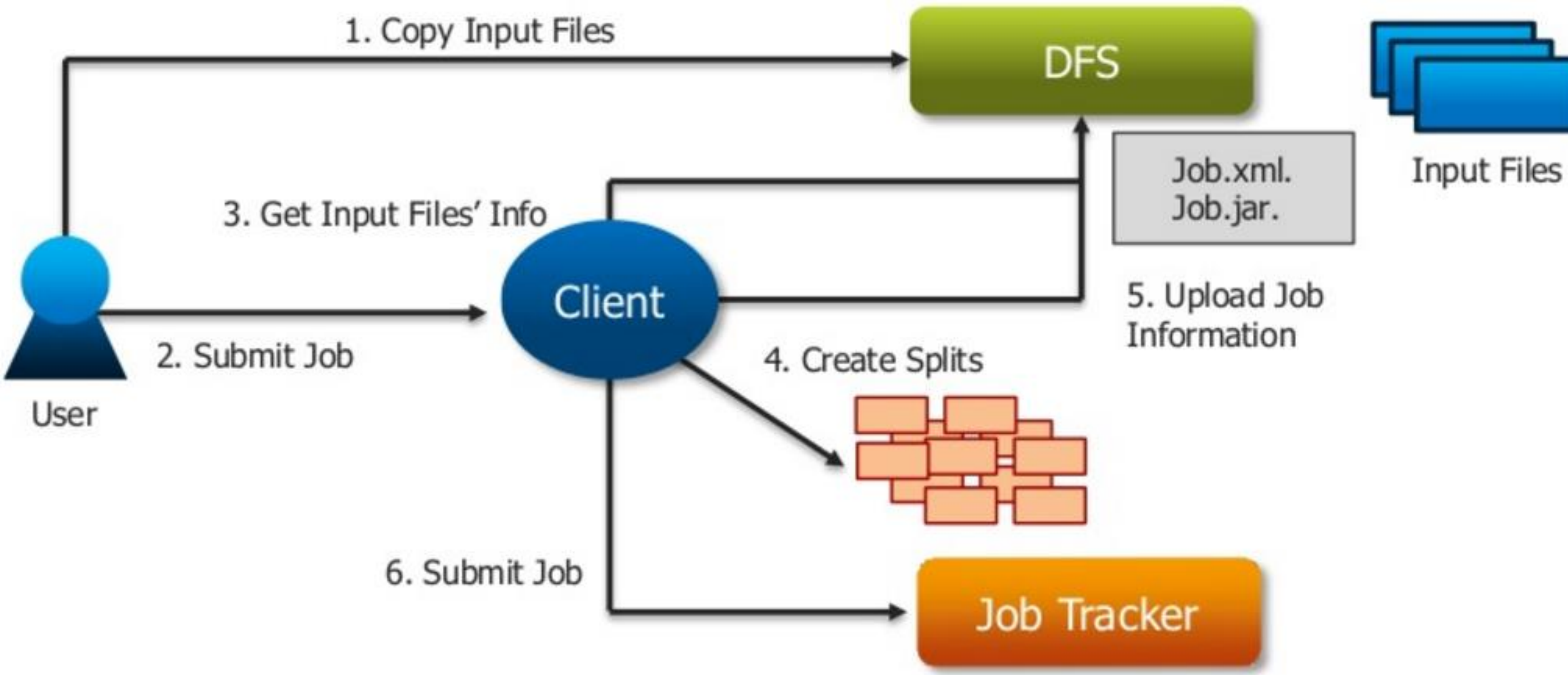
- Accept MR jobs
- Assign tasks to workers
- Handle tasks and failures

Client

- UI for submitting jobs
- Get various status information

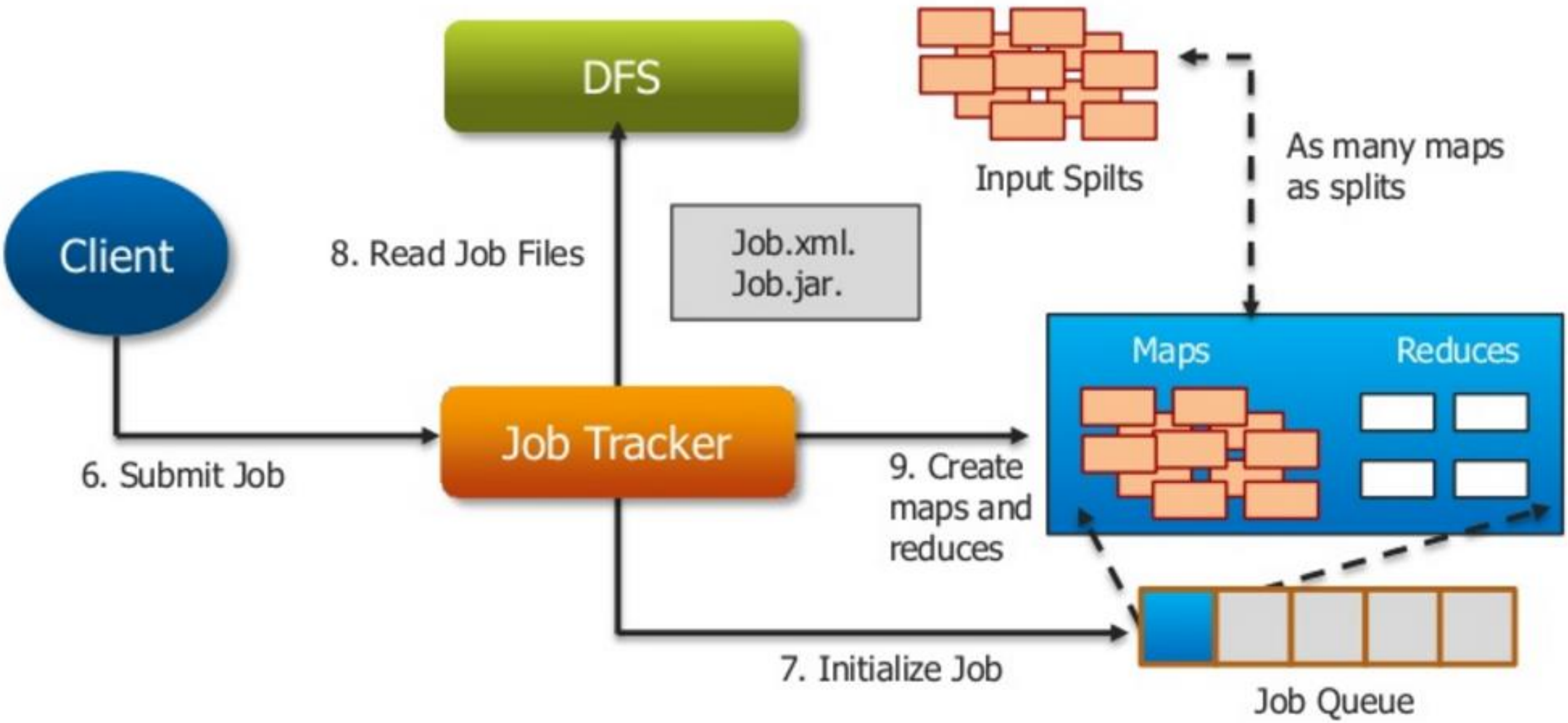
MapReduce V1 with JobTracker

- Job submission



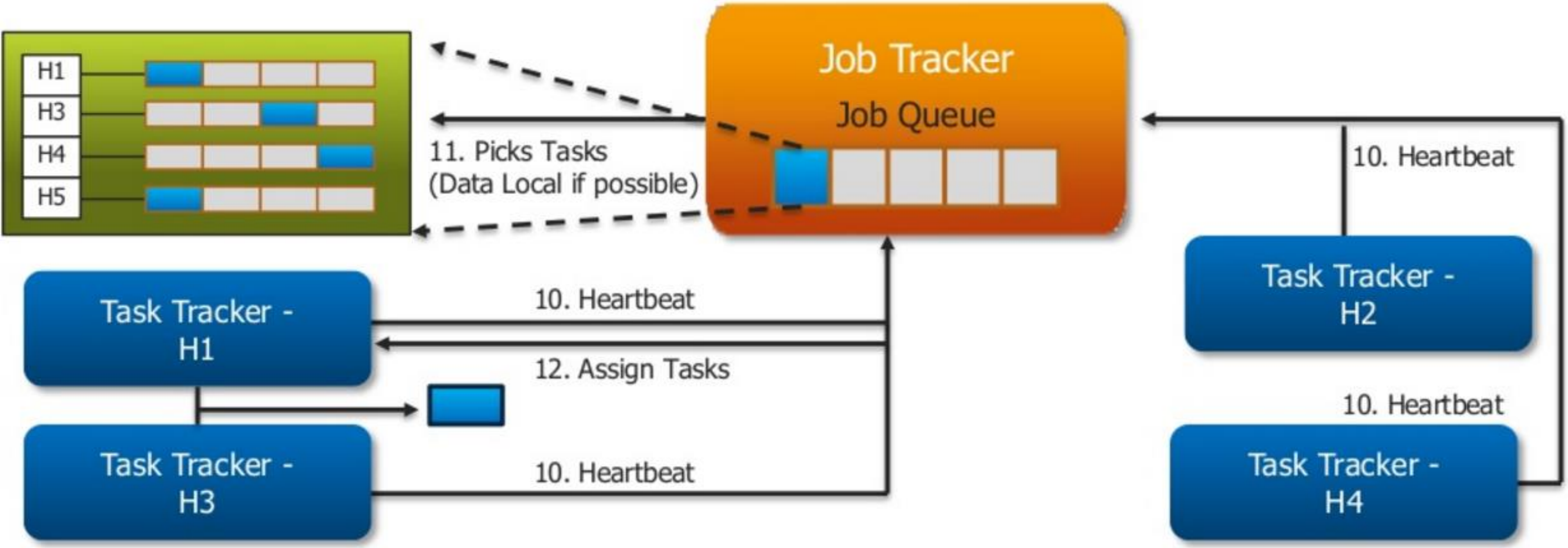
MapReduce V1 with JobTracker

- Job initialization



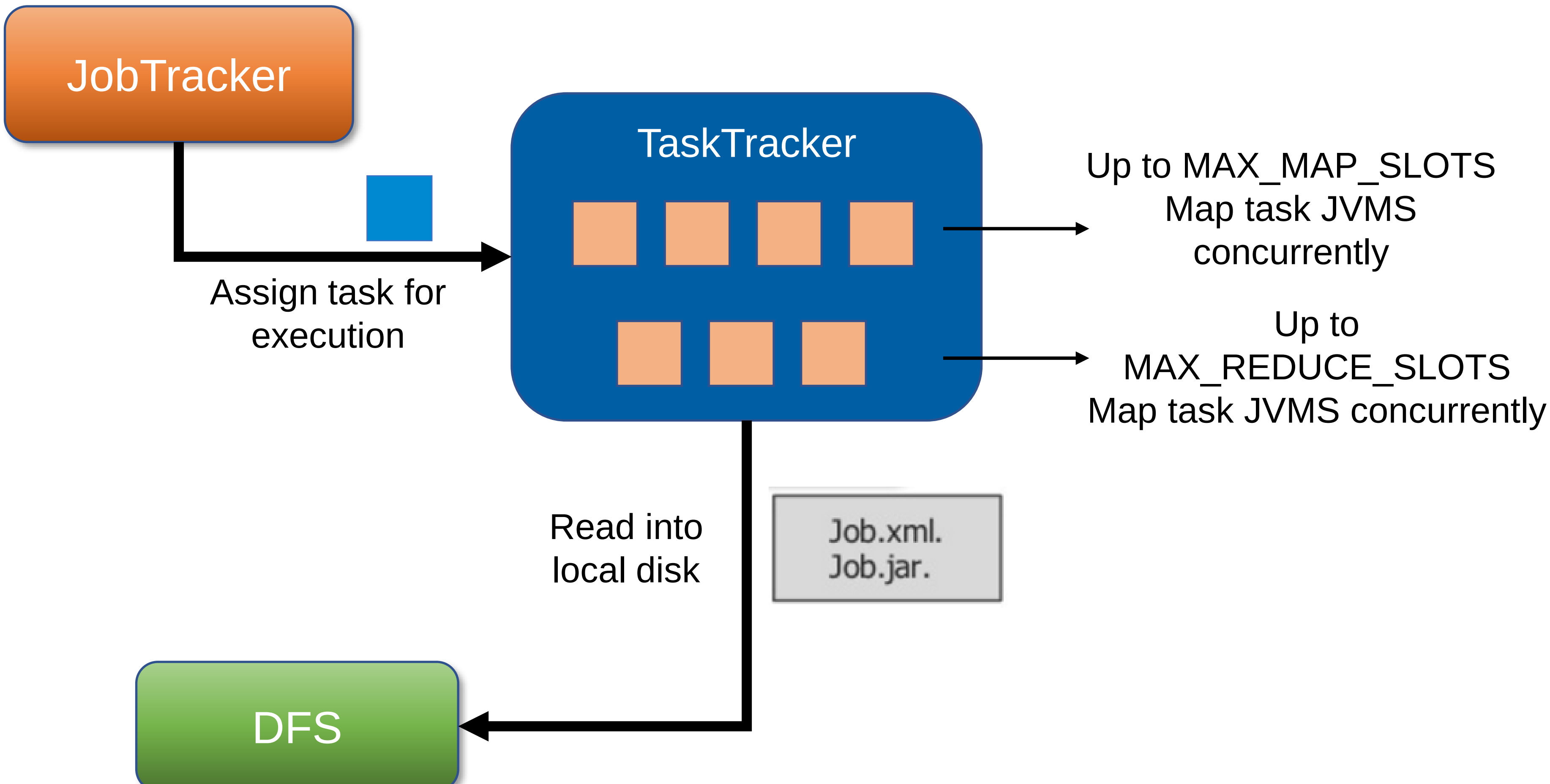
MapReduce V1 with JobTracker

- Job scheduling

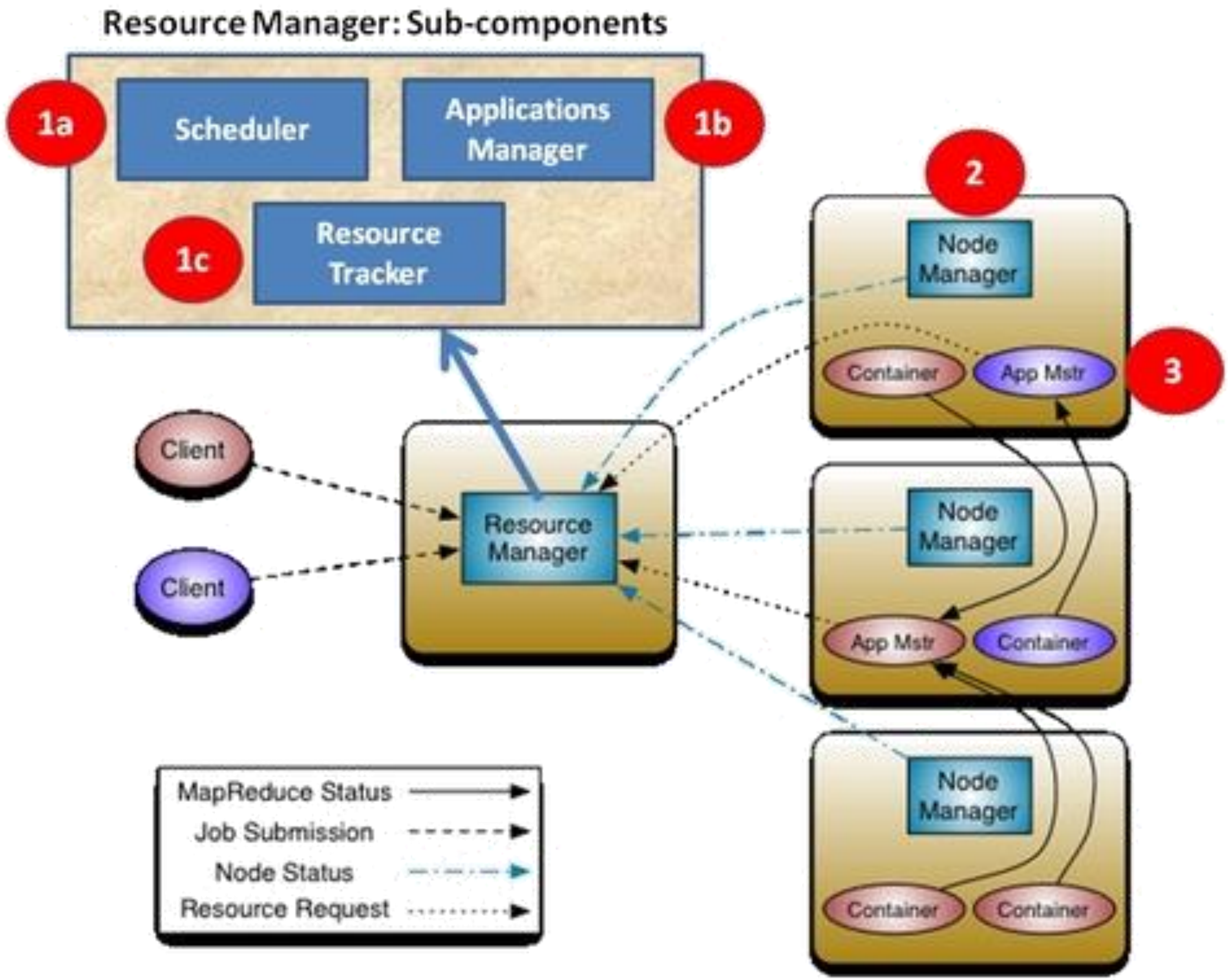


MapReduce V1 with JobTracker

- Job execution



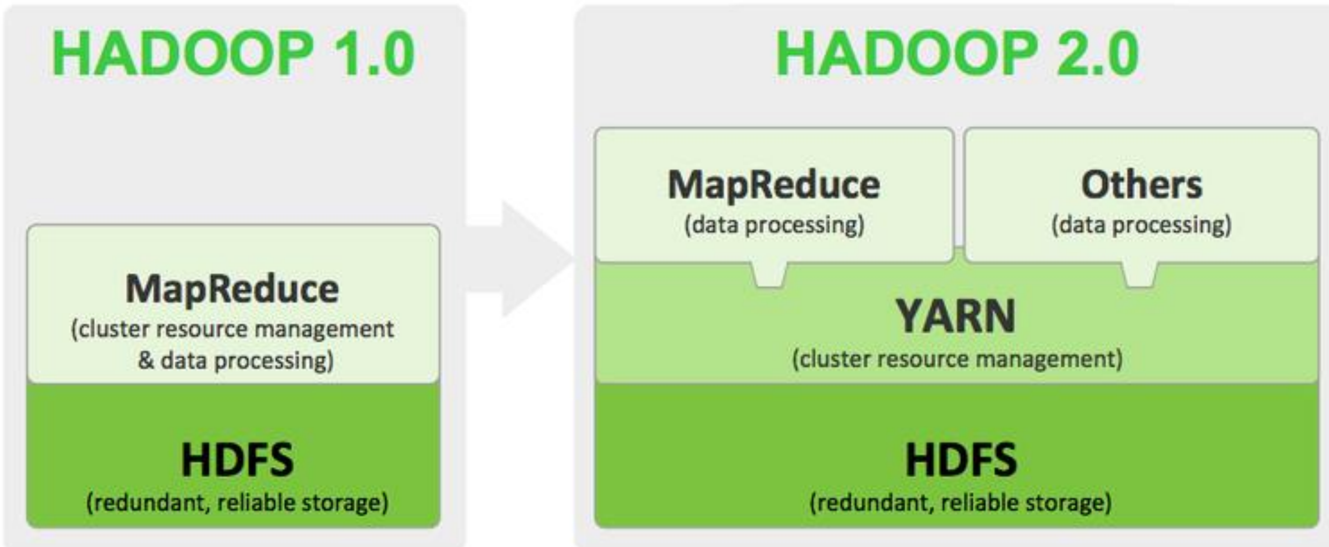
Hadoop MapReduce V2

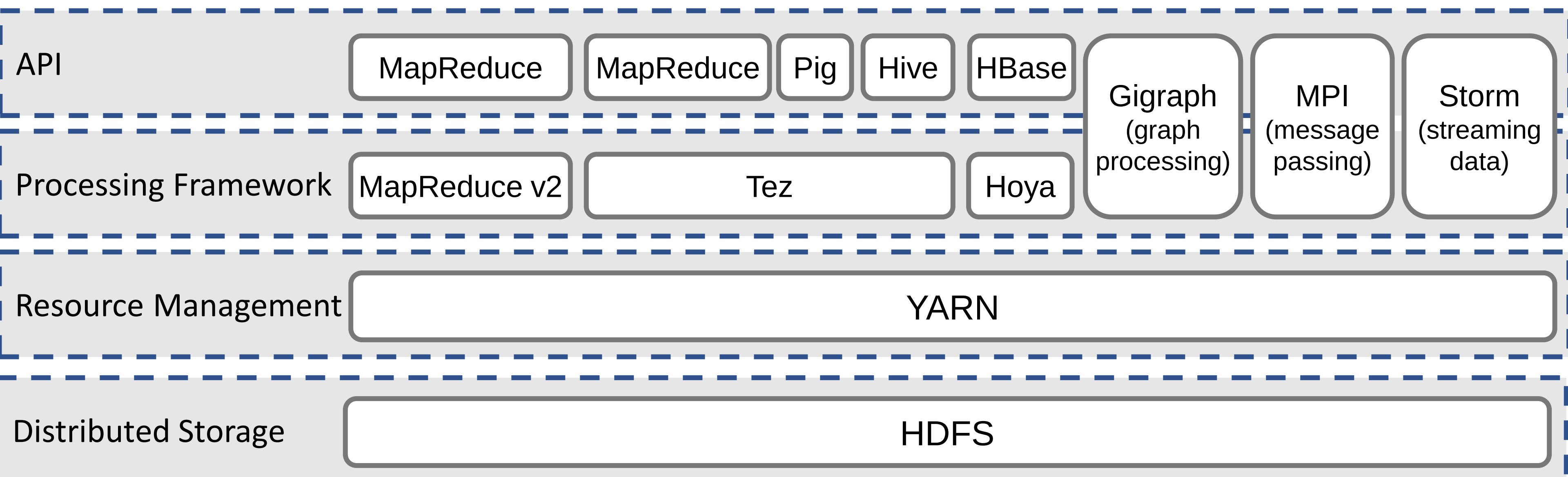


Yet Another Resource Negotiator (YARN)

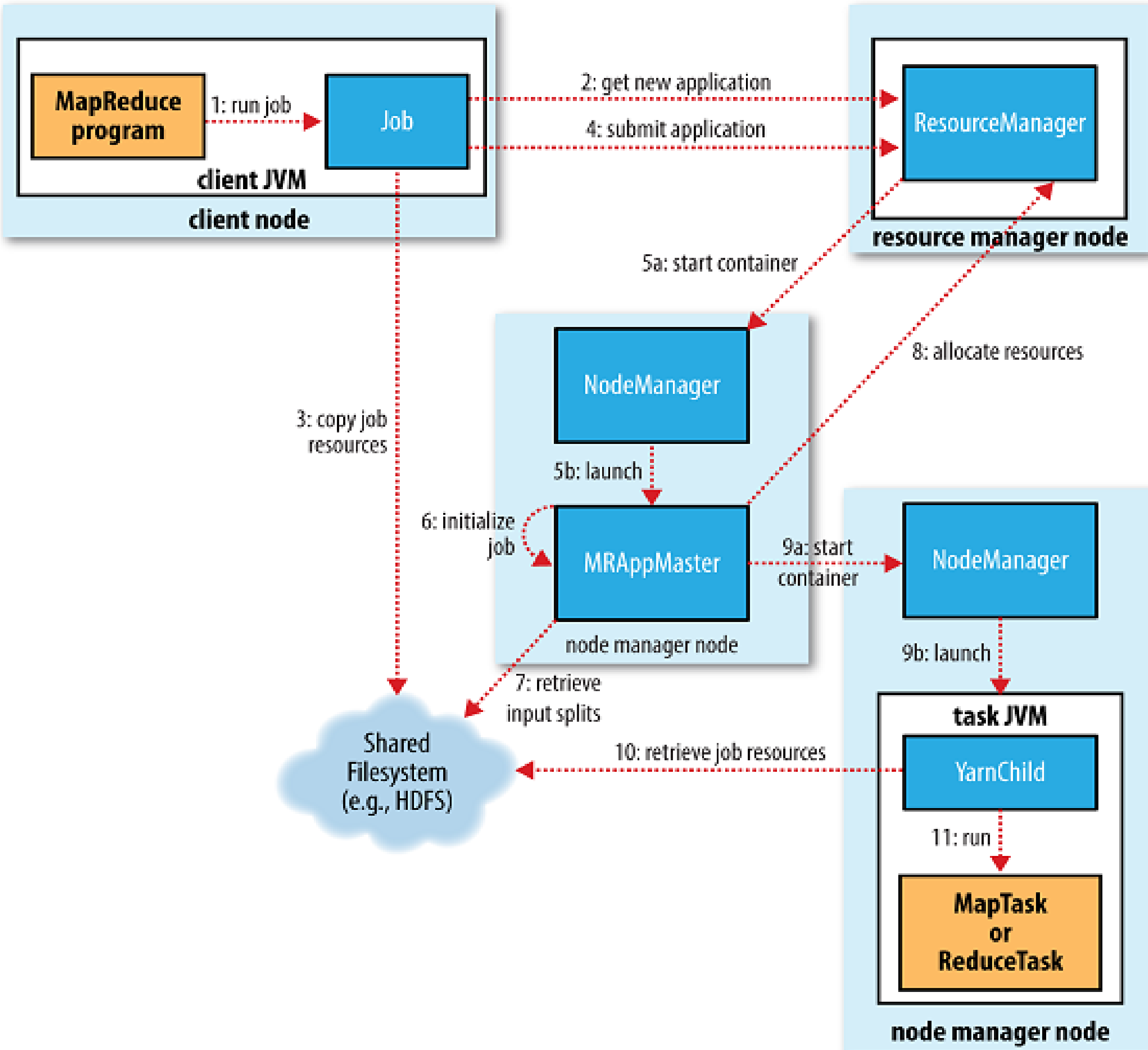
- YARN is a cluster resource manager and scheduler that is external to any framework.
- It offers generic scheduling and resource management.
  - Hadoop now can support more than just MR and batch processing.
- Thus, more efficient scheduling and workload management.
  - No more balancing between Map slots and Reduce slots

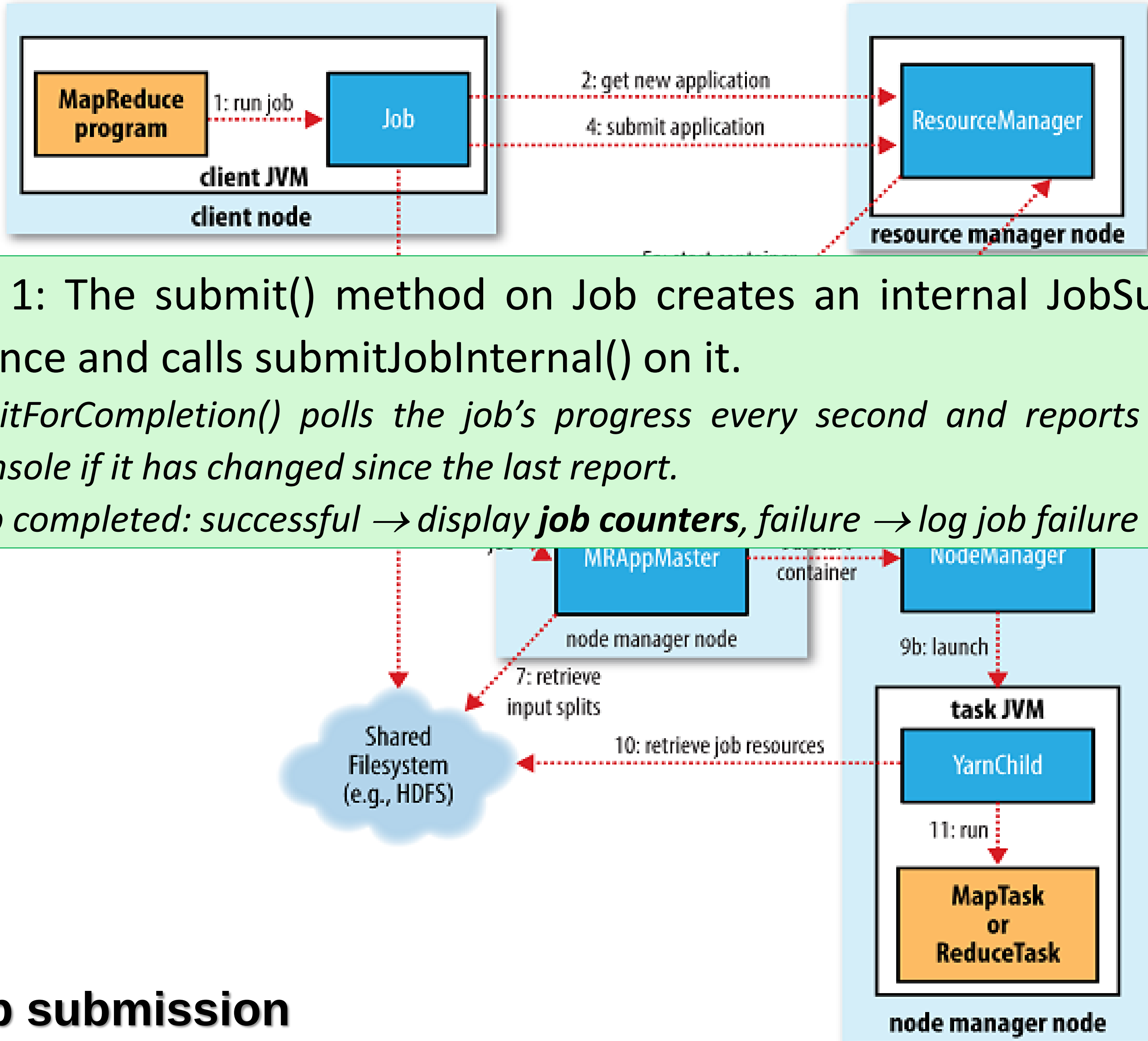
Yet Another Resource Negotiator (YARN)





MapReduce V2 with YARN

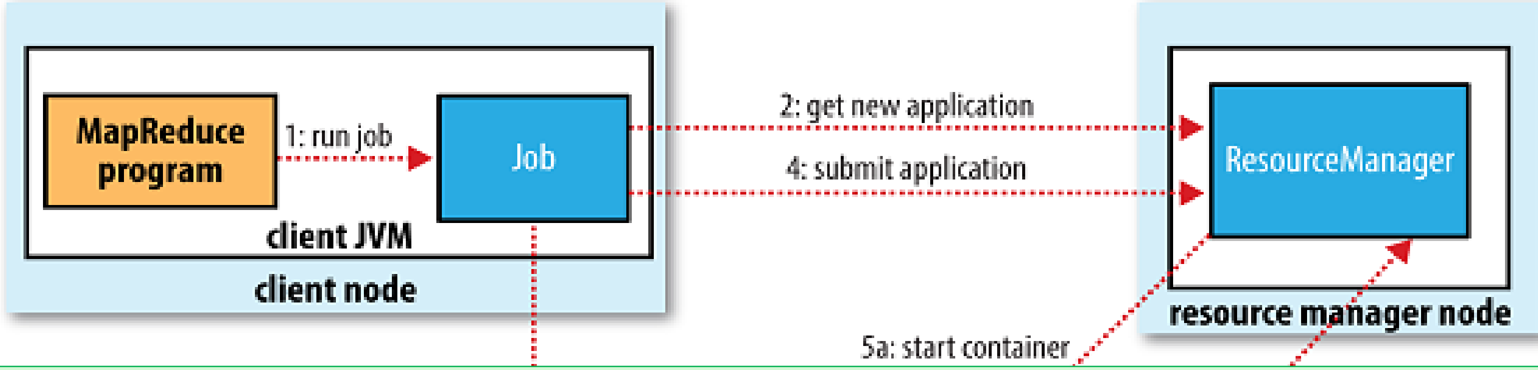




Step 1: The submit() method on Job creates an internal JobSubmitter instance and calls submitJobInternal() on it.

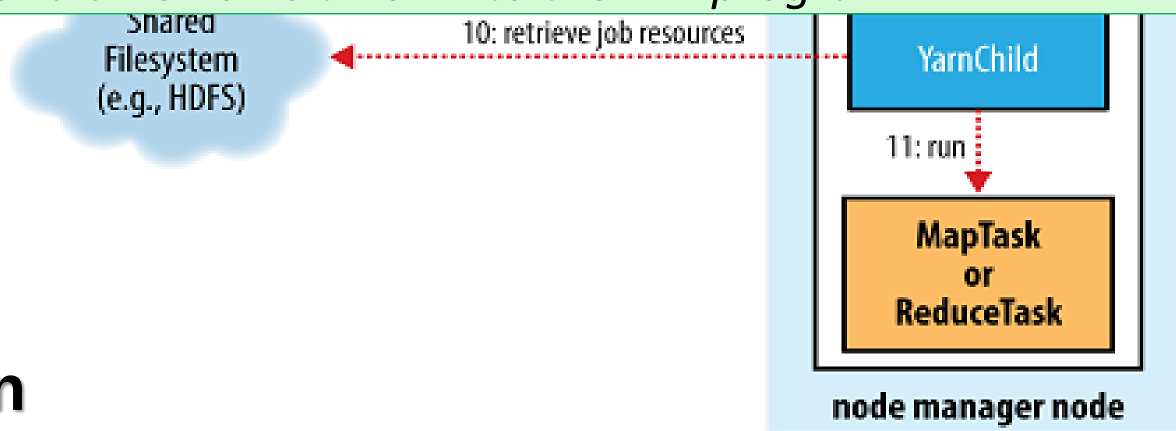
- *waitForCompletion() polls the job's progress every second and reports it to the console if it has changed since the last report.*
- *Job completed: successful → display **job counters**, failure → log job failure error*

Job submission

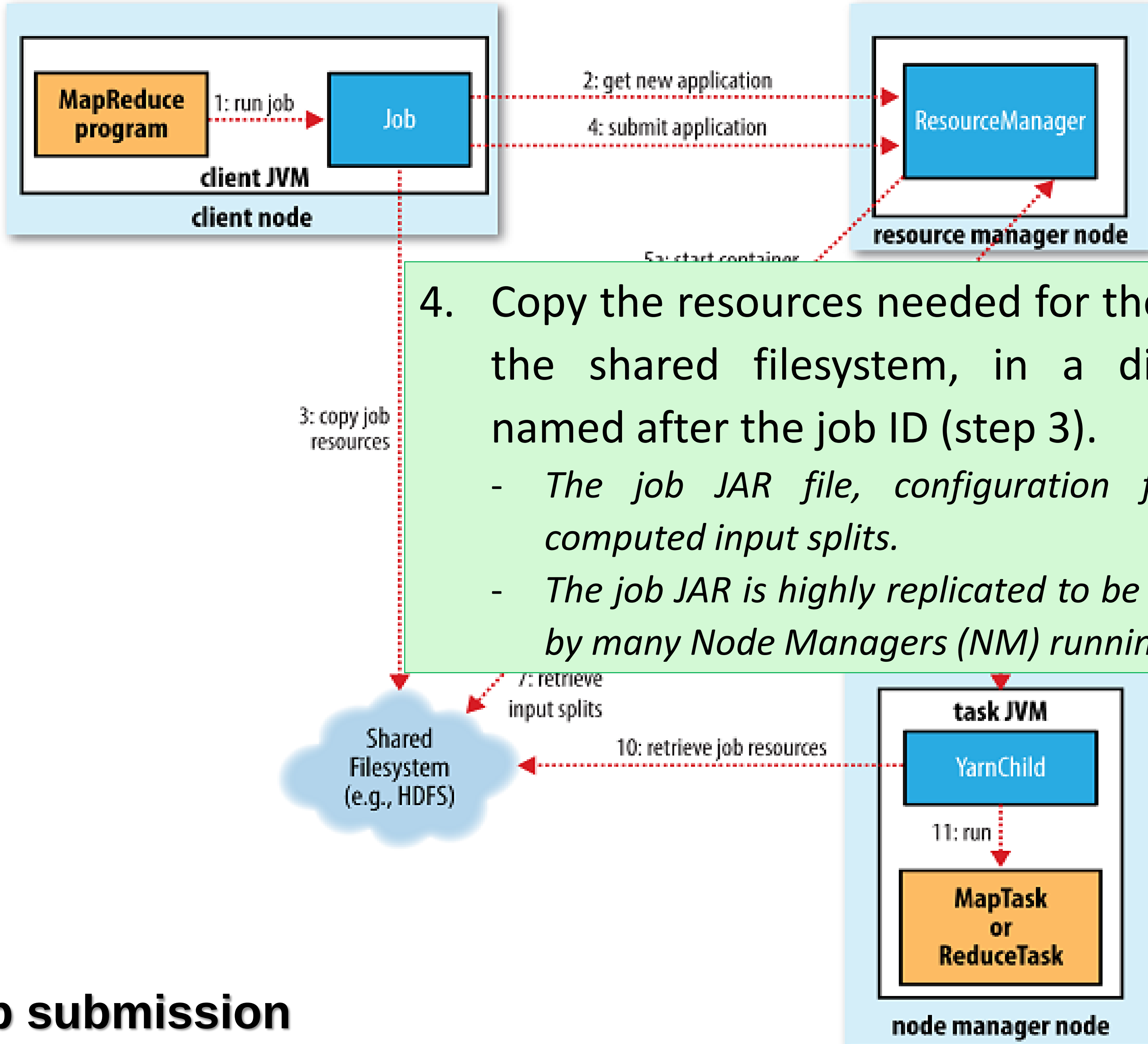


The job submission implemented by JobSubmitter does the following:

1. Ask Resource Manager (RM) for a new MR 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 MR program.*

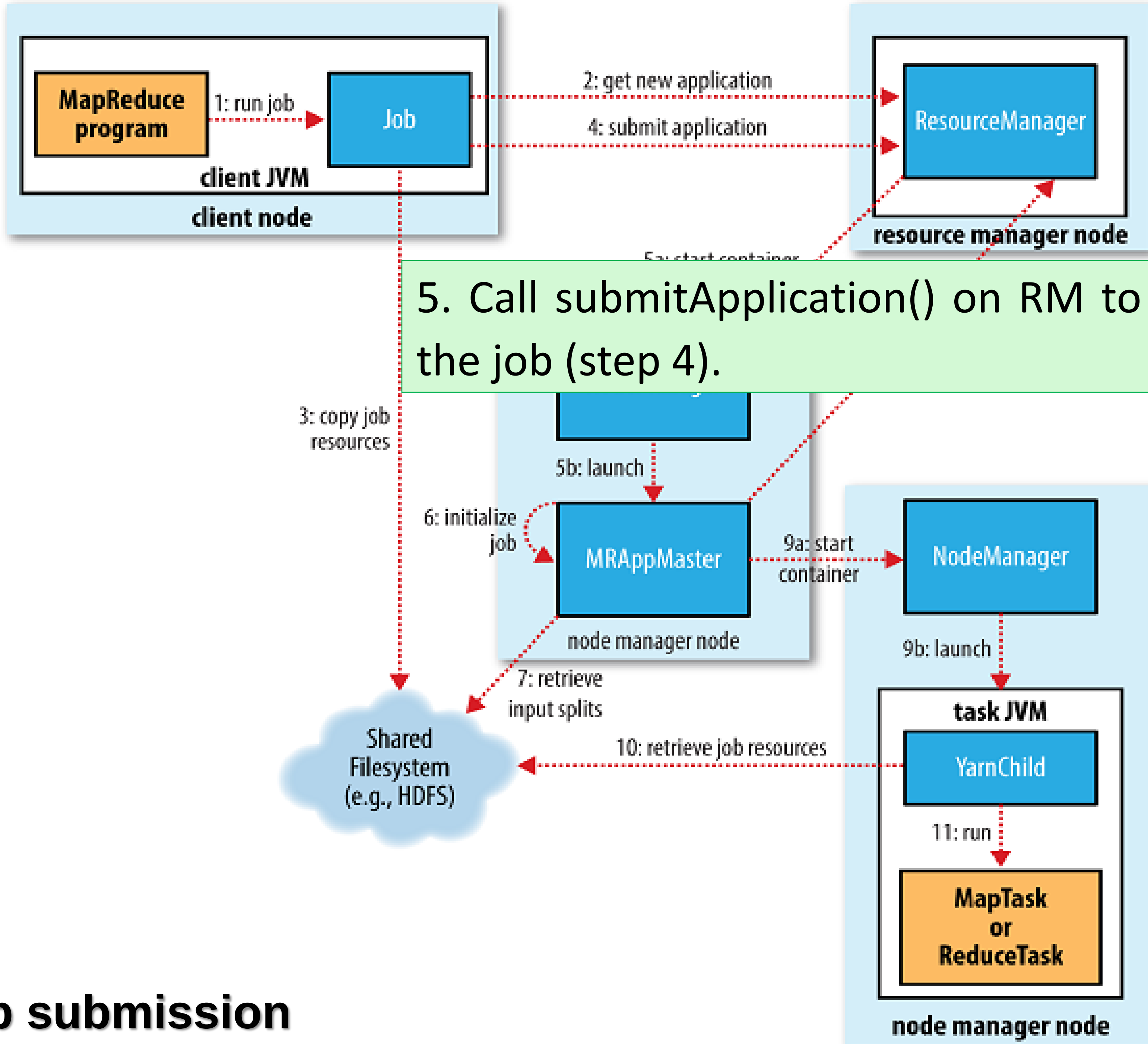


Job submission



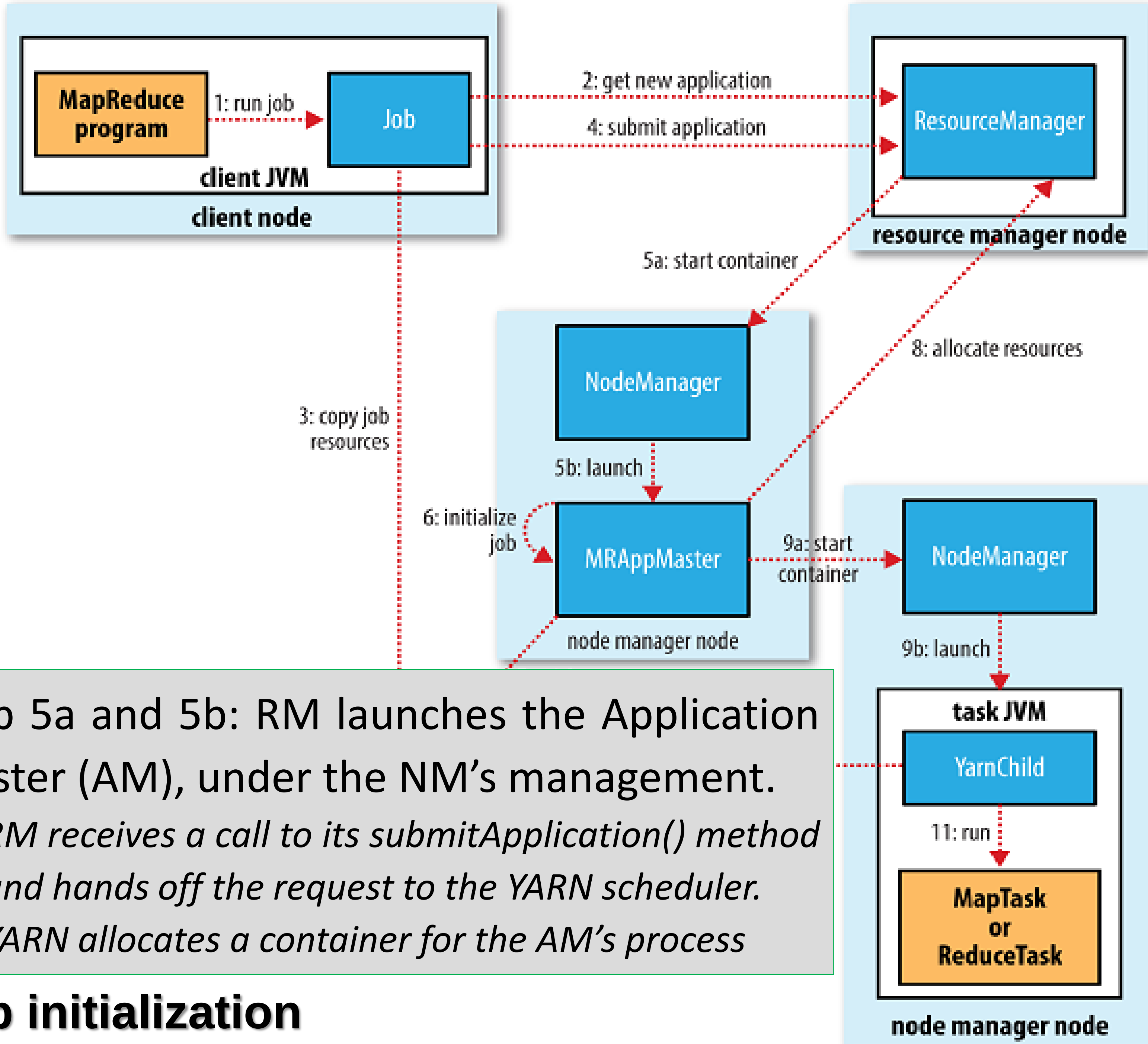
4. Copy the resources needed for the job to the shared filesystem, in a directory named after the job ID (step 3).
   - *The job JAR file, configuration file, and computed input splits.*
   - *The job JAR is highly replicated to be accessed by many Node Managers (NM) running tasks.*

Job submission



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

Job submission

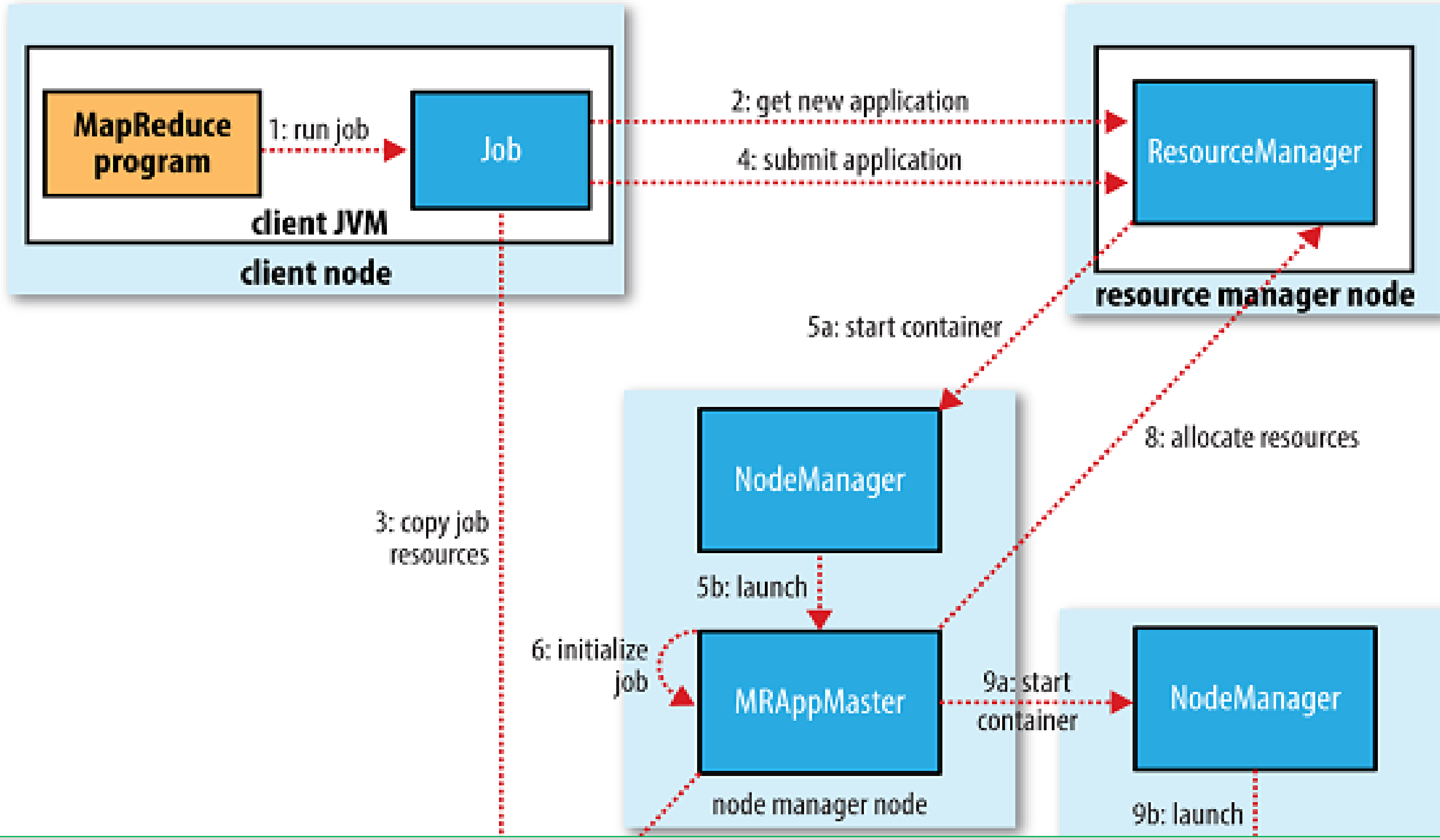


Step 5a and 5b: RM launches the Application Master (AM), under the NM's management.

- *RM receives a call to its submitApplication() method and hands off the request to the YARN scheduler.*
- *YARN allocates a container for the AM's process*

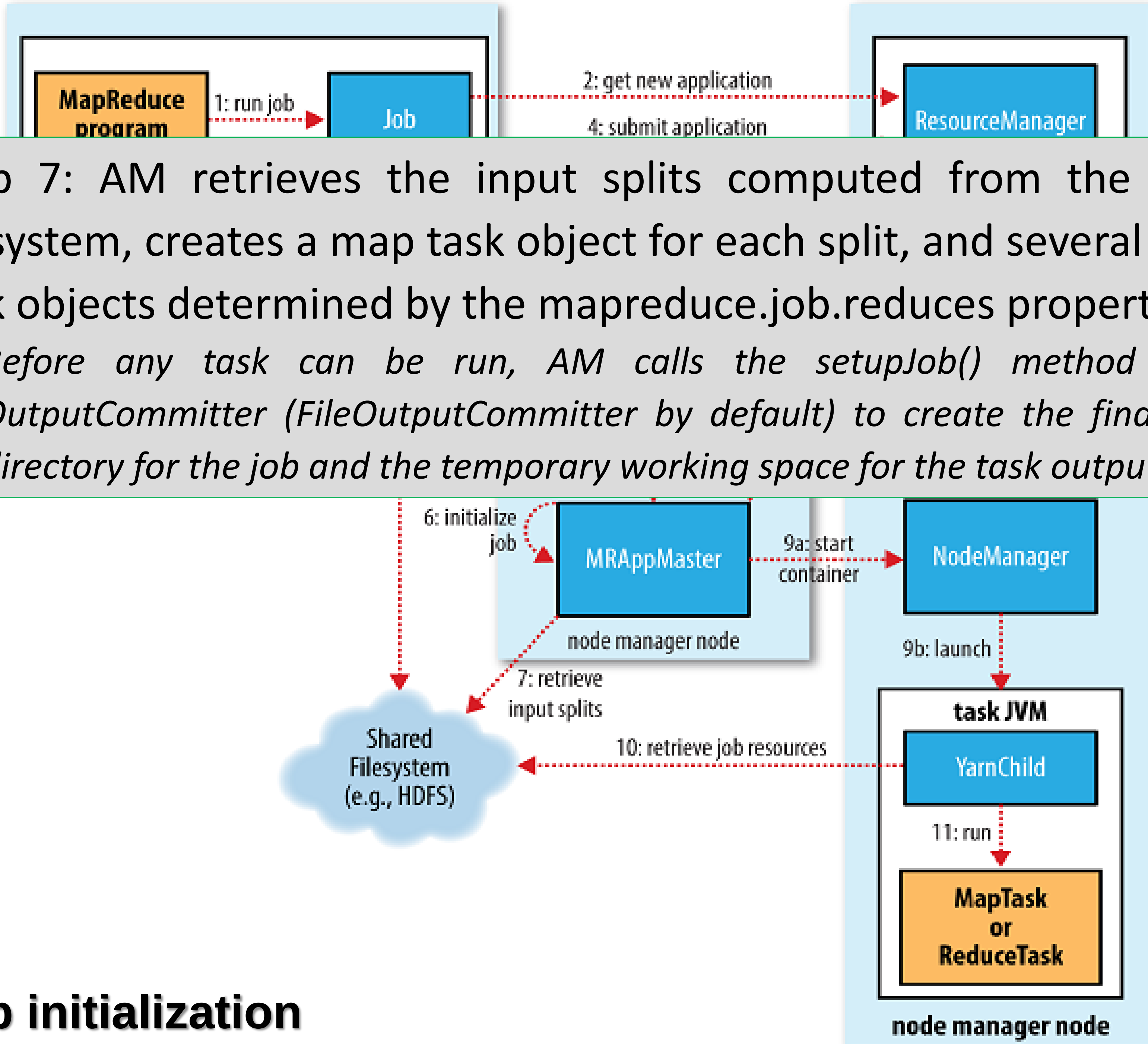
Job initialization

Job initialization



Step 6: The AM (of a job) initializes the job by creating several bookkeeping objects to *track the job's progress.*

- *AM is a Java application whose main class is MRAppMaster.*
- *Each bookkeeping objects receives progress and completion reports from a task.*



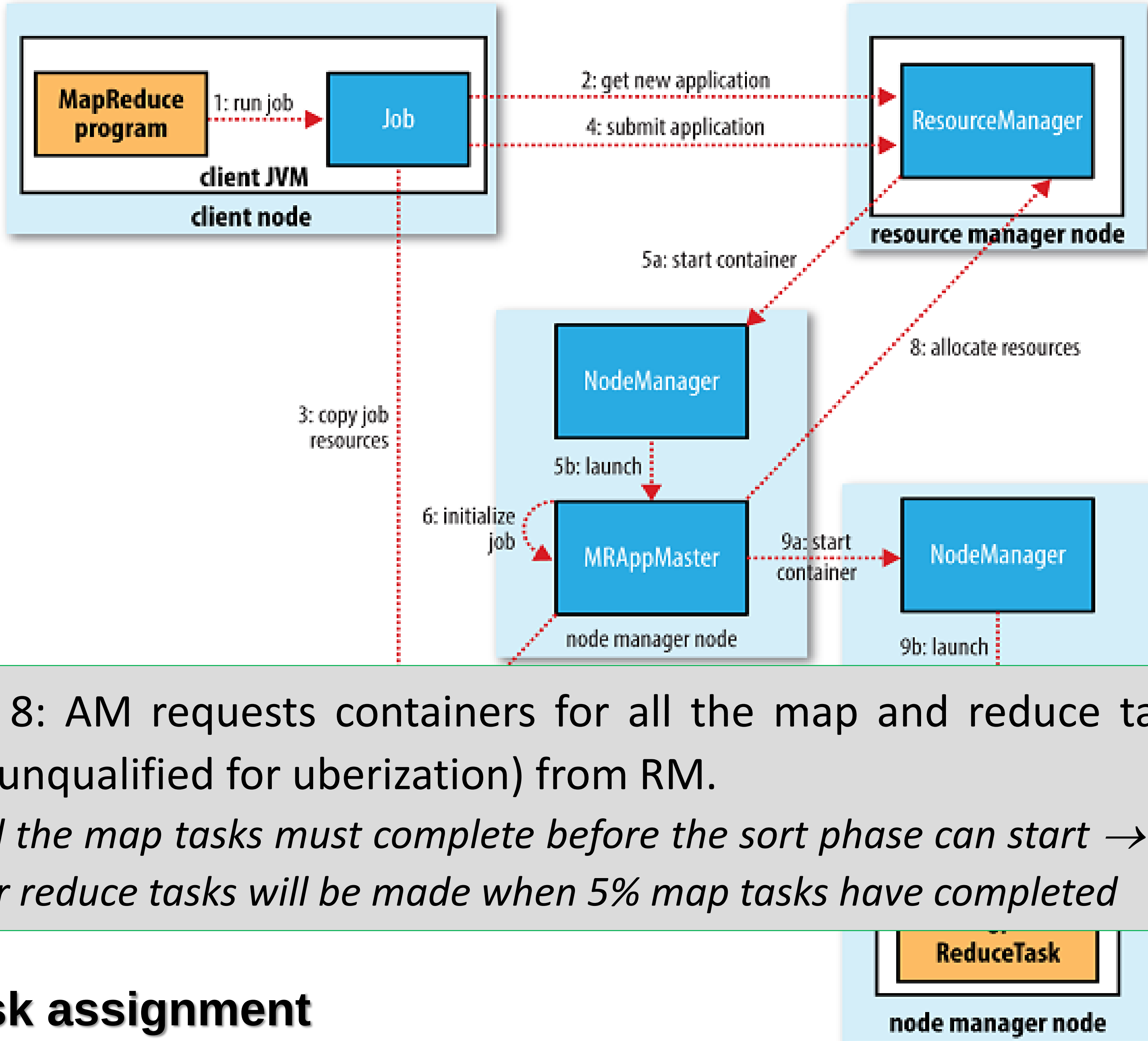
Step 7: AM retrieves the input splits computed from the shared filesystem, creates a map task object for each split, and several reduce task objects determined by the mapreduce.job.reduces property.

- *Before any task can be run, AM 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

Uber task

- AM decides how to run the tasks that make up the MR job.
- For a small job, AM may choose to run the tasks in the same JVM as itself.
  - Running tasks in parallel requires the allocation of new containers and the management of tasks' operations.
  - These overheads may outweigh the gain compared to running them sequentially on one node.
- Such a job is said to be uberized or run as an uber task.



Step 8: AM requests containers for all the map and reduce tasks (for jobs unqualified for uberization) from RM.

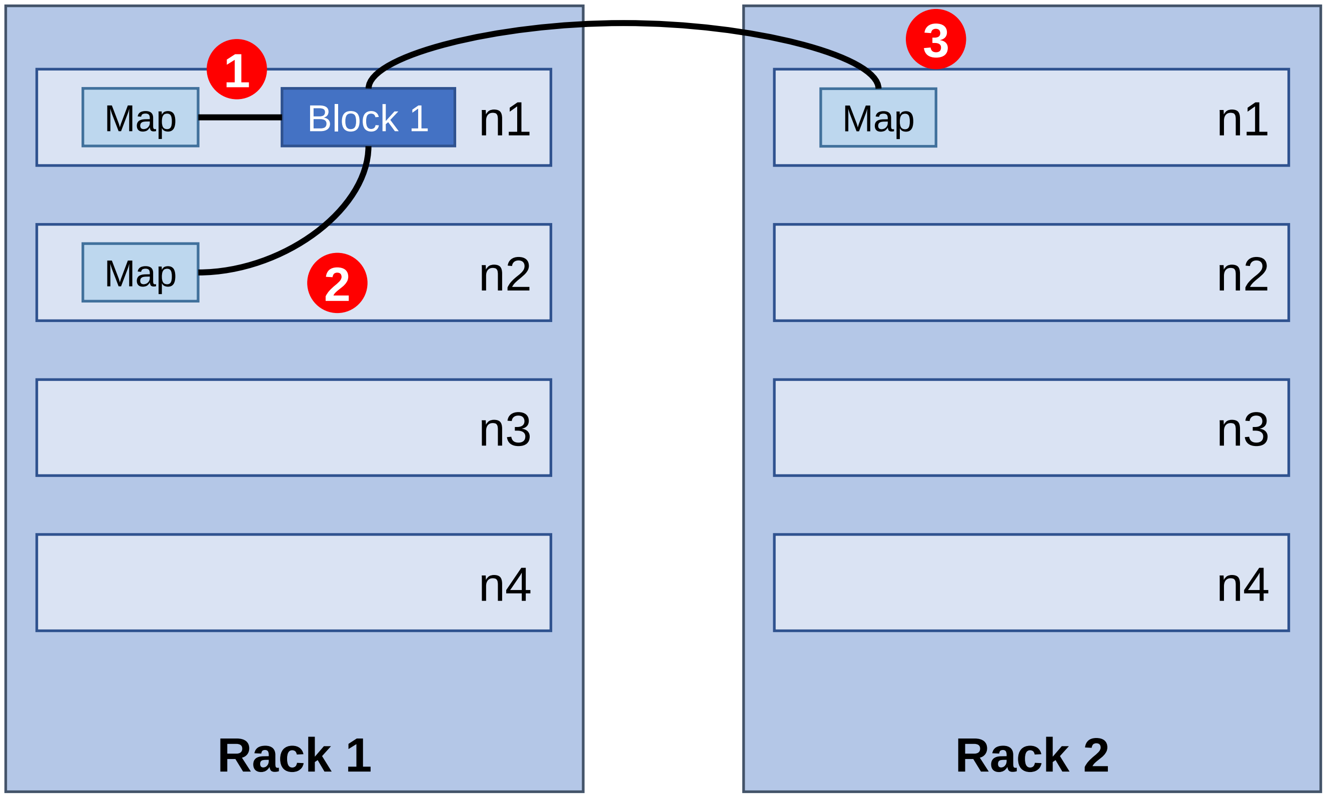
- *All the map tasks must complete before the sort phase can start → requests for reduce tasks will be made when 5% map tasks have completed*

Task assignment

Data locality constraints for Maps

- Reduce tasks can run anywhere in the cluster.
- 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.

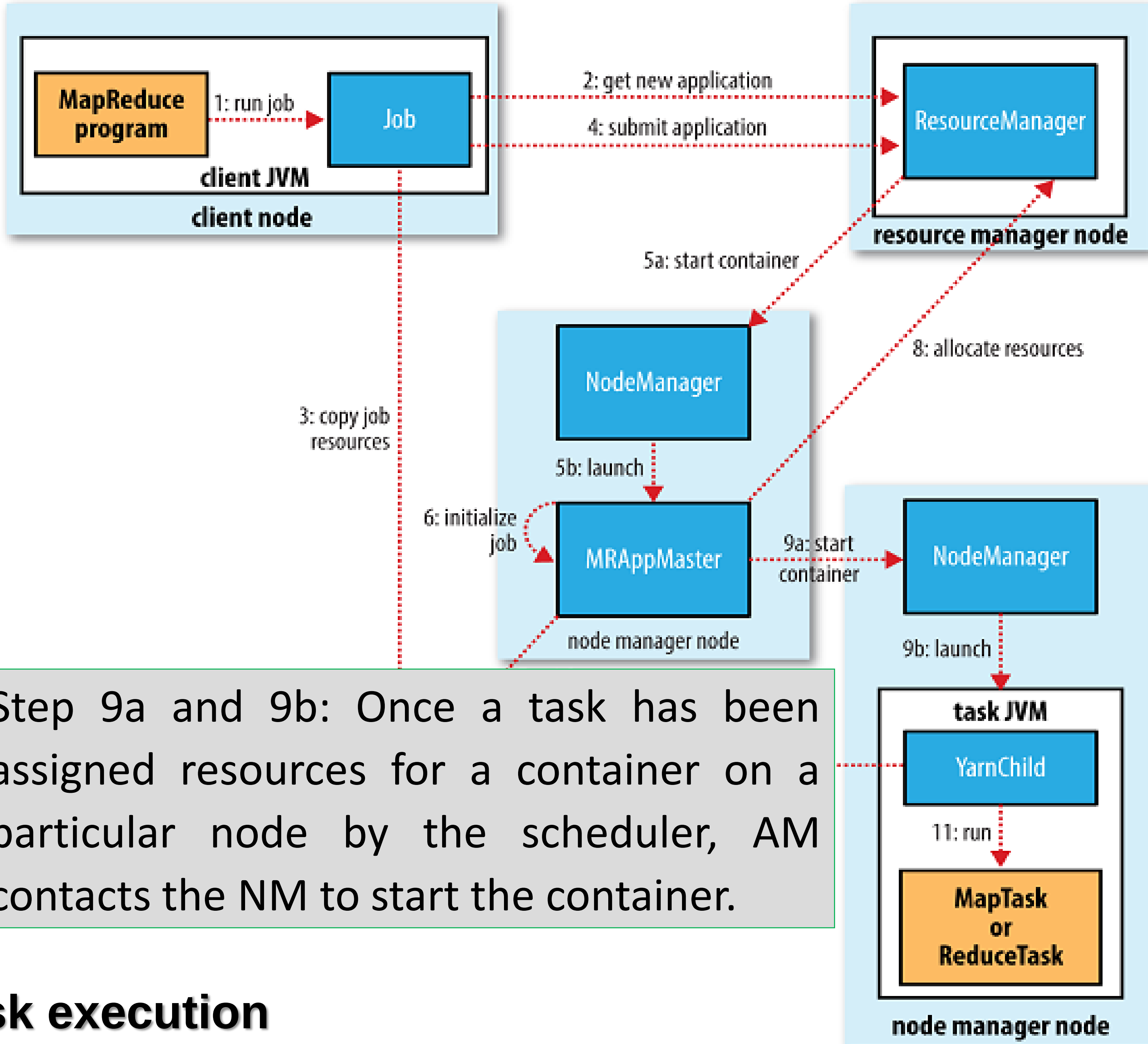
Data locality constraints for Maps



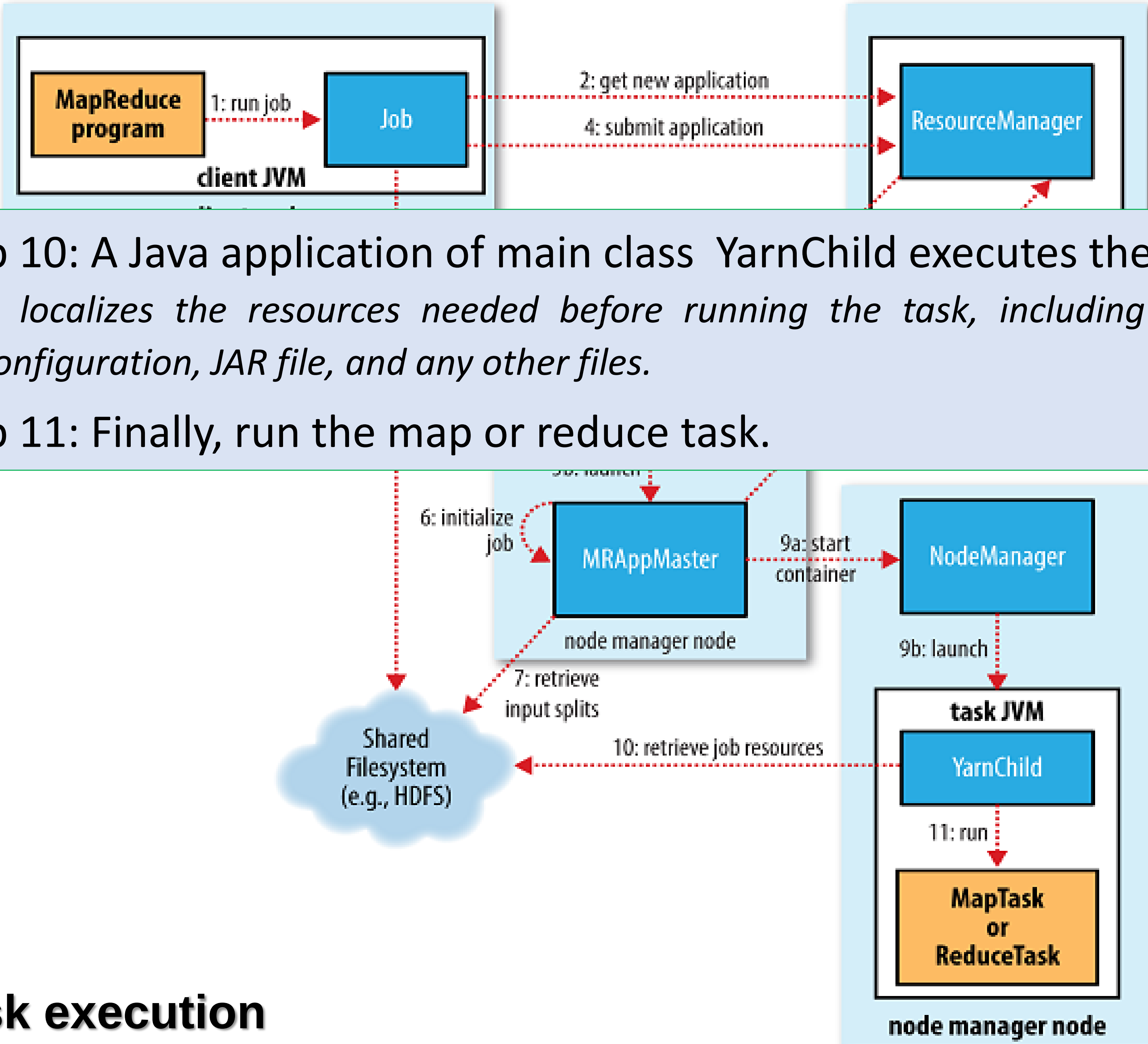
The administrator defines the topology in the *topology.script.file.name* property in *core-site.xml*

1 — 3 Best to worst options

Task execution



Step 9a and 9b: Once a task has been assigned resources for a container on a particular node by the scheduler, AM contacts the NM to start the container.



Step 10: A Java application of main class YarnChild executes the tasks.

- *It localizes the resources needed before running the task, including the job configuration, JAR file, and any other files.*

Step 11: Finally, run the map or reduce task.

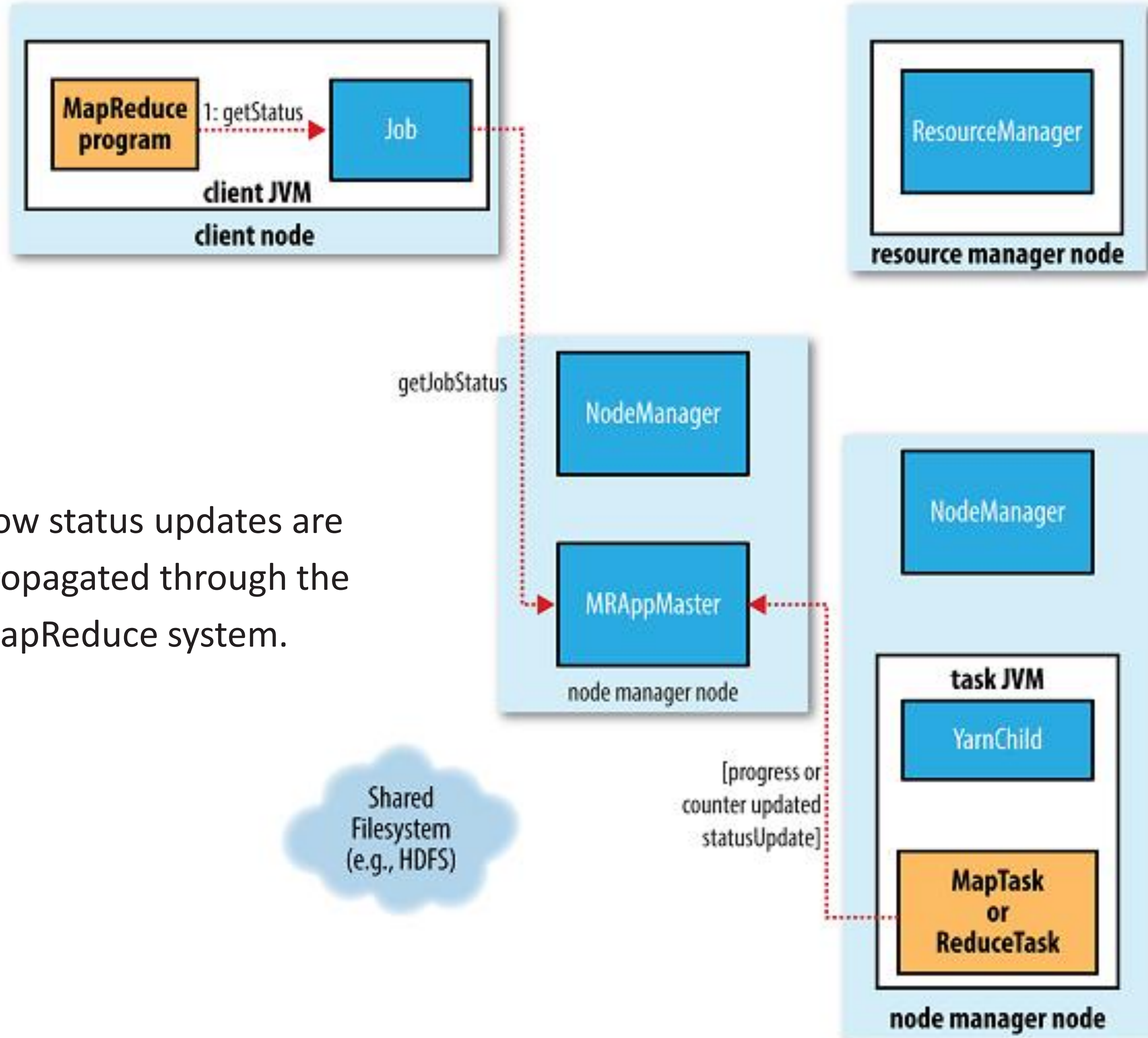
Task execution

Progress and status updates

- MapReduce jobs are **long-running batch jobs**, taking anything from tens of seconds to hours to run.
- A job and each of its tasks have a status, which includes
  - State: running, successfully completed, or failed
  - The progress of map and reduce tasks
  - The values of the job counters or task 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 task's progress is tracked while it is running.
  - **Map task:** the proportion of the input processed
  - **Reduce task:** the proportion of the reduce input processed
- A task reports its progress and status back to its AM via the umbilical interface every three seconds.
- Client gets the job's latest status by polling AM every second
  - Instead, obtain a JobStatus instance from Job's getStatus() method
  - mapreduce.client.progressmonitor.pollinterval



How status updates are propagated through the MapReduce system.

Job completion

- AM turns the status of a job to "successful" when the last task for a job is complete.
  - It also sends an HTTP job notification if it is configured to do so.
- `waitForCompletion()` method is returned and a message is printed to the console.
  - Job statistics and counters are also shown at this point.
- AM and task containers clean up their working state
  - Intermediate output is thus deleted.
  - Job information is archived by the job history server to enable later interrogation by users if desired

Hadoop MapReduce counters

- There are Task Counters and Job Counters in a MR run.
  - Counters are either built into the framework or defined by users.

Built-in counters shown after a job has successfully completed.

```
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
```

See an example in “A test run” on page 27, Hadoop: The Definitive Guide, 4th edition

Hadoop MapReduce counters

Task Counters

- Gather information about tasks over their execution
  - Periodically sent from each task to the master units (i.e., TaskTracker → JobTracker, or Application Master → Resource Manager).
  - The results are aggregated over all the tasks in a job.
- 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 webUI.
  - E.g., PHYSICAL_MEMORY_BYTES, COMMITTED_HEAP_BYTES, etc.

Job Counters

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

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
<?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
```

hadoop

Map Reduce

works in phases

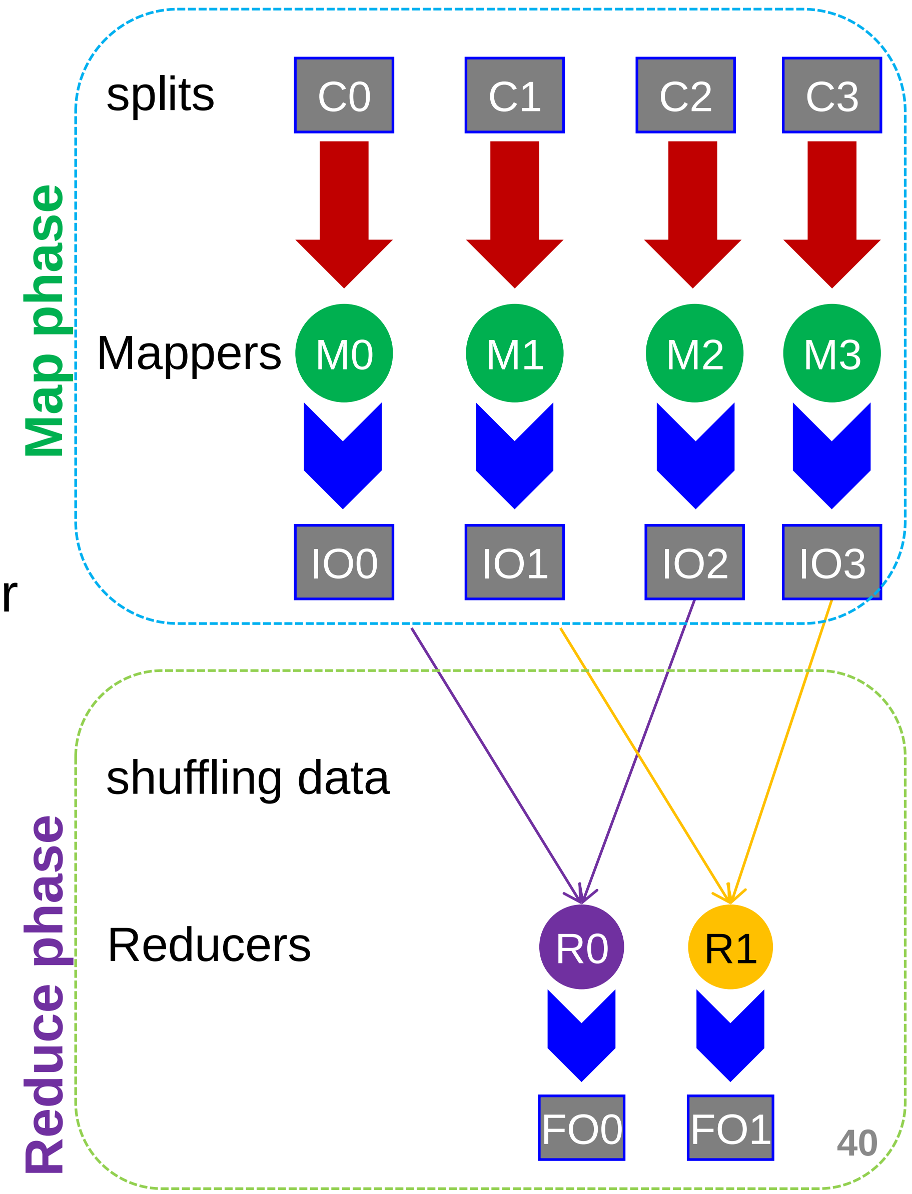
Phases of a MapReduce job

- The MR data flow includes **Map** and **Reduce** phases.

- **Mappers** first handles the splits.

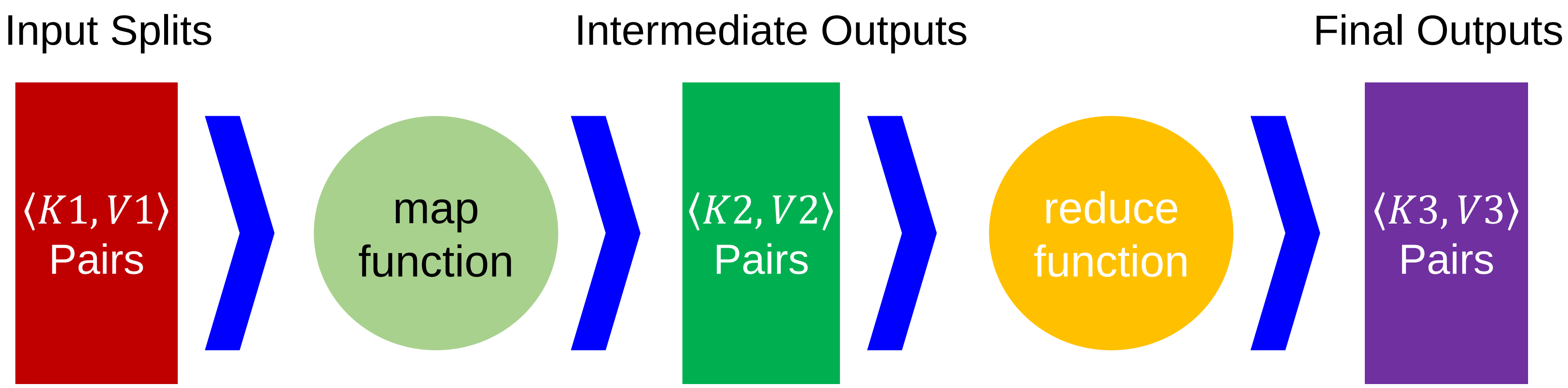
- **Sort and shuffling process** transfer IOs from Mappers into Reducers

- **Reducers** produce the final outputs (FOs).



Keys and Values

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

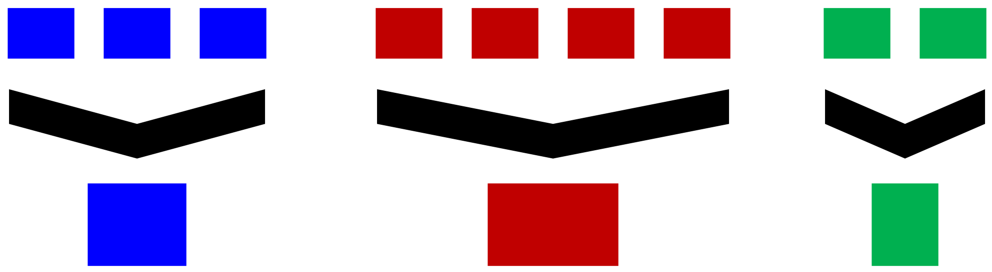


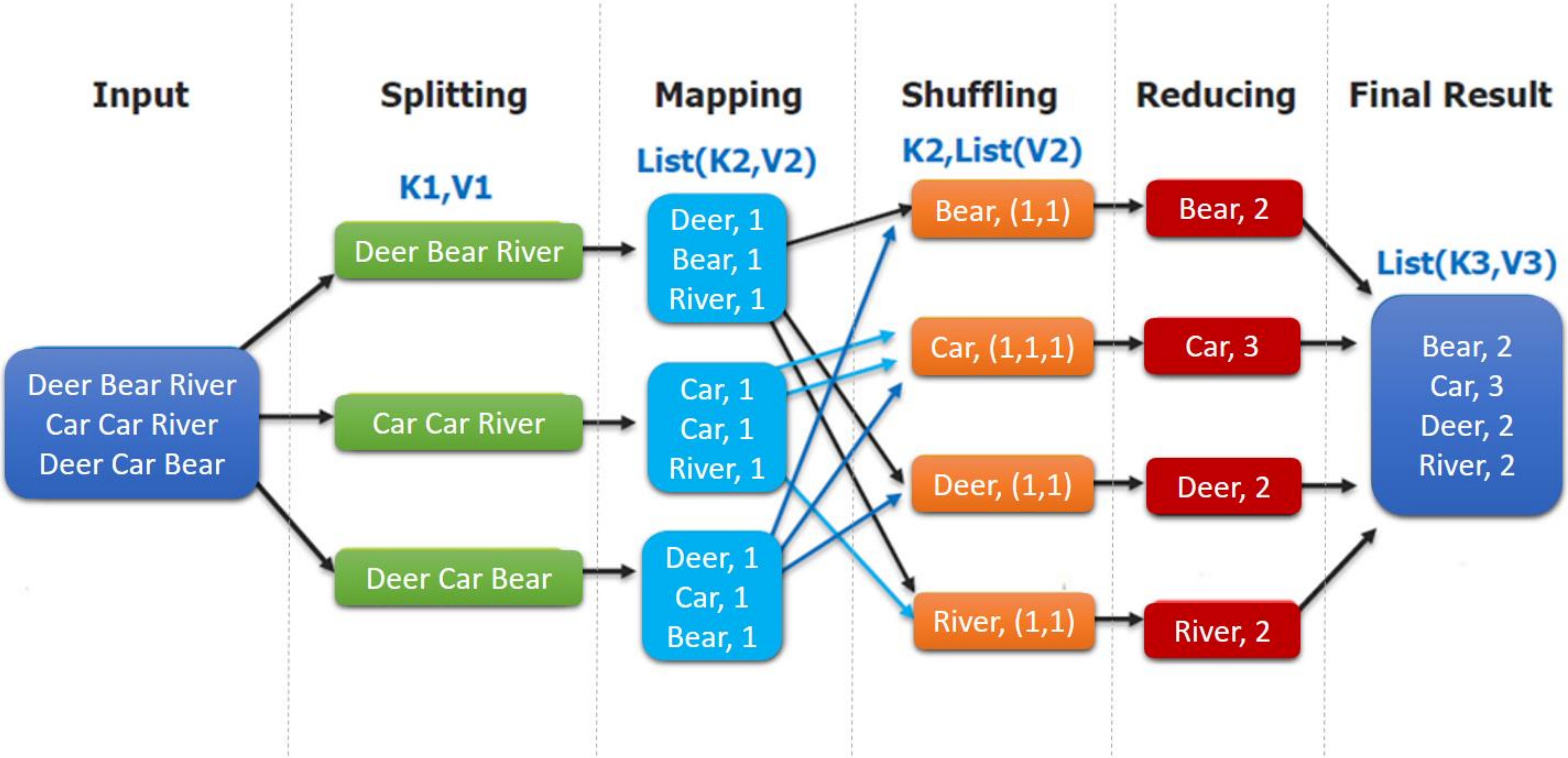
Partitions

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

Different colors represent different keys (potentially) from different Mappers

Partitions are the input to Reducers

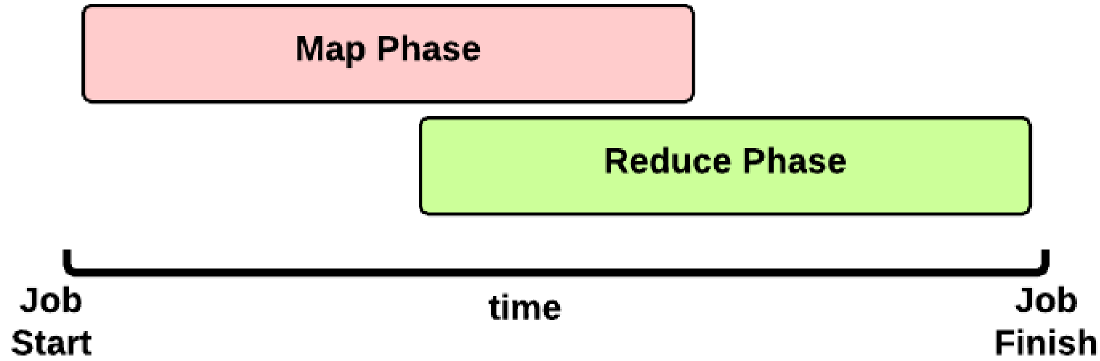


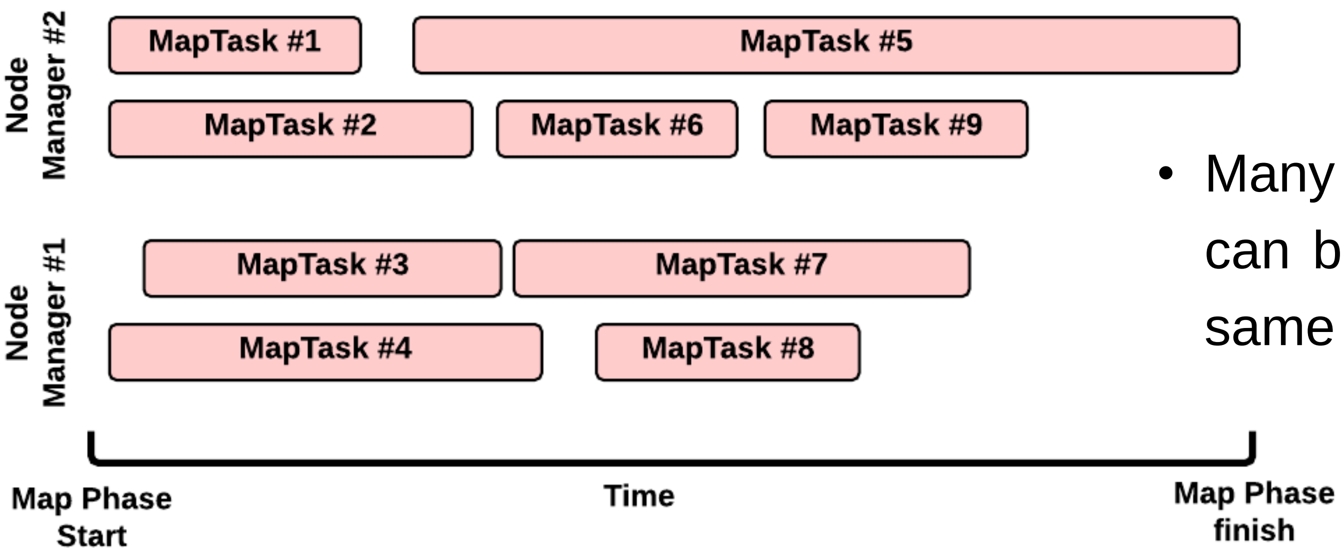


MapReduce Word Count Process

Lifecycle of a MapReduce job

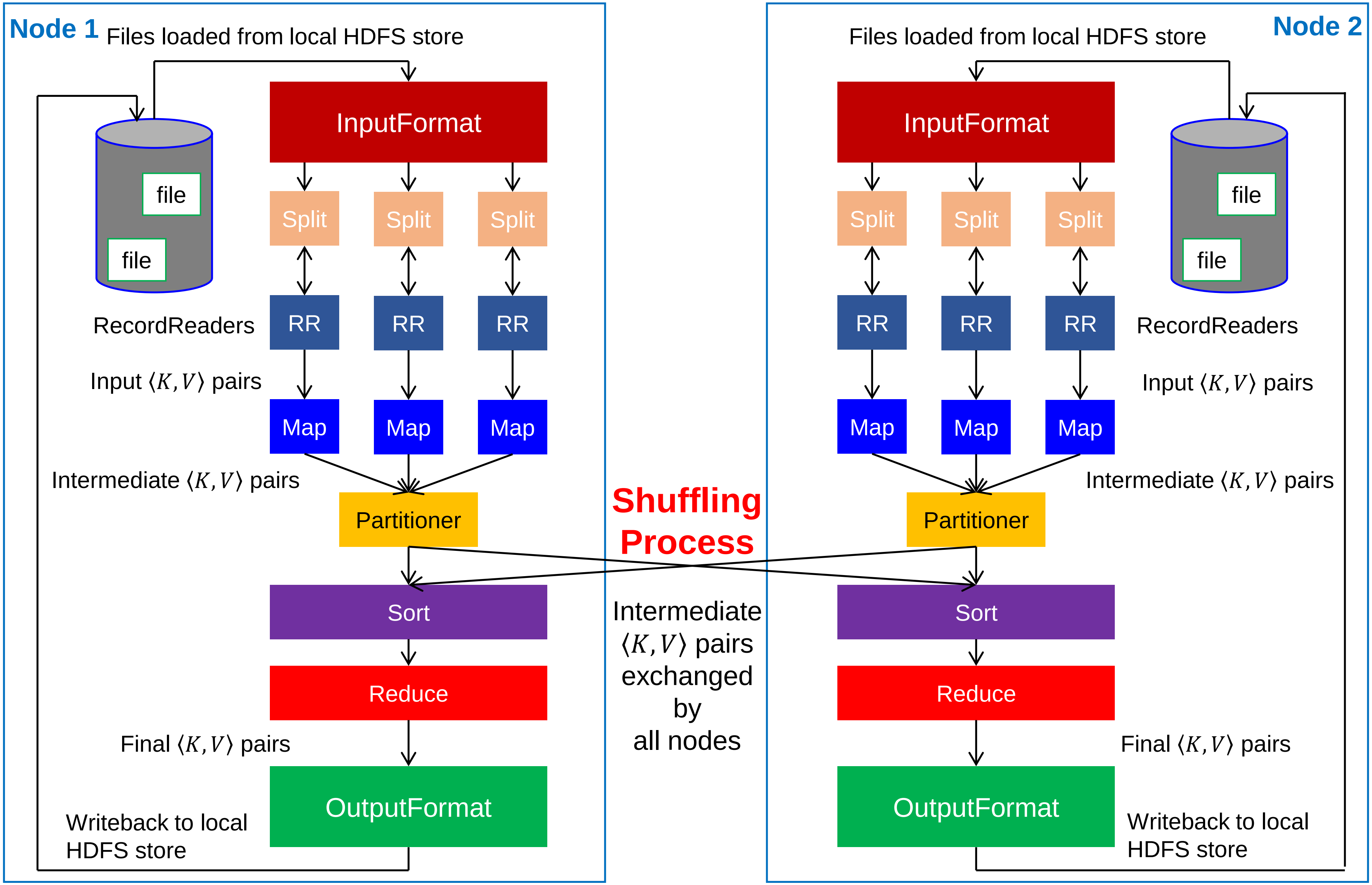
- An interleaving between map and reduce phases is possible.





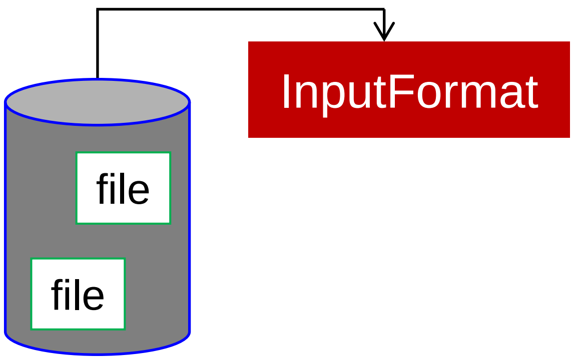
- Many map/reduce tasks can be launched on the same or different nodes.

A general MapReduce workflow



Input files and InputFormat

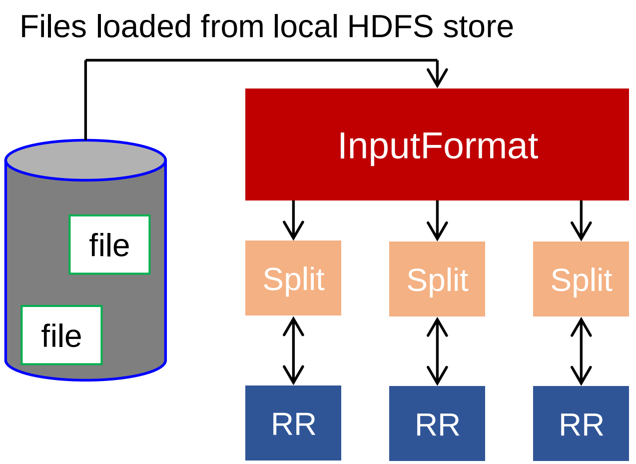
- Input files stores the initial data for a MR task, typically in a DFS.
  - Arbitrary format: line-based log files, binary files, multi-line input records, etc.



- InputFormat defines how the input files are split up and read.
  - Select the input files, define the input splits that break a file, and provide a factory for RecordReader objects that read the file
  - Several formats provided: TextInputFormat, KeyValueInputFormat, SequenceFileInputFormat, etc.

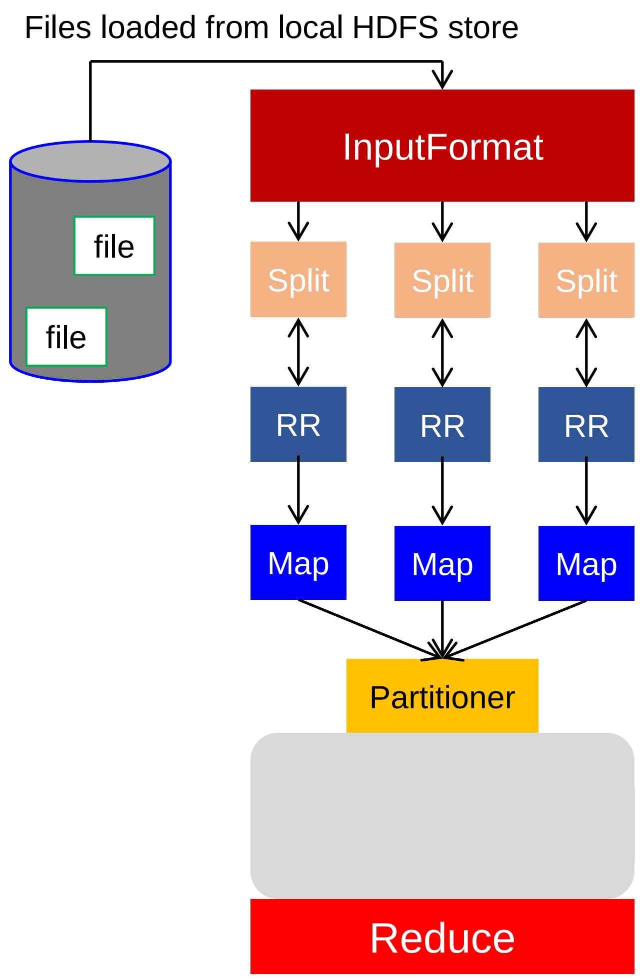
Input splits and RecordReader

- An input split describes a unit of work that makes a single Map task.
  - The InputFormat breaks a file up into 64MB splits by default.
  - Several Map tasks can operate on a single file in parallel, improving the performance on large files significantly.
- RecordReader loads the data 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.



Mapper, Reducer, and Partitioner

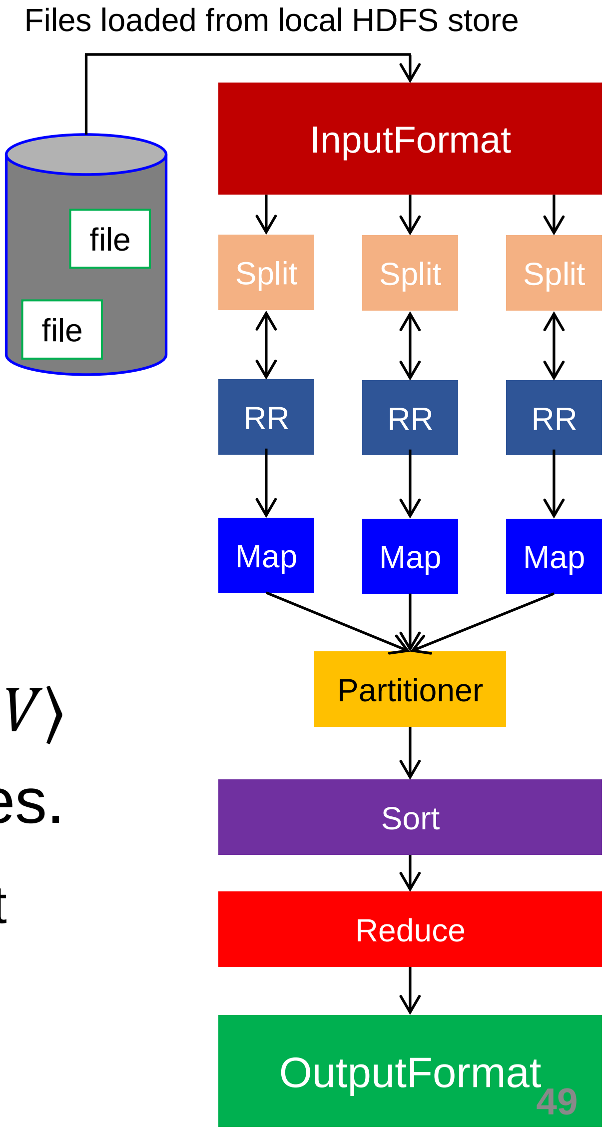
- Mapper does the user-defined work of the Map phase.
  - One Mapper instance for each split.
- Reducer performs the user-defined work of the Reduce phase.
  - One Reducer instance for each partition
  - The Reducer is called once for each key in the partition assigned to it.
- Partitioner decides to which partition a $\langle K, V \rangle$ pair from any Mapper will go.
  - HashPartitioner by default



Sort and Output Format

- Intermediate $\langle K, V \rangle$ pairs can be sorted in both map and reduce phases.
  - Map phase: during the process of partition, sort and spill to disk
  - Reduce phase: during the merge process, before the Reducer works

- OutputFormat defines how to write $\langle K,V \rangle$ pairs produced by Reducers to output files.
  - TextOutputFormat, SequenceFileOutputFormat

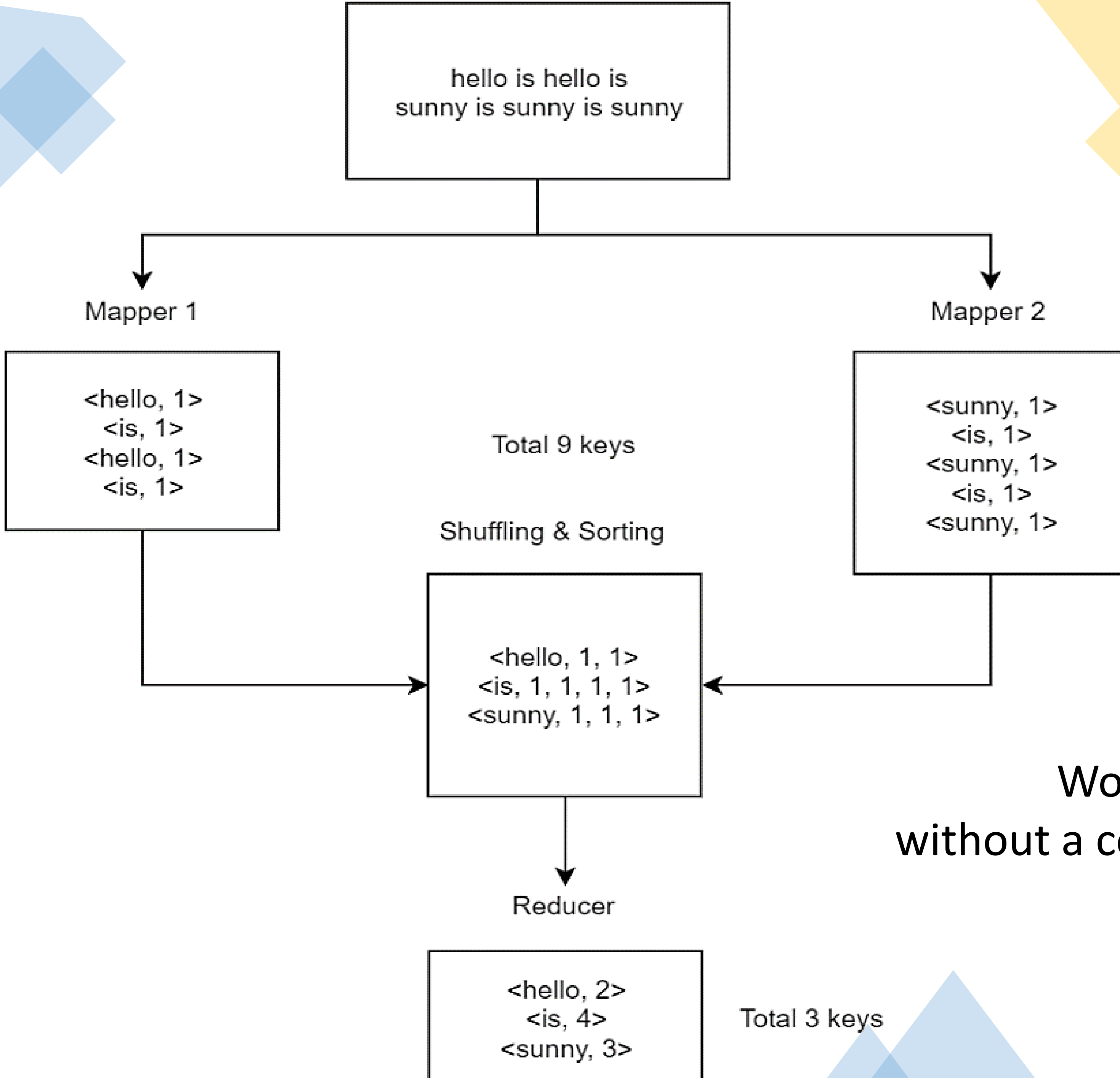


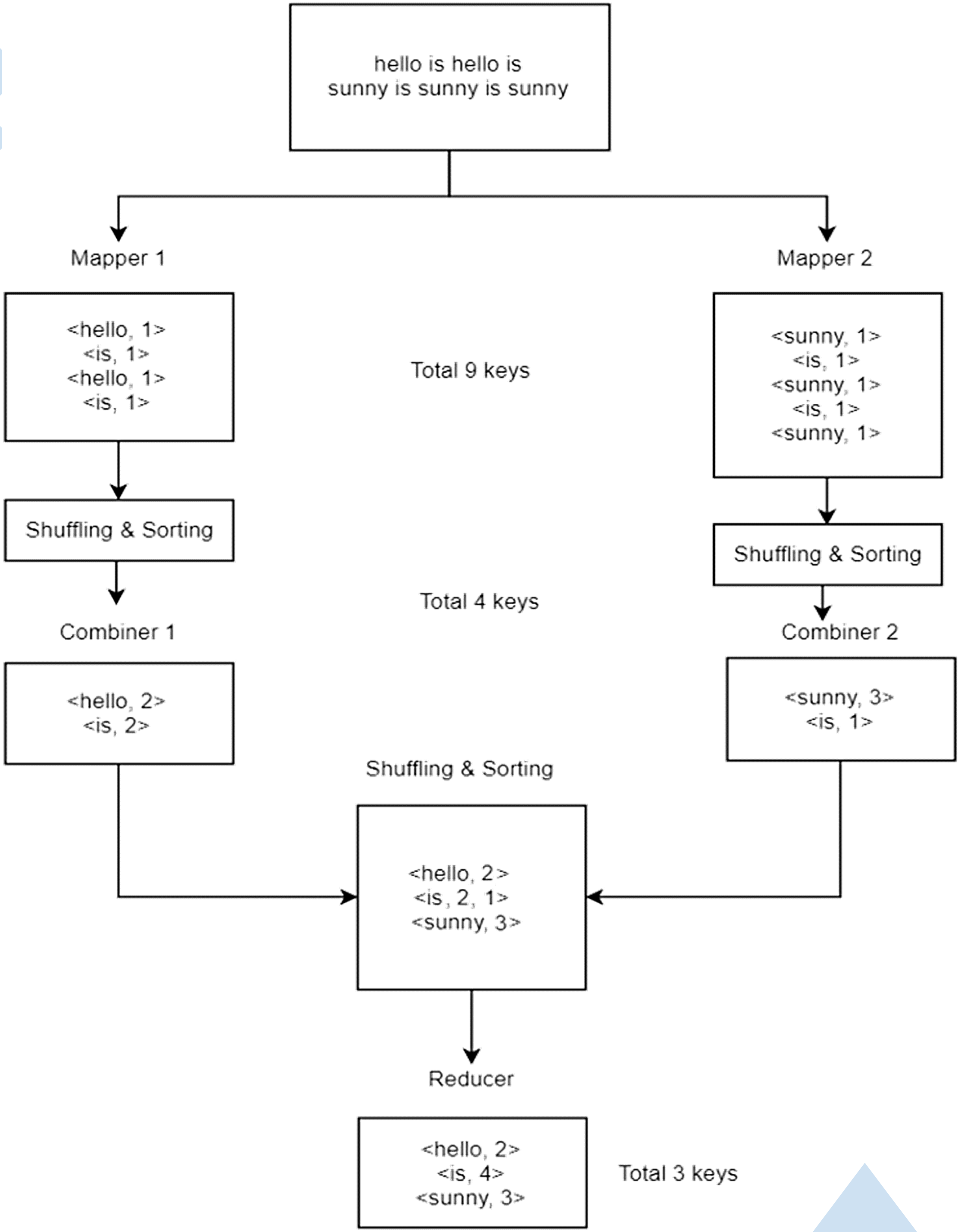
Combiner function

Combiner function

- MR jobs are limited by the bandwidth on the cluster.
- A combiner function helps cutting down the amount of data shuffled between the maps and reduce tasks.
    - It operates on each map task, and it must be set with Reducer class.
    - Input: all key-value pairs from a map task.
    - Output: key-value collection pairs, which will be fed to reducer tasks.
- Combiner function does not replace reduce function.
    - A reducer handles records of the same key from different mappers.
- Not all functions are eligible for combiners.
    - E.g., mean(1, 2, 3) = 2 $\neq$ mean(mean(1, 2), mean(3)) = 2.25

WordCount without a combiner



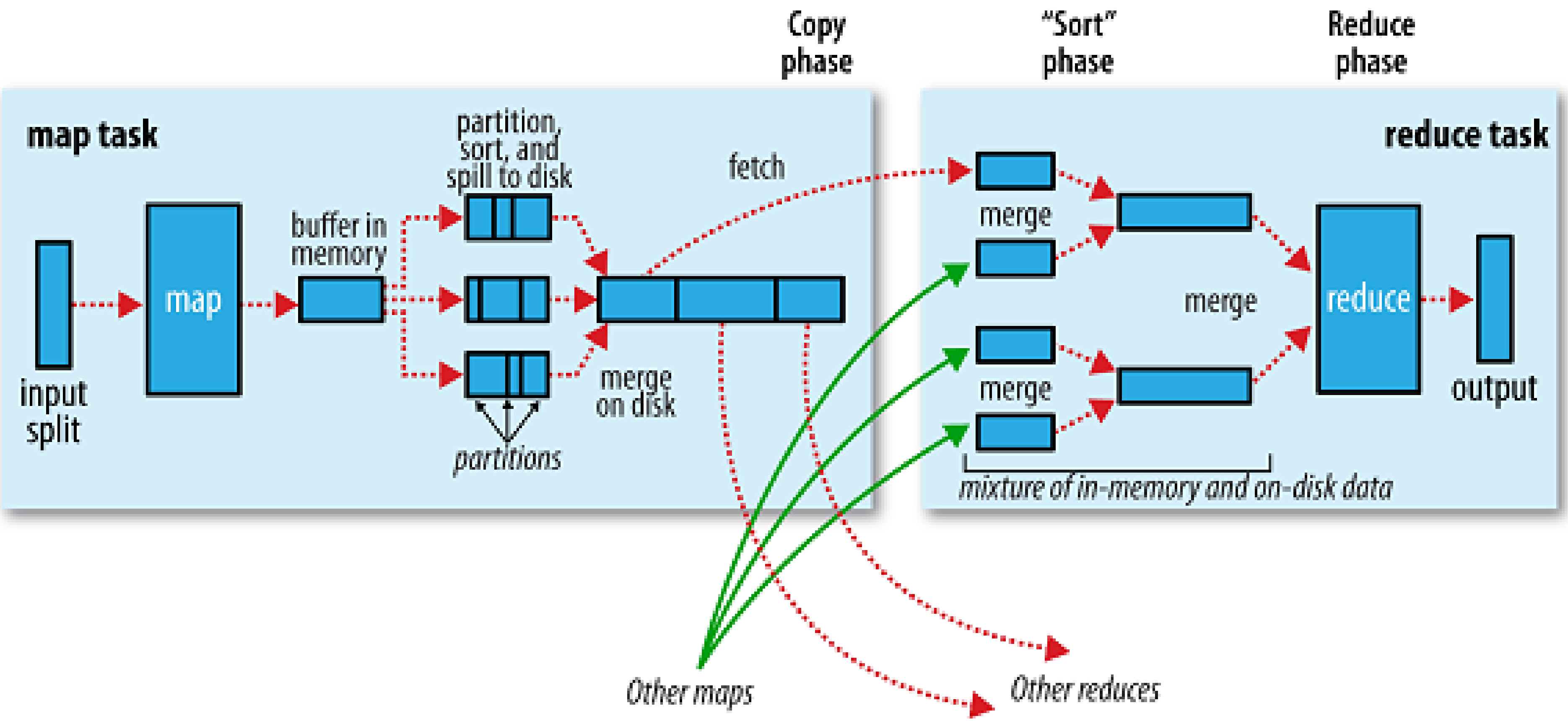


WordCount with a combiner

Shuffle and Sort

Shuffle and Sort in MapReduce

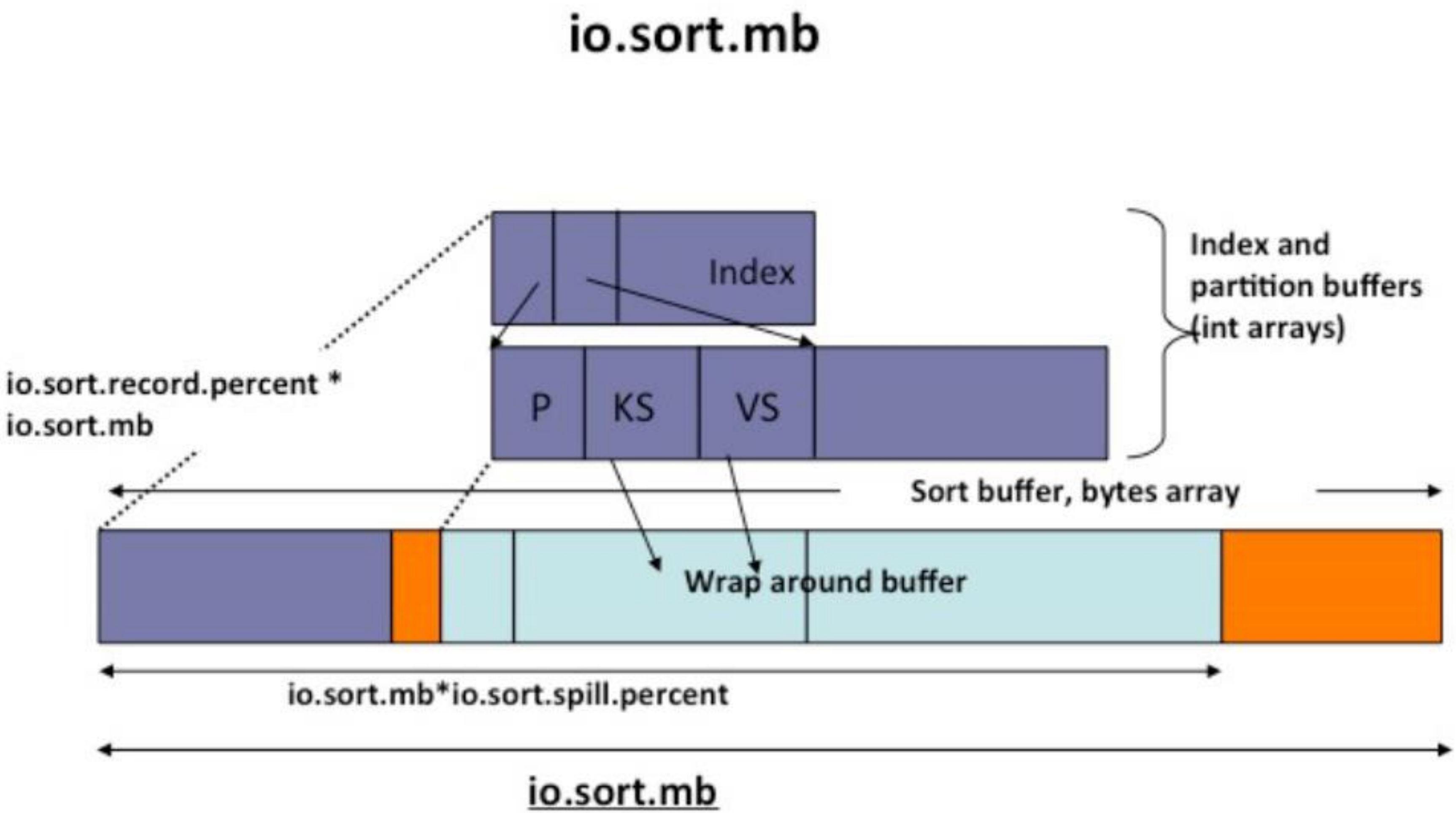
- The system performs the sort and transfers the map outputs to the Reducers as inputs.



Map side: Write outputs to buffer

- Each map task writes the outputs to a circular memory buffer.
  - 100 MB by default
  - Buffering writes in memory allows some presorting for efficiency reasons.
- When the buffer content reaches a certain threshold size, a background thread will start to spill the data to disk.
  - Map outputs keeps going 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.
- The background thread divides the data into partitions corresponding to the Reducers to which they will ultimately be sent.
  - An in-memory sort by key is done within each partition.
  - A combiner may further process the sort's outputs to lessen the data written to local disk and transferred to the Reducer.

Map side: The in-memory buffer



Map side: Spill the data to disk

- Each time the buffer reaches the threshold, a new spill file is created.
  - There may be multiple spill files after the map task has done its last write.
- The spill files are merged into a single partitioned and sorted output file when the task is finished.
  - If there are at least three spill files, run the Combiner again before writing the output file.

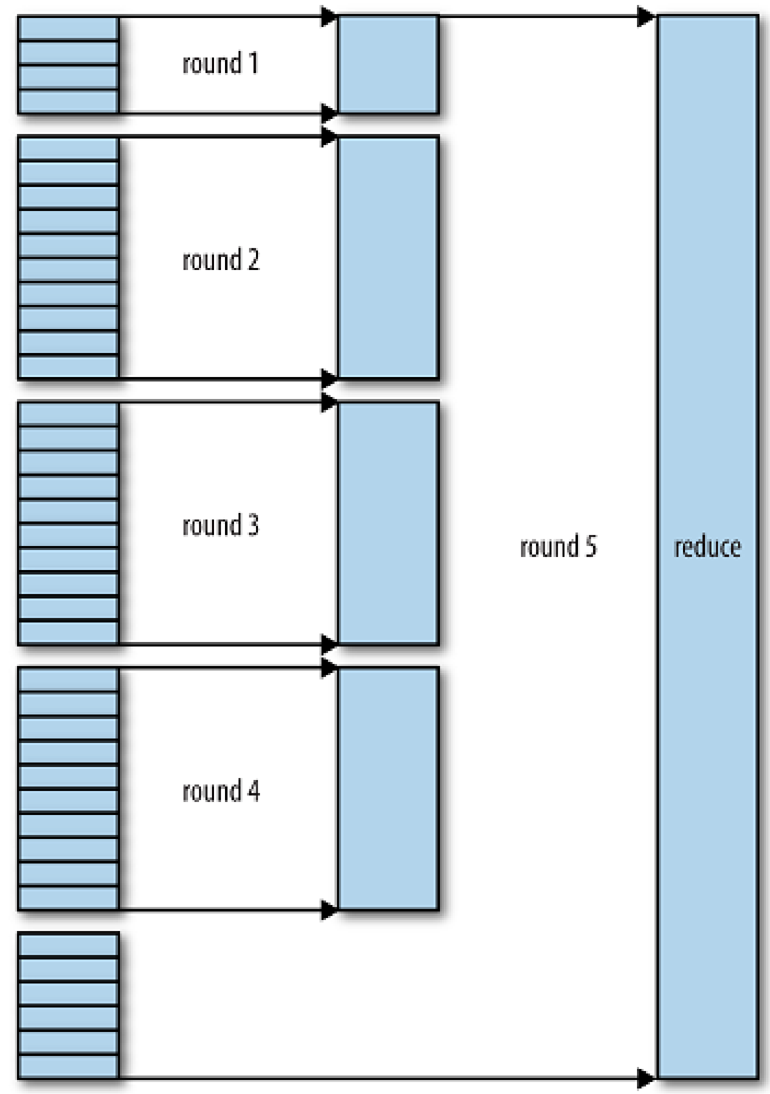
Reduce side: Copy the data

- **Copy phase:** the Reduce task uses a small number of copier threads to fetch map outputs in parallel.
  - Map tasks may finish at different times $\rightarrow$ the reduce task starts copying their outputs as soon as each completes.
- Small map outputs $\rightarrow$ copy them to the Reduce task JVM's memory

  Otherwise, copy to disk.
- The buffer merges the data and spills it to disk when reaching a threshold
  - A Combiner may run during the merge to reduce the data written to disk.
  - As the copies accumulate on disk, a background thread merges these files into larger and sorted files.
- Note that any compressed map outputs mut be decompressed in memory to perform a merge on them.

Reduce side: Sort the data

- **Sort phase:** When all the map outputs have been copied, they are merged **in rounds** to maintain their sort ordering.
- For example, 50 map outputs and merge factor 10 $\rightarrow$ 5 rounds.
    - Each round would merge 10 files into 1, and there would be 5 intermediate files.
- Rather than have a final round that merges the intermediate files into a single sorted file, the merge directly feed the reduce function.



Handling failures in MapReduce

Failures in MapReduce

- 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

- User code in the map/reduce task throws a runtime exception
- The sudden exit of the task JVM due to a JVM bug.
  - *NM notices that the process has exited and informs AM to mark the attempt as failed.*
  - *AM frees up the container to save the resources for other tasks.*

- Hanging task: AM notices the absence of a progress update for a while (10 mins by default)
  - *AM marks the task as failed.*
  - *The task JVM process will be killed automatically after that.*

Task failure

- AM avoids rescheduling a task on a NM where it has failed.
- Any failed task will be retried for a few attempts.
  - mapreduce.map (or reduce).maxattempts, (by default, 4)
  - The whole job fails if exceeded
- The results of a job are still useful even when a few tasks fail $\rightarrow$ undesirable to abort the job
  - mapeduce.map (or reduce).failures.maxpercent: the maximum % tasks allowed to fail without triggering job failure (per job)
- A task may also be intentionally killed (i.e., itself not failed).
  - E.g., a NM fails $\rightarrow$ AM marked all the task running on it as killed.
  - Killed task attempts do not affect the task's number of attempts.

Application Master failure

- RM spots failure when there is no heartbeat from AM.
- Any failed AM will be retried for a limited number of attempts
  - mapreduce.am.max-attempts property (by default, 2)
  - The whole job fails if exceeded
- RM starts a new AM in a new container (managed by a NM)
  - It uses the **job history** to recover the tasks run by the failed AM.
  - Completed tasks do not have to run again.
    - yarn.app.mapreduce.am.job.recovery.enable (default = true).
- The client will experience a timeout when it issues a status update to the failed AM, while it is transparent to the user.
  - The client asks RM for the new instance's address.

Node Manager failure

- A failed NM stops sending heartbeats to the RM or does it very infrequently.
  - yarn.resourcemanager.nm.liveness-monitor.expiry-interval-ms
- RM removes the failed node from its pool of nodes to schedule containers on.
- Any task or AM running on the failed NM are recovered using the previous mechanisms.
  - Map tasks belonging to an incomplete jobs will be rerun, though they were run and complete successfully.

Node Manager failure: Blacklist

- AM may blacklist a NM if its number of failures for applications is high, even if the node is still fine.
  - mapreduce.job.maxtaskfailures.per.tracker (by default, 3).
- RM does not do blacklisting across applications.
  - Tasks of a new job may be scheduled on bad nodes, which have been blacklisted by an AM running earlier.

Resource Manager failure

- SPOF in the default configuration, serious failure.
- High availability: run a pair of RMs in an active-standby mode
  - Information about all the running applications is stored in a highly available state store (backed by ZooKeeper or HDFS).
- A **failover controller** manages the transition from standby to active nodes (e.g., ZooKeeper).
  - Clients and NMs are configured to try connecting to each RM in a **round-robin fashion** until they find the active one.
- The new RM restarts the AMs for all the applications running on the cluster.
  - yarn.resourcemanager.am.max-attempts is not affected

Speculative execution

- Straggling task: A slow-running task that makes the job execution time is significantly longer than it would have been.
  - E.g., hardware degradation or software misconfiguration, etc.
- Speculative execution: Hadoop launches an equivalent task for any task that is running slower than expected
  - The scheduler tracks the progress of all tasks of the same type (map/reduce) in a job.
  - Speculative duplicates are only for a small proportion running significantly slower than the average.
- When a task completes well, all duplicate tasks are killed.

Speculative execution

- An optimization, not a feature to make jobs run more reliably
  - E.g., problems caused by user-code bugs
- There are cases that you may want to turn speculative execution off!
- It may reduce the overall throughput on a busy cluster.
- It is not applicable for nonidempotent tasks
- Speculative execution for reduce tasks can significantly increase network traffic on the cluster.

A simple MapReduce program

About the NCDC dataset

- Weather sensors collect data every hour at many locations across the globe and gather a large volume of log data.
- Datafiles are grouped by date and weather station.
  - A directory for each year, from 1901 to 2001
  - Each directory contains a gzipped file for a weather station with its readings for that year.

```
010010-99999-1990.gz
010014-99999-1990.gz
010015-99999-1990.gz
010016-99999-1990.gz
010017-99999-1990.gz
```

The first entries for 1990

- The data is stored following the line-oriented ASCII format.
  - A rich set of meteorological elements, many of which are optional or with variable data lengths.

```
0035029070999991902010106004+64333+023450FM-12+000599999V0201401N011819999999N0000001N9-00941+99999100551ADDGF104991999999999999999999MW1381
0035029070999991902010113004+64333+023450FM-12+000599999V0201401N011819999999N0000001N9-01001+99999100311ADDGF104991999999999999999999MW1381
0035029070999991902010120004+64333+023450FM-12+000599999V0201401N013919999999N0000001N9-01171+99999100121ADDGF108991999999999999999999MW1381
0035029070999991902010206004+64333+023450FM-12+000599999V0200901N009819999999N0000001N9-01611+99999100121ADDGF108991999999999999999999MW1381
0029029070999991902010213004+64333+023450FM-12+000599999V0200901N011819999999N0000001N9-01721+99999100121ADDGF108991999999999999999999
0029029070999991902010220004+64333+023450FM-12+000599999V0200901N009819999999N0000001N9-01781+99999100421ADDGF108991999999999999999999
0029029070999991902010306004+64333+023450FM-12+000599999V0209991C000019999999N0000001N9-01781+99999100871ADDGF108991999999999999999999
0035029070999991902010313004+64333+023450FM-12+000599999V0209991C000019999999N0000001N9-01721+99999100901ADDGF108991999999999999999999MW1721
```

```
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
```

Analyzing the data

- ***Goal****: 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
  - It extracts those pieces of data so that the **reduce function** can find the maximum temperature for each year.
  - Bad records with missing or erroneous temperatures are filtered out.

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...)
```

- Extract the year and air temperature and emit them as output
  - The temperature values have been interpreted as integers.

Transfer from Map to Reduce

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

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

Map outputs

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

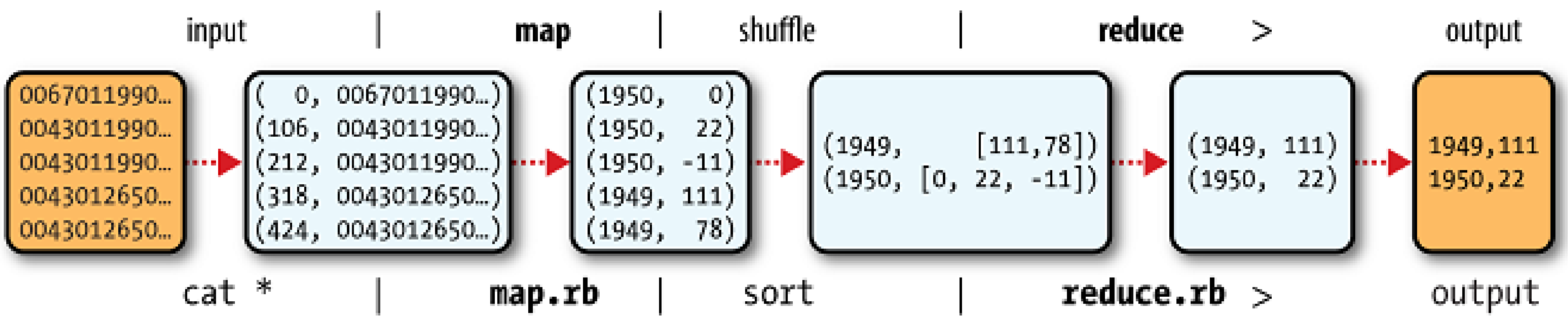
Inputs to reducers

Analyzing the data: Reduce function

- Iterate through the list and pick up the maximum reading

```
(1949, 111)
(1950, 22)
```

- The MapReduce logical dataflow



```java
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 for the **`map`** function.
  - Input key, input value, output key, and output value types.
  - `map()` gets a key and a value, and writes the output to an instance of the `Context` class.
- Implementation
  - Convert the Text value of the input line into a Java String.
  - Extract the interested columns with `substring()`
  - Write the year as a `Text` object and temperature as an `IntWritable`
    - An output record is written only if the temperature is present, and the quality code indicates the temperature reading is OK.

```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

- The **`Reducer class`** has four formal type parameters for the **`reduce`** function.
  - The input types of the reduce function must match the output types of the map function.
- Implementation
  - Input: `Text – IntWritable`. Output: `Text – IntWritable`.
  - Iterate through the temperatures and compare each with the highest value found so far
  - Output a record for each year and its maximum temperature

```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: main function

- A **`Job object`** forms the specification of the job and controls over how the job is run.
- The code is compiled into a JAR file, which is distributed in the cluster.
  - If `setJarByClass()` is called, Hadoop locates the relevant JAR file by looking for the one that contains this class.
- `setMapperClass()` and `setReducerClass():` define the **`map`** and **`reduce`** steps, respectively
- `waitForCompletion()` submits the job and waits for it to end.
  - The single flag indicates whether information about the progress is printed to the console.
  - The return Boolean value indicates success (true) or failure (false), which is translated into the program's exit code of 0 or 1.

Source code: main function

- `FileOutputFormat.setOutputPath()` sets the `output path` (of which there is only one)
  - It specifies a directory to write the output files from the reduce function.
- `FileInputFormat.setInputPath()` specifies the `input path`.
- `setOutputKeyClass()` and `setOutputValueClass()` determines 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.
  - If different, we can set these types using `setMapOutputKeyClass()` and `setMapOutputValueClass()`.

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.

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)
```

- Without combiners, the reduce function received **`(1950, [0, 20, 10, 25, 15])`** and produced **`1950 25`**.
- A combiner finds the maximum temperature for each map output.
- Thus, the reduce function would be called with **`(1950, [20, 25])`**.

```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);
```


...the end.