Putting it Together

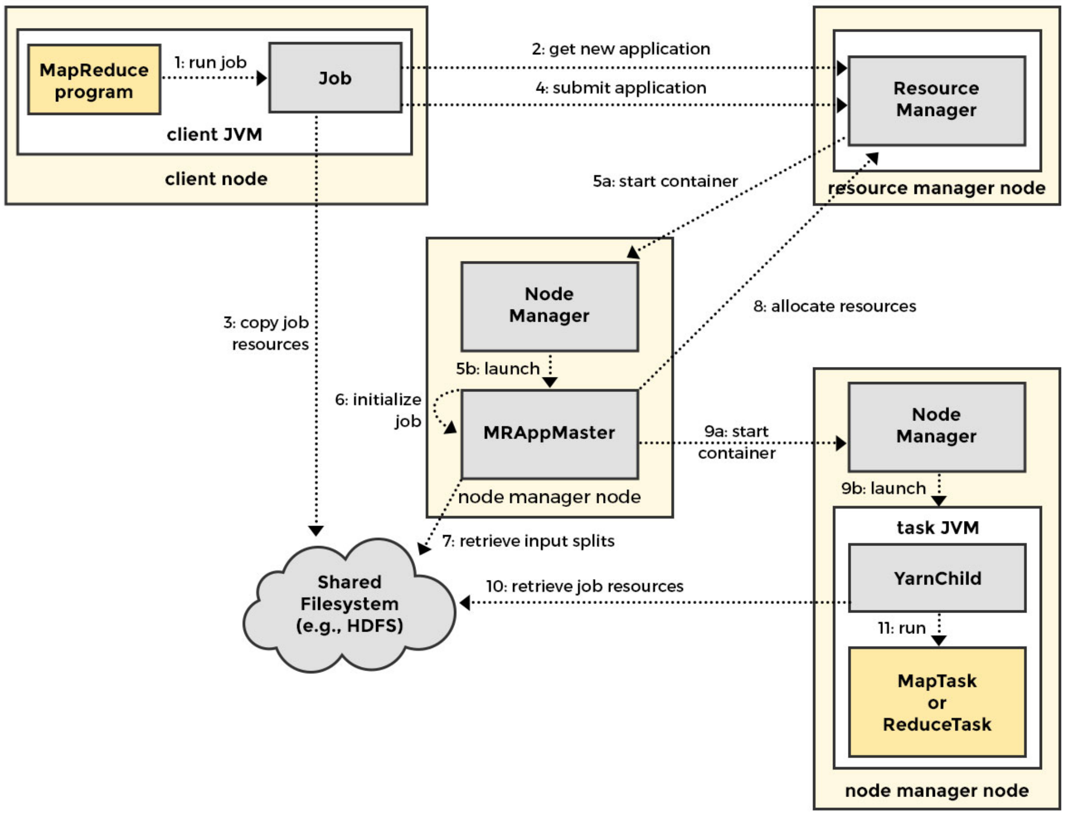
This lesson explains end to end working of MapReduce.

We'll cover the following

- Putting it Together

Putting it Together

Now that we know how MapReduce works, we can now dive into the end to end workflow of a MapReduce job in a Hadoop cluster.



How Hadoop runs a MapReduce job

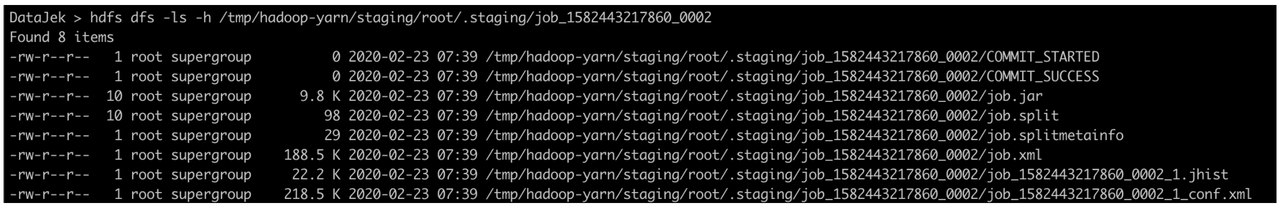
1. In the driver program of our example, you'll see the job is submitte using the method `waitForCompletion()`.

```
job.waitForCompletion(true);
```

This method returns when the job has successfully completed. A lot goes on behind the scenes before this method returns. We'll trace the various steps involved in the execution of a job when submitted to a Hadoop cluster.

2. The class `JobSubmitter` is responsible for talking to the resource manager and retrieving a new application ID used as the ID of the MR job. The class also performs sanity checks, such as verifying if the output path exists and the input splits can be successfully computed.
3. Next, the resources for running the job are copied over to HDFS in a staging directory, with the ID of the job in the path. Resources include the jar file, which holds the mapper and reducer code to execute. This file is renamed as **job.jar**. Configuration files and metadata about input splits is also copied. After the job successfully finishes, the framework deletes this staging directory. You can set the property `mapreduce.task.files.preserve.filepattern` to choose what files to keep for debug purposes.

   The jar file is replicated across the cluster to be readily available for node managers to access in order to run tasks.



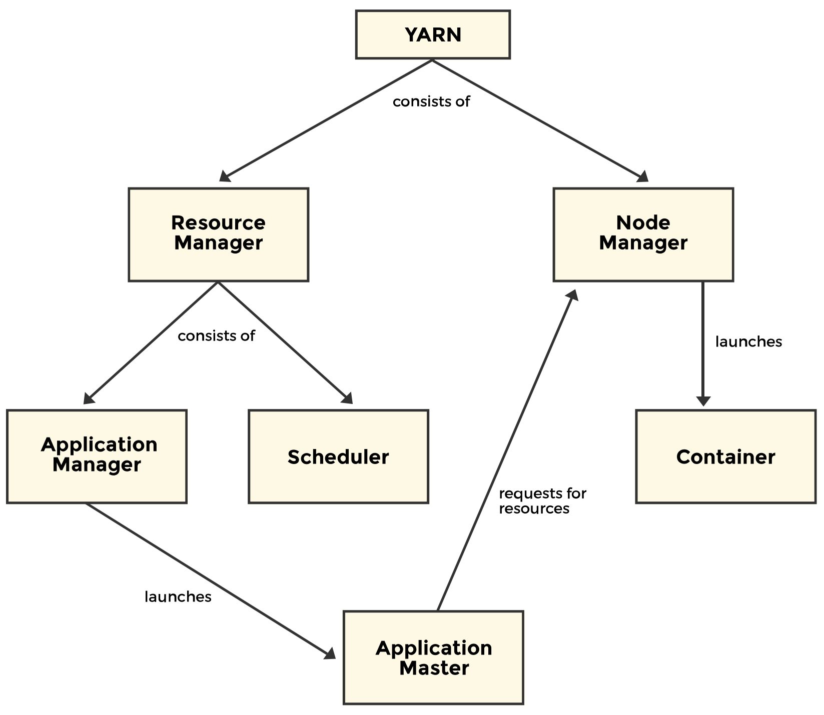
```
DataJek > hdfs dfs -ls -h /tmp/hadoop-yarn/staging/root/.staging/job_1582443217860_0002
Found 8 items
-rw-r--r--   1 root supergroup          0 2020-02-23 07:39 /tmp/hadoop-yarn/staging/root/.staging/job_1582443217860_0002/COMMIT_STARTED
-rw-r--r--   1 root supergroup          0 2020-02-23 07:39 /tmp/hadoop-yarn/staging/root/.staging/job_1582443217860_0002/COMMIT_SUCCESS
-rw-r--r--  10 root supergroup      9.8 K 2020-02-23 07:39 /tmp/hadoop-yarn/staging/root/.staging/job_1582443217860_0002/job.jar
-rw-r--r--  10 root supergroup         98 2020-02-23 07:39 /tmp/hadoop-yarn/staging/root/.staging/job_1582443217860_0002/job.split
-rw-r--r--   1 root supergroup         29 2020-02-23 07:39 /tmp/hadoop-yarn/staging/root/.staging/job_1582443217860_0002/job.splitmetainfo
-rw-r--r--   1 root supergroup    188.5 K 2020-02-23 07:39 /tmp/hadoop-yarn/staging/root/.staging/job_1582443217860_0002/job.xml
-rw-r--r--   1 root supergroup     22.2 K 2020-02-23 07:39 /tmp/hadoop-yarn/staging/root/.staging/job_1582443217860_0002/job_1582443217860_0002_1.jhist
-rw-r--r--   1 root supergroup    218.5 K 2020-02-23 07:39 /tmp/hadoop-yarn/staging/root/.staging/job_1582443217860_0002/job_1582443217860_0002_1_conf.xml
```

4. The job is submitted for execution to the resource manager by invoking the method `submitApplication()` on the resource manager.
5. The resource manager, upon receiving the `submitApplication()` call,

   and an application master process, represented by the class `MRAppMaster`, is launched in the container. The node manager manages the launching of the master process.

To understand the YARN and MapReduce framework[illegible]s together, we'll take a brief detour and explain the topology of the various entities.

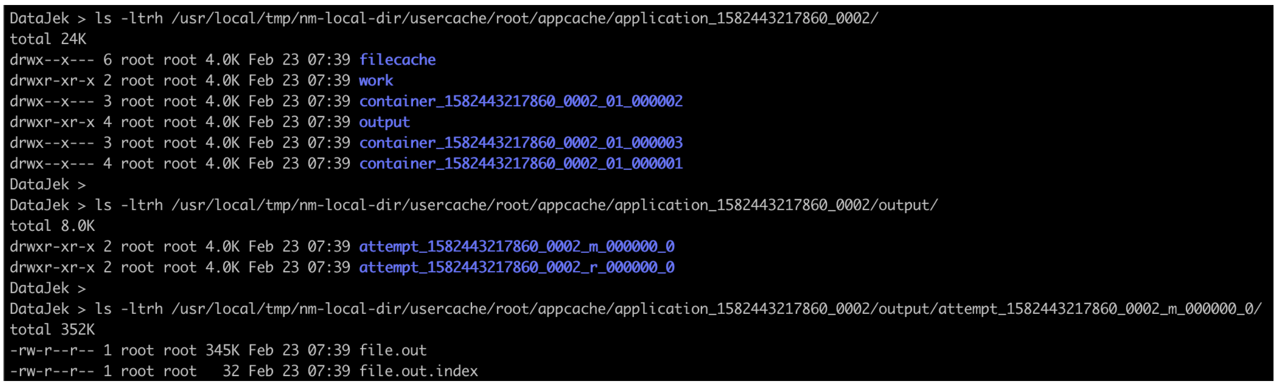
Once the ResourceManager is submitted an application, a container is created for the ApplicationMaster. An ApplicationMaster is an entity responsible for managing the lifecycle of a job. Once instantiated, it talks to the ResourceManager to negotiate resources and then works with the node managers to execute and monitor the tasks. In the case of MapReduce, the ApplicationMaster is the MRAppMaster. But in general, a framework-specific library negotiates resources from the ResourceManager and works with the NodeManagers.



6. The created ApplicationMaster initializes a number of book-keeping objects needed that tracks the progress of the map and reduce tasks that will be shortly created. The computed input splits are retrieved, and a map task object is created for each split. The number of reduce tasks is controlled by the property `mapreduce.job.reduces`. Tasks are given IDs at this point. Note that this is just a programmatic representation of the map and reduce tasks, the execution of tasks [illegible] yet to be scheduled.

7. The ApplicationMaster then needs to decide to run th p reduce tasks in the same JVM as itself or request containers so the tasks can run in parallel. Sometimes the ApplicationMaster runs the tasks in the same JVM. A small job may finish sooner when map and reduce tasks run sequentially in the same JVM versus when they are run in separate containers on different nodes. The overhead of requesting containers and managing their execution may be greater than the running time of the job. If a job is executed in the same JVM as the ApplicationMaster, it is **uberized** and the tasks are referred to as uber tasks. There are various knobs exposed by the mapreduce framework to tweak when a job may get scheduled as an uber job.

   The following screen-shot captures the temporary files produced by the mapper on the local filesystem:

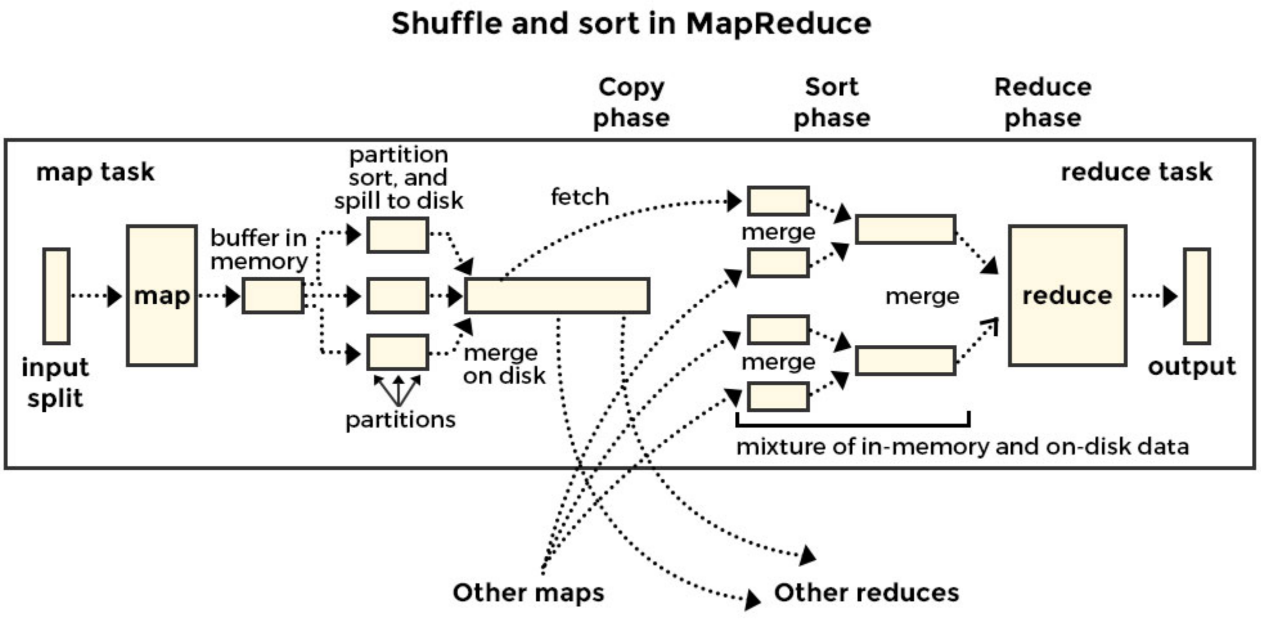


```
DataJek > ls -ltrh /usr/local/tmp/nm-local-dir/usercache/root/appcache/application_1582443217860_0002/
total 24K
drwx--x--- 6 root root 4.0K Feb 23 07:39 filecache
drwxr-xr-x 2 root root 4.0K Feb 23 07:39 work
drwx--x--- 3 root root 4.0K Feb 23 07:39 container_1582443217860_0002_01_000002
drwxr-xr-x 4 root root 4.0K Feb 23 07:39 output
drwx--x--- 3 root root 4.0K Feb 23 07:39 container_1582443217860_0002_01_000003
drwx--x--- 4 root root 4.0K Feb 23 07:39 container_1582443217860_0002_01_000001
DataJek >
DataJek > ls -ltrh /usr/local/tmp/nm-local-dir/usercache/root/appcache/application_1582443217860_0002/output/
total 8.0K
drwxr-xr-x 2 root root 4.0K Feb 23 07:39 attempt_1582443217860_0002_m_000000_0
drwxr-xr-x 2 root root 4.0K Feb 23 07:39 attempt_1582443217860_0002_r_000000_0
DataJek >
DataJek > ls -ltrh /usr/local/tmp/nm-local-dir/usercache/root/appcache/application_1582443217860_0002/output/attempt_1582443217860_0002_m_000000_0/
total 352K
-rw-r--r-- 1 root root 345K Feb 23 07:39 file.out
-rw-r--r-- 1 root root   32 Feb 23 07:39 file.out.index
```

8. If a job can't run as an uber job, then the ApplicationMaster requests containers for all the map and reduce tasks from the resource manager. Memory and CPU requirements can be specified when making these requests and are configurable on a per job basis. Requests for reduce tasks are not made by the ApplicationMaster until 5% of the map tasks have finished.

   Reduce tasks can be run anywhere on the cluster. In contrast, the map tasks are constrained by data locality requirements, which the Scheduler tries to satisfy. Ideally, a map task should run on the same node which hosts that task's input split. The next preferred option is to run the map task on a node in the same rack that hosts the input split. And the least preferred option is to run the map [illegible] rack node.

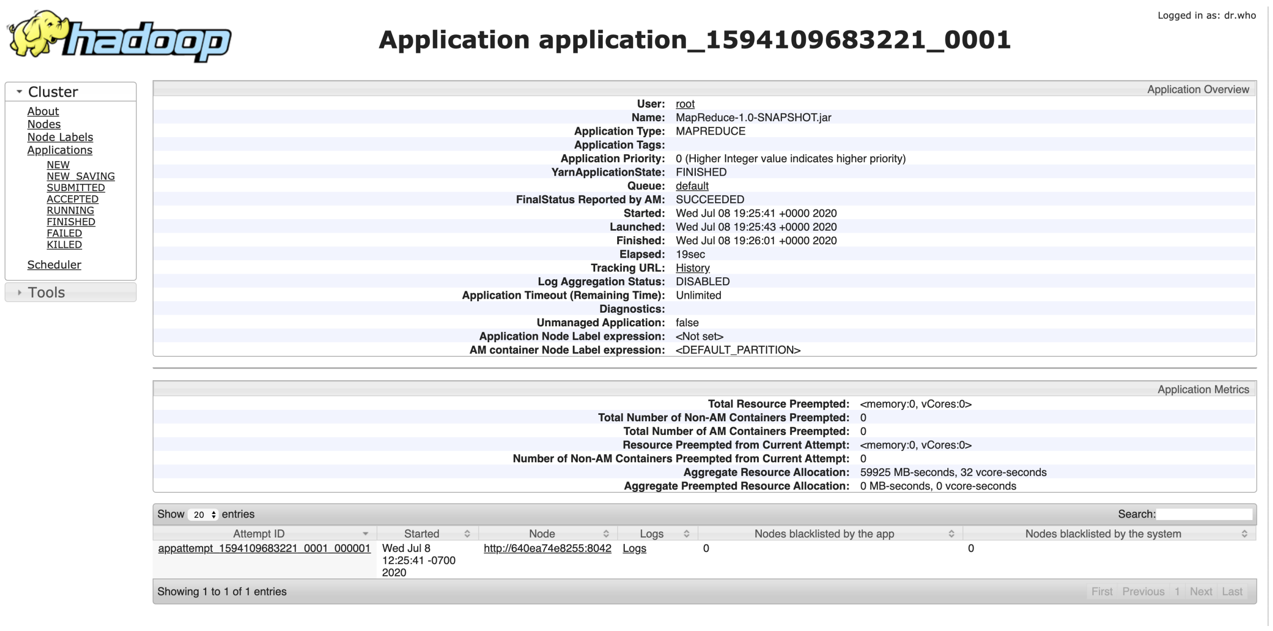
9. Once ResourceManager's Scheduler has aptly scheduled on a particular node, the ApplicationMaster contacts the node manager to start the container. The needed resources for the task, such as job configuration, jar file, and any other files in the distributed cache, are localized. A Java application whose main class is `YarnChild`executes the task. The `YarnChild` process runs with the map or reduce tasks in a dedicated JVM. Then, in the case of any failures or crashes, the Node Manager process is not affected which itself runs in a different JVM.



10. So far, we only worked with file-based outputs. However, the framework provides the ability to use a different output format, like writing to S3 or in Parquet format on disk. The `OutputFormat` class determines the format used. This class has a method `getOutputCommitter()` that can be overridden to return an implementation of a custom derived instance of the class `OutputCommitter`. The object returned from this method determines the output. In our code, we used the class `FileOutputFormat` which is a derived class of `OutputFormat`, returns an object of class `PathOutputCommitter` which is, in turn, derived from `OutputCommitter`.

A custom output committer can override the method s cla
to add actions to execute for task, as well as, job setup and cleanup.

11. The map and reduce tasks report their progress to the ApplicationMaster. The client program can poll the ApplicationMaster for various counters and status. The RM web UI displays all the running applications with links to their ApplicationMasters, each of which displays further granular information on the job and its progress.



Logged in as: dr.who

Application application_1594109683221_0001

Cluster
- About
- Nodes
- Node Labels
- Applications
  - NEW
  - NEW_SAVING
  - SUBMITTED
  - ACCEPTED
  - RUNNING
  - FINISHED
  - FAILED
  - KILLED
- Scheduler

Tools

Application Overview

User:	root
Name:	MapReduce-1.0-SNAPSHOT.jar
Application Type:	MAPREDUCE
Application Tags:	
Application Priority:	0 (Higher Integer value indicates higher priority)
YarnApplicationState:	FINISHED
Queue:	default
FinalStatus Reported by AM:	SUCCEEDED
Started:	Wed Jul 08 19:25:41 +0000 2020
Launched:	Wed Jul 08 19:25:43 +0000 2020
Finished:	Wed Jul 08 19:26:01 +0000 2020
Elapsed:	19sec
Tracking URL:	History
Log Aggregation Status:	DISABLED
Application Timeout (Remaining Time):	Unlimited
Diagnostics:	
Unmanaged Application:	false
Application Node Label expression:	<Not set>
AM container Node Label expression:	<DEFAULT_PARTITION>

Application Metrics

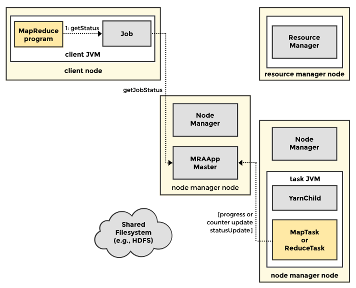
Total Resource Preempted:	<memory:0, vCores:0>
Total Number of Non-AM Containers Preempted:	0
Total Number of AM Containers Preempted:	0
Resource Preempted from Current Attempt:	<memory:0, vCores:0>
Number of Non-AM Containers Preempted from Current Attempt:	0
Aggregate Resource Allocation:	59925 MB-seconds, 32 vcore-seconds
Aggregate Preempted Resource Allocation:	0 MB-seconds, 0 vcore-seconds

Show 20 entries Search:

Attempt ID	Started	Node	Logs	Nodes blacklisted by the app	Nodes blacklisted by the system
appattempt_1594109683221_0001_000001	Wed Jul 8 12:25:41 -0700 2020	http://640ea74e8255:8042	Logs	0	0

Showing 1 to 1 of 1 entries First Previous 1 Next Last

The client can receive the status of a job by polling it periodically. This interval can be set using the property `mapreduce.client.progressmonitor.pollinterval`. The `Job` has a method `getStatus()` that can retrieve an instance of the class `JobInstance` that has all the status information for a job.



How status updates are propagated through the MapReduce system

12. Once the ApplicationMaster is notified of the last task completing, t
ApplicationMaster marks the job status as successful. The

`waitForCompletion()` method on the client side returns is n counters for the job are printed on the console. The ApplicationMaster can be configured to invoke a callback URL once the job completes, using the `mapreduce.job.end-notification.url` property.

The application master and task containers also clean up their state and delete any intermediate output once the job completes.

Back
Combiner and Partitioner

Next
Resiliency

✓ Mark as Completed

Report an Issue