# Spark 技术分享

王社英

北京回龙观

April 17, 2020

# Outline

# Outline

# Spark Introduction

## Spark Introduction

Apache Spark is a unified analytics engine for large-scale data processing.

You can run Spark using its standalone cluster mode, on EC2, on Hadoop YARN, on Mesos, or on Kubernetes.

Access data in HDFS, Alluxio, Apache Cassandra, Apache HBase, Apache Hive, and hundreds of other data sources.

# Spark Introduction

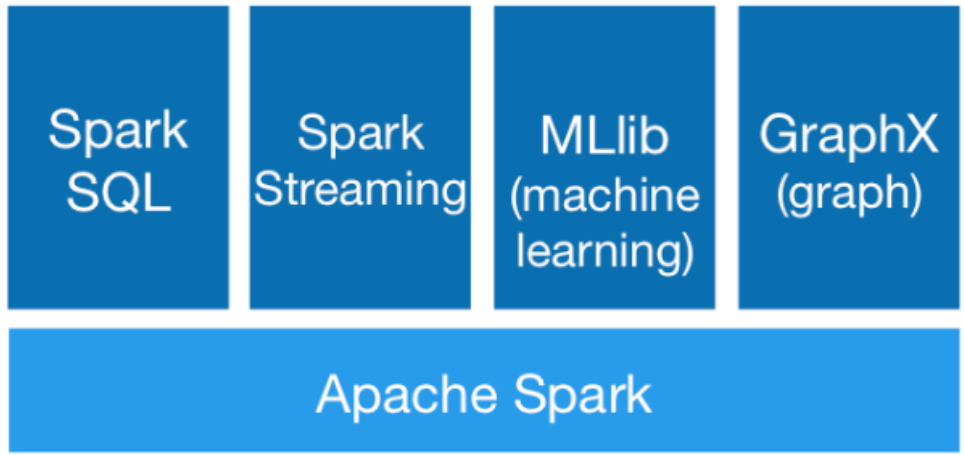
## Spark Introduction






# Spark Introduction

## Spark Introduction



# Outline

# Spark Initialization

## Spark Submit

```
$ ./bin/spark-submit --class org.apache.spark.examples.SparkPi \
    --master yarn \
    --deploy-mode cluster \
    --driver-memory 4g \
    --executor-memory 2g \
    --executor-cores 1 \
    --queue thequeue \
    examples/jars/spark-examples*.jar \
    10
```

```
$ ./bin/spark-shell --master yarn --deploy-mode client
```

# ApplicationMaster

## ApplicationMaster Initialization

```scala
object ApplicationMaster extends Logging {

  // exit codes for different causes, no reason behind the values
  private val EXIT_SUCCESS = 0
  private val EXIT_UNCAUGHT_EXCEPTION = 10
  private val EXIT_MAX_EXECUTOR_FAILURES = 11
  private val EXIT_REPORTER_FAILURE = 12
  private val EXIT_SC_NOT_INITED = 13
  private val EXIT_SECURITY = 14
  private val EXIT_EXCEPTION_USER_CLASS = 15
  private val EXIT_EARLY = 16

  private var master: ApplicationMaster = _

  def main(args: Array[String]): Unit = {
    SignalUtils.registerLogger(log)
    val amArgs = new ApplicationMasterArguments(args)
    master = new ApplicationMaster(amArgs)
    System.exit(master.run())
  }
```

Figure 1: ApplicationMaster#main

# ApplicationMaster

## ApplicationMaster Initialization

```
final def run(): Int = {
  doAsUser {
    runImpl()
  }
  exitCode
}
```

Figure 2: ApplicationMaster#run

# ApplicationMaster

## ApplicationMaster Initialization

```
private def runImpl(): Unit = {
  try {
    val appAttemptId = client.getAttemptId()

    var attemptID: Option[String] = None

    if (isClusterMode) {...}

    new CallerContext(...).setCurrentContext()

    logInfo( msg = "ApplicationAttemptId: " + appAttemptId)

    // This shutdown hook should run *after* the SparkContext is shut down.
    val priority = ShutdownHookManager.SPARK_CONTEXT_SHUTDOWN_PRIORITY - 1
    ShutdownHookManager.addShutdownHook(priority) (...)

    /.../
    if (sparkConf.contains(CREDENTIALS_FILE_PATH)) {...}

    if (isClusterMode) {
      runDriver()
    } else {
      runExecutorLauncher()
    }
  } catch {
    case e: Exception =>
      // catch everything else if not specifically handled
      ...
  }
}
```

Figure 3: ApplicationMaster#runImpl

# ApplicationMaster

## ApplicationMaster Initialization Cluster

```
private def runDriver(): Unit = {
  addAmIpFilter(None)
  userClassThread = startUserApplication()

  /.../
  logInfo( msg = "Waiting for spark context initialization...")
  val totalWaitTime = sparkConf.get(AM_MAX_WAIT_TIME)
  try {
    val sc = ThreadUtils.awaitResult(...)
    if (sc != null) {
      rpcEnv = sc.env.rpcEnv
      val driverRef = createSchedulerRef(
        sc.getConf.get("spark.driver.host"),
        sc.getConf.get("spark.driver.port"))
      registerAM(sc.getConf, rpcEnv, driverRef, sc.ui.map(_.webUrl))
      registered = true
    } else {...}
    resumeDriver()
    userClassThread.join()
  } catch {
    case e: SparkException if e.getCause().isInstanceOf[TimeoutException] =>
      logError(...)
      finish(...)
  } finally {
    resumeDriver()
  }
}
```

Figure 4: ApplicationMaster#runDriver

# ApplicationMaster

## ApplicationMaster Initialization Cluster

```scala
/**
 * Start the user class, which contains the spark driver, in a separate Thread.
 * If the main routine exits cleanly or exits with System.exit(N) for any N
 * we assume it was successful, for all other cases we assume failure.
 *
 * Returns the user thread that was started.
 */
private def startUserApplication(): Thread = {
  logInfo( msg = "Starting the user application in a separate Thread")

  var userArgs = args.userArgs
  if (args.primaryPyFile != null && args.primaryPyFile.endsWith(".py")) {...}
  if (args.primaryRFile != null && args.primaryRFile.endsWith(".R")) {...}

  val mainMethod = userClassLoader.loadClass(args.userClass)
    .getMethod( name = "main", classOf[Array[String]])

  val userThread = new Thread {
    override def run() {
      try {
        mainMethod.invoke( obj = null, userArgs.toArray)
        finish(FinalApplicationStatus.SUCCEEDED, ApplicationMaster.EXIT_SUCCESS)
        logDebug( msg = "Done running users class")
      } catch {
        case e: InvocationTargetException =>
          ...
      } finally {
        /.../
        sparkContextPromise.trySuccess(null)
      }
    }
  }
  userThread.setContextClassLoader(userClassLoader)
  userThread.setName("Driver")
  userThread.start()
  userThread
}
```

# ApplicationMaster

## ApplicationMaster Initialization Client

```
private def runExecutorLauncher(): Unit = {
  val hostname = Utils.localHostName
  val amCores = sparkConf.get(AM_CORES)
  rpcEnv = RpcEnv.create( name = "sparkYarnAM", hostname, hostname,  port = -1, sparkConf, securityMgr,
    amCores,  clientMode = true)
  val driverRef = waitForSparkDriver()
  addAmIpFilter(Some(driverRef))
  registerAM(sparkConf, rpcEnv, driverRef, sparkConf.getOption( key = "spark.driver.appUIAddress"))
  registered = true

  // In client mode the actor will stop the reporter thread.
  reporterThread.join()
}
```

Figure 6: ApplicationMaster#runExecutorLauncher

# ApplicationMaster

## ApplicationMaster Initialization Client

```
private def waitForSparkDriver(): RpcEndpointRef = {
  logInfo( msg = "Waiting for Spark driver to be reachable.")
  var driverUp = false
  val hostport = args.userArgs(0)
  val (driverHost, driverPort) = Utils.parseHostPort(hostport)

  // Spark driver should already be up since it launched us, but we don't want to
  // wait forever, so wait 100 seconds max to match the cluster mode setting.
  val totalWaitTimeMs = sparkConf.get(AM_MAX_WAIT_TIME)
  val deadline = System.currentTimeMillis + totalWaitTimeMs

  while (!driverUp && !finished && System.currentTimeMillis < deadline) {
    try {
      val socket = new Socket(driverHost, driverPort)
      socket.close()
      logInfo("Driver now available: %s:%s".format(driverHost, driverPort))
      driverUp = true
    } catch {
      case e: Exception =>
        ...
    }
  }

  if (!driverUp) {...}

  sparkConf.set("spark.driver.host", driverHost)
  sparkConf.set("spark.driver.port", driverPort.toString)
  createSchedulerRef(driverHost, driverPort.toString)
}
```

Figure 7: ApplicationMaster#waitForSparkDriver

# ApplicationMaster

## Request Resources

```
private def registerAM(
    _sparkConf: SparkConf,
    _rpcEnv: RpcEnv,
    driverRef: RpcEndpointRef,
    uiAddress: Option[String]) : Unit  = {
  val appId = client.getAttemptId().getApplicationId().toString()
  val attemptId = client.getAttemptId().getAttemptId().toString()
  val historyAddress = ApplicationMaster
    .getHistoryServerAddress(_sparkConf, yarnConf, appId, attemptId)

  val driverUrl = RpcEndpointAddress(...).toString

  // Before we initialize the allocator, let's log the information about how executors will
  // be run up front, to avoid printing this out for every single executor being launched.
  // Use placeholders for information that changes such as executor IDs.
  logInfo (...)

  allocator = client.register(...)

  /.../
  rpcEnv.setupEndpoint( name = "YarnAM", new AMEndpoint(rpcEnv, driverRef))

  allocator.allocateResources()
  reporterThread = launchReporterThread()
}
```

Figure 8: ApplicationMaster#registerAM

# ApplicationMaster

## Request Resources

```
19/09/16 17:18:08 INFO yarn.ApplicationMaster: Driver now available: hdp-nano01cn-p001.pek3.wecash.net:46539
19/09/16 17:18:08 INFO client.TransportClientFactory: Successfully created connection to hdp-nano01cn-p001.pek3.wecash.net/10.40.110.101:46539 after 54 ms ((
19/09/16 17:18:08 INFO yarn.ApplicationMaster:
===============================================================================
YARN executor launch context:
  env:
    CLASSPATH -> {{HADOOP_COMMON_HOME}}/../../../SPARK2-2.3.0.cloudera4-1.cdh5.13.3.p0.611179/lib/spark2/jars/spark-catalyst_2.11-2.3.0.cloudera4.jar:/usr/l
    SPARK_DIST_CLASSPATH -> /opt/cloudera/parcels/SPARK2-2.3.0.cloudera4-1.cdh5.13.3.p0.611179/lib/spark2/kafka-0.10/*:/opt/cloudera/parcels/CDH-5.12.2-1.cdh
    SPARK_YARN_STAGING_DIR -> hdfs://nameservice1/user/root/.sparkStaging/application_1561804288428_204915
    SPARK_USER -> root
    PYTHONHASHSEED -> 42
    PYTHONPATH -> /data/anaconda3/envs/py36/bin/python

  command:
    LD_LIBRARY_PATH="{{HADOOP_COMMON_HOME}}/../../../CDH-5.12.2-1.cdh5.12.2.p0.4/lib/hadoop/lib/native:$LD_LIBRARY_PATH" \
      {{JAVA_HOME}}/bin/java \
      -server \
      -Xmx7168m \
      -Djava.io.tmpdir={{PWD}}/tmp \
      '-Dspark.port.maxRetries=30' \
      '-Dspark.driver.port=46539' \
      '-Dspark.authenticate=false' \
      '-Dspark.network.crypto.enabled=false' \
      '-Dspark.shuffle.service.port=7337' \
      -Dspark.yarn.app.container.log.dir=<LOG_DIR> \
      -XX:OnOutOfMemoryError='kill %p' \
      org.apache.spark.executor.CoarseGrainedExecutorBackend \
      --driver-url \
      spark://CoarseGrainedScheduler@hdp-nano01cn-p001.pek3.wecash.net:46539 \
      --executor-id \
      <executorId> \
      --hostname \
      <hostname> \
      --cores \
      1 \
      --app-id \
      application_1561804288428_204915 \
      --user-class-path \
      file:$PWD/__app__.jar \
      1><LOG_DIR>/stdout \
      2><LOG_DIR>/stderr

  resources:
    __spark_conf__ -> resource { scheme: "hdfs" host: "nameservice1" port: -1 file: "/user/root/.sparkStaging/application_1561804288428_204915/__spark_conf_

===============================================================================
19/09/16 17:18:08 INFO client.RMProxy: Connecting to ResourceManager at master01/10.40.66.249:8030
19/09/16 17:18:08 INFO yarn.YarnRMClient: Registering the ApplicationMaster
```

Figure 9: YARN executor launch context

# ApplicationMaster

## Request Resources

```
override def onStart() {
  logInfo( msg = "Connecting to driver: " + driverUrl)
  rpcEnv.asyncSetupEndpointRefByURI(driverUrl).flatMap { ref =>
    // This is a very fast action so we can use "ThreadUtils.sameThread"
    driver = Some(ref)
    ref.ask[Boolean](RegisterExecutor(executorId, self, hostname, cores, extractLogUrls))
  }(ThreadUtils.sameThread).onComplete {
    // This is a very fast action so we can use "ThreadUtils.sameThread"
    case Success(msg) =>
      // Always receive `true`. Just ignore it
    case Failure(e) =>
      exitExecutor( code = 1,  reason = s"Cannot register with driver: $driverUrl", e, notifyDriver = false)
  }(ThreadUtils.sameThread)
}
```

Figure 10: CoarseGrainedExecutorBackend#onStart

# ApplicationMaster

## Request Resources

```
override def receiveAndReply(context: RpcCallContext): PartialFunction[Any, Unit] = {

  case RegisterExecutor(executorId, executorRef, hostname, cores, logUrls) =>
    if (executorDataMap.contains(executorId)) {...} else if (scheduler.nodeBlacklist != null &&
      scheduler.nodeBlacklist.contains(hostname)) {...} else {
      // If the executor's rpc env is not listening for incoming connections, `hostPort`
      // will be null, and the client connection should be used to contact the executor.
      val executorAddress = if (executorRef.address != null) {...} else {...}
      logInfo( msg = s"Registered executor $executorRef ($executorAddress) with ID $executorId")
      addressToExecutorId(executorAddress) = executorId
      totalCoreCount.addAndGet(cores)
      totalRegisteredExecutors.addAndGet( delta = 1)
      val data = new ExecutorData(...)
      /.../
      CoarseGrainedSchedulerBackend.this.synchronized (...)
      executorRef.send(RegisteredExecutor)
      // Note: some tests expect the reply to come after we put the executor in the map
      context.reply( response = true)
      listenerBus.post(...)
      makeOffers()
    }
```

Figure 11: DriverEndpoint#receiveAndReply

# ApplicationMaster

## Request Resources

```
override def receive: PartialFunction[Any, Unit] = {
  case RegisteredExecutor =>
    logInfo( msg = "Successfully registered with driver")
    try {
      executor = new Executor(executorId, hostname, env, userClassPath, isLocal = false)
    } catch {
      case NonFatal(e) =>
        exitExecutor( code = 1,  reason = "Unable to create executor due to " + e.getMessage, e)
    }
```

Figure 12: CoarseGrainedExecutorBackend#receive

```
// Start worker thread pool
private val threadPool = {
  val threadFactory = new ThreadFactoryBuilder()
    .setDaemon(true)
    .setNameFormat("Executor task launch worker-%d")
    .setThreadFactory(new ThreadFactory {...})
    .build()
  Executors.newCachedThreadPool(threadFactory).asInstanceOf[ThreadPoolExecutor]
}
```

Figure 13: Executor#threadPool

# ApplicationMaster

## Request Resources

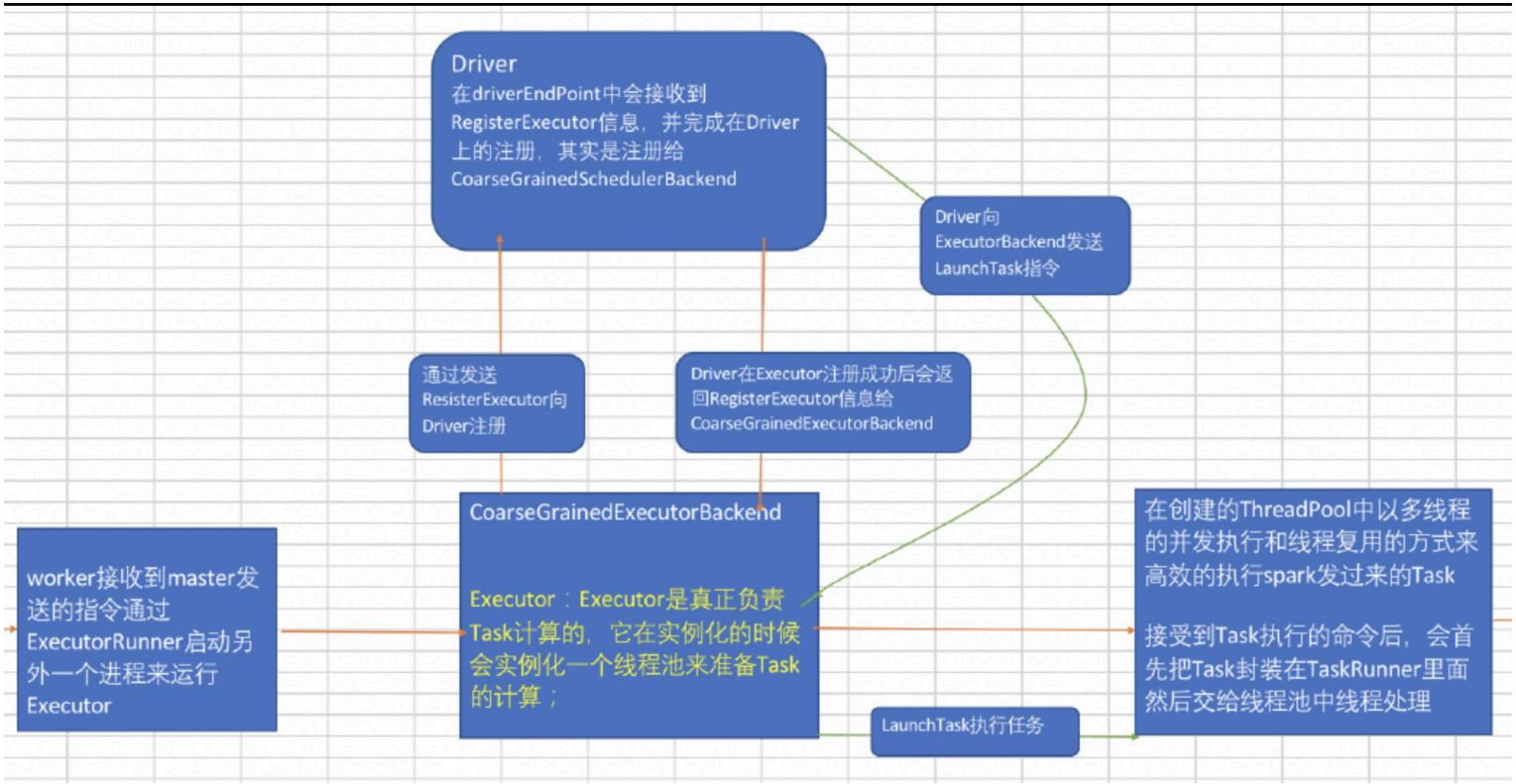



Figure 14: 资源请求流程示意图

# SparkContext

## Main entry point for Spark functionality

A SparkContext represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster.

- LiveListenerBus
- TaskScheduler
- SchedulerBackend
- DAGScheduler
- MapOutputTrackerMaster
- BlockManagerMaster

# SparkContext

## SparkContext Initialization

```scala
import org.apache.spark.sql.SparkSession

val spark = SparkSession
  .builder()
  .appName("Spark SQL basic example")
  .config("spark.some.config.option", "some-value")
  .getOrCreate()

// For implicit conversions like converting RDDs to DataFrames
import spark.implicits._
```

# SparkContext

## SparkContext Initialization

```
// Create and start the scheduler
val (sched, ts) = SparkContext.createTaskScheduler(this, master, deployMode)
_schedulerBackend = sched
_taskScheduler = ts
_dagScheduler = new DAGScheduler( sc = this)
_heartbeatReceiver.ask[Boolean](TaskSchedulerIsSet)

// start TaskScheduler after taskScheduler sets DAGScheduler reference in DAGScheduler's
// constructor
_taskScheduler.start()
```

Figure 15: SparkContext#init

# SparkContext

## Spark Local

```
// When running locally, don't try to re-execute tasks on failure.
val MAX_LOCAL_TASK_FAILURES = 1

master match {
  case "local" =>
    val scheduler = new TaskSchedulerImpl(sc, MAX_LOCAL_TASK_FAILURES, isLocal = true)
    val backend = new LocalSchedulerBackend(sc.getConf, scheduler, totalCores = 1)
    scheduler.initialize(backend)
    (backend, scheduler)
```

Figure 16: SparkContext#createTaskScheduler

```
/**
 * Used when running a local version of Spark where the executor, backend, and master all run in
 * the same JVM. It sits behind a [[TaskSchedulerImpl]] and handles launching tasks on a single
 * Executor (created by the [[LocalSchedulerBackend]]) running locally.
 */
private[spark] class LocalSchedulerBackend(
    conf: SparkConf,
    scheduler: TaskSchedulerImpl,
    val totalCores: Int)
  extends SchedulerBackend with ExecutorBackend with Logging {...}
```

Figure 17: LocalSchedulerBackend

# SparkContext

## Spark on Yarn

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

Figure 18: SparkContext#createTaskScheduler

# SparkContext

## Spark on Yarn

```scala
package org.apache.spark.scheduler

import org.apache.spark.SparkContext

/**
 * A cluster manager interface to plugin external scheduler.
 */
private[spark] trait ExternalClusterManager {

  /**...*/
  def canCreate(masterURL: String): Boolean

  /**...*/
  def createTaskScheduler(sc: SparkContext, masterURL: String): TaskScheduler

  /**...*/
  def createSchedulerBackend(sc: SparkContext,
      masterURL: String,
      scheduler: TaskScheduler): SchedulerBackend

  /**...*/
  def initialize(scheduler: TaskScheduler, backend: SchedulerBackend): Unit
}
```

Figure 19: ExternalClusterManager

# SparkContext

## Spark on Yarn

```scala
/**
 * Cluster Manager for creation of Yarn scheduler and backend
 */
private[spark] class YarnClusterManager extends ExternalClusterManager {

  override def canCreate(masterURL: String): Boolean = {...}

  override def createTaskScheduler(sc: SparkContext, masterURL: String): TaskScheduler = {
    sc.deployMode match {
      case "cluster" => new YarnClusterScheduler(sc)
      case "client" => new YarnScheduler(sc)
      case _ => throw new SparkException(s"Unknown deploy mode '${sc.deployMode}' for Yarn")
    }
  }

  override def createSchedulerBackend(sc: SparkContext,
      masterURL: String,
      scheduler: TaskScheduler): SchedulerBackend = {
    sc.deployMode match {
      case "cluster" =>
        new YarnClusterSchedulerBackend(scheduler.asInstanceOf[TaskSchedulerImpl], sc)
      case "client" =>
        new YarnClientSchedulerBackend(scheduler.asInstanceOf[TaskSchedulerImpl], sc)
      case  _ =>
        throw new SparkException(s"Unknown deploy mode '${sc.deployMode}' for Yarn")
    }
  }

  override def initialize(scheduler: TaskScheduler, backend: SchedulerBackend): Unit = {
    scheduler.asInstanceOf[TaskSchedulerImpl].initialize(backend)
  }
}
```

Figure 20: YarnClusterManager

# SparkContext

## Spark on Yarn

```scala
def initialize(backend: SchedulerBackend) {
  this.backend = backend
  schedulableBuilder = {
    schedulingMode match {
      case SchedulingMode.FIFO =>
        new FIFOSchedulableBuilder(rootPool)
      case SchedulingMode.FAIR =>
        new FairSchedulableBuilder(rootPool, conf)
      case _ =>
        throw new IllegalArgumentException(s"Unsupported $SCHEDULER_MODE_PROPERTY: " +
        s"$schedulingMode")
    }
  }
  schedulableBuilder.buildPools()
}
```

Figure 21: TaskSchedulerImpl#initialize

# SparkContext

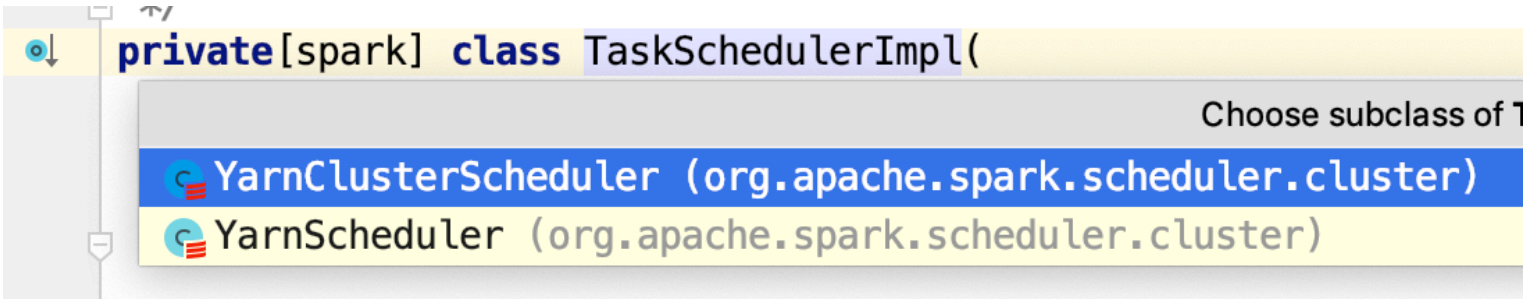
## Spark on Yarn

```
// Create and start the scheduler
val (sched, ts) = SparkContext.createTaskScheduler(this, master, deployMode)
_schedulerBackend = sched
_taskScheduler = ts
_dagScheduler = new DAGScheduler( sc = this)
_heartbeatReceiver.ask[Boolean](TaskSchedulerIsSet)

// start TaskScheduler after taskScheduler sets DAGScheduler reference in DAGScheduler's
// constructor
_taskScheduler.start()
```

Figure 22: SparkContext#init

# SparkContext

## Spark on Yarn

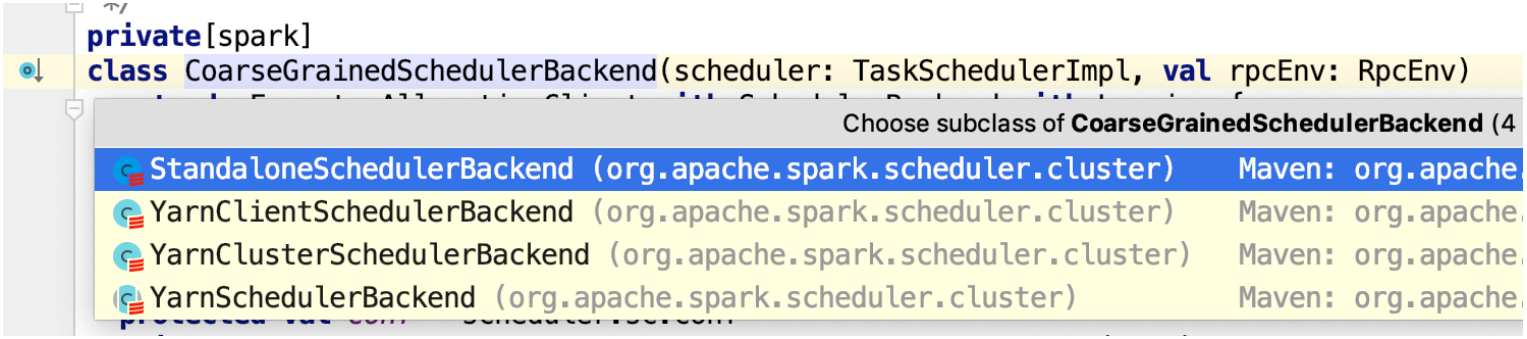



```
override def start() {
  backend.start()

  if (!isLocal && conf.getBoolean( key = "spark.speculation",  defaultValue = false)) {...}
}
```

Figure 23: TaskSchedulerImpl#start

# SparkContext

## Spark on Yarn Cluster




```
private[spark] class YarnClusterSchedulerBackend(
    scheduler: TaskSchedulerImpl,
    sc: SparkContext)
  extends YarnSchedulerBackend(scheduler, sc) {

  override def start() {
    val attemptId = ApplicationMaster.getAttemptId
    bindToYarn(attemptId.getApplicationId(), Some(attemptId))
    super.start()
    totalExpectedExecutors = SchedulerBackendUtils.getInitialTargetExecutorNumber(sc.conf)
  }
```

Figure 24: YarnClusterSchedulerBackend#start

# SparkContext

## Spark on Yarn Client

```scala
/**
 * Create a Yarn client to submit an application to the ResourceManager.
 * This waits until the application is running.
 */
override def start() {
  val driverHost = conf.get("spark.driver.host")
  val driverPort = conf.get("spark.driver.port")
  val hostport = driverHost + ":" + driverPort
  sc.ui.foreach { ui => conf.set("spark.driver.appUIAddress", ui.webUrl) }

  val argsArrayBuf = new ArrayBuffer[String]()
  argsArrayBuf += ("--arg", hostport)

  logDebug( msg = "ClientArguments called with: " + argsArrayBuf.mkString(" "))
  val args = new ClientArguments(argsArrayBuf.toArray)
  totalExpectedExecutors = SchedulerBackendUtils.getInitialTargetExecutorNumber(conf)
  client = new Client(args, conf)
  bindToYarn(client.submitApplication(), None)

  /.../
  super.start()
  waitForApplication()
```

Figure 25: YarnClientSchedulerBackend#start

# SparkContext

## Spark on Yarn

```
override def start() {
  val properties = new ArrayBuffer[(String, String)]
  for ((key, value) <- scheduler.sc.conf.getAll) {...}

  // TODO (prashant) send conf instead of properties
  driverEndpoint = createDriverEndpointRef(properties)
}
```

Figure 26: CoarseGrainedSchedulerBackend#start

# SparkContext

## Spark on Yarn

```
override def createDriverEndpoint(properties: Seq[(String, String)]): DriverEndpoint = {
  new YarnDriverEndpoint(rpcEnv, properties)
}
```

Figure 27: YarnSchedulerBackend#createDriverEndpoint

```
/**
 * Override the DriverEndpoint to add extra logic for the case when an executor is disconnected.
 * This endpoint communicates with the executors and queries the AM for an executor's exit
 * status when the executor is disconnected.
 */
private class YarnDriverEndpoint(rpcEnv: RpcEnv, sparkProperties: Seq[(String, String)])
    extends DriverEndpoint(rpcEnv, sparkProperties) {

  /**...*/
  override def onDisconnected(rpcAddress: RpcAddress): Unit = {...}
}
```

Figure 28: YarnDriverEndpoint

# SparkContext

## Spark on Yarn

```
// Post init
_taskScheduler.postStartHook()
_env.metricsSystem.registerSource(_dagScheduler.metricsSource)
_env.metricsSystem.registerSource(new BlockManagerSource(_env.blockManager))
```

Figure 29: SparkContext#init

```
// Invoked after system has successfully initialized (typically in spark context).
// Yarn uses this to bootstrap allocation of resources based on preferred locations,
// wait for slave registrations, etc.
def postStartHook() { }
```

Figure 30: TaskScheduler#postStartHook

# Outline

# Spark Program

## Word Count

```scala
import org.apache.spark.sql.SparkSession

object WordCount {
  def main(args: Array[String]): Unit = {

    val spark = SparkSession.builder()
      .master( master = "local")
      .appName(this.getClass.getSimpleName)
      .getOrCreate()

    val input = "data.txt"
    val rdd1 = spark.sparkContext.textFile(input)
    val rdd2 = rdd1.flatMap(_.split( regex = " "))
    val rdd3 = rdd2.map(_.toLowerCase)
    val rdd4 = rdd3.map((_, 1))
    val rdd5 = rdd4.reduceByKey(_ + _)
    val rdd6 = rdd5.filter(!_._1.startsWith("a"))
    val rdd7 = rdd6.map {
      case (word, count) => (count, word)
    }
    val rdd8 = rdd7.sortBy(identity)

    val output = "output"
    rdd8.saveAsTextFile(output)

  }
}
```

Figure 31: code

# Spark Program

## Word Count

```
1  I have a dog
2  My mother has a cat
3  My father has a bird
4  My sister has a bike
5  My brother has a car
```

Figure 32: data

# Spark Program

## Word Count

```
1  (1,bike)
2  (1,bird)
3  (1,brother)
4  (1,car)
5  (1,cat)
6  (1,dog)
7  (1,father)
8  (1,have)
9  (1,i)
10 (1,mother)
11 (1,sister)
12 (4,has)
13 (4,my)
14
```

Figure 33: result

# The key concepts

## Concept

1. Jobs
2. Stage
3. Task
4. Cache tracking
5. Preferred locations
6. Cleanup

# The key concepts

## Concept

There are three kinds of transformations:

- narrow dependencies
- wide dependencies

There are three kinds of actions:

- Actions to view data in the console
- Actions to collect data to native object in the respective language
- Actions to write output data sources

# The key concepts

## Spark UI


Figure 34: Job

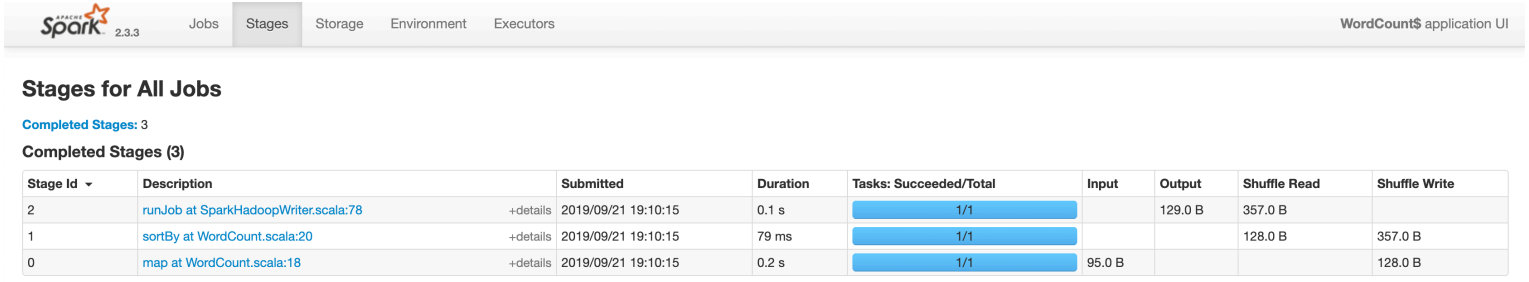



Figure 35: Stage

# The high-level scheduling layer

## DAGScheduler

The high-level scheduling layer that implements stage-oriented scheduling.

It computes a DAG of stages for each job, keeps track of which RDDs and stage outputs are materialized, and finds a minimal schedule to run the job.

Spark stages are created by breaking the RDD graph at shuffle boundaries.

# The high-level scheduling layer

## DAGScheduler

In addition to coming up with a DAG of stages, the DAGScheduler also determines the preferred locations to run each task on, based on the current cache status, and passes these to the low-level TaskScheduler.

Furthermore, it handles failures due to shuffle output files being lost, in which case old stages may need to be resubmitted.

# The high-level scheduling layer

## DAGScheduler

```
private[scheduler] def handleJobSubmitted(jobId: Int,
    finalRDD: RDD[_],
    func: (TaskContext, Iterator[_]) => _,
    partitions: Array[Int],
    callSite: CallSite,
    listener: JobListener,
    properties: Properties) {
  var finalStage: ResultStage = null
  try {
    // New stage creation may throw an exception if, for example, jobs are run on a
    // HadoopRDD whose underlying HDFS files have been deleted.
    finalStage = createResultStage(finalRDD, func, partitions, jobId, callSite)
  } catch {
    case e: Exception =>
      ...
  }

  val job = new ActiveJob(jobId, finalStage, callSite, listener, properties)
  clearCacheLocs()
  logInfo("Got job %s (%s) with %d output partitions".format(...))
  logInfo( msg = "Final stage: " + finalStage + " (" + finalStage.name + ")")
  logInfo( msg = "Parents of final stage: " + finalStage.parents)
  logInfo( msg = "Missing parents: " + getMissingParentStages(finalStage))

  val jobSubmissionTime = clock.getTimeMillis()
  jobIdToActiveJob(jobId) = job
  activeJobs += job
  finalStage.setActiveJob(job)
  val stageIds = jobIdToStageIds(jobId).toArray
  val stageInfos = stageIds.flatMap(id => stageIdToStage.get(id).map(_.latestInfo))
  listenerBus.post(...)
  submitStage(finalStage)
}
```

Figure 36: DAGScheduler#handleJobSubmitted

# The high-level scheduling layer

## DAGScheduler

```
/**
 * Create a ResultStage associated with the provided jobId.
 */
private def createResultStage(
    rdd: RDD[_],
    func: (TaskContext, Iterator[_]) => _,
    partitions: Array[Int],
    jobId: Int,
    callSite: CallSite): ResultStage = {
  val parents = getOrCreateParentStages(rdd, jobId)
  val id = nextStageId.getAndIncrement()
  val stage = new ResultStage(id, rdd, func, partitions, parents, jobId, callSite)
  stageIdToStage(id) = stage
  updateJobIdStageIdMaps(jobId, stage)
  stage
}
```

Figure 37: DAGScheduler#createResultStage

# The high-level scheduling layer

## DAGScheduler

```
/**
 * Get or create the list of parent stages for a given RDD.  The new Stages will be created with
 * the provided firstJobId.
 */
private def getOrCreateParentStages(rdd: RDD[_], firstJobId: Int): List[Stage] = {
  getShuffleDependencies(rdd).map { shuffleDep =>
    getOrCreateShuffleMapStage(shuffleDep, firstJobId)
  }.toList
}
```

Figure 38: DAGScheduler#getOrCreateParentStages

# The high-level scheduling layer

## DAGScheduler

```scala
/**
 * Returns shuffle dependencies that are immediate parents of the given RDD.
 *
 * This function will not return more distant ancestors.  For example, if C has a shuffle
 * dependency on B which has a shuffle dependency on A:
 *
 * A <-- B <-- C
 *
 * calling this function with rdd C will only return the B <-- C dependency.
 *
 * This function is scheduler-visible for the purpose of unit testing.
 */
private[scheduler] def getShuffleDependencies(
    rdd: RDD[_]): HashSet[ShuffleDependency[_, _, _]] = {
  val parents = new HashSet[ShuffleDependency[_, _, _]]
  val visited = new HashSet[RDD[_]]
  val waitingForVisit = new ArrayStack[RDD[_]]
  waitingForVisit.push(rdd)
  while (waitingForVisit.nonEmpty) {
    val toVisit = waitingForVisit.pop()
    if (!visited(toVisit)) {
      visited += toVisit
      toVisit.dependencies.foreach {
        case shuffleDep: ShuffleDependency[_, _, _] =>
          parents += shuffleDep
        case dependency =>
          waitingForVisit.push(dependency.rdd)
      }
    }
  }
  parents
}
```

Figure 39: DAGScheduler#getShuffleDependencies

# The high-level scheduling layer

## DAGScheduler

```scala
/**
 * Gets a shuffle map stage if one exists in shuffleIdToMapStage. Otherwise, if the
 * shuffle map stage doesn't already exist, this method will create the shuffle map stage in
 * addition to any missing ancestor shuffle map stages.
 */
private def getOrCreateShuffleMapStage(
    shuffleDep: ShuffleDependency[_, _, _],
    firstJobId: Int): ShuffleMapStage = {
  shuffleIdToMapStage.get(shuffleDep.shuffleId) match {
    case Some(stage) =>
      stage

    case None =>
      // Create stages for all missing ancestor shuffle dependencies.
      getMissingAncestorShuffleDependencies(shuffleDep.rdd).foreach { dep =>
        // Even though getMissingAncestorShuffleDependencies only returns shuffle dependencies
        // that were not already in shuffleIdToMapStage, it's possible that by the time we
        // get to a particular dependency in the foreach loop, it's been added to
        // shuffleIdToMapStage by the stage creation process for an earlier dependency. See
        // SPARK-13902 for more information.
        if (!shuffleIdToMapStage.contains(dep.shuffleId)) {
          createShuffleMapStage(dep, firstJobId)
        }
      }
      // Finally, create a stage for the given shuffle dependency.
      createShuffleMapStage(shuffleDep, firstJobId)
  }
}
```

Figure 40: DAGScheduler#getOrCreateShuffleMapStage

# The high-level scheduling layer

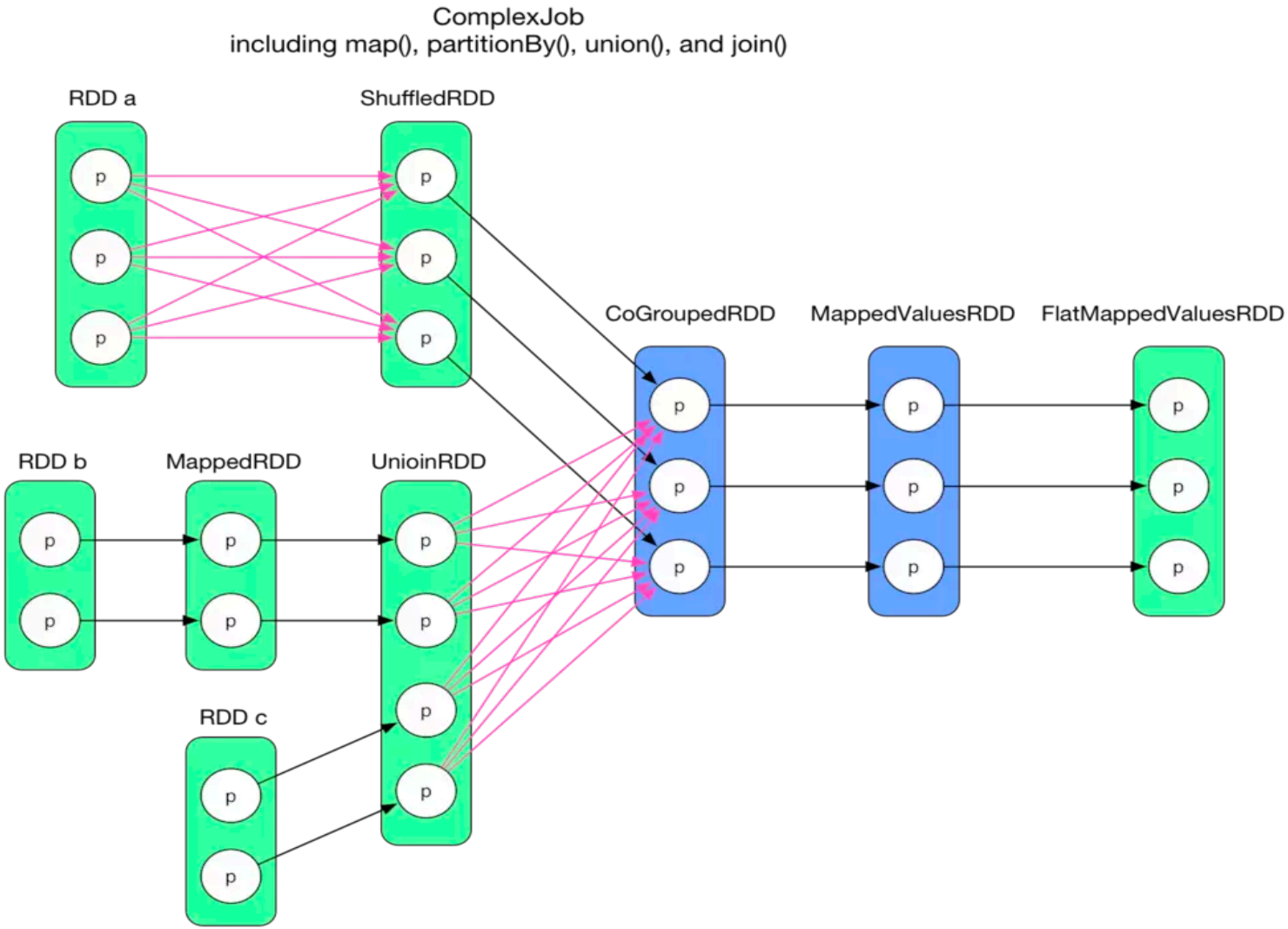
## DAGScheduler

```
,

/** Find ancestor shuffle dependencies that are not registered in shuffleToMapStage yet */
private def getMissingAncestorShuffleDependencies(
    rdd: RDD[_]): ArrayStack[ShuffleDependency[_, _, _]] = {
  val ancestors = new ArrayStack[ShuffleDependency[_, _, _]]
  val visited = new HashSet[RDD[_]]
  // We are manually maintaining a stack here to prevent StackOverflowError
  // caused by recursively visiting
  val waitingForVisit = new ArrayStack[RDD[_]]
  waitingForVisit.push(rdd)
  while (waitingForVisit.nonEmpty) {
    val toVisit = waitingForVisit.pop()
    if (!visited(toVisit)) {
      visited += toVisit
      getShuffleDependencies(toVisit).foreach { shuffleDep =>
        if (!shuffleIdToMapStage.contains(shuffleDep.shuffleId)) {
          ancestors.push(shuffleDep)
          waitingForVisit.push(shuffleDep.rdd)
        } // Otherwise, the dependency and its ancestors have already been registered.
      }
    }
  }
  ancestors
}
```

Figure 41: DAGScheduler#getMissingAncestorShuffleDependencies

# The high-level scheduling layer

## DAGScheduler



# The high-level scheduling layer

## DAGScheduler

It then submits stages as TaskSets to an underlying TaskScheduler implementation that runs them on the cluster.

A TaskSet contains fully independent tasks that can run right away based on the data that's already on the cluster (e.g. map output files from previous stages), though it may fail if this data becomes unavailable.

# The high-level scheduling layer

## DAGScheduler

```
...
var taskBinary: Broadcast[Array[Byte]] = null
var partitions: Array[Partition] = null
try {
  /.../
  var taskBinaryBytes: Array[Byte] = null
  /.../
  RDDCheckpointData.synchronized {
    taskBinaryBytes = stage match {
      case stage: ShuffleMapStage =>
        JavaUtils.bufferToArray(
          closureSerializer.serialize((stage.rdd, stage.shuffleDep): AnyRef))
      case stage: ResultStage =>
        JavaUtils.bufferToArray(closureSerializer.serialize((stage.rdd, stage.func): AnyRef))
    }

    partitions = stage.rdd.partitions
  }

  taskBinary = sc.broadcast(taskBinaryBytes)
} catch {
  // In the case of a failure during serialization, abort the stage.
  case e: NotSerializableException =>
    ...
  case NonFatal(e) =>
```

Figure 42: DAGScheduler#submitMissingTasks

# The high-level scheduling layer

## DAGScheduler

```
val tasks: Seq[Task[_]] = try {
  val serializedTaskMetrics = closureSerializer.serialize(stage.latestInfo.taskMetrics).array()
  stage match {
    case stage: ShuffleMapStage =>
      stage.pendingPartitions.clear()
      partitionsToCompute.map { id =>
        val locs = taskIdToLocations(id)
        val part = partitions(id)
        stage.pendingPartitions += id
        new ShuffleMapTask(stage.id, stage.latestInfo.attemptNumber,
          taskBinary, part, locs, properties, serializedTaskMetrics, Option(jobId),
          Option(sc.applicationId), sc.applicationAttemptId)
      }

    case stage: ResultStage =>
      partitionsToCompute.map { id =>
        val p: Int = stage.partitions(id)
        val part = partitions(p)
        val locs = taskIdToLocations(id)
        new ResultTask(stage.id, stage.latestInfo.attemptNumber,
          taskBinary, part, locs, id, properties, serializedTaskMetrics,
          Option(jobId), Option(sc.applicationId), sc.applicationAttemptId)
      }
  }
} catch {
  case NonFatal(e) =>
    ...
}
```

Figure 43: DAGScheduler#submitMissingTasks

# The high-level scheduling layer

## DAGScheduler

```
if (tasks.size > 0) {
  logInfo( msg = s"Submitting ${tasks.size} missing tasks from $stage (${stage.rdd}) (first 15 " +
    s"tasks are for partitions ${tasks.take(15).map(_.partitionId)})")
  taskScheduler.submitTasks(new TaskSet(
    tasks.toArray, stage.id, stage.latestInfo.attemptNumber, jobId, properties))
} else {
  // Because we posted SparkListenerStageSubmitted earlier, we should mark
  // the stage as completed here in case there are no tasks to run
  markStageAsFinished(stage, None)

  val debugString = stage match {...}
  logDebug(debugString)

  submitWaitingChildStages(stage)
}
```

Figure 44: DAGScheduler#submitMissingTasks

# The Low-level scheduling layer

## TaskScheduler

```scala
/**
 * A set of tasks submitted together to the low-level TaskScheduler, usually representing
 * missing partitions of a particular stage.
 */
private[spark] class TaskSet(
    val tasks: Array[Task[_]],
    val stageId: Int,
    val stageAttemptId: Int,
    val priority: Int,
    val properties: Properties) {
  val id: String = stageId + "." + stageAttemptId

  override def toString: String = "TaskSet " + id
}
```

Figure 45: TaskSet

# The Low-level scheduling layer

## TaskScheduler

```
override def submitTasks(taskSet: TaskSet) {
  val tasks = taskSet.tasks
  logInfo( msg = "Adding task set " + taskSet.id + " with " + tasks.length + " tasks")
  this.synchronized {
    val manager = createTaskSetManager(taskSet, maxTaskFailures)
    val stage = taskSet.stageId
    val stageTaskSets =
      taskSetsByStageIdAndAttempt.getOrElseUpdate(stage, new HashMap[Int, TaskSetManager])
    stageTaskSets(taskSet.stageAttemptId) = manager
    val conflictingTaskSet = stageTaskSets.exists { case (_, ts) =>
      ts.taskSet != taskSet && !ts.isZombie
    }
    if (conflictingTaskSet) {
      throw new IllegalStateException(s"more than one active taskSet for stage $stage:" +
        s" ${stageTaskSets.toSeq.map{_._2.taskSet.id}.mkString(",")}")
    }
    schedulableBuilder.addTaskSetManager(manager, manager.taskSet.properties)

    if (!isLocal && !hasReceivedTask) {...}
    hasReceivedTask = true
  }
  backend.reviveOffers()
}
```

Figure 46: TaskSchedulerImpl#submitTasks

# The Low-level scheduling layer

## CoarseGrainedSchedulerBackend

A scheduler backend that waits for coarse-grained executors to connect.

This backend holds onto each executor for the duration of the Spark job rather than relinquishing executors whenever a task is done and asking the scheduler to launch a new executor for each new task.

```
override def reviveOffers() {
  driverEndpoint.send(ReviveOffers)
}
```

Figure 47: CoarseGrainedSchedulerBackend#reviveOffers

# The Low-level scheduling layer

## DriverEndpoint

```
override def receive: PartialFunction[Any, Unit] = {
  case StatusUpdate(executorId, taskId, state, data) =>
    ...

  case ReviveOffers =>
    makeOffers()

  case KillTask(taskId, executorId, interruptThread, reason) =>
    executorDataMap.get(executorId) match {...}

  case KillExecutorsOnHost(host) =>
    scheduler.getExecutorsAliveOnHost(host).foreach (...)

  case UpdateDelegationTokens(newDelegationTokens) =>
    executorDataMap.values.foreach (...)

  case RemoveExecutor(executorId, reason) =>
    /.../
    executorDataMap.get(executorId).foreach(_.executorEndpoint.send(StopExecutor))
    removeExecutor(executorId, reason)
}
```

Figure 48: CoarseGrainedSchedulerBackend.DriverEndpoint#receive

# The Low-level scheduling layer

## DriverEndpoint

```
// Make fake resource offers on all executors
private def makeOffers() {
  // Make sure no executor is killed while some task is launching on it
  val taskDescs = CoarseGrainedSchedulerBackend.this.synchronized {
    // Filter out executors under killing
    val activeExecutors = executorDataMap.filterKeys(executorIsAlive)
    val workOffers = activeExecutors.map {
      case (id, executorData) =>
        new WorkerOffer(id, executorData.executorHost, executorData.freeCores)
    }.toIndexedSeq
    scheduler.resourceOffers(workOffers)
  }
  if (!taskDescs.isEmpty) {
    launchTasks(taskDescs)
  }
}
```

Figure 49: CoarseGrainedSchedulerBackend.DriverEndpoint#makeOffers

# The Low-level scheduling layer

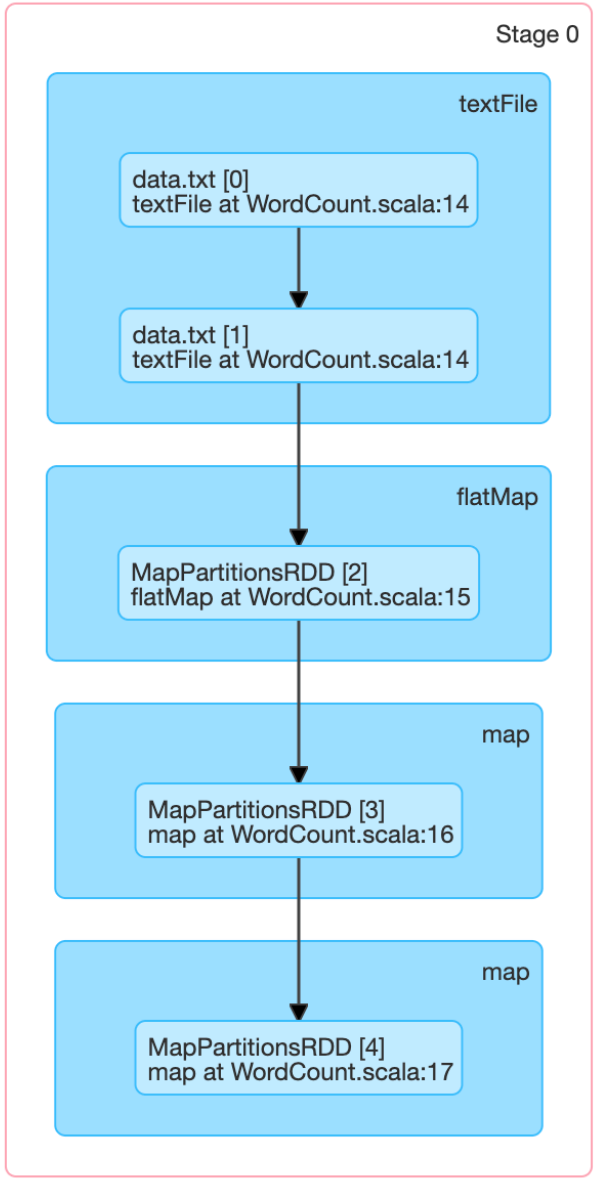
## TaskScheduler

```
// Take each TaskSet in our scheduling order, and then offer it each node in increasing order
// of locality levels so that it gets a chance to launch local tasks on all of them.
// NOTE: the preferredLocality order: PROCESS_LOCAL, NODE_LOCAL, NO_PREF, RACK_LOCAL, ANY
for (taskSet <- sortedTaskSets) {
  var launchedAnyTask = false
  var launchedTaskAtCurrentMaxLocality = false
  for (currentMaxLocality <- taskSet.myLocalityLevels) {
    do {
      launchedTaskAtCurrentMaxLocality = resourceOfferSingleTaskSet(
        taskSet, currentMaxLocality, shuffledOffers, availableCpus, tasks)
      launchedAnyTask |= launchedTaskAtCurrentMaxLocality
    } while (launchedTaskAtCurrentMaxLocality)
  }
  if (!launchedAnyTask) {
    taskSet.abortIfCompletelyBlacklisted(hostToExecutors)
  }
}
```

Figure 50: TaskSchedulerImpl#resourceOffers

# The Low-level scheduling layer

## TaskScheduler



# The Low-level scheduling layer

## DriverEndpoint

```
// Launch tasks returned by a set of resource offers
private def launchTasks(tasks: Seq[Seq[TaskDescription]]) {
  for (task <- tasks.flatten) {
    val serializedTask = TaskDescription.encode(task)
    if (serializedTask.limit() >= maxRpcMessageSize) {...}
    else {
      val executorData = executorDataMap(task.executorId)
      executorData.freeCores -= scheduler.CPUS_PER_TASK

      logDebug( msg = s"Launching task ${task.taskId} on executor id: ${task.executorId} hostname: " +
        s"${executorData.executorHost}.")

      executorData.executorEndpoint.send(LaunchTask(new SerializableBuffer(serializedTask)))
    }
  }
}
```

Figure 51: DriverEndpoint#launchTasks

# Executor

## CoarseGrainedExecutorBackend

```
override def receive: PartialFunction[Any, Unit] = {
  case RegisteredExecutor =>
    ...

  case RegisterExecutorFailed(message) =>
    exitExecutor( code = 1,  reason = "Slave registration failed: " + message)

  case LaunchTask(data) =>
    if (executor == null) {
      exitExecutor( code = 1,  reason = "Received LaunchTask command but executor was null")
    } else {
      val taskDesc = TaskDescription.decode(data.value)
      logInfo( msg = "Got assigned task " + taskDesc.taskId)
      executor.launchTask( context = this, taskDesc)
    }

  case KillTask(taskId, _, interruptThread, reason) =>
    if (executor == null) {...} else {...}

  case StopExecutor =>
    ...

  case Shutdown =>
    ...

  case UpdateDelegationTokens(tokenBytes) =>
    ...
}
```

Figure 52: CoarseGrainedExecutorBackend#receive

# Executor

## Executor

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

Figure 53: Executor#launchTask

```
class TaskRunner(
    execBackend: ExecutorBackend,
    private val taskDescription: TaskDescription)
  extends Runnable {...}
```

Figure 54: TaskRunner

# Shuffle

## Task

A ShuffleMapTask divides the elements of an RDD into multiple buckets (based on a partitioner specified in the ShuffleDependency).

A ResultTask that sends back the output to the driver application.

# Shuffle

## ShuffleMapTask

```scala
override def runTask(context: TaskContext): MapStatus = {
  // Deserialize the RDD using the broadcast variable.
  val threadMXBean = ManagementFactory.getThreadMXBean
  val deserializeStartTime = System.currentTimeMillis()
  val deserializeStartCpuTime = if (threadMXBean.isCurrentThreadCpuTimeSupported) {...} else 0L
  val ser = SparkEnv.get.closureSerializer.newInstance()
  val (rdd, dep) = ser.deserialize[(RDD[_], ShuffleDependency[_, _, _])](...)
  _executorDeserializeTime = System.currentTimeMillis() - deserializeStartTime
  _executorDeserializeCpuTime = if (threadMXBean.isCurrentThreadCpuTimeSupported) {...} else 0L

  var writer: ShuffleWriter[Any, Any] = null
  try {
    val manager = SparkEnv.get.shuffleManager
    writer = manager.getWriter[Any, Any](dep.shuffleHandle, partitionId, context)
    writer.write(rdd.iterator(partition, context).asInstanceOf[Iterator[_ <: Product2[Any, Any]]])
    writer.stop(success = true).get
  } catch {
    case e: Exception =>
      ...
  }
}
```

Figure 55: ShuffleMapTask#runTask

# Shuffle

## ResultTask

```
override def runTask(context: TaskContext): U = {
  // Deserialize the RDD and the func using the broadcast variables.
  val threadMXBean = ManagementFactory.getThreadMXBean
  val deserializeStartTime = System.currentTimeMillis()
  val deserializeStartCpuTime = if (threadMXBean.isCurrentThreadCpuTimeSupported) {...} else 0L
  val ser = SparkEnv.get.closureSerializer.newInstance()
  val (rdd, func) = ser.deserialize[(RDD[T], (TaskContext, Iterator[T]) => U)](...)
  _executorDeserializeTime = System.currentTimeMillis() - deserializeStartTime
  _executorDeserializeCpuTime = if (threadMXBean.isCurrentThreadCpuTimeSupported) {...} else 0L

  func(context, rdd.iterator(partition, context))
}
```

Figure 56: ResultTask#runTask

# Shuffle

## RDD

```scala
/**
 * Internal method to this RDD; will read from cache if applicable, or otherwise compute it.
 * This should ''not'' be called by users directly, but is available for implementors of custom
 * subclasses of RDD.
 */
final def iterator(split: Partition, context: TaskContext): Iterator[T] = {
  if (storageLevel != StorageLevel.NONE) {
    getOrCompute(split, context)
  } else {
    computeOrReadCheckpoint(split, context)
  }
}
```

Figure 57: RDD#iterator

# Shuffle

## RDD

```scala
/**
 * :: DeveloperApi ::
 * Implemented by subclasses to compute a given partition.
 */
@DeveloperApi
def compute(split: Partition, context: TaskContext): Iterator[T]
```

Figure 58: RDD#compute

# Shuffle

## RDD

```scala
override def compute(split: Partition, context: TaskContext): Iterator[(K, C)] = {
  val dep = dependencies.head.asInstanceOf[ShuffleDependency[K, V, C]]
  SparkEnv.get.shuffleManager.getReader(dep.shuffleHandle, split.index, split.index + 1, context)
    .read()
    .asInstanceOf[Iterator[(K, C)]]
}
```

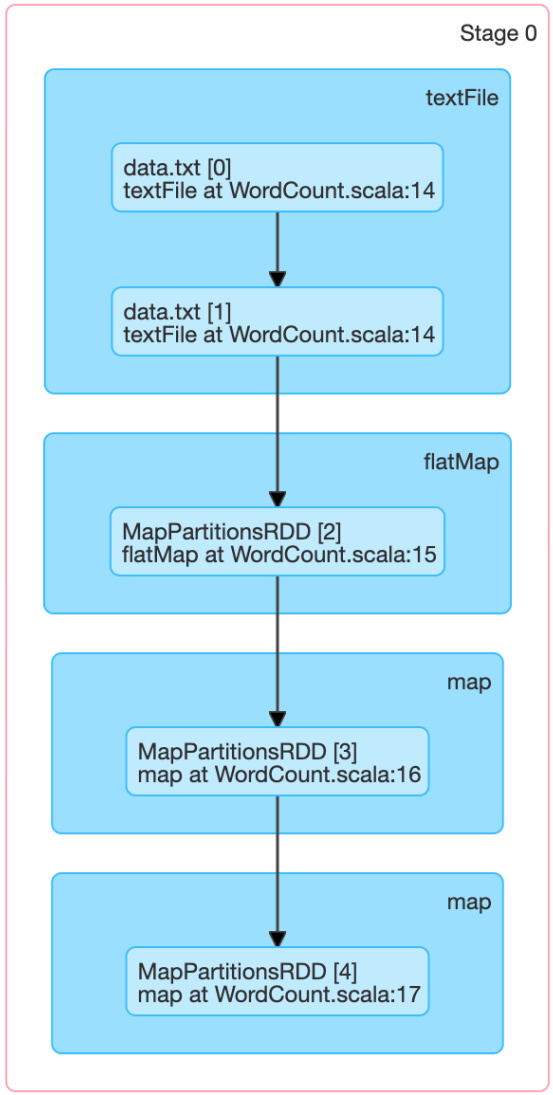
Figure 59: ShuffledRDD#compute

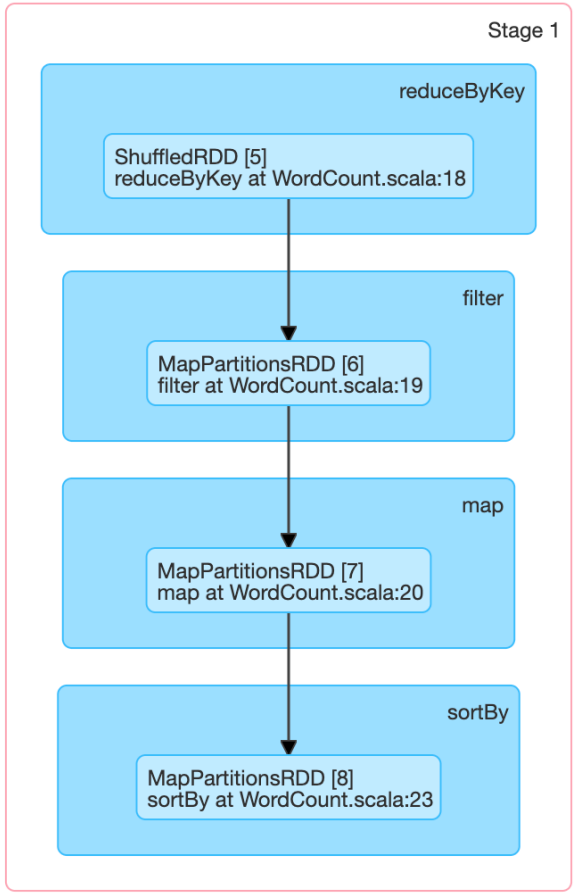
```scala
override def compute(theSplit: Partition, context: TaskContext): InterruptibleIterator[(K, V)] = {
  val iter = new Iterator[(K, V)] {...}
  new InterruptibleIterator(context, iter)
}
```

Figure 60: NewHadoopRDD#compute

# Shuffle

## ShuffleWriter



# Shuffle

## ShuffleWriter

```scala
/** Write a bunch of records to this task's output */
override def write(records: Iterator[Product2[K, V]]): Unit = {
  sorter = if (dep.mapSideCombine) {
    require(dep.aggregator.isDefined, "Map-side combine without Aggregator specified!")
    new ExternalSorter[K, V, C](
      context, dep.aggregator, Some(dep.partitioner), dep.keyOrdering, dep.serializer)
  } else {
    /.../
    new ExternalSorter[K, V, V](
      context, aggregator = None, Some(dep.partitioner), ordering = None, dep.serializer)
  }
  sorter.insertAll(records)

  ...
  val output = shuffleBlockResolver.getDataFile(dep.shuffleId, mapId)
  val tmp = Utils.tempFileWith(output)
  try {
    val blockId = ShuffleBlockId(dep.shuffleId, mapId, IndexShuffleBlockResolver.NOOP_REDUCE_ID)
    val partitionLengths = sorter.writePartitionedFile(blockId, tmp)
    shuffleBlockResolver.writeIndexFileAndCommit(dep.shuffleId, mapId, partitionLengths, tmp)
    mapStatus = MapStatus(blockManager.shuffleServerId, partitionLengths)
  } finally {
    if (tmp.exists() && !tmp.delete()) {...}
  }
}
```

Figure 61: SortShuffleWriter#write

# Shuffle

## ShuffleWriter

```scala
def insertAll(records: Iterator[Product2[K, V]]): Unit = {
  // TODO: stop combining if we find that the reduction factor isn't high
  val shouldCombine = aggregator.isDefined

  if (shouldCombine) {
    // Combine values in-memory first using our AppendOnlyMap
    val mergeValue = aggregator.get.mergeValue
    val createCombiner = aggregator.get.createCombiner
    var kv: Product2[K, V] = null
    val update = (hadValue: Boolean, oldValue: C) => {
      if (hadValue) mergeValue(oldValue, kv._2) else createCombiner(kv._2)
    }
    while (records.hasNext) {
      addElementsRead()
      kv = records.next()
      map.changeValue((getPartition(kv._1), kv._1), update)
      maybeSpillCollection(usingMap = true)
    }
  } else {
    // Stick values into our buffer
    while (records.hasNext) {
      addElementsRead()
      val kv = records.next()
      buffer.insert(getPartition(kv._1), kv._1, kv._2.asInstanceOf[C])
      maybeSpillCollection(usingMap = false)
    }
  }
}
```

Figure 62: ExternalSorter#insertAll

# Shuffle

## ShuffleWriter

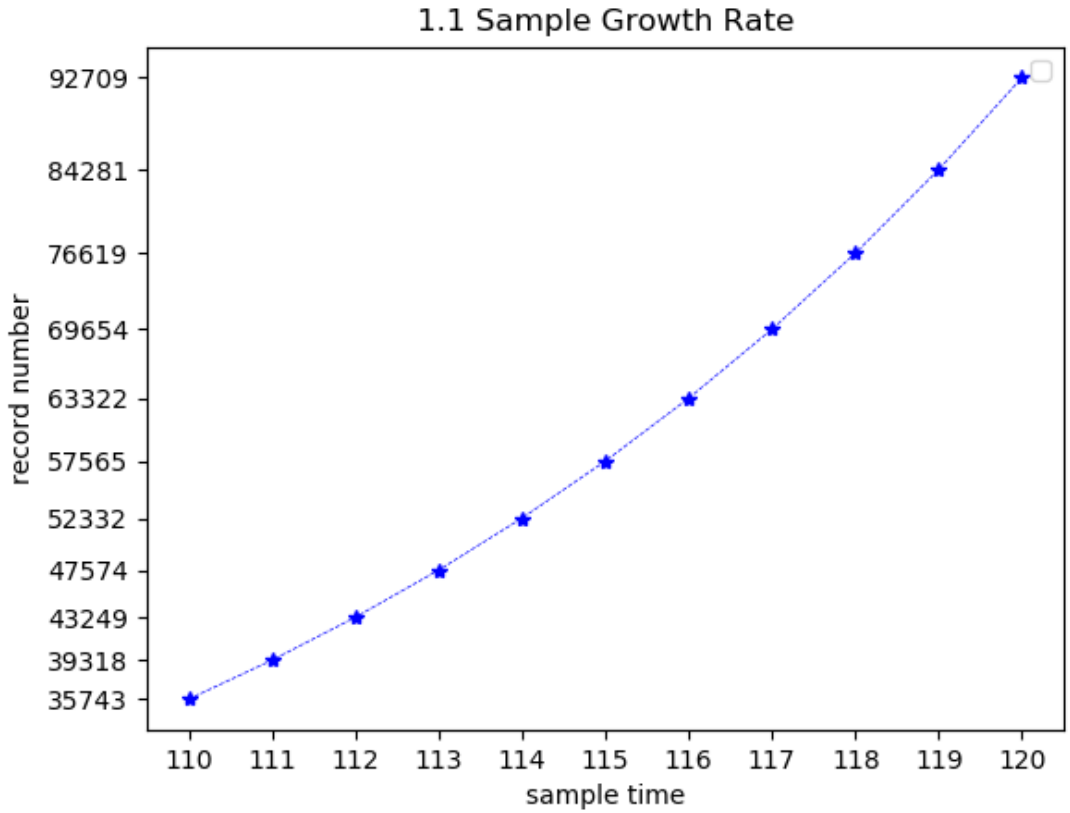



Figure 63: SizeTracker#SAMPLE_GROWTH_RATE

# Shuffle

## ShuffleWriter

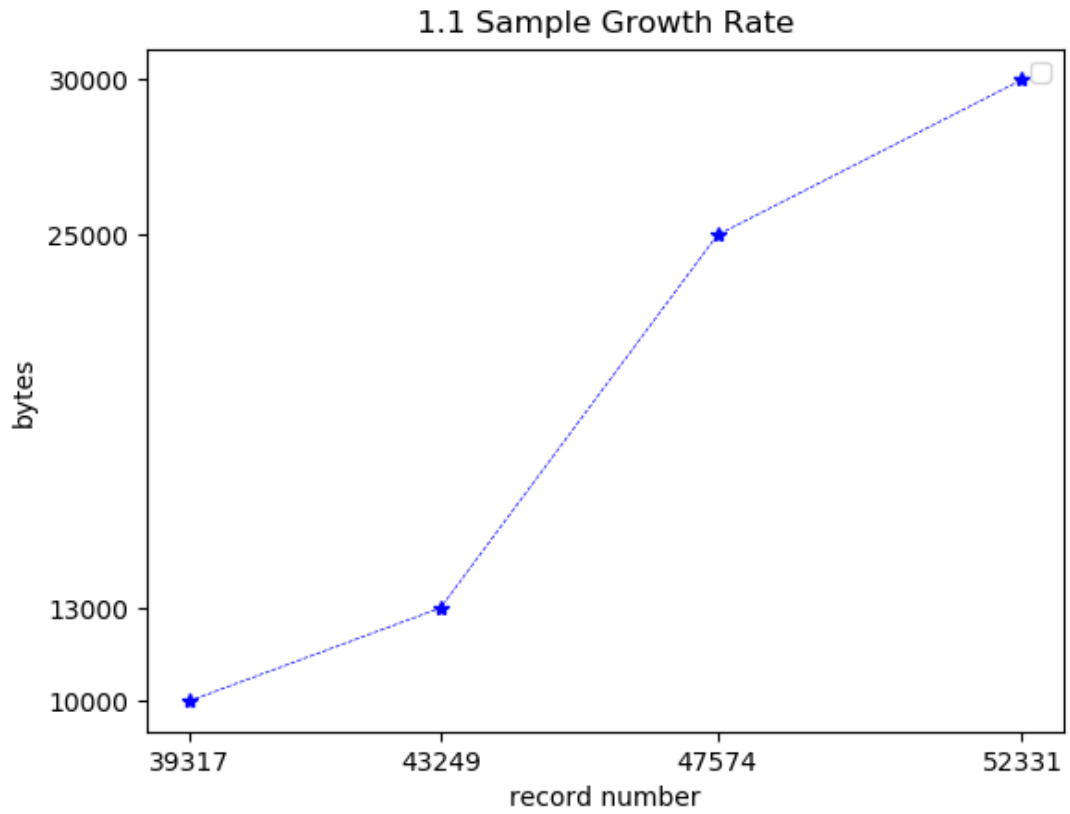



Figure 64: SizeTracker#SAMPLE_GROWTH_RATE

# Shuffle

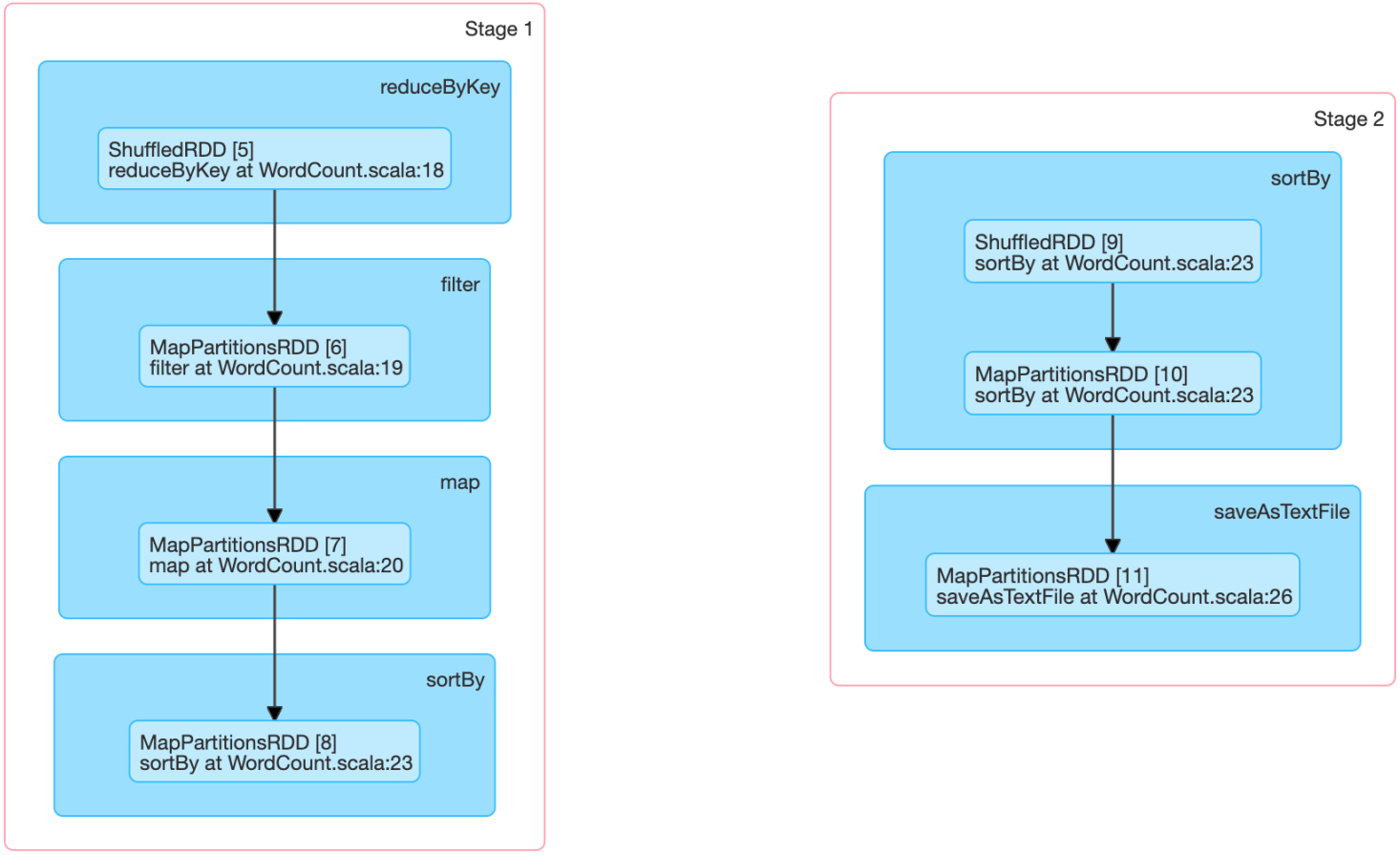
## ShuffleWriter

```
def writePartitionedFile(
    blockId: BlockId,
    outputFile: File): Array[Long] = {

  // Track location of each range in the output file
  val lengths = new Array[Long](numPartitions)
  val writer = blockManager.getDiskWriter(...)

  if (spills.isEmpty) {...} else {...}

  writer.close()
  context.taskMetrics().incMemoryBytesSpilled(memoryBytesSpilled)
  context.taskMetrics().incDiskBytesSpilled(diskBytesSpilled)
  context.taskMetrics().incPeakExecutionMemory(peakMemoryUsedBytes)

  lengths
}
```

Figure 65: ExternalSorter#writePartitionedFile

# Shuffle

## ShuffleReader



# Shuffle

## ShuffleReader

```scala
/** Read the combined key-values for this reduce task */
override def read(): Iterator[Product2[K, C]] = {
  val wrappedStreams = new ShuffleBlockFetcherIterator(...)

  val serializerInstance = dep.serializer.newInstance()

  // Create a key/value iterator for each stream
  val recordIter = wrappedStreams.flatMap (...)

  // Update the context task metrics for each record read.
  val readMetrics = context.taskMetrics.createTempShuffleReadMetrics()
  val metricIter = CompletionIterator[(Any, Any), Iterator[(Any, Any)]](...)

  // An interruptible iterator must be used here in order to support task cancellation
  val interruptibleIter = new InterruptibleIterator[(Any, Any)](context, metricIter)

  val aggregatedIter: Iterator[Product2[K, C]] = if (dep.aggregator.isDefined) {...} else {
    require(!dep.mapSideCombine, "Map-side combine without Aggregator specified!")
    interruptibleIter.asInstanceOf[Iterator[Product2[K, C]]]
  }

  // Sort the output if there is a sort ordering defined.
  dep.keyOrdering match {...}
}
```

Figure 66: BlockStoreShuffleReader#read

# Shuffle

## ShuffleReader

```scala
val aggregatedIter: Iterator[Product2[K, C]] = if (dep.aggregator.isDefined) {
  if (dep.mapSideCombine) {
    // We are reading values that are already combined
    val combinedKeyValuesIterator = interruptibleIter.asInstanceOf[Iterator[(K, C)]]
    dep.aggregator.get.combineCombinersByKey(combinedKeyValuesIterator, context)
  } else {
    ...
    val keyValuesIterator = interruptibleIter.asInstanceOf[Iterator[(K, Nothing)]]
    dep.aggregator.get.combineValuesByKey(keyValuesIterator, context)
  }
} else {
  require(!dep.mapSideCombine, "Map-side combine without Aggregator specified!")
  interruptibleIter.asInstanceOf[Iterator[Product2[K, C]]]
}
```

Figure 67: BlockStoreShuffleReader#read

# Shuffle

## ShuffleReader

```
def combineValuesByKey(
    iter: Iterator[_ <: Product2[K, V]],
    context: TaskContext): Iterator[(K, C)] = {
  val combiners = new ExternalAppendOnlyMap[K, V, C](createCombiner, mergeValue, mergeCombiners)
  combiners.insertAll(iter)
  updateMetrics(context, combiners)
  combiners.iterator
}

def combineCombinersByKey(
    iter: Iterator[_ <: Product2[K, C]],
    context: TaskContext): Iterator[(K, C)] = {
  val combiners = new ExternalAppendOnlyMap[K, C, C](identity, mergeCombiners, mergeCombiners)
  combiners.insertAll(iter)
  updateMetrics(context, combiners)
  combiners.iterator
}
```

Figure 68: Aggregator

# Shuffle

## ShuffleReader

```
// Sort the output if there is a sort ordering defined.
dep.keyOrdering match {
  case Some(keyOrd: Ordering[K]) =>
    // Create an ExternalSorter to sort the data.
    val sorter =
      new ExternalSorter[K, C, C](context, ordering = Some(keyOrd), serializer = dep.serializer)
    sorter.insertAll(aggregatedIter)
    context.taskMetrics().incMemoryBytesSpilled(sorter.memoryBytesSpilled)
    context.taskMetrics().incDiskBytesSpilled(sorter.diskBytesSpilled)
    context.taskMetrics().incPeakExecutionMemory(sorter.peakMemoryUsedBytes)
    CompletionIterator[Product2[K, C], Iterator[Product2[K, C]]](sorter.iterator, sorter.stop())
  case None =>
    aggregatedIter
}
```

Figure 69: BlockStoreShuffleReader#read

# The Final Result

## Actions

There are three kinds of actions:

- Actions to collect data
- Actions to write output data sources
- Actions to view data in the console

# The Final Result

## Back to the driver

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

Figure 70: RDD#count

```scala
def runJob[T, U: ClassTag](
    rdd: RDD[T],
    func: (TaskContext, Iterator[T]) => U,
    partitions: Seq[Int]): Array[U] = {
  val results = new Array[U](partitions.size)
  runJob[T, U](rdd, func, partitions, (index, res) => results(index) = res)
  results
}
```

Figure 71: SparkContext#runJob

# The Final Result

## Back to the driver

```
def runJob[T, U: ClassTag](
    rdd: RDD[T],
    func: (TaskContext, Iterator[T]) => U,
    partitions: Seq[Int],
    resultHandler: (Int, U) => Unit): Unit = {
  if (stopped.get()) {...}
  val callSite = getCallSite
  val cleanedFunc = clean(func)
  logInfo( msg = "Starting job: " + callSite.shortForm)
  if (conf.getBoolean( key = "spark.logLineage",  defaultValue = false)) {...}
  dagScheduler.runJob(rdd, cleanedFunc, partitions, callSite, resultHandler, localProperties.get)
  progressBar.foreach(_.finishAll())
  rdd.doCheckpoint()
}
```

Figure 72: SparkContext#runJob

```
override def taskSucceeded(index: Int, result: Any): Unit = {
  // resultHandler call must be synchronized in case resultHandler itself is not thread safe.
  synchronized {
    resultHandler(index, result.asInstanceOf[T])
  }
  if (finishedTasks.incrementAndGet() == totalTasks) {
    jobPromise.success(())
  }
}
```

Figure 73: JobWaiter#taskSucceeded

# The Final Result

## Back to the driver

```
event.reason match {
  case Success =>
    task match {
      case rt: ResultTask[_, _] =>
        // Cast to ResultStage here because it's part of the ResultTask
        // TODO Refactor this out to a function that accepts a ResultStage
        val resultStage = stage.asInstanceOf[ResultStage]
        resultStage.activeJob match {
          case Some(job) =>
            if (!job.finished(rt.outputId)) {
              job.finished(rt.outputId) = true
              job.numFinished += 1
              // If the whole job has finished, remove it
              if (job.numFinished == job.numPartitions) {...}

              // taskSucceeded runs some user code that might throw an exception. Make sure
              // we are resilient against that.
              try {
                job.listener.taskSucceeded(rt.outputId, event.result)
              } catch {
                case e: Exception =>
                  // TODO: Perhaps we want to mark the resultStage as failed?
                  job.listener.jobFailed(new SparkDriverExecutionException(e))
              }
            }
          case None =>
            logInfo( msg = "Ignoring result from " + rt + " because its job has finished")
        }

      case smt: ShuffleMapTask =>
        ...
    }
```

Figure 74: DAGScheduler#handleTaskCompletion

# The Final Result

## Write to Storage System

- saveAsTextFile
- saveAsNewAPIHadoopDataset
- parquet
- foreachPartition

# The Final Result

## View data in the console

```
/**
 * Displays the Dataset in a tabular form. For example:
 * {{{
 *   year  month AVG('Adj Close) MAX('Adj Close)
 *   1980  12    0.503218        0.595103
 *   1981  01    0.523289        0.570307
 *   1982  02    0.436504        0.475256
 *   1983  03    0.410516        0.442194
 *   1984  04    0.450090        0.483521
 * }}}
 * @param numRows Number of rows to show
 * @param truncate Whether truncate long strings. If true, strings more than 20 characters will
 *              be truncated and all cells will be aligned right
 *
 * @group action
 * @since 1.6.0
 */
// scalastyle:off println
def show(numRows: Int, truncate: Boolean): Unit = if (truncate) {
  println(showString(numRows, truncate = 20))
} else {
  println(showString(numRows, truncate = 0))
}
```

Figure 75: Dataset#show

# Questions and Answers?

Questions and Answers?

Thank You!

# References I