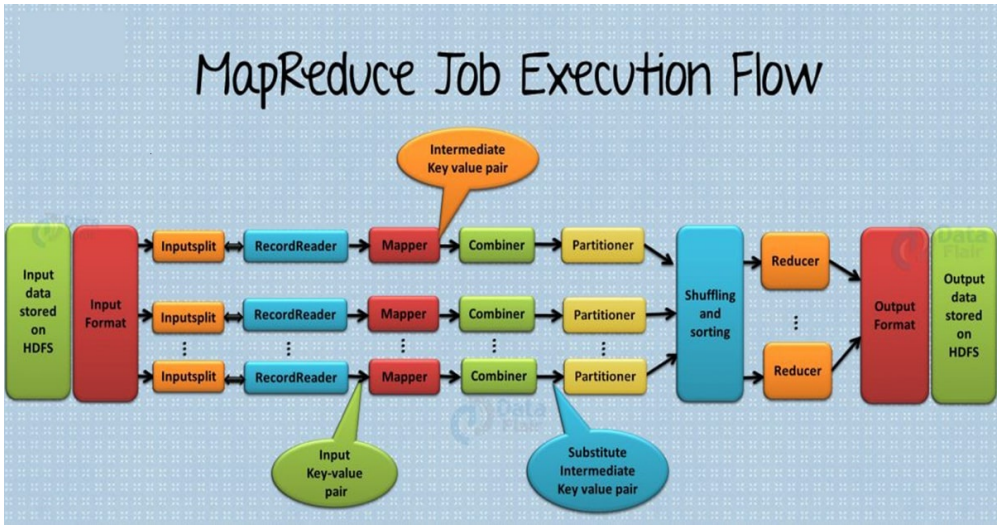
Large Scale Data Processing CSE3025

Dr. Ramesh Ragala

April 15, 2021

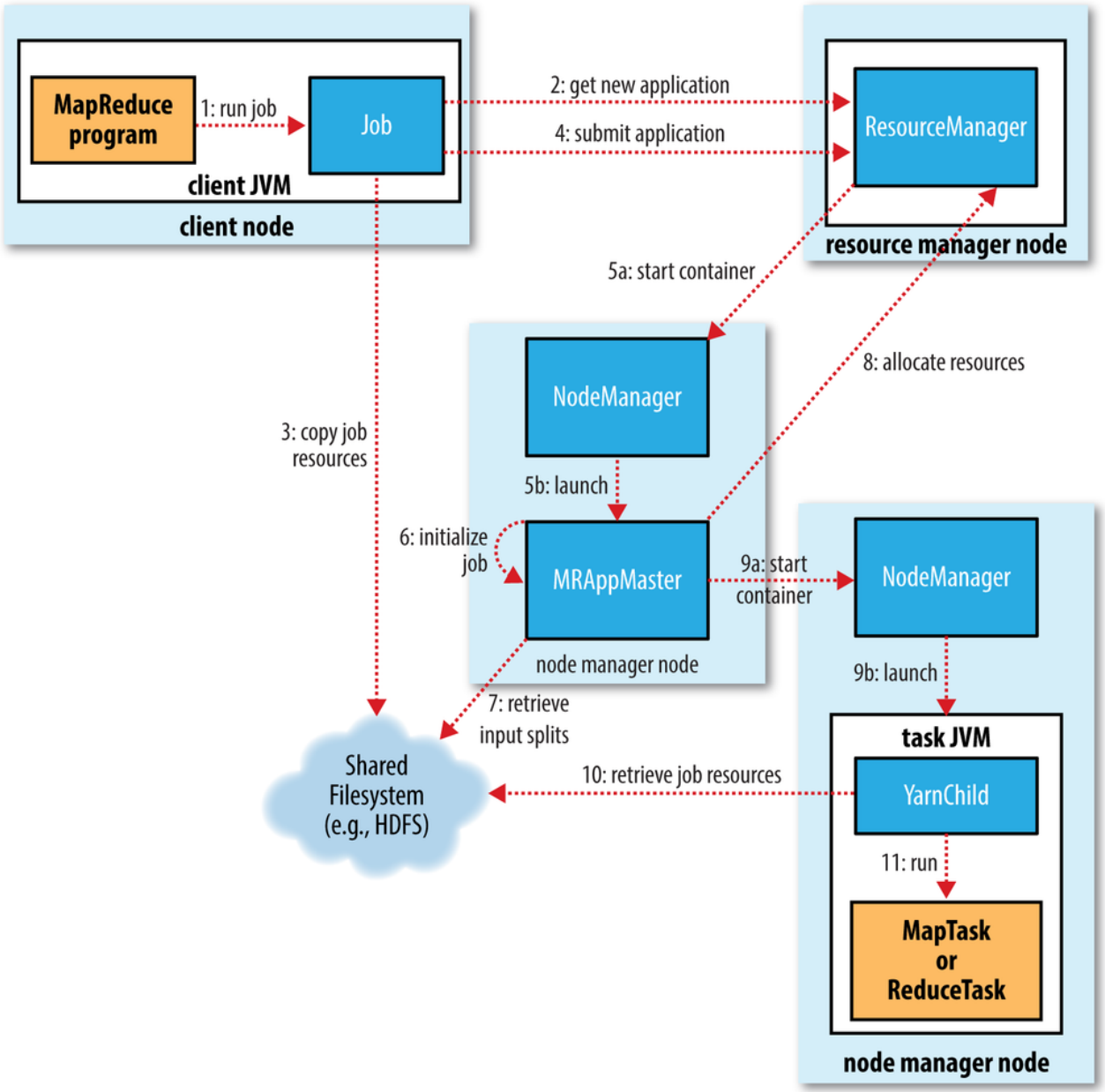
- In Hadoop-2 MapReduce is batch processing framework implemented on top of Yarn.
- To understand how MapReduce work in Hadoop you have to know how MapReduce job is running.
- As we know the important parts of MapReduce framework and how work:



Anatomy of MapReduce

- Client calls the submit() (or waitForCompletion()) method on Job. → submits the job.
- At the highest level, there are five independent entities for this
  - The client, which submits the MapReduce job.
  - The YARN resource manager, which coordinates the allocation of compute resources on the cluster.
  - The YARN node managers, which launch and monitor the compute containers on machines in the cluster.
  - 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.
  - The distributed filesystem (normally HDFS), which is used for sharing job files between the other entities.

Anatomy of MapReduce Job



- **Job Submission**
  - The submit() method on Job creates an internal JobSubmitter instance and calls **submitJobInternal()** on it
  - **waitForCompletion()** polls the job's progress once per second and reports the progress to the console if it has changed since the last report if job is submitted already.
  - The job **counters** are displayed on the console once the job is completed successfully.
  - If the job is not completed successfully, the error logs will (that specifies the causes to job fail) on the console.
  - The job submission process implemented by **JobSubmitter** does the following:
    - Asks the resource manager for a new application ID, used for the MapReduce job ID (step 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.

- **Job Submission**
  - The job submission process implemented by **JobSubmitter** does the following:
    - 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.
    - Copies the resources needed to run the job, including the job JAR file, the configuration file, and the computed input splits, to the shared filesystem in a directory named after the job ID (step 3). The job JAR is copied with a high replication factor, so that there are lots of copies across the cluster for the node managers to access when they run tasks for the job.
    - 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. $\rightarrow$ 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 determined by the mapreduce.job.reduces property (set by the setNumReduceTasks() method on Job). Tasks are given IDs at this point.
  - If the job is small, the application master may choose to run the tasks in the same JVM as itself.

- **Job Initialization**
  - 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**.
  - **On what basis we can decide the job can be 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.
  - 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

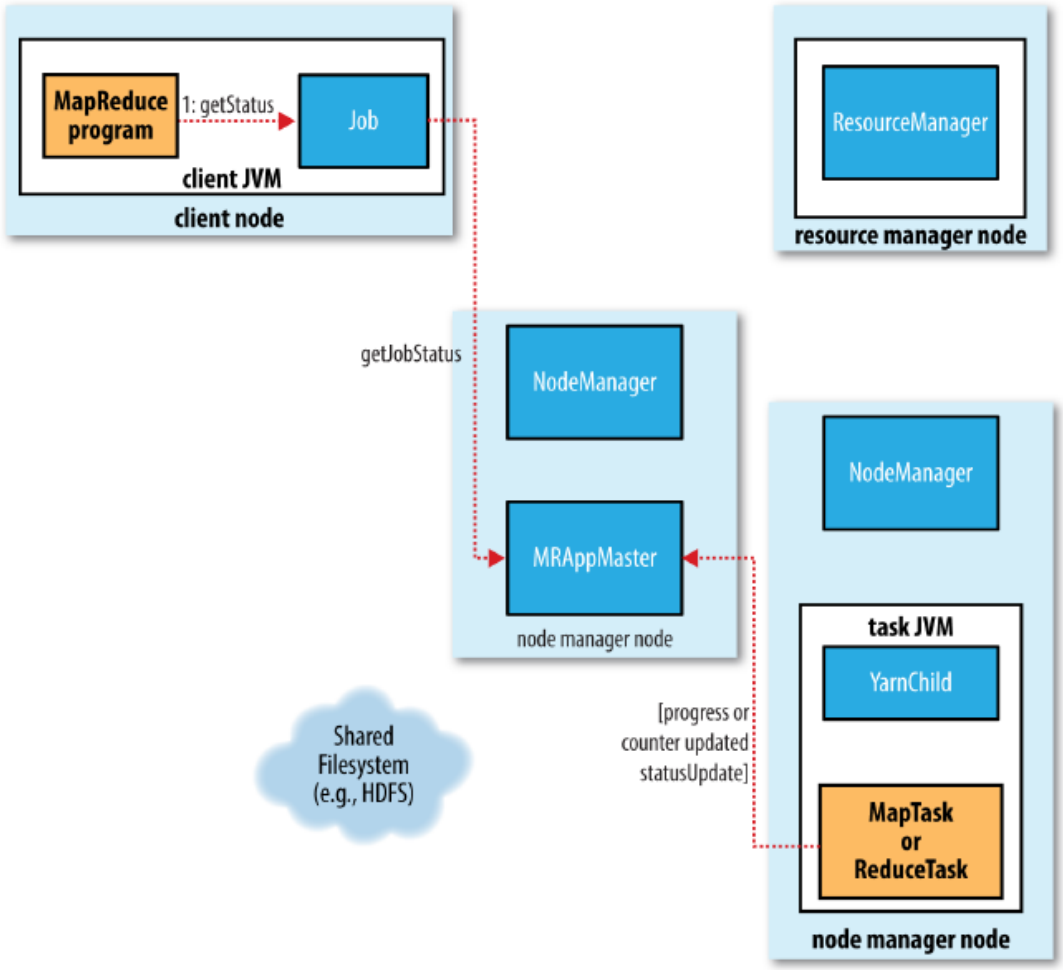
- **Task Assignment**
  - If the job does not qualify for running as an uber task, then the application master requests containers for all the map and reduce tasks in the job from the resource manager (step 8).
  - Requests for map tasks are made first and with a higher priority than those for reduce tasks.
  - 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
  - **Data Local:** The task is data local $\rightarrow$ i.e, running on the same node that the split resides on
  - **Rack Local:** The task may be rack local $\rightarrow$ i.e, 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.
  - Requests also specify memory requirements and CPUs for tasks. $\rightarrow$ By default, each map and reduce task is allocated 1,024 MB of memory and one virtual core. $\rightarrow$ These values can be configured.

- **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
  - 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 $\rightarrow$ The commit protocol ensures that when speculative execution is enabled

- **Progress and Status Updates**
  - MapReduce jobs are long-running batch jobs, taking anything from tens of seconds to hours to run. → user should get the feedback on how job is progressing.
  - A job and each of its tasks have followings status, which includes
    - the state of the job or task → running, successfully completed and failed
    - the progress of maps and reduces
    - the values of the job's counters
    - 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 → the proportion of the task completed
  - For map tasks, this is the proportion of the input that has been processed.
  - For reduce tasks, the system can still estimate the proportion of the reduce input processed

- **Progress and Status Updates**
  - With the help of counter features in MapReduce, it can be known how many of map output records are written.
  - While running of the map or reduce task, the child process communicates with its parent application master through the umbilical interface.
  - The task reports its progress and status (including counters) back to its application master, which has an aggregate view of the job, every three seconds over the umbilical interface.
  - Clients can also use Job's getStatus() method to obtain a JobStatus instance, which contains all of the status information for the job.

- **Progress and Status Updates**



- **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.
  - Finally, on job completion, the application master and the task containers clean up their working state, and the OutputCommitter's commit Job() method is called.
  - Job information is archived by the job history server to enable later interrogation by users if desired.

Failure Handling in Hadoop

- **Failures:**
  - In real world scenario, user code is may be buggy → processes may crash, and machines may fail
  - The best part of the Hadoop is that, it can handle those failures smoothly and allows the jobs to be finish successfully.
  - The different cases where failures can be occurred:
    - The failure of task
    - The failure of Application Master
    - The failure of Node Manager
    - The failure of Resource Manager