JOBS ➔ STAGES ➔ TASKS
The key to tuning spark apps is a sound grasp of Spark's internal mechanisms.
UNITS OF PHYSICAL EXECUTION

Job #1

Stage 1
Stage 2
Stage 3
Stage 4
Stage 5

Task #1
Task #2
Task #3

Job #2
SCHEDULING PROCESS

RDD Objects

DAG Objects

DAG Scheduler

- Build operator DAG

- Split graph into stages of tasks

- Submit each stage as ready

Task Scheduler

- Launches individual tasks

- Retry failed or straggling tasks

Executor

- Execute tasks

- Store and serve blocks

Agnostic to operators Doesn’t know about stages

Stage failed
How does a user program get translated into units of physical execution?

application -> jobs, stages, tasks
// Read input file
val input = sc.textFile("input.txt")

val tokenized = input
  .map(line => line.split(" "))
  .filter(words => words.size > 0) // remove empty lines

val counts = tokenized  // frequency of log levels
  .map(words => (words(0), 1))
  .reduceByKey{ (a, b) => a + b, 2 }
// Read input file
val input = sc.textFile( )

tokenized = input
.map( )
.filter( ) // remove empty lines

val counts = tokenized // frequency of log levels
.map( )
.reduceByKey{ }
TRANSFORMATIONS

sc.textFile().map().filter().map().reduceByKey()
DAG VIEW OF RDDS

textFile()  map()  filter()  map()  reduceByKey()

Hadoop RDD
- Partition 1
- Partition 2
- Partition 3

Mapped RDD
- Partition 1
- Partition 2
- Partition 3

Filtered RDD
- Partition 1
- Partition 2
- Partition 3

Mapped RDD
- Partition 1
- Partition 2
- Partition 3

Shuffle RDD
- Partition 1
- Partition 2

input  tokenized  counts
DAG’s are materialized through a method `sc.runJob`:

```python
def runJob[T, U](
    rdd: RDD[T],
    partitions: Seq[Int],
    func: (Iterator[T]) => U)
: Array[U]
```

1. RDD to compute
2. Which partitions
3. Fn to produce results
   → results for each part.
HOW RUNJOB WORKS

Needs to compute my parents, parents, parents, etc all the way back to an RDD with no dependencies (e.g. HadoopRDD).
HOW RUNJOB WORKS

Needs to compute my parents, parents, parents, etc all the way back to an RDD with no dependencies (e.g. HadoopRDD).
STAGE GRAPH

Each task will:

1) Read Hadoop input
2) Perform maps & filters
3) Write partial sums

Each task will:

1) Read partial sums
2) Invoke user function passed to runJob
**narrow**

- each partition of the parent RDD is used by at most one partition of the child RDD

- map, filter

- union

- join w/ inputs co-partitioned

**wide**

- multiple child RDD partitions may depend on a single parent RDD partition

- groupByKey

- join w/ inputs not co-partitioned
scala> counts.toDebugString
res84: String =
(2) ShuffledRDD[296] at reduceByKey at <console>:17
+- (3) MappedRDD[295] at map at <console>:17
  | FilteredRDD[294] at filter at <console>:15
  | MappedRDD[293] at map at <console>:15
  | input.text MappedRDD[292] at textFile at <console>:13
  | input.text HadoopRDD[291] at textFile at <console>:13

(indentations indicate a shuffle boundary)
“This distinction is useful for two reasons:

1) Narrow dependencies allow for pipelined execution on one cluster node, which can compute all the parent partitions. For example, one can apply a map followed by a filter on an element-by-element basis.

In contrast, wide dependencies require data from all parent partitions to be available and to be shuffled across the nodes using a MapReduce-like operation.

2) Recovery after a node failure is more efficient with a narrow dependency, as only the lost parent partitions need to be recomputed, and they can be recomputed in parallel on different nodes. In contrast, in a lineage graph with wide dependencies, a single failed node might cause the loss of some partition from all the ancestors of an RDD, requiring a complete re-execution."
How many Stages will this DAG require?
How many Stages will this DAG require?

Source: Cloudera
PUTTING IT ALL TOGETHER

### Details for Job 0

**Status:** SUCCEEDED  
**Completed Stages:** 2

#### Completed Stages (2)

<table>
<thead>
<tr>
<th>Stage Id</th>
<th>Description</th>
<th>Submitted</th>
<th>Duration</th>
<th>Tasks: Succeeded/Total</th>
<th>Input</th>
<th>Output</th>
<th>Shuffle Read</th>
<th>Shuffle Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>count at &lt;console&gt;</td>
<td>2015/02/19 21:35:56</td>
<td>38 ms</td>
<td>2/2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>map at &lt;console&gt;</td>
<td>2015/02/19 21:35:56</td>
<td>0.1 s</td>
<td>2/2</td>
<td>72.0 B</td>
<td>354.0 B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Named after action calling `runJob`

Named after last RDD in pipeline
How do you know if a shuffle will be called on a Transformation?

- `repartition`, `join`, `cogroup`, and any of the `*By` or `*ByKey` transformations can result in shuffles

- If you declare a `numPartitions` parameter, it’ll probably shuffle

- If a transformation constructs a shuffledRDD, it’ll probably shuffle

- `combineByKey` calls a shuffle (so do other transformations like `groupByKey`, which actually end up calling `combineByKey`)

*Note that `repartition` just calls `coalesce` w/ `true`:

```scala
  coalesce(numPartitions, shuffle = true)
}
RDD.scala*
How do you know if a shuffle will be called on a Transformation?

Transformations that use "numPartitions" like distinct will probably shuffle:

```scala
  map(x => (x, null)).reduceByKey((x, y) => x, numPartitions).map(_._1)
```