Unit – II

How MapReduce Works: Anatomy of a MapReduce Job Run, Job Submission, Job Initialization, Task Assignment, Task Execution, Progress and Status Updates, Job Completion, Failures, Task Failure, Application Master Failure, Node Manager Failure, Resource Manager Failure, Shuffle and Sort, The Map Side, The Reduce Side

MapReduce Types and Formats: MapReduce Types, The Default MapReduce Job, Input Formats, Input Splits and Records, Text Input, Output Formats, Text Output

Anatomy of a MapReduce Job Run

- You can run a MapReduce job with a single method call: **submit()** on a Job object.
- you can also call **waitForCompletion(),** which submits the job if it hasn't been submitted already, then waits for it to finish

- At the highest level, there are **five independent entities**: (Refer figure 1)

1. The **client**, which submits the MapReduce job.
2. The YARN **resource manager**, which coordinates the allocation of compute resources on the cluster.
3. The YARN **node managers**, which launch and monitor the compute containers on machines in the cluster.
4. The MapReduce **application master**, which coordinates the tasks running the MapReduce job. The application master and the MapReduce tasks run in containers that are scheduled by the resource manager and managed by the node managers.
5. The distributed file system (normally **HDFS**, covered in Chapter 3), which is used for sharing job files between the other entities.

- The whole process of map reduce is consists of the following steps: (Refer figure 1)

  1. Job submission
  2. Job Initialization
  3. Task Assignment
  4. Task Execution

     Streaming
  5. Progress and status update
  6. Job Completion

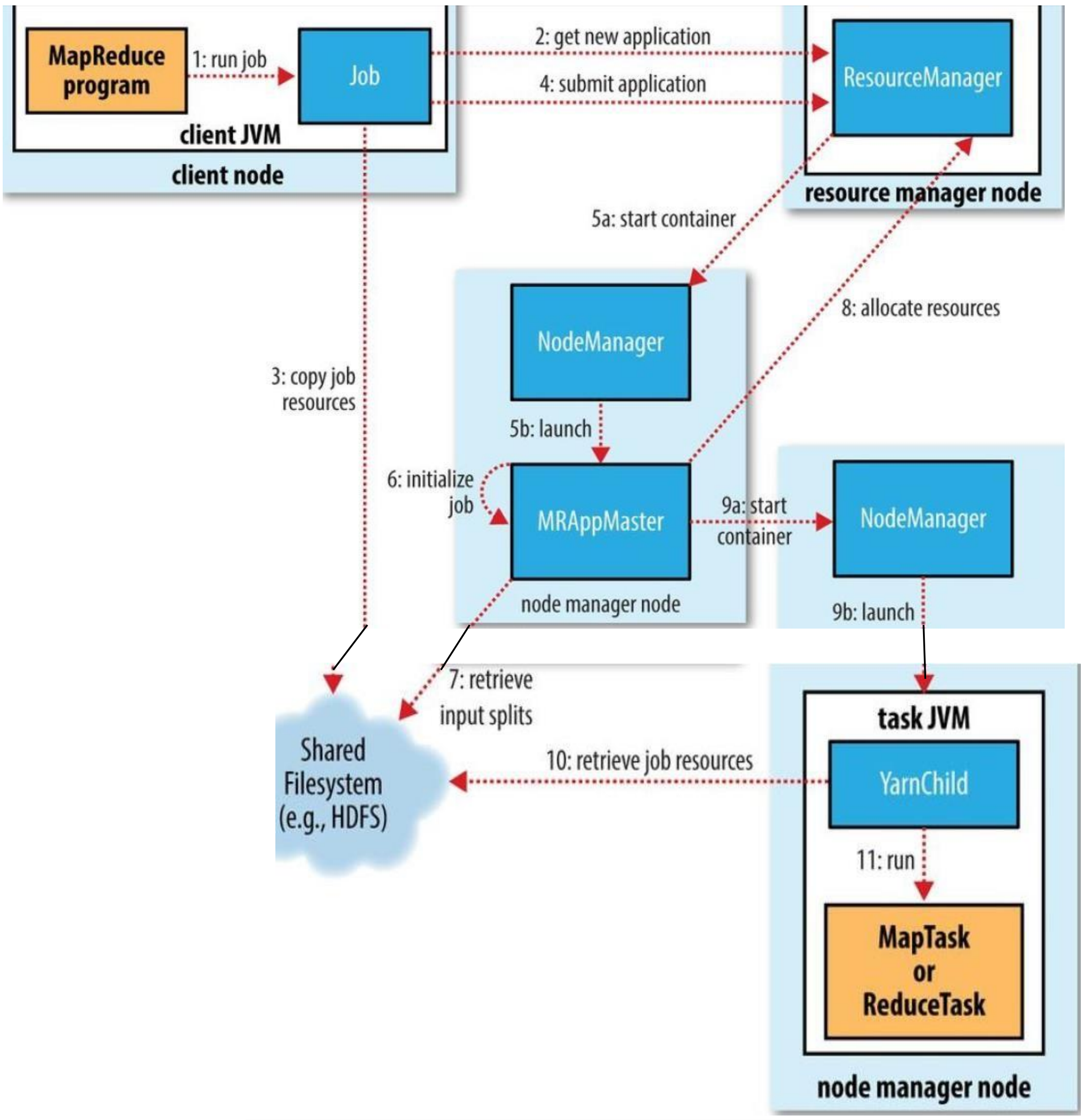


Figure 1: How Hadoop runs a MapReduce job

Job Submission

- The **submit()** method on Job creates an internal **JobSubmitter** instance and calls submitJobInternal() on it.
- Having submitted the job, **waitForCompletion()** polls the job's progress once per second and reports the progress to When the job completes successfully, the job counters are displayed.
- Otherwise, the error that caused the job to fail is logged to the console.
- The **job submission process implemented by JobSubmitter** does the following:

1. Asks the resource manager for a **new application ID**, used for the MapReduce job ID (step 2).
2. Checks **the output specification** of the job. For example, if the output directory has not been specified or it already exists, the job is not submitted and an error is thrown to the MapReduce program.
3. **Computes the input splits** for the job. If the splits cannot be computed (because the input paths don't exist, for example), the job is not submitted and an error is thrown to the MapReduce program.
4. **Copies the resources** needed to run the job, including the job JAR file, the configuration file, and the computed input splits, to the shared file system in a directory named after the job ID (step 3). The job JAR is copied with a **high replication factor** (which defaults to 10) so that there are lots of copies across the cluster for the node managers to access when they run tasks for the job.
5. **Submits the job** by calling submitApplication() on the resource manager (step 4).

Job Initialization

- When the resource manager receives a call to its submitApplication() method, it 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 (steps 5a and 5b).
- The application master for MapReduce jobs is a Java application whose main class is MRAppMaster. It initializes the job by creating a number of bookkeeping objects to keep track of the job's progress, as it will receive progress and completion reports from the tasks (step 6).
- Next, it retrieves the input splits computed in the client from the shared filesystem (step 7). It then creates a map task object for each split, as well as a number of reduce task objects. Tasks are given IDs at this point.
- The application master must decide how to run the tasks that make up the MapReduce job.
- If the job is small, the application master may choose to run the tasks in the same JVM as itself. This happens when it judges that the overhead of allocating and running tasks in new containers outweighs 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.*
- **What qualifies as a small job?** By default, a small job is one that has less than 10 mappers, only one reducer, and an input size that is less than the size of one HDFS block. (Note that Uber tasks must be enabled explicitly (for an individual job, or across the cluster) by setting mapreduce.job.ubertask.enable to true.
- Finally, before any tasks can be run, the application master calls the setupJob() method on the OutputCommitter. For FileOutputCommitter, which is the default, it will create the final output directory for the job and the temporary working space for the task output.

Task Assignment

- **If the job does not qualify for running as an uber task,** then the application master requests containers for all the map and reduce tasks in the job from the resource manager **(step 8).**
- Requests for **map tasks are made first** and 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
- **Reduce tasks can run anywhere in the cluster, but requests for map tasks have data locality constraints that the scheduler tries to honor.**
- In the optimal case, the task is **data local —** that is, running on the same node that the split resides on.
- Alternatively, the task may be **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, you can determine the number of tasks that ran at each locality level by looking at the job's counters.
- Requests also specify memory requirements and CPUs for tasks. By default, each map and reduce task is allocated 1,024 MB of memory and one virtual core.

Task Execution

- Once a task has been assigned resources for a container on a particular node by the resource manager's scheduler, the application master starts the container by contacting the node manager (**steps 9a and 9b**).
- The task is executed by a Java application whose main class is **YarnChild**.
- Before it can run 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 10**).

- Finally, it runs the map or reduce task (**step 11**).
- The YarnChild runs in a dedicated JVM, so that any bugs in the user-defined map and reduce functions (or even in YarnChild) don't affect the node manager — by causing it to crash or hang, for example. Each task can perform setup and commit actions, which are run in the same JVM as the task itself and are determined by the OutputCommitter for the job
- For file-based jobs, the commit action moves the task output from a temporary location to its final location. The commit protocol ensures that when speculative execution is enabled, only one of the duplicate tasks is committed and the other is aborted.

Streaming

- Streaming runs special map and reduce tasks for the purpose of launching **the user supplied executable and communicating with it.**
- The Streaming task communicates with the process (**which may be written in any language)** using **standard input and output streams**.
- During execution of the task, the Java process passes input key-value pairs to the external process, which runs it through the user-defined map or reduce function and passes the output key-value pairs back to the Java process. From the node manager's point of view, it is as if the child process ran the map or reduce code itself.

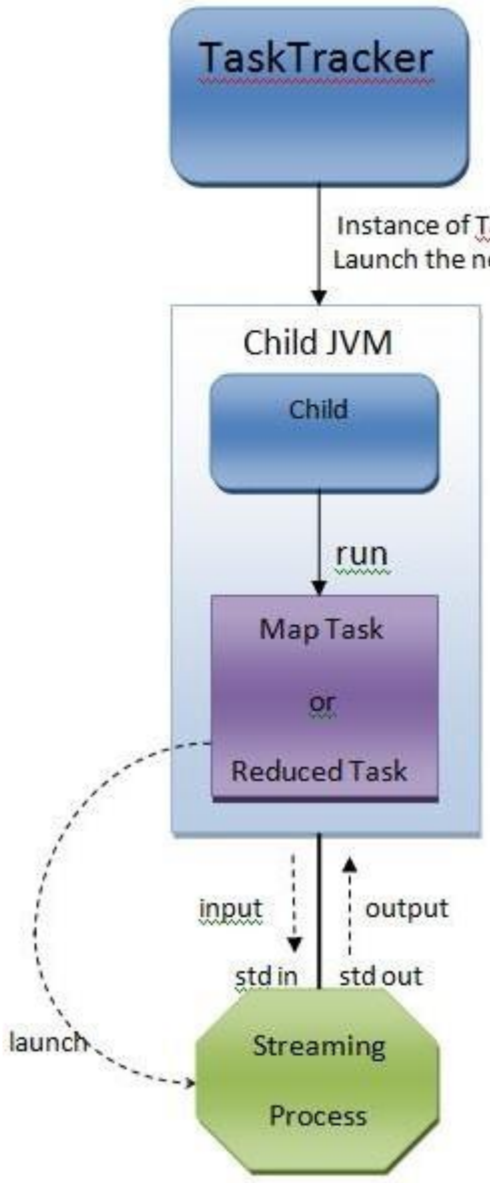


Diagram - 2

Progress and Status Updates

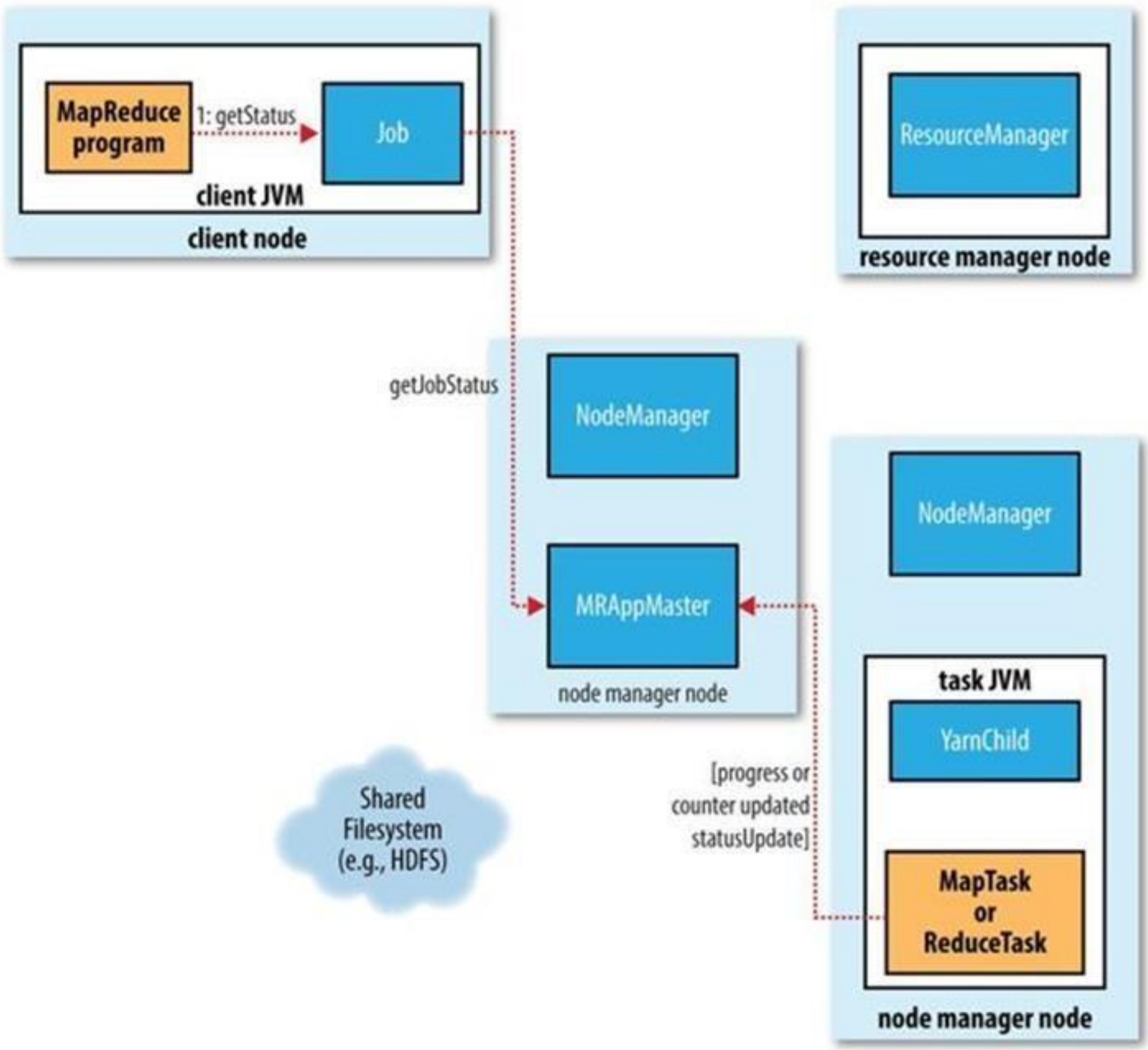
- MapReduce jobs are long-running batch jobs, taking anything from tens of seconds to hours to run. Because this can be a significant length of time, it's important for the user to get feedback on how the job is progressing.
- A job and each of its tasks have a *status*, which includes such things as the state of the job or task (e.g., running, successfully completed, failed), the progress of maps and reduces, the values of the job's counters, and a status message or description (which may be set by user code).
- These statuses change over the course of the job, so how do they get communicated back to the client?

- When a task is running, it keeps track of its *progress* (i.e., the proportion of the task completed).
- For map tasks, this is the proportion of the input that has been processed. For reduce tasks, it’s a little more complex, but the system can still estimate the proportion of the reduce input processed.
- It does this by dividing the total progress into three parts, corresponding to the three phases of the shuffle (see Shuffle and Sort).

WHAT CONSTITUTES PROGRESS IN MAPREDUCE?

- Progress is not always measurable, but nevertheless, it tells Hadoop that a task is doing something.
- For example, a task writing output records is making progress, even when it cannot be expressed as a percentage of the total number that will be written (because the latter figure may not be known, even by the task producing the output).
- Progress reporting is important, as Hadoop will not fail a task that’s making progress. All of the following operations constitute progress:

1. Reading an input record (in a mapper or reducer)
2. Writing an output record (in a mapper or reducer)
3. Setting the status description (via Reporter’s or TaskAttemptContext’s setStatus() method)
4. Incrementing a counter (using Reporter’s incrCounter() method or Counter’s increment() method)
5. Calling Reporter’s or TaskAttemptContext’s progress() method



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.” Then, when the Job polls for status, it learns that the job has completed successfully, so it prints a message to tell the user and then returns from the waitForCompletion() method. Job statistics and counters are printed to the console at this point.
- The application master also sends an HTTP job notification if it is configured to do so.
- This can be configured by clients wishing to receive callbacks, via the mapreduce.job.end-notification.url property.
- Finally, on job completion, the application master and the task containers clean up their working state (so intermediate output is deleted), and the OutputCommitter’s commitJob() method is called. Job information is archived by the job history server to enable later interrogation by users if desired.