Apache Spark Internals

Pietro Michiardi

Eurecom

Acknowledgments & Sources

- **Sources**
  - Research papers:
    - `https://spark.apache.org/research.html`
  - Blogs:
    - `https://trongkhoanguyenblog.wordpress.com/`
    - `http://hxquangnhat.com/`
- **Contributors**
  - Quang-Nhat Hoang-Xuan
  - Khoa Nguyen Trong

Introduction and Motivations

What is Apache Spark

- **Project goals**
  - Generality: diverse workloads, operators, job sizes
  - Low latency: sub-second
  - Fault tolerance: faults are the norm, not the exception
  - Simplicity: often comes from generality

Motivations

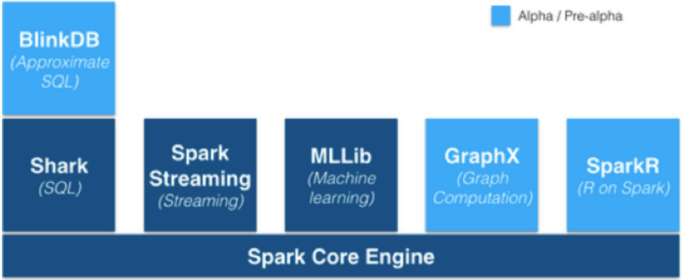
- **Software engineering point of view**
  - Hadoop code base is huge
  - Contributions/Extensions to Hadoop are cumbersome
  - Java-only hinders wide adoption, but Java support is fundamental
- **System/Framework point of view**
  - Unified pipeline
  - Simplified data flow
  - Faster processing speed
- **Data abstraction point of view**
  - New fundamental abstraction RDD
  - Easy to extend with new operators
  - More descriptive computing model

Hadoop: No Unified Vision

General Batching	Specialized systems			
	Streaming	Iterative	Ad-hoc / SQL	Graph
MapReduce	Storm	Mahout	Pig	Giraph
	S4		Hive	
	Samza		Drill	
			Impala	

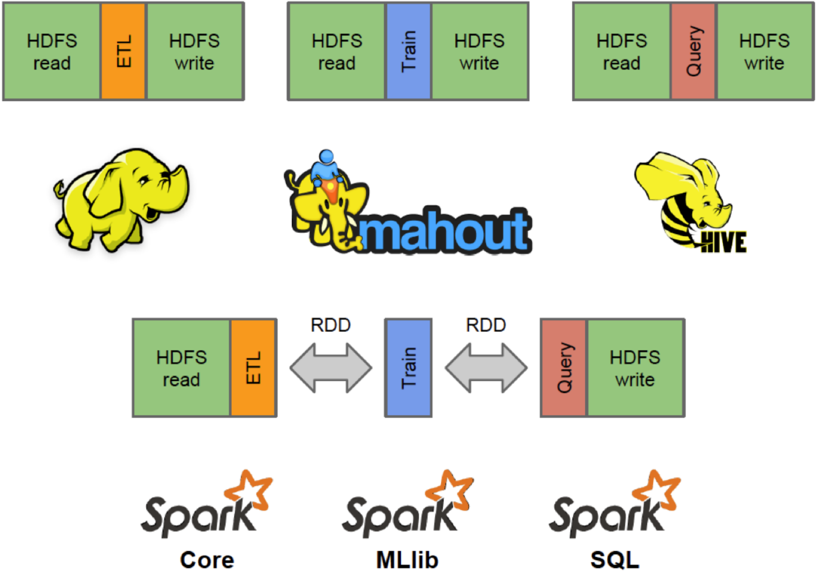
- Sparse modules
- Diversity of APIs
- Higher operational costs

SPARK: A Unified Pipeline



- Spark Streaming (stream processing)
- GraphX (graph processing)
- MLLib (machine learning library)
- Spark SQL (SQL on Spark)

A Simplified Data Flow



Hadoop: Bloated Computing Model

```java
public class WordCount {

 public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
        }
    }
 }

 public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        context.write(key, new IntWritable(sum));
    }
 }

 public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);

    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.waitForCompletion(true);
 }
```

SPARK: Descriptive Computing Model

```scala
val file = sc.textFile("hdfs://...")

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

counts.saveAsTextFile("hdfs://...")
```

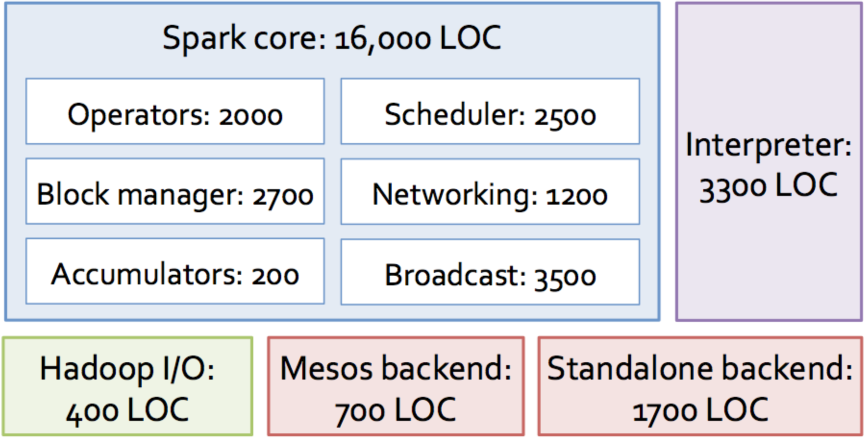
- Organize computation into multiple stages in a processing pipeline
  - **Transformations** apply user code to distributed data in parallel
  - **Actions** assemble final output of an algorithm, from distributed data

Faster Processing Speed

- **Let's focus on iterative algorithms**
  - Spark is faster thanks to the simplified data flow
  - We avoid materializing data on HDFS after each iteration

- **Example: k-means algorithm, 1 iteration**
  - `HDFS Read`
  - **`Map`**(*Assign sample to closest centroid*)
  - **`GroupBy`**(Centroid_ID)
  - `NETWORK Shuffle`
  - **`Reduce`**(*Compute new centroids*)
  - `HDFS Write`

Code Base (2012)



- 2012 (version 0.6.x): 20,000 lines of code
- 2014 (branch-1.0): 50,000 lines of code

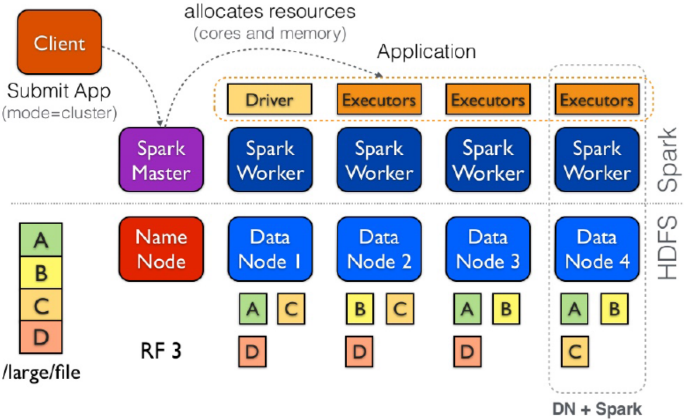
Anatomy of a Spark Application

A Very Simple Application Example

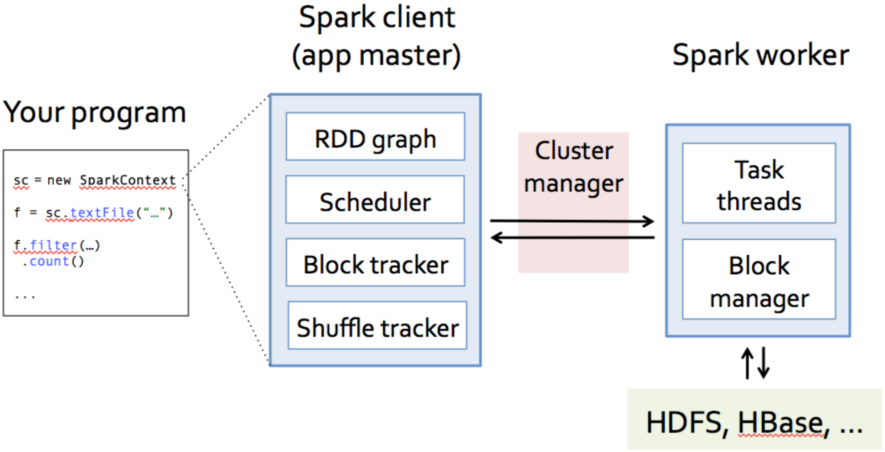
```scala
val sc = new SparkContext("spark://...", "MyJob", home,
    jars)

val file = sc.textFile("hdfs://...") // This is an RDD

val errors = file.filter(_.contains("ERROR")) // This is
    an RDD

errors.cache()

errors.count() // This is an action
```

Spark Applications: The Big Picture

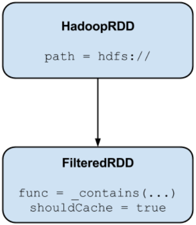
- There are two ways to manipulate data in Spark
  - Use the interactive shell, *i.e.*, the REPL
  - Write standalone applications, *i.e.*, driver programs

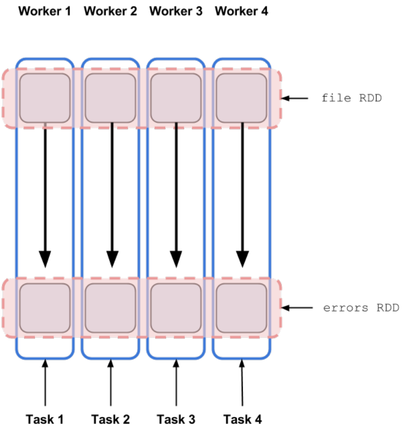


Spark Components: details



The RDD graph: dataset vs. partition views



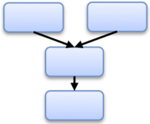


Data Locality

- **Data locality principle**
  - Same as for Hadoop MapReduce
  - Avoid network I/O, workers should manage local data
- **Data locality and caching**
  - First run: data not in cache, so use HadoopRDD's locality prefs (from HDFS)
  - Second run: FilteredRDD is in cache, so use its locations
  - If something falls out of cache, go back to HDFS

Lifetime of a Job in Spark

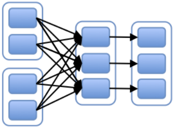
RDD Objects



```
rdd1.join(rdd2)
.groupBy(...)
.filter(...)
```

Build the operator DAG

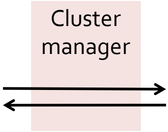
DAG Scheduler



Split the DAG into *stages* of *tasks*

Submit each stage and its tasks as ready

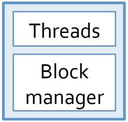
Task Scheduler



Launch tasks via Master

Retry failed and straggler tasks

Worker



Execute tasks

Store and serve blocks

In Summary

- **Our example Application**: a **jar** file
  - Creates a `SparkContext`, which is the core component of the driver
  - Creates an input RDD, from a file in HDFS
  - Manipulates the input RDD by applying a `filter(f: T => Boolean)` transformation
  - Invokes the action `count()` on the transformed RDD
- **The DAG Scheduler**
  - Gets: RDDs, functions to run on each partition and a listener for results
  - Builds *Stages* of *Tasks* objects (code + preferred location)
  - Submits Tasks to the **Task Scheduler** as ready
  - Resubmits failed *Stages*
- **The Task Scheduler**
  - Launches *Tasks* on executors
  - Relaunches failed *Tasks*
  - Reports to the DAG Scheduler

Resilient Distributed Datasets

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,
USENIX Symposium on Networked Systems Design and Implementation, 2012

What is an RDD

- **RDD are partitioned, locality aware, distributed collections**
  - RDD are *immutable*
- **RDD are data structures that:**
  - Either point to a direct data source (e.g. HDFS)
  - Apply some transformations to its parent RDD(s) to generate new data elements
- **Computations on RDDs**
  - Represented by *lazily evaluated* lineage *DAGs* composed by chained RDDs

RDD Abstraction

- **Overall objective**
  - Support a wide array of operators (more than just `Map` and `Reduce`)
  - Allow arbitrary composition of such operators
- **Simplify scheduling**
  - Avoid to modify the scheduler for each operator

→ The question is: *How to capture dependencies in a general way?*

RDD Interfaces

- **Set of partitions ("splits")**
  - Much like in Hadoop MapReduce, each RDD is associated to (input) partitions
- **List of dependencies on parent RDDs**
  - This is completely new w.r.t. Hadoop MapReduce
- **Function to compute a partition given parents**
  - This is actually the "user-defined code" we referred to when discussing about the `Mapper` and `Reducer` classes in Hadoop
- **Optional preferred locations**
  - This is to enforce data locality
- **Optional partitioning info (Partitioner)**
  - This really helps in some "advanced" scenarios in which you want to pay attention to the behavior of the shuffle mechanism

Hadoop RDD

- partitions = one per HDFS block
- dependencies = none
- compute(partition) = read corresponding block
- preferredLocations(part) = HDFS block location
- partitioner = none

Filtered RDD

- partitions = same as parent RDD
- dependencies = *one-to-one* on parent
- compute(partition) = compute parent and filter it
- preferredLocations(part) = none (*ask parent*)
- partitioner = none

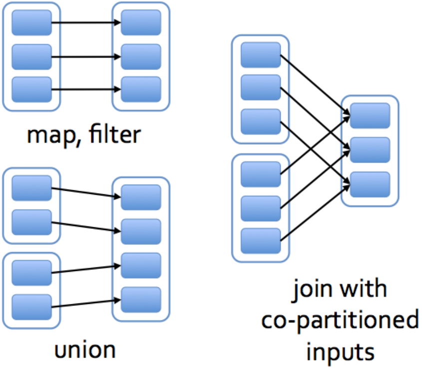
Joined RDD

- partitions = one per reduce task
- dependencies = *shuffle* on *each* parent
- compute(partition) = read and join shuffled data
- preferredLocations(part) = none
- partitioner = HashPartitioner(numTask)[1]

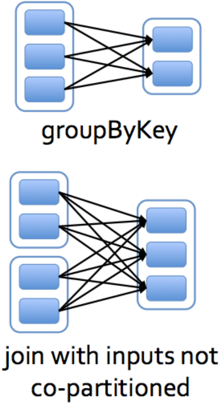
[1] Spark knows this data is hashed.

Dependency Types (1)

Narrow dependencies



Wide dependencies



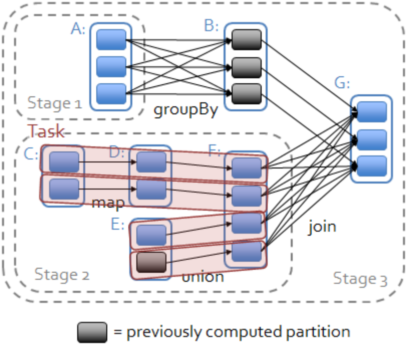
Dependency Types (2)

- **Narrow dependencies**
  - Each partition of the parent RDD is used by at most one partition of the child RDD
  - Task can be executed locally and we don't have to shuffle. (Eg: `map`, `flatMap`, `filter`, `sample`)

- **Wide Dependencies**
  - Multiple child partitions may depend on one partition of the parent RDD
  - This means we have to shuffle data **unless the parents are hash-partitioned** (Eg: `sortByKey`, `reduceByKey`, `groupByKey`, `cogroupByKey`, `join`, `cartesian`)

Dependency Types: Optimizations

- **Benefits of Lazy evaluation**
  - The DAG Scheduler optimizes *Stages* and *Tasks* before submitting them to the Task Scheduler
  - **Piplining** narrow dependencies within a Stage
  - **Join plan selection** based on partitioning
  - **Cache reuse**



Operations on RDDs: Transformations

- **Transformations**
  - Set of operations on a RDD that define how they should be transformed
  - As in relational algebra, the application of a transformation to an RDD yields a new RDD (because RDD are *immutable*)
  - Transformations are lazily evaluated, which allow for optimizations to take place before execution

- **Examples** (not exhaustive)
  - `map(func)`, `flatMap(func)`, `filter(func)`
  - `grouByKey()`
  - `reduceByKey(func)`, `mapValues(func)`, `distinct()`, `sortByKey(func)`
  - `join(other)`, `union(other)`
  - `sample()`

Operations on RDDs: Actions

- **Actions**
  - Apply transformation chains on RDDs, eventually performing some additional operations (e.g., counting)
  - Some actions only store data to an external data source (e.g. HDFS), others fetch data from the RDD (and its transformation chain) upon which the action is applied, and convey it to the driver
- **Examples** (not exhaustive)
  - `reduce(func)`
  - `collect(), first(), take(), foreach(func)`
  - `count(), countByKey()`
  - `saveAsTextFile()`

Operations on RDDs: Final Notes

- **Look at return types!**
  - Return type: RDD $\rightarrow$ transformation
  - Return type: built-in scala/java types such as `int`, `long`, `List<Object>`, `Array<Object>` $\rightarrow$ action
- **Caching is a transformation**
  - Hints to keep RDD in memory after its first evaluation
- **Transformations depend on RDD "flavor"**
  - `PairRDD`
  - `SchemaRDD`

RDD Code Snippet

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

- **SparkContext**
  - This is the main entity responsible for setting up a job
  - Contains SparkConfig, Scheduler, entry point of running jobs (runJobs)
- **Dependencies**
  - Input RDD(s)

RDD.map operation Snippet

- **Map: RDD[T] $\rightarrow$ 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 Code Snipped

- **Method to go through an RDD and apply function *f***
  - First, check local cache
  - If not found, compute the RDD

```
/**
 * 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)
  }
}
```

- **Storage Levels**
  - Disk
  - Memory
  - Off Heap (e.g. external memory stores like Tachyon)
  - De-serialized

Making RDD from local collections

- **Convert a local (on the driver) 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)
}
```

Hadoop RDD Code Snippet

- **Reading HDFS data as <key, value> records**

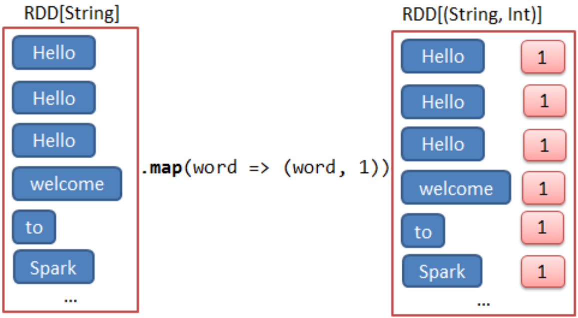
```scala
/** Get an RDD for a Hadoop file with an arbitrary InputFormat
  *
  * '''Note:''' Because Hadoop's RecordReader class re-uses the same Writable object for each
  * record, directly caching the returned RDD will create many references to the same object.
  * If you plan to directly cache Hadoop 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)
}
```

Understanding RDD Operations

Common Transformations

`map(f: T => U)`

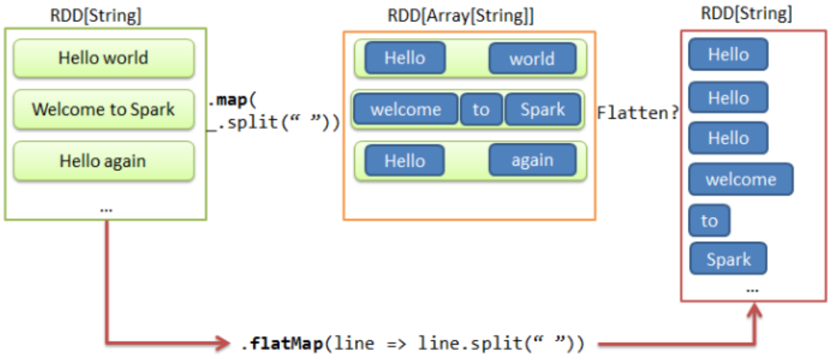
Returns a `MappedRDD[U]` by applying *f* to each element



Common Transformations

`flatMap(f: T => TraversableOnce[U])`

Returns a `FlatMappedRDD[U]` by first applying *f* to each element, then flattening the results



Detailed Example: Word Count

Spark Word Count: the driver

```
import org.apache.spark.SparkContext

import org.apache.spark.SparkContext._

val sc = new SparkContext("spark://...", "MyJob", "spark
    home", "additional jars")
```

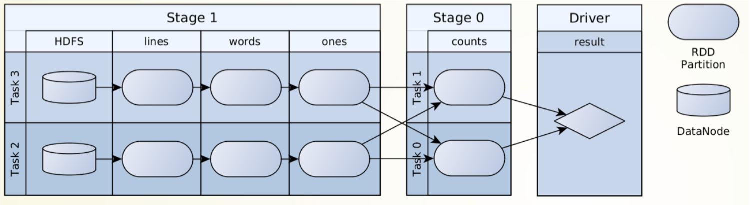
- **Driver and `SparkContext`**
  - A SparkContext initializes the application driver, the latter then registers the application to the cluster manager, and gets a list of executors
  - Then, the driver takes full control of the Spark job

Spark Word Count: the code

```
val lines = sc.textFile("input")
val words = lines.flatMap(_.split(" "))
val ones = words.map(_ -> 1)
val counts = ones.reduceByKey(_ + _)
val result = counts.collectAsMap()
```

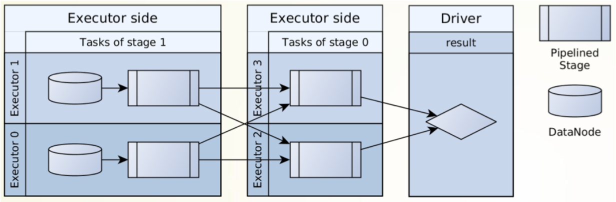
- **RDD lineage DAG is built on driver side with**
  - Data source RDD(s)
  - Transformation RDD(s), which are created by transformations
- **Job submission**
  - An *action* triggers the DAG scheduler to submit a job

Spark Word Count: the DAG



- **Directed Acyclic Graph**
  - Built from the RDD lineage
- **DAG scheduler**
  - Transforms the DAG into stages and turns each partition of a stage into a single task
  - Decides what to run

Spark Word Count: the execution plan



- **Spark Tasks**
  - Serialized RDD lineage DAG + closures of transformations
  - Run by Spark executors

- **Task scheduling**
  - The driver side task scheduler launches tasks on executors according to resource and locality constraints
  - The task scheduler decides where to run tasks

Spark Word Count: the Shuffle phase

```
val lines = sc.textFile("input")
val words = lines.flatMap(_.split(" "))
val ones = words.map(_ -> 1)
val counts = ones.reduceByKey(_ + _)
val result = counts.collectAsMap()
```

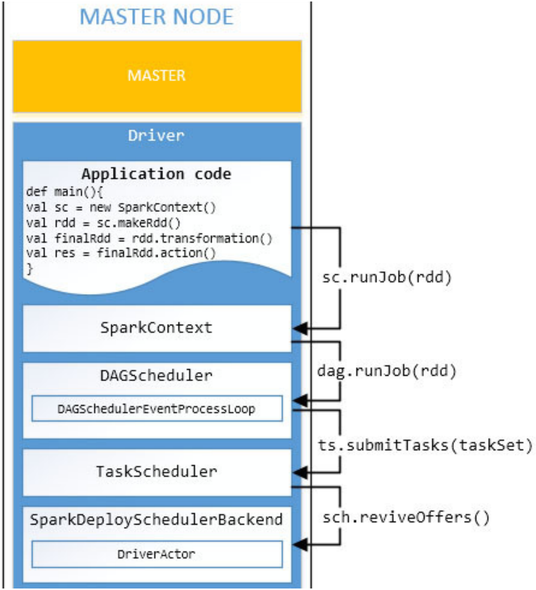
- **reduceByKey transformation**
  - Induces the shuffle phase
  - In particular, we have a *wide dependency*
  - Like in Hadoop MapReduce, intermediate <key,value> pairs are stored on the local file system
- **Automatic combiners!**
  - The `reduceByKey` transformation implements map-side combiners to pre-aggregate data

Resource Allocation: Spark Schedulers

Spark Schedulers

- **Two main scheduler components, executed by the driver**
  - The DAG scheduler
  - The Task scheduler
- **Objectives**
  - Gain a broad understanding of how Spark submits Applications and Jobs
  - Understand how *Stages* and *Tasks* are built, and their optimization
  - Understand interaction among various other Spark components

Submitting a Spark Application: A Walk Through



Submitting a Spark Application: Details

```
- finalRdd.action()
-- sc.runJob()
--- dagScheduler.runJob()
---- dagScheduler.submitJob()
----- DAGSchedulerEventProcessLoop.post(JobSubmitted)
```

```
- DAGSchedulerEventProcessLoop.onRecieve(JobSubmitted)
-- dagScheduler.handleJobSubmitted()
--- finalStage = new Stage()
--- submitStage(finalStage)
---- missingStages = getMissingParentStages(stage).sortById()
---- if missingStages != Nil
     foreach parent <- missingStages: submitStage(parent)
---- else
     dagScheduler.submitMissingTasks(stage)
```

```
def getMissingParentStages(stage){
     case ShuffleDependency: missing += new Stage()
     case NarrowDependency: waiting4Visit.push()
}
```

```
def submitMissingTasks(stage){
     val loc = getPreferredLocs(stage.rdd)
     if stage.isShuffleMap
             tasks:Seq[Task] = partitions.foreach(yield new ShuffleMapTask(p, loc))
     else
             tasks:Seq[Task] = partitions.foreach(yield new ResultTask())

     taskScheduler.submitTasks(new TaskSet(tasks))
}
```

TaskSchedulerimpl.scala

```
def submitTasks(taskSet){
     schedulableBuilder.addTaskSetManager(new TaskSetManager(taskSet))
     if (!runInLocal && !hasReceivedTask){
             starvationTimer.scheduleAtFixedRate(new TimerTask(){…, STARVA
     }
     schedulerBackend.reviveOffers()
}
```

```
def sparkDeploySchedulerBackend.reviveOffers(){
     driverActor ! ReviveOffers
}
```

CoarseGrainedSchedulerBackend.scala

```
def receiveWithLogging(){
     case ReviveOffers => makeOffers(){
             launchTasks(tasks){
                     foreach task <- tasks:
                             executorActor(task.executorId) ! LaunchTask(t
             }
     }
}
```

The DAG Scheduler

- **Stage-oriented scheduling**
  - Computes a DAG of stages for each job in the application
    `Lines 10-14`, details in `Lines 15-27`
  - Keeps track of which RDD and stage output are materialized
  - Determines an optimal schedule, minimizing stages
  - Submit stages as sets of Tasks (`TaskSets`) to the Task scheduler
    `Line 26`
- **Data locality principle**
  - Uses "preferred location" information (optionally) attached to each RDD
    `Line 20`
  - Package this information into Tasks and send it to the Task scheduler
- **Manages Stage failures**
  - Failure type: (intermediate) data loss of shuffle output files
  - Failed stages will be resubmitted
  - NOTE: Task failures are handled by the Task scheduler, which simply resubmit them if they can be computed with no dependency on previous output

The DAG Scheduler: Implementation Details

- **Implemented as an event queue**
  - Uses a daemon thread to handle various kinds of events `Line 6`
  - `JobSubmitted`, `JobCancelled`, `CompletionEvent`
  - The thread "swipes" the queue, and routes event to the corresponding handlers
- **What happens when a job is submitted to the DAGScheduler?**
  - `JobWaiter` object is created
  - `JobSubmitted` event is fired
  - The daemon thread blocks and wait for a job result `Lines 3,4`

The DAG Scheduler: Implementation Details (2)

- **Who handles the `JobSubmitted` event?**
  - Specific handler called `handleJobSubmitted`
    `Line 6`
- **Walk-through to the Job Submitted handler**
  - Create a new job, called `ActiveJob`
  - New job starts with only 1 stage, corresponding to the last stage of the job upon which an action is called
    `Lines 8-9`
  - Use the dependency information to produce additional stages
    - Shuffle Dependency: create a new map stage
      `Line 16`
    - Narrow Dependency: pipes them into a single stage
      `getMissingParentStages`

More About Stages

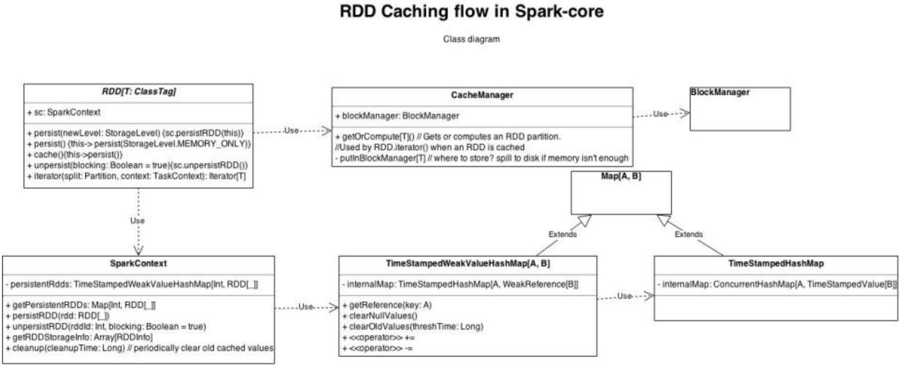
Data Shuffling

Caching

Spark's Storage Module

- **The storage module**
  - Access (I/O) "external" data sources: HDFS, Local Disk, RAM, remote data access through the network
  - Caches RDDs using a variety of "storage levels"
- **Main components**
  - The Cache Manager: uses the Block Manager to perform caching
  - The Block Manager: distributed key/value store

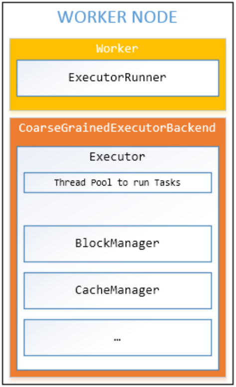
Class Diagram of the Caching Component



How Caching Works

- **Frequently used RDD can be stored in memory**
  - Deciding which RDD to cache is an art!
  - One method, one short-cut: `persist()`, `cache()`
- **`SparkContext` keeps track of cached RDD**
  - Uses a data-structed called `persistentRDD`
  - Maintains references to cached RDD, and eventually call the garbage collector
  - Time-stamp based invalidation using `TimeStampedWeakValueHashMap[A, B]`

How Caching Works




```
- rdd.iterator(split:Partition)
-- cacheManager.getOrCompute(rdd, partition)
--- val key = RDDBlockId(rdd.id, partition.index)
--- switch(blockManager.get(key))
---     case Some: return
---     case None:
                val computedValue = rdd.computeOrReadCheckPoint(partition)
                if isRunningLocally
                        return computedValue
                else
                        cachedValue = putInBlockManager(key, computedValue, storageLevel)
```

```
BlockManager.scala

// write-once key-value
// serves both cachedRdds and shuffle data
// tracks storage level of each block
// can drop data on disk if RAM mem is low
// can replicate data across nodes
def get(blockId){
        tryGetLocal()
        tryGetRemote()
        if no, return None
}
```

The Block Manager

- **"Write-once" key-value store**
  - One node per worker
  - No updates, data is immutable
- **Main tasks**
  - Serves shuffle data (local or remote connections) and cached RDDs
  - Tracks the "Storage Level" (RAM, disk) for each block
  - Spills data to disk if memory is insufficient
  - Handles data replication, if required

Storage Levels

- **The Block Manager can hold data in various storage tiers**
  - `org.apache.spark.storage.StorageLevel` contains flags to indicate which tier to use
  - Manual configuration, in the application
  - Deciding the storage level to use for RDDs is not trivial

- **Available storage tiers**
  - RAM (default option): if the the RDD doesn't fit in memory, some partitions will not be cached (will be re-computed when needed)
  - Tachyon (off java heap): reduces garbage collection overhead, the crash of an executor no longer leads to cached data loss
  - Disk

- **Data format**
  - Serialized or as Java objects
  - Replicated partitions

Spark Tuning with a Running Example