Spark Internals

Taro L. Saito
Treasure Data, Inc.

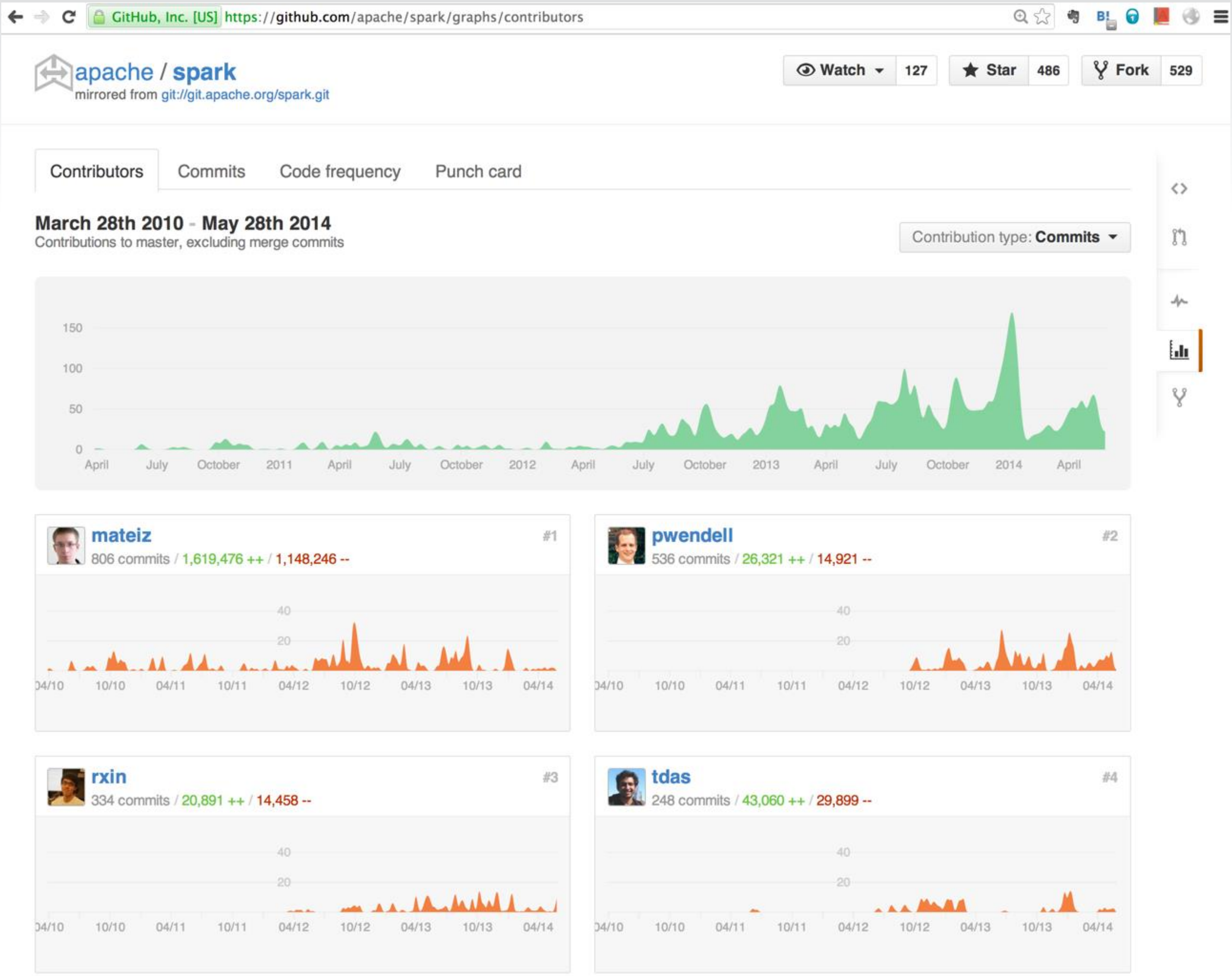
Hadoop Source Code Reading #16
NTT Data, Tokyo
May 29, 2014

Spark Code Base Size

- **spark/core/src/main/scala**
- **2012 (version 0.6.x)**
  - 20,000 lines of code
- **2014 (branch-1.0)**
  - 50,000 lines of code

- **Other components**
  - Spark Streaming
  - Bagel (graph processing library)
  - MLLib (machine learning library)
  - Container support: Mesos, YARN, Docker, etc.
  - Spark SQL (Shark: Hive on Spark)

Spark Core Developers



IntelliJ Tips

- **Install Scala Plugin**

- **Useful commands for code reading**
  - Go to definition (Ctrl + Click)
  - Show Usage
  - Navigate Class/Symbol/File
  - Bookmark, Show Bookmarks
  - Ctrl + Q (Show type info)
  - Find Action (Ctrl + Shift + A)

- **Use your favorite key bindings**

Scala Console (REPL)

- $ brew install scala

```
$ scala
Welcome to Scala version 2.9.1.final (Java HotSpot(TM) 64-Bit Server VM, Java 1.6.0_33).
Type in expressions to have them evaluated.
Type :help for more information.

scala> val l = List("A", "B", "C", "D")
l: List[java.lang.String] = List(A, B, C, D)

scala> l.reverse
res0: List[java.lang.String] = List(D, C, B, A)

# 演算結果は自動的に変数に代入される
scala> res0
res1: List[java.lang.String] = List(D, C, B, A)

scala>
```

Scala Basics

- **object**
  - Singleton, static methods

```
object SparkContext extends Logging {

  private[spark] val SPARK_VERSION = "1.0.0"
```

- **Package-private scope**
  - private[spark]   visible only from spark package.

- **Pattern matching**

```
private def getLocalityWait(level: TaskLocality.TaskLocality): Long = {
  val defaultWait = conf.get("spark.locality.wait", "3000")
  level match {
    case TaskLocality.PROCESS_LOCAL =>
      conf.get("spark.locality.wait.process", defaultWait).toLong
    case TaskLocality.NODE_LOCAL =>
      conf.get("spark.locality.wait.node", defaultWait).toLong
    case TaskLocality.RACK_LOCAL =>
      conf.get("spark.locality.wait.rack", defaultWait).toLong
    case TaskLocality.ANY =>
      0L
  }
}
```

Scala: Case Classes

- **Case classes**
  - Immutable and serializable

```scala
// Driver to executors
case class LaunchTask(task: TaskDescription) extends CoarseGrainedClusterMessage

case class KillTask(taskId: Long, executor: String, interruptThread: Boolean)
  extends CoarseGrainedClusterMessage

case class RegisteredExecutor(sparkProperties: Seq[(String, String)])
  extends CoarseGrainedClusterMessage

case class RegisterExecutorFailed(message: String) extends CoarseGrainedClusterMessage
```

  - Can be used with pattern match.

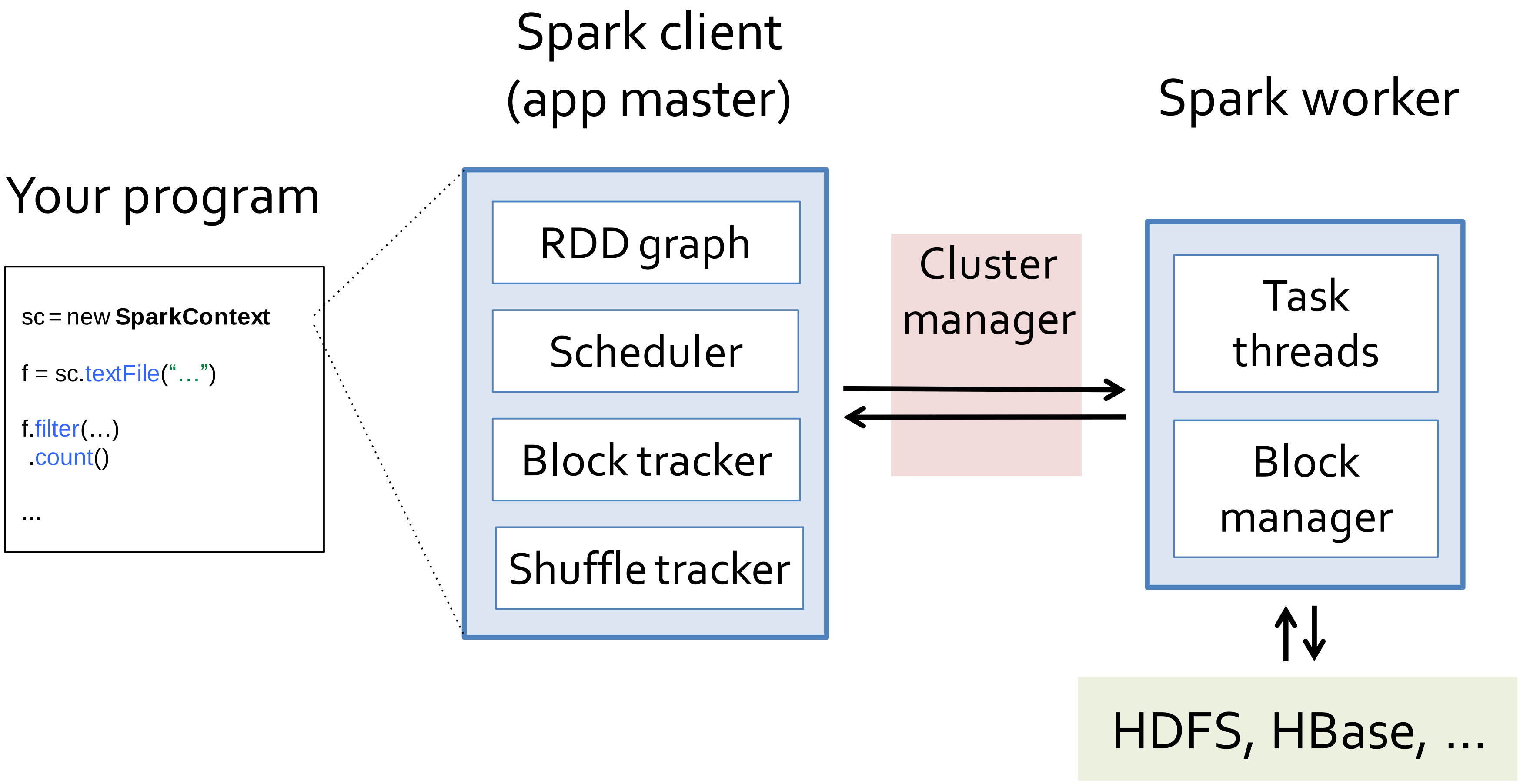
```scala
case LaunchTask(taskDesc) =>
  logInfo("Got assigned task " + taskDesc.taskId)
  if (executor == null) {
    logError("Received LaunchTask command but executor was null")
    System.exit(1)
  } else {
    executor.launchTask(this, taskDesc.taskId, taskDesc.serializedTask)
  }

case KillTask(taskId, _, interruptThread) =>
  if (executor == null) {
    logError("Received KillTask command but executor was null")
    System.exit(1)
  } else {
    executor.killTask(taskId, interruptThread)
  }
```

Scala Cookbook

- http://xerial.org/scala-cookbook

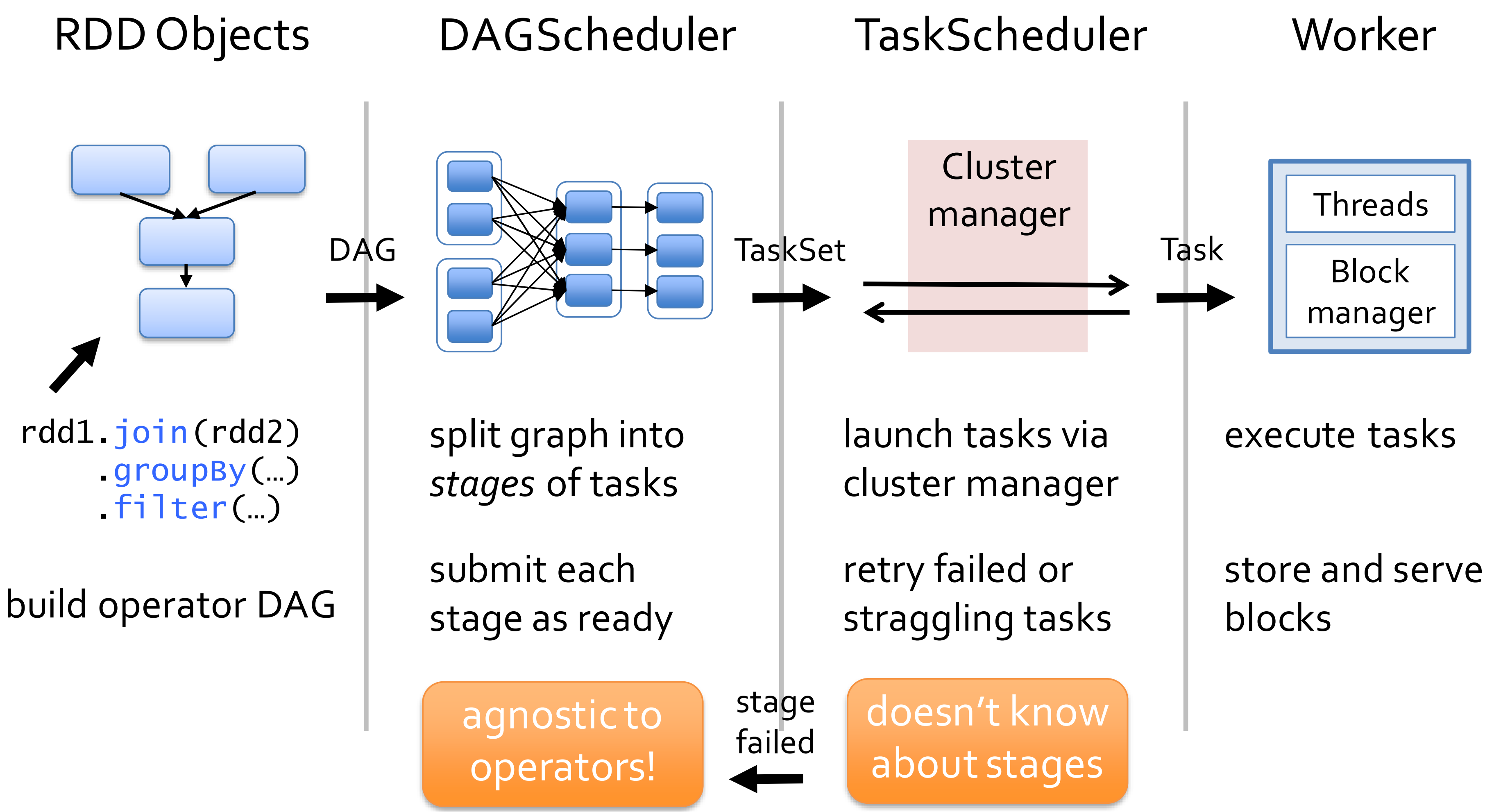
Spark Internals
https://cwiki.apache.org/confluence/display/SPARK/Spark+Internals

Components



Spark Internals
https://cwiki.apache.org/confluence/display/SPARK/Spark+Internals

Scheduling Process



RDD


```scala
abstract class RDD[T: ClassTag](
    @transient private var sc: SparkContext,
    @transient private var deps: Seq[Dependency[_]]
  ) extends Serializable with Logging {
```

- **Reference**
  - M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing, NSDI 2012, April 2012
- **SparkContext**
  - Contains SparkConfig, Scheduler, entry point of running jobs (runJobs)
- **Dependency**
  - Input RDDs

RDD.map operation

- **Map: RDD[T] -> RDD[U]**

```
/**
 * Return a new RDD by applying a function to all elements of this RDD.
 */
def map[U: ClassTag](f: T => U): RDD[U] = new MappedRDD(this, sc.clean(f))
```

- **MappedRDD**
  - For each element in a partition, apply function *f*

```
private[spark]
class MappedRDD[U: ClassTag, T: ClassTag](prev: RDD[T], f: T => U)
  extends RDD[U](prev) {

  override def getPartitions: Array[Partition] = firstParent[T].partitions

  override def compute(split: Partition, context: TaskContext) =
    firstParent[T].iterator(split, context).map(f)
}
```

RDD Iterator

- **First, check the local cache**
  - If not found, compute the 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) {
    SparkEnv.get.cacheManager.getOrCompute(this, split, context, storageLevel)
  } else {
    computeOrReadCheckpoint(split, context)
  }
}
```

- **StorageLevel**
  - Off-heap
    - Tachiyon
      - distributed memory store

```scala
class StorageLevel private(
    private var useDisk_ : Boolean,
    private var useMemory_ : Boolean,
    private var useOffHeap_ : Boolean,
    private var deserialized_ : Boolean,
    private var replication_ : Int = 1)
  extends Externalizable {
```

Task

- **DAGScheduler organizes stages**
  - Each stage has several tasks
  - Each task has preferred locations (host names)
    - Favor data local computation

```scala
/**
 * A unit of execution. We have two kinds of Task's in Spark:
 * - [[org.apache.spark.scheduler.ShuffleMapTask]]
 * - [[org.apache.spark.scheduler.ResultTask]]
 *
 * A Spark job consists of one or more stages. The very last stage in a job consists of multiple
 * ResultTasks, while earlier stages consist of ShuffleMapTasks. A ResultTask executes the task
 * and sends the task output back to the driver application. A ShuffleMapTask executes the task
 * and divides the task output to multiple buckets (based on the task's partitioner).
 *
 * @param stageId id of the stage this task belongs to
 * @param partitionId index of the number in the RDD
 */
private[spark] abstract class Task[T](val stageId: Int, var partitionId: Int) extends Serializable

  final def run(attemptId: Long): T = {
    context = new TaskContext(stageId, partitionId, attemptId, runningLocally = false)
    taskThread = Thread.currentThread()
    if (_killed) {
      kill(interruptThread = false)
    }
    runTask(context)
  }

  def runTask(context: TaskContext): T

  def preferredLocations: Seq[TaskLocation] = Nil
```

Task Locality

- **Preferred location to run a task**
  - Process, Node, Rack

```
@DeveloperApi
object TaskLocality extends Enumeration {
  // Process local is expected to be used ONLY within TaskSetManager for now.
  val PROCESS_LOCAL, NODE_LOCAL, RACK_LOCAL, ANY = Value

  type TaskLocality = Value

  def isAllowed(constraint: TaskLocality, condition: TaskLocality): Boolean = {
    condition <= constraint
  }
}
```

Delay Scheduling

- **Reference**
  - M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker and I. Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling, EuroSys 2010, April 2010.
- **Try to run tasks in the following order:**
  - Local
  - Rack local
    - Involves data serialization, local transfer
  - At any node
    - It might involve remote data transfer

```scala
/**
 * Dequeue a pending task for a given node and return its index and locality level.
 * Only search for tasks matching the given locality constraint.
 */
private def findTask(execId: String, host: String, locality: TaskLocality.Value)
  : Option[(Int, TaskLocality.Value)] =
{
  for (index <- findTaskFromList(execId, getPendingTasksForExecutor(execId))) {
    return Some((index, TaskLocality.PROCESS_LOCAL))
  }

  if (TaskLocality.isAllowed(locality, TaskLocality.NODE_LOCAL)) {
    for (index <- findTaskFromList(execId, getPendingTasksForHost(host))) {
      return Some((index, TaskLocality.NODE_LOCAL))
    }
  }

  if (TaskLocality.isAllowed(locality, TaskLocality.RACK_LOCAL)) {
    for {
      rack <- sched.getRackForHost(host)
      index <- findTaskFromList(execId, getPendingTasksForRack(rack))
    } {
      return Some((index, TaskLocality.RACK_LOCAL))
    }
  }

  // Look for no-pref tasks after rack-local tasks since they can run anywhere.
  for (index <- findTaskFromList(execId, pendingTasksWithNoPrefs)) {
    return Some((index, TaskLocality.PROCESS_LOCAL))
  }

  if (TaskLocality.isAllowed(locality, TaskLocality.ANY)) {
    for (index <- findTaskFromList(execId, allPendingTasks)) {
      return Some((index, TaskLocality.ANY))
    }
  }

  // Finally, if all else has failed, find a speculative task
  findSpeculativeTask(execId, host, locality)
}
```

Serializing Tasks

- **TaskDescription**

```
private[spark] class TaskDescription(
    val taskId: Long,
    val executorId: String,
    val name: String,
    val index: Int,    // Index within this task's TaskSet
    _serializedTask: ByteBuffer)
```

- **ResultTask**
  - RDD
  - Function
  - Stage ID, outputID
  - func
    - aggregation

```
/**
 * A task that sends back the output to the driver applicatio
 *
 * See [[org.apache.spark.scheduler.Task]] for more informati
 *
 * @param stageId id of the stage this task belongs to
 * @param rdd input to func
 * @param func a function to apply on a partition of the RDD
 * @param _partitionId index of the number in the RDD
 * @param locs preferred task execution locations for localit
 * @param outputId index of the task in this job (a job can l
 *                 input RDD's partitions).
 */
private[spark] class ResultTask[T, U](
    stageId: Int,
    var rdd: RDD[T],
    var func: (TaskContext, Iterator[T]) => U,
    _partitionId: Int,
    @transient locs: Seq[TaskLocation],
    var outputId: Int)
  extends Task[U](stageId, _partitionId) with Externalizable
```

TaskScheduler: submitTasks

- **Serialize Task Request**
  - Then, send task requests to ExecutorBackend

```
// Launch tasks returned by a set of resource offers
def launchTasks(tasks: Seq[Seq[TaskDescription]]) {
  for (task <- tasks.flatten) {
    freeCores(task.executorId) -= scheduler.CPUS_PER_TASK
    executorActor(task.executorId) ! LaunchTask(task)
  }
}
```

- **ExecutorBackend handles task requests (Akka Actor)**

```
case LaunchTask(taskDesc) =>
  logInfo("Got assigned task " + taskDesc.taskId)
  if (executor == null) {
    logError("Received LaunchTask command but executor was null")
    System.exit(1)
  } else {
    executor.launchTask(this, taskDesc.taskId, taskDesc.serializedTask)
  }
```

ClosureSerializer

- **Clean**

```
/**
 * Return a new RDD by applying a function to all elements of this RDD.
 */
def map[U: ClassTag](f: T => U): RDD[U] = new MappedRDD(this, sc.clean(f))
```

- **Function in scala: Closure**

```
val N = 100
val M = new Array[Byte](100000)
rdd.map(x => x + N)
```

  - Closure: free variable + function body (class)
    - x: bound variable, N: free variable, M:unused variable
  - class A$apply$1 extends Function1[T, U] {
    val $outer : A$outer
    def apply(T:input) : U = …
    }
  - class A$outer {
    val N = 100, val M = (large object)
    }

- **Fill M with null, then serialize the closure.**

Traversing Byte Codes

- **Closure is a class in Scala**
  - Traverse outer variable accesses
  - Using ASM4 library

```scala
private[spark]
class FieldAccessFinder(output: Map[Class[_], Set[String]]) extends ClassVisitor(ASM4) {
  override def visitMethod(access: Int, name: String, desc: String,
      sig: String, exceptions: Array[String]): MethodVisitor = {
    new MethodVisitor(ASM4) {
      override def visitFieldInsn(op: Int, owner: String, name: String, desc: String) {
        if (op == GETFIELD) {
          for (cl <- output.keys if cl.getName == owner.replace('/', '.')) {
            output(cl) += name
          }
        }
      }

      override def visitMethodInsn(op: Int, owner: String, name: String,
          desc: String) {
        // Check for calls a getter method for a variable in an interpreter wrapper object.
        // This means that the corresponding field will be accessed, so we should save it.
        if (op == INVOKEVIRTUAL && owner.endsWith("$iwC") && !name.endsWith("$outer")) {
          for (cl <- output.keys if cl.getName == owner.replace('/', '.')) {
            output(cl) += name
          }
        }
      }
    }
  }
}
```

JVM Bytecode Instructions

Objects, fields and methods

NEW *class*	...	... , new *class*
GETFIELD $c\ f\ t$	... , o	... , o.f
PUTFIELD $c\ f\ t$	... , o , v	...
GETSTATIC $c\ f\ t$	...	... , c.f
PUTSTATIC $c\ f\ t$	... , v	...
INVOKEVIRTUAL $c\ m\ t$	... , o , v_1 , ... , v_n	... , o.m(v_1, ... v_n)
INVOKESPECIAL $c\ m\ t$	... , o , v_1 , ... , v_n	... , o.m(v_1, ... v_n)
INVOKESTATIC $c\ m\ t$	... , v_1 , ... , v_n	... , c.m(v_1, ... v_n)
INVOKEINTERFACE $c\ m\ t$	... , o , v_1 , ... , v_n	... , o.m(v_1, ... v_n)
INVOKEDYNAMIC $m\ t\ bsm$	... , o , v_1 , ... , v_n	... , o.m(v_1, ... v_n)
INSTANCEOF *class*	... , o	... , o instanceof *class*
MONITORENTER	... , o	...
MONITOREXIT	... , o	...

Cache/Block Manager

- **CacheManager**
  - Stores computed RDDs to BlockManager
- **BlockManager**
  - Write-once storage
  - Manages block data according to StorageLevel
    - memoryStore
    - diskStore
    - shuffleStore
  - Serializes/deserializes block data
    - For remote data
  - Compression
    - ning LZF
    - Snappy-java
      - Faster decompression

```scala
try {
  // If we got here, we have to load the split
  logInfo("Partition %s not found, computing it".format(key))
  val computedValues = rdd.computeOrReadCheckpoint(split, context)

  // Persist the result, so long as the task is not running locally
  if (context.runningLocally) {
    return computedValues
  }

  // Keep track of blocks with updated statuses
  var updatedBlocks = Seq[(BlockId, BlockStatus)]()
  val returnValue: Iterator[T] = {
    if (storageLevel.useDisk && !storageLevel.useMemory) {
      /* In the case that this RDD is to be persisted using DISK_ONLY
       * the iterator will be passed directly to the blockManager (rather then
       * caching it to an ArrayBuffer first), then the resulting block data iterator
       * will be passed back to the user. If the iterator generates a lot of data,
       * this means that it doesn't all have to be held in memory at one time.
       * This could also apply to MEMORY_ONLY_SER storage, but we need to make sure
       * blocks aren't dropped by the block store before enabling that. */
      updatedBlocks = blockManager.put(key, computedValues, storageLevel, tellMaster = true)
      blockManager.get(key) match {
        case Some(values) =>
          values.asInstanceOf[Iterator[T]]
        case None =>
          logInfo("Failure to store %s".format(key))
          throw new Exception("Block manager failed to return persisted valued")
      }
    } else {
      // In this case the RDD is cached to an array buffer. This will save the results
      // if we're dealing with a 'one-time' iterator
      val elements = new ArrayBuffer[Any]
      elements ++= computedValues
      updatedBlocks = blockManager.put(key, elements, storageLevel, tellMaster = true)
      elements.iterator.asInstanceOf[Iterator[T]]
    }
  }

  // Update task metrics to include any blocks whose storage status is updated
  val metrics = context.taskMetrics
  metrics.updatedBlocks = Some(updatedBlocks)

  new InterruptibleIterator(context, returnValue)

} finally {
  loading.synchronized {
    loading.remove(key)
    loading.notifyAll()
  }
}
```

Storing Block Data

- **IteratorValues**
  - Raw objects
- **ArrayBufferValues**
  - Array[Byte]
- **ByteBufferValues**
  - ByteBuffer

```scala
putBlockInfo.synchronized {
  logTrace("Put for block " + blockId + " took " + Utils.getUsedTimeMs(startTimeMs)
    + " to get into synchronized block")

  var marked = false
  try {
    if (level.useMemory) {
      // Save it just to memory first, even if it also has useDisk set to true; we will
      // drop it to disk later if the memory store can't hold it.
      val res = data match {
        case IteratorValues(iterator) =>
          memoryStore.putValues(blockId, iterator, level, true)
        case ArrayBufferValues(array) =>
          memoryStore.putValues(blockId, array, level, true)
        case ByteBufferValues(bytes) =>
          bytes.rewind()
          memoryStore.putBytes(blockId, bytes, level)
      }
      size = res.size
      res.data match {
        case Right(newBytes) => bytesAfterPut = newBytes
        case Left(newIterator) => valuesAfterPut = newIterator
      }
      // Keep track of which blocks are dropped from memory
      res.droppedBlocks.foreach { block => updatedBlocks += block }
    } else if (level.useOffHeap) {
      // Save to Tachyon.
      val res = data match {
        case IteratorValues(iterator) =>
```

ConnectionManager

- **Asynchronous Data I/O server**
  - Using its own protocol
  - Send and receive block data (BufferMessage)
    - Split data into 64KB chunks
    - ChunkHeader

```scala
private[spark] class MessageChunkHeader(
    val typ: Long,
    val id: Int,
    val totalSize: Int,
    val chunkSize: Int,
    val other: Int,
    val securityNeg: Int,
    val address: InetSocketAddress) {
  lazy val buffer = {
    // No need to change this, at 'use' time, we do a reverse lookup of the hostname.
    // Refer to network.Connection
    val ip = address.getAddress.getAddress()
    val port = address.getPort()
    ByteBuffer.
      allocate(MessageChunkHeader.HEADER_SIZE).
      putLong(typ).
      putInt(id).
      putInt(totalSize).
      putInt(chunkSize).
      putInt(other).
      putInt(securityNeg).
      putInt(ip.size).
      put(ip).
      putInt(port).
      position(MessageChunkHeader.HEADER_SIZE).
      flip.asInstanceOf[ByteBuffer]
  }
```

RDD.compute

- **Local Collection**

```
private[spark] class ParallelCollectionRDD[T: ClassTag](
    @transient sc: SparkContext,
    @transient data: Seq[T],
    numSlices: Int,
    locationPrefs: Map[Int, Seq[String]])
    extends RDD[T](sc, Nil) {
  // TODO: Right now, each split sends along its full data, even if later down the RDD chain it gets
  // cached. It might be worthwhile to write the data to a file in the DFS and read it in the split
  // instead.
  // UPDATE: A parallel collection can be checkpointed to HDFS, which achieves this goal.

  override def getPartitions: Array[Partition] = {
    val slices = ParallelCollectionRDD.slice(data, numSlices).toArray
    slices.indices.map(i => new ParallelCollectionPartition(id, i, slices(i))).toArray
  }

  override def compute(s: Partition, context: TaskContext) = {
    new InterruptibleIterator(context, s.asInstanceOf[ParallelCollectionPartition[T]].iterator)
  }

  override def getPreferredLocations(s: Partition): Seq[String] = {
    locationPrefs.getOrElse(s.index, Nil)
  }
}
```

```
// If we got here, we have to load the split
logInfo("Partition %s not found, computing it".format(key))
val computedValues = rdd.computeOrReadCheckpoint(split, context)

// Persist the result, so long as the task is not running locally
if (context.runningLocally) {
  return computedValues
}
```

SparkContext - RunJob

- **RDD -> DAG Scheduler**

```scala
/**
 * Run a function on a given set of partitions in an RDD and pass the results to the given
 * handler function. This is the main entry point for all actions in Spark. The allowLocal
 * flag specifies whether the scheduler can run the computation on the driver rather than
 * shipping it out to the cluster, for short actions like first().
 */
def runJob[T, U: ClassTag](
    rdd: RDD[T],
    func: (TaskContext, Iterator[T]) => U,
    partitions: Seq[Int],
    allowLocal: Boolean,
    resultHandler: (Int, U) => Unit) {
  if (dagScheduler == null) {
    throw new SparkException("SparkContext has been shutdown")
  }
  val callSite = getCallSite
  val cleanedFunc = clean(func)
  logInfo("Starting job: " + callSite)
  val start = System.nanoTime
  dagScheduler.runJob(rdd, cleanedFunc, partitions, callSite, allowLocal,
    resultHandler, localProperties.get)
  logInfo("Job finished: " + callSite + ", took " + (System.nanoTime - start) / 1e9 + " s")
  rdd.doCheckpoint()
}
```

SparkConf

- **Key-Value configuration**
  - Master address, jar file address, environment variables, JAVA_OPTS, etc.

```scala
/**
 * Configuration for a Spark application. Used to set various Spark parameters as key-value pairs.
 *
 * Most of the time, you would create a SparkConf object with `new SparkConf()`, which will load
 * values from any `spark.*` Java system properties set in your application as well. In this case,
 * parameters you set directly on the `SparkConf` object take priority over system properties.
 *
 * For unit tests, you can also call `new SparkConf(false)` to skip loading external settings and
 * get the same configuration no matter what the system properties are.
 *
 * All setter methods in this class support chaining. For example, you can write
 * `new SparkConf().setMaster("local").setAppName("My app")`.
 *
 * Note that once a SparkConf object is passed to Spark, it is cloned and can no longer be modified
 * by the user. Spark does not support modifying the configuration at runtime.
 *
 * @param loadDefaults whether to also load values from Java system properties
 */
class SparkConf(loadDefaults: Boolean) extends Cloneable with Logging {

  /** Create a SparkConf that loads defaults from system properties and the classpath */
  def this() = this(true)

  private val settings = new HashMap[String, String]()

  if (loadDefaults) {
    // Load any spark.* system properties
    for ((k, v) <- System.getProperties.asScala if k.startsWith("spark.")) {
      settings(k) = v
    }
  }
```

SparkEnv

- **Holding spark components**

```scala
/**
 * :: DeveloperApi ::
 * Holds all the runtime environment objects for a running Spark instance (either master or worker),
 * including the serializer, Akka actor system, block manager, map output tracker, etc. Currently
 * Spark code finds the SparkEnv through a thread-local variable, so each thread that accesses these
 * objects needs to have the right SparkEnv set. You can get the current environment with
 * SparkEnv.get (e.g. after creating a SparkContext) and set it with SparkEnv.set.
 *
 * NOTE: This is not intended for external use. This is exposed for Shark and may be made private
 *       in a future release.
 */
@DeveloperApi
class SparkEnv (
    val executorId: String,
    val actorSystem: ActorSystem,
    val serializer: Serializer,
    val closureSerializer: Serializer,
    val cacheManager: CacheManager,
    val mapOutputTracker: MapOutputTracker,
    val shuffleFetcher: ShuffleFetcher,
    val broadcastManager: BroadcastManager,
    val blockManager: BlockManager,
    val connectionManager: ConnectionManager,
    val securityManager: SecurityManager,
    val httpFileServer: HttpFileServer,
    val sparkFilesDir: String,
    val metricsSystem: MetricsSystem,
    val conf: SparkConf) extends Logging {
```

SparkContext.makeRDD

- Convert local Seq[T] into RDD[T]

```scala
// Methods for creating RDDs

/** Distribute a local Scala collection to form an RDD. */
def parallelize[T: ClassTag](seq: Seq[T], numSlices: Int = defaultParallelism): RDD[T] = {
  new ParallelCollectionRDD[T](this, seq, numSlices, Map[Int, Seq[String]]())
}

/** Distribute a local Scala collection to form an RDD. */
def makeRDD[T: ClassTag](seq: Seq[T], numSlices: Int = defaultParallelism): RDD[T] = {
  parallelize(seq, numSlices)
}

/** Distribute a local Scala collection to form an RDD, with one or more
  * location preferences (hostnames of Spark nodes) for each object.
  * Create a new partition for each collection item. */
 def makeRDD[T: ClassTag](seq: Seq[(T, Seq[String])]): RDD[T] = {
  val indexToPrefs = seq.zipWithIndex.map(t => (t._2, t._1._2)).toMap
  new ParallelCollectionRDD[T](this, seq.map(_._1), seq.size, indexToPrefs)
}
```

HadoopRDD

Spark Internals

- **Reading HDFS data as (Key, Value) records**

```
/** Get an RDD for a Hadoop file with an arbitrary InputFormat
  *
  * '''Note:''' Because Hadoop's RecordReader class re-uses the same Writable object for each
  * record, directly caching the returned RDD will create many references to the same object.
  * If you plan to directly cache Hadoop writable objects, you should first copy them using
  * a `map` function.
  * */
def hadoopFile[K, V](
    path: String,
    inputFormatClass: Class[_ <: InputFormat[K, V]],
    keyClass: Class[K],
    valueClass: Class[V],
    minPartitions: Int = defaultMinPartitions
    ): RDD[(K, V)] = {
  // A Hadoop configuration can be about 10 KB, which is pretty big, so broadcast it.
  val confBroadcast = broadcast(new SerializableWritable(hadoopConfiguration))
  val setInputPathsFunc = (jobConf: JobConf) => FileInputFormat.setInputPaths(jobConf, path)
  new HadoopRDD(
    this,
    confBroadcast,
    Some(setInputPathsFunc),
    inputFormatClass,
    keyClass,
    valueClass,
    minPartitions)
}
```

Mesos Scheduler – Fine Grained

- **Mesos**
  - Offer slave resources
- **Scheduler**
  - Determine resource usage
  - Task lists are stored in TaskScheduler
  - Launches JVM for each task
    - createMesosTask
    - createExecutorInfo

```scala
/**
 * Method called by Mesos to offer resources on slaves. We respond by asking our active task sets
 * for tasks in order of priority. We fill each node with tasks in a round-robin manner so that
 * tasks are balanced across the cluster.
 */
override def resourceOffers(d: SchedulerDriver, offers: JList[Offer]) {
  val oldClassLoader = setClassLoader()
  try {
    synchronized {
      // Build a big list of the offerable workers, and remember their indices so that we can
      // figure out which Offer to reply to for each worker
      val offerableIndices = new ArrayBuffer[Int]
      val offerableWorkers = new ArrayBuffer[WorkerOffer]

      def enoughMemory(o: Offer) = {
        val mem = getResource(o.getResourcesList, "mem")
        val slaveId = o.getSlaveId.getValue
        mem >= sc.executorMemory || slaveIdsWithExecutors.contains(slaveId)
      }

      for ((offer, index) <- offers.zipWithIndex if enoughMemory(offer)) {
        offerableIndices += index
        offerableWorkers += new WorkerOffer(
          offer.getSlaveId.getValue,
          offer.getHostname,
          getResource(offer.getResourcesList, "cpus").toInt)
      }

      // Call into the TaskSchedulerImpl
      val taskLists = scheduler.resourceOffers(offerableWorkers)

      // Build a list of Mesos tasks for each slave
      val mesosTasks = offers.map(o => Collections.emptyList[MesosTaskInfo]())
      for ((taskList, index) <- taskLists.zipWithIndex) {
        if (!taskList.isEmpty) {
          val offerNum = offerableIndices(index)
          val slaveId = offers(offerNum).getSlaveId.getValue
          slaveIdsWithExecutors += slaveId
          mesosTasks(offerNum) = new JArrayList[MesosTaskInfo](taskList.size)
          for (taskDesc <- taskList) {
            taskIdToSlaveId(taskDesc.taskId) = slaveId
            mesosTasks(offerNum).add(createMesosTask(taskDesc, slaveId))
          }
        }
      }

      // Reply to the offers
      val filters = Filters.newBuilder().setRefuseSeconds(1).build() // TODO: lower timeout?
      for (i <- 0 until offers.size) {
        d.launchTasks(offers(i).getId, mesosTasks(i), filters)
      }
    }
  } finally {
    restoreClassLoader(oldClassLoader)
  }
}
```

Mesos Fine-Grained Executor


```scala
def createExecutorInfo(execId: String): ExecutorInfo = {
  val sparkHome = sc.getSparkHome().getOrElse(throw new SparkException(
    "Spark home is not set; set it through the spark.home system " +
    "property, the SPARK_HOME environment variable or the SparkContext constructor"))
  val environment = Environment.newBuilder()
  sc.executorEnvs.foreach { case (key, value) =>
    environment.addVariables(Environment.Variable.newBuilder()
      .setName(key)
      .setValue(value)
      .build())
  }
  val command = CommandInfo.newBuilder()
    .setEnvironment(environment)
  val uri = sc.conf.get("spark.executor.uri", null)
  if (uri == null) {
    command.setValue(new File(sparkHome, "/sbin/spark-executor").getCanonicalPath)
  } else {
    // Grab everything to the first '.'. We'll use that and '*' to
    // glob the directory "correctly".
    val basename = uri.split('/').last.split('.').head
    command.setValue("cd %s*; ./sbin/spark-executor".format(basename))
    command.addUris(CommandInfo.URI.newBuilder().setValue(uri))
  }
  val memory = Resource.newBuilder()
    .setName("mem")
    .setType(Value.Type.SCALAR)
    .setScalar(Value.Scalar.newBuilder().setValue(sc.executorMemory).build())
    .build()
  ExecutorInfo.newBuilder()
    .setExecutorId(ExecutorID.newBuilder().setValue(execId).build())
    .setCommand(command)
    .setData(ByteString.copyFrom(createExecArg()))
    .addResources(memory)
    .build()
}
```

Mesos Fine-Grained Executor

- **spark-executor**
  - Shell script for launching JVM

```
FWDIR="$(cd `dirname $0`/..; pwd)"

export PYTHONPATH=$FWDIR/python:$PYTHONPATH
export PYTHONPATH=$FWDIR/python/lib/py4j-0.8.1-src.zip:$PYTHONPATH

echo "Running spark-executor with framework dir = $FWDIR"
exec $FWDIR/bin/spark-class org.apache.spark.executor.MesosExecutorBackend
```

Coarse-grained Mesos Scheduler

- **Launches Spark executor on Mesos slave**
  - Runs CoarseGrainedExecutorBackend

```scala
/**
 * Method called by Mesos to offer resources on slaves. We respond by launching an executor,
 * unless we've already launched more than we wanted to.
 */
override def resourceOffers(d: SchedulerDriver, offers: JList[Offer]) {
  synchronized {
    val filters = Filters.newBuilder().setRefuseSeconds(-1).build()

    for (offer <- offers) {
      val slaveId = offer.getSlaveId.toString
      val mem = getResource(offer.getResourcesList, "mem")
      val cpus = getResource(offer.getResourcesList, "cpus").toInt
      if (totalCoresAcquired < maxCores && mem >= sc.executorMemory && cpus >= 1 &&
          failuresBySlaveId.getOrElse(slaveId, 0) < MAX_SLAVE_FAILURES &&
          !slaveIdsWithExecutors.contains(slaveId)) {
        // Launch an executor on the slave
        val cpusToUse = math.min(cpus, maxCores - totalCoresAcquired)
        totalCoresAcquired += cpusToUse
        val taskId = newMesosTaskId()
        taskIdToSlaveId(taskId) = slaveId
        slaveIdsWithExecutors += slaveId
        coresByTaskId(taskId) = cpusToUse
        val task = MesosTaskInfo.newBuilder()
          .setTaskId(TaskID.newBuilder().setValue(taskId.toString).build())
          .setSlaveId(offer.getSlaveId)
          .setCommand(createCommand(offer, cpusToUse + extraCoresPerSlave))
          .setName("Task " + taskId)
          .addResources(createResource("cpus", cpusToUse))
          .addResources(createResource("mem", sc.executorMemory))
          .build()
        d.launchTasks(offer.getId, Collections.singletonList(task), filters)
      } else {
        // Filter it out
        d.launchTasks(offer.getId, Collections.emptyList[MesosTaskInfo](), filters)
      }
    }
  }
}
```

Coarse-grained ExecutorBackend

- **Akka Actor**
- **Register itself to the master**
- **Initialize the executor after response**

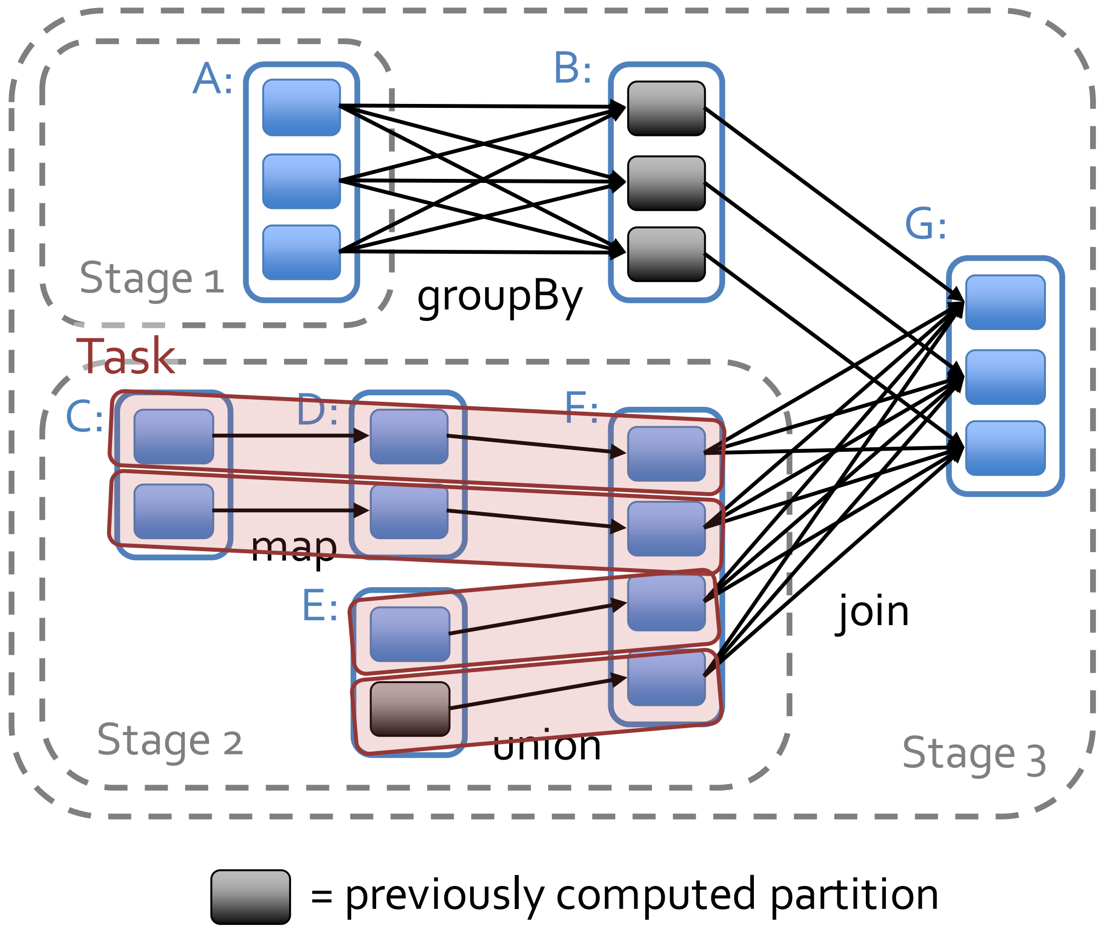
```scala
private[spark] class CoarseGrainedExecutorBackend(
    driverUrl: String,
    executorId: String,
    hostPort: String,
    cores: Int)
  extends Actor
  with ExecutorBackend
  with Logging {

  Utils.checkHostPort(hostPort, "Expected hostport")

  var executor: Executor = null
  var driver: ActorSelection = null

  override def preStart() {
    logInfo("Connecting to driver: " + driverUrl)
    driver = context.actorSelection(driverUrl)
    driver ! RegisterExecutor(executorId, hostPort, cores)
    context.system.eventStream.subscribe(self, classOf[RemotingLifecycleEvent])
  }

  override def receive = {
    case RegisteredExecutor(sparkProperties) =>
      logInfo("Successfully registered with driver")
      // Make this host instead of hostPort ?
      executor = new Executor(executorId, Utils.parseHostPort(hostPort)._1, sparkProperties,
        false)

    case RegisterExecutorFailed(message) =>
      logError("Slave registration failed: " + message)
      System.exit(1)

    case LaunchTask(taskDesc) =>
      logInfo("Got assigned task " + taskDesc.taskId)
      if (executor == null) {
        logError("Received LaunchTask command but executor was null")
        System.exit(1)
      } else {
        executor.launchTask(this, taskDesc.taskId, taskDesc.serializedTask)
      }
```

Scheduler Optimizations

Pipelines narrow ops. within a stage

Picks join algorithms based on partitioning (minimize shuffles)

Reuses previously cached data



- **ReferenceQueue**
  - Notified when weakly referenced objects are garbage collected.

```
/**
 * An asynchronous cleaner for RDD, shuffle, and broadcast state.
 *
 * This maintains a weak reference for each RDD, ShuffleDependency, and Broadcast of ir
 * to be processed when the associated object goes out of scope of the application. Act
 * cleanup is performed in a separate daemon thread.
 */
private[spark] class ContextCleaner(sc: SparkContext) extends Logging {

  private val referenceBuffer = new ArrayBuffer[CleanupTaskWeakReference]
    with SynchronizedBuffer[CleanupTaskWeakReference]

  private val referenceQueue = new ReferenceQueue[AnyRef]

  private val listeners = new ArrayBuffer[CleanerListener]
    with SynchronizedBuffer[CleanerListener]

  private val cleaningThread = new Thread() { override def run() { keepCleaning() }}
```

WE ARE HIRING!