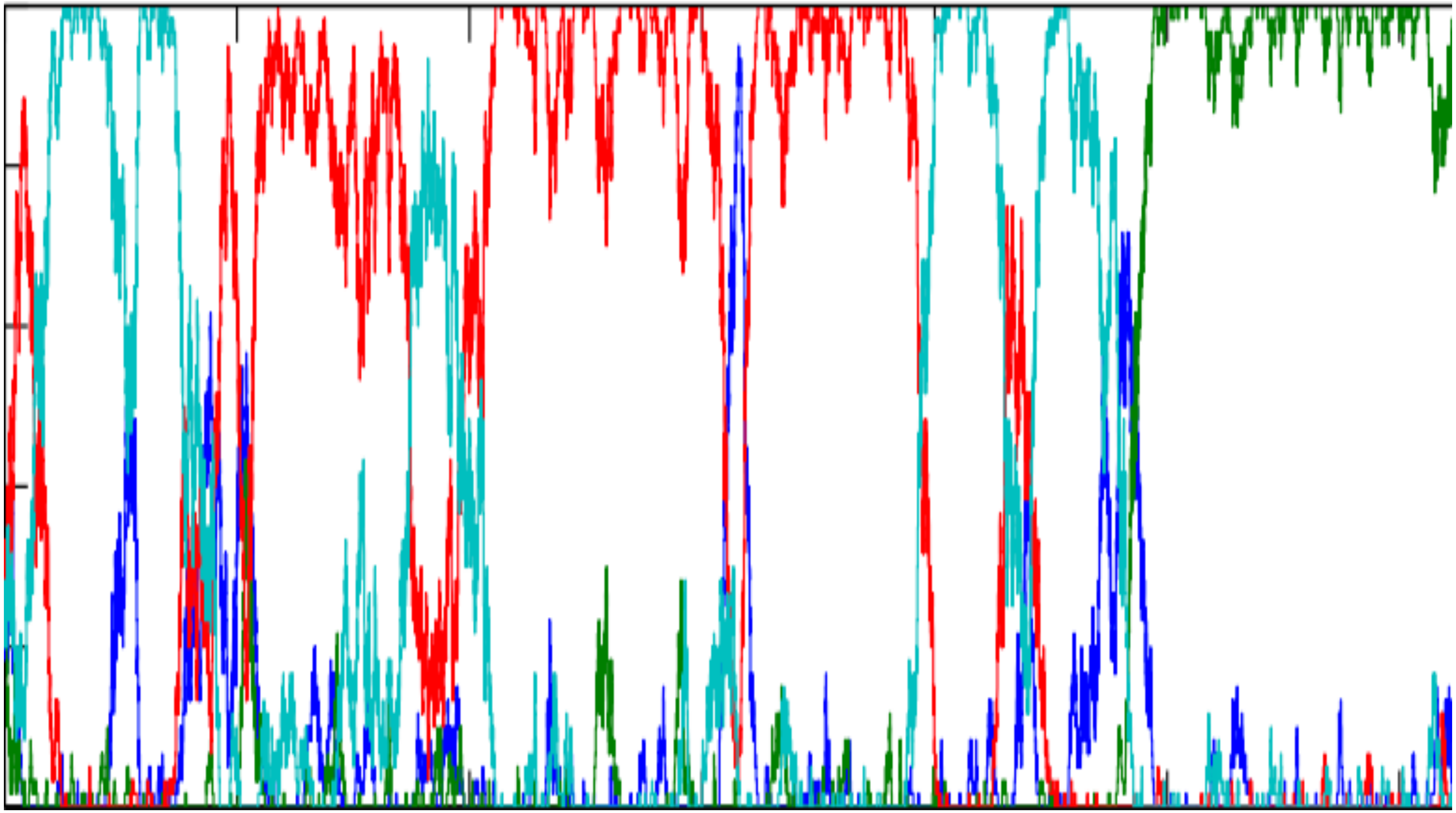


Monitoring and tuning Apache Spark

TANG Gen

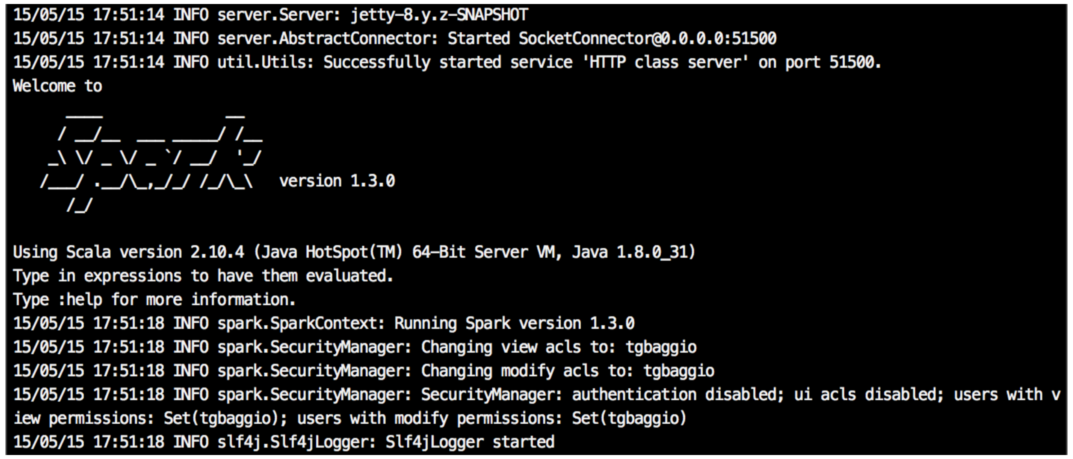
Outline

- Components of execution
  - Jobs, stages and tasks
  - Evaluation of DAGs
  - Spark UI
- Spark UI
- Spark logs
- Key performance considerations

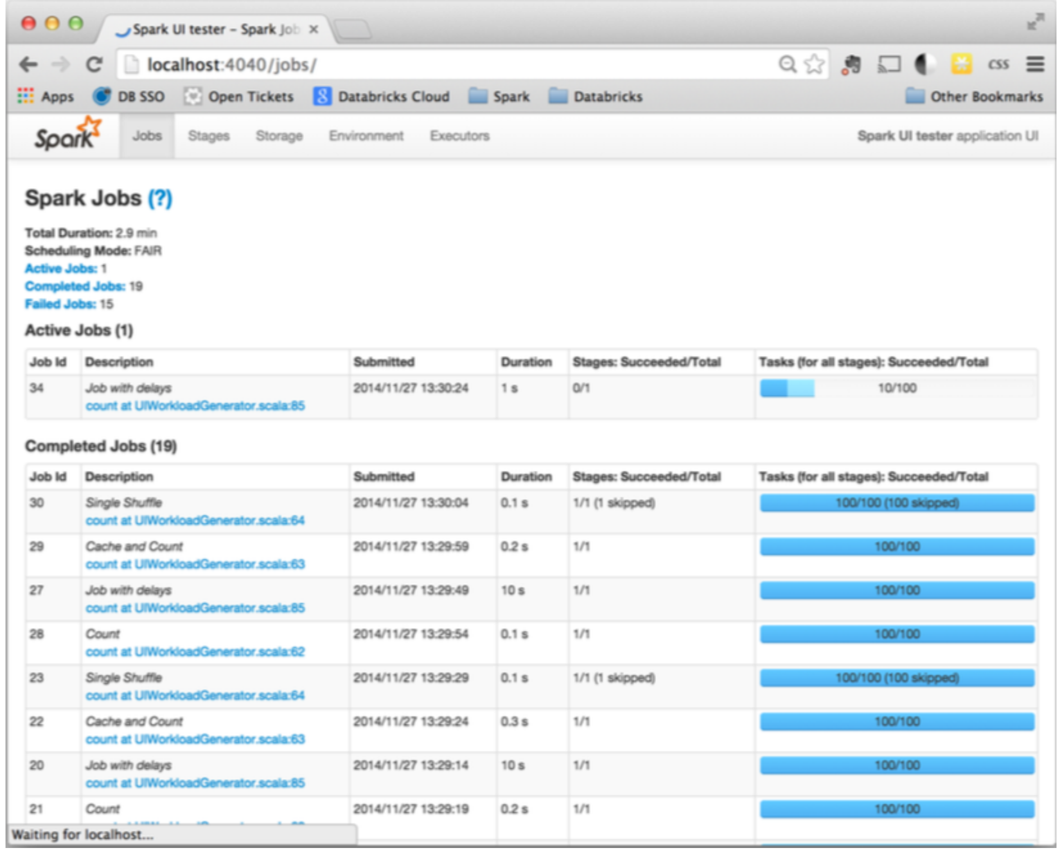
Components of execution

/Jobs, stages and tasks

- How does a user program get translated into units of physical execution: jobs, stages and tasks ?



```
15/05/15 17:51:14 INFO server.Server: jetty-8.y.z-SNAPSHOT
15/05/15 17:51:14 INFO server.AbstractConnector: Started SocketConnector@0.0.0.0:51500
15/05/15 17:51:14 INFO util.Utils: Successfully started service 'HTTP class server' on port 51500.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.3.0
      /_/

Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_31)
Type in expressions to have them evaluated.
Type :help for more information.
15/05/15 17:51:18 INFO spark.SparkContext: Running Spark version 1.3.0
15/05/15 17:51:18 INFO spark.SecurityManager: Changing view acls to: tgbaggio
15/05/15 17:51:18 INFO spark.SecurityManager: Changing modify acls to: tgbaggio
15/05/15 17:51:18 INFO spark.SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users with v
iew permissions: Set(tgbaggio); users with modify permissions: Set(tgbaggio)
15/05/15 17:51:18 INFO slf4j.Slf4jLogger: Slf4jLogger started
```



Spark Jobs (?)

Total Duration: 2.9 min
Scheduling Mode: FAIR
Active Jobs: 1
Completed Jobs: 19
Failed Jobs: 15

Active Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
34	*Job with delays* count at UIWorkloadGenerator.scala:85	2014/11/27 13:30:24	1 s	0/1	10/100

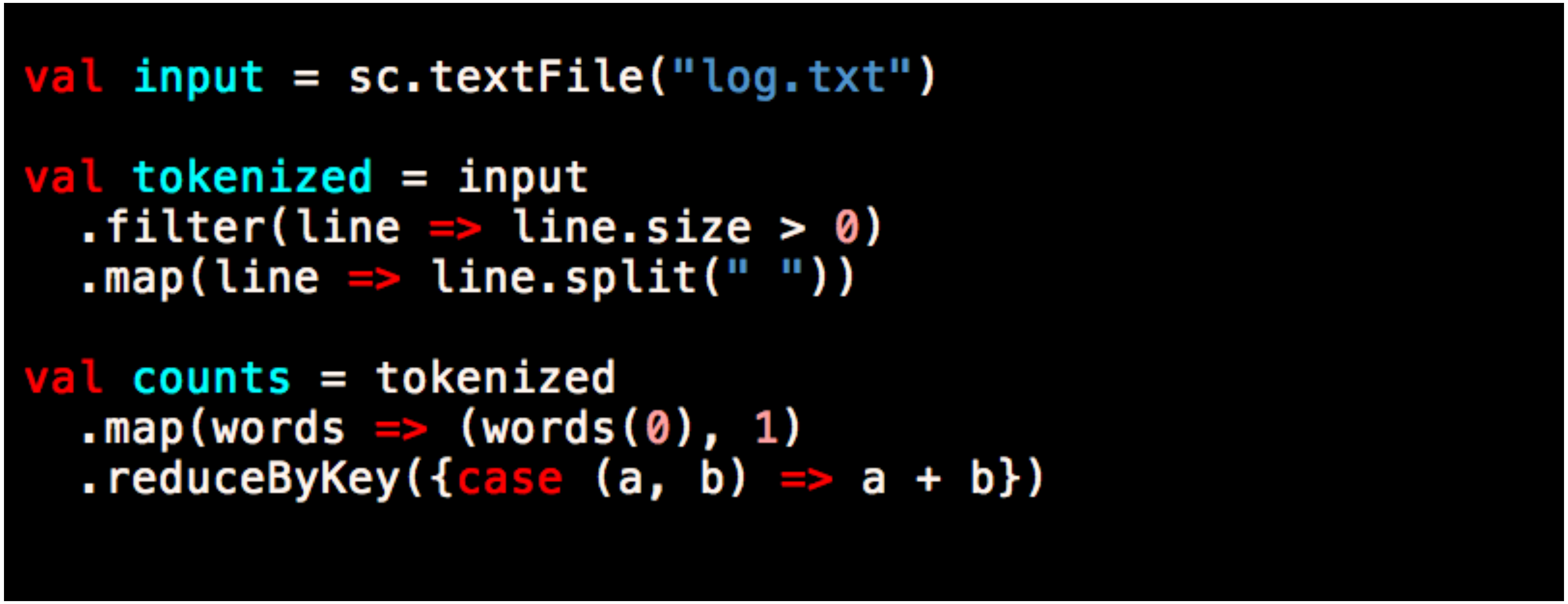
Completed Jobs (19)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
30	*Single Shuffle* count at UIWorkloadGenerator.scala:64	2014/11/27 13:30:04	0.1 s	1/1 (1 skipped)	100/100 (100 skipped)
29	*Cache and Count* count at UIWorkloadGenerator.scala:63	2014/11/27 13:29:59	0.2 s	1/1	100/100
27	*Job with delays* count at UIWorkloadGenerator.scala:85	2014/11/27 13:29:49	10 s	1/1	100/100
28	*Count* count at UIWorkloadGenerator.scala:62	2014/11/27 13:29:54	0.1 s	1/1	100/100
23	*Single Shuffle* count at UIWorkloadGenerator.scala:64	2014/11/27 13:29:29	0.1 s	1/1 (1 skipped)	100/100 (100 skipped)
22	*Cache and Count* count at UIWorkloadGenerator.scala:63	2014/11/27 13:29:24	0.3 s	1/1	100/100
20	*Job with delays* count at UIWorkloadGenerator.scala:85	2014/11/27 13:29:14	10 s	1/1	100/100
21	*Count*	2014/11/27 13:29:19	0.2 s	1/1	100/100

Components of execution

/Jobs, stages and tasks

- Consider a very simple example
  - Read a log file
  - Split into words and remove empty lines
  - Extract the log level and do a count

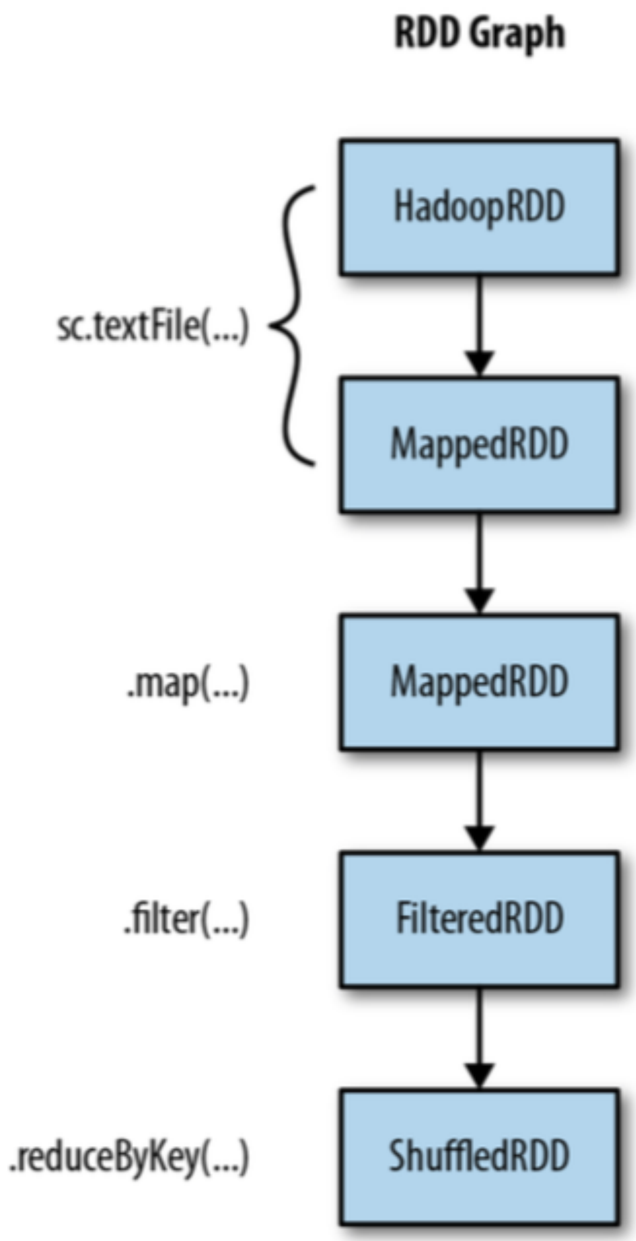


```
val input = sc.textFile("log.txt")

val tokenized = input
  .filter(line => line.size > 0)
  .map(line => line.split(" "))

val counts = tokenized
  .map(words => (words(0), 1)
  .reduceByKey({case (a, b) => a + b})
```

Components of execution

/Jobs, stages and tasks

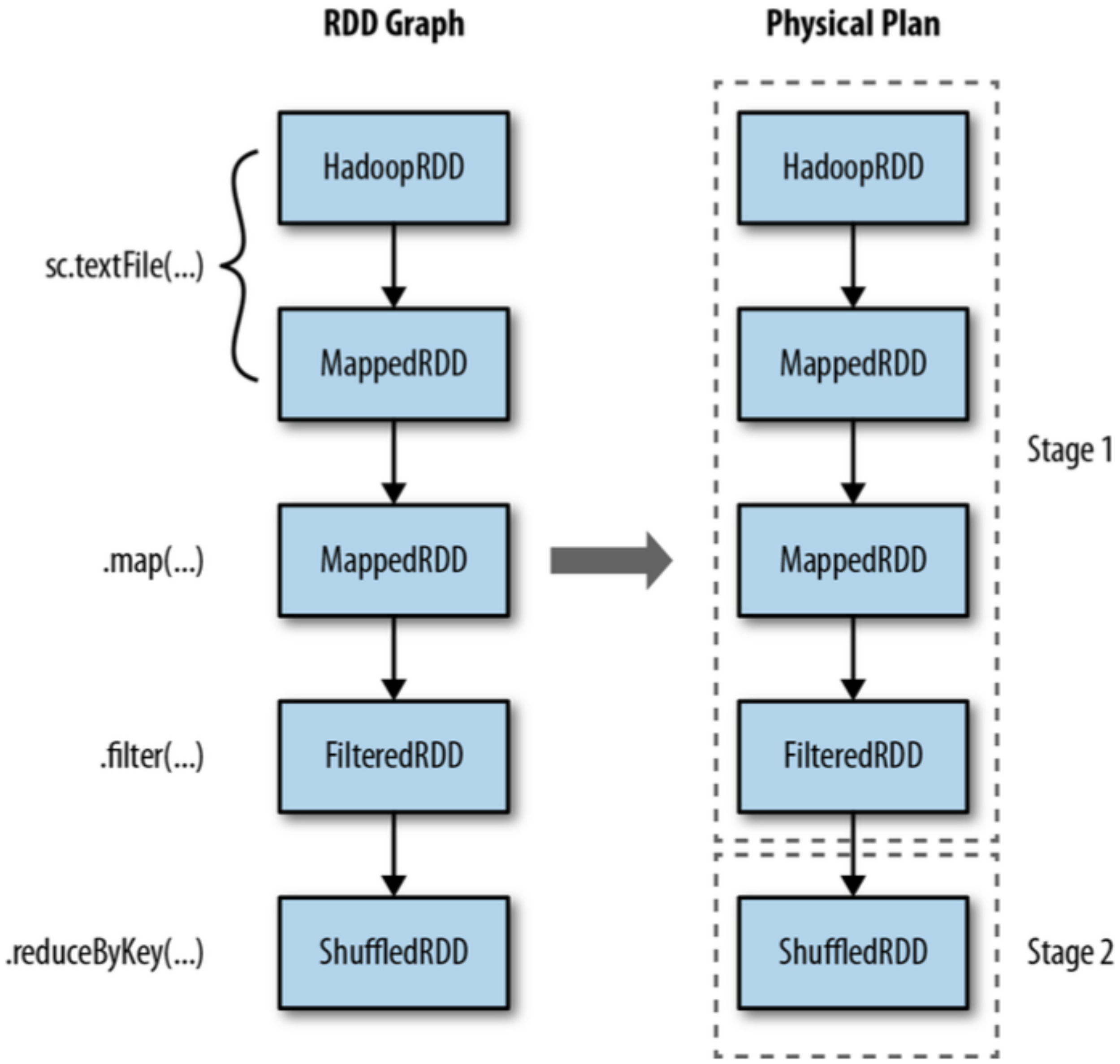
- If we execute the above commands in Spark shell, the program has not performed any actions
- Transformations build up a **DAG**, but don’t “do anything
- DAG is directed acyclic graph which will remember the lineage of RDD, including parents, dependencies, etc. It provides fault-tolerance of Spark
- Spark’s scheduler creates a physical execution plan according to DAG when an action runs on RDD



Components of execution

/Jobs, stages and tasks

- If we add an action to RDD counts, for example, counts.count()
- Spark's scheduler starts at the final RDD being computed and works backward to find what it much compute
- The scheduler outputs a computation stages for DAG. Each stage has tasks for each partition in RDD. Those stages are executed in reverse order

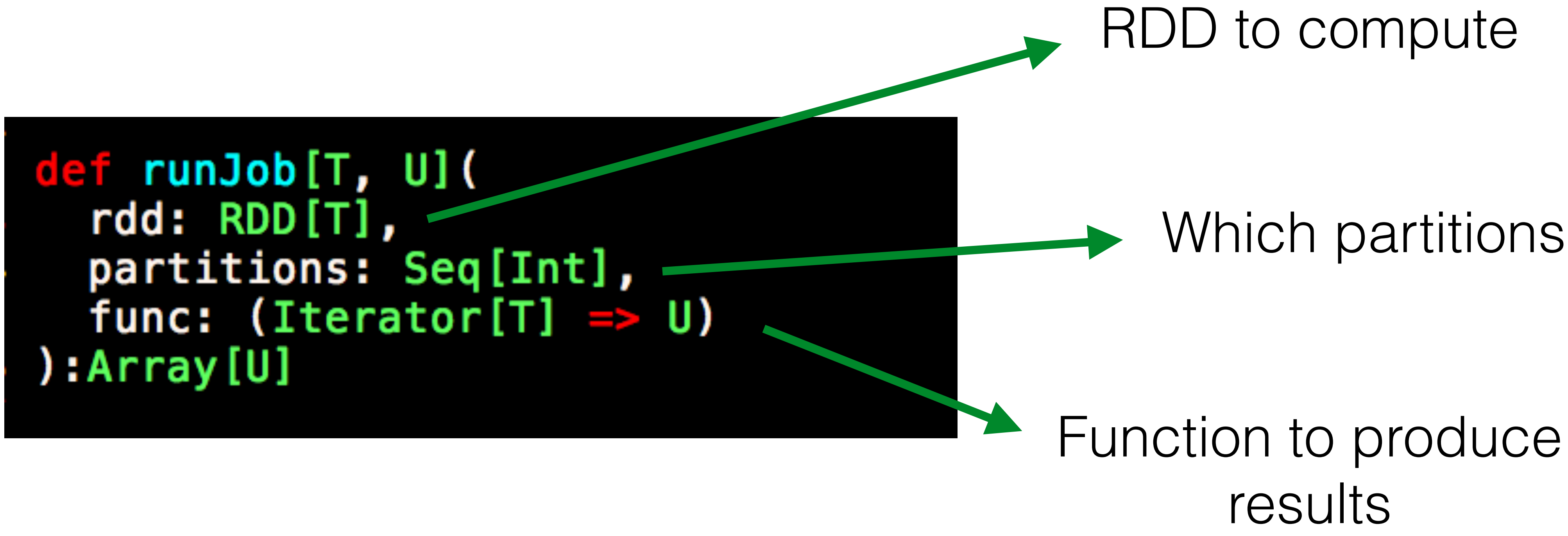


Components of execution

/Evaluation of DAGs

- Before, we say that “actions” force the evaluation of RDD which is True. But Let’s forget it for a moment

- However, in fact, DAGs are materialized through a method sc.runJob in Spark Core



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

RDD to compute

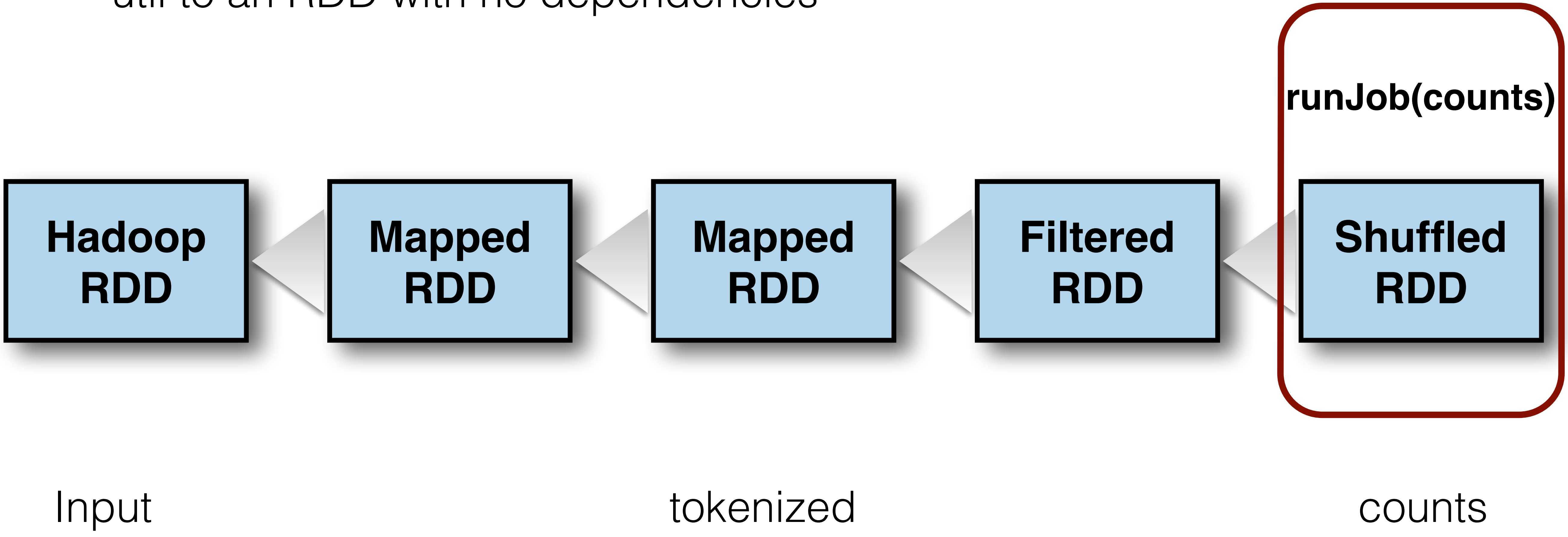
Which partitions

Function to produce results

Components of execution

/Evaluation of DAGs

- RunJob needs to compute RDD parents, parents' parents and etc, util to an RDD with no dependencies



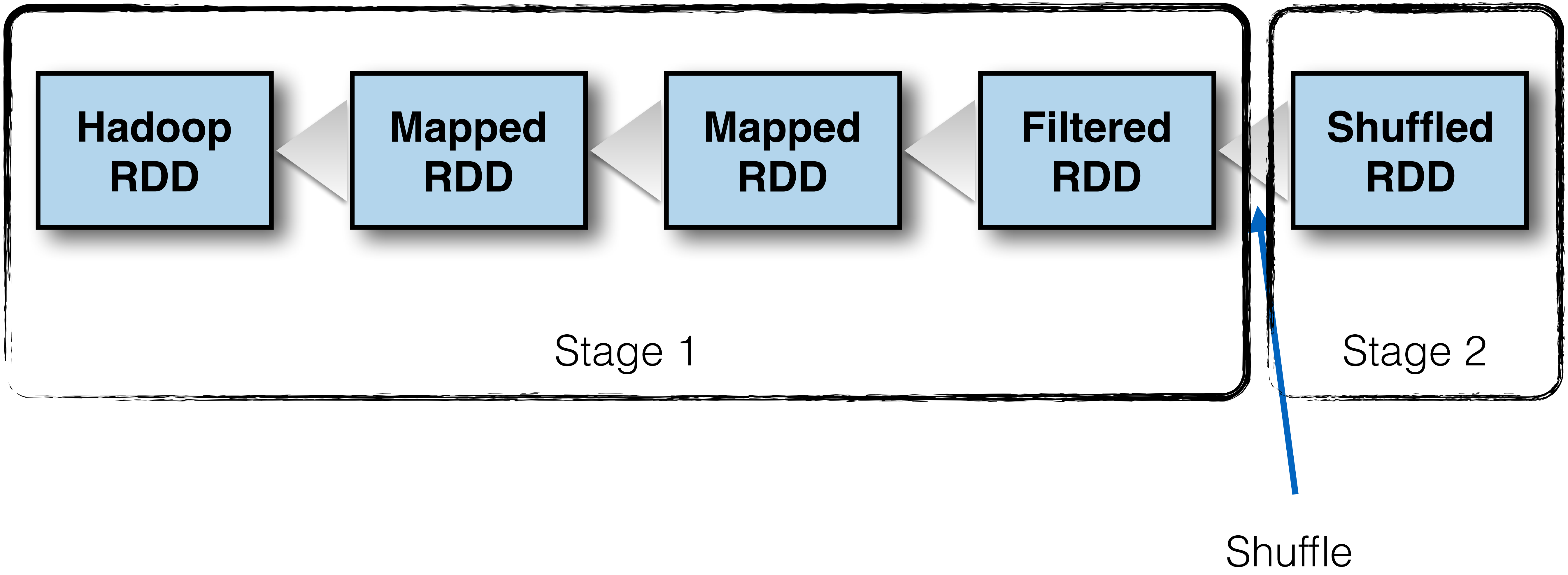
Components of execution

/Physical optimization

- Certain types of transformation can be **pipelined**
- If dependent RDD’s have already been cached (or persisted in a shuffle), the graph can be **truncated**
- Once pipelining and truncation occur, Spark produces a set of **stages**, each stage is composed of **tasks**

Components of execution

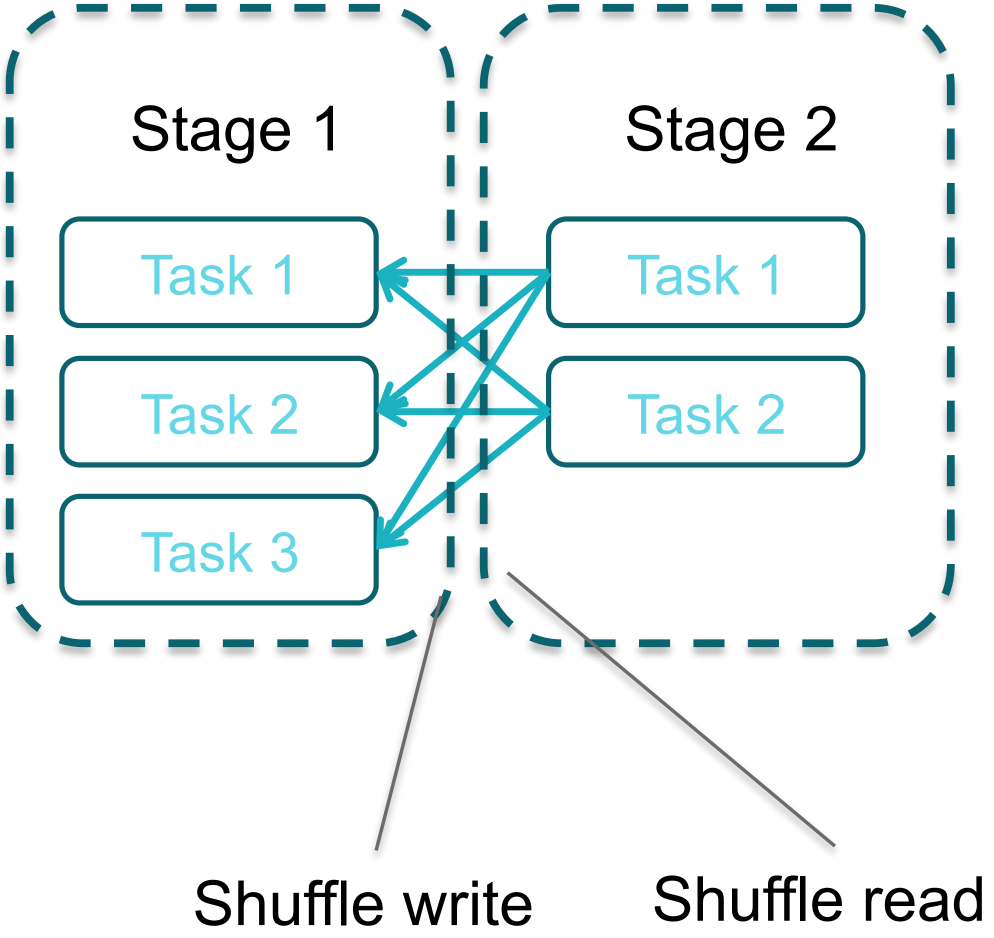
/Physical optimization



Components of execution

/Stage graph

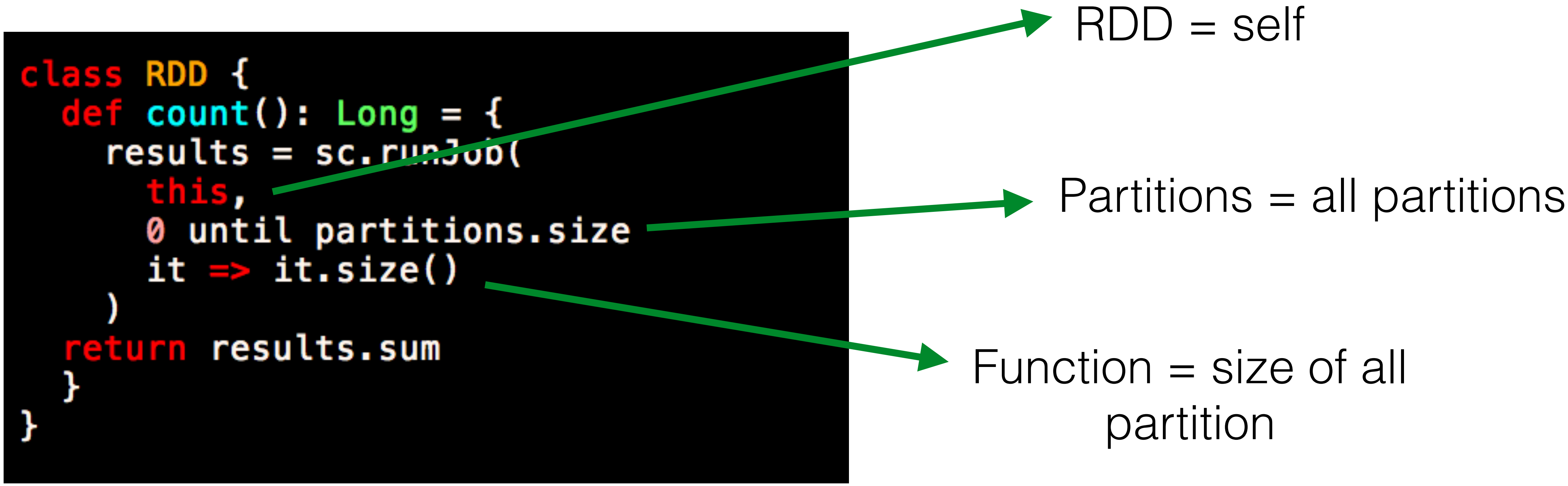
- In stage 1, each task will
    1. Read hadoop file
    2. Perform maps and filters
    3. Write partial sums



- In stage 2, each task will
    1. Read partial sums
    2. Invoke user function passed to runJob

Components of execution

/Count() action



```
class RDD {
  def count(): Long = {
    results = sc.runJob(
      this,
      0 until partitions.size
      it => it.size()
    )
  return results.sum
  }
}
```

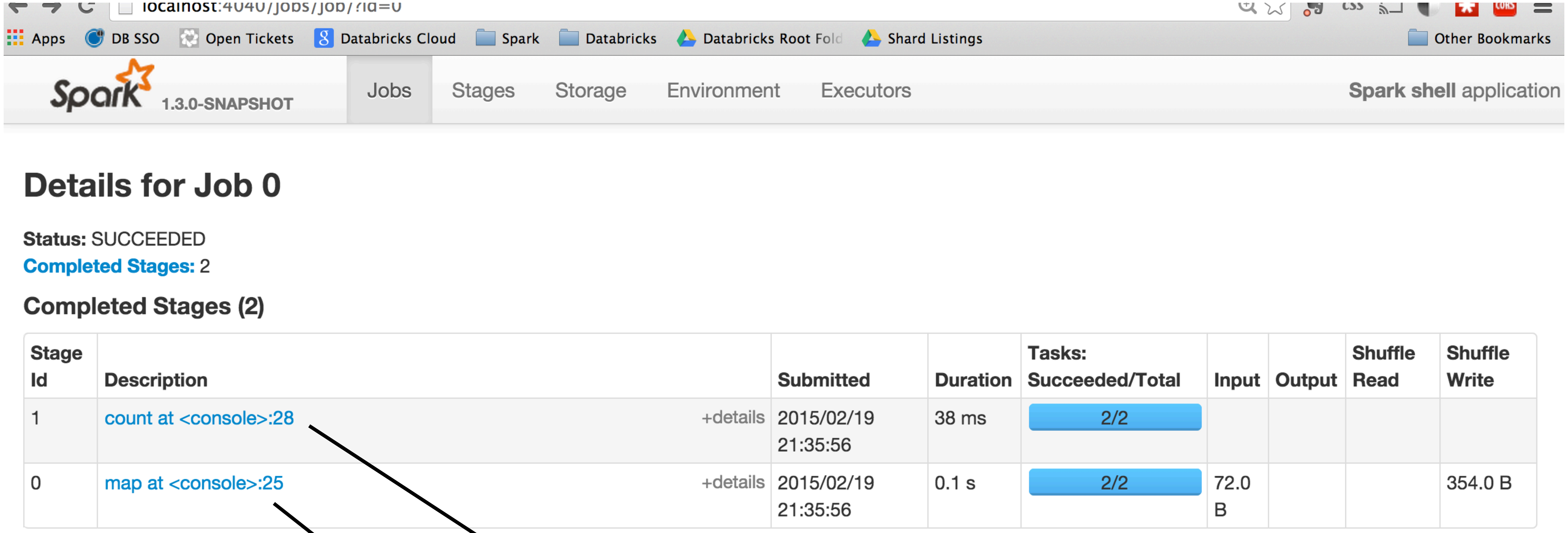
RDD = self

Partitions = all partitions

Function = size of all partition

Components of execution

/Spark UI



Details for Job 0

Status: SUCCEEDED

Completed Stages: 2

Completed Stages (2)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
1	count at <console>:28 +details	2015/02/19 21:35:56	38 ms	2/2				
0	map at <console>:25 +details	2015/02/19 21:35:56	0.1 s	2/2	72.0 B			354.0 B

Named after action calling runJob

Named after last RDD in pipeline

Components of execution

/Summary

- Definition:
  - ***Jobs*: ** Work required to compute RDD in runJob.
  - ***Stages:*** A wave of work within a job, corresponding to one or more pipelined RDD’s.
  - ***Tasks:*** A unit of work within a stage, corresponding to one RDD partition.
  - ***Shuffle:*** The transfer of data between stages.

- Phases occur during Spark execution:
  - User code defines a DAG of RDDs
  - Actions force translation of the DAG to an execution plan
  - Tasks are scheduled and executed on a cluster

Spark UI

/Demo time

- If Spark runs on standalone or Mesos cluster, Spark’s built-in web UI is available on the machine where driver is running at port 4040 by default
- If Spark runs on YARN cluster mode， the driver runs inside the cluster, the UI is accessed through the YARN ResourceManager

Spark logs

/Driver and executor logs

- The location of Spark's log files depends on the deployment mode:
  - Standalone, application logs are directly displayed in the stand-alone master's web UI. They are stored by default in the *work/* directory of the Spark distribution on each worker.
  - In Mesos, logs are stored in the *work/* directory of a Mesos slave, and accessible from Mesos master UI
  - In YARN mode, when the application finishes, use the following command to produce a report containing logs

    yarn logs -applicationID <app ID>
  - In YARN mode, when the application is running, we can access the logs of certain containers via ResourceManager UI

Spark logs

/Logging system

- Spark's logging subsystem is based on *log4j,* a widely used Java library

- An example log4j configuration file is bundled with Spark at *conf/log4j.properties.template*. To customize Spark's logging,

  1. Copy the example to a file called *log4j.properties*.

  2. Modify behavior such as the root logging level (the threshold level for logging output).

  3. Add the *log4j.properties* file using the *--files* flag of *spark-submit*

Key performance considerations

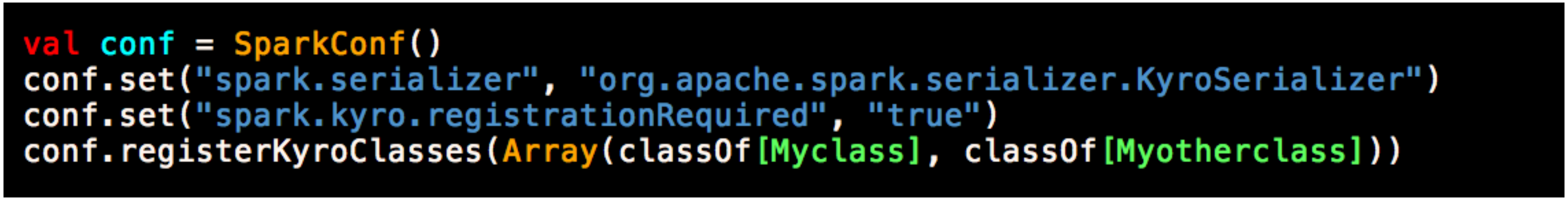
/Level of Parallelism

- RDD’s partitions size decides the level of parallelism during execution, which can be modified by *coalesce(Num)* or *repartition(Num)*
- Demo time
    - Too little parallelism: idle cores
    - Too much parallelism: too much shuffle

Key performance considerations

/Serialization format

- When Spark is transferring data over the network or spilling data to disk, it needs to serialize objects into a binary format
- By default, Spark will use Java's built-in serializer.
- However, Spark also support the use of Kyro, a third-party serialization library (Kyro is used in mllib library)
- To use Kyro serializer



```
val conf = SparkConf()
conf.set("spark.serializer", "org.apache.spark.serializer.KyroSerializer")
conf.set("spark.kyro.registrationRequired", "true")
conf.registerKyroClasses(Array(classOf[Myclass], classOf[Myotherclass]))
```

Key performance considerations

/Memory management

- In general, memory is used for the following three ways:
  - RDD storage
  - Shuffle and aggregation buffers
  - User code
- The more detailed presentation of memory management will be in tommorrow