Deep Dive into Spark Internals

by

Wei Shung Chung

Spark Application

```
def main(args: Array[String]) = {
  val conf = new SparkConf().setAppName("My Spark App")
  val session = SparkSession.builder().getOrCreate()
  val sc = session.sparkContext


  val textFile = sc.textFile(myInputPath)
  val counts = textFile.flatMap(line => line.split(","))
    .map(word => (word, 1))
    .reduceByKey(_ + _)


  counts.saveAsTextFile(myOutputPath)
}
```

SparkConf

```
val conf = new SparkConf().setAppName("My Spark")
```

Configuration for a Spark application.
Used to set various Spark parameters as key-value pairs.

spark.executor.extraJavaOptions	-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+UseConcMarkSweepGC -XX:CMSInitiatingOccupancyFraction=70 -XX:MaxHeapFreeRatio=70 -XX:+CMSClassUnloadingEnabled -XX:OnOutOfMemoryError='kill -9 %p'
spark.executor.extraLibraryPath	/usr/lib/hadoop/lib/native:/usr/lib/hadoop-lzo/lib/native
spark.executor.id	driver
spark.executor.memory	5120M
spark.hadoop.yarn.timeline-service.enabled	false
spark.history.fs.logDirectory	hdfs:///var/log/spark/apps
spark.history.ui.port	18080
spark.home	/usr/lib/spark
spark.jars	
spark.master	yarn

SparkConf

```
private[spark] def loadFromSystemProperties(silent: Boolean): SparkConf = {
    // Load any spark.* system properties
    for ((key, value) <- Utils.getSystemProperties if key.startsWith("spark."))
{
      set(key, value, silent)
    }
    this
  }
```

SparkConf

```
def setMaster(master: String): SparkConf = {
    set("spark.master", master)
  }



  /** Set a name for your application. Shown in the Spark web UI. */
  def setAppName(name: String): SparkConf = {
    set("spark.app.name", name)
  }

  /** Set JAR files to distribute to the cluster. */
  def setJars(jars: Seq[String]): SparkConf = {
    for (jar <- jars if (jar == null)) logWarning("null jar passed to SparkContext
constructor")
    set("spark.jars", jars.filter(_ != null).mkString(","))
  }
```

```
  def setJars(jars: Array[String]): SparkConf = {
    setJars(jars.toSeq)
  }
```

RDD

```
val logRDD = spark.read.textFile(“logDirectory")
```

A list of partitions

A function for computing each split

A list of dependencies on other RDDs (dependency/lineage graph)

Optionally, a Partitioner for key-value RDDs
Optionally, a list of preferred locations to compute
each split on (e.g. block locations for an HDFS file)

Transformation

val linesWithSpark = logRDD.filter(line => line.contains("Spark"))

Function to be executed on RDD

Action

linesWithSpark.count()

Function to be executed on RDD

SparkContext

val sc = session.SparkContext

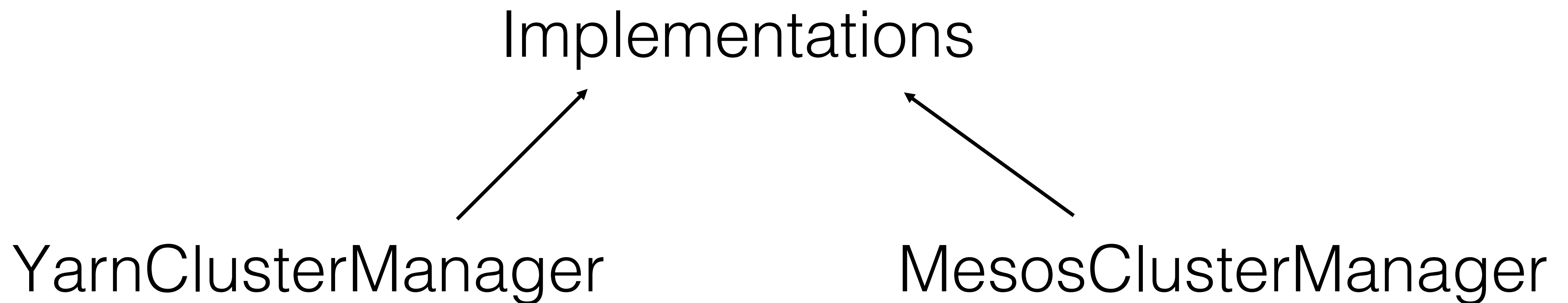
2922 lines of codes
represents the connection to a Spark cluster
create RDDs, accumulators, and broadcast variables

Components in SparkContext

SparkEnv
[BlockManager, MemoryManager, BroadcastManager,ShuffleManager]

ClusterManager
DAGScheduler
TaskScheduler
SchedulerBackend
LiveListenerBus
HeartbeatReceiver
Collection of jars, files
applicationId, applicationAttemptId

ExternalClusterManager trait



To create SchedulerBackend and TaskScheduler

SchedulerBackend

LocalSchedulerBackend

StandaloneSchedulerBackend

YarnSchedulerBackend

YarnClusterSchedulerBackend (deploy-mode cluster)

YarnClientSchedulerBackend (deploy-mode client)

Communicate with YARN to request executors

TaskScheduler

- YarnClusterScheduler for cluster deploy mode
- YarnScheduler for client deploy mode

schedule tasks on Executors

DAGScheduler

- **Job**

  Top-level work items submitted to the scheduler

  Require execution of multiple stages

  eg. count()

- **Stage**

  Set of tasks computing the same function on partitions of the same RDD

  Stages are separated at shuffle boundaries

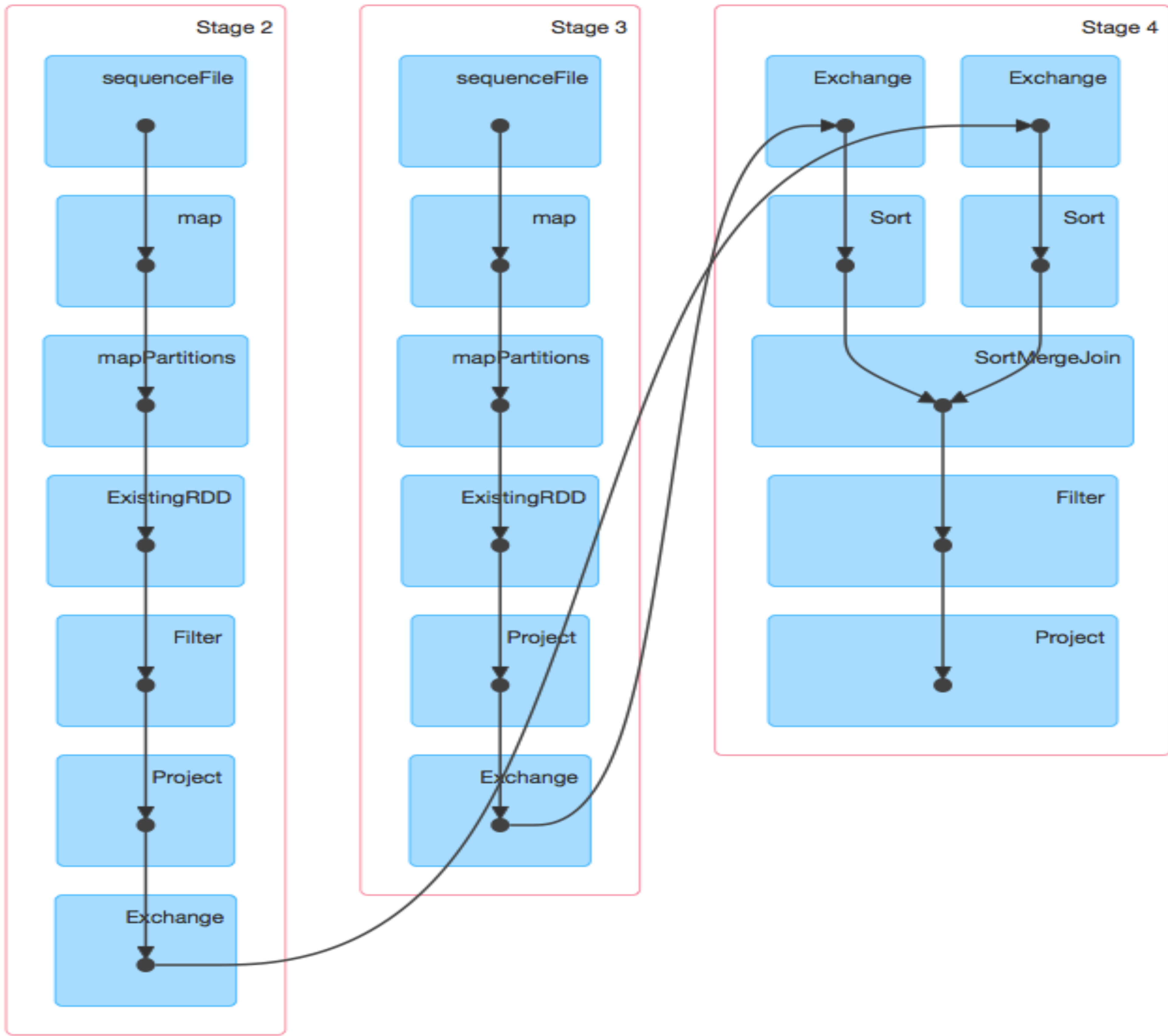
- **Task**

  Individual unit of work

  Computes where to run each task based on the preferred locations of its underlying RDDs

Cluster Manager in SparkContext

```
case masterUrl =>
    val cm = getClusterManager(masterUrl) match {
      case Some(clusterMgr) => clusterMgr
      case None => throw new SparkException("Could not parse Master URL: '" + master + "'")
    }
    try {
      val scheduler = cm.createTaskScheduler(sc, masterUrl)
      val backend = cm.createSchedulerBackend(sc, masterUrl, scheduler)
      cm.initialize(scheduler, backend)
      (backend, scheduler)
    } catch {
      case se: SparkException => throw se
      case NonFatal(e) =>
        throw new SparkException("External scheduler cannot be instantiated", e)
}
```


Stage 2
sequenceFile
map
mapPartitions
ExistingRDD
Filter
Project
Exchange
Stage 3
sequenceFile
map
mapPartitions
ExistingRDD
Project
Exchange
Stage 4
Exchange
Exchange
Sort
Sort
SortMergeJoin
Filter
Project

DAGScheduler

Know about the locations that each RDD's partitions are cached on

```
private val cacheLocs = new HashMap[Int,
    IndexedSeq[Seq[TaskLocation]]]
```

by using BlockManagerMaster.getLocations

RDD.count()

```
/**
 * Return the number of elements in the RDD.
 */
def count(): Long = sc.runJob(this, Utils.getIteratorSize _).sum
```

SparkContext

```
def runJob[T, U: ClassTag](
    rdd: RDD[T],
    func: (TaskContext, Iterator[T]) => U,
    partitions: Seq[Int],
    resultHandler: (Int, U) => Unit): Unit = {
  if (stopped.get()) {
    throw new IllegalStateException("SparkContext has been shutdown")
  }
  val callSite = getCallSite
  val cleanedFunc = clean(func)
  logInfo("Starting job: " + callSite.shortForm)
  if (conf.getBoolean("spark.logLineage", false)) {
    logInfo("RDD's recursive dependencies:\n" + rdd.toDebugString)
  }
  dagScheduler.runJob(rdd, cleanedFunc, partitions, callSite,
               resultHandler, localProperties.get)
  progressBar.foreach(_.finishAll())
  rdd.doCheckpoint()
}
```

DAGScheduler

```
def submitJob[T, U](
      rdd: RDD[T],
      func: (TaskContext, Iterator[T]) => U,
      partitions: Seq[Int],
      callSite: CallSite,
      resultHandler: (Int, U) => Unit,
      properties: Properties): JobWaiter[U] = {

Some codes omitted

    assert(partitions.size > 0)
    val func2 = func.asInstanceOf[(TaskContext, Iterator[_]) => _]
    val waiter = new JobWaiter(this, jobId, partitions.size, resultHandler)
    eventProcessLoop.post(JobSubmitted(
      jobId, rdd, func2, partitions.toArray, callSite, waiter,
      SerializationUtils.clone(properties)))
    waiter
  }
```

SparkContext

SparkContext submits job

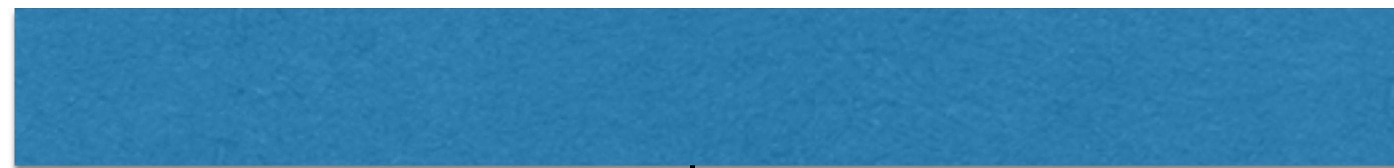
post to EventLoop

daemon thread takes event from event queue

call sparkContext's handleJobSubmittedEvent method

DAGScheduler EventProcessLoop

Event Queue



DAGSchedulerEvent

Event thread
poll event

Call DAGScheduler's
event handling methods

DAGSchedulerEvent

JobSubmitted
MapStageSubmitted
StageCancelled
JobCancelled
JobGroupCancelled
AllJobsCancelled
ExecutorAdded
ExecutorLost
WorkerRemoved
BeginEvent
SpeculativeTaskSubmitted
GettingResultEvent
CompletionEvent
TaskSetFailed
ResubmitFailedStages

Event Handling

```
private def doOnReceive(event: DAGSchedulerEvent): Unit = event match {
    case JobSubmitted(jobId, rdd, func, partitions, callSite, listener, properties) =>
      dagScheduler.handleJobSubmitted(jobId, rdd, func, partitions, callSite, listener, properties)

    case MapStageSubmitted(jobId, dependency, callSite, listener, properties) =>
      dagScheduler.handleMapStageSubmitted(jobId, dependency, callSite, listener, properties)

    case StageCancelled(stageId, reason) =>
      dagScheduler.handleStageCancellation(stageId, reason)

    case JobCancelled(jobId, reason) =>
      dagScheduler.handleJobCancellation(jobId, reason)

    case JobGroupCancelled(groupId) =>
      dagScheduler.handleJobGroupCancelled(groupId)
```

DAGScheduler HandleJobSubmitted

- Create stages

function to be executed

```
finalStage = createResultStage(finalRDD, func, partitions,
                                jobId, callSite)
```

list of partitions

- Submit stage, recursively submits any missing parents

```
submitStage(finalStage)
```

DAGScheduler

- Come up with the tasks for stage (partitions, function)
- taskScheduler.submitTasks(TaskSet)

Executor

Driver has TaskScheduler
    YarnClusterTaskScheduler will communicate with executors

Container -> start CoarseGrainedExecutorBackend
    listen for subsequent tasks submitted from Driver

CoarseGrainedExecutorBackend has a main method
    start Executor (threadPool)
    receive task -> launch task

Executor

Use a thread pool to run tasks (TaskRunners)

```
def launchTask(context: ExecutorBackend, taskDescription: TaskDescription): Unit
= {

    val tr = new TaskRunner(context, taskDescription)

    runningTasks.put(taskDescription.taskId, tr)

    threadPool.execute(tr)

  }
```

Executor

```
  // Application dependencies (added through SparkContext) that we've fetched so
far on this node.

  // Each map holds the master's timestamp for the version of that file or JAR
we got.

  private val currentFiles: HashMap[String, Long] = new HashMap[String, Long]()

  private val currentJars: HashMap[String, Long] = new HashMap[String, Long]()
```

Executor Thread Pool

```
// Start worker thread pool
  private val threadPool = {
    val threadFactory = new ThreadFactoryBuilder()
      .setDaemon(true)
      .setNameFormat("Executor task launch worker-%d")
      .setThreadFactory(new ThreadFactory {
        override def newThread(r: Runnable): Thread =
          // Use UninterruptibleThread to run tasks so that we can allow running codes without being
          // interrupted by `Thread.interrupt()`. Some issues, such as KAFKA-1894, HADOOP-10622,
          // will hang forever if some methods are interrupted.
          new UninterruptibleThread(r, "unused") // thread name will be set by ThreadFactoryBuilder
      })
      .build()
    Executors.newCachedThreadPool(threadFactory).asInstanceOf[ThreadPoolExecutor]
  }
```

Executor

- Deserialize a Task binary from the driver
- Deserialize the RDD and function to be run

TaskRunner

```
override def run(): Unit = {
      threadId = Thread.currentThread.getId
      Thread.currentThread.setName(threadName)
      val threadMXBean = ManagementFactory.getThreadMXBean
      val taskMemoryManager = new TaskMemoryManager(env.memoryManager, taskId)
      val deserializeStartTime = System.currentTimeMillis()
      val deserializeStartCpuTime = if (threadMXBean.isCurrentThreadCpuTimeSupported) {
        threadMXBean.getCurrentThreadCpuTime
      } else 0L
      Thread.currentThread.setContextClassLoader(replClassLoader)
      val ser = env.closureSerializer.newInstance()
      logInfo(s"Running $taskName (TID $taskId)")
      execBackend.statusUpdate(taskId, TaskState.RUNNING, EMPTY_BYTE_BUFFER)
      var taskStart: Long = 0
      var taskStartCpu: Long = 0
      startGCTime = computeTotalGcTime()

      try {
        // Must be set before updateDependencies() is called, in case fetching dependencies
        // requires access to properties contained within (e.g. for access control).
        Executor.taskDeserializationProps.set(taskDescription.properties)

        updateDependencies(taskDescription.addedFiles, taskDescription.addedJars)
        task = ser.deserialize[Task[Any]](
          taskDescription.serializedTask, Thread.currentThread.getContextClassLoader)
        task.localProperties = taskDescription.properties
        task.setTaskMemoryManager(taskMemoryManager)
```