Apache Spark Resource Management and YARN App Models

- by Sandy Ryza
- May 30, 2014
- 7 comments
- http://blog.cloudera.com/blog/2014/05/apache-spark-resource-management-and-yarn-app-models/

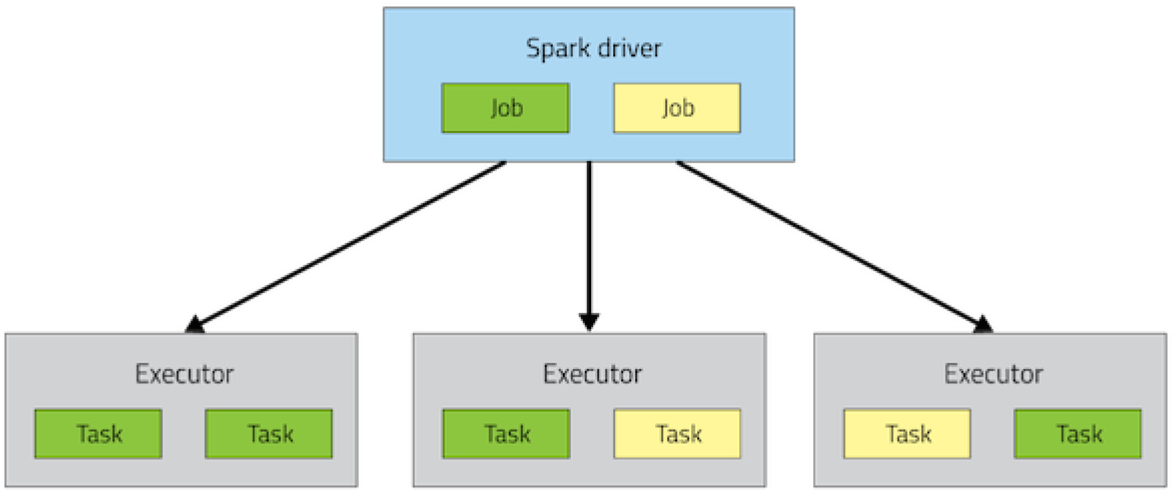
A concise look at the differences between how Spark and MapReduce manage cluster resources under YARN

The most popular Apache YARN application after MapReduce itself is Apache Spark. At Cloudera, we have worked hard to stabilize Spark-on-YARN (SPARK-1101), and CDH 5.0.0 added support for Spark on YARN clusters.

In this post, you’ll learn about the differences between the Spark and MapReduce architectures, why you should care, and how they run on the YARN cluster ResourceManager.

Applications

In MapReduce, the highest-level unit of computation is a job. The system loads the data, applies a map function, shuffles it, applies a reduce function, and writes it back out to persistent storage. Spark has a similar job concept (although a job can consist of more stages than just a single map and reduce), but it also has a higher-level construct called an “application,” which can run multiple jobs, in sequence or in parallel.



Spark application architecture

For those familiar with the Spark API, an application corresponds to an instance of the SparkContext class. An application can be used for a single batch job, an interactive session with multiple jobs spaced apart, or a long-lived server continually satisfying requests. Unlike MapReduce, an application will have processes, called *Executors*, running on the cluster on its behalf even when it’ s not running any jobs. This approach enables data storage in memory for quick access, as well as lightning-fast task startup time.

Executors

MapReduce runs each task in its own process. When a task completes, the process goes away. In Spark, many tasks can run concurrently in a single process, and this process sticks around for the lifetime of the Spark application, even when no jobs are running.

The advantage of this model, as mentioned above, is speed: Tasks can start up very quickly and process in-memory data. The disadvantage is coarser-grained resource management. As the number of executors for an app is fixed and each executor has a fixed allotment of resources, an app takes up the same amount of resources for the full duration that it’ s running. (When YARN supports container resizing, we plan to take advantage of it in Spark to acquire and give back resources dynamically.)

Active Driver

To manage the job flow and schedule tasks Spark relies on an active driver process. Typically, this driver process is the same as the client process used to initiate the job, although in YARN mode (covered later), the driver can run on the cluster. In contrast, in MapReduce, the client process can go away and the job can continue running. In Hadoop 1.x, the JobTracker was responsible for task scheduling, and in Hadoop 2.x, the MapReduce application master took over this responsibility.

Pluggable Resource Management

Spark supports pluggable cluster management. The cluster manager is responsible for starting executor processes. Spark application writers do not need to worry about what cluster manager against which Spark is running.

Spark supports YARN, Mesos, and its own “standalone” cluster manager. All three of these frameworks have two components. A central master service (the YARN ResourceManager, Mesos master, or Spark standalone master) decides which applications get to run executor processes, as well as where and when they get to run. A slave service running on every node (the YARN NodeManager, Mesos slave, or Spark standalone slave) actually starts the executor processes. It may also monitor their liveliness and resource consumption.

Why Run on YARN?

Using YARN as Spark’s cluster manager confers a few benefits over Spark standalone and Mesos:

- YARN allows you to dynamically share and centrally configure the same pool of cluster resources between all frameworks that run on YARN. You can throw your entire cluster at a MapReduce job, then use some of it on an Impala query and the rest on Spark application, without any changes in configuration.
- You can take advantage of all the features of YARN schedulers for categorizing, isolating, and prioritizing workloads.
- Spark standalone mode requires each application to run an executor on every node in the cluster, whereas with YARN, you choose the number of executors to use.
- Finally, YARN is the only cluster manager for Spark that supports security. With YARN, Spark can run against Kerberized

Hadoop clusters and uses secure authentication between its processes.

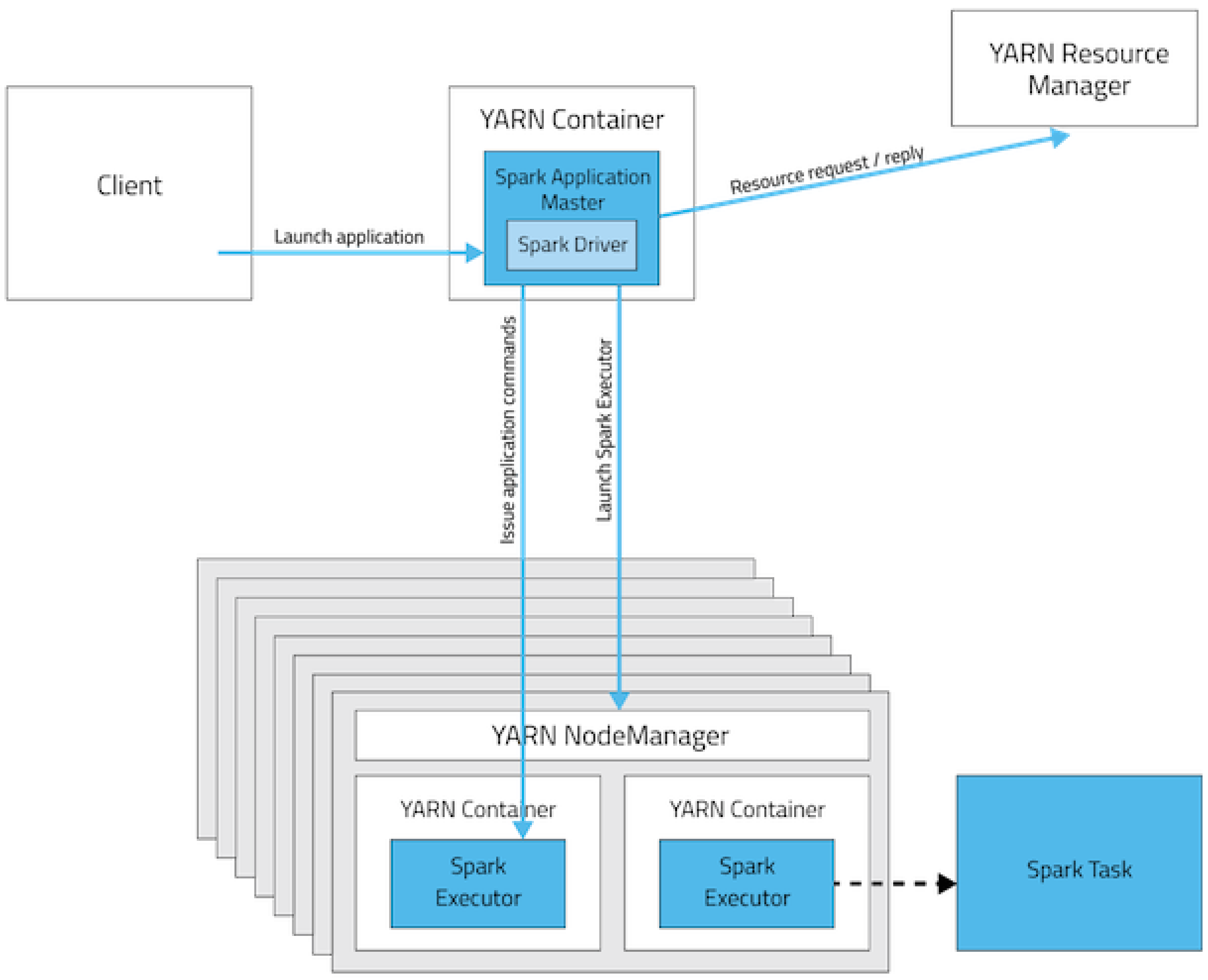
Running on YARN

When running Spark on YARN, each Spark executor runs as a YARN container. Where MapReduce schedules a container and fires up a JVM for each task, Spark hosts multiple tasks within the same container. This approach enables several orders of magnitude faster task startup time.

Spark supports two modes for running on YARN, “yarn-cluster” mode and “yarn-client” mode. Broadly, yarn-cluster mode makes sense for production jobs, while yarn-client mode makes sense for interactive and debugging uses where you want to see your application’s output immediately.

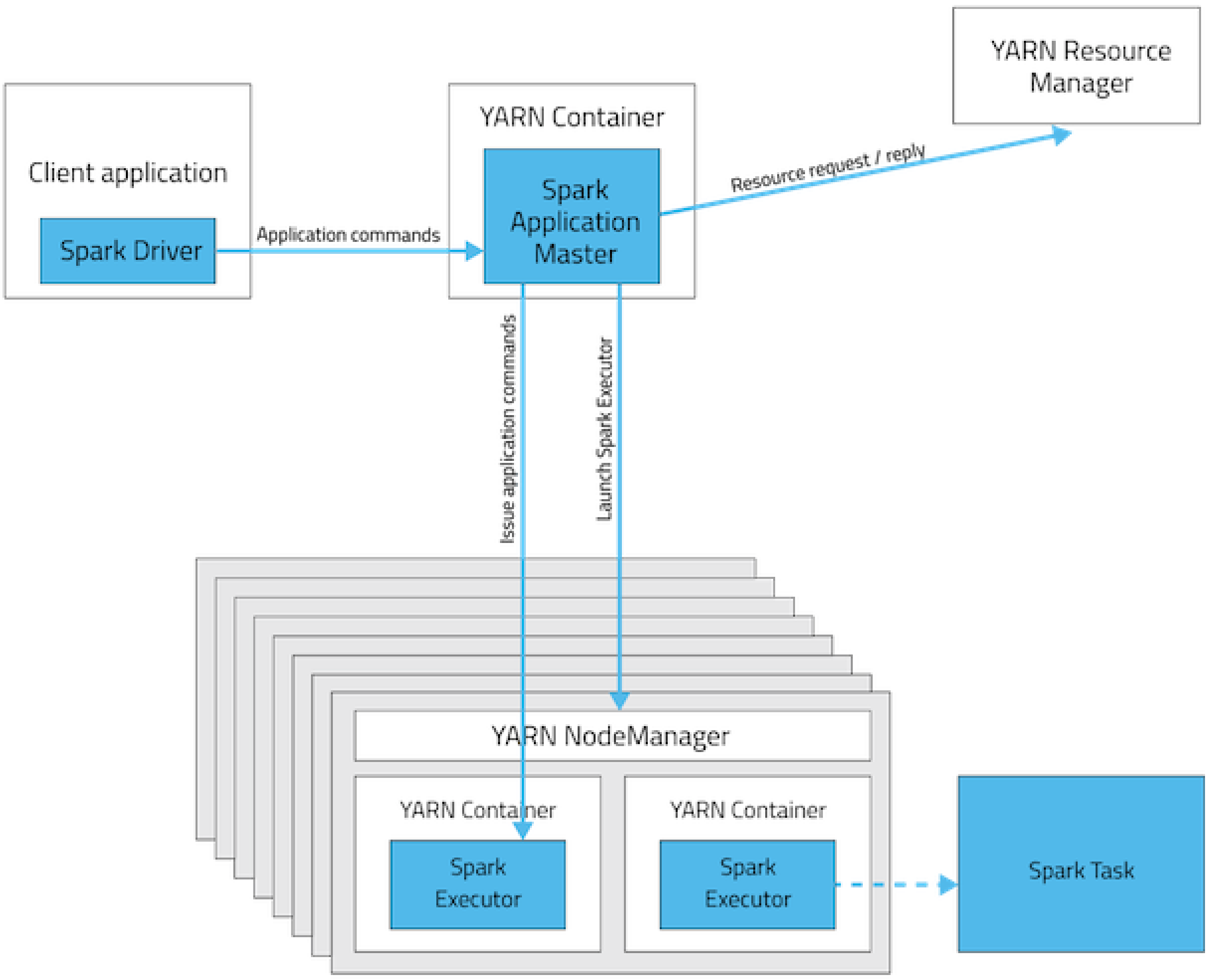
Understanding the difference requires an understanding of YARN’s *Application Master* concept. In YARN, each application instance has an Application Master process, which is the first container started for that application. The application is responsible for requesting resources from the ResourceManager, and, when allocated them, telling NodeManagers to start containers on its behalf. Application Masters obviate the need for an active client — the process starting the application can go away and coordination continues from a process managed by YARN running on the cluster.

In yarn-cluster mode, the driver runs in the Application Master. This means that the same process is responsible for both driving the application and requesting resources from YARN, and this process runs inside a YARN container. The client that starts the app doesn’t need to stick around for its entire lifetime.



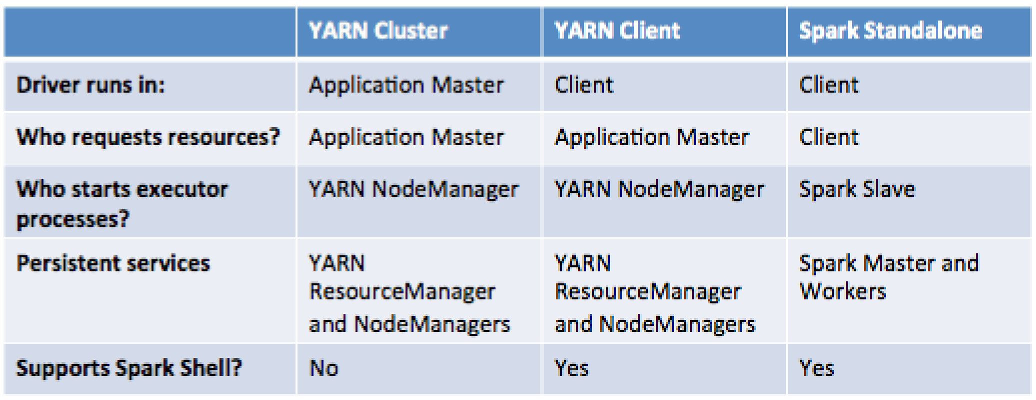
yarn-cluster mode

The yarn-cluster mode, however, is not well suited to using Spark interactively. Spark applications that require user input, like spark-shell and PySpark, need the Spark driver to run inside the client process that initiates the Spark application. In yarn-client mode, the Application Master is merely present to request executor containers from YARN. The client communicates with those containers to schedule work after they start:



yarn-client mode

This table offers a concise list of differences between these modes:



	YARN Cluster	YARN Client	Spark Standalone
Driver runs in:	Application Master	Client	Client
Who requests resources?	Application Master	Application Master	Client
Who starts executor processes?	YARN NodeManager	YARN NodeManager	Spark Slave
Persistent services	YARN ResourceManager and NodeManagers	YARN ResourceManager and NodeManagers	Spark Master and Workers
Supports Spark Shell?	No	Yes	Yes

Key Concepts in Summary

- **Application:** This may be a single job, a sequence of jobs, a long-running service issuing new commands as needed or an interactive exploration session.
- **Spark Driver:** The Spark driver is the process running the spark context (which represents the application session). This driver is responsible for converting the application to a directed graph of individual steps to execute on the cluster. There is one driver per application.
- **Spark Application Master:** The Spark Application Master is responsible for negotiating resource requests made by the driver with YARN and finding a suitable set of hosts/containers in which to run the Spark applications. There is one Application Master per application.
- **Spark Executor:** A single JVM instance on a node that serves a single Spark application. An executor runs multiple tasks over its lifetime, and multiple tasks concurrently. A node may have several Spark executors and there are many nodes running Spark Executors for each client application.
- **Spark Task:** A Spark Task represents a unit of work on a partition of a distributed dataset