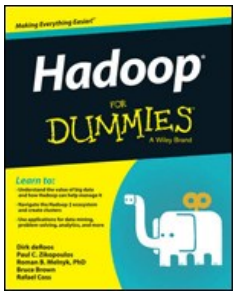

Hadoop for Dummies

by Dirk deRoos et al.

Chapter 7: Frameworks for Processing Data in Hadoop—YARN and MapReduce

In This Chapter

- Examining distributed data processing in Hadoop
- Looking at MapReduce execution
- Venturing into YARN architecture
- Anticipating future directions for data processing on Hadoop

My, how time flies. If we had written this book a year (well, a few months) earlier, this chapter on data processing would have talked only about MapReduce, for the simple reason that MapReduce was then the only way to process data in Hadoop. With the release of Hadoop 2, however, YARN was introduced, ushering in a whole new world of data processing opportunities.

YARN stands for Yet Another Resource Negotiator — a rather modest label considering its key role in the Hadoop ecosystem. (The Yet Another label is a long-running gag in computer science that celebrates programmers' propensity to be lazy about feature names.) A (Hadoop-centric) thumbnail sketch would describe YARN as a tool that enables other data processing frameworks to run on Hadoop. A more substantive take on YARN would describe it as a general-purpose resource management facility that can schedule and assign CPU cycles and memory (and in the future, other resources, such as network bandwidth) from the Hadoop cluster to waiting applications.

REMEMBER At the time of this writing, only batch-mode MapReduce applications were supported in production. A number of additional application frameworks being ported to YARN are in various stages of development, however, and many of them will soon be production ready.

For us authors, as Hadoop enthusiasts, YARN raises exciting possibilities. Singlehandedly, YARN has converted Hadoop from simply a batch processing engine into a platform for many different modes of data processing, from traditional batch to interactive queries to streaming analysis.

Running Applications before Hadoop 2

Because many existing Hadoop deployments still aren't yet using YARN, we take a quick look at how Hadoop managed its data processing before the days of Hadoop 2. We concentrate on the role that JobTracker master daemons and TaskTracker slave daemons played in handling MapReduce processing.

Before tackling the daemons, however, let us back up and remind you that the whole point of employing distributed systems is to be able to deploy computing resources in a network of self-contained computers in a manner that's fault-tolerant, easy, and inexpensive. In a distributed system such as Hadoop, where you have a cluster of self-contained compute nodes all working in parallel, a great deal of complexity goes into ensuring that all the pieces work together. As such, these systems typically have distinct layers to handle different tasks to support parallel data processing. This concept, known as the *separation of concerns*, ensures that if you are, for example, the application programmer, you don't need to worry about the specific details for, say, the failover of map tasks. In Hadoop, the system consists of these four distinct layers, as shown in Figure 7-1:

- **Distributed storage**: The Hadoop Distributed File System (HDFS) is the storage layer where the data, interim results, and final result sets are stored.
- **Resource management**: In addition to disk space, all slave nodes in the Hadoop cluster have CPU cycles, RAM, and network bandwidth. A system such as Hadoop needs to be able to parcel out these resources so that multiple applications and users can share the cluster in predictable and tunable ways. This job is done by the JobTracker daemon.
- **Processing framework**: The MapReduce process flow defines the execution of all applications in Hadoop 1. As we saw in Chapter 6, this begins with the map phase; continues with aggregation with shuffle, sort, or merge; and ends with the reduce phase. In Hadoop 1, this is also managed by the JobTracker daemon, with local execution being managed by TaskTracker daemons running on the slave nodes.
- **Application Programming Interface (API)**: Applications developed for Hadoop 1 needed to be coded using the MapReduce API. In Hadoop 1, the Hive and Pig projects provide programmers with easier interfaces for writing Hadoop applications, and underneath the hood, their code compiles down to MapReduce.

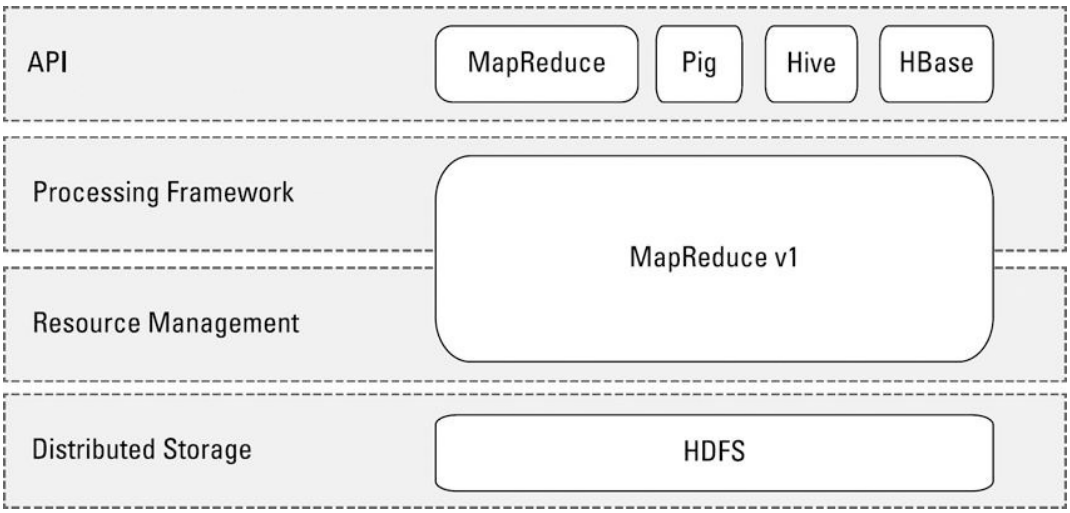

Figure 7-1: Hadoop 1 data processing architecture

REMEMBER In the world of Hadoop 1 (which was the only world we had until quite recently), all data processing revolved around MapReduce.

Tracking JobTracker

MapReduce processing in Hadoop 1 is handled by the JobTracker and TaskTracker daemons. The JobTracker maintains a view of all available processing resources in the Hadoop cluster and, as application requests come in, it schedules and deploys them to the TaskTracker nodes for execution. As applications are running, the JobTracker receives status updates from the TaskTracker nodes to track their progress and, if necessary, coordinate the handling of any failures. The JobTracker needs to run on a master node in the Hadoop cluster as it coordinates the execution of all MapReduce applications in the cluster, so it's a mission-critical service.

Tracking TaskTracker

An instance of the TaskTracker daemon runs on every slave node in the Hadoop cluster, which means that each slave node has a service that ties it to the processing (TaskTracker) and the storage (DataNode), which enables Hadoop to be a distributed system. As a slave process, the TaskTracker receives processing requests from the JobTracker. Its primary responsibility is to track the execution of MapReduce workloads happening locally on its slave node and to send status updates to the JobTracker.

TaskTrackers manage the processing resources on each slave node in the form of processing slots — the slots defined for map tasks and reduce tasks, to be exact. The total number of map and reduce slots indicates how many map and reduce tasks can be executed at one time on the slave node.

REMEMBER When it comes to tuning a Hadoop cluster, setting the optimal number of map and reduce slots is critical. The number of slots needs to be carefully configured based on available memory, disk, and CPU resources on each slave node. Memory is the most critical of these three resources from a performance perspective. As such, the total number of task slots needs to be balanced with the maximum amount of memory allocated to the Java heap size. Keep in mind that every map and reduce task spawns its own Java virtual machine (JVM) and that the heap represents the amount of memory that's allocated for each JVM. The ratio of map slots to reduce slots is also an important consideration. For example, if you have too many map slots and not enough reduce slots for your workloads, map slots will tend to sit idle, while your jobs are waiting for reduce slots to become available.

Distinct sets of slots are defined for map tasks and reduce tasks because they use computing resources quite differently. Map tasks are assigned based on data locality, and they depend heavily on disk I/O and CPU. Reduce tasks are assigned based on availability, not on locality, and they depend heavily on network bandwidth because they need to receive output from map tasks.

Launching a MapReduce Application

To see how the JobTracker and TaskTracker work together to carry out a MapReduce action, take a look at the execution of a MapReduce application, as shown in Figure 7-2. The figure shows the interactions, and the following step list lays out the play-by-play:

1. The client application submits an application request to the JobTracker.
2. The JobTracker determines how many processing resources are needed to execute the entire application. This is done by requesting the locations and names of the files and data blocks that the application needs from the NameNode, and calculating how many map tasks and reduce tasks will be needed to process all this data.
3. The JobTracker looks at the state of the slave nodes and queues all the map tasks and reduce tasks for execution.
4. As processing slots become available on the slave nodes, map tasks are deployed to the slave nodes. Map tasks assigned to specific blocks of data are assigned to nodes where that same data is stored.

5. The JobTracker monitors task progress, and in the event of a task failure or a node failure, the task is restarted on the next available slot. If the same task fails after four attempts (which is a default value and can be customized), the whole job will fail.
6. After the map tasks are finished, reduce tasks process the interim result sets from the map tasks.
7. The result set is returned to the client application.

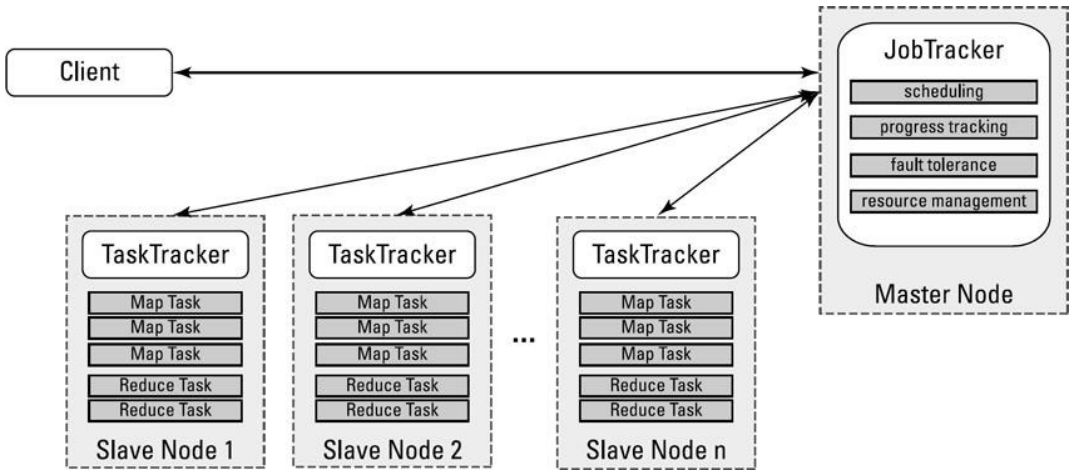

Figure 7-2: Hadoop 1 daemons and application execution

REMEMBER More complicated applications can have multiple rounds of map/reduce phases, where the result of one round is used as input for the second round. This is quite common with SQL-style workloads, where there are, for example, join and group-by operations.

Seeing a World beyond MapReduce

MapReduce has been (and continues to be) a successful batch-oriented programming model. You need look no further than the wide adoption of Hadoop to recognize the truth of this statement. But Hadoop itself has been hitting a glass ceiling in terms of wider use. The most significant factor in this regard has been Hadoop's exclusive tie to MapReduce, which means that it could be used only for batch-style workloads and for general-purpose analysis. Hadoop's success has created demand for additional data processing modes: graph analysis, for example, or streaming data processing or message passing. To top it all off, demand is growing for real-time and ad-hoc analysis, where analysts ask many smaller questions against subsets of the data and need a near-instant response. This approach, which is what analysts are accustomed to using with relational databases, is a significant departure from the kind of batch processing Hadoop can currently support.

When you start noticing a technology's limitations, you're reminded of all its other little quirks that bother you, such as Hadoop 1's restrictions around scalability — the limitation of the number of data blocks that the NameNode could track, for example. (See Chapter 4 for more on these — and other — restrictions.) The JobTracker also has a practical limit to the amount of processing resources and running tasks it can track – this (like the NameNode's limitations) is between 4,000 and 5,000 nodes.

And finally, to the extent that Hadoop could support different kinds of workloads other than MapReduce — largely with HBase and other third-party services running on slave nodes — there was no easy way to handle competing requests for limited resources.

Where there's a will, there's often a way, and the will to move beyond the limitations of a Hadoop 1/MapReduce world led to a way out — the YARN way.

Scouting out the YARN Architecture

YARN, for those just arriving at this particular party, stands for Yet Another Resource Negotiator, a tool that enables other data processing frameworks to run on Hadoop. The glory of YARN is that it presents Hadoop with an elegant solution to a number of longstanding challenges, many of which are outlined in some detail in the previous section. If you can't be bothered to reread that section, just know that YARN is meant to provide a more efficient and flexible workload scheduling as well as a resource management facility, both of which will ultimately enable Hadoop to run more than just MapReduce jobs.

Figure 7-3 shows in general terms how YARN fits into Hadoop and also makes clear how it has enabled Hadoop to become a truly general-purpose platform for data processing. The following list gives the lyrics to the melody — and it wouldn't hurt to compare Figure 7-3 with Figure 7-1:

- **Distributed storage**: Nothing has changed here with the shift from MapReduce to YARN — HDFS is still the storage layer for Hadoop.
- **Resource management**: The key underlying concept in the shift to YARN from Hadoop 1 is decoupling resource management from data processing. This enables YARN to provide resources to any processing framework written for Hadoop, including MapReduce.
- **Processing framework**: Because YARN is a general-purpose resource management facility, it can allocate cluster resources to any data processing framework written for Hadoop. The processing framework then handles application runtime issues. To maintain compatibility for all the code that was developed for Hadoop 1, MapReduce serves as the first framework available for use on YARN. At the time of this

writing, the Apache Tez project was an incubator project in development as an alternative framework for the execution of Pig and Hive applications. Tez will likely emerge as a standard Hadoop configuration.

- **Application Programming Interface (API)**: With the support for additional processing frameworks, support for additional APIs will come. At the time of this writing, Hoya (for running HBase on YARN), Apache Giraph (for graph processing), Open MPI (for message passing in parallel systems), Apache Storm (for data stream processing) are in active development.

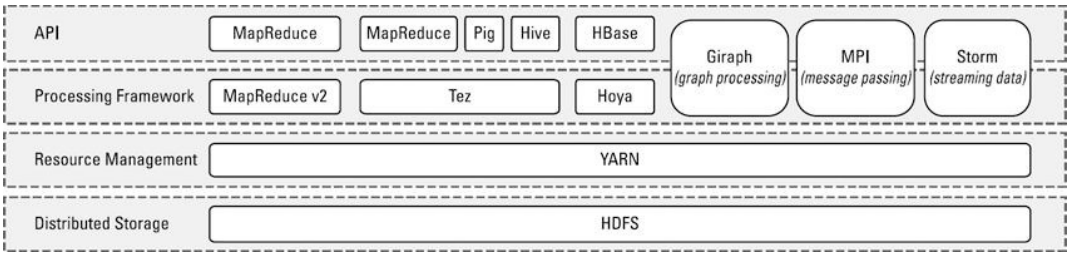

Figure 7-3: Hadoop data processing architecture with YARN

YARN's Resource Manager

The core component of YARN is the Resource Manager, which governs all the data processing resources in the Hadoop cluster. Simply put, the Resource Manager is a dedicated scheduler that assigns resources to requesting applications. Its only tasks are to maintain a global view of all resources in the cluster, handling resource requests, scheduling the request, and then assigning resources to the requesting application.

The Resource Manager, a critical component in a Hadoop cluster, should run on a dedicated master node.

Even though the Resource Manager is basically a pure scheduler, it relies on scheduler modules for the actual scheduling logic. You can choose from the same schedulers that were available in Hadoop 1, which have all been updated to work with YARN: FIFO (first in, first out), Capacity, or Fair Share. We'll discuss these schedulers in greater detail in Chapter 17.

The Resource Manager is completely agnostic with regard to both applications and frameworks — it doesn't have any dogs in those particular hunts, in other words. It has no concept of map or reduce tasks, it doesn't track the progress of jobs or their individual tasks, and it doesn't handle failovers. In short, the Resource Manager is a complete departure from the JobTracker daemon we looked at for Hadoop 1 environments. What the Resource Manager does do is schedule workloads, and it does that job well. This high degree of separating duties — concentrating on one aspect while ignoring everything else — is exactly what makes YARN much more scalable, able to provide a generic platform for applications, and able to support a *multi-tenant* Hadoop cluster — multi-tenant because different business units can share the same Hadoop cluster.

YARN's Node Manager

Each slave node has a Node Manager daemon, which acts as a slave for the Resource Manager. As with the TaskTracker, each slave node has a service that ties it to the processing service (Node Manager) and the storage service (DataNode) that enable Hadoop to be a distributed system. Each Node Manager tracks the available data processing resources on its slave node and sends regular reports to the Resource Manager.

The processing resources in a Hadoop cluster are consumed in bite-size pieces called containers. A *container* is a collection of all the resources necessary to run an application: CPU cores, memory, network bandwidth, and disk space. A deployed container runs as an individual process on a slave node in a Hadoop cluster.

REMEMBER The concept of a container may remind you of a *slot*, the unit of processing used by the JobTracker and TaskTracker, but they have some notable differences. Most significantly, containers are generic and can run whatever application logic they're given, unlike slots, which are specifically defined to run either map or reduce tasks. Also, containers can be requested with custom amounts of resources, while slots are all uniform. As long as the requested amount is within the minimum and maximum bounds of what's acceptable for a container (and as long as the requested amount of memory is a multiple of the minimum amount), the Resource Manager will grant and schedule that container.

All container processes running on a slave node are initially provisioned, monitored, and tracked by that slave node's Node Manager daemon.

YARN's Application Master

Unlike the YARN components we've described already, no component in Hadoop 1 maps directly to the Application Master. In essence, this is work that the JobTracker did for every application, but the implementation is radically different. Each application running on the Hadoop cluster has its own, dedicated Application Master instance, which actually runs in a container process on a slave node (as compared to the JobTracker, which was a single daemon that ran on a master node and tracked the progress of all applications).

Throughout its life (for example, while the application is running), the Application Master sends heartbeat messages to the Resource Manager with its status and the state of the application's resource needs. Based on the results of the Resource Manager's scheduling, it assigns *container resource leases* — basically reservations for the resources containers need — to the Application Master on specific slave nodes.

The Application Master oversees the full lifecycle of an application, all the way from requesting the needed containers from the Resource Manager to submitting container lease requests to the NodeManager.

Each application framework that's written for Hadoop must have its own Application Master implementation. MapReduce, for example, has a specific Application Master that's designed to execute map tasks and reduce tasks in sequence.

Job History Server

The Job History Server is another example of a function that the JobTracker used to handle, and it has been siphoned off as a self-contained daemon. Any client requests for a job history or the status of current jobs are served by the Job History Server.

Launching a YARN-based Application

To show how the various YARN components work together, we walk you through the execution of an application. For the sake of argument, it can be a MapReduce application, such as the one we describe earlier in this chapter, with the JobTracker and TaskTracker architecture. Just remember that, with YARN, it can be any kind of application for which there's an application framework. Figure 7-4 shows the interactions, and the prose account is set down in the following step list:

1. The client application submits an application request to the Resource Manager.
2. The Resource Manager asks a Node Manager to create an Application Master instance for this application. The Node Manager gets a container for it and starts it up.
3. This new Application Master initializes itself by registering itself with the Resource Manager.
4. The Application Master figures out how many processing resources are needed to execute the entire application. This is done by requesting from the NameNode the names and locations of the files and data blocks the application needs and calculating how many map tasks and reduce tasks are needed to process all this data.
5. The Application Master then requests the necessary resources from the Resource Manager. The Application Master sends heartbeat messages to the Resource Manager throughout its lifetime, with a standing list of requested resources and any changes (for example, a kill request).
6. The Resource Manager accepts the resource request and queues up the specific resource requests alongside all the other resource requests that are already scheduled.
7. As the requested resources become available on the slave nodes, the Resource Manager grants the Application Master leases for containers on specific slave nodes.
8. The Application Master requests the assigned container from the Node Manager and sends it a Container Launch Context (CLC). The CLC includes everything the application task needs in order to run: environment variables, authentication tokens, local resources needed at runtime (for example, additional data files, or application logic in JARs), and the command string necessary to start the actual process. The Node Manager then creates the requested container process and starts it.
9. The application executes while the container processes are running. The Application Master monitors their progress, and in the event of a container failure or a node failure, the task is restarted on the next available slot. If the same task fails after four attempts (a default value which can be customized), the whole job will fail. During this phase, the Application Master also communicates directly with the client to respond to status requests.
10. Also, while containers are running, the Resource Manager can send a kill order to the Node Manager to terminate a specific container. This can be as a result of a scheduling priority change or a normal operation, such as the application itself already being completed.
11. In the case of MapReduce applications, after the map tasks are finished, the Application Master requests resources for a round of reduce tasks to process the interim result sets from the map tasks.
12. When all tasks are complete, the Application Master sends the result set to the client application, informs the Resource Manager that the application has successfully completed, deregisters itself from the Resource Manager, and shuts itself down.

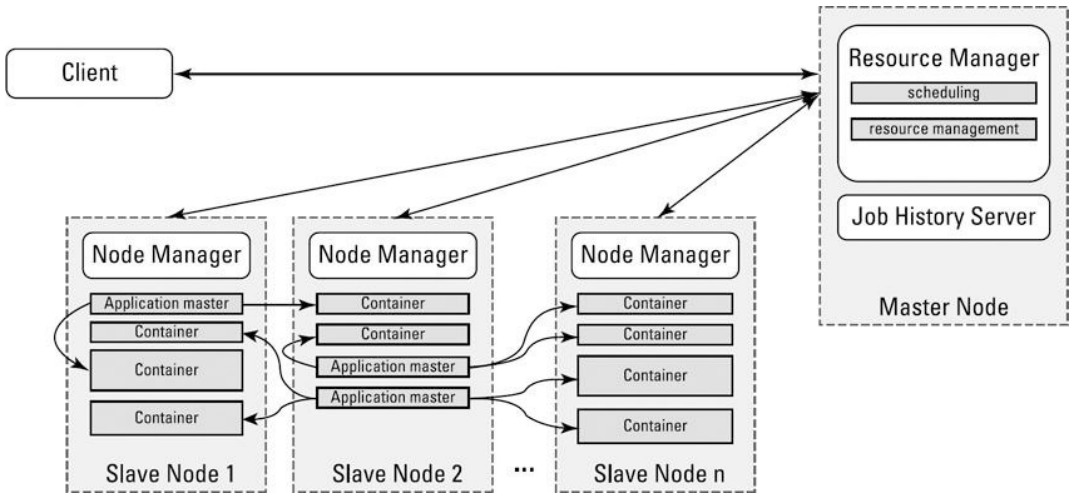

Figure 7-4: YARN daemons and application execution

Like the JobTracker and TaskTracker daemons and processing slots in Hadoop 1, all of the YARN daemons and containers are Java processes, running in JVMs. With YARN, you're no longer required to define how many map and reduce slots you need — you simply decide how much memory map and reduce tasks can have. The Resource Manager will allocate containers for map or reduce tasks on the cluster based on how much memory is available.

In this section, we have described what happens underneath the hood when applications run on YARN. When you're writing Hadoop applications, you don't need to worry about requesting resources and monitoring containers. Whatever application framework you're using does all that for you. It's always a good idea, however, to understand what goes on when your applications are running on the cluster. This knowledge can help you immensely when you're monitoring application progress or debugging a failed task.

Real-Time and Streaming Applications

The process flow we describe in our coverage of YARN looks an awful lot like a framework for batch execution. You might wonder, "What happened to this idea of flexibility for different modes of applications?" Well, the only application framework that was ready for production use at the time of this writing was MapReduce. Soon, the Apache Tez and Apache Storm will be ready for production use, and you can use Hadoop for more than just batch processing.

Tez, for example, will support *real-time* applications — an interactive kind of application where the user expects an immediate response. One design goal of Tez is to provide an interactive facility for users to issue Hive queries and receive a result set in just a few seconds or less.

Another example of a non-batch type of application is Storm, which can analyze streaming data. This concept is completely different from either MapReduce or Tez, both of which operate against data that is already persisted to disk — in other words, data at rest. Storm processes data that hasn't yet been stored to disk — more specifically, data that's streaming into an organization's network. It's data in motion, in other words.

In both cases, the interactive and streaming-data processing goals wouldn't work if Application Masters need to be instantiated, along with all the required containers, like we described in the steps involved in running a YARN application. What YARN allows here is the concept of an ongoing service (a session), where there's a dedicated Application Master that stays alive, waiting to coordinate requests. The Application Master also has open leases on reusable containers to execute any requests as they arrive.