Hadoop in Practice

SECOND EDITION

Alex Holmes

INCLUDES 104 TECHNIQUES

SAMPLE CHAPTER

MANNING

Hadoop in Practice
Second Edition

by Alex Holmes

Chapter 2

brief contents

2 Introduction to YARN

This chapter covers

- Understanding how YARN works
- How MapReduce works as a YARN application
- A look at other YARN applications

Imagine buying your first car, which upon delivery has a steering wheel that doesn't function and brakes that don't work. Oh, and it only drives in first gear. No speeding on winding back roads for you! That empty, sad feeling is familiar to those of us who want to run some cool new tech such as graph or real-time data processing with Hadoop 1,[1] only to be reminded that our powerful Hadoop clusters were good for one thing, and one thing only: MapReduce.

Luckily for us the Hadoop committers took these and other constraints to heart and dreamt up a vision that would metamorphose Hadoop above and beyond MapReduce. YARN is the realization of this dream, and it's an exciting new development that transitions Hadoop into a distributed computing kernel that can support any type of workload.[2] This opens up the types of applications that can be run on

[1] While you can do graph processing in Hadoop 1, it's not a native fit, which means you're either incurring the inefficiencies of multiple disk barriers between each iteration on your graph, or hacking around in MapReduce to avoid such barriers.

Hadoop to efficiently support computing models for machine learning, graph processing, and other generalized computing projects (such as Tez), which are discussed later in this chapter

The upshot of all this is that you can now run MapReduce, Storm, and HBase all on a single Hadoop cluster. This allows for exciting new possibilities, not only in computational multi-tenancy, but also in the ability to efficiently share data between applications.

Because YARN is a new technology, we'll kick off this chapter with a look at how YARN works, followed by a section that covers how to interact with YARN from the command line and the UI. Combined, these sections will give you a good grasp of what YARN is and how to use it.

Once you have a good handle on how YARN works, you'll see how MapReduce has been rewritten to be a YARN application (titled MapReduce 2, or MRv2), and look at some of the architectural and systems changes that occurred in MapReduce to make this happen. This will help you better understand how to work with MapReduce in Hadoop 2 and give you some background into why some aspects of MapReduce changed in version 2.

> **YARN development** If you're looking for details on how to write YARN applications, feel free to skip to chapter 10. But if you're new to YARN, I recommend you read this chapter before you move on to chapter 10.

In the final section of this chapter, you'll examine several YARN applications and their practical uses.

Let's get things started with an overview of YARN.

2.1 YARN overview

With Hadoop 1 and older versions, you were limited to only running MapReduce jobs. This was great if the type of work you were performing fit well into the MapReduce processing model, but it was restrictive for those wanting to perform graph processing, iterative computing, or any other type of work.

In Hadoop 2 the scheduling pieces of MapReduce were externalized and reworked into a new component called YARN, which is short for *Yet Another Resource Negotiator*. YARN is agnostic to the type of work you do on Hadoop—all that it requires is that applications that wish to operate on Hadoop are implemented as YARN applications. As a result, MapReduce is now a YARN application. The old and new Hadoop stacks can be seen in figure 2.1.

There are multiple benefits to this architectural change, which you'll examine in the next section.

[2] Prior to YARN, Hadoop only supported MapReduce for computational work.

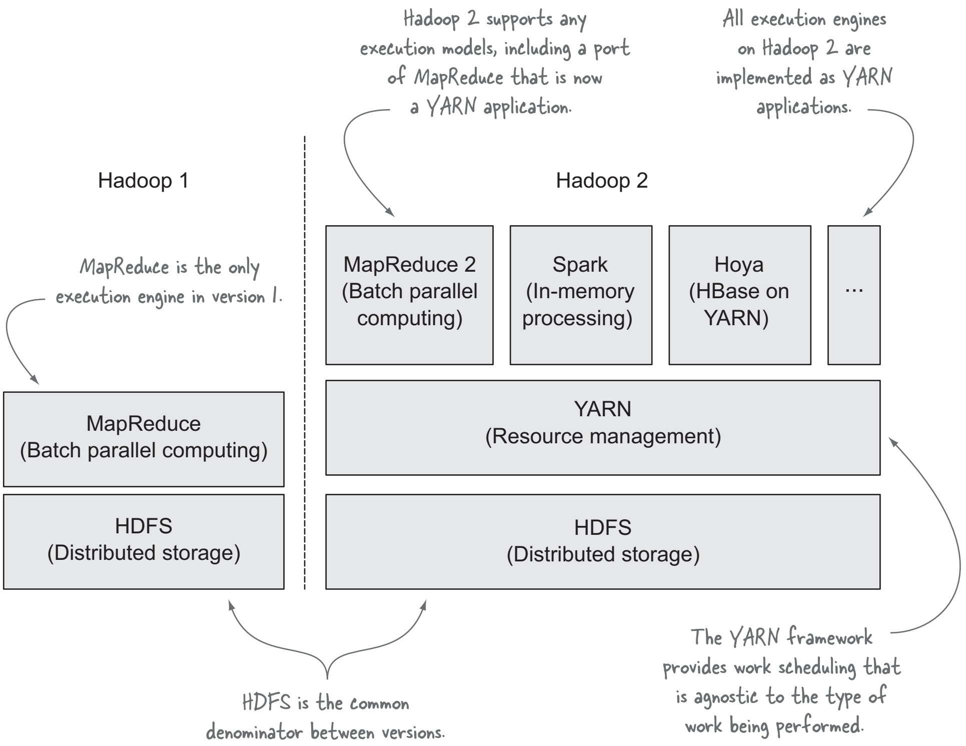

Hadoop 2 supports any execution models, including a port of MapReduce that is now a YARN application.
All execution engines on Hadoop 2 are implemented as YARN applications.
Hadoop 1
Hadoop 2
MapReduce is the only execution engine in version 1.
MapReduce 2 (Batch parallel computing)
Spark (In-memory processing)
Hoya (HBase on YARN)
...
YARN (Resource management)
MapReduce (Batch parallel computing)
HDFS (Distributed storage)
HDFS (Distributed storage)
HDFS is the common denominator between versions.
The YARN framework provides work scheduling that is agnostic to the type of work being performed.

Figure 2.1 Hadoop 1 and 2 architectures, showing YARN as a generalized scheduler and various YARN applications

2.1.1 Why YARN?

We've touched on how YARN enables work other than MapReduce to be performed on Hadoop, but let's expand on that and also look at additional advantages that YARN brings to the table.

MapReduce is a powerful distributed framework and programming model that allows batch-based parallelized work to be performed on a cluster of multiple nodes. Despite being very efficient at what it does, though, MapReduce has some disadvantages; principally that it's batch-based, and as a result isn't suited to real-time or even near-real-time data processing. Historically this has meant that processing models such as graph, iterative, and real-time data processing are not a natural fit for MapReduce.[3]

The bottom line is that Hadoop version 1 restricts you from running exciting new processing frameworks.

YARN changes all of this by taking over the scheduling portions of MapReduce, and nothing else. At its core, YARN is a distributed scheduler and is responsible for two activities:

[3] HBase is an exception; it uses HDFS for storage but doesn't use MapReduce for the processing engine.

- *Responding to a client's request to create a container*—A container is in essence a process, with a contract governing the physical resources that it's permitted to use.
- *Monitoring containers that are running, and terminating them if needed*—Containers can be terminated if a YARN scheduler wants to free up resources so that containers from other applications can run, or if a container is using more than its allocated resources.

Table 2.1 compares MapReduce 1 and YARN (in Hadoop versions 1 and 2) to show why YARN is such a revolutionary jump.

Table 2.1 Comparison of MapReduce 1 and YARN

Capability	MapReduce 1	YARN
Execution model	Only MapReduce is supported on Hadoop 1, limiting the types of activities you can perform to batch-based flows that fit within the confines of the MapReduce processing model.	YARN places no restrictions on the type of work that can be executed in Hadoop; you pick which execution engines you need (whether it's real-time processing with Spark, graph processing with Giraph, or MapReduce batch processing), and they can all be executing in parallel on the same cluster.
Concurrent processes	MapReduce had the notion of "slots," which were node-specific static configurations that determined the maximum number of map and reduce processes that could run concurrently on each node. Based on where in the lifecycle a MapReduce application was, this would often lead to underutilized clusters.	YARN allows for more fluid resource allocation, and the number of processes is limited only by the configured maximum amount of memory and CPU for each node.
Memory limits	Slots in Hadoop 1 also had a maximum limit, so typically Hadoop 1 clusters were provisioned such that the number of slots multiplied by the maximum configured memory for each slot was less than the available RAM. This often resulted in smaller than desired maximum slot memory sizes, which impeded your ability to run memory-intensive jobs.[a] Another drawback of MRv1 was that it was more difficult for memory-intensive and IO-intensive jobs to coexist on the same cluster or machines. Either you had more slots to boost the I/O jobs, or fewer slots but more RAM for RAM jobs. Once again, the static nature of these slots made it a challenge to tune clusters for mixed workloads.	YARN allows applications to request resources of varying memory sizes. YARN has minimum and maximum memory limits, but because the number of slots is no longer fixed, the maximum values can be much larger to support memory-intensive workloads. YARN therefore provides a much more dynamic scheduling model that doesn't limit the number of processes or the amount of RAM requested by a process.
Scalability	There were concurrency issues with the JobTracker, which limited the number of nodes in a Hadoop cluster to 3,000–4,000 nodes.	By separating out the scheduling parts of MapReduce into YARN and making it lightweight by delegating fault tolerance to YARN applications, YARN can scale to much larger numbers than prior versions of Hadoop.[b]

Table 2.1 Comparison of MapReduce 1 and YARN *(continued)*

Capability	MapReduce 1	YARN
Execution	Only a single version of MapReduce could be supported on a cluster at a time. This was problematic in large multi-tenant environments where product teams that wanted to upgrade to newer versions of MapReduce had to convince all the other users. This typically resulted in huge coordination and integration efforts and made such upgrades huge infrastructure projects.	MapReduce is no longer at the core of Hadoop, and is now a YARN application running in user space. This means that you can now run different versions of MapReduce on the same cluster at the same time. This is a huge productivity gain in large multi-tenant environments, and it allows you to organizationally decouple product teams and roadmaps.

[a] This limitation in MapReduce was especially painful for those running machine-learning tasks using tools such as Mahout, as they often required large amounts of RAM for processing—amounts often larger than the maximum configured slot size in MapReduce.

[b] The goal of YARN is to be able to scale to 10,000 nodes; scaling beyond that number could result in the ResourceManager becoming a bottleneck, as it's a single process.

Now that you know about the key benefits of YARN, it's time to look at the main components in YARN and examine their roles.

2.1.2 YARN concepts and components

YARN comprises a framework that's responsible for resource scheduling and monitoring, and applications that execute application-specific logic in a cluster. Let's examine YARN concepts and components in more detail, starting with the YARN framework components.

YARN FRAMEWORK

The YARN framework performs one primary function, which is to schedule resources (*containers* in YARN parlance) in a cluster. Applications in a cluster talk to the YARN framework, asking for application-specific containers to be allocated, and the YARN framework evaluates these requests and attempts to fulfill them. An important part of the YARN scheduling also includes monitoring currently executing containers. There are two reasons that container monitoring is important: Once a container has completed, the scheduler can then use freed-up capacity to schedule more work. Additionally, each container has a contract that specifies the system resources that it's allowed to use, and in cases where containers overstep these bounds, the scheduler can terminate the container to avoid rogue containers impacting other applications.

The YARN framework was intentionally designed to be as simple as possible; as such, it doesn't know or care about the type of applications that are running. Nor does it care about keeping any historical information about what has executed on the cluster. These design decisions are the primary reasons that YARN can scale beyond the levels of MapReduce.

There are two primary components that comprise the YARN framework—the ResourceManager and the NodeManager—which are seen in figure 2.2.

- *ResourceManager*—A Hadoop cluster has a single *ResourceManager* (RM) for the entire cluster. The ResourceManager is the YARN master process, and its sole function is to arbitrate resources on a Hadoop cluster. It responds to client requests to create containers, and a scheduler determines when and where a container can be created according to scheduler-specific multi-tenancy rules that govern who can create containers where and when. Just like with Hadoop 1, the scheduler part of the ResourceManager is pluggable, which means that you can pick the scheduler that works best for your environment. The actual creation of containers is delegated to the NodeManager.
- *NodeManager*—The *NodeManager* is the slave process that runs on every node in a cluster. Its job is to create, monitor, and kill containers. It services requests from the ResourceManager and ApplicationMaster to create containers, and it reports on the status of the containers to the ResourceManager. The ResourceManager uses the data contained in these status messages to make scheduling decisions for new container requests.

In non-HA mode, only a single instance of the ResourceManager exists.[4]

The YARN framework exists to manage applications, so let's take a look at what components a YARN application is composed of.

YARN APPLICATIONS

A YARN application implements a specific function that runs on Hadoop. MapReduce is an example of a YARN application, as are projects such as Hoya, which allows multiple HBase instances to run on a single cluster, and storm-yarn, which allows Storm to run inside a Hadoop cluster. You'll see more details on these projects and other YARN applications later in this chapter.

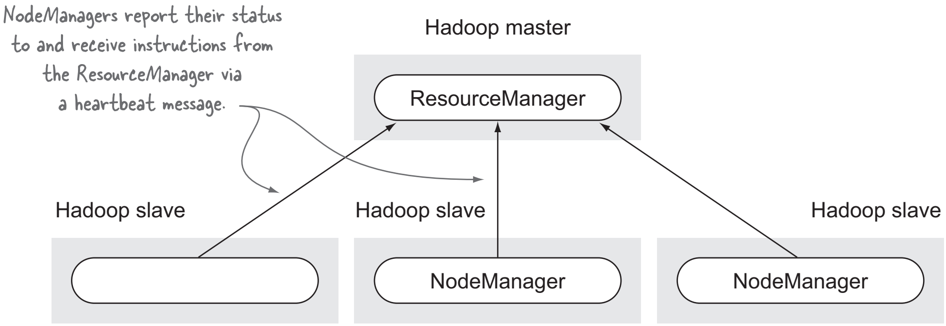

NodeManagers report their status to and receive instructions from the ResourceManager via a heartbeat message.
Hadoop master
ResourceManager
Hadoop slave
Hadoop slave
Hadoop slave
NodeManager
NodeManager

Figure 2.2 YARN framework components and their interactions. Application-specific components, such as the YARN client, ApplicationMaster, and containers are not shown.

[4] As of the time of writing, YARN ResourceManager HA is still actively being developed, and its progress can be followed on a JIRA ticket titled "ResourceManager (RM) High-Availability (HA)," https://issues.apache.org/jira/browse/YARN-149.

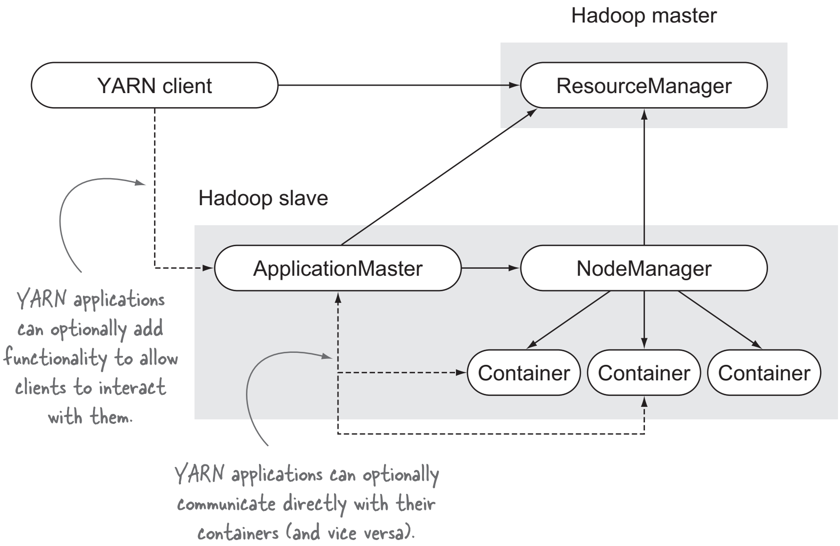

Hadoop master
YARN client
ResourceManager
Hadoop slave
ApplicationMaster
NodeManager
Container
Container
Container
YARN applications can optionally add functionality to allow clients to interact with them.
YARN applications can optionally communicate directly with their containers (and vice versa).

Figure 2.3 Typical interactions of a YARN application

A YARN application involves three components—the client, the ApplicationMaster (AM), and the container, which can be seen in figure 2.3.

Launching a new YARN application starts with a YARN client communicating with the ResourceManager to create a new YARN ApplicationMaster instance. Part of this process involves the YARN client informing the ResourceManager of the ApplicationMaster's physical resource requirements.

The *ApplicationMaster* is the master process of a YARN application. It doesn't perform any application-specific work, as these functions are delegated to the containers. Instead, it's responsible for managing the application-specific containers: asking the ResourceManager of its intent to create containers and then liaising with the NodeManager to actually perform the container creation.

As part of this process, the ApplicationMaster must specify the resources that each container requires in terms of which host should launch the container and what the container's memory and CPU requirements are.[5] The ability of the ResourceManager to schedule work based on exact resource requirements is a key to YARN's flexibility, and it enables hosts to run a mix of containers, as highlighted in figure 2.4.

The ApplicationMaster is also responsible for the specific fault-tolerance behavior of the application. It receives status messages from the ResourceManager when its containers fail, and it can decide to take action based on these events (by asking the ResourceManager to create a new container), or to ignore these events.[6]

[5] Future versions of Hadoop may allow network, disk, and GPU requirements to be specified.

[6] Containers can fail for a variety of reasons, including a node going down, a container being killed by YARN to allow another application's container to be launched, or YARN killing a container when the container exceeds its configured physical/virtual memory.

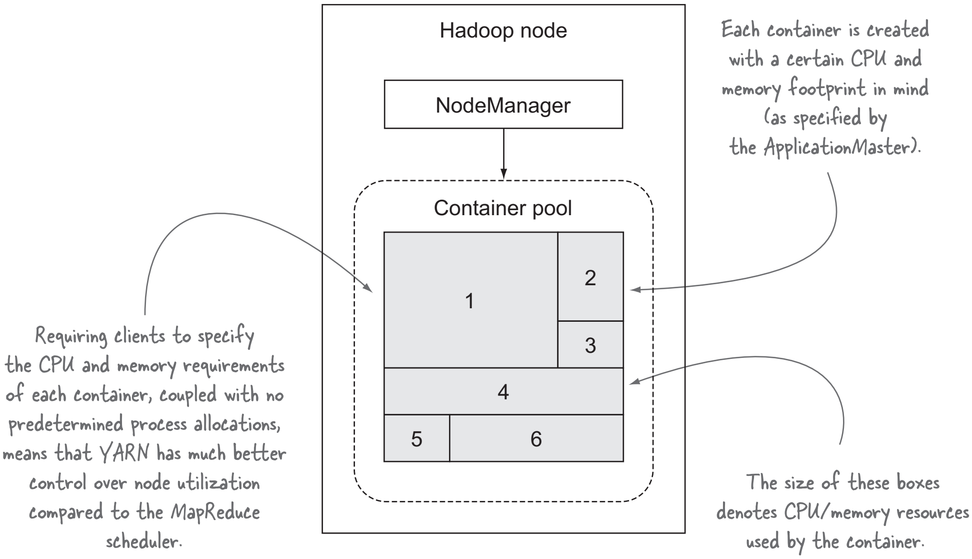

Hadoop node
NodeManager
Container pool
1
2
3
4
5
6
Each container is created with a certain CPU and memory footprint in mind (as specified by the ApplicationMaster).
Requiring clients to specify the CPU and memory requirements of each container, coupled with no predetermined process allocations, means that YARN has much better control over node utilization compared to the MapReduce scheduler.
The size of these boxes denotes CPU/memory resources used by the container.

Figure 2.4 Various container configurations running on a single YARN-managed Hadoop node

A *container* is an application-specific process that's created by a NodeManager on behalf of an ApplicationMaster. The ApplicationManager itself is also a container, created by the ResourceManager. A container created by an ApplicationManager can be an arbitrary process—for example, a container process could simply be a Linux command such as `awk`, a Python application, or any process that can be launched by the operating system. This is the power of YARN—the ability to launch and manage any process across any node in a Hadoop cluster.

By this point, you should have a high-level understanding of the YARN components and what they do. Next we'll look at common YARN configurables.

2.1.3 YARN configuration

YARN brings with it a whole slew of configurations for various components, such as the UI, remote procedure calls (RPCs), the scheduler, and more.[7] In this section, you'll learn how you can quickly access your running cluster's configuration.

TECHNIQUE 1 Determining the configuration of your cluster

Figuring out the configuration for a running Hadoop cluster can be a nuisance—it often requires looking at several configuration files, including the default configuration files, to determine the value for the property you're interested in. In this technique, you'll see how to sidestep the hoops you normally need to jump through, and instead focus on how to expediently get at the configuration of a running Hadoop cluster.

[7] Details on the default YARN configurations can be seen at http://hadoop.apache.org/docs/r2.2.0/hadoop-yarn/hadoop-yarn-common/yarn-default.xml.

Problem

You want to access the configuration of a running Hadoop cluster.

Solution

View the configuration using the ResourceManager UI.

Discussion

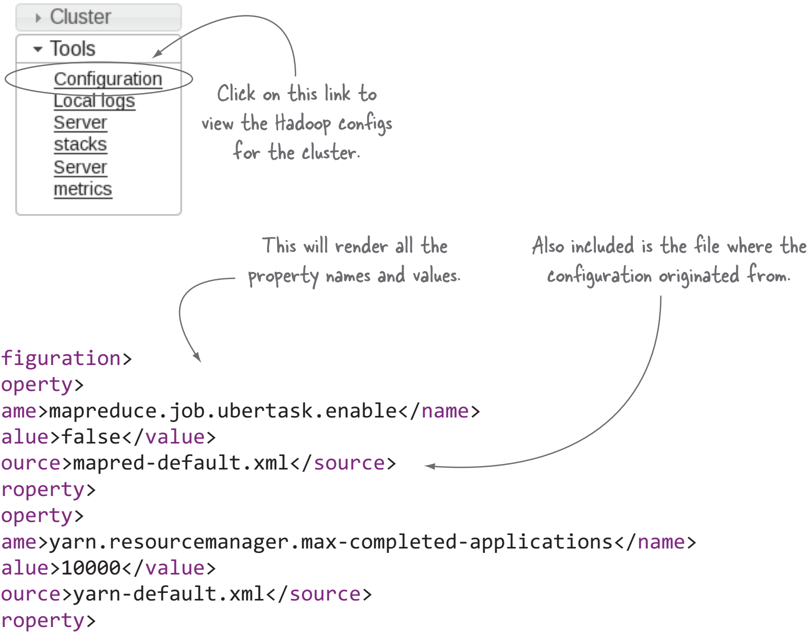
The ResourceManager UI shows the configuration for your Hadoop cluster; figure 2.5 shows how you can navigate to this information.

What's useful about this feature is that the UI shows not only a property value, but also which file it originated from. If the value wasn't defined in a <component>-site.xml file, then it'll show the default value and the default filename.

Another useful feature of this UI is that it'll show you the configuration from multiple files, including the core, HDFS, YARN, and MapReduce files.

The configuration for an individual Hadoop slave node can be navigated to in the same way from the NodeManager UI. This is most helpful when working with Hadoop clusters that consist of heterogeneous nodes, where you often have varying configurations that cater to differing hardware resources.

By this point, you should have a high-level understanding of the YARN components, what they do, and how to configure them for your cluster. The next step is to actually see YARN in action by using the command line and the UI.



Click on this link to view the Hadoop configs for the cluster.

This will render all the property names and values.

Also included is the file where the configuration originated from.

```
-<configuration>
 -<property>
   <name>mapreduce.job.ubertask.enable</name>
   <value>false</value>
   <source>mapred-default.xml</source>
  </property>
 -<property>
   <name>yarn.resourcemanager.max-completed-applications</name>
   <value>10000</value>
   <source>yarn-default.xml</source>
  </property>
```

Figure 2.5 The YARN ResourceManager UI showing the cluster's configuration

2.1.4 Interacting with YARN

Out of the box, Hadoop 2 is bundled with two YARN applications—MapReduce 2 and DistributedShell. You'll learn more about MapReduce 2 later in this chapter, but for now, you can get your toes wet by taking a look at a simpler example of a YARN application: the DistributedShell. You'll see how to run your first YARN application and where to go to examine the logs.

If you don't know the configured values for your cluster, you have two options:

- Examine the contents of yarn-site.xml to view the property values. If an entry doesn't exist, the default value will be in effect.[8]
- Even better, use the ResourceManager UI, which gives you more detailed information on the running configuration, including what the default values are and if they're in effect.

Let's now take a look at how to quickly view the YARN configuration for a running Hadoop cluster.

TECHNIQUE 2 Running a command on your YARN cluster

Running a command on your cluster is a good first step when you start working with a new YARN cluster. It's the "hello world" in YARN, if you will.

Problem

You want to run a Linux command on a node in your Hadoop cluster.

Solution

Use the DistributedShell example application bundled with Hadoop.

Discussion

YARN is bundled with the DistributedShell application, which serves two primary purposes—it's a reference YARN application that's also a handy utility for running a command in parallel across your Hadoop cluster. Start by issuing a Linux `find` command in a single container:

The command you're running—in this case the Linux find command.

The JAR file that contains the distributed shell example.

Supply additional args for the command. Here you want to perform the find in the current directory of the container.

```
     org.apache.hadoop.yarn.applications.distributedshell.Client \
       \
-shell_command find \
-shell_args '`pwd`'  \
-jar ${HADOOP_HOME}/share/hadoop/yarn/*-distributedshell-*.jar \
-container_memory 350 \
-master_memory 350
```

The amount of memory reserved for the container that will run the command.

The amount of memory for the ApplicationMaster.

If all is well with your cluster, then executing the preceding command will result in the following log message:

```
INFO distributedshell.Client: Application completed successfully
```

[8] Visit the following URL for YARN default values: http://hadoop.apache.org/docs/r2.2.0/hadoop-yarn/hadoop-yarn-common/yarn-default.xml.

There are various other logging statements that you'll see in the command's output prior to this line, but you'll notice that none of them contain the actual results of your `find` command. This is because the DistributedShell ApplicationMaster launches the `find` command in a separate container, and the standard output (and standard error) of the `find` command is redirected to the log output directory of the container. To see the output of your command, you need to get access to that directory. That, as it happens, is covered in the next technique!

TECHNIQUE 3 Accessing container logs

Turning to the log files is the most common first step one takes when trying to diagnose an application that behaved in an unexpected way, or to simply understand more about the application. In this technique, you'll learn how to access these application log files.

Problem

You want to access container log files.

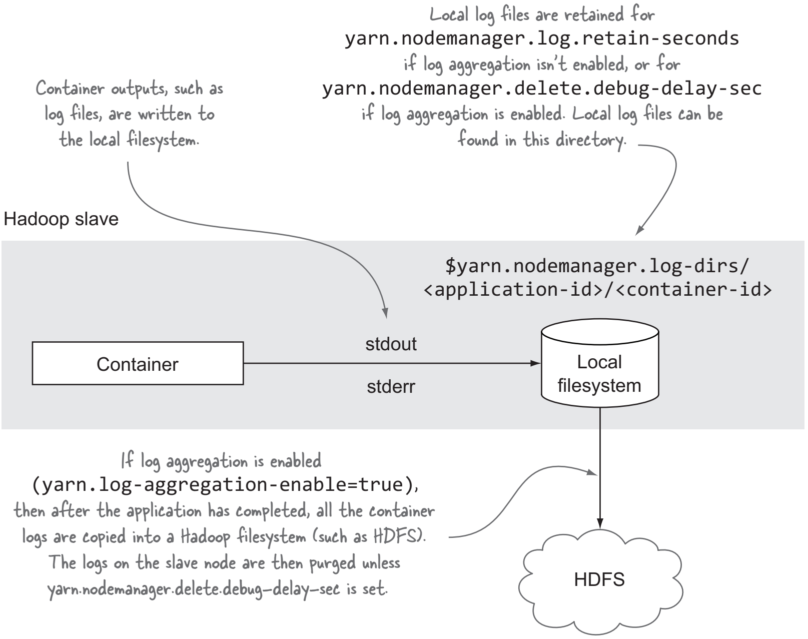
Solution

Use YARN's UI and the command line to access the logs.

Discussion

Each container that runs in YARN has its own output directory, where the standard output, standard error, and any other output files are written. Figure 2.6 shows the location of the output directory on a slave node, including the data retention details for the logs.

Access to container logs is not as simple as it should be—let's take a look at how you can use the CLI and the UIs to access logs.



Container outputs, such as log files, are written to the local filesystem.

Local log files are retained for `yarn.nodemanager.log.retain-seconds` if log aggregation isn't enabled, or for `yarn.nodemanager.delete.debug-delay-sec` if log aggregation is enabled. Local log files can be found in this directory.

If log aggregation is enabled (`yarn.log-aggregation-enable=true`), then after the application has completed, all the container logs are copied into a Hadoop filesystem (such as HDFS). The logs on the slave node are then purged unless yarn.nodemanager.delete.debug-delay-sec is set.

Figure 2.6 **Container log locations and retention**

Accessing container logs using the YARN command line

YARN comes with a command-line interface (CLI) for accessing YARN application logs. To use the CLI, you need to know the ID of your application.

> **How do I find the application ID?** Most YARN clients will display the application ID in their output and logs. For example, the DistributedShell command that you executed in the previous technique echoed the application ID to standard output:
>
> ```
> $ hadoop o.a.h.y.a.d.Client ...
> ...
> INFO impl.YarnClientImpl:
>      Submitted application application_1388257115348_0008 to
>      ResourceManager at /0.0.0.0:8032
> ...
> ```
>
> Alternatively, you can use the CLI (using `yarn application -list`) or the ResourceManager UI to browse and find your application ID.

If you attempt to use the CLI when the application is still running, you'll be presented with the following error message:

```
$ yarn logs -applicationId application_1398974791337_0070
Application has not completed. Logs are only available after
an application completes
```

The message tells it all—the CLI is only useful once an application has completed. You'll need to use the UI to access the container logs when the application is running, which we'll cover shortly.

Once the application has completed, you may see the following output if you attempt to run the command again:

```
$ yarn logs -applicationId application_1400286711208_0001
Logs not available at /tmp/.../application_1400286711208_0001
Log aggregation has not completed or is not enabled.
```

Basically, the YARN CLI only works if the application has completed and log aggregation is enabled. Log aggregation is covered in the next technique. If you enable log aggregation, the CLI will give you the logs for all the containers in your application, as you can see in the next example:

```
$ yarn logs -applicationId application_1400287920505_0002
client.RMProxy: Connecting to ResourceManager at /0.0.0.0:8032

Container: container_1400287920505_0002_01_000002
           on localhost.localdomain_57276
=================================================
LogType: stderr
LogLength: 0
Log Contents:

LogType: stdout
```

```
LogLength: 1355
Log Contents:
/tmp
default_container_executor.sh
/launch_container.sh
/.launch_container.sh.crc
/.default_container_executor.sh.crc
/.container_tokens.crc
/AppMaster.jar
/container_tokens



Container: container_1400287920505_0002_01_000001
           on localhost.localdomain_57276
=================================================
LogType: AppMaster.stderr
LogLength: 17170
Log Contents:
distributedshell.ApplicationMaster: Initializing ApplicationMaster
...

LogType: AppMaster.stdout
LogLength: 8458
Log Contents:
System env: key=TERM, val=xterm-256color
...
```

The preceding output shows the contents of the logs of the DistributedShell example that you ran in the previous technique. There are two containers in the output—one for the `find` command that was executed, and the other for the ApplicationMaster, which is also executed within a container.

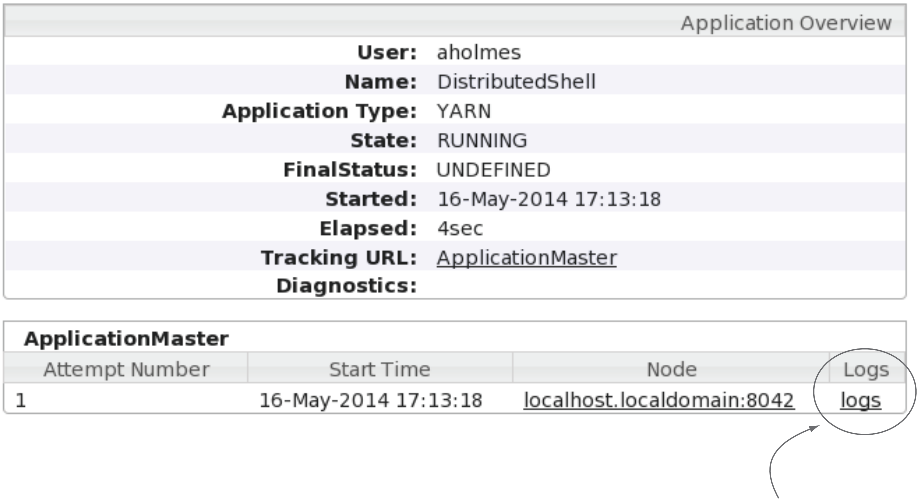
Accessing logs using the YARN UIs

YARN provides access to the ApplicationMaster logs via the ResourceManager UI. On a pseudo-distributed setup, point your browser at http://localhost:8088/cluster. If you're working with a multi-node Hadoop cluster, point your browser at http://$yarn.resourcemanager.webapp.address/cluster. Click on the application you're interested in, and then select the Logs link as shown in figure 2.7.

Great, but how do you access the logs for containers other than the ApplicationMaster? Unfortunately, things get a little murky here. The ResourceManager doesn't keep track of a YARN application's containers, so it can't provide you with a way to list and navigate to the container logs. Therefore, the onus is on individual YARN applications to provide their users with a way to access container logs.

> **Hey, ResourceManager, what are my container IDs?** In order to keep the ResourceManager lightweight, it doesn't keep track of the container IDs for an application. As a result, the ResourceManager UI only provides a way to access the ApplicationMaster logs for an application.

Case in point is the DistributedShell application. It's a simple application that doesn't provide an ApplicationMaster UI or keep track of the containers that it's launched.



Application Overview

User:	aholmes
Name:	DistributedShell
Application Type:	YARN
State:	RUNNING
FinalStatus:	UNDEFINED
Started:	16-May-2014 17:13:18
Elapsed:	4sec
Tracking URL:	ApplicationMaster
Diagnostics:	

ApplicationMaster

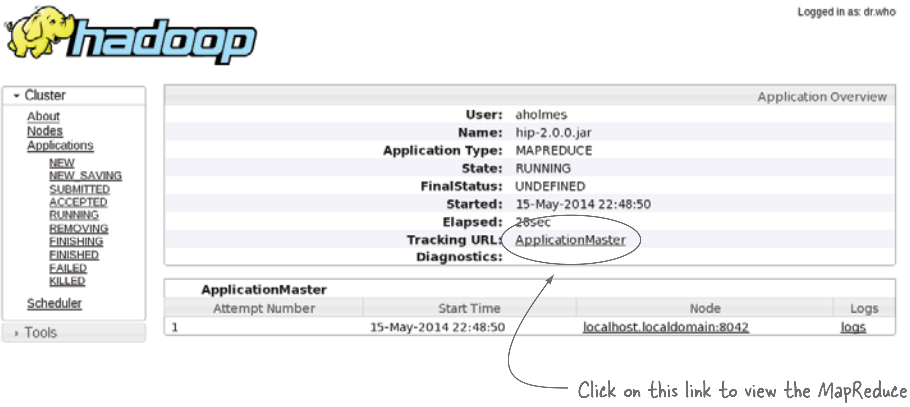
Attempt Number	Start Time	Node	Logs
1	16-May-2014 17:13:18	localhost.localdomain:8042	logs

Click on this link to view the ApplicationMaster logs.

Figure 2.7 The YARN ResourceManager UI showing the ApplicationMaster container

Therefore, there's no easy way to view the container logs other than by using the approach presented earlier: using the CLI.

Luckily, the MapReduce YARN application provides an ApplicationMaster UI that you can use to access the container (the map and reduce task) logs, as well as a JobHistory UI that can be used to access logs after a MapReduce job has completed. When you run a MapReduce job, the ResourceManager UI gives you a link to the MapReduce ApplicationMaster UI, as shown in figure 2.8, which you can use to access the map and reduce logs (much like the JobTracker in MapReduce 1).



Application Overview

User:	aholmes
Name:	hip-2.0.0.jar
Application Type:	MAPREDUCE
State:	RUNNING
FinalStatus:	UNDEFINED
Started:	15-May-2014 22:48:50
Elapsed:	28sec
Tracking URL:	ApplicationMaster
Diagnostics:	

ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	15-May-2014 22:48:50	localhost.localdomain:8042	logs

Click on this link to view the MapReduce UI for a running job; this also provides access to map and reduce logs.

Figure 2.8 Accessing the MapReduce UI for a running job

If your YARN application provides some way for you to identify container IDs and the hosts that they execute on, you can either access the container logs using the NodeManager UI or you can use a shell to `ssh` to the slave node that executed a container.

The NodeManager URL for accessing a container's logs is http://<nodemanager-host>:8042/node/containerlogs/<container-id>/<username>. Alternatively, you can `ssh` to the NodeManager host and access the container logs directory at $yarn.nodemanager.log-dirs/<application-id>/<container-id>.

Really, the best advice I can give here is that you should enable log aggregation, which will allow you to use the CLI, HDFS, and UIs, such as the MapReduce ApplicationMaster and JobHistory, to access application logs. Keep reading for details on how to do this.

TECHNIQUE 4 Aggregating container log files

Log aggregation is a feature that was missing from Hadoop 1, making it challenging to archive and access task logs. Luckily Hadoop 2 has this feature baked-in, and you have a number of ways to access aggregated log files. In this technique you'll learn how to configure your cluster to archive log files for long-term storage and access.

■ Problem

You want to aggregate container log files to HDFS and manage their retention policies.

■ Solution

Use YARN's built-in log aggregation capabilities.

■ Discussion

In Hadoop 1 your logs were stowed locally on each slave node, with the JobTracker and TaskTracker being the only mechanisms for getting access to these logs. This was cumbersome and didn't easily support programmatic access to them. In addition, log files would often disappear due to aggressive log-retention policies that existed to prevent local disks on slave nodes from filling up.

Log aggregation in Hadoop 2 is therefore a welcome feature, and if enabled, it copies container log files into a Hadoop filesystem (such as HDFS) after a YARN application has completed. By default, this behavior is disabled, and you need to set `yarn.log-aggregation-enable` to `true` to enable this feature. Figure 2.9 shows the data flow for container log files.

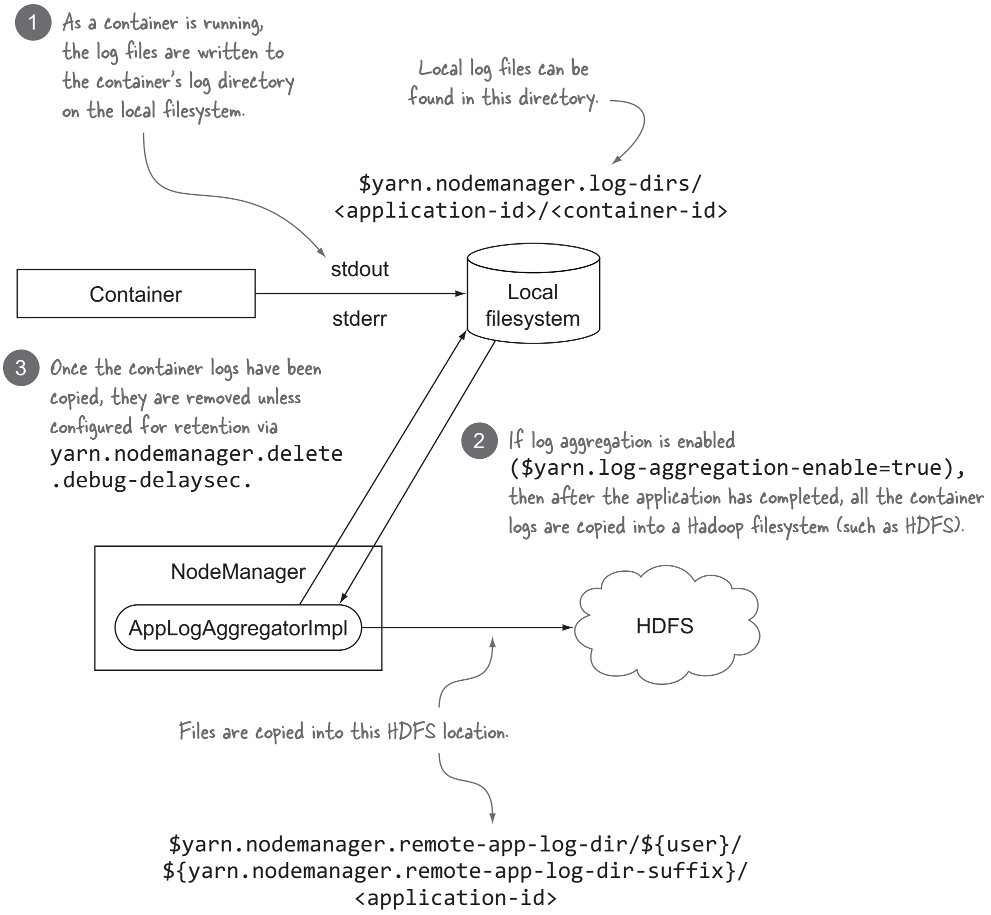
Now that you know how log aggregation works, let's take a look at how you can access aggregated logs.

Accessing log files using the CLI

With your application ID in hand (see technique 3 for details on how to get it), you can use the command line to fetch all the logs and write them to the console:

```
$ yarn logs -applicationId application_1388248867335_0003
```

> **Enabling log aggregation** If the preceding `yarn logs` command yields the following output, then it's likely that you don't have YARN log aggregation enabled:
>
> ```
> Log aggregation has not completed or is not enabled.
> ```


Container
stdout
stderr
Local filesystem
NodeManager
AppLogAggregatorImpl
HDFS
$yarn.nodemanager.log-dirs/
<application-id>/<container-id>
$yarn.nodemanager.remote-app-log-dir/${user}/
${yarn.nodemanager.remote-app-log-dir-suffix}/
<application-id>

1 As a container is running, the log files are written to the container's log directory on the local filesystem.

Local log files can be found in this directory.

2 If log aggregation is enabled ($yarn.log-aggregation-enable=true), then after the application has completed, all the container logs are copied into a Hadoop filesystem (such as HDFS).

3 Once the container logs have been copied, they are removed unless configured for retention via yarn.nodemanager.delete .debug-delaysec.

Files are copied into this HDFS location.

Figure 2.9 Log file aggregation from local filesystem to HDFS

This will dump out all the logs for all the containers for the YARN application. The output for each container is delimited with a header indicating the container ID, followed by details on each file in the container's output directory. For example, if you ran a DistributedShell command that executed `ls -l`, then the output of the `yarn logs` command would yield something like the following:

```
Container: container_1388248867335_0003_01_000002 on localhost
==============================================================
LogType: stderr
LogLength: 0
Log Contents:

LogType: stdoutLogLength: 268
Log Contents:
total 32
-rw-r--r-- 1 aholmes 12:29 container_tokens
```

```
-rwx------ 1 aholmes 12:29 default_container_executor.sh
-rwx------ 1 aholmes launch_container.sh
drwx--x--- 2 aholmes tmp

Container: container_1388248867335_0003_01_000001 on localhost
==============================================================
LogType: AppMaster.stderr
(the remainder of the ApplicationMaster logs removed for brevity)
```

The stdout file contains the directory listing of the `ls` process's current directory, which is a container-specific working directory.

Accessing aggregated logs via the UI

Fully featured YARN applications such as MapReduce provide an ApplicationMaster UI that can be used to access container logs. Similarly, the JobHistory UI can also access aggregated logs.

> **UI aggregated log rendering** If log aggregation is enabled, you'll need to update yarn-site.xml and set `yarn.log.server.url` to point at the job history server so that the ResourceManager UI can render the logs.

Accessing log files in HDFS

By default, aggregated log files go into the following directory in HDFS:

```
/tmp/logs/${user}/logs/application_<appid>
```

The directory prefix can be configured via the `yarn.nodemanager.remote-app-log-dir` property; similarly, the path name after the username ("logs" in the previous example, which is the default) can be customized via `yarn.nodemanager.remote-app-log-dir-suffix`.

Differences between log files in local filesystem and HDFS

As you saw earlier, each container results in two log files in the local filesystem: one for standard output and another for standard error. As part of the aggregation process, all the files for a given node are concatenated together into a node-specific log. For example, if you had five containers running across three nodes, you'd end up with three log files in HDFS.

Compression

Compression of aggregated logs is disabled by default, but you can enable it by setting the value of `yarn.nodemanager.log-aggregation.compression-type` to either `lzo` or `gzip` depending on your compression requirements. As of Hadoop 2.2, these are the only two compression codecs supported.

Log retention

When log aggregation is turned off, the container log files on the local host are retained for `yarn.nodemanager.log.retain-seconds` seconds, the default being 10,800 (3 hours).

When log aggregation is turned on, the `yarn.nodemanager.log.retain-seconds` configurable is ignored, and instead the local container log files are deleted as soon as they are copied into HDFS. But all is not lost if you want to retain them on the local filesystem—simply set `yarn.nodemanager.delete.debug-delay-sec` to a value that you want to keep the files around for. Note that this applies not only to the log files but also to all other metadata associated with the container (such as JAR files).

The data retention for the files in HDFS is configured via a different setting, `yarn.log-aggregation.retain-seconds`.

NameNode considerations

At scale, you may want to consider an aggressive log retention setting so that you don't overwhelm the NameNode with all the log file metadata. The NameNode keeps the metadata in memory, and on a large active cluster, the number of log files can quickly overwhelm the NameNode.

> **Real-life example of NameNode impact** Take a look at Bobby Evans' "Our Experience Running YARN at Scale" (http://www.slideshare.net/Hadoop_Summit/evans-june27-230pmroom210c) for a real-life example of how Yahoo! utilized 30% of their NameNode with seven days' worth of aggregated logs.

Alternative solutions

The solution highlighted in this technique is useful for getting your logs into HDFS, but if you will need to organize any log mining or visualization activities yourself, there are other options available such as Hunk, which supports aggregating logs from both Hadoop 1 and 2 and providing first-class query, visualization, and monitoring features, just like regular Splunk. You could also set up a query and visualization pipeline using tools such as Logstash, ElasticSearch, and Kibana if you want to own the log management process. Other tools such as Loggly are worth investigating.

For now, this concludes our hands-on look at YARN. That's not the end of the story, however. Section 2.2 looks at how MapReduce works as a YARN application, and later in chapter 10, you'll learn how to write your own YARN applications.

2.1.5 YARN challenges

There are some gotchas to be aware of with YARN:

- *YARN currently isn't designed to work well with long-running processes.* This has created challenges for projects such as Impala and Tez that would benefit from such a feature. Work is currently underway to bring this feature to YARN, and it's being tracked in a JIRA ticket titled "Roll up for long-lived services in YARN," https://issues.apache.org/jira/browse/YARN-896.
- *Writing YARN applications is quite complex, as you're required to implement container management and fault tolerance.* This may require some complex Application-Master and container-state management so that upon failure the work can continue from some previous well-known state. There are several frameworks whose goal is to simplify development—refer to chapter 10 for more details.
- *Gang scheduling, which is the ability to rapidly launch a large number of containers in parallel, is currently not supported.* This is another feature that projects such as Impala and Hamster (OpenMPI) would require for native YARN integration. The

Hadoop committers are currently working on adding support for gang scheduling, which is being tracked in the JIRA ticket titled "Support gang scheduling in the AM RM protocol," https://issues.apache.org/jira/browse/YARN-624.

So far we've focused on the capabilities of the core YARN system. Let's move on to look at how MapReduce works as a YARN application.

2.2 YARN and MapReduce

In Hadoop 1, MapReduce was the only way to process your data natively in Hadoop. YARN was created so that Hadoop clusters could run any type of work, and its only requirement was that applications adhere to the YARN specification. This meant MapReduce had to become a YARN application and required the Hadoop developers to rewrite key parts of MapReduce.

Given that MapReduce had to go through some open-heart surgery to get it working as a YARN application, the goal of this section is to demystify how MapReduce works in Hadoop 2. You'll see how MapReduce 2 executes in a Hadoop cluster, and you'll also get to look at configuration changes and backward compatibility with MapReduce 1. Toward the end of this section, you'll learn how to run and monitor jobs, and you'll see how small jobs can be quickly executed.

There's a lot to go over, so let's take MapReduce into the lab and see what's going on under the covers.

2.2.1 Dissecting a YARN MapReduce application

Architectural changes had to be made to MapReduce to port it to YARN. Figure 2.10 shows the processes involved in MRv2 and some of the interactions between them.

Each MapReduce job is executed as a separate YARN application. When you launch a new MapReduce job, the client calculates the input splits and writes them along with other job resources into HDFS (step 1). The client then communicates with the ResourceManager to create the ApplicationMaster for the MapReduce job (step 2). The ApplicationMaster is actually a container, so the ResourceManager will allocate the container when resources become available on the cluster and then communicate with a NodeManager to create the ApplicationMaster container (steps 3–4).[9]

The MapReduce ApplicationMaster (MRAM) is responsible for creating map and reduce containers and monitoring their status. The MRAM pulls the input splits from HDFS (step 5) so that when it communicates with the ResourceManager (step 6) it can request that map containers are launched on nodes local to their input data.

Container allocation requests to the ResourceManager are piggybacked on regular heartbeat messages that flow between the ApplicationMaster and the ResourceManager. The heartbeat responses may contain details on containers that are allocated for the application. Data locality is maintained as an important part of the architecture—when it requests map containers, the MapReduce ApplicationManager will use the input splits' location details to request that the containers are assigned to

[9] If there aren't any available resources for creating the container, the ResourceManager may choose to kill one or more existing containers to free up space.

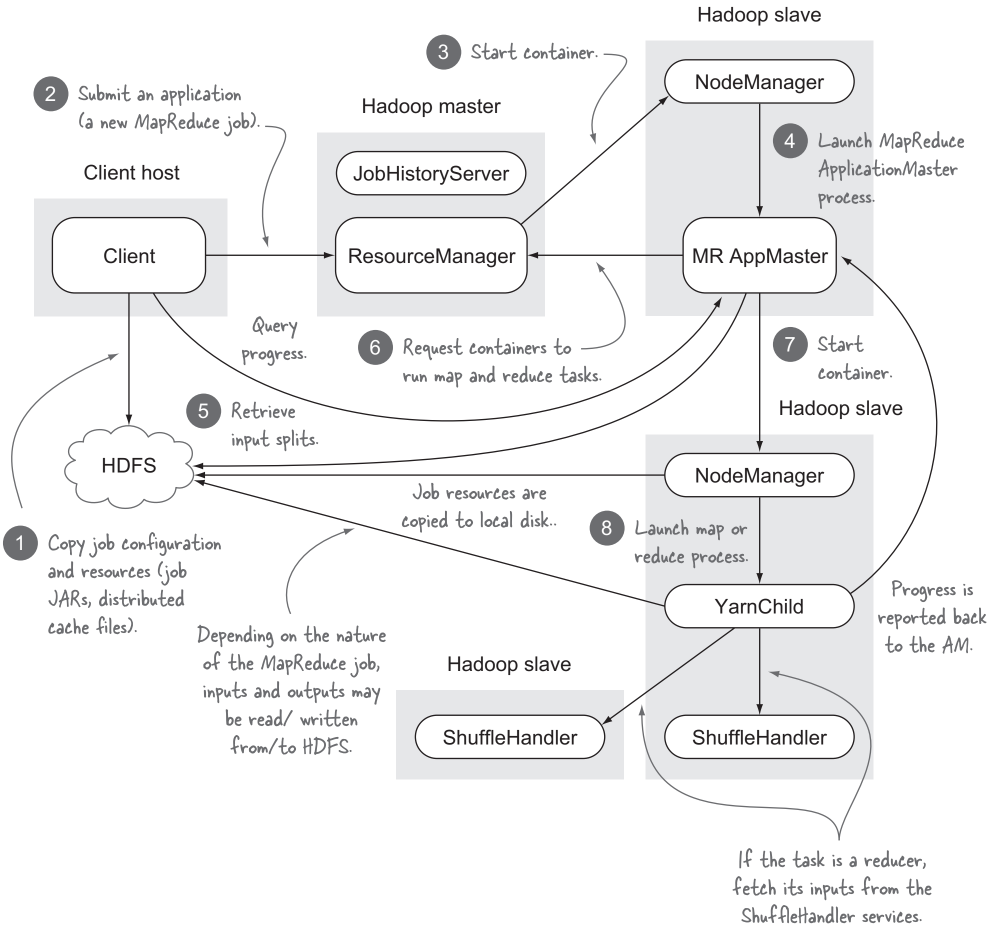

Hadoop slave
3 Start container.
NodeManager
2 Submit an application (a new MapReduce job).
Hadoop master
4 Launch MapReduce ApplicationMaster process.
Client host
JobHistoryServer
Client
ResourceManager
MR AppMaster
Query progress.
6 Request containers to run map and reduce tasks.
7 Start container.
5 Retrieve input splits.
Hadoop slave
HDFS
NodeManager
Job resources are copied to local disk..
1 Copy job configuration and resources (job JARs, distributed cache files).
8 Launch map or reduce process.
Progress is reported back to the AM.
YarnChild
Depending on the nature of the MapReduce job, inputs and outputs may be read/ written from/to HDFS.
Hadoop slave
ShuffleHandler
ShuffleHandler
If the task is a reducer, fetch its inputs from the ShuffleHandler services.

Figure 2.10 The interactions of a MapReduce 2 YARN application

one of the nodes that contains the input splits, and the ResourceManager will make a best attempt at container allocation on these input split nodes.

Once the MapReduce ApplicationManager is allocated a container, it talks to the NodeManager to launch the map or reduce task (steps 7–8). At this point, the map/reduce process acts very similarly to the way it worked in MRv1.

THE SHUFFLE

The shuffle phase in MapReduce, which is responsible for sorting mapper outputs and distributing them to the reducers, didn't fundamentally change in MapReduce 2. The main difference is that the map outputs are fetched via ShuffleHandlers, which are auxiliary YARN services that run on each slave node.[10] Some minor memory

[10] The ShuffleHandler must be configured in your yarn-site.xml; the property name is `yarn.nodemanager.aux-services` and the `value is mapreduce_shuffle`.

management tweaks were made to the shuffle implementation; for example, io.sort.record.percent is no longer used.

WHERE'S THE JOBTRACKER?

You'll note that the JobTracker no longer exists in this architecture. The scheduling part of the JobTracker was moved as a general-purpose resource scheduler into the YARN ResourceManager. The remaining part of JobTracker, which is primarily the metadata about running and completed jobs, was split in two. Each MapReduce ApplicationMaster hosts a UI that renders details on the current job, and once jobs are completed, their details are pushed to the JobHistoryServer, which aggregates and renders details on all completed jobs. Refer to section 2.2.5 for additional details, including how to access the MapReduce ApplicationMaster UI.

Hopefully, you now have a better sense of how MapReduce 2 works. MapReduce configuration didn't go untouched in the move to YARN, so let's take a look at what's hot and what's not.

2.2.2 Configuration

The port of MapReduce 2 to YARN brought with it some major changes in the MapReduce properties. In this section, we'll cover some of the frequently used properties that have been affected.

NEW PROPERTIES

There are several new properties in MapReduce 2, identified in table 2.2.

Table 2.2 New MapReduce 2 properties

Property name	Default value	Description
mapreduce.framework.name	local	Determines which framework should be used to run MapReduce jobs. There are three possible values: ■ local, which means the LocalJobRunner is used (the entire MapReduce job is run in a single JVM). ■ classic, which means that the job will be launched on a MapReduce 1 cluster. In this case, the mapreduce.jobtracker.address property will be used to retrieve the JobTracker that the job will be submitted to. ■ yarn, which runs the MapReduce job in YARN. This can either be in a pseudo-distributed or full-blown YARN cluster.
mapreduce.job.ubertask.enable	false	Uber jobs are small jobs that can be run inside the MapReduce ApplicationMaster process to avoid the overhead of spawning map and reduce containers. Uber jobs are covered in more detail in section 2.2.6.

Table 2.2 New MapReduce 2 properties *(continued)*

Property name	Default value	Description
`mapreduce.shuffle.max.connections`	`0`	The maximum allowed connections for the shuffle. Set to `0` (zero) to indicate no limit on the number of connections. This is similar to the old (now unused) MapReduce 1 property `tasktracker.http.threads`, which defined the number of TaskTracker threads that would be used to service reducer requests for map outputs.
`yarn.resourcemanager.am.max-attempts`	`2`	The maximum number of application attempts. It's a global setting for all ApplicationMasters. Each application master can specify its individual maximum number of application attempts via the API, but the individual number can't be more than the global upper bound. If it is, the ResourceManager will override it. The default value is `2`, to allow at least one retry for AM.
`yarn.resourcemanager.recovery.enabled`	`false`	Enable RM to recover state after starting. If `true`, then `yarn.resourcemanager.store.class` must be specified. Hadoop 2.4.0 also brings in a ZooKeeper-based mechanism to store the RM state (class `org.apache.hadoop.yarn.server.resourcemanager.recovery.ZKRMStateStore`).
`yarn.resourcemanager.store.class`	`org.apache.hadoop.yarn.server.resourcemanager.recovery.FileSystemRMStateStore`	Writes ResourceManager state into a filesystem for recovery purposes.

CONTAINER PROPERTIES

Table 2.3 shows the MapReduce properties that are related to the map and reduce processes that run the tasks.

Table 2.3 MapReduce 2 properties that impact containers (map/reduce tasks)

Property name	Default value	Description
`mapreduce.map.memory.mb`	`1024`	The amount of memory to be allocated to containers (processes) that run mappers, in megabytes. The YARN scheduler uses this information to determine whether there's available capacity on nodes in a cluster. The old property name, `mapred.job.map.memory.mb`, has been deprecated.

Table 2.3 MapReduce 2 properties that impact containers (map/reduce tasks)

Property name	Default value	Description
`mapreduce.reduce.memory.mb`	`1024`	The amount of memory to be allocated to containers (processes) that run reducers, in megabytes. The YARN scheduler uses this information to determine whether there's available capacity on nodes in a cluster. The old property name, `mapreduce.reduce.memory.mb`, has been deprecated.
`mapreduce.map.cpu.vcores`	`1`	The number of virtual cores to be allocated to the map processes.
`mapreduce.reduce.cpu.vcores`	`1`	The number of virtual cores to be allocated to the reduce processes.
`mapred.child.java.opts`	`-Xmx200m`	Java options for the map and reduce processes. The `@taskid@` symbol, if present, will be replaced by the current TaskID. Any other occurrences of `@` will go unchanged. For example, to enable verbose garbage collection logging to a file named for the TaskID in /tmp and to set the heap maximum to be a gigabyte, pass a value of `-Xmx1024m -verbose:gc -Xloggc:/tmp/@taskid@.gc`. Usage of `-Djava.library.path` can cause programs to no longer function if Hadoop-native libraries are used. These values should instead be set as part of `LD_LIBRARY_PATH` in the map/reduce JVM environment using the `mapreduce.map.env` and `mapreduce.reduce.env` configuration settings.
`mapred.map.child.java.opts`	None	Map process–specific JVM arguments. The old property name, `mapred.map.child.java.opts`, has been deprecated.
`mapreduce.reduce.java.opts`	None	Reduce process–specific JVM arguments. The old property name, `mapred.reduce.child.java.opts`, has been deprecated.

CONFIGURATION NO LONGER IN EFFECT

Common properties in MapReduce 1 that are no longer in effect in MapReduce 2 are shown in table 2.4, along with explanations as to why they no longer exist.

Table 2.4 Old MapReduce 1 properties that are no longer in use

Property name	Description
`mapred.job.tracker` `mapred.job.tracker.http.address`	The JobTracker no longer exists in YARN; it's been replaced by the ApplicationMaster UI and the Job-History UI.
`mapred.task.tracker.http.address` `mapred.task.tracker.report.address`	The TaskTracker also doesn't exist in YARN—it's been replaced by the YARN NodeManager.

Table 2.4 **Old MapReduce 1 properties that are no longer in use *(continued)***

Property name	Description
`mapred.local.dir`	This used to be the local directory where intermediary data for MapReduce jobs was stored. This has been deprecated, and the new property name is `mapreduce.jobtracker.system.dir`. Its use has also been relegated to use only in the LocalJobRunner, which comes into play if you're running a local job (not on a YARN cluster).
`mapred.system.dir`	Much like `mapred.local.dir`, this is relegated to duty when running the LocalJobRunner.
`mapred.tasktracker.map.tasks.maximum` `mapred.tasktracker.reduce.tasks.maximum`	This was used to control the maximum number of map and reduce task processes that could run on a node. These were called "slots," and they were static in Hadoop 1. In Hadoop 2, YARN doesn't impose a static limit on the number of concurrent containers on a node, so these properties are no longer needed.
`mapred.job.reuse.jvm.num.tasks`	You used to be able to sequentially run multiple tasks in the same JVM, which was useful for tasks that were short-lived (and to diminish the overhead of creating a separate process per task). This is no longer supported in YARN.
`tasktracker.http.threads`	This is no longer used in MRv2. Map outputs are now fetched from a new ShuffleHandler service, which is NIO-based and is by default configured with no cap in the number of open connections (configured via `mapreduce.shuffle.max.connections`).
`io.sort.record.percent`	This shuffle property used to control how much accounting space was used in the map-side sort buffer (`io.sort.mb`). MapReduce 2 is smarter about how to fill up `io.sort.mb`.[a]

[a] "Map-side sort is hampered by io.sort.record.percent" and details can be seen at https://issues.apache.org/jira/browse/MAPREDUCE-64.

DEPRECATED PROPERTIES

Most of the MapReduce 1 (and many HDFS) properties have been deprecated in favor of property names that are better organized.[11] Currently Hadoop 2 supports both the deprecated and new property names, but it would be prudent for you to update your properties, as there's no guarantee that Hadoop 3 and later will support deprecated properties. Luckily, you get a dump of all the deprecated configuration properties on standard output when you run a MapReduce job, an example of which is shown here:

```
Configuration.deprecation: mapred.cache.files is deprecated.
Instead, use mapreduce.job.cache.files
```

[11] See the web page "Deprecated properties" at http://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/DeprecatedProperties.html for the properties that have been deprecated and their new names.

It's clear that there were quite a few changes to MapReduce properties. You may be curious to know how the rest of MapReduce changed and what parts managed to retain strong backward compatibility. Did the MapReduce APIs and binaries escape unscathed with the major version bump in Hadoop?[12]

2.2.3 Backward compatibility

Backward compatibility is an important consideration for systems with large, established user bases, as it ensures that they can rapidly move to a new version of a system with little or no change. This section covers various parts of the MapReduce system and help you determine whether you need to change your systems to be able to function on MapReduce 2.

SCRIPT COMPATIBILITY

The scripts that are bundled with Hadoop remain unchanged. This means that you can continue to use `hadoop jar ...` to launch jobs, and all other uses of the main `hadoop` script continue to work, as do the other scripts bundled with Hadoop.

CONFIGURATION

With the introduction of YARN, and MapReduce becoming an application, many MapReduce 1 property names are now deprecated in MapReduce 2, and some are no longer in effect. Section 2.2.2 covers changes to some of the more commonly used properties.

API BACKWARD COMPATIBILITY

In porting MapReduce to YARN, the developers did their best to maintain backward compatibility for existing MapReduce applications. They were able to achieve code compatibility, but in some cases weren't able to preserve binary compatibility:

- *Code compatibility* means that any MapReduce code that exists today will run fine on YARN as long as the code is recompiled. This is great, as it means that you don't need to modify your code to get it working on YARN.
- *Binary compatibility* means that MapReduce bytecode will run unchanged on YARN. In other words, you don't have to recompile your code—you can use the same classes and JARs that worked on Hadoop 1, and they'll work just fine on YARN.

Code that uses the "old" MapReduce API (`org.apache.hadoop.mapreduce package`) is binary compatible, so if your existing MapReduce code only uses the old API, you're all set—no recompilation of your code is required.

This isn't the case for certain uses of the "new" MapReduce API (`org.apache.hadoop.mapreduce`). If you use the new API, it's possible that you are using some features of the API that changed; namely, some classes were changed to interfaces. A few of these classes are as follows:

[12] Semantic versioning (http://semver.org/) permits APIs to change in ways that break backward compatibility when the major version number is incremented.

- `JobContext`
- `TaskAttemptContext`
- `Counter`

This begs the question of what to do if you're using the new MapReduce API and have code that needs to run on both versions of Hadoop.

TECHNIQUE 5 Writing code that works on Hadoop versions 1 and 2

If you're using the "new" MapReduce API and have your own Input/OutputFormat classes or use counters (to name a few operations that are not code-compatible across MapReduce versions), then you have JARs that will likely need to be recompiled to work with MapReduce 2. This is a nuisance if you have to support both MapReduce 1 and 2. You could create two sets of JARs targeting each version of MapReduce, but you would likely owe your build team several beers and end up with more complicated build and deployment systems. Or you can use the tip in this technique and continue to distribute a single JAR.

Problem

You're using MapReduce code that isn't binary compatible with MapReduce 2, and you want to be able to update your code in a way that will be compatible with both MapReduce versions.

Solution

Use a Hadoop compatibility library that works around the API differences.

Discussion

The Elephant Bird project includes a HadoopCompat class, which dynamically figures out which version of Hadoop you're running on and uses Java reflection to invoke the appropriate method calls to work with your version of Hadoop. The following code shows an example of its usage, where inside an `InputFormat` implementation, the `TaskAttemptContext` changed from a class to an interface, and the `HadoopCompat` class is being used to extract the `Configuration` object:

```
import com.alexholmes.hadooputils.util.HadoopCompat;
  import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.RecordReader;
import org.apache.hadoop.mapreduce.TaskAttemptContext;

public class MyInputFormat implements InputFormat {
  @Override
  public RecordReader createRecordReader(InputSplit split,
                                         TaskAttemptContext context)
         throws IOException {
    final Configuration conf = HadoopCompat.getConfiguration(context);
        ...
  }
}
```

Which classes changed to interfaces in Hadoop 2? Some of the notable ones are `TaskAttemptContext`, `JobContext`, and `MapContext`. Table 2.5 shows a selection of some of the methods available in the `HadoopCompat` class.

Table 2.5 Common classes and methods that are not binary compatible across MapReduce versions

Hadoop class and method	HadoopCompat call	Where you'd encounter the interface

`JobContext`
`.getConfiguration`
	`HadoopCompat`
	`.getConfiguration`
		This is probably the most commonly used class (now an interface). You'll likely bump into this interface as it's how you get to a map or reduce task's configuration.

`TaskAttemptContext`
`.setStatus`
	`HadoopCompat`
	`.setStatus`
		You'll encounter this interface if you have a custom `InputFormat`, `OutputFormat`, `RecordReader`, or `RecordWriter`.

`TaskAttemptContext`
`.getTaskAttemptID`
	`HadoopCompat`
	`.getTaskAttemptID`
		You'll use this interface if you have a custom `InputFormat`, `OutputFormat`, `RecordReader`, or `RecordWriter`.

`TaskAttemptContext`
`.getCounter`
	`HadoopCompat`
	`.getCounter`
		You'll bump into this interface if you have a custom `InputFormat`, `OutputFormat`, `RecordReader`, or `RecordWriter`.

`Counter`
`.incrementCounter`
	`HadoopCompat`
	`.incrementCounter`
		If you use counters in your jobs, you'll need to use the `HadoopCompat` call.

The `HadoopCompat` class also has a handy method called `isVersion2x`, which returns a Boolean if the class has determined that your runtime is running against version 2 of Hadoop.

This is just a sample of the methods on this class—for complete details, see the Elephant Bird project's HadoopCompat page on GitHub: https://github.com/kevinweil/elephant-bird/blob/master/hadoop-compat/src/main/java/com/twitter/elephantbird/util/HadoopCompat.java.

Maven Central contains a package with this library in it, and you can take a look at the Maven repository's page on "Elephant Bird Hadoop Compatibility" at http://mvnrepository.com/artifact/com.twitter.elephantbird/elephant-bird-hadoop-compat for an example entry you can add to your Maven file.

As you saw earlier, the main script in Hadoop 1, `hadoop`, continues to exist unchanged in Hadoop 2. In the next section you'll see how a newer version of the script should be used to run not only MapReduce jobs but also issue YARN commands.

2.2.4 Running a job

It's time to run a MapReduce 2 job. Don't worry, doing so is pretty much identical to how you did it in MapReduce 1.

TECHNIQUE 6 Using the command line to run a job

In this technique you'll learn how to use the command line to run a MapReduce job.

Problem

You want to use the YARN command line to run a MapReduce job.

Solution

Use the `yarn` command.

Discussion

In Hadoop 1, the `hadoop` command was the one used to launch jobs. This command still works for backward compatibility reasons, but the YARN form of this command is the `yarn` script, which works much like the old `hadoop` script works. As an example, this is how you'd run the `pi` job bundled in the Hadoop examples JAR:[13]

```
$ yarn jar ${HADOOP_HOME}/share/hadoop/mapreduce/*-examples-*.jar pi 2 10

Estimated value of Pi is 3.1428000
```

If you're in the habit of using `hadoop` to run your jobs, give some thought to replacing it with the `yarn` command. It's unclear whether there are plans to deprecate and remove the `hadoop` command, but you can be sure that the `yarn` equivalent is here to stay.

The ways you can launch MapReduce jobs have changed in version 2, and so has the mechanism by which you view the status and details of running and completed jobs.

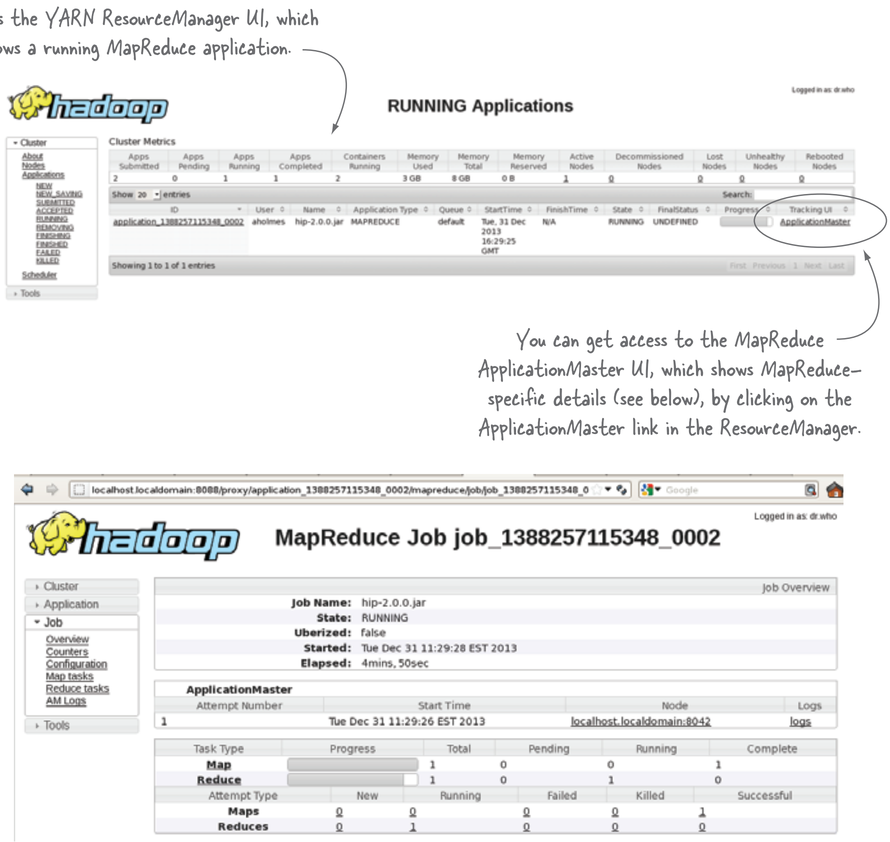
2.2.5 Monitoring running jobs and viewing archived jobs

When running MapReduce jobs, it's important for monitoring and debugging purposes to be able to view the status of a job and its tasks and to gain access to the task logs. In MapReduce 1 this would have all been carried out using the JobTracker UI, which could be used to view details on running and completed or archived jobs.

As highlighted in section 2.2.1, the JobTracker no longer exists in MapReduce 2; it has been replaced with an ApplicationMaster-specific UI, and the JobHistoryServer for completed jobs. The ApplicationMaster UI can be seen in figure 2.11. For fetching map and reduce task logs, the UI redirects to the NodeManager.

> **Figuring out where your ResourceManager UI is running** You can retrieve the host and port of the ResourceManager by examining the value of `yarn.resourcemanager.webapp.address` (or `yarn.resourcemanager.webapp.https.address` if HTTPS access is required). In the case of a pseudo-distributed installation, this will be http://localhost:8088 (or port 8090 for HTTPS). Copying the host and port into your browser is sufficient to access the UI as a URL path isn't required.

[13] This example calculates the value of pi using the quasi-Monte Carlo method.



This is the YARN ResourceManager UI, which shows a running MapReduce application.

RUNNING Applications

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	Active Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes
2	0	1	1	2	3 GB	8 GB	0 B	1	0	0	0	0

Show 20 entries Search:

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI
application_1388257115348_0002	aholmes	hip-2.0.0.jar	MAPREDUCE	default	Tue, 31 Dec 2013 16:29:25 GMT	N/A	RUNNING	UNDEFINED		ApplicationMaster

Showing 1 to 1 of 1 entries

You can get access to the MapReduce ApplicationMaster UI, which shows MapReduce-specific details (see below), by clicking on the ApplicationMaster link in the ResourceManager.

localhost.localdomain:8088/proxy/application_1388257115348_0002/mapreduce/job/job_1388257115348_0

MapReduce Job job_1388257115348_0002

Job Overview

Job Name: hip-2.0.0.jar
State: RUNNING
Uberized: false
Started: Tue Dec 31 11:29:28 EST 2013
Elapsed: 4mins, 50sec

ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	Tue Dec 31 11:29:26 EST 2013	localhost.localdomain:8042	logs

Task Type	Progress	Total	Pending	Running	Complete
Map		1	0	0	1
Reduce		1	0	1	0

Attempt Type	New	Running	Failed	Killed	Successful
Maps	0	0	0	0	1
Reduces	0	1	0	0	0

Figure 2.11 The YARN ResourceManager UI, showing applications that are currently executing

The JobHistoryServer can be seen in figure 2.12.

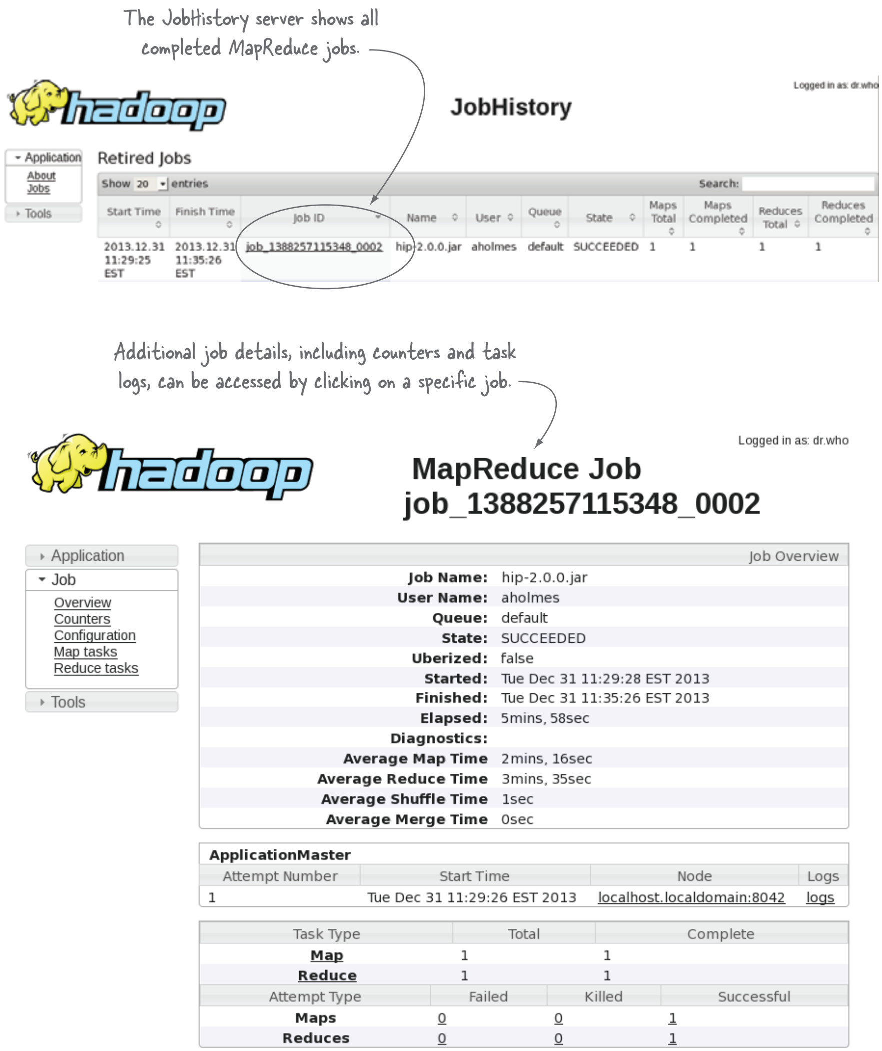
MapReduce 2 has changed how jobs are executed, configured, and monitored. It has also introduced new features, such as uber jobs, which are up next.

2.2.6 Uber jobs

When running small MapReduce jobs, the time taken for resource scheduling and process forking is often a large percentage of the overall runtime. In MapReduce 1 you didn't have any choice about this overhead, but MapReduce 2 has become smarter and can now cater to your needs to run lightweight jobs as quickly as possible.

TECHNIQUE 7 Running small MapReduce jobs

This technique looks at how you can run MapReduce jobs within the MapReduce ApplicationMaster. This is useful when you're working with a small amount of data, as you remove the additional time that MapReduce normally spends spinning up and bringing down map and reduce processes.



The JobHistory server shows all completed MapReduce jobs.

JobHistory

Retired Jobs

Show 20 entries Search:

Start Time	Finish Time	Job ID	Name	User	Queue	State	Maps Total	Maps Completed	Reduces Total	Reduces Completed
2013.12.31 11:29:25 EST	2013.12.31 11:35:26 EST	job_1388257115348_0002	hip-2.0.0.jar	aholmes	default	SUCCEEDED	1	1	1	1

Additional job details, including counters and task logs, can be accessed by clicking on a specific job.

MapReduce Job job_1388257115348_0002

Job Overview

Job Name:	hip-2.0.0.jar
User Name:	aholmes
Queue:	default
State:	SUCCEEDED
Uberized:	false
Started:	Tue Dec 31 11:29:28 EST 2013
Finished:	Tue Dec 31 11:35:26 EST 2013
Elapsed:	5mins, 58sec
Diagnostics:	
Average Map Time	2mins, 16sec
Average Reduce Time	3mins, 35sec
Average Shuffle Time	1sec
Average Merge Time	0sec

ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	Tue Dec 31 11:29:26 EST 2013	localhost.localdomain:8042	logs

Task Type	Total	Complete	
Map	1	1	
Reduce	1	1	
Attempt Type	Failed	Killed	Successful
Maps	0	0	1
Reduces	0	0	1

Figure 2.12 The JobHistory UI, showing MapReduce applications that have completed

Problem

You have a MapReduce job that operates on a small dataset, and you want to avoid the overhead of scheduling and creating map and reduce processes.

Solution

Configure your job to enable uber jobs; this will run the mappers and reducers in the same process as the ApplicationMaster.

Discussion

Uber jobs are jobs that are executed within the MapReduce ApplicationMaster. Rather than liaise with the ResourceManager to create the map and reduce containers, the

ApplicationMaster runs the map and reduce tasks within its own process and avoids the overhead of launching and communicating with remote containers.

To enable uber jobs, you need to set the following property:

```
mapreduce.job.ubertask.enable=true
```

Table 2.6 lists some additional properties that control whether a job qualities for uberization.

Table 2.6 Properties for customizing uber jobs

Property	Default value	Description
`mapreduce.job .ubertask.maxmaps`	9	The number of mappers for a job must be less than or equal to this value for the job to be uberized.
`mapreduce.job .ubertask.maxreduces`	1	The number of reducers for a job must be less than or equal to this value for the job to be uberized.
`mapreduce.job .ubertask.maxbytes`	Default block size	The total input size of a job must be less than or equal to this value for the job to be uberized.

When running uber jobs, MapReduce disables speculative execution and also sets the maximum attempts for tasks to 1.

> **Reducer restrictions** Currently only map-only jobs and jobs with one reducer are supported for uberization.

Uber jobs are a handy new addition to the MapReduce capabilities, and they only work on YARN. This concludes our look at MapReduce on YARN. Next you'll see examples of other systems running on YARN.

2.3 YARN applications

So far you've seen what YARN is, how it works, and how MapReduce 2 works as a YARN application. But this is only the first step of YARN's journey; there are already several projects that work on YARN, and over time, you should expect to see rapid growth in YARN's ecosystem.

At this point, you may be asking yourself why YARN applications are compelling and why the Hadoop community put so much work into YARN's architecture and the port of MapReduce to a YARN application. There are many reasons that we touched on at the start of the chapter, but the most important reason behind this revolutionary change in Hadoop is to open up the platform. Think about how our systems work today—gone are the days when we worked on monolithic systems; instead, we live in a world where we run multiple disparate systems in our datacenters, as shown in figure 2.13.

That's a lot of systems! And chances are that you're already running many of them in production right now. If you're an engineer, you're probably excited about having

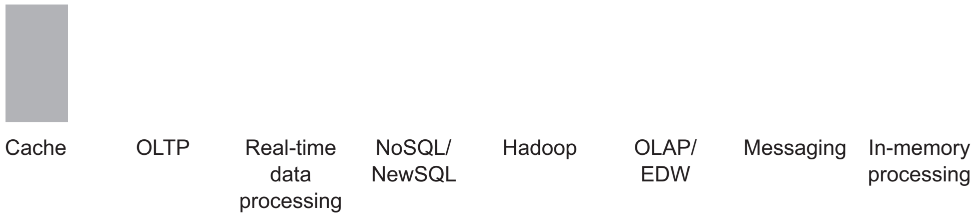


Figure 2.13 Common systems we run today. They are siloed, which complicates data and resource sharing.

all these systems in play, but systems administrators and architects get migraines thinking about the challenges that supporting all these systems brings:

- They have to build the in-house knowledge to administer and keep the systems up and healthy. Systems fail, especially complicated distributed systems, and being open source, many of these systems don't have the tooling to facilitate easy management.
- Data exchange between systems is painful, primarily due to the volume of data and the lack of tooling for the data movement. Large, expensive projects ensue.[14]
- Each system has to solve the same distributed problems, such as fault tolerance, distributed storage, log handling, and resource scheduling.

YARN promises a single cluster that can have its resources managed in a uniform way, support multi-tenant applications and users, and offer elastic computation over shared storage. HBase coupled with Hoya gives us a sneak peek at what the future could look like: strong data locality properties are used for efficient movement of data in and out of HBase; and Hoya, with its YARN integration, provides elastic, on-demand computing, with the ability to run multiple HBase clusters on a single YARN cluster.

In the following sections, you'll be introduced to several systems across a broad spectrum of technologies that are built on YARN. We'll look at one or more examples of these technologies that have been built with YARN compatibility.

2.3.1 NoSQL

NoSQL covers a wide array of technologies, but, in short, they're systems that provide real-time CRUD operations in a way that doesn't hold ACID properties sacred. These systems were created to work around the shortcomings of monolithic OLAP systems, which impeded the ability of system architectures to scale out and provide responsive services.

There are many NoSQL systems out there, but none have been more integrated with Hadoop than HBase. Even prior to YARN, the goal of HBase was to use HDFS for

[14] LinkedIn addresses this with a "data plane" architecture highlighted in Jay Kreps' blog post, "The Log: What every software engineer should know about real-time data's unifying abstraction"—take a look at the "unified log" image and the surrounding text for an architectural solution to help reduce these pain points: http://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying.

its storage, and HBase benefited from close integration with MapReduce, allowing for batch-processing facilities that often eluded its competitors.

YARN solves two challenges for HBase. HBase and MapReduce 1 coexisting on a cluster brought resource management challenges, as there were no easy ways to guarantee SLAs to both systems. YARN capitalizes on cgroups in Linux, which provide concurrently executing processes with guaranteed access to their required resources. The second opportunity that YARN gave HBase was the ability to run multiple HBase clusters on the same Hadoop cluster. This support is being carried out in a project called Hoya, short for HBase on Yarn.

2.3.2 Interactive SQL

Up until recently, running SQL on Hadoop has been an exercise in patience—kick up your Hive shell, enter your query, and wait, often minutes, until you get a result.[15] Data scientists and analysts would likely not find this to be the most conducive environment for quickly probing and experimenting with data.

There have been several initiatives to work around this issue. Cloudera's solution was to create the Impala project, which bypasses MapReduce altogether and operates by running its own daemon on each slave node in your cluster (colocated with the HDFS slave daemon, the DataNode, for data locality). To help with multi-tenancy on YARN clusters, Cloudera has developed Llama (http://cloudera.github.io/llama/), which aims to work with YARN in such a way that YARN understands the resources that the Impala daemons are utilizing on a cluster.

Hortonworks has taken a different approach—they've focused on making improvements to Hive and have made significant steps toward making Hive more interactive. They've combined their improvements under a project called Stinger (http://hortonworks.com/labs/stinger/), and the most significant change involves bypassing MapReduce and using Tez, a YARN DAG processing framework, to execute their work.

Apache Drill is another SQL-on-Hadoop solution that promises the ability to work over many persistent stores, such as Cassandra and MongoDB. They have an open ticket to add YARN support to the project (https://issues.apache.org/jira/browse/DRILL-142).

Facebook Presto is also in the SQL-on-Hadoop camp, but so far there's no word on whether there will be YARN support.

2.3.3 Graph processing

Modern graph-processing systems allow distributed graph algorithms to execute against large graphs that contain billions of nodes and trillions of edges. Graph operations using traditional MapReduce typically result in one job per iteration,[16] which is

[15] The reason Hive queries used to take a long time is that they would be translated to one or more MapReduce jobs, so job startup times (coupled with writing intermediary outputs to and from disk) resulted in long query times.

[16] Giraph in its MapReduce 1 implementation works around this by using long-running map tasks that exchange state with ZooKeeper and pass messages to each other.

slow and cumbersome, as it requires the entire graph data structure to be serialized to and from disk on each iteration.

Apache Giraph, a popular graph-processing project, has worked on Hadoop since version 1 and earlier, and the committers have also updated Giraph so that it runs as a native YARN application.

Apache Hama also has some graph-processing capabilities on YARN.

2.3.4 Real-time data processing

Real-time data processing systems are computational systems that work on unbounded streams of data. The features of these systems are similar to those of MapReduce, as they allow operations such as filtering, projection, joins, and aggregations. A typical use of these systems is to process real-time events occurring in a system, perform some aggregations, and then push the results out to a NoSQL store for retrieval by another system.

Arguably, the real-time data processing system with most traction at the time of writing is Apache Storm, originally built by Nathan Marz, which is a key part of his Lambda Architecture.[17] To bring Storm to YARN, Yahoo has created a project called storm-yarn. This project offers several advantages—not only will this allow multiple Storm clusters to run on YARN, but it promises elasticity for Storm clusters: the ability to quickly provision additional resources for Storm. More details on the project can be seen at https://github.com/yahoo/storm-yarn.

Spark Streaming is another notable real-time data processing project developed as an extension to the Spark API, and it supports consuming data sources such as HDFS, Kafka, Flume, and more. Spark is also supported on YARN. Spark Streaming may become a strong competitor for Storm, notably because once you master Spark, you also know how to do Spark Streaming, and vice versa. This means you have a single programming paradigm for both offline and real-time data analysis.

Other real-time data processing systems with YARN integration are Apache S4, Apache Samza (which came out of LinkedIn), and DataTorrent.

2.3.5 Bulk synchronous parallel

Bulk synchronous parallel (BSP) is a distributed processing method whereby multiple parallel workers independently work on a subset of an overall problem, after which they exchange data among themselves and then use a global synchronization mechanism to wait for all workers to complete before repeating the process. Google Pregel published how their graph processing framework is inspired by BSP, and Apache Giraph uses a similar BSP model for graph iteration.

Apache Hama is a general-purpose BSP implementation that can work on YARN. It also has built-in graph-processing capabilities.

[17] The Lambda Architecture plays to the strengths of batch and real-time. Read more in Nathan Marz's book, *Big Data* (Manning, 2014).

2.3.6 MPI

MPI (Message Passing Interface) is a mechanism that allows messages to be exchanged on clusters of hosts. Open MPI is an open source MPI implementation. There's currently an open ticket to complete work on integrating Open MPI support into Hadoop (https://issues.apache.org/jira/browse/MAPREDUCE-2911). The work that has been completed so far for this integration is in mpich2-yarn at https://github.com/clarkyzl/mpich2-yarn.

2.3.7 In-memory

In-memory computing uses the ever-increasing memory footprint in our systems to quickly perform computing activities such as iterative processing and interactive data mining.

Apache Spark is a popular example that came out of Berkeley. It's a key part of an overall set of solutions that also includes Shark for SQL operations and GraphX for graph processing. Cloudera's CDH5 distribution includes Spark running on YARN.

For additional details on how to run Spark on YARN, see Spark's "Launching Spark on YARN" page at http://spark.apache.org/docs/0.9.0/running-on-yarn.html.

2.3.8 DAG execution

Directed Acyclic Graph (DAG) execution engines allow you to model data-processing logic as a DAG and then execute it in parallel over a large dataset.

Apache Tez is an example of a DAG execution engine; it was born out of the need to provide a more generalized MapReduce system that would preserve the parallelism and throughput of MapReduce, and at the same time support additional processing models and optimizations beyond that which MapReduce provides. Examples of Tez's abilities include not imposing a specific data model, so that both the key/value model of MapReduce, as well as the tuple-based models of Hive and Pig, can be supported.

Tez provides a number of advantages over MapReduce, which include eliminating replicated write barriers that exist in MapReduce between multiple jobs—a major performance bottleneck for systems like Hive and Pig. Tez can also support reduce operations without the sorting overhead that MapReduce requires, resulting in more efficient pipelines where sorting isn't necessary for the application. Tez also supports sophisticated operations such as Map-Map-Reduce, or any arbitrary graph of operations, freeing up developers to more naturally express their data pipelines. Tez can also be used to make dynamic data flow choices when executing—for example, based on the size of intermediary data in your flow, you may decide to store it in memory or in HDFS or local disk.

The upshot of all of this is that Tez can shake off the batch-only shackles of MapReduce and support interactive use cases. As an example, the original scope of Tez is a large step in Hortonworks' goal of making Hive interactive—moving from MapReduce to Tez is a key part of that work.

2.4 Chapter summary

Hadoop version 2 turns the old way work has been done in Hadoop upside down. No longer are you limited to running MapReduce on your clusters. This chapter covered the essentials that you need to get going with YARN. You looked at why YARN is important in Hadoop, saw a high-level overview of the architecture, and learned about some of the salient YARN configuration properties that you'll need to use.

The advent of YARN has also introduced significant changes in how MapReduce works. MapReduce has been ported into a YARN application, and in section 2.2 you saw how MapReduce executes on Hadoop 2, learned what configuration properties have changed, and also picked up some new features, such as uber jobs.

The last section of this chapter covered some exciting examples of up-and-coming YARN applications to give you a sense of what capabilities you should expect to be able to unleash on your YARN cluster. For additional YARN coverage, feel free to skip ahead to chapter 10 and look at how to develop your very own YARN application!

Now that you understand the lay of the land with YARN, it's time to move on to look at data storage in Hadoop. The focus of the next chapter is on working with common file formats such as XML and JSON, as well as picking file formats better suited for life in Hadoop, such as Parquet and Avro.

DATABASES/PROGRAMMING

Hadoop IN PRACTICE Second Edition

Alex Holmes

It's always a good time to upgrade your Hadoop skills! Hadoop in Practice, Second Edition provides a collection of 104 tested, instantly useful techniques for analyzing real-time streams, moving data securely, machine learning, managing large-scale clusters, and taming big data using Hadoop.

This completely revised second edition covers changes and new features in Hadoop core, including MapReduce 2 and YARN. You'll pick up hands-on best practices for integrating Spark, Kafka, and Impala with Hadoop, and get new and updated techniques for the latest versions of Flume, Sqoop, and Mahout. In short, this is the most practical, up-to date coverage of Hadoop available.

What's Inside

- Thoroughly updated for Hadoop 2
- How to write YARN applications
- Integrate real-time technologies like Storm, Impala, and Spark
- Predictive analytics using Mahout and RR

Readers need to know a programming language like Java and have basic familiarity with Hadoop.

Alex Holmes works on tough big data problems. He is a software engineer, author, speaker, and blogger specializing in large-scale Hadoop projects.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/HadoopinPracticeSecondEdition

Free eBook
SEE INSERT

"Very insightful. A deep dive into the Hadoop world."
—Andrea Tarocchi, Red Hat, Inc.

"The most complete material on Hadoop and its ecosystem known to mankind!"
—Arthur Zubarev, Vital Insights

"Clear and concise, full of insights and highly applicable information."
—Edward de Oliveira Ribeiro
DataStax, Inc.

"Comprehensive up-to-date coverage of Hadoop 2."
—Muthusamy Manigandan
OzoneMedia

ISBN 13: 978-1-617292-22-4
ISBN 10: 1-61729-222-2
54999
9 781617 292224

MANNING

$49.99 / Can $52.99 [INCLUDING eBOOK]