Hadoop 2.2 brought about architectural changes to MapReduce. As Hadoop has matured, people have found that it can be used for more than running MapReduce jobs. But to keep each new framework from having its own resource manager and scheduler, that would compete with the other framework resource managers and schedulers, it was decided to have the recourse manager and schedulers to be external to any framework. This new architecture is called YARN. (Yet Another Resource Negotiator) . You still have DataNodes but there are no longer TaskTrackers and the JobTracker.

You are not required to run YARN with Hadoop 2.2. MapReduce V1 is still supported.

There are two main ideas with YARN.

Provide generic scheduling and resource management. This way Hadoop can support more than just MapReduce.

The other is to try to provide a more efficient scheduling and workload management. With MapReduce V1, the administrator had to define how many map slots and how many reduce slots there were on each node. Since the hardware capabilities for each node in a Hadoop cluster can vary, for performance reasons, you might want to limit the number of tasks on certain nodes. With YARN, this is no longer required.

With YARN, the resource manager is aware of the capabilities of each node via communication with the NodeManager running on each node. When an application gets invoked , an Application Master gets started. The Application Master is then responsible for negotiating resourses from the ResourceManager. These resources are assigned to Containers on each slave-node and you can think that tasks then run in Containers. With this architecture, you are no long forced into a one size fits all.

The NameNode is a single point of failure. Is there anything that can be done about that? Hadoop now supports high availability. In this setup, there are now two NameNodes, one active and one standby.

Also, now there are JournalNodes. There must be at least three and there must be an odd number. Only one of the NameNodes can be active at a time. It is the JournalNodes, working together , that decide which of the NameNodes is to be the active one and if the active NameNode has been lost and whether the backup NameNode should take over.

The NameNode loads the metadata for the file system into memory. This is the reason that we said that NameNodes needed large amounts of RAM. But you are going to be limited at some point when you use this vertical growth model. Hadoop Federation allows you to grow your system horizontally. This setup also utilizes multiple NameNodes. But they act independently. However, they do all share all of the DataNodes. Each NameNode has its own namespace and therefore has control over its own set of files. For example, one file that has blocks on DataNode 1 and DataNode 2 might be owned by NameNode 1. NameNode 2 might own a file that has blocks on DataNode 2 and DataNode 3. And NameNode 3 might have a file with blocks on all three DataNodes.

Hadoop has awareness of the topology of the network. This allows it to optimize where it sends the computations to be applied to the data. Placing the work as close as possible to the data it operates on maximizes the bandwidth available for reading the data. In the diagram, the data we wish to apply processing to is block B1, the dark blue rectangle on node n1 on rack 1.

When deciding which TaskTracker should receive a MapTask that reads data from B1, the best option is to choose the TaskTracker that runs on the same node as the data.

If we can't place the computation on the same node, our next best option is to place it on a node in the same rack as the data.

The worst case that Hadoop currently supports is when the computation must be processed from a node in a different rack than the data. When rack-awareness is configured for your cluster, Hadoop will always try to run the task on the TaskTracker node with the highest bandwidth access to the data.

Let us walk through an example of how a file gets written to HDFS.

First, the client submits a "create" request to the NameNode. The NameNode checks that the file does not already exist and the client has permission to write the file.

If that succeeds, the NameNode determines the DataNode to where the first block is to be written.

If the client is running on a DataNode, it will try to place it there. Otherwise, it chooses DataNode at random.

By default, data is replicated to two other places in the cluster. A pipeline is built between the three DataNodes that make up the pipeline. The second DataNode is a randomly chosen node on a rack other than that of the first replica of the block. This is to increase redundancy.

The final replica is placed on a random node within the same rack as the second replica. The data is piped from the second DataNode to the third.

To ensure the write was successful before continuing, acknowledgment packets are sent from the third DataNode to the second,

From the second DataNode to the first

And from the first DataNode to the client

This process occurs for each of the blocks that makes up the file

Notice that, for every block, by default, there is a replica on at least two racks.

When the client is done writing to the DataNode pipeline and has received acknowledgements, it tells the NameNode that the write has completed. The NameNode then checks that the blocks are at least minimally replicated before responding.

This lesson continues with the next video in this unit.