Q- describes the responsibility of the executors in Spark?

Ans- The executors, which JVM processes, accept tasks from the driver, execute those tasks, and return the results to the driver.

Q- statements about executors is correct, assuming that one can consider each of the JVMs working as executors as a pool of task execution slots?

Ans-**Tasks run in parallel via slots.**

Given the assumption, an executor then has one or more "slots", defined by the equation `spark.executor.cores / spark.task.cpus`. With the executor's resources divided into slots, each task takes up a slot and multiple tasks can be executed in parallel.

**The hierarchy is, from top to bottom: Job, Stage, Task.

Cluster Manager Types

The system currently supports several cluster managers:

- Standalone – a simple cluster manager included with Spark that makes it easy to set up a cluster.
- Apache Mesos – a general cluster manager that can also run Hadoop MapReduce and service applications. (Deprecated)
- Hadoop YARN – the resource manager in Hadoop 2 and 3.
- Kubernetes – an open-source system for automating deployment, scaling, and management of containerized applications.

difference between client and cluster execution modes?

Cluster mode→ driver run on the worker node

Client mode→ driver runs in client machine

RDD

The high-level DataFrame API is built on top of the low-level RDD API.
RDDs are immutable.
RDD stands for Resilient Distributed Dataset.
RDDs are great for precisely instructing Spark on how to do a query.

When you execute something on a cluster, the processing of your job is split up into stages, and each stage is split into tasks. Each task is scheduled separately. You can consider each of the JVMs working as executors as a pool of task execution slots, each executor would give you *spark.executor.cores* / *spark.task.cpus* execution slots for your tasks, with a total of *spark.executor.instances* executors. Here's an example.

The cluster with 12 nodes running YARN Node Managers, 64GB of RAM each and 32 CPU cores each (16 physical cores with hyper threading). This way on each node you can start 2 executors with 26GB of RAM each (leave some RAM for system processes, YARN NM and DataNode), each executor with 12 cores to be utilized for tasks (leave some cores for system processes, YARN NM and DataNode). So In total your cluster would handle 12 machines * 2 executors per machine * 12 cores per executor / 1 core for each task = 288 task slots. This means that your Spark cluster would be able to run up to 288 tasks in parallel thus utilizing almost all the resources you have on this cluster. The amount of RAM you can use for caching your data on this cluster is 0.9 *spark.storage.safetyFraction* * 0.6 *spark.storage.memoryFraction* * 12 machines * 2 executors per machine * 26 GB per executor = 336.96 GB. Not that much, but in most cases it is enough.

So far so good, now you know how the Spark uses its JVM's memory and what are the execution slots you have on your cluster. As you might already noticed, I didn't stop in details on what the "task" really is. This would be a subject of the next article, but basically it is a single unit of work performed by Spark, and is executed as a ***thread*** in the executor JVM. This is the secret under the Spark low job startup time – forking additional thread inside of the JVM is much faster that bringing up the whole JVM, which is performed when you start a MapReduce job in Hadoop.

The driver is responsible for scheduling query for execution on worker node

Responsibilities of the client process component

The client process starts the driver program. For example, the client process can be a `spark-submit` script for running applications, a `spark-shell` script, or a custom application using Spark API. The client process prepares the classpath and all configuration options for the Spark application. It also passes application arguments, if any, to the application running inside the driver.

Responsibilities of the driver component

The *driver* orchestrates and monitors execution of a Spark application. There's always one driver per Spark application. You can think of the driver as a wrapper around the application. The driver and its subcomponents – the Spark context and scheduler – are responsible for:

- requesting memory and CPU resources from cluster managers
- breaking application logic into stages and tasks
- sending tasks to executors
- collecting the results

Responsibilities of the executors

The *executors,* which JVM processes, accept tasks from the driver, execute those tasks, and return the results to the driver.

Spark cluster types

Spark can run in local mode and inside Spark standalone, YARN, and Mesos clusters. Although Spark runs on all of them, one might be more applicable for your environment and use cases. In this section, you'll find the pros and cons of each cluster type.

Spark standalone cluster

A Spark standalone cluster is a Spark-specific cluster. Because a standalone cluster's built specifically for Spark applications, it doesn't support communication with an HDFS secured with Kerberos authentication protocol. If you need that kind of security, use YARN for running Spark. A Spark standalone cluster, but provides faster job startup than those jobs running on YARN.

YARN cluster

YARN is Hadoop's resource manager and execution system. It's also known as MapReduce 2 because it superseded the MapReduce engine in Hadoop 1 that supported only MapReduce jobs.