Prof R. Madana Mohana

BIG DATA ANALYTICS

How MapReduce Works

Anatomy of a MapReduce Job Run

https://www.youtube.com/c/RASINENIMADANAMOHANA

Anatomy of a MapReduce Job Run: *Outline*

- Job Submission
- Job Initialization

- Task Assignment
- Task Execution

- Progress and Status Updates
- Job Completion

Anatomy of a MapReduce Job Run

Anatomy of a MapReduce Job Run

- We can run a **MapReduce** job with a single method call: **submit()** on a Job object (We can also call **waitForCompletion()**, which submits the job if it hasn't been submitted already, then waits for it to finish).
- The whole process is illustrated in *below Figure:*

Anatomy of a MapReduce Job Run

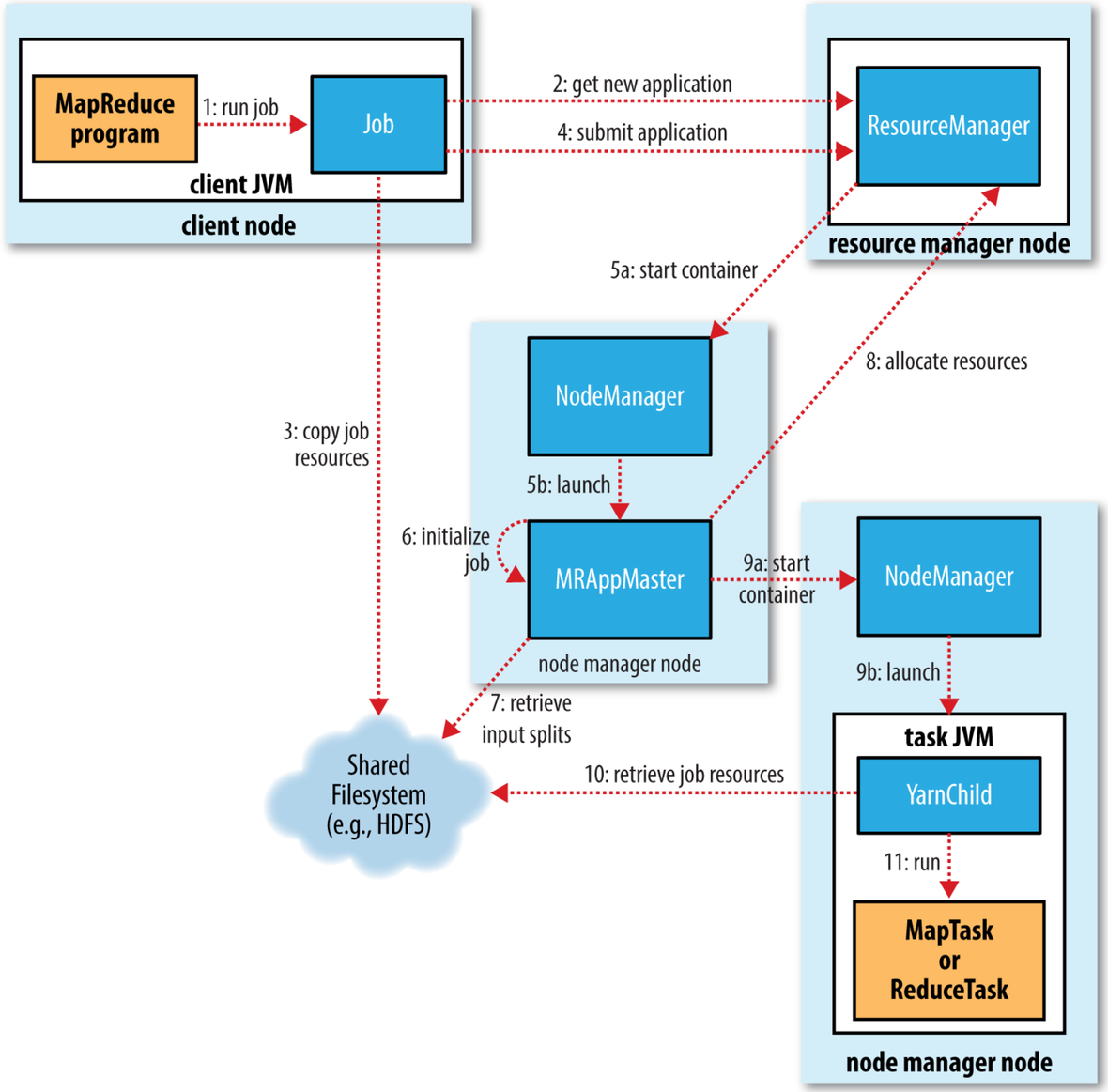


Figure. *The flow of data in a simple MapReduce job*

Anatomy of a MapReduce Job Run

At the highest level, there are *five independent* entities:

- The client, which submits the **MapReduce** job.
- The **YARN** (**Yet Another Resource Negotiator**) 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.

Anatomy of a MapReduce Job Run

At the highest level, there are *five independent* entities:

- The **MapReduce** application master, which coordinates the tasks running the **Map-Reduce** 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 a MapReduce Job Run

Job Submission:

- The **submit()** method on Job creates an internal **JobSubmitter** instance and calls **submitJobInternal()** on it (*step 1* in above Figure).
- Having submitted the job, **waitForCompletion()** polls the job's progress once per second and reports the progress to the console if it has changed since the last report.

Anatomy of a MapReduce Job Run

Job Submission:

- When the job *completes successfully*, the job counters are displayed. Otherwise, the error that caused the job to fail is logged to the console.

Anatomy of a MapReduce Job Run

Job Submission:

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* in above *Figure*).
- 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.

Anatomy of a MapReduce Job Run

Job Submission:

- 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.

Anatomy of a MapReduce Job Run

Job Submission:

- 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* in above *Fig.*). The job **JAR** is copied with a high replication factor (controlled by the **mapreduce.client.submit.file.replication** property, 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.

Anatomy of a MapReduce Job Run

Job Submission:

- Submits the job by calling **submitApplication()** on the resource manager (*step 4* in above *Fig.*).

Anatomy of a MapReduce Job Run

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* in above *Fig.*).

Anatomy of a MapReduce Job Run

Job Initialization:

- 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* in above *Fig.*).
- Next, it retrieves the input splits computed in the client from the shared filesystem (*step 7* in above *Fig*.). 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 **ID**s at this point.

Anatomy of a MapReduce Job Run

Job Initialization:

- 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***.

Anatomy of a MapReduce Job Run

Job Initialization:

- 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.

Anatomy of a MapReduce Job Run

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* in above *Fig.*).
- 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.

Anatomy of a MapReduce Job Run

Task Assignment:

- **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.

Anatomy of a MapReduce Job Run

Task Assignment:

- 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 **CPU**s for tasks. By default, each **map** and **reduce** task is allocated **1,024 MB** of memory and one virtual core. The values are configurable on a per-job basis via the following properties:

`mapreduce.map.memory.mb`, `mapreduce.reduce.memory.mb`, `mapreduce.map.cpu.vcores` and `mapreduce.reduce.cpu.vcores`

Anatomy of a MapReduce Job Run

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* in above *Fig.*).
- The task is executed by a **Java** application whose main class is **YarnChild**.

Anatomy of a MapReduce Job Run

Task Execution:

- 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* in above *Fig.*).
- Finally, it runs the **map** or **reduce** task (*step 11* in above *Fig.*).

Anatomy of a MapReduce Job Run

Task Execution:

- 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.

Anatomy of a MapReduce Job Run

Task Execution:

- 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 uncertain execution is enabled , only one of the duplicate tasks is committed and the other is aborted.

Anatomy of a MapReduce Job Run

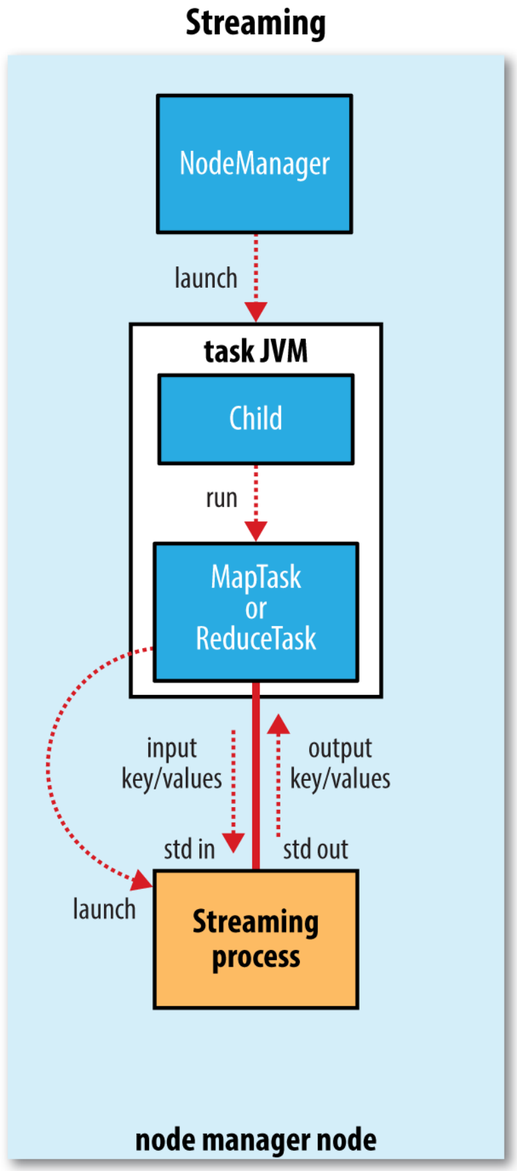
Task Execution: *Streaming*

- Streaming runs special **map** and **reduce** tasks for the purpose of launching the user-supplied executable and communicating with it (Shown in *below Figure*).

Anatomy of a MapReduce Job Run

Task Execution: *Streaming*

Figure. *The relationship of the Streaming executable to the node manager and the task container*



Anatomy of a MapReduce Job Run

- **Task Execution: *Streaming***
- 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.

Anatomy of a MapReduce Job Run

Task Execution: *Streaming*

- From the node manager's point of view, it is as if the child process ran the **map** or **reduce** code itself.

Anatomy of a MapReduce Job Run

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.

Anatomy of a MapReduce Job Run

Progress and Status Updates:

- 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?*

Anatomy of a MapReduce Job Run

Progress and Status Updates:

- 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.

Anatomy of a MapReduce Job Run

Progress and Status Updates:

- It does this by dividing the total progress into three parts, corresponding to the three phases of the shuffle.
- For example, 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**).

Anatomy of a MapReduce Job Run

Progress and Status Updates:

What Constitutes Progress in MapReduce?

Progress reporting is important, as **Hadoop** will not fail a task that's making progress. All of the *following operations* constitute progress:

- Reading an input record (in a **mapper** or **reducer**)
- Writing an output record (in a **mapper** or **reducer**)
- Setting the status description (via **Reporter's** or **TaskAttemptContext's setStatus()** method)

Anatomy of a MapReduce Job Run

Progress and Status Updates:

What Constitutes Progress in MapReduce?

- Incrementing a counter (using **Reporter**'s **incrCounter()** method or **Counter**'s **increment()** method).
- Calling **Reporter**'s or **TaskAttemptContext**'s **progress()** method.

Anatomy of a MapReduce Job Run

Progress and Status Updates:

How status updates are propagated through the **MapReduce** system is shown in *below Figure*:

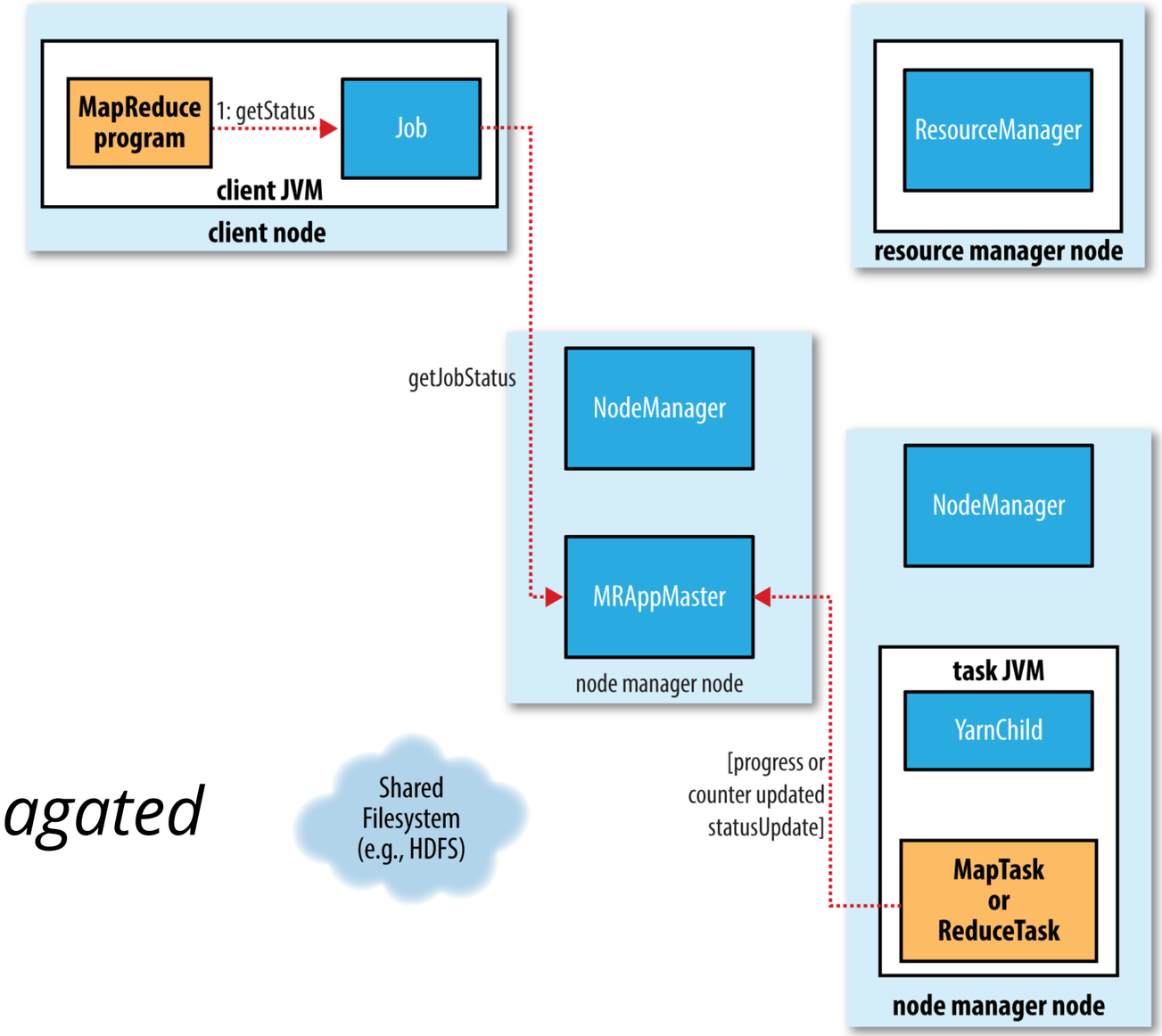


Figure. *How status updates are propagated through the MapReduce system*

Anatomy of a MapReduce Job Run

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.

Anatomy of a MapReduce Job Run

Job Completion:

- 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.

Anatomy of a MapReduce Job Run

Job Completion:

- 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 **commit Job()** method is called. Job information is archived by the job history server to enable later interrogation by users if desired.