mastering-apache-spark—jacek-lackowski.pdf

Table of Contents

Apache Spark Overview....3
Spark Architecture....5
  Spark Codebase comparison 2.3.0 versus 1.3.0....6
    SparkEnv....6
    Executor....6
    SparkConf....9
Driver....10
Executor....12
  Creating an Executor Instance....17
  launchTask method....19
  startDriverHeartBeats....20
  reportHeartbeat....22
  Coarse-Grained Executors....24
  TaskRunner....26
    long-ass run method....27
SparkContext - The main entry point to Spark functionality....27
  SparkContext Constructors....30
  submitJob - to submit jobs Asynchronously....31
  runJob - to run jobs Sychronously....32
  stop....35
  parallelize - Creating RDDs....37
  unpersist - marking RDDs as Non-Persistent....37
  Creating Built-In Accumulators....38
  Creating Broadcast Variables....39
  Inside creation of a SparkContext....41
  createTaskScheduler....42
   - HeartbeatReceiver RPC Endpoint....43
    Creating a HeartbeatReceiver Instance....45
    Heartbeat....46
SparkConf - Spark Application Configuration....48
  Spark Properties....48
  Setting up Spark Properties....48
  Default Configuration....48
SparkEnv - the Spark Runtime Environment....50
  SparkEnv Services & Internal Properties....51
  SparkEnv Factory-Object....52
    Base --- SparkEnv - create method....52
    registerOrLookupEndpoint method....55
    createDriverEnv method....55
    createExecutorEnv method....56
  Accesing current SparkEnv — get method....57
RDD — Resilient Distributed Dataset....58
  RDD - Additional Traits....60

RDD Motivations – Iterative Algorithms & Interactive Query Tools....62
persist and persist Internal Methods....63
RDD Contract....64
Types of RDDs....65
Creating RDDs....66
SparkContext.parallelize....66
SparkContext.makeRDD....66
SparkContext.textFile....66
RDD Caching and Persistence....68
StorageLevel....69
StorageLevel types (number _2 in the name denotes 2 replicas):....69
Transformations....71
Narrow and Wide Transformations on RDDs....72
Actions....73
Broadcast Variables....75
Broadcast Spark Developer-Facing Contract....76
Lifecycle of a Broadcast Variable....76
value method....77
unpersist & destroy methods....78
Accumulators....79
AccumulatorV2....80
AccumulatorMetadata....81
AccumulableInfo....81
AccumulatorContext....82
AccumulatorContext.SQL_ACCUM_IDENTIFIER....82
DAGScheduler – Stage-Oriented Scheduler....84
Creating a *DAGScheduler* Instance....87
liveListenerBus method....87
executorHeartbeatReceived method....88
submitJob method Synchronous....88
runJob method Asynchronous....90
getOrCreateShuffleMapStage method....90
createShuffleMapStage method....91
getShuffleDepenendancies method....93

Apache Spark Overview

Apache Spark is an **open-source distributed general-purpose cluster computing framework** with (mostly) **in-memory data processing engine** that can do ETL, analytics, machine learning and graph processing on large volumes of data at rest (batch processing) or in motion (streaming processing) with rich concise high-level APIs for the programming languages: Scala, Python, Java, R, and SQL.

You could also describe Spark as a distributed, data processing engine for **batch and streaming modes** featuring SQL queries, graph processing, and machine learning.

In contrast to Hadoop's two-stage disk-based MapReduce computation engine, Spark's multi-stage (mostly) in-memory computing engine allows for running most computations in memory, and hence most of the time provides better performance for certain applications, e.g. iterative algorithms or interactive data mining (read Spark officially sets a new record in large-scale sorting).

Spark aims at speed, ease of use, extensibility and interactive analytics.

Spark is often called **cluster computing engine** or simply **execution engine**.

Spark is a **distributed platform for executing complex multi-stage applications**, like **machine learning algorithms**, and **interactive ad hoc queries**. Spark provides an efficient abstraction for in-memory cluster computing called Resilient Distributed Dataset.

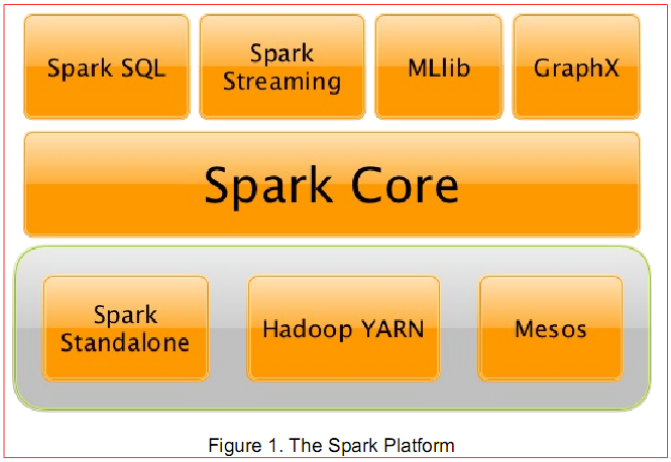


Figure 1. The Spark Platform

…

Spark embraces many concepts in a single unified development and runtime environment.

- Machine learning that is so tool- and feature-rich in Python, e.g. SciKit library, can now be used by Scala developers (as Pipeline API in Spark MLlib or calling `pipe()`).
- DataFrames from R are available in Scala, Java, Python, R APIs.
- Single node computations in machine learning algorithms are migrated to their distributed versions in Spark MLlib.

Apache Spark uses a directed acyclic graph (DAG) of computation stages (aka **execution DAG**). It postpones any processing until really required for actions. Spark's **lazy evaluation** gives plenty of opportunities to induce low-level optimizations (so users have to know less to do more).

Spark supports diverse workloads, but successfully targets low-latency iterative ones. They are often used in Machine Learning and graph algorithms.

Many Machine Learning algorithms require plenty of iterations before the result models get optimal, like logistic regression. The same applies to graph algorithms to traverse all the nodes and edges when needed. Such computations can increase their performance when the interim partial results are stored in memory or at very fast solid state drives.

Spark can cache intermediate data in memory for faster model building and training. Once the data is loaded to memory (as an initial step), reusing it multiple times incurs no performance slowdowns.

Also, graph algorithms can traverse graphs one connection per iteration with the partial result in memory.

……

Spark Architecture

Spark uses a **master/worker architecture**. There is a driver that talks to a single coordinator called master that manages workers in which executors run.

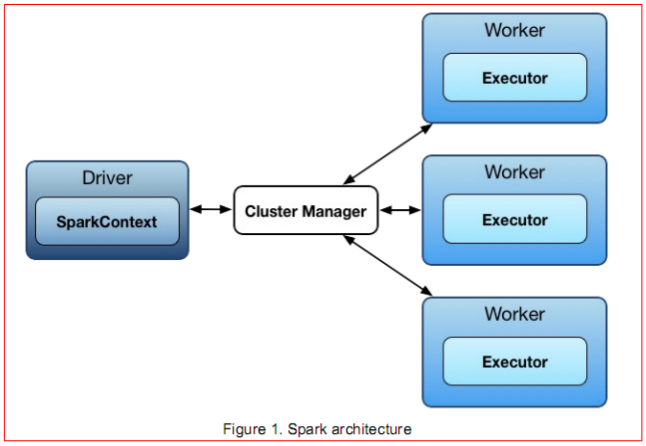


Figure 1. Spark architecture

The driver and the executors run in their own Java processes. You can run them all on the same (*horizontal cluster*) or separate machines (*vertical cluster*) or in a mixed machine configuration.

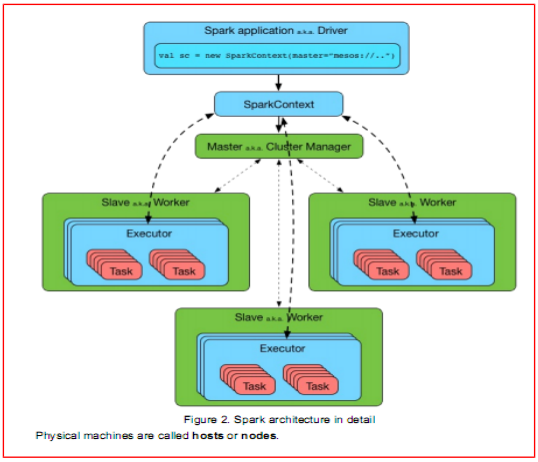


Figure 2. Spark architecture in detail

Physical machines are called **hosts** or **nodes**.

Spark Codebase comparison 2.3.0 versus 1.3.0

NOTE:-- Akka's ActorSystem is not used as of Spark 1.4.0

2.3.0 1.3.0

SparkEnv

```
@DeveloperApi
class SparkEnv (
    val executorId: String,
    private[spark] val rpcEnv: RpcEnv,
    val serializer: Serializer,
    val closureSerializer: Serializer,
    val serializerManager: SerializerManager,
    val mapOutputTracker: MapOutputTracker,
    val shuffleManager: ShuffleManager,
    val broadcastManager: BroadcastManager,
    val blockManager: BlockManager,
    val securityManager: SecurityManager,
    val metricsSystem: MetricsSystem,
    val memoryManager: MemoryManager,
    val outputCommitCoordinator: OutputCommitCoordinator,
    val conf: SparkConf) extends Logging {

  private[spark] var isStopped = false
  private val pythonWorkers = mutable.HashMap[(String, Map[String, String]), PythonWork

  // A general, soft-reference map for metadata needed during HadoopRDD split computati
  // (e.g., HadoopFileRDD uses this to cache JobConfs and InputFormats).
  private[spark] val hadoopJobMetadata = new MapMaker().softValues().makeMap[String, Ar

  private[spark] var driverTmpDir: Option[String] = None
```

```
@DeveloperApi
class SparkEnv (
    val executorId: String,
    private[spark] val rpcEnv: RpcEnv,
    _actorSystem: ActorSystem, // TODO Remove actorSystem
    val serializer: Serializer,
    val closureSerializer: Serializer,
    val cacheManager: CacheManager,
    val mapOutputTracker: MapOutputTracker,
    val shuffleManager: ShuffleManager,
    val broadcastManager: BroadcastManager,
    val blockTransferService: BlockTransferService,
    val blockManager: BlockManager,
    val securityManager: SecurityManager,
    val sparkFilesDir: String,
    val metricsSystem: MetricsSystem,
    val memoryManager: MemoryManager,
    val outputCommitCoordinator: OutputCommitCoordinator,
    val conf: SparkConf) extends Logging {

  // TODO Remove actorSystem
  @deprecated("Actor system is no longer supported as of 1.4.0", "1.4.0")
  val actorSystem: ActorSystem = _actorSystem

  private[spark] var isStopped = false
  private val pythonWorkers = mutable.HashMap[(String, Map[String, String]), Pytho
```

Executor

```
/**
 * Spark executor, backed by a threadpool to run tasks.
 *
 * This can be used with Mesos, YARN, and the standalone scheduler.
 * An internal RPC interface is used for communication with the driver,
 * except in the case of Mesos fine-grained mode.
 */
private[spark] class Executor(
    executorId: String,
    executorHostname: String,
    env: SparkEnv,
    userClassPath: Seq[URL] = Nil,
    isLocal: Boolean = false,
    uncaughtExceptionHandler: UncaughtExceptionHandler = new SparkUncaughtExceptionHand
  extends Logging {
```

```
import org.apache.spark.util.{ChildFirstURLClassLoader, MutableURLClassLoader,
  SparkUncaughtExceptionHandler, AkkaUtils, Utils}

/**
 * Spark executor used with Mesos, YARN, and the standalone scheduler.
 * In coarse-grained mode, an existing actor system is provided.
 */
private[spark] class Executor(
    executorId: String,
    executorHostname: String,
    env: SparkEnv,
    userClassPath: Seq[URL] = Nil,
    isLocal: Boolean = false)
  extends Logging
```

```
  val threadPool = Utils.newDaemonCachedThreadPool("Executor task launch worker")

  val executorSource = new ExecutorSource(this, executorId)

  if (!isLocal) {
    env.metricsSystem.registerSource(executorSource)
    env.blockManager.initialize(conf.getAppId)
  }

  // Create an actor for receiving RPCs from the driver
  private val executorActor = env.actorSystem.actorOf(
    Props(new ExecutorActor(executorId)), "ExecutorActor")

  // Whether to load classes in user jars before those in Spark jars
  private val userClassPathFirst: Boolean = {
    conf.getBoolean("spark.executor.userClassPathFirst",
      conf.getBoolean("spark.files.userClassPathFirst", false))
  }

  // Create our ClassLoader
  // do this after SparkEnv creation so can access the SecurityManager
  private val urlClassLoader = createClassLoader()
  private val replClassLoader = addReplClassLoaderIfNeeded(urlClassLoader)

  // Set the classloader for serializer
  env.serializer.setDefaultClassLoader(replClassLoader)

  // Akka's message frame size. If task result is bigger than this, we use the blo
  // to send the result back.
  private val akkaFrameSize = AkkaUtils.maxFrameSizeBytes(conf)
```

```
81
82  if (!isLocal) {
83    // Setup an uncaught exception handler for non-local mode.
84    // Make any thread terminations due to uncaught exceptions kill the entire
85    // executor process to avoid surprising stalls.
86    Thread.setDefaultUncaughtExceptionHandler(uncaughtExceptionHandler)
87  }
88
89  // Start worker thread pool
90  private val threadPool = {
91    val threadFactory = new ThreadFactoryBuilder()
92      .setDaemon(true)
93      .setNameFormat("Executor task launch worker-%d")
94      .setThreadFactory(new ThreadFactory {
95        override def newThread(r: Runnable): Thread =
96          // Use UninterruptibleThread to run tasks so that we can allow running codes
97          // interrupted by `Thread.interrupt()`. Some issues, such as KAFKA-1894, HADO
98          // will hang forever if some methods are interrupted.
99          new UninterruptibleThread(r, "unused") // thread name will be set by ThreadFa
100     })
101     .build()
102   Executors.newCachedThreadPool(threadFactory).asInstanceOf[ThreadPoolExecutor]
103 }
104 private val executorSource = new ExecutorSource(threadPool, executorId)
105 // Pool used for threads that supervise task killing / cancellation
106 private val taskReaperPool = ThreadUtils.newDaemonCachedThreadPool("Task reaper")
107 // For tasks which are in the process of being killed, this map holds the most recent
108 // TaskReaper. All accesses to this map should be synchronized on the map itself (thi
109 // a ConcurrentHashMap because we use the synchronization for purposes other than sim
110 // the integrity of the map's internal state). The purpose of this map is to prevent
111 // of a separate TaskReaper for every killTask() of a given task. Instead, this map a
112 // track whether an existing TaskReaper fulfills the role of a TaskReaper that we wou
113 // create. The map key is a task id.
```

```
73
74  if (!isLocal) {
75    // Setup an uncaught exception handler for non-local mode.
76    // Make any thread terminations due to uncaught exceptions kill the entire
77    // executor process to avoid surprising stalls.
78    Thread.setDefaultUncaughtExceptionHandler(SparkUncaughtExceptionHandler)
79  }
80
81  // Start worker thread pool
82  val threadPool = Utils.newDaemonCachedThreadPool("Executor task launch worker")
83
84  val executorSource = new ExecutorSource(this, executorId)
85
86  if (!isLocal) {
87    env.metricsSystem.registerSource(executorSource)
88    env.blockManager.initialize(conf.getAppId)
89  }
90
91  // Create an actor for receiving RPCs from the driver
92  private val executorActor = env.actorSystem.actorOf(
93    Props(new ExecutorActor(executorId)), "ExecutorActor")
94
95  // Whether to load classes in user jars before those in Spark jars
96  private val userClassPathFirst: Boolean = {
97    conf.getBoolean("spark.executor.userClassPathFirst",
98      conf.getBoolean("spark.files.userClassPathFirst", false))
99  }
100
101 // Create our ClassLoader
102 // do this after SparkEnv creation so can access the SecurityManager
103 private val urlClassLoader = createClassLoader()
104 private val replClassLoader = addReplClassLoaderIfNeeded(urlClassLoader)
105
```

```
145   private val maxResultSize = conf.get(MAX_RESULT_SIZE)
146
147   // Maintains the list of running tasks.
148   private val runningTasks = new ConcurrentHashMap[Long, TaskRunner]
149
150   // Executor for the heartbeat task.
151   private val heartbeater = ThreadUtils.newDaemonSingleThreadScheduledExecutor("driver-
152
153   // must be initialized before running startDriverHeartbeat()
154   private val heartbeatReceiverRef =
155     RpcUtils.makeDriverRef(HeartbeatReceiver.ENDPOINT_NAME, conf, env.rpcEnv)
156
157   /**
158    * When an executor is unable to send heartbeats to the driver more than `HEARTBEAT_M
159    * times, it should kill itself. The default value is 60. It means we will retry to s
160    * heartbeats about 10 minutes because the heartbeat interval is 10s.
161    */
162   private val HEARTBEAT_MAX_FAILURES = conf.getInt("spark.executor.heartbeat.maxFailure
163
164   /**
165    * Count the failure times of heartbeat. It should only be accessed in the heartbeat
166    * successful heartbeat will reset it to 0.
167    */
168   private var heartbeatFailures = 0
169
170   startDriverHeartbeater()
171
172   private[executor] def numRunningTasks: Int = runningTasks.size()
173
174   def launchTask(context: ExecutorBackend, taskDescription: TaskDescription): Unit = {
175     val tr = new TaskRunner(context, taskDescription)
176     runningTasks.put(taskDescription.taskId, tr)
177     threadPool.execute(tr)
178   }
```

```
115
116   // Maintains the list of running tasks.
117   private val runningTasks = new ConcurrentHashMap[Long, TaskRunner]
118
119   startDriverHeartbeater()
120
121   def launchTask(
122       context: ExecutorBackend,
123       taskId: Long,
124       attemptNumber: Int,
125       taskName: String,
126       serializedTask: ByteBuffer) {
127     val tr = new TaskRunner(context, taskId = taskId, attemptNumber = attemptNumbe
128       serializedTask)
129     runningTasks.put(taskId, tr)
130     threadPool.execute(tr)
131   }
132
133   def killTask(taskId: Long, interruptThread: Boolean) {
134     val tr = runningTasks.get(taskId)
135     if (tr != null) {
136       tr.kill(interruptThread)
137     }
138   }
139
140   def stop() {
141     env.metricsSystem.report()
142     env.actorSystem.stop(executorActor)
143     isStopped = true
144     threadPool.shutdown()
145     if (!isLocal) {
146       env.stop()
147     }
148   }
```

```
168   private var heartbeatFailures = 0
169
170   startDriverHeartbeater()
171
172   private[executor] def numRunningTasks: Int = runningTasks.size()
173
174   def launchTask(context: ExecutorBackend, taskDescription: TaskDescription): Unit = {
175     val tr = new TaskRunner(context, taskDescription)
176     runningTasks.put(taskDescription.taskId, tr)
177     threadPool.execute(tr)
178   }
179
180   def killTask(taskId: Long, interruptThread: Boolean, reason: String): Unit = {
181     val taskRunner = runningTasks.get(taskId)
182     if (taskRunner != null) {
183       if (taskReaperEnabled) {
```

```
121   def launchTask(
122       context: ExecutorBackend,
123       taskId: Long,
124       attemptNumber: Int,
125       taskName: String,
126       serializedTask: ByteBuffer) {
127     val tr = new TaskRunner(context, taskId = taskId, attemptNumber = attemptNumbe
128       serializedTask)
129     runningTasks.put(taskId, tr)
130     threadPool.execute(tr)
131   }
132
133   def killTask(taskId: Long, interruptThread: Boolean) {
134     val tr = runningTasks.get(taskId)
135     if (tr != null) {
136       tr.kill(interruptThread)
```

```
  }

  /** Reports heartbeat and metrics for active tasks to the driver. */
  private def reportHeartBeat(): Unit = {
    // list of (task id, accumUpdates) to send back to the driver
    val accumUpdates = new ArrayBuffer[(Long, Seq[AccumulatorV2[_, _]])]()
    val curGCTime = computeTotalGcTime()

    for (taskRunner <- runningTasks.values().asScala) {
      if (taskRunner.task != null) {
        taskRunner.task.metrics.mergeShuffleReadMetrics()
        taskRunner.task.metrics.setJvmGCTime(curGCTime - taskRunner.startGCTime)
        accumUpdates += ((taskRunner.taskId, taskRunner.task.metrics.accumulators()))
      }
    }

    val message = Heartbeat(executorId, accumUpdates.toArray, env.blockManager.blockMar
    try {
      val response = heartbeatReceiverRef.askSync[HeartbeatResponse](
          message, RpcTimeout(conf, "spark.executor.heartbeatInterval", "10s"))
      if (response.reregisterBlockManager) {
        logInfo("Told to re-register on heartbeat")
        env.blockManager.reregister()
      }
      heartbeatFailures = 0
    } catch {
      case NonFatal(e) =>
        logWarning("Issue communicating with driver in heartbeater", e)
        heartbeatFailures += 1
        if (heartbeatFailures >= HEARTBEAT_MAX_FAILURES) {
          logError(s"Exit as unable to send heartbeats to driver " +
            s"more than $HEARTBEAT_MAX_FAILURES times")
```

```
  }

  def startDriverHeartbeater() {
    val interval = conf.getInt("spark.executor.heartbeatInterval", 10000)
    val timeout = AkkaUtils.lookupTimeout(conf)
    val retryAttempts = AkkaUtils.numRetries(conf)
    val retryIntervalMs = AkkaUtils.retryWaitMs(conf)
    val heartbeatReceiverRef = AkkaUtils.makeDriverRef("HeartbeatReceiver", conf,

    val t = new Thread() {
      override def run() {
        // Sleep a random interval so the heartbeats don't end up in sync
        Thread.sleep(interval + (math.random * interval).asInstanceOf[Int])

        while (!isStopped) {
          val tasksMetrics = new ArrayBuffer[(Long, TaskMetrics)]()
          val curGCTime = gcTime

          for (taskRunner <- runningTasks.values()) {
            if (taskRunner.attemptedTask.nonEmpty) {
              Option(taskRunner.task).flatMap(_.metrics).foreach { metrics =>
                metrics.updateShuffleReadMetrics()
                metrics.updateInputMetrics()
                metrics.setJvmGCTime(curGCTime - taskRunner.startGCTime)

                if (isLocal) {
                  // JobProgressListener will hold an reference of it during
                  // onExecutorMetricsUpdate(), then JobProgressListener can not s
                  // the changes of metrics any more, so make a deep copy of it
                  val copiedMetrics = Utils.deserialize[TaskMetrics](Utils.seriali
                  tasksMetrics += ((taskRunner.taskId, copiedMetrics))
                } else {
                  // It will be copied by serialization
```

```
        logWarning("Issue communicating with driver in heartbeater", e)
        heartbeatFailures += 1
        if (heartbeatFailures >= HEARTBEAT_MAX_FAILURES) {
          logError(s"Exit as unable to send heartbeats to driver " +
            s"more than $HEARTBEAT_MAX_FAILURES times")
          System.exit(ExecutorExitCode.HEARTBEAT_FAILURE)
        }
    }
  }

  /**
   * Schedules a task to report heartbeat and partial metrics for active tasks to drive
   */
  private def startDriverHeartbeater(): Unit = {
    val intervalMs = conf.getTimeAsMs("spark.executor.heartbeatInterval", "10s")

    // Wait a random interval so the heartbeats don't end up in sync
    val initialDelay = intervalMs + (math.random * intervalMs).asInstanceOf[Int]

    val heartbeatTask = new Runnable() {
      override def run(): Unit = Utils.logUncaughtExceptions(reportHeartBeat())
    }
    heartbeater.scheduleAtFixedRate(heartbeatTask, initialDelay, intervalMs, TimeUnit.M
  }
}

private[spark] object Executor {
  // This is reserved for internal use by components that need to read task properties
  // task is fully deserialized. When possible, the TaskContext.getLocalProperty call s
  // used instead.
  val taskDeserializationProps: ThreadLocal[Properties] = new ThreadLocal[Properties]
}
```

```
                  // the changes of metrics any more, so make a deep copy of it
                  val copiedMetrics = Utils.deserialize[TaskMetrics](Utils.seriali
                  tasksMetrics += ((taskRunner.taskId, copiedMetrics))
                } else {
                  // It will be copied by serialization
                  tasksMetrics += ((taskRunner.taskId, metrics))
                }
              }
            }
          }

          val message = Heartbeat(executorId, tasksMetrics.toArray, env.blockManag
          try {
            val response = AkkaUtils.askWithReply[HeartbeatResponse](message, hear
              retryAttempts, retryIntervalMs, timeout)
            if (response.reregisterBlockManager) {
              logWarning("Told to re-register on heartbeat")
              env.blockManager.reregister()
            }
          } catch {
            case NonFatal(t) => logWarning("Issue communicating with driver in hea
          }

          Thread.sleep(interval)
        }
      }
    }
    t.setDaemon(true)
    t.setName("Driver Heartbeater")
    t.start()
  }
}
```

SparkConf

```
214  /** Get all akka conf variables set on this SparkConf */
215  def getAkkaConf: Seq[(String, String)] =
216    /* This is currently undocumented. If we want to make this public we should co
217     * nesting options under the spark namespace to avoid conflicts with user akka
218     * Otherwise users configuring their own akka code via system properties could
219     * spark's akka options.
220     *
221     *   E.g. spark.akka.option.x.y.x = "value"
222     */
223    getAll.filter { case (k, _) => isAkkaConf(k) }
224
225  /**
226   * Returns the Spark application id, valid in the Driver after TaskScheduler reg
227   * from the start in the Executor.
228   */
229  def getAppId: String = get("spark.app.id")
230
231  /** Does the configuration contain a given parameter? */
232  def contains(key: String): Boolean = settings.containsKey(key)
233
234  /** Copy this object */
235  override def clone: SparkConf = {
236    new SparkConf(false).setAll(getAll)
237  }
238
```

```
376  /**
377   * Return whether the given config is an akka config (e.g. akka.actor.provider).
378   * Note that this does not include spark-specific akka configs (e.g. spark.akka.
379   */
380  def isAkkaConf(name: String): Boolean = name.startsWith("akka.")
381
382  /**
383   * Return whether the given config should be passed to an executor on start-up.
384   *
385   * Certain akka and authentication configs are required of the executor when it
386   * the scheduler, while the rest of the spark configs can be inherited from the
387   */
388  def isExecutorStartupConf(name: String): Boolean = {
389    isAkkaConf(name) ||
390    name.startsWith("spark.akka") ||
391    name.startsWith("spark.auth") ||
392    name.startsWith("spark.ssl") ||
393    isSparkPortConf(name)
394  }
395
396  /**
397   * Return true if the given config matches either `spark.*.port` or `spark.port.
```

Driver

Driver

A **Spark driver** (*aka* **an application's driver process**) is a JVM process that hosts SparkContext for a Spark application. It is the master node in a Spark application.

It is the cockpit of jobs and tasks execution (using DAGScheduler and Task Scheduler). It hosts Web UI for the environment.

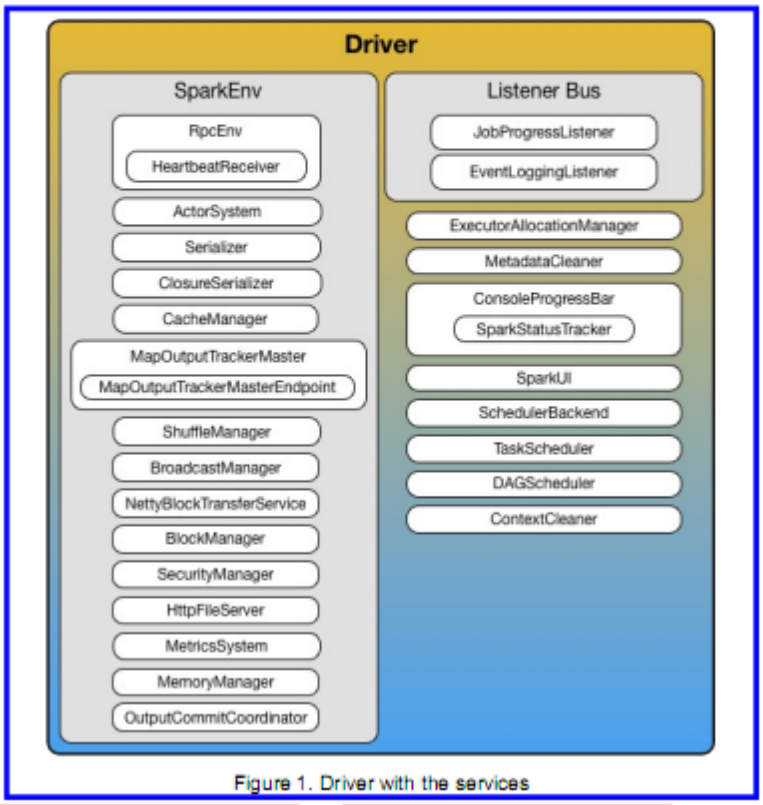


Figure 1. Driver with the services

It splits a Spark application into tasks and schedules them to run on executors.

A driver is where the task scheduler lives and spawns tasks across workers.

A driver coordinates workers and overall execution of tasks.

Note	Spark shell is a Spark application and the driver. It creates a `SparkContext` that is available as `sc`.

Driver requires the additional services (beside the common ones like ShuffleManager, MemoryManager, BlockTransferService, BroadcastManager, CacheManager):

- Listener Bus
- RPC Environment
- MapOutputTrackerMaster with the name **MapOutputTracker**
- BlockManagerMaster with the name **BlockManagerMaster**
- HttpFileServer
- MetricsSystem with the name **driver**
- OutputCommitCoordinator with the endpoint's name **OutputCommitCoordinator**

Caution	FIXME Diagram of RpcEnv for a driver (and later executors). Perhaps it should be in the notes about RpcEnv?

- High-level control flow of work
- Your Spark application runs as long as the Spark driver.
  - Once the driver terminates, so does your Spark application.
- Creates `SparkContext`, RDD's, and executes transformations and actions
- Launches tasks

...

Executor

`Executor` is a distributed agent that is responsible for executing tasks.

`Executor` is created when:

- `CoarseGrainedExecutorBackend` receives `RegisteredExecutor` message (for Spark Standalone and YARN)
- Spark on Mesos's `MesosExecutorBackend` does `registered`
- `LocalEndpoint` is created (for local mode)

`Executor` *typically* runs for the entire lifetime of a Spark application which is called **static allocation of executors** (but you could also opt in for dynamic allocation).

Note	Executors are managed exclusively by executor backends.

Executors reports heartbeat and partial metrics for active tasks to `HeartbeatReceiver` RPC Endpoint on the driver.

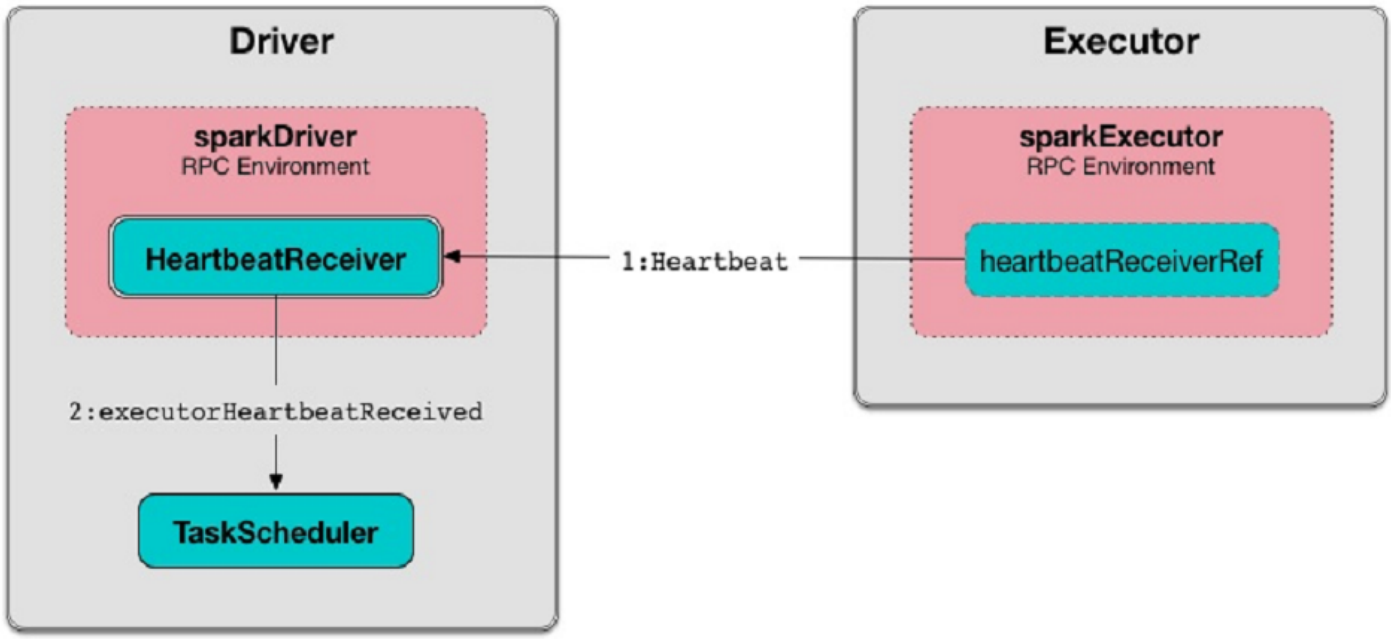


Figure 1. HeartbeatReceiver's Heartbeat Message Handler

Executors provide in-memory storage for RDDs that are cached in Spark applications (via Block Manager).

When an executor starts it first registers with the driver and communicates directly to execute tasks.

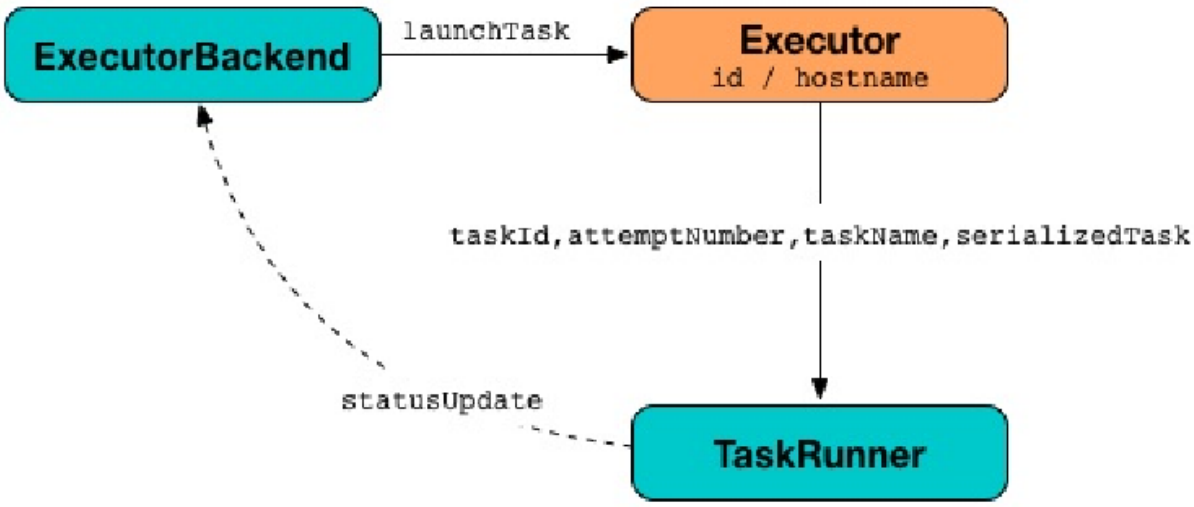


Figure 2. Launching tasks on executor using TaskRunners

Executor offers are described by executor id and the host on which an executor runs (see Resource Offers in this document).

Executors can run multiple tasks over its lifetime, both in parallel and sequentially. They track running tasks (by their task ids in runningTasks internal registry). Consult Launching Tasks section.

Executors use a Executor task launch worker thread pool for launching tasks.

Executors send metrics (and heartbeats) using the internal heartbeater - Heartbeat Sender Thread.

It is recommended to have as many executors as data nodes and as many cores as you can get from the cluster.

Executors are described by their **id**, **hostname**, **environment** (as `SparkEnv`), and **classpath** (and, less importantly, and more for internal optimization, whether they run in local or cluster mode).

Executors have the following internal properties:

Name	InitialValue
executorSource	ExecutorSource
heartbeatFailures	
heartbeatReceiverRef	RPC endpoint reference to HeartbeatReceiver on the driver (available on spark.driver.host at spark.driver.port port). Set when `Executor` is created. Used exclusively when `Executor` reports heartbeats and partial metrics for active tasks to the driver (that happens every spark.executor.heartbeatInterval interval).
maxDirectResultSize	
maxResultSize	
runningTasks	Lookup table of TaskRunners

```
47 /**
48  * Spark executor, backed by a threadpool to run tasks.
49  *
50  * This can be used with Mesos, YARN, and the standalone scheduler.
51  * An internal RPC interface is used for communication with the driver,
52  * except in the case of Mesos fine-grained mode.
53  */
54 private[spark] class Executor(
55     executorId: String,
56     executorHostname: String,
57     env: SparkEnv,
58     userClassPath: Seq[URL] = Nil,
59     isLocal: Boolean = false,
60     uncaughtExceptionHandler: UncaughtExceptionHandler = new SparkUncaughtExceptionHandler)
61   extends Logging {
62
63   logInfo(s"Starting executor ID $executorId on host $executorHostname")
```

```
89   // Start worker thread pool
90   private val threadPool = {
91     val threadFactory = new ThreadFactoryBuilder()
92       .setDaemon(true)
93       .setNameFormat("Executor task launch worker-%d")
94       .setThreadFactory(new ThreadFactory {
95         override def newThread(r: Runnable): Thread =
96           // Use UninterruptibleThread to run tasks so that we can allow running codes without being
97           // interrupted by `Thread.interrupt()`. Some issues, such as KAFKA-1894, HADOOP-10622,
98           // will hang forever if some methods are interrupted.
99           new UninterruptibleThread(r, "unused") // thread name will be set by ThreadFactoryBuilder
100      })
101      .build()
102    Executors.newCachedThreadPool(threadFactory).asInstanceOf[ThreadPoolExecutor]
103  }
104  private val executorSource = new ExecutorSource(threadPool, executorId)
```

```scala
  private val maxResultSize = conf.get(MAX_RESULT_SIZE)

  // Maintains the list of running tasks.
  private val runningTasks = new ConcurrentHashMap[Long, TaskRunner]

  // Executor for the heartbeat task.
  private val heartbeater = ThreadUtils.newDaemonSingleThreadScheduledExecutor("driver-heartbeater")

  // must be initialized before running startDriverHeartbeat()
  private val heartbeatReceiverRef =
    RpcUtils.makeDriverRef(HeartbeatReceiver.ENDPOINT_NAME, conf, env.rpcEnv)

  /**
   * When an executor is unable to send heartbeats to the driver more than `HEARTBEAT_MAX_FAILURES`
   * times, it should kill itself. The default value is 60. It means we will retry to send
   * heartbeats about 10 minutes because the heartbeat interval is 10s.
   */
  private val HEARTBEAT_MAX_FAILURES = conf.getInt("spark.executor.heartbeat.maxFailures", 60)

  /**
   * Count the failure times of heartbeat. It should only be accessed in the heartbeat thread. Each
   * successful heartbeat will reset it to 0.
   */
  private var heartbeatFailures = 0

  startDriverHeartbeater()

  private[executor] def numRunningTasks: Int = runningTasks.size()

  def launchTask(context: ExecutorBackend, taskDescription: TaskDescription): Unit = {
    val tr = new TaskRunner(context, taskDescription)
    runningTasks.put(taskDescription.taskId, tr)
    threadPool.execute(tr)
  }
```

```scala
    /**
     * Schedules a task to report heartbeat and partial metrics for active tasks to driver.
     */
    private def startDriverHeartbeater(): Unit = {
      val intervalMs = conf.getTimeAsMs("spark.executor.heartbeatInterval", "10s")

      // Wait a random interval so the heartbeats don't end up in sync
      val initialDelay = intervalMs + (math.random * intervalMs).asInstanceOf[Int]

      val heartbeatTask = new Runnable() {
        override def run(): Unit = Utils.logUncaughtExceptions(reportHeartBeat())
      }
      heartbeater.scheduleAtFixedRate(heartbeatTask, initialDelay, intervalMs, TimeUnit.MILLISECONDS)
    }
  }

  private[spark] object Executor {
    // This is reserved for internal use by components that need to read task properties before a
    // task is fully deserialized. When possible, the TaskContext.getLocalProperty call should be
    // used instead.
    val taskDeserializationProps: ThreadLocal[Properties] = new ThreadLocal[Properties]
  }
```

Creating an Executor Instance

Executor takes the following when created:

1. Executor ID
2. Executor's host name
3. SparkEnv
4. Collection of user-defined JARs (to add to tasks' class path). Empty by default
5. Boolean Flag whether it runs in local or cluster mode (disabled by default, i.e. cluster is preferred)
6. `SparkUncaughtExceptionHandler`

Note	User-defined JARs are defined using `--user-class-path` command-line option of `CoarseGrainedExecutorBackend` that can be set using spark.executor.extraClassPath property.

Note	`isLocal` is enabled exclusively for LocalEndpoint (for Spark in local mode).

When created, you should see the following INFO messages in the logs:

```
INFO Executor: Starting executor ID [executorId] on host [executorHostname]
```

(only for non-local mode) `Executor` registers `ExecutorSource` and initializes the local `BlockManager`.

Note	`Executor` uses `SparkEnv` to access the local MetricsSystem and BlockManager.

`Executor` creates a task class loader (optionally with REPL support) that the current `Serializer` is requested to use (when deserializing task later).

Note	`Executor` uses `SparkEnv` to access the local `Serializer`.

`Executor` starts sending heartbeats and active tasks metrics.

`Executor` initializes the internal registries and counters in the meantime (not necessarily at the very end).

launchTask method

Launching Task — `launchTask` Method

```
launchTask(
  context: ExecutorBackend,
  taskId: Long,
  attemptNumber: Int,
  taskName: String,
  serializedTask: ByteBuffer): Unit
```

`launchTask` executes the input `serializedTask` task concurrently.

Internally, `launchTask` creates a TaskRunner, registers it in `runningTasks` internal registry (by `taskId`), and finally executes it on "Executor task launch worker" thread pool.

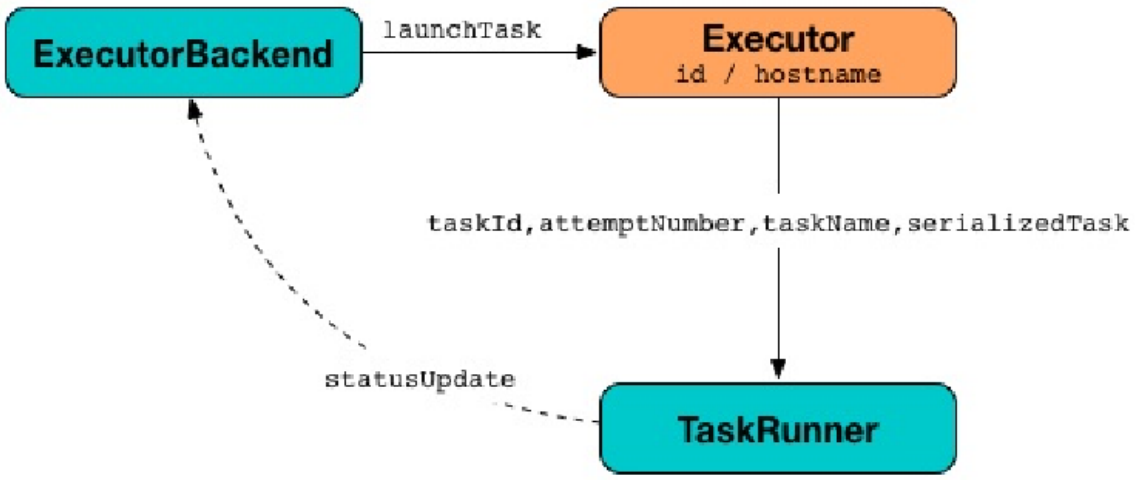


Figure 3. Launching tasks on executor using TaskRunners

Note	`launchTask` is called by CoarseGrainedExecutorBackend (when it handles LaunchTask message), MesosExecutorBackend, and LocalEndpoint.

startDriverHeartBeats

Sending Heartbeats and Active Tasks Metrics — `startDriverHeartbeater` Method

Executors keep sending metrics for active tasks to the driver every spark.executor.heartbeatInterval (defaults to `10s` with some random initial delay so the heartbeats from different executors do not pile up on the driver).

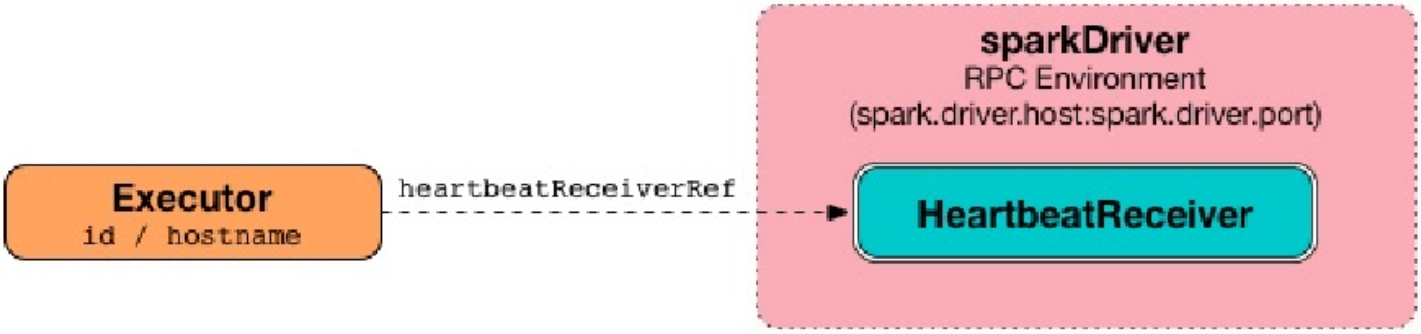


Figure 4. Executors use HeartbeatReceiver endpoint to report task metrics

An executor sends heartbeats using the internal heartbeater — Heartbeat Sender Thread.

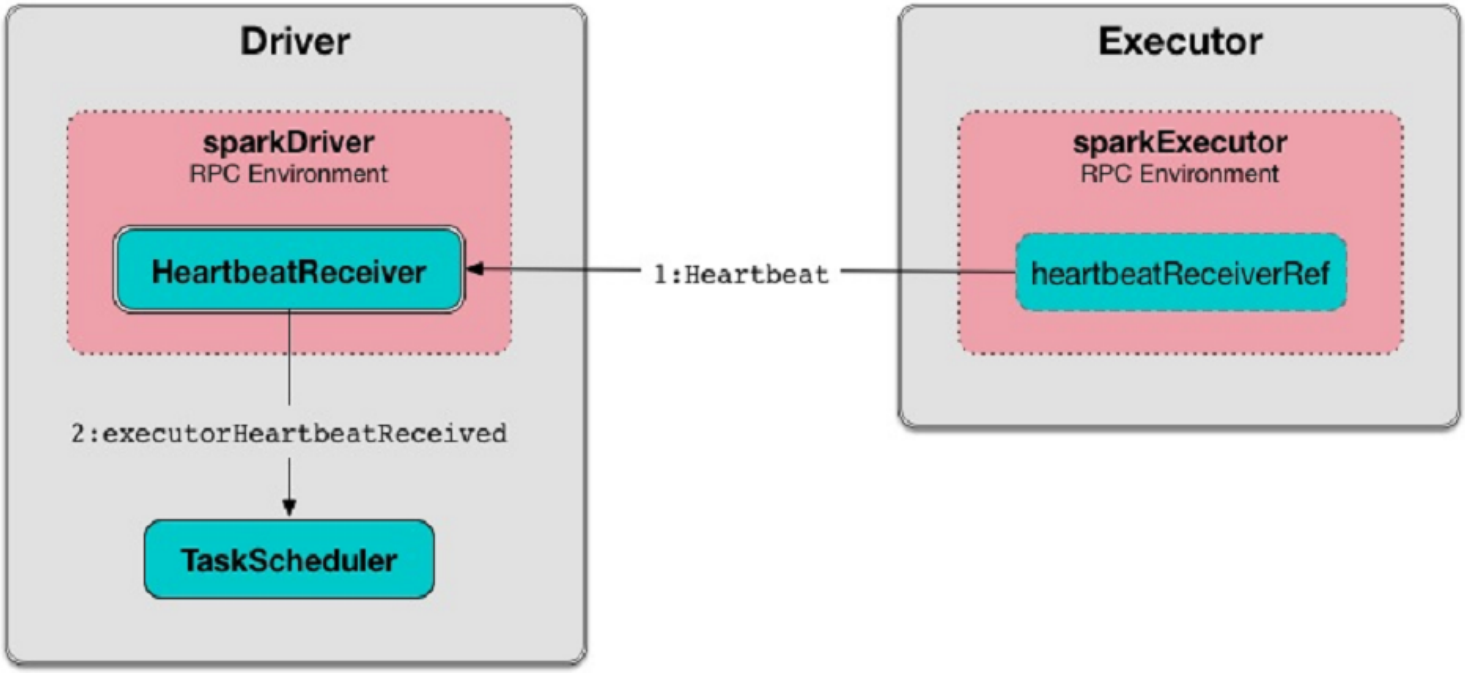


Figure 5. HeartbeatReceiver's Heartbeat Message Handler

For each task in TaskRunner (in runningTasks internal registry), the task's metrics are computed (i.e. `mergeShuffleReadMetrics` and `setJvmGCTime`) that become part of the heartbeat (with accumulators).

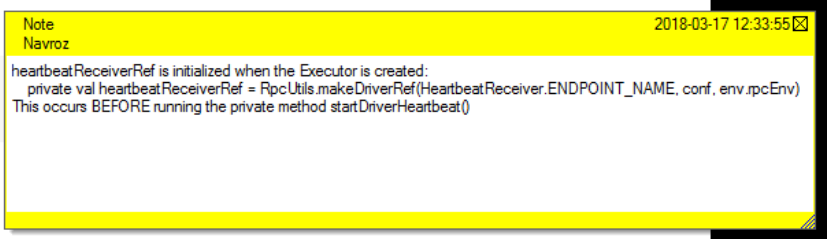
Note	Executors track the TaskRunner that run tasks. A task might not be assigned to a TaskRunner yet when the executor sends a heartbeat.

A blocking Heartbeat message that holds the executor id, all accumulator updates (per task id), and BlockManagerId is sent to HeartbeatReceiver RPC endpoint (with spark.executor.heartbeatInterval timeout).

Caution	FIXME When is `heartbeatReceiverRef` created?

If the response requests to reregister BlockManager, you should see the following INFO message in the logs:

```
INFO Executor: Told to re-register on heartbeat
```



The BlockManager is reregistered.

The internal heartbeatFailures counter is reset (i.e. becomes `0`).

If there are any issues with communicating with the driver, you should see the following WARN message in the logs:

```
WARN Executor: Issue communicating with driver in heartbeater
```

The internal heartbeatFailures is incremented and checked to be less than the acceptable number of failures (i.e. `spark.executor.heartbeat.maxFailures` Spark property). If the number is greater, the following ERROR is printed out to the logs:

```
ERROR Executor: Exit as unable to send heartbeats to driver more than [HEARTBEAT_MAX_F
AILURES] times
```

The executor exits (using `System.exit` and exit code 56).

reportHeartbeat

Reporting Heartbeat and Partial Metrics for Active Tasks to Driver — `reportHeartBeat` Internal Method

```
reportHeartBeat(): Unit
```

`reportHeartBeat` collects TaskRunners for currently running tasks (aka *active tasks*) with their tasks deserialized (i.e. either ready for execution or already started).

Note	TaskRunner has task deserialized when it runs the task.

For every running task, `reportHeartBeat` takes its TaskMetrics and:

- Requests ShuffleRead metrics to be merged
- Sets jvmGCTime metrics

`reportHeartBeat` then records the latest values of internal and external accumulators for every task.

Note	Internal accumulators are a task's metrics while external accumulators are a Spark application's accumulators that a user has created.

`reportHeartBeat` sends a blocking Heartbeat message to `HeartbeatReceiver` endpoint (running on the driver). `reportHeartBeat` uses spark.executor.heartbeatInterval for the RPC timeout.

Note	A `Heartbeat` message contains the executor identifier, the accumulator updates, and the identifier of the BlockManager.

Note	`reportHeartBeat` uses `SparkEnv` to access the current `BlockManager`.

If the response (from `HeartbeatReceiver` endpoint) is to re-register the `BlockManager`, you should see the following INFO message in the logs and `reportHeartBeat` requests `BlockManager` to re-register (which will register the blocks the `BlockManager` manages with the driver).

```
INFO Told to re-register on heartbeat
```

Note	`HeartbeatResponse` requests `BlockManager` to re-register when either TaskScheduler or HeartbeatReceiver know nothing about the executor.

When posting the `Heartbeat` was successful, `reportHeartBeat` resets heartbeatFailures internal counter.

Every failure `reportHeartBeat` increments heartbeat failures up to spark.executor.heartbeat.maxFailures Spark property. When the heartbeat failures reaches the maximum, you should see the following ERROR message in the logs and the executor terminates with the error code: `56`.

```
ERROR Exit as unable to send heartbeats to driver more than [HEARTBEAT_MAX_FAILURES] t
imes
```

Note	`reportHeartBeat` is used when `Executor` schedules reporting heartbeat and partial metrics for active tasks to the driver (that happens every spark.executor.heartbeatInterval Spark property).

heartbeater — Heartbeat Sender Thread

`heartbeater` is a daemon ScheduledThreadPoolExecutor with a single thread.

The name of the thread pool is **driver-heartbeater**.

Coarse-Grained Executors

Coarse-grained executors are executors that use CoarseGrainedExecutorBackend for task scheduling.

```
40 private[spark] class CoarseGrainedExecutorBackend(
41     override val rpcEnv: RpcEnv,
42     driverUrl: String,
43     executorId: String,
44     hostname: String,
45     cores: Int,
46     userClassPath: Seq[URL],
47     env: SparkEnv)
48   extends ThreadSafeRpcEndpoint with ExecutorBackend with Logging {
49
50   private[this] val stopping = new AtomicBoolean(false)
51   var executor: Executor = null
52   @volatile var driver: Option[RpcEndpointRef] = None
53
54   // If this CoarseGrainedExecutorBackend is changed to support multiple threads, then this may need
55   // to be changed so that we don't share the serializer instance across threads
56   private[this] val ser: SerializerInstance = env.closureSerializer.newInstance()
57
```

```
58   override def onStart() {
59     logInfo("Connecting to driver: " + driverUrl)
60     rpcEnv.asyncSetupEndpointRefByURI(driverUrl).flatMap { ref =>
61       // This is a very fast action so we can use "ThreadUtils.sameThread"
62       driver = Some(ref)
63       ref.ask[Boolean](RegisterExecutor(executorId, self, hostname, cores, extractLogUrls))
64     }(ThreadUtils.sameThread).onComplete {
65       // This is a very fast action so we can use "ThreadUtils.sameThread"
66       case Success(msg) =>
67         // Always receive `true`. Just ignore it
68       case Failure(e) =>
69         exitExecutor(1, s"Cannot register with driver: $driverUrl", e, notifyDriver = false)
70     }(ThreadUtils.sameThread)
71   }
72
73   def extractLogUrls: Map[String, String] = {
74     val prefix = "SPARK_LOG_URL_"
75     sys.env.filterKeys(_.startsWith(prefix))
76       .map(e => (e._1.substring(prefix.length).toLowerCase(Locale.ROOT), e._2))
77   }
```

```scala
78
79  override def receive: PartialFunction[Any, Unit] = {
80    case RegisteredExecutor =>
81      logInfo("Successfully registered with driver")
82      try {
83        executor = new Executor(executorId, hostname, env, userClassPath, isLocal = false)
84      } catch {
85        case NonFatal(e) =>
86          exitExecutor(1, "Unable to create executor due to " + e.getMessage, e)
87      }
88
89    case RegisterExecutorFailed(message) =>
90      exitExecutor(1, "Slave registration failed: " + message)
91
92    case LaunchTask(data) =>
93      if (executor == null) {
94        exitExecutor(1, "Received LaunchTask command but executor was null")
95      } else {
96        val taskDesc = TaskDescription.decode(data.value)
97        logInfo("Got assigned task " + taskDesc.taskId)
98        executor.launchTask(this, taskDesc)
99      }
100
101   case KillTask(taskId, _, interruptThread, reason) =>
102     if (executor == null) {
103       exitExecutor(1, "Received KillTask command but executor was null")
104     } else {
105       executor.killTask(taskId, interruptThread, reason)
106     }
107
108   case StopExecutor =>
109     stopping.set(true)
110     logInfo("Driver commanded a shutdown")
111     // Cannot shutdown here because an ack may need to be sent back to the caller. So send
112     // a message to self to actually do the shutdown.
113     self.send(Shutdown)
114
115   case Shutdown =>
116     stopping.set(true)
117     new Thread("CoarseGrainedExecutorBackend-stop-executor") {
118       override def run(): Unit = {
119         // executor.stop() will call `SparkEnv.stop()` which waits until RpcEnv stops totally.
120         // However, if `executor.stop()` runs in some thread of RpcEnv, RpcEnv won't be able to
121         // stop until `executor.stop()` returns, which becomes a dead-lock (See SPARK-14180).
122         // Therefore, we put this line in a new thread.
123         executor.stop()
124       }
125     }.start()
126
127   case UpdateDelegationTokens(tokenBytes) =>
128     logInfo(s"Received tokens of ${tokenBytes.length} bytes")
129     SparkHadoopUtil.get.addDelegationTokens(tokenBytes, env.conf)
130 }
131
132 override def onDisconnected(remoteAddress: RpcAddress): Unit = {
133   if (stopping.get()) {
134     logInfo(s"Driver from $remoteAddress disconnected during shutdown")
135   } else if (driver.exists(_.address == remoteAddress)) {
136     exitExecutor(1, s"Driver $remoteAddress disassociated! Shutting down.", null,
137       notifyDriver = false)
138   } else {
139     logWarning(s"An unknown ($remoteAddress) driver disconnected.")
140   }
141 }
142
```

TaskRunner

This is an *internal private class* of `org.apache.spark.executor.Executor`

TaskRunner

`TaskRunner` is a thread of execution that manages a single individual task.

`TaskRunner` is created exclusively when `Executor` is requested to launch a task.

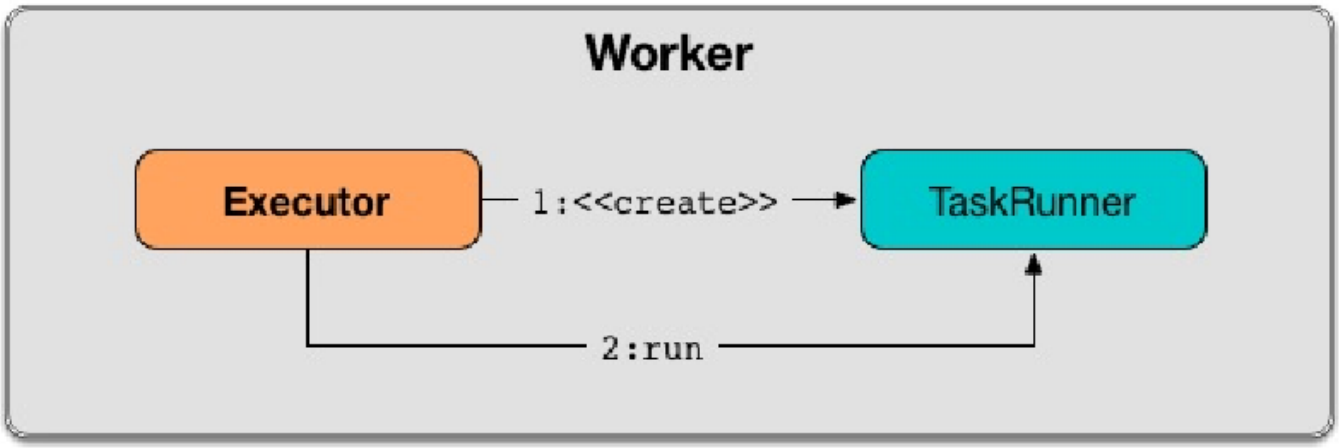


Figure 1. Executor creates TaskRunner and runs (almost) immediately

```
232  class TaskRunner(
233      execBackend: ExecutorBackend,
234      private val taskDescription: TaskDescription)
235    extends Runnable {
236
237    val taskId = taskDescription.taskId
238    val threadName = s"Executor task launch worker for task $taskId"
239    private val taskName = taskDescription.name
240
241    /** If specified, this task has been killed and this option contains the reason. */
242    @volatile private var reasonIfKilled: Option[String] = None
243
244    @volatile private var threadId: Long = -1
245
246    def getThreadId: Long = threadId
247
248    /** Whether this task has been finished. */
249    @GuardedBy("TaskRunner.this")
250    private var finished = false
251
252    def isFinished: Boolean = synchronized { finished }
253
254    /** How much the JVM process has spent in GC when the task starts to run. */
255    @volatile var startGCTime: Long = _
256
257    /**
258     * The task to run. This will be set in run() by deserializing the task binary coming
259     * from the driver. Once it is set, it will never be changed.
260     */
261    @volatile var task: Task[Any] = _
262
263    def kill(interruptThread: Boolean, reason: String): Unit = {
264      logInfo(s"Executor is trying to kill $taskName (TID $taskId), reason: $reason")
265      reasonIfKilled = Some(reason)
```

A `TaskRunner` object is created when an executor is requested to launch a task.

It is created with an ExecutorBackend (to send the task's status updates to), task and attempt ids, task name, and serialized version of the task (as `ByteBuffer`).

long-ass run method

SparkContext - The main entry point to Spark functionality

Anatomy of Spark Application

Every Spark application starts from creating SparkContext.

Note	Without SparkContext no computation (as a Spark job) can be started.

Note	A Spark application is an instance of SparkContext. Or, put it differently, a Spark context constitutes a Spark application.

A Spark application is uniquely identified by a pair of the application and application attempt ids.

For it to work, you have to create a Spark configuration using SparkConf or use a custom SparkContext constructor.

Tip	Spark shell creates a Spark context and SQL context for you at startup.

When a Spark application starts (using spark-submit script or as a standalone application), it connects to Spark master as described by master URL. It is part of Spark context's initialization.

Note	Your Spark application can run locally or on the cluster which is based on the cluster manager and the deploy mode (`--deploy-mode`). Refer to Deployment Modes.

You can then create RDDs, transform them to other RDDs and ultimately execute actions. You can also cache interim RDDs to speed up data processing.

After all the data processing is completed, the Spark application finishes by stopping the Spark context.

SparkContext — Entry Point to Spark Core

`SparkContext` (aka **Spark context**) is the heart of a Spark application.

Note	You could also assume that a SparkContext instance *is* a Spark application.

Spark context sets up internal services and establishes a connection to a Spark execution environment.

Once a `SparkContext` is created you can use it to create RDDs, accumulators and broadcast variables, access Spark services and run jobs (until `SparkContext` is stopped).

A Spark context is essentially a client of Spark's execution environment and acts as the *master of your Spark application* (don't get confused with the other meaning of Master in Spark, though).

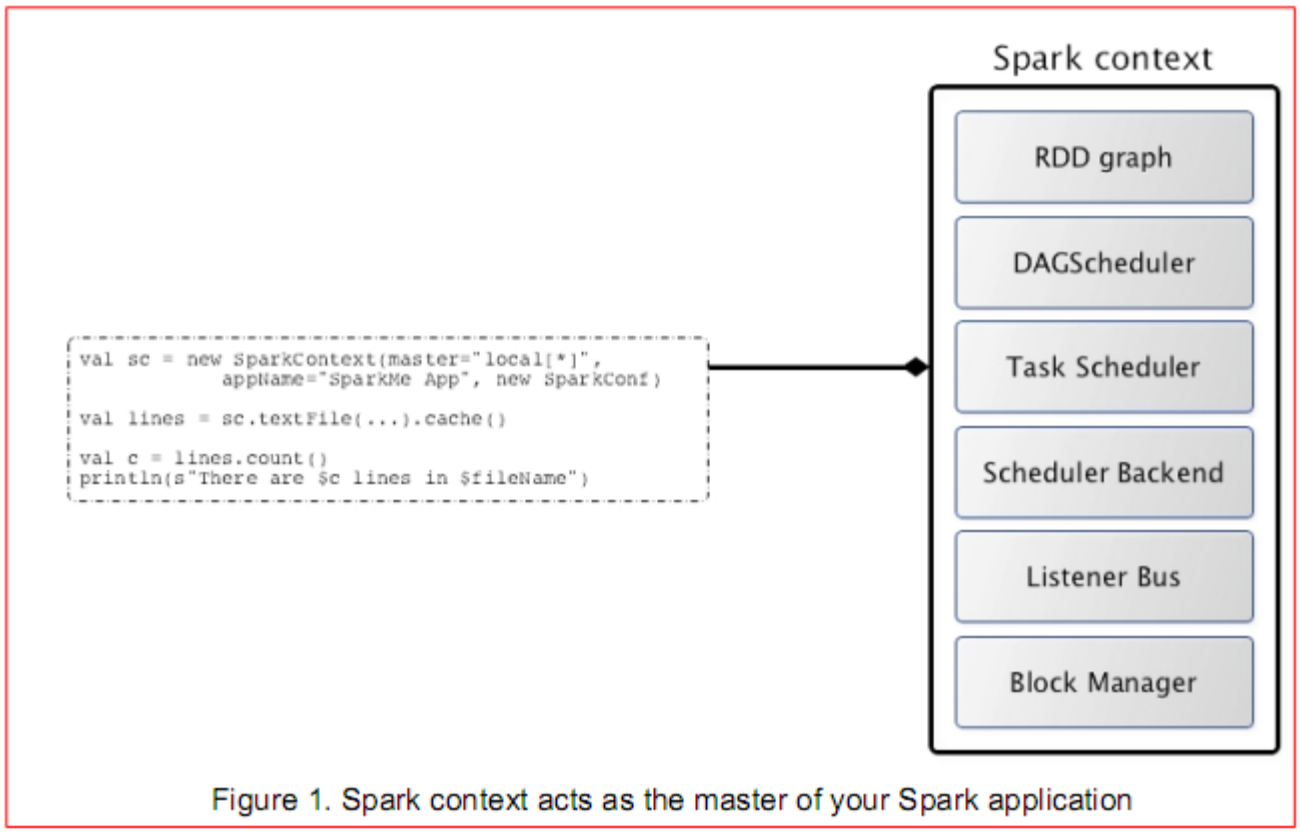


Figure 1. Spark context acts as the master of your Spark application

SparkContext Constructors

Constructors

```
SparkContext()
SparkContext(conf: SparkConf)
SparkContext(master: String, appName: String, conf: SparkConf)
SparkContext(
  master: String,
  appName: String,
  sparkHome: String = null,
  jars: Seq[String] = Nil,
  environment: Map[String, String] = Map())
```

You can create a `SparkContext` instance using the four constructors.

When a Spark context starts up you should see the following INFO in the logs (amongst the other messages that come from the Spark services):

```
INFO SparkContext: Running Spark version 2.0.0-SNAPSHOT
```

Note	Only one SparkContext may be running in a single JVM (check out SPARK-2243 Support multiple SparkContexts in the same JVM). Sharing access to a SparkContext in the JVM is the solution to share data within Spark (without relying on other means of data sharing using external data stores).

submitJob - to submit jobs Asynchronously

Submitting Jobs Asynchronously — `submitJob` Method

```
submitJob[T, U, R](
  rdd: RDD[T],
  processPartition: Iterator[T] => U,
  partitions: Seq[Int],
  resultHandler: (Int, U) => Unit,
  resultFunc: => R): SimpleFutureAction[R]
```

`submitJob` submits a job in an asynchronous, non-blocking way to DAGScheduler.

It cleans the `processPartition` input function argument and returns an instance of SimpleFutureAction that holds the JobWaiter instance.

It is used in:

- AsyncRDDActions methods
- Spark Streaming for ReceiverTrackerEndpoint.startReceiver

runJob - to run jobs Sychronously

RDD actions run jobs using one of `runJob` methods.

```
runJob[T, U](
  rdd: RDD[T],
  func: (TaskContext, Iterator[T]) => U,
  partitions: Seq[Int],
  resultHandler: (Int, U) => Unit): Unit
runJob[T, U](
  rdd: RDD[T],
  func: (TaskContext, Iterator[T]) => U,
  partitions: Seq[Int]): Array[U]
runJob[T, U](
  rdd: RDD[T],
  func: Iterator[T] => U,
  partitions: Seq[Int]): Array[U]
runJob[T, U](rdd: RDD[T], func: (TaskContext, Iterator[T]) => U): Array[U]
runJob[T, U](rdd: RDD[T], func: Iterator[T] => U): Array[U]
runJob[T, U](
  rdd: RDD[T],
  processPartition: (TaskContext, Iterator[T]) => U,
  resultHandler: (Int, U) => Unit)
runJob[T, U: ClassTag](
  rdd: RDD[T],
  processPartition: Iterator[T] => U,
  resultHandler: (Int, U) => Unit)
```

`runJob` executes a function on one or many partitions of a RDD (in a `SparkContext` space) to produce a collection of values per partition.

Note	`runJob` can only work when a `SparkContext` is *not* stopped.

Internally, `runJob` first makes sure that the `SparkContext` is not stopped. If it is, you should see the following `IllegalStateException` exception in the logs:

```
java.lang.IllegalStateException: SparkContext has been shutdown
  at org.apache.spark.SparkContext.runJob(SparkContext.scala:1893)
  at org.apache.spark.SparkContext.runJob(SparkContext.scala:1914)
  at org.apache.spark.SparkContext.runJob(SparkContext.scala:1934)
  ... 48 elided
```

`runJob` then calculates the call site and cleans a `func` closure.

`runJob` requests `DAGScheduler` to run a job.

Tip	`runJob` just prepares input parameters for `DAGScheduler` to run a job.

After `DAGScheduler` is done and the job has finished, `runJob` stops `ConsoleProgressBar` and performs RDD checkpointing of `rdd`.

Tip	For some actions, e.g. `first()` and `lookup()`, there is no need to compute all the partitions of the RDD in a job. And Spark knows it.

```
// RDD to work with
val lines = sc.parallelize(Seq("hello world", "nice to see you"))

import org.apache.spark.TaskContext
scala> sc.runJob(lines, (t: TaskContext, i: Iterator[String]) => 1) (1)
res0: Array[Int] = Array(1, 1)  (2)
```

Running a job is essentially executing a `func` function on all or a subset of partitions in an `rdd` RDD and returning the result as an array (with elements being the results per partition).

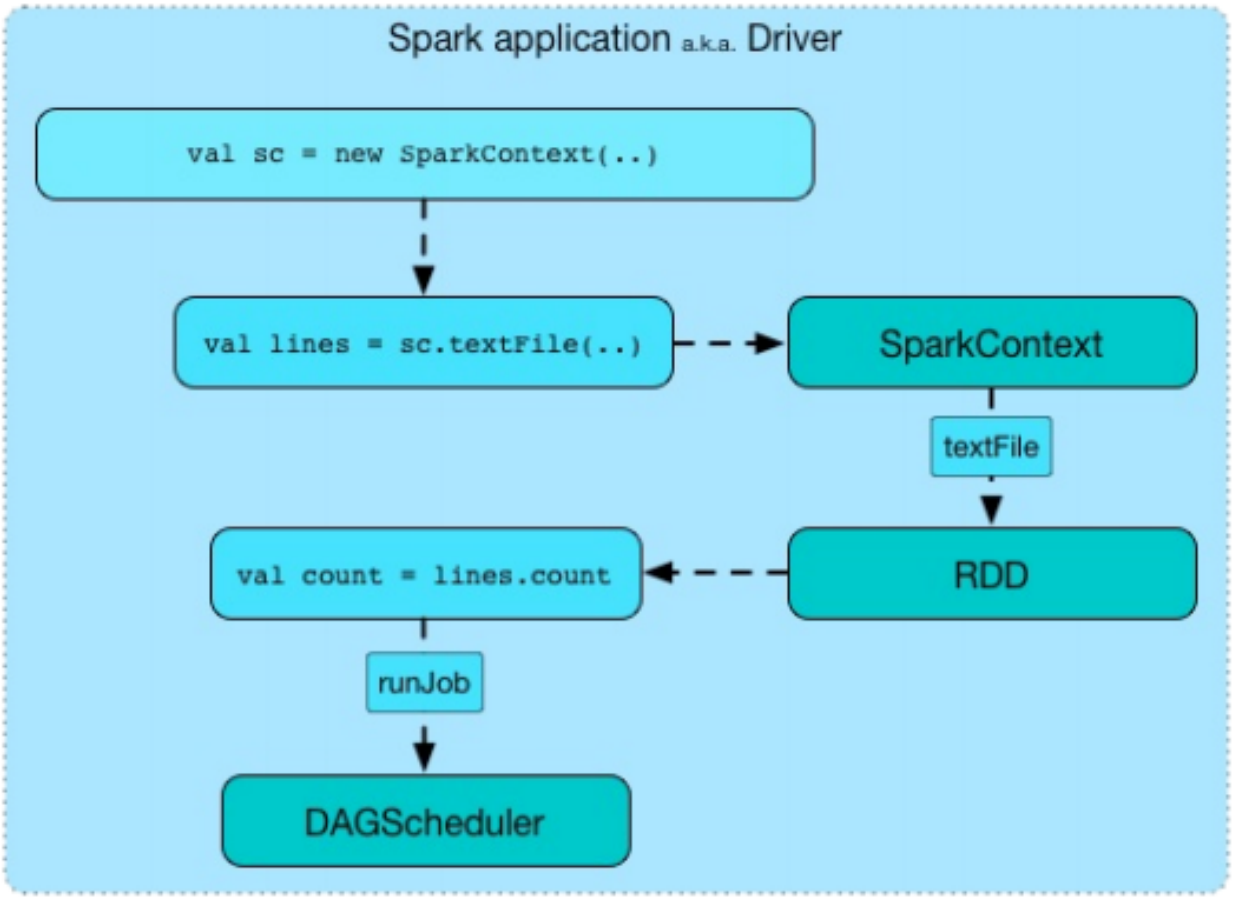


Figure 6. Executing action

stop

Stopping `SparkContext` — `stop` Method

```
stop(): Unit
```

`stop` stops the `SparkContext`.

Internally, `stop` enables `stopped` internal flag. If already stopped, you should see the following INFO message in the logs:

```
INFO SparkContext: SparkContext already stopped.
```

`stop` then does the following:

1. Removes `_shutdownHookRef` from `ShutdownHookManager`
2. Posts a `SparkListenerApplicationEnd` (to LiveListenerBus Event Bus)
3. Stops web UI
4. Requests `MetricsSystem` to report metrics (from all registered sinks)
5. Stops `ContextCleaner`
6. Requests `ExecutorAllocationManager` to stop
7. If `LiveListenerBus` was started, requests `LiveListenerBus` to stop
8. Requests `EventLoggingListener` to stop
9. Requests `DAGScheduler` to stop
10. Requests RpcEnv to stop `HeartbeatReceiver` endpoint
11. Requests `ConsoleProgressBar` to stop
12. Clears the reference to `TaskScheduler`, i.e. `_taskScheduler` is `null`
13. Requests `SparkEnv` to stop and clears `SparkEnv`
14. Clears `SPARK_YARN_MODE` flag
15. Clears an active `SparkContext`

Ultimately, you should see the following INFO message in the logs:

```
INFO SparkContext: Successfully stopped SparkContext
```

parallelize - Creating RDDs

Creating RDD — `parallelize` Method

`SparkContext` allows you to create many different RDDs from input sources like:

- Scala's collections, i.e. `sc.parallelize(0 to 100)`
- local or remote filesystems, i.e. `sc.textFile("README.md")`
- Any Hadoop `InputSource` using `sc.newAPIHadoopFile`

unpersist - marking RDDs as Non-Persistent

Unpersisting RDD (Marking RDD as Non-Persistent) — `unpersist` Method

Caution	FIXME

`unpersist` removes an RDD from the master's Block Manager (calls `removeRdd(rddId: Int, blocking: Boolean)`) and the internal persistentRdds mapping.

It finally posts SparkListenerUnpersistRDD message to `listenerBus`.

Creating Built-In Accumulators

```
longAccumulator: LongAccumulator
longAccumulator(name: String): LongAccumulator
doubleAccumulator: DoubleAccumulator
doubleAccumulator(name: String): DoubleAccumulator
collectionAccumulator[T]: CollectionAccumulator[T]
collectionAccumulator[T](name: String): CollectionAccumulator[T]
```

You can use `longAccumulator`, `doubleAccumulator` or `collectionAccumulator` to create and register accumulators for simple and collection values.

`longAccumulator` returns LongAccumulator with the zero value `0`.

`doubleAccumulator` returns DoubleAccumulator with the zero value `0.0`.

`collectionAccumulator` returns CollectionAccumulator with the zero value `java.util.List[T]`.

```
scala> val acc = sc.longAccumulator
acc: org.apache.spark.util.LongAccumulator = LongAccumulator(id: 0, name: None, value: 0)

scala> val counter = sc.longAccumulator("counter")
counter: org.apache.spark.util.LongAccumulator = LongAccumulator(id: 1, name: Some(counter), value: 0)

scala> counter.value
res0: Long = 0

scala> sc.parallelize(0 to 9).foreach(n => counter.add(n))

scala> counter.value
res3: Long = 45
```

Creating Broadcast Variables

Creating Broadcast Variable — `broadcast` Method

```
broadcast[T](value: T): Broadcast[T]
```

`broadcast` method creates a broadcast variable. It is a shared memory with `value` (as broadcast blocks) on the driver and later on all Spark executors.

```
val sc: SparkContext = ???
scala> val hello = sc.broadcast("hello")
hello: org.apache.spark.broadcast.Broadcast[String] = Broadcast(0)
```

Spark transfers the value to Spark executors *once*, and tasks can share it without incurring repetitive network transmissions when the broadcast variable is used multiple times.

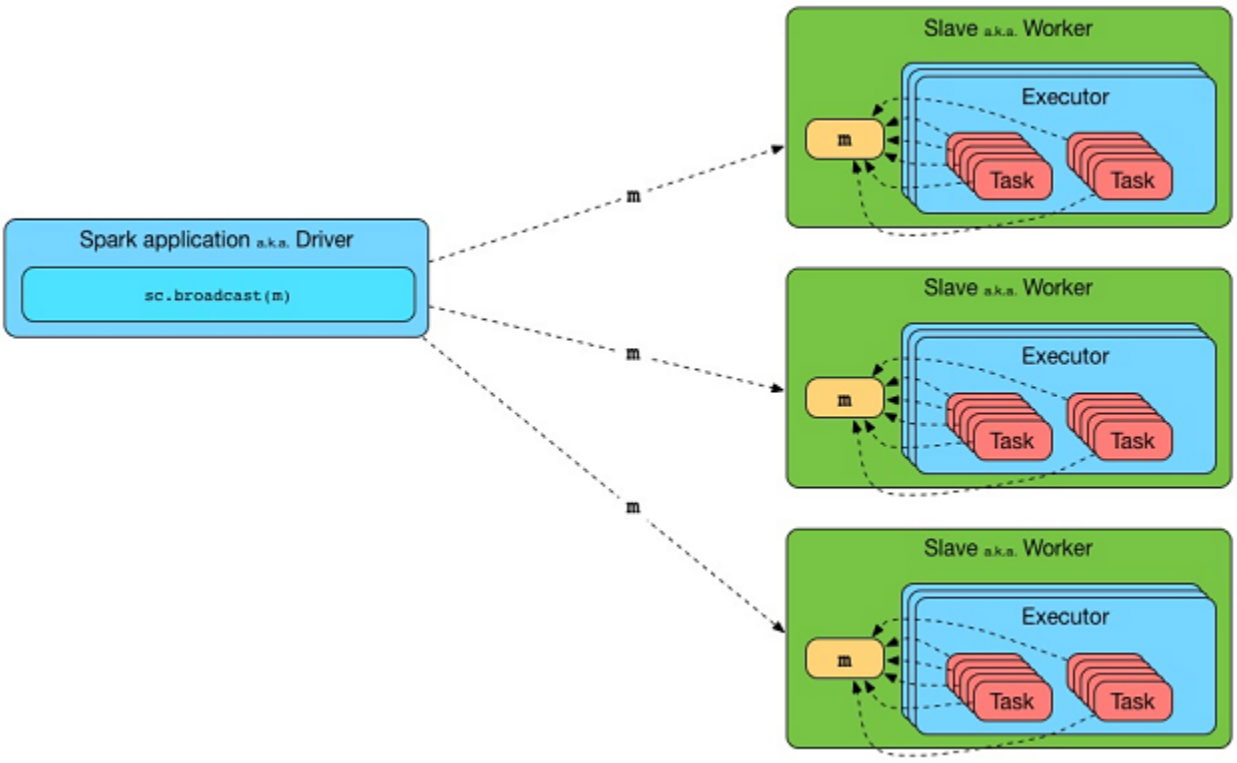


Figure 4. Broadcasting a value to executors

Internally, `broadcast` requests the current `BroadcastManager` to create a new broadcast variable.

Note	The current `BroadcastManager` is available using `SparkEnv.broadcastManager` attribute and is always BroadcastManager (with few internal configuration changes to reflect where it runs, i.e. inside the driver or executors).

You should see the following INFO message in the logs:

```
INFO SparkContext: Created broadcast [id] from [callSite]
```

If `ContextCleaner` is defined, the new broadcast variable is registered for cleanup.

Note	Spark does not support broadcasting RDDs.
	`scala> sc.broadcast(sc.range(0, 10))`
	`java.lang.IllegalArgumentException: requirement failed: Can not directly broadc`
	`  at scala.Predef$.require(Predef.scala:224)`
	`  at org.apache.spark.SparkContext.broadcast(SparkContext.scala:1392)`
	`  ... 48 elided`

Once created, the broadcast variable (and other blocks) are displayed per executor and the driver in web UI (under Executors tab).

Inside creation of a SparkContext

```
import org.apache.spark.{SparkConf, SparkContext}

// 1. Create Spark configuration
val conf = new SparkConf()
  .setAppName("SparkMe Application")
  .setMaster("local[*]")  // local mode

// 2. Create Spark context
val sc = new SparkContext(conf)
```

Creating `SparkContext` instance starts by setting the internal `allowMultipleContexts` field with the value of spark.driver.allowMultipleContexts and marking this `SparkContext` instance as partially constructed. It makes sure that no other thread is creating a `SparkContext` instance in this JVM. It does so by synchronizing on `SPARK_CONTEXT_CONSTRUCTOR_LOCK` and using the internal atomic reference `activeContext` (that eventually has a fully-created `SparkContext` instance).

Note	The entire code of `SparkContext` that creates a fully-working `SparkContext` instance is between two statements:

```
SparkContext.markPartiallyConstructed(this, allowMultipleContexts)

// the SparkContext code goes here

SparkContext.setActiveContext(this, allowMultipleContexts)
```

createTaskScheduler

The private `createTaskScheduler` is executed as part of creating an instance of SparkContext to create TaskScheduler and SchedulerBackend objects.

It uses the master URL to select right implementations.

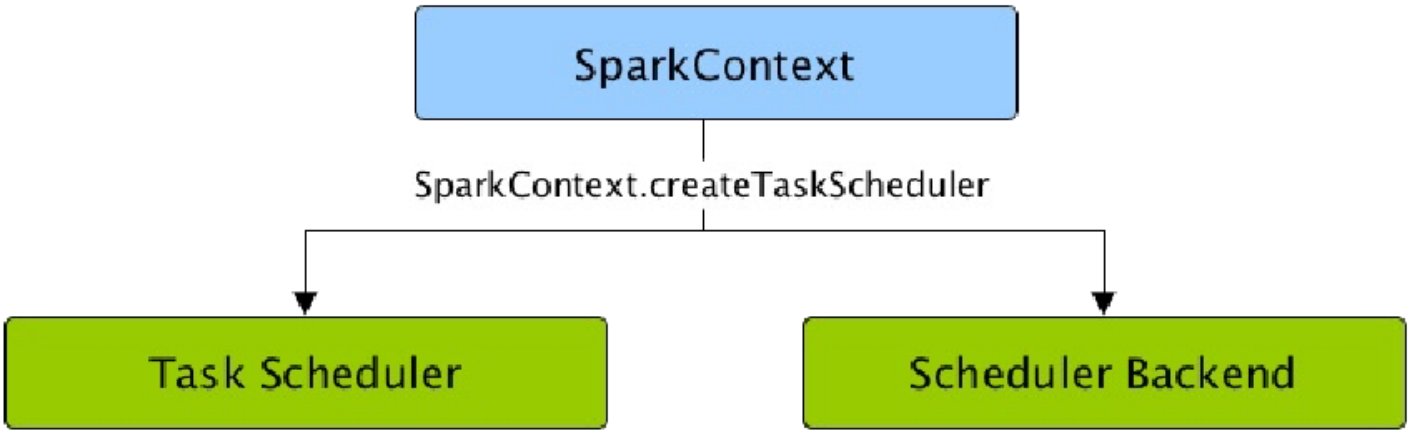


Figure 1. SparkContext creates Task Scheduler and Scheduler Backend

`createTaskScheduler` understands the following master URLs:

- `local` - local mode with 1 thread only
- `local[n]` or `local[*]` - local mode with `n` threads.
- `local[n, m]` or `local[*, m]` — local mode with `n` threads and `m` number of failures.
- `spark://hostname:port` for Spark Standalone.
- `local-cluster[n, m, z]` — local cluster with `n` workers, `m` cores per worker, and `z` memory per worker.
- `mesos://hostname:port` for Spark on Apache Mesos.
- any other URL is passed to `getClusterManager` to load an external cluster manager.

- **HeartbeatReceiver RPC Endpoint**

HeartbeatReceiver RPC Endpoint

`HeartbeatReceiver` is a ThreadSafeRpcEndpoint registered on the driver under the name **HeartbeatReceiver**.

`HeartbeatReceiver` receives Heartbeat messages from executors that Spark uses as the mechanism to receive accumulator updates (with task metrics and a Spark application's accumulators) and pass them along to `TaskScheduler`.

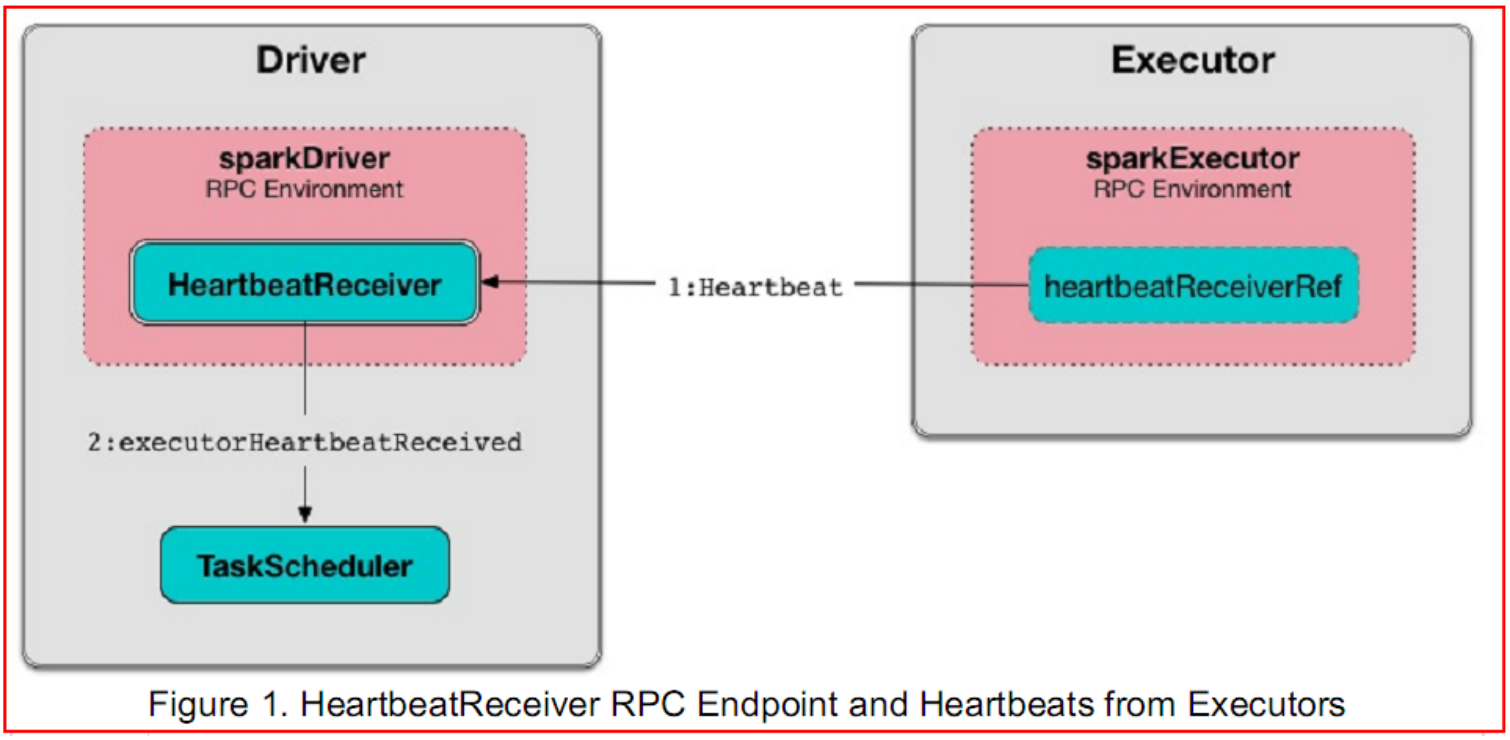


Figure 1. HeartbeatReceiver RPC Endpoint and Heartbeats from Executors

Note	`HeartbeatReceiver` is registered immediately after a Spark application is started, i.e. when `SparkContext` is created.

`HeartbeatReceiver` is a SparkListener to get notified when a new executor is added to or no longer available in a Spark application. `HeartbeatReceiver` tracks executors (in executorLastSeen registry) to handle Heartbeat and ExpireDeadHosts messages from executors that are assigned to the Spark application.

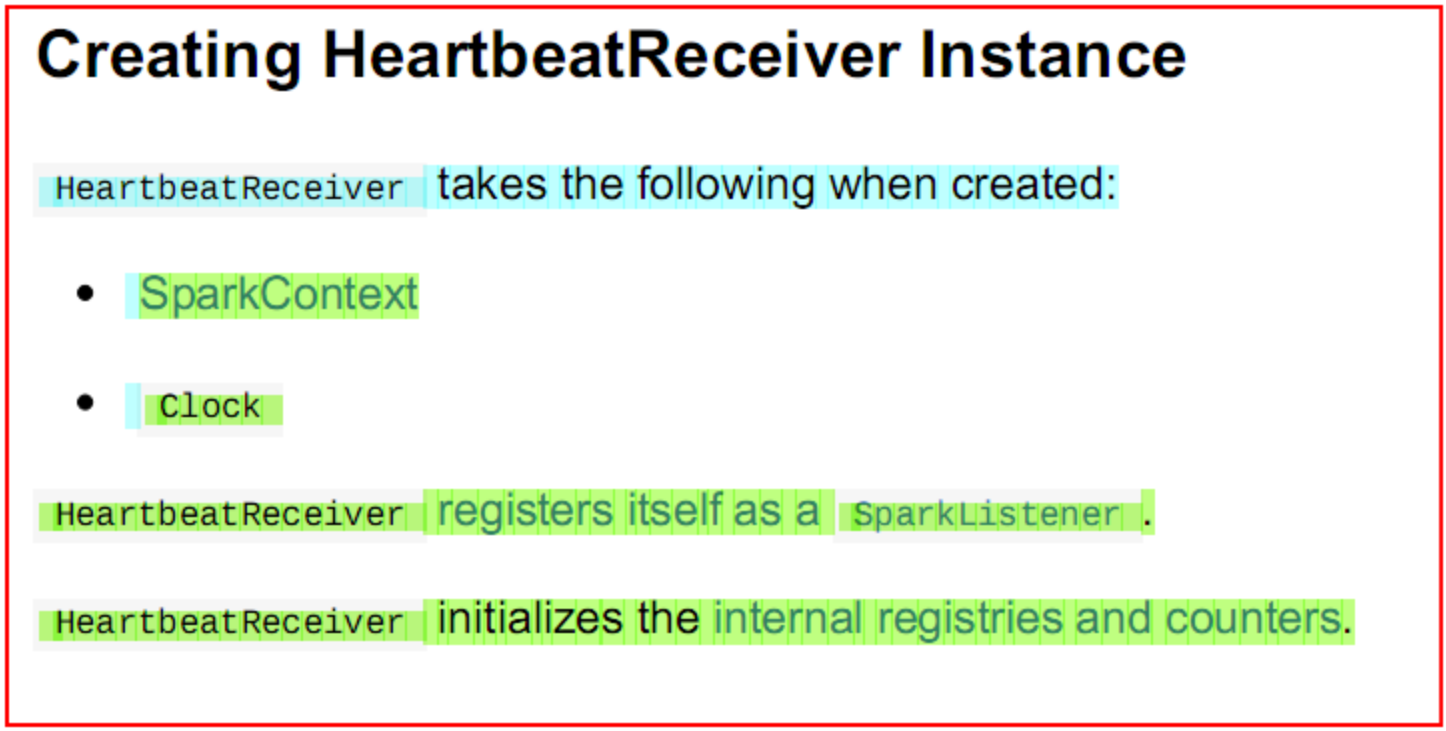
Table 1. HeartbeatReceiver RPC Endpoint's Messages (in alphabetical order)

Message	Description
ExecutorRemoved	Posted when `HeartbeatReceiver` is notified that an executor is no longer available (to a Spark application).
ExecutorRegistered	Posted when `HeartbeatReceiver` is notified that a new executor has been registered (with a Spark application).
ExpireDeadHosts	FIXME
Heartbeat	Posted when `Executor` informs that it is alive and reports task metrics.
TaskSchedulerIsSet	Posted when `SparkContext` informs that `TaskScheduler` is available.

Table 2. HeartbeatReceiver's Internal Registries and Counters

Name	Description
`executorLastSeen`	Executor ids and the timestamps of when the last heartbeat was received.
`scheduler`	TaskScheduler

Creating a HeartbeatReceiver Instance



Heartbeat

Heartbeat

```
Heartbeat(executorId: String,
  accumUpdates: Array[(Long, Seq[AccumulatorV2[_, _]])],
  blockManagerId: BlockManagerId)
```

When received, `HeartbeatReceiver` finds the `executorId` executor (in executorLastSeen registry).

When the executor is found, `HeartbeatReceiver` updates the time the heartbeat was received (in executorLastSeen).

Note	`HeartbeatReceiver` uses the internal Clock to know the current time.

`HeartbeatReceiver` then submits an asynchronous task to notify `TaskScheduler` that the heartbeat was received from the executor (using TaskScheduler internal reference). `HeartbeatReceiver` posts a `HeartbeatResponse` back to the executor (with the response from `TaskScheduler` whether the executor has been registered already or not so it may eventually need to re-register).

If however the executor was not found (in executorLastSeen registry), i.e. the executor was not registered before, you should see the following DEBUG message in the logs and the response is to notify the executor to re-register.

SparkConf - Spark Application Configuration

Spark Properties

Mandatory Settings - spark.master and spark.app.name

There are two mandatory settings of any Spark application that have to be defined before this Spark application could be run — spark.master and spark.app.name.

Spark Properties

Every user program starts with creating an instance of `SparkConf` that holds the master URL to connect to ([illegible]), the name for your Spark application (that is later

Setting up Spark Properties

There are the following places where a Spark application looks for Spark properties (in the order of importance from the least important to the most important):

- `conf/spark-defaults.conf` - the configuration file with the default Spark properties. Read spark-defaults.conf.
- `--conf` or `-c` - the command-line option used by spark-submit (and other shell scripts that use `spark-submit` or `spark-class` under the covers, e.g. `spark-shell`)
- `SparkConf`

```
spark.repl.class.uri=http://10.5.10.20:64055
spark.submit.deployMode=client
...
```

Use `sc.getConf.toDebugString` to have a richer output once SparkContext has finished initializing.

```
import org.apache.spark.SparkConf
val conf = new SparkConf
```

It simply loads `spark.*` system properties.

You can use `conf.toDebugString` or `conf.getAll` to have the `spark.*` system properties loaded printed out.

```
scala> conf.getAll
res0: Array[(String, String)] = Array((spark.app.name,Spark shell), (spark.jars,""), (
spark.master,local[*]), (spark.submit.deployMode,client))

scala> conf.toDebugString
res1: String =
spark.app.name=Spark shell
spark.jars=
spark.master=local[*]
spark.submit.deployMode=client

scala> println(conf.toDebugString)
spark.app.name=Spark shell
spark.jars=
spark.master=local[*]
spark.submit.deployMode=client
```

Unique Identifier of Spark Application — `getAppId` Method

```
getAppId: String
```

`getAppId` gives spark.app.id Spark property or reports `NoSuchElementException` if not set.

Note	`getAppId` is used when: • `NettyBlockTransferService` is initialized (and creates a NettyBlockRpcServer as well as saves the identifier for later use). • `Executor` is created (in non-local mode and requests `BlockManager` to initialize).

SparkEnv - the Spark Runtime Environment

Spark Runtime Environment (`SparkEnv`) is the runtime environment with Spark's public services that interact with each other to establish a distributed computing platform for a Spark application.

Spark Runtime Environment is represented by a SparkEnv object that holds all the required runtime services for a running Spark application with separate environments for the driver and executors.

The idiomatic way in Spark to access the current `SparkEnv` when on the driver or executors is to use get method.

```
import org.apache.spark._
scala> SparkEnv.get
res0: org.apache.spark.SparkEnv = org.apache.spark.SparkEnv@49322d04
```

SparkEnv Services & Internal Properties

Table 1. `SparkEnv` Services

Property	Service	Description
rpcEnv	RpcEnv	
serializer	Serializer	
closureSerializer	Serializer	
serializerManager	SerializerManager	
mapOutputTracker	MapOutputTracker	
shuffleManager	ShuffleManager	
broadcastManager	BroadcastManager	
blockManager	BlockManager	
securityManager	SecurityManager	
metricsSystem	MetricsSystem	
memoryManager	MemoryManager	

Table 2. SparkEnv's Internal Properties

Name	Initial Value	Description
`isStopped`	Disabled, i.e. `false`	Used to mark `SparkEnv` stopped. FIXME
`driverTmpDir`		

SparkEnv Factory-Object

Base --- SparkEnv - create method

Creating "Base" SparkEnv — create Method

```
create(
  conf: SparkConf,
  executorId: String,
  hostname: String,
  port: Int,
  isDriver: Boolean,
  isLocal: Boolean,
  numUsableCores: Int,
  listenerBus: LiveListenerBus = null,
  mockOutputCommitCoordinator: Option[OutputCommitCoordinator] = None): SparkEnv
```

`create` is a internal helper method to create a "base" `SparkEnv` regardless of the target environment, i.e. a driver or an executor.

Table 3. `create` 's Input Arguments and Their Usage

Input Argument	Usage
`bindAddress`	Used to create RpcEnv and NettyBlockTransferService.
`advertiseAddress`	Used to create RpcEnv and NettyBlockTransferService.
`numUsableCores`	Used to create MemoryManager, NettyBlockTransferService and BlockManager.

When executed, `create` creates a `Serializer` (based on spark.serializer setting). You should see the following `DEBUG` message in the logs:

```
DEBUG SparkEnv: Using serializer: [serializer]
```

It creates another `Serializer` (based on spark.closure.serializer).

It creates a ShuffleManager based on spark.shuffle.manager Spark property.

It creates a MemoryManager based on spark.memory.useLegacyMode setting (with UnifiedMemoryManager being the default and `numCores` the input `numUsableCores`).

`create` creates a NettyBlockTransferService. It uses `spark.driver.blockManager.port` for the port on the driver and `spark.blockManager.port` for the port on executors.

`create` creates a BlockManagerMaster object with the `BlockManagerMaster` RPC endpoint reference (by registering or looking it up by name and BlockManagerMasterEndpoint), the input SparkConf, and the input `isDriver` flag.

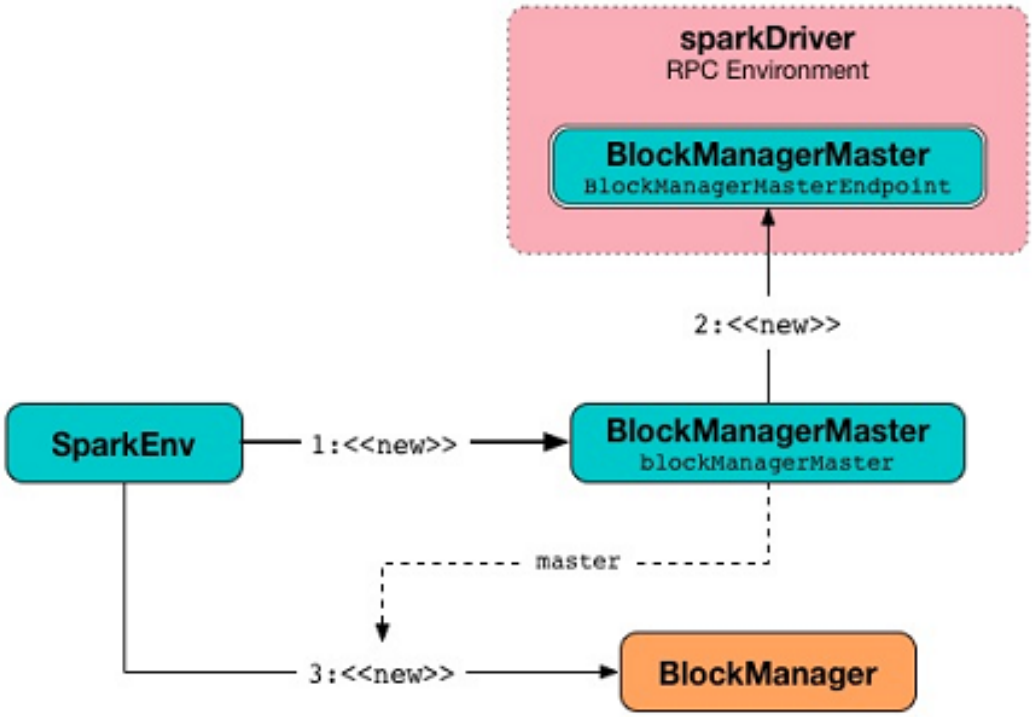


Figure 1. Creating BlockManager for the Driver

Note	`create` registers the **BlockManagerMaster** RPC endpoint for the driver and looks it up for executors.

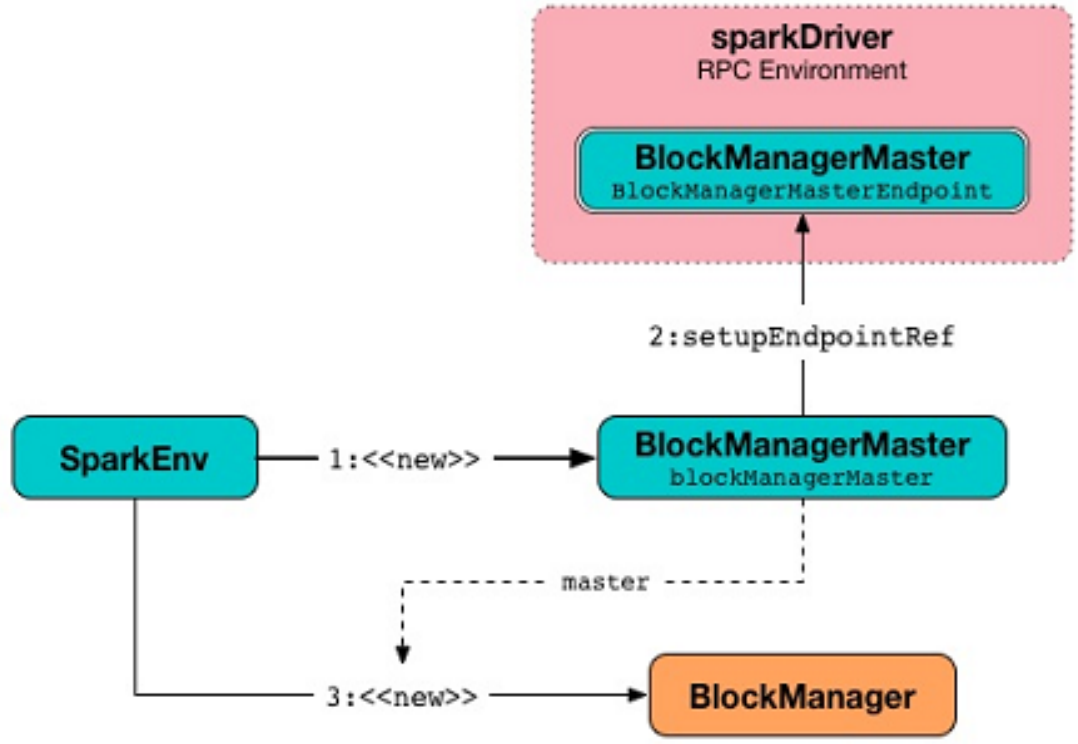


Figure 2. Creating BlockManager for Executor

It creates a BlockManager (using the above BlockManagerMaster, NettyBlockTransferService and other services).

`create` creates a BroadcastManager.

`create` creates a MapOutputTrackerMaster or MapOutputTrackerWorker for the driver and executors, respectively.

Note	The choice of the real implementation of MapOutputTracker is based on whether the input `executorId` is **driver** or not.

`create` registers or looks up `RpcEndpoint` as **MapOutputTracker**. It registers MapOutputTrackerMasterEndpoint on the driver and creates a RPC endpoint reference on executors. The RPC endpoint reference gets assigned as the MapOutputTracker RPC endpoint.

Caution	FIXME

It creates a CacheManager.

It creates a MetricsSystem for a driver and a worker separately.

It initializes `userFiles` temporary directory used for downloading dependencies for a driver while this is the executor's current working directory for an executor.

An OutputCommitCoordinator is created.

Note	`create` is called by createDriverEnv and createExecutorEnv.

registerOrLookupEndpoint method

Registering or Looking up RPC Endpoint by Name — `registerOrLookupEndpoint` Method

```
registerOrLookupEndpoint(name: String, endpointCreator: => RpcEndpoint)
```

`registerOrLookupEndpoint` registers or looks up a RPC endpoint by `name`.

If called from the driver, you should see the following INFO message in the logs:

```
INFO SparkEnv: Registering [name]
```

And the RPC endpoint is registered in the RPC environment.

Otherwise, it obtains a RPC endpoint reference by `name`.

createDriverEnv method

```
createDriverEnv(
  conf: SparkConf,
  isLocal: Boolean,
  listenerBus: LiveListenerBus,
  numCores: Int,
  mockOutputCommitCoordinator: Option[OutputCommitCoordinator] = None): SparkEnv
```

`createDriverEnv` creates a `SparkEnv` execution environment for the driver.

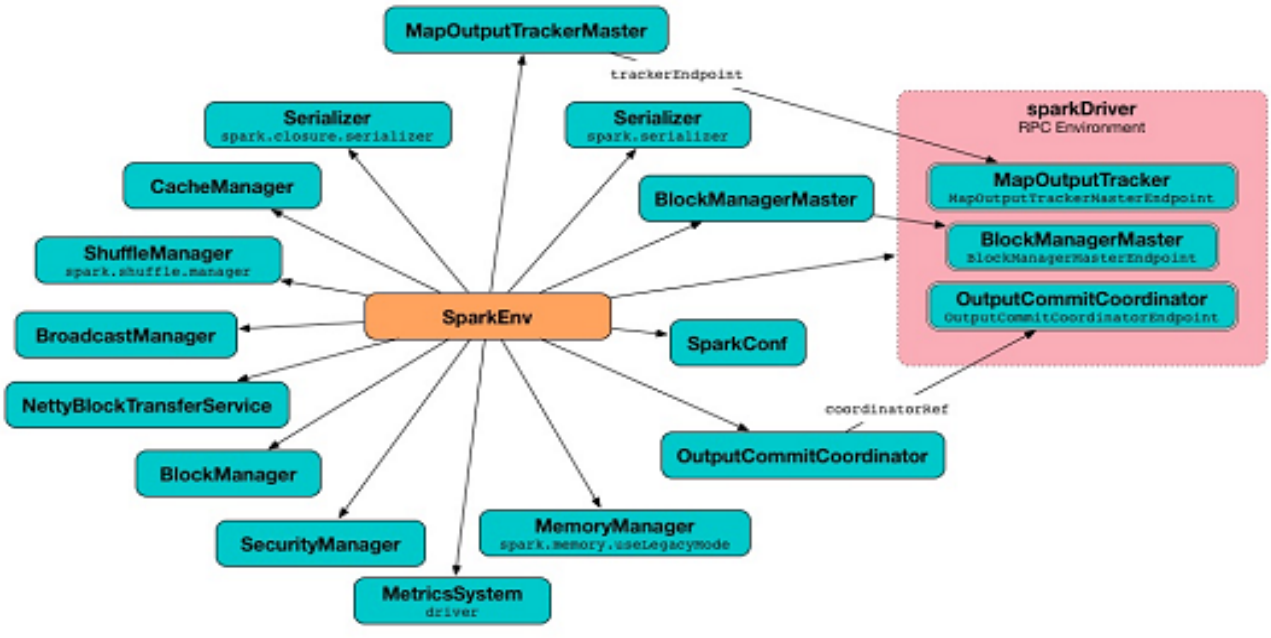


Figure 3. Spark Environment for driver

`createDriverEnv` accepts an instance of SparkConf, whether it runs in local mode or not, LiveListenerBus, the number of cores to use for execution in local mode or `0` otherwise, and a OutputCommitCoordinator (default: none).

`createDriverEnv` ensures that spark.driver.host and spark.driver.port settings are defined.

It then passes the call straight on to the create helper method (with `driver` executor id, `isDriver` enabled, and the input parameters).

Note	`createDriverEnv` is exclusively used by SparkContext to create a `SparkEnv` (while a SparkContext is being created for the driver).

createExecutorEnv method

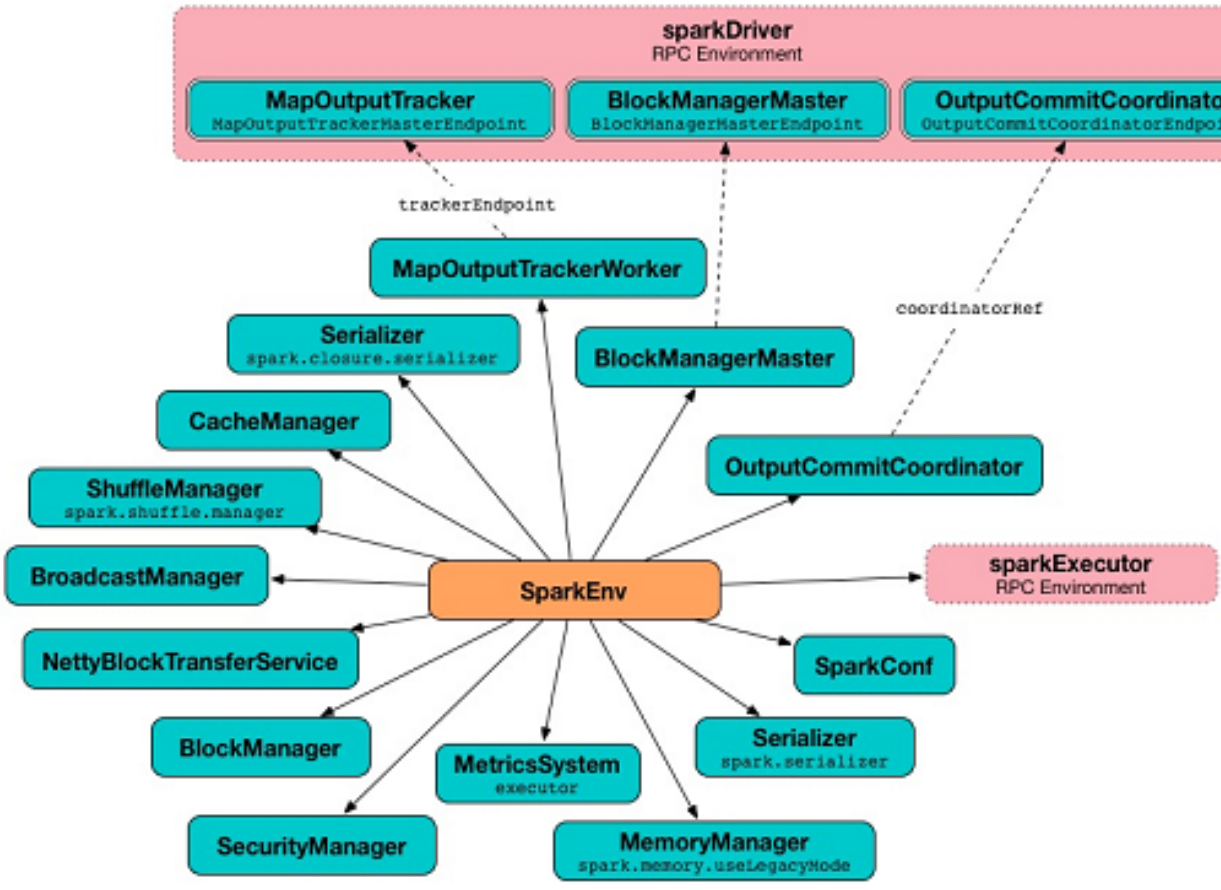


Figure 4. Spark Environment for executor

Note	`createExecutorEnv` is a `private[spark]` method.

`createExecutorEnv` simply creates the base `SparkEnv` (passing in all the input parameters) and sets it as the current `SparkEnv`.

Accesing current SparkEnv — get method

Getting Current SparkEnv — `get` Method

```
get: SparkEnv
```

`get` returns the current `SparkEnv`.

```
import org.apache.spark._
scala> SparkEnv.get
res0: org.apache.spark.SparkEnv = org.apache.spark.SparkEnv@49322d04
```

RDD — Resilient Distributed Dataset

Resilient Distributed Dataset (aka **RDD**) is the primary data abstraction in Apache Spark and the core of Spark (that I often refer to as "Spark Core").

The origins of RDD

The original paper that gave birth to the concept of RDD is Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing by Matei Zaharia, et al.

A RDD is a resilient and distributed collection of records spread over one or many partitions.

Note	One could compare RDDs to collections in Scala, i.e. a RDD is computed on many JVMs while a Scala collection lives on a single JVM.

Using RDD Spark hides data partitioning and so distribution that in turn allowed them to design parallel computational framework with a higher-level programming interface (API) for four mainstream programming languages.

namely Scala, Python, Java & R
c# module too Also for .NET?

The features of RDDs (decomposing the name):

- **Resilient**, i.e. fault-tolerant with the help of RDD lineage graph and so able to recompute missing or damaged partitions due to node failures.
- **Distributed** with data residing on multiple nodes in a cluster.
- **Dataset** is a collection of partitioned data with primitive values or values of values, e.g. tuples or other objects (that represent records of the data you work with).

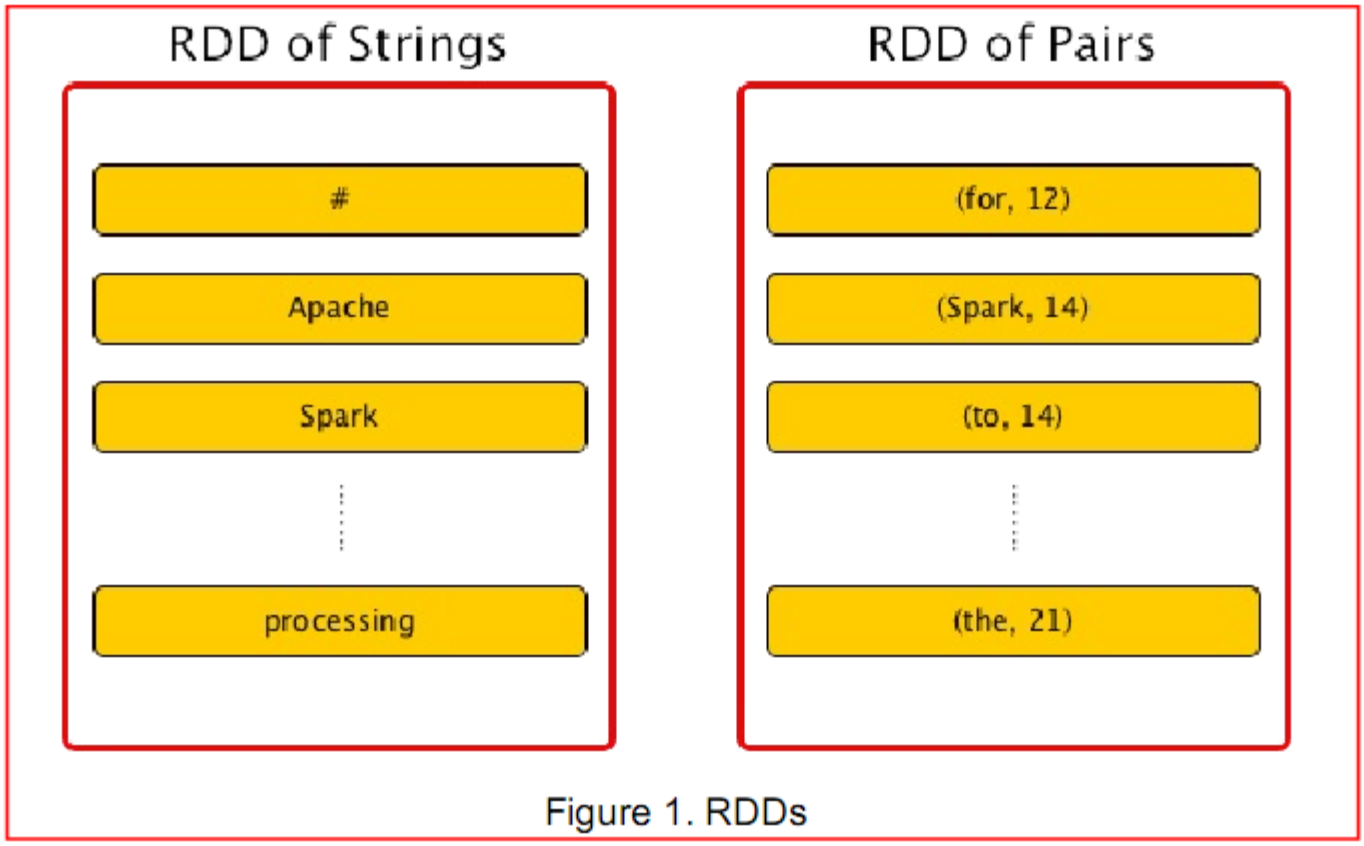


Figure 1. RDDs

From the scaladoc of org.apache.spark.rdd.RDD:

> A Resilient Distributed Dataset (RDD), the basic abstraction in Spark. Represents an immutable, partitioned collection of elements that can be operated on in parallel.

From the original paper about RDD - Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing:

> Resilient Distributed Datasets (RDDs) are a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner.

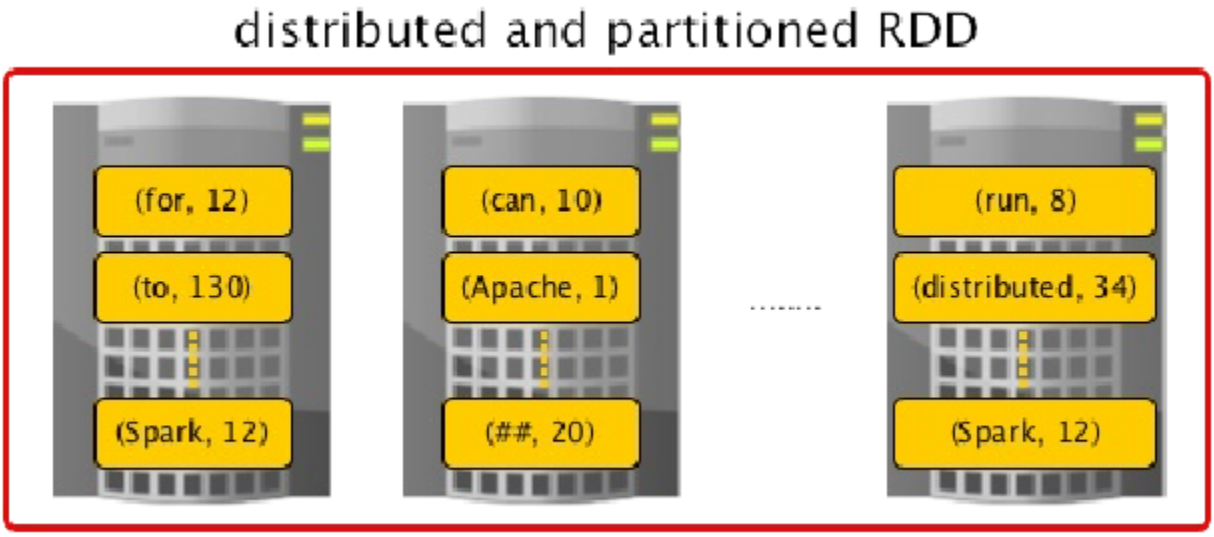
RDD - Additional Traits

Beside the above traits (that are directly embedded in the name of the data abstraction - RDD) it has the following additional traits:

- **In-Memory**, i.e. data inside RDD is stored in memory as much (size) and long (time) as possible.
- **Immutable** or **Read-Only**, i.e. it does not change once created and can only be transformed using transformations to new RDDs.
- **Lazy evaluated**, i.e. the data inside RDD is not available or transformed until an action is executed that triggers the execution.
- **Cacheable**, i.e. you can hold all the data in a persistent "storage" like memory (default and the most preferred) or disk (the least preferred due to access speed).
- **Parallel**, i.e. process data in parallel.
- **Typed** — RDD records have types, e.g. `Long` in `RDD[Long]` or `(Int, String)` in `RDD[(Int, String)]`.
- **Partitioned** — records are partitioned (split into logical partitions) and distributed across nodes in a cluster.
- **Location-Stickiness** — `RDD` can define placement preferences to compute partitions (as close to the records as possible).

Note	**Preferred location** (aka *locality preferences* or *placement preferences* or *locality info*) is information about the locations of RDD records (that Spark's DAGScheduler uses to place computing partitions on to have the tasks as close to the data as possible).

Computing partitions in a RDD is a distributed process by design and to achieve even **data distribution** as well as leverage data locality (in distributed systems like HDFS or Cassandra in which data is partitioned by default), they are **partitioned** to a fixed number of partitions - logical chunks (parts) of data. The logical division is for processing only and internally it is not divided whatsoever. Each partition comprises of **records**.



Partitions are the units of parallelism. You can control the number of partitions of a RDD using repartition or coalesce transformations. Spark tries to be as close to data as possible without wasting time to send data across network by means of RDD shuffling, and creates as many partitions as required to follow the storage layout and thus optimize data access. It leads to a one-to-one mapping between (physical) data in distributed data storage, e.g. HDFS or Cassandra, and partitions.

RDDs support two kinds of operations:

- transformations - lazy operations that return another RDD.
- actions - operations that trigger computation and return values.

RDD Motivations – Iterative Algorithms & Interactive Query Tools

The motivation to create RDD were (after the authors) two types of applications that current computing frameworks handle inefficiently:

- **iterative algorithms** in machine learning and graph computations.
- **interactive data mining tools** as ad-hoc queries on the same dataset.

The goal is to reuse intermediate in-memory results across multiple data-intensive workloads with no need for copying large amounts of data over the network.

Technically, RDDs follow the contract defined by the five main intrinsic properties:

- List of parent RDDs that are the dependencies of the RDD.
- An array of partitions that a dataset is divided to.
- A compute function to do a computation on partitions.

- An optional Partitioner that defines how keys are hashed, and the pairs partitioned (for key-value RDDs)
- Optional preferred locations (aka **locality info**), i.e. hosts for a partition where the records live or are the closest to read from.

This RDD abstraction supports an expressive set of operations without having to modify scheduler for each one.

An RDD is a named (by `name`) and uniquely identified (by `id`) entity in a SparkContext (available as `context` property).

RDDs live in one and only one SparkContext that creates a logical boundary.

Note	RDDs cannot be shared between `SparkContexts` (see SparkContext and RDDs).

RDDs are a container of instructions on how to materialize big (arrays of) distributed data, and how to split it into partitions so Spark (using executors) can hold some of them.

In general data distribution can help executing processing in parallel so a task processes a chunk of data that it could eventually keep in memory.

Spark does jobs in parallel, and RDDs are split into partitions to be processed and written in parallel. Inside a partition, data is processed sequentially.

> Saving partitions results in part-files instead of one single file (unless there is a single partition).

persist and persist Internal Methods

`persist` Methods

```
persist(): this.type
persist(newLevel: StorageLevel): this.type
```

Refer to Persisting RDD — `persist` Methods.

`persist` Internal Method

```
persist(newLevel: StorageLevel, allowOverride: Boolean): this.type
```

Caution	FIXME

Note	`persist` is used when `RDD` is requested to persist itself and marks itself for local checkpointing.

RDD Contract

RDD Contract

```
abstract class RDD[T] {
  def compute(split: Partition, context: TaskContext): Iterator[T]
  def getPartitions: Array[Partition]
  def getDependencies: Seq[Dependency[_]]
  def getPreferredLocations(split: Partition): Seq[String] = Nil
  val partitioner: Option[Partition] = None
}
```

Note	`RDD` is an abstract class in Scala.

Table 1. RDD Contract

Method	Description
compute	Used exclusively when `RDD` computes a partition (possibly by reading from a checkpoint).
`getPartitions`	Used exclusively when `RDD` is requested for its partitions (called only once as the value is cached).
`getDependencies`	Used when `RDD` is requested for its dependencies (called only once as the value is cached).
`getPreferredLocations`	Defines **placement preferences** of a partition. Used exclusively when `RDD` is requested for the preferred locations of a partition.
`partitioner`	Defines the Partitioner of a `RDD`.

Types of RDDs

There are some of the most interesting types of RDDs:

- ParallelCollectionRDD
- CoGroupedRDD
- HadoopRDD is an RDD that provides core functionality for reading data stored in HDFS using the older MapReduce API. The most notable use case is the return RDD of `SparkContext.textFile`.
- **MapPartitionsRDD** - a result of calling operations like `map`, `flatMap`, `filter`, mapPartitions, etc.
- **CoalescedRDD** - a result of repartition or coalesce transformations.
- ShuffledRDD - a result of shuffling, e.g. after repartition or coalesce transformations.
- **PipedRDD** - an RDD created by piping elements to a forked external process.
- **PairRDD** (implicit conversion by PairRDDFunctions) that is an RDD of key-value pairs that is a result of `groupByKey` and `join` operations.
- **DoubleRDD** (implicit conversion as `org.apache.spark.rdd.DoubleRDDFunctions`) that is an RDD of `Double` type.
- **SequenceFileRDD** (implicit conversion as `org.apache.spark.rdd.SequenceFileRDDFunctions`) that is an RDD that can be saved as a `SequenceFile`.

Appropriate operations of a given RDD type are automatically available on a RDD of the right type, e.g. `RDD[(Int, Int)]`, through implicit conversion in Scala.

Transformations

A **transformation** is a lazy operation on a RDD that returns another RDD, like `map`, `flatMap`, `filter`, `reduceByKey`, `join`, `cogroup`, etc.

Actions

An **action** is an operation that triggers execution of RDD transformations and returns a value (to a Spark driver - the user program).

Creating RDDs

SparkContext.parallelize

SparkContext.parallelize

One way to create a RDD is with `SparkContext.parallelize` method. It accepts a collection of elements as shown below (`sc` is a SparkContext instance):

```
scala> val rdd = sc.parallelize(1 to 1000)
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <conso
le>:25
```

Given the reason to use Spark to process more data than your own laptop could handle, `SparkContext.parallelize` is mainly used to learn Spark in the Spark shell. `SparkContext.parallelize` requires all the data to be available on a single machine - the Spark driver - that eventually hits the limits of your laptop.

SparkContext.makeRDD

```
scala> sc.makeRDD(0 to 1000)
res0: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[1] at makeRDD at <console>
:25
```

SparkContext.textFile

One of the easiest ways to create an RDD is to use `SparkContext.textFile` to read files.

```
scala> val words = sc.textFile("README.md").flatMap(_.split("\\W+")).cache
words: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[27] at flatMap at <console>
:24
```

Note	You cache it so the computation is not performed every time you work with `words` .

RDD Caching and Persistence

Caching or **persistence** are optimisation techniques for (iterative and interactive) Spark computations. They help saving interim partial results so they can be reused in subsequent stages. These interim results as RDDs are thus kept in memory (default) or more solid storages like disk and/or replicated.

RDDs can be **cached** using cache operation. They can also be **persisted** using persist operation.

The difference between `cache` and `persist` operations is purely syntactic. `cache` is a synonym of `persist` or `persist(MEMORY_ONLY)`, i.e. `cache` is merely `persist` with the default storage level `MEMORY_ONLY`.

Note	Due to the very small and purely syntactic difference between caching and persistence of RDDs the two terms are often used interchangeably and I will follow the "pattern" here.

RDDs can also be unpersisted to remove RDD from a permanent storage like memory and/or disk.

Caching RDD — `cache` Method

```
cache(): this.type = persist()
```

`cache` is a synonym of persist with `MEMORY_ONLY` storage level.

Persisting RDD — `persist` Methods

```
persist(): this.type
persist(newLevel: StorageLevel): this.type
```

`persist` marks a RDD for persistence using `newLevel` storage level.

You can only change the storage level once or `persist` reports an `UnsupportedOperationException`:

```
Cannot change storage level of an RDD after it was already assigned a level
```

Note	You can *pretend* to change the storage level of an RDD with already-assigned storage level only if the storage level is the same as it is currently assigned.

If the RDD is marked as persistent the first time, the RDD is registered to `ContextCleaner` (if available) and `SparkContext`.

The internal `storageLevel` attribute is set to the input `newLevel` storage level.

StorageLevel

StorageLevel describes how an RDD is persisted (and addresses the following concerns):

- Does RDD use disk?
- How much of RDD is in memory?
- Does RDD use off-heap memory?
- Should an RDD be serialized (while persisting)?
- How many replicas (default: `1`) to use (can only be less than `40`)?

StorageLevel types (number _2 in the name denotes 2 replicas):

NONE (default)

DISK_ONLY

DISK_ONLY_2

MEMORY_ONLY (default for cache operation for RDDs)

MEMORY_ONLY_2

MEMORY_ONLY_SER

MEMORY_ONLY_SER_2

MEMORY_AND_DISK

MEMORY_AND_DISK_2

MEMORY_AND_DISK_SER

MEMORY_AND_DISK_SER_2

OFF_HEAP

You can check out the storage level using `getStorageLevel()` operation.

```
val lines = sc.textFile("README.md")

scala> lines.getStorageLevel
res0: org.apache.spark.storage.StorageLevel = StorageLevel(disk=false, memory=false, o
ffheap=false, deserialized=false, replication=1)
```

Transformations

Transformations are lazy operations on a RDD that create one or many new RDDs, e.g. `map`, `filter`, `reduceByKey`, `join`, `cogroup`, `randomSplit`.

```
transformation: RDD => RDD
transformation: RDD => Seq[RDD]
```

In other words, transformations are *functions* that take a RDD as the input and produce one or many RDDs as the output. They do not change the input RDD (since RDDs are immutable and hence cannot be modified), but always produce one or more new RDDs by applying the computations they represent.

By applying transformations you incrementally build a RDD lineage with all the parent RDDs of the final RDD(s).

Transformations are lazy, i.e. are not executed immediately. Only after calling an action are transformations executed.

After executing a transformation, the result RDD(s) will always be different from their parents and can be smaller (e.g. `filter`, `count`, `distinct`, `sample`), bigger (e.g. `flatMap`, `union`, `cartesian`) or the same size (e.g. `map`).

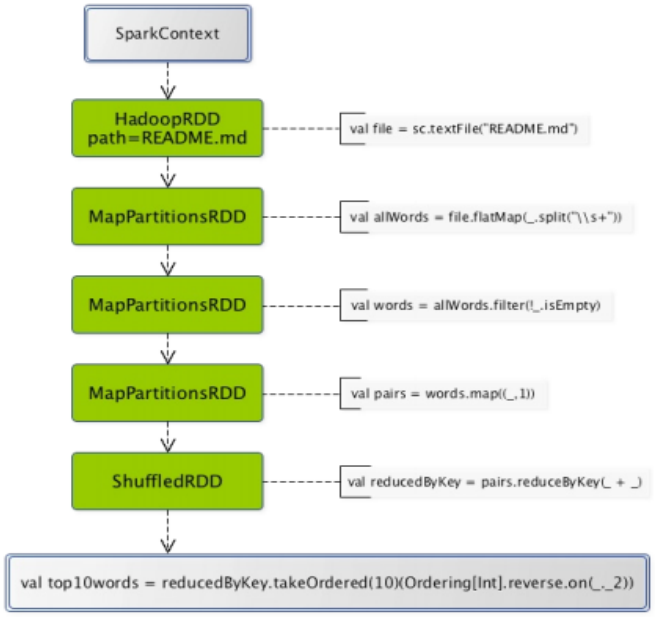


Figure 1. From SparkContext by transformations to the result

Certain transformations can be **pipelined** which is an optimization that Spark uses to improve performance of computations.

```
scala> val file = sc.textFile("README.md")
file: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[54] at textFile at <console>:
24

scala> val allWords = file.flatMap(_.split("\\W+"))
allWords: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[55] at flatMap at <conso
le>:26

scala> val words = allWords.filter(!_.isEmpty)
words: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[56] at filter at <console>:
28

scala> val pairs = words.map((_,1))
pairs: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[57] at map at <conso
le>:30

scala> val reducedByKey = pairs.reduceByKey(_ + _)
reducedByKey: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[59] at reduceByKey
 at <console>:32

scala> val top10words = reducedByKey.takeOrdered(10)(Ordering[Int].reverse.on(_._2))
INFO SparkContext: Starting job: takeOrdered at <console>:34
...
INFO DAGScheduler: Job 18 finished: takeOrdered at <console>:34, took 0.074386 s
top10words: Array[(String, Int)] = Array((the,21), (to,14), (Spark,13), (for,11), (and,
10), (##,8), (a,8), (run,7), (can,6), (is,8))
```

Narrow and Wide Transformations on RDDs

Narrow Transformations

Narrow transformations are the result of `map`, `filter` and such that is from the data from a single partition only, i.e. it is self-sustained.

An output RDD has partitions with records that originate from a single partition in the parent RDD. Only a limited subset of partitions used to calculate the result.

Spark groups narrow transformations as a stage which is called **pipelining**.

Wide Transformations

Wide transformations are the result of `groupByKey` and `reduceByKey`. The data required to compute the records in a single partition may reside in many partitions of the parent RDD.

Note	Wide transformations are also called shuffle transformations as they may or may not depend on a shuffle.

All of the tuples with the same key must end up in the same partition, processed by the same task. To satisfy these operations, Spark must execute RDD shuffle, which transfers data across cluster and results in a new stage with a new set of partitions.

Actions

Actions are RDD operations that produce non-RDD values. They materialize a value in a Spark program. In other words, a RDD operation that returns a value of any type but `RDD[T]` is an action.

```
action: RDD => a value
```

Note	Actions are synchronous. You can use AsyncRDDActions to release a calling thread while calling actions.

They trigger execution of RDD transformations to return values. Simply put, an action evaluates the RDD lineage graph.

You can think of actions as a valve and until action is fired, the data to be processed is not even in the pipes, i.e. transformations. Only actions can materialize the entire processing pipeline with real data.

Actions are one of two ways to send data from executors to the driver (the other being accumulators).

Actions in org.apache.spark.rdd.RDD:

- aggregate
- collect
- count
- countApprox*
- countByValue*
- first
- fold
- foreach
- foreachPartition
- max
- min
- reduce

saveAs* actions, e.g. saveAsTextFile, saveAsHadoopFile

take

takeOrdered

takeSample

toLocalIterator

top

treeAggregate

treeReduce

Actions run jobs using SparkContext.runJob or directly DAGScheduler.runJob.

```
scala> words.count  (1)
res0: Long = 502
```

1. `words` is an RDD of `String` .

Tip	You should cache RDDs you work with when you want to execute two or more actions on it for a better performance. Refer to RDD Caching and Persistence.

Before calling an action, Spark does closure/function cleaning (using `SparkContext.clean`) to make it ready for serialization and sending over the wire to executors. Cleaning can throw a `SparkException` if the computation cannot be cleaned.

Note	Spark uses `ClosureCleaner` to clean closures.

Broadcast Variables

Use Broadcast Variables for efficient data distribution, Spark itself uses them quite often.

A very notable use case is when Spark distributes tasks to executors for their execution.

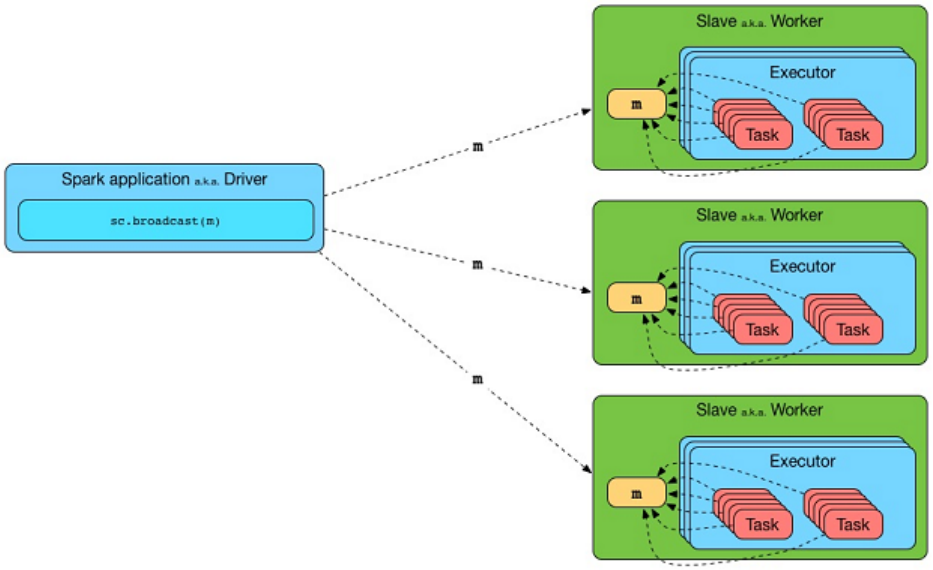


Figure 4. Broadcasting a value to executors

To use a broadcast value in a Spark transformation you have to create it first using SparkContext.broadcast and then use `value` method to access the shared value.

The Broadcast feature in Spark uses SparkContext to create broadcast values and BroadcastManager and ContextCleaner to manage their lifecycle.

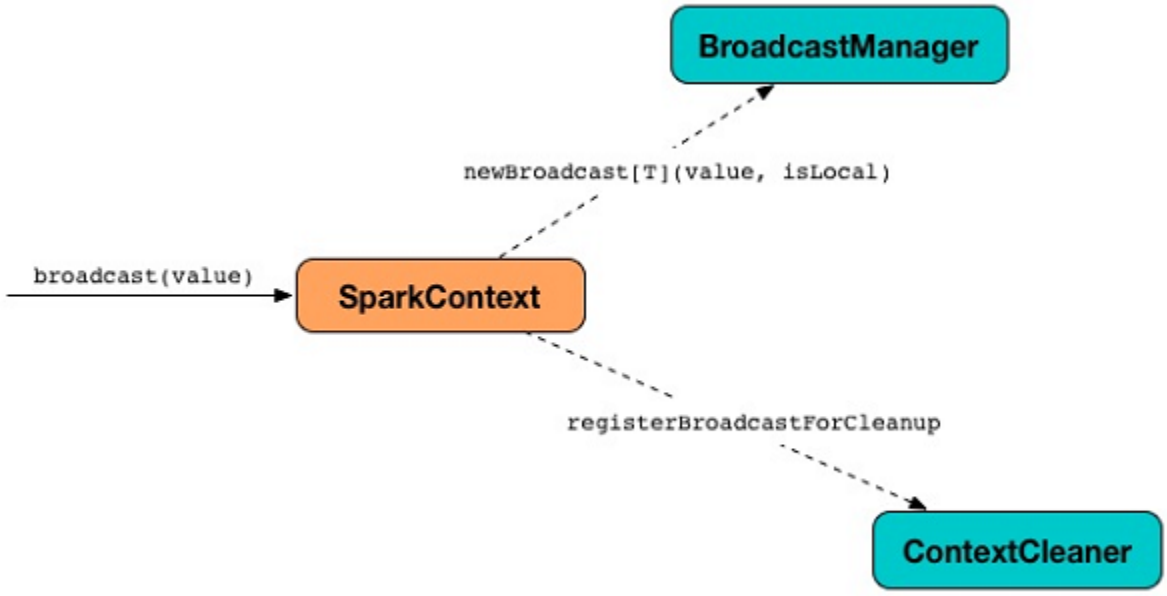


Figure 2. SparkContext to broadcast using BroadcastManager and ContextCleaner

Broadcast Spark Developer-Facing Contract

`Broadcast` Spark Developer-Facing Contract

The developer-facing `Broadcast` contract allows Spark developers to use it in their applications.

Table 1. Broadcast API

Method Name	Description
`id`	The unique identifier
value	The value
unpersist	Asynchronously deletes cached copies of this broadcast on the executors.
destroy	Destroys all data and metadata related to this broadcast variable.
`toString`	The string representation

Lifecycle of a Broadcast Variable

You can create a broadcast variable of type `T` using SparkContext.broadcast method.

```
scala> val b = sc.broadcast(1)
b: org.apache.spark.broadcast.Broadcast[Int] = Broadcast(0)
```

Tip	Enable `DEBUG` logging level for `org.apache.spark.storage.BlockManager` logger to debug `broadcast` method. Read BlockManager to find out how to enable the logging level.

After creating an instance of a broadcast variable, you can then reference the value using value method.

```
scala> b.value
res0: Int = 1
```

Note	`value` method is the only way to access the value of a broadcast variable.

When you are done with a broadcast variable, you should destroy it to release memory.

```
scala> b.destroy
```

value method

```
value: T
```

`value` returns the value of a broadcast variable. You can only access the value until it is destroyed after which you will see the following `SparkException` exception in the logs:

```
org.apache.spark.SparkException: Attempted to use Broadcast(0) after it was destroyed
(destroy at <console>:27)
  at org.apache.spark.broadcast.Broadcast.assertValid(Broadcast.scala:144)
  at org.apache.spark.broadcast.Broadcast.value(Broadcast.scala:69)
  ... 48 elided
```

Internally, `value` makes sure that the broadcast variable is **valid**, i.e. destroy was not called, and, if so, calls the abstract `getValue` method.

Note	`getValue` is abstracted and broadcast variable implementations are supposed to provide a concrete behaviour. Refer to TorrentBroadcast.

unpersist & destroy methods

Unpersisting Broadcast Variable — `unpersist` Methods

```
unpersist(): Unit
unpersist(blocking: Boolean): Unit
```

Destroying Broadcast Variable — `destroy` Method

```
destroy(): Unit
```

`destroy` removes a broadcast variable.

Note	Once a broadcast variable has been destroyed, it cannot be used again.

Broadcast is part of Spark that is responsible for broadcasting information across nodes in a cluster.

You use broadcast variable to implement **map-side join**, i.e. a join using a `map`. For this, lookup tables are distributed across nodes in a cluster using `broadcast` and then looked up inside `map` (to do the join implicitly).

When you broadcast a value, it is copied to executors only once (while it is copied multiple times for tasks otherwise). It means that broadcast can help to get your Spark application faster if you have a large value to use in tasks or there are more tasks than executors.

It appears that a Spark idiom emerges that uses `broadcast` with `collectAsMap` to create a `Map` for broadcast. When an RDD is `map` over to a smaller dataset (column-wise not record-wise), `collectAsMap`, and `broadcast`, using the very big RDD to map its elements to the broadcast RDDs is computationally faster.

Accumulators

Accumulators are variables that are "added" to through an associative and commutative "add" operation. They act as a container for accumulating partial values across multiple tasks (running on executors). They are designed to be used safely and efficiently in parallel and distributed Spark computations and are meant for distributed counters and sums (e.g. task metrics).

You can create built-in accumulators for longs, doubles, or collections or register custom accumulators using the SparkContext.register methods. You can create accumulators with or without a name, but only named accumulators are displayed in web UI (under Stages tab for a given stage).

Accumulator are write-only variables for executors. They can be added to by executors and read by the driver only.

```
executor1: accumulator.add(incByExecutor1)
executor2: accumulator.add(incByExecutor2)

driver:  println(accumulator.value)
```

Accumulators are serializable so they can safely be referenced in the code executed in executors and then safely send over the wire for execution.

```
val counter = sc.longAccumulator("counter")
sc.parallelize(1 to 9).foreach(x => counter.add(x))
```

Internally, longAccumulator, doubleAccumulator, and collectionAccumulator methods create the built-in typed accumulators and call SparkContext.register.

AccumulatorV2

AccumulatorV2

```
abstract class AccumulatorV2[IN, OUT]
```

`AccumulatorV2` parameterized class represents an accumulator that accumulates `IN` values to produce `OUT` result.

register method

Registering Accumulator — `register` Method

```
register(
  sc: SparkContext,
  name: Option[String] = None,
  countFailedValues: Boolean = false): Unit
```

`register` creates a AccumulatorMetadata metadata object for the accumulator (with a new unique identifier) that is then used to register the accumulator with.

In the end, `register` registers the accumulator for cleanup (only when `ContextCleaner` is defined in the `SparkContext`).

`register` reports a `IllegalStateException` if metadata is already defined (which means that `register` was called already).

```
Cannot register an Accumulator twice.
```

Note	`register` is a `private[spark]` method.

Note	`register` is used when: • `SparkContext` registers accumulators • `TaskMetrics` registers the internal accumulators • SQLMetrics creates metrics.

AccumulatorMetadata

AccumulatorMetadata

`AccumulatorMetadata` is a container object with the metadata of an accumulator:

- Accumulator ID
- (optional) name
- Flag whether to include the latest value of an accumulator on failure

Note	`countFailedValues` is used exclusively when `Task` collects the latest values of accumulators (irrespective of task status — a success or a failure).

AccumulableInfo

- optional partial `update` to the accumulator from a task
- `value`
- whether or not it is `internal`
- whether or not to `countFailedValues` to the final value of the accumulator for failed tasks
- optional `metadata`

`AccumulableInfo` is used to transfer accumulator updates from executors to the driver every executor heartbeat or when a task finishes.

AccumulatorContext

AccumulatorContext

`AccumulatorContext` is a `private[spark]` internal object used to track accumulators by Spark itself using an internal `originals` lookup table. Spark uses the `AccumulatorContext` object to register and unregister accumulators.

The `originals` lookup table maps accumulator identifier to the accumulator itself.

Every accumulator has its own unique accumulator id that is assigned using the internal `nextId` counter.

AccumulatorContext.SQL_ACCUM_IDENTIFIER

AccumulatorContext.SQL_ACCUM_IDENTIFIER

`AccumulatorContext.SQL_ACCUM_IDENTIFIER` is an internal identifier for Spark SQL's internal accumulators. The value is `sql` and Spark uses it to distinguish Spark SQL metrics from others.

DAGScheduler – Stage-Oriented Scheduler

DAGScheduler is the scheduling layer of Apache Spark that implements **stage-oriented scheduling**. It transforms a **logical execution plan** (i.e. RDD lineage of dependencies built using RDD transformations) to a **physical execution plan** (using stages).

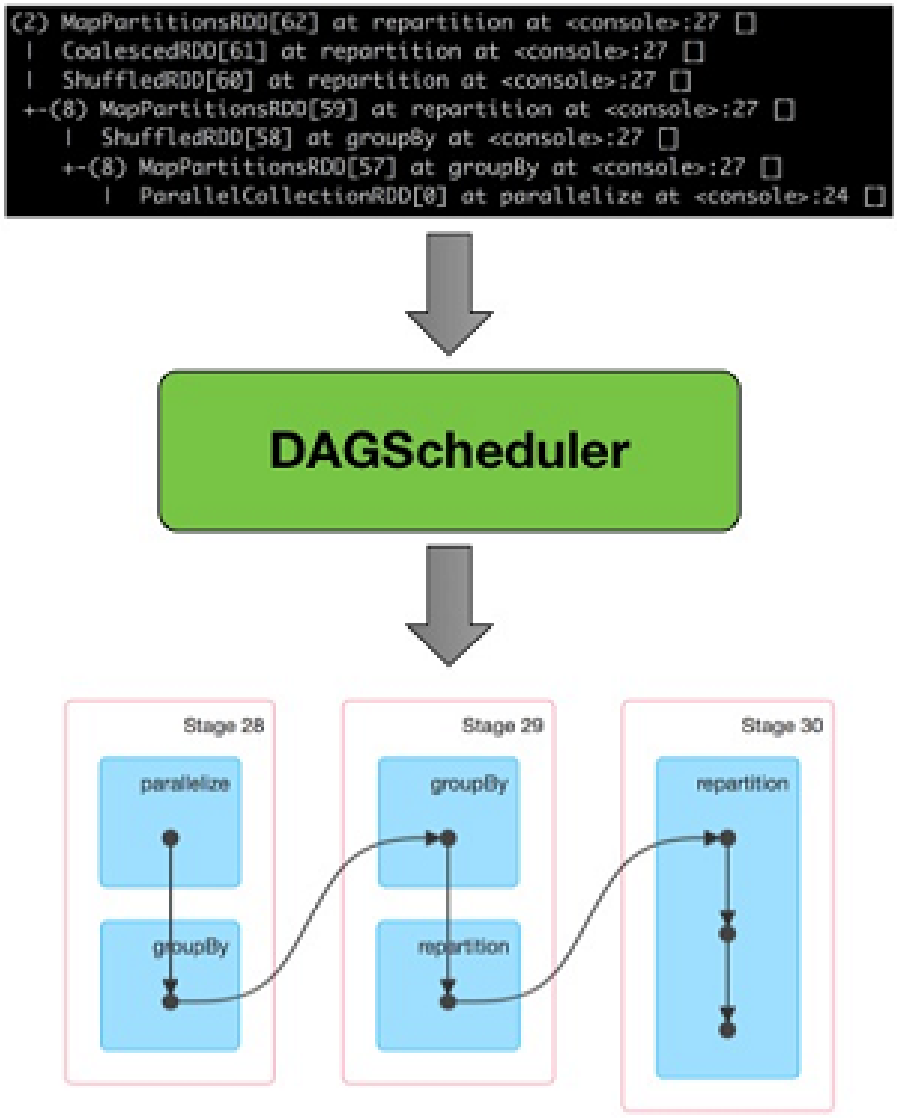


Figure 1. `DAGScheduler` Transforming RDD Lineage Into Stage DAG

The fundamental concepts of `DAGScheduler` are **jobs** and **stages** (refer to Jobs and Stages respectively) that it tracks through internal registries and counters.

DAGScheduler works solely on the driver and is created as part of SparkContext's initialization (right after TaskScheduler and SchedulerBackend are ready).

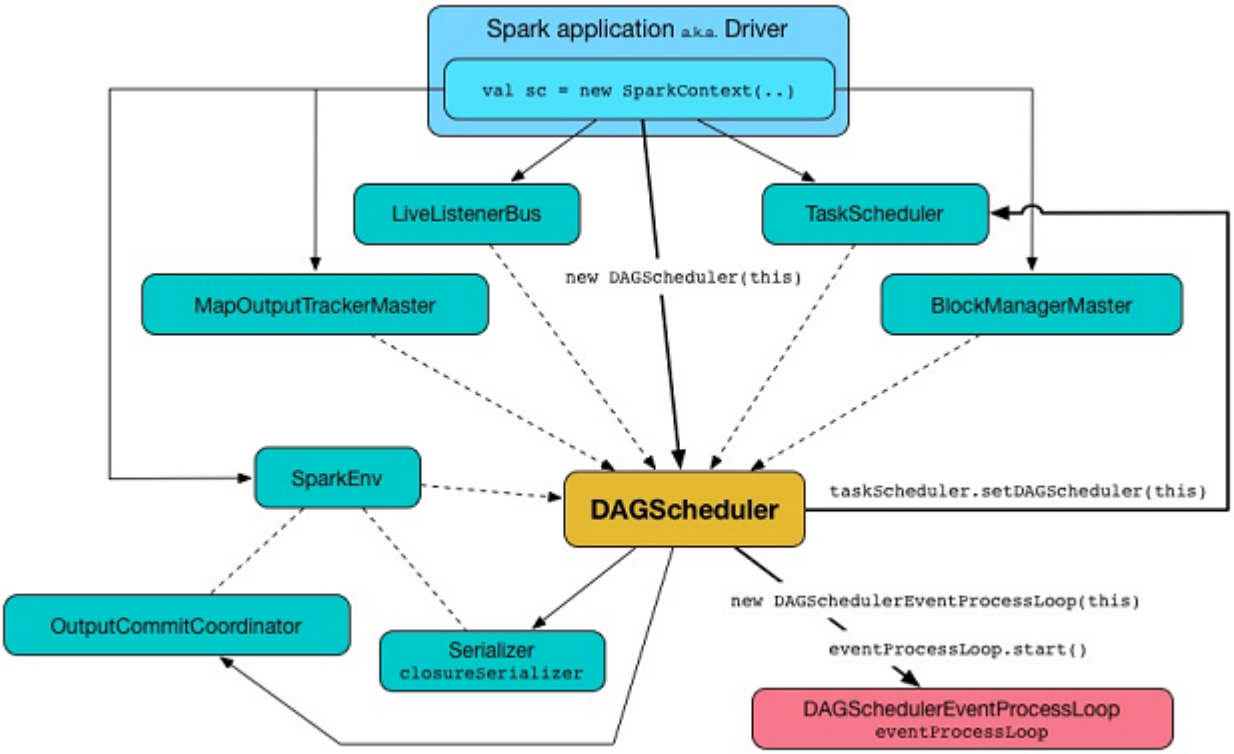


Figure 3. DAGScheduler as created by SparkContext with other services

DAGScheduler does three things in Spark (thorough explanations follow):

- Computes an **execution DAG**, i.e. DAG of stages, for a job.
- Determines the preferred locations to run each task on.
- Handles failures due to **shuffle output files** being lost.

`DAGScheduler` computes a directed acyclic graph (DAG) of stages for each job, keeps track of which RDDs and stage outputs are materialized, and finds a minimal schedule to run jobs. It then submits stages to TaskScheduler.

In addition to coming up with the execution DAG, DAGScheduler also determines the preferred locations to run each task on, based on the current cache status, and passes the information to TaskScheduler.

`DAGScheduler` tracks which RDDs are cached (or persisted) to avoid "recomputing" them, i.e. redoing the map side of a shuffle. `DAGScheduler` remembers what ShuffleMapStages have already produced output files (that are stored in BlockManagers).

DAGScheduler is only interested in cache location coordinates, i.e. host and executor id, per partition of a RDD.

Furthermore, it handles failures due to shuffle output files being lost, in which case old stages may need to be resubmitted. Failures within a stage that are not caused by shuffle file loss are handled by the TaskScheduler itself, which will retry each task a small number of times before cancelling the whole stage.

DAGScheduler uses an **event queue architecture** in which a thread can post `DAGSchedulerEvent` events, e.g. a new job or stage being submitted, that DAGScheduler reads and executes sequentially. See the section Internal Event Loop - dag-scheduler-event-loop.

DAGScheduler runs stages in topological order.

DAGScheduler uses SparkContext, TaskScheduler, LiveListenerBus, MapOutputTracker and BlockManager for its services. However, at the very minimum, `DAGScheduler` takes a `SparkContext` only (and requests `SparkContext` for the other services).

DAGScheduler reports metrics about its execution (refer to the section Metrics).

Creating a *DAGScheduler* Instance

`DAGScheduler` takes the following when created:

- SparkContext
- TaskScheduler
- LiveListenerBus
- MapOutputTrackerMaster
- BlockManagerMaster
- SparkEnv
- `clock` (defaults to `SystemClock`)

`DAGScheduler` initializes the internal registries and counters.

`DAGScheduler` sets itself in the given `TaskScheduler` and in the end starts DAGScheduler Event Bus.

Note	`DAGScheduler` can reference all the services through a single SparkContext with or without specifying explicit TaskScheduler.

liveListenerBus method

executorHeartbeatReceived method

`executorHeartbeatReceived` Method

```
executorHeartbeatReceived(
  execId: String,
  accumUpdates: Array[(Long, Int, Int, Seq[AccumulableInfo])],
  blockManagerId: BlockManagerId): Boolean
```

`executorHeartbeatReceived` posts a SparkListenerExecutorMetricsUpdate (to listenerBus) and informs BlockManagerMaster that `blockManagerId` block manager is alive (by posting BlockManagerHeartbeat).

Note	`executorHeartbeatReceived` is called when `TaskSchedulerImpl` handles `executorHeartbeatReceived`.

submitJob method Synchronous

Submitting Job — `submitJob` method

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

`submitJob` creates a JobWaiter and posts a `JobSubmitted` event.

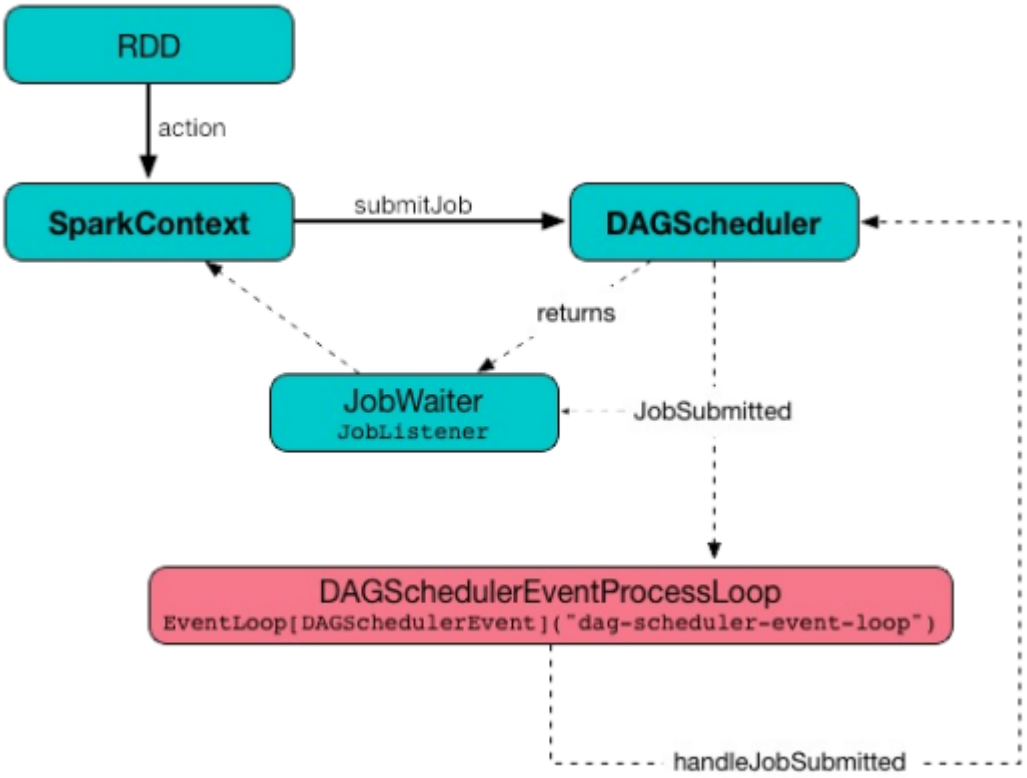


Figure 4. DAGScheduler.submitJob

Internally, `submitJob` does the following:

1. Checks whether `partitions` reference available partitions of the input `rdd`.
2. Increments nextJobId internal job counter.
3. Returns a 0-task JobWaiter when the number of `partitions` is zero.
4. Posts a `JobSubmitted` event and returns a `JobWaiter`.

You may see a `IllegalArgumentException` thrown when the input `partitions` references partitions not in the input `rdd`:

```
Attempting to access a non-existent partition: [p]. Total number of partitions: [maxPa
rtitions]
```

Note	`submitJob` is called when `SparkContext` submits a job and `DAGScheduler` runs a job.

Note	`submitJob` assumes that the partitions of a RDD are indexed from 0 onwards in sequential order.

runJob method Asynchronous

runJob is used when SparkContext runs a job

Running Job — `runJob` Method

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

`runJob` submits an action job to the `DAGScheduler` and waits for a result.

Internally, `runJob` executes submitJob and then waits until a result comes using JobWaiter.

getOrCreateShuffleMapStage method

```
getOrCreateShuffleMapStage(
  shuffleDep: ShuffleDependency[_, _, _],
  firstJobId: Int): ShuffleMapStage
```

`getOrCreateShuffleMapStage` finds or creates the ShuffleMapStage for the input ShuffleDependency.

Internally, `getOrCreateShuffleMapStage` finds the `ShuffleDependency` in `shuffleIdToMapStage` internal registry and returns one when found.

If no `ShuffleDependency` was available, `getOrCreateShuffleMapStage` finds all the missing shuffle dependencies and creates corresponding `ShuffleMapStage` stages (including one for the input `shuffleDep`).

Note	All the new `ShuffleMapStage` stages are associated with the input `firstJobId`.
Note	`getOrCreateShuffleMapStage` is used when `DAGScheduler` finds or creates missing direct parent ShuffleMapStages (for ShuffleDependencies of given RDD), getMissingParentStages (for ShuffleDependencies), is notified that `ShuffleDependency` was submitted, and checks if a stage depends on another.

createShuffleMapStage method

Creating ShuffleMapStage for ShuffleDependency (Copying Shuffle Map Output Locations From Previous Jobs) — `createShuffleMapStage` Method

```
createShuffleMapStage(
  shuffleDep: ShuffleDependency[_, _, _],
  jobId: Int): ShuffleMapStage
```

`createShuffleMapStage` creates a ShuffleMapStage for the input ShuffleDependency and `jobId` (of a ActiveJob) possibly copying shuffle map output locations from previous jobs to avoid recomputing records.

Note	When a ShuffleMapStage is created, the `id` is generated (using `nextStageId` internal counter), `rdd` is from `ShuffleDependency`, `numTasks` is the number of partitions in the RDD, all `parents` are looked up (and possibly created), the `jobId` is given, `callSite` is the `creationSite` of the RDD, and `shuffleDep` is the input `ShuffleDependency`.

Internally, `createShuffleMapStage` first finds or creates missing parent `ShuffleMapStage` stages of the associated RDD.

Note	ShuffleDependency is associated with exactly one `RDD[Product2[K, V]]`.

`createShuffleMapStage` creates a `ShuffleMapStage` (with the stage id from `nextStageId` internal counter).

Note	The RDD of the new `ShuffleMapStage` is from the input ShuffleDependency.

`createShuffleMapStage` registers the `ShuffleMapStage` in stageIdToStage and shuffleIdToMapStage internal registries.

`createShuffleMapStage` calls updateJobIdStageIdMaps.

If `MapOutputTrackerMaster` tracks the input `ShuffleDependency` (because other jobs have already computed it), `createShuffleMapStage` requests the serialized `ShuffleMapStage` outputs, deserializes them and registers with the new `ShuffleMapStage`.

Note	MapOutputTrackerMaster was defined when `DAGScheduler` was created.

If `MapOutputTrackerMaster` tracks the input `ShuffleDependency` (because other jobs have already computed it), `createShuffleMapStage` requests the serialized `ShuffleMapStage` outputs, deserializes them and registers with the new `ShuffleMapStage`.

Note	MapOutputTrackerMaster was defined when `DAGScheduler` was created.

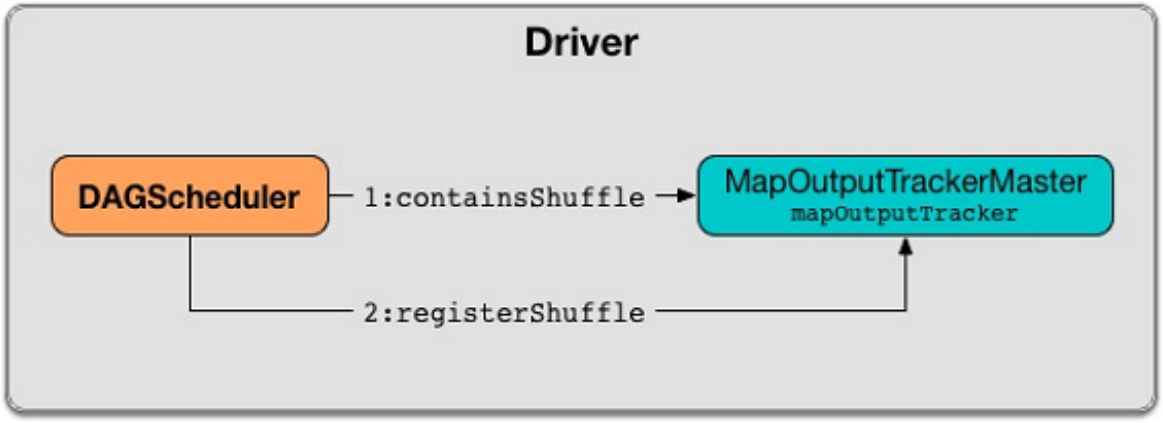


Figure 5. `DAGScheduler` Asks `MapOutputTrackerMaster` Whether Shuffle Map Output Is Already Tracked

`createShuffleMapStage` returns the new `ShuffleMapStage`.

Note	`createShuffleMapStage` is executed only when `DAGScheduler` finds or creates parent `ShuffleMapStage` stages for a `ShuffleDependency`.

getShuffleDepenendancies method

```
getShuffleDependencies(rdd: RDD[_]): HashSet[ShuffleDependency[_, _, _]]
```

`getShuffleDependencies` finds direct parent shuffle dependencies for the given RDD.

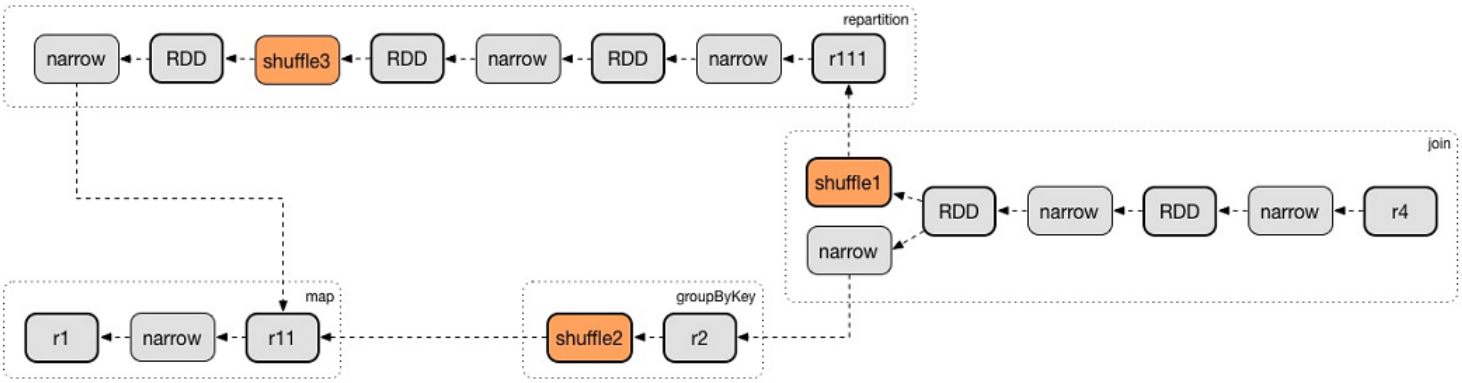


Figure 6. getShuffleDependencies Finds Direct Parent ShuffleDependencies (shuffle1 and shuffle2)

Internally, `getShuffleDependencies` takes the direct shuffle dependencies of the input RDD and direct shuffle dependencies of all the parent non-`ShuffleDependencies` in the dependency chain (aka *RDD lineage*).

Note	`getShuffleDependencies` is used when `DAGScheduler` finds or creates missing direct parent ShuffleMapStages (for ShuffleDependencies of given RDD) and finds all missing shuffle dependencies for a given RDD.