DATABRICKS

Advanced Spark

Reynold Xin, July 2, 2014 @ Spark Summit Training

Word Count

In this example, we use a few more transformations to build a dataset of (String, Int) pairs called `counts` and then save it to a file.

Python | Scala | Java

```
file = spark.textFile("hdfs://...")
counts = file.flatMap(lambda line: line.split(" ")) \
             .map(lambda word: (word, 1)) \
             .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://...")
```

This Talk

Formalize RDD concept

Life of a Spark Application

Performance Debugging

"Mechanical sympathy" by Jackie Stewart: a driver does not need to know how to build an engine but they need to know the fundamentals of how one works to get the best out of it

* Assumes you can write word count, knows what transformation/action is

Reynold Xin

Apache Spark committer (worked on almost every module: core, sql, mllib, graph)

Product & open-source eng @ Databricks

On leave from PhD @ UC Berkeley AMPLab

DATABRICKS

Example Application

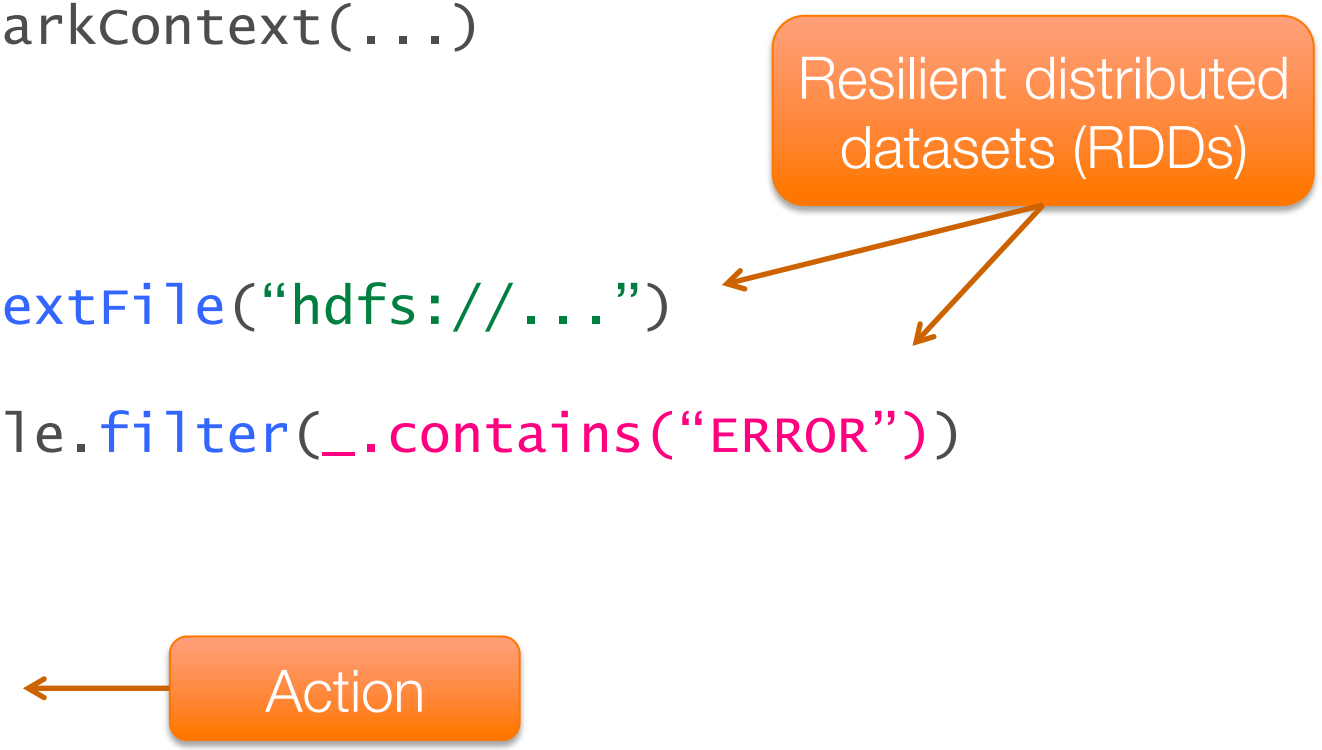
```
val sc = new SparkContext(...)

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

val errors = file.filter(_.contains("ERROR"))

errors.cache()

errors.count()
```



Quiz: what is an “RDD”?

A: distributed collection of objects on disk

B: distributed collection of objects in memory

C: distributed collection of objects in Cassandra

Answer: could be any of the above!

Scientific Answer: RDD is an Interface!

1. Set of *partitions* ("splits" in Hadoop)
2. List of *dependencies* on parent RDDs
3. Function to *compute* a partition (as an Iterator) given its parent(s)

"lineage"

4. (Optional) *partitioner* (hash, range)
5. (Optional) *preferred location*(s) for each partition

optimized execution

Example: HadoopRDD

partitions = one per HDFS block

dependencies = none

compute(*part*) = read corresponding block

preferredLocations(*part*) = HDFS block location

partitioner = none

Example: Filtered RDD

partitions = same as parent RDD

dependencies = “one-to-one” on parent

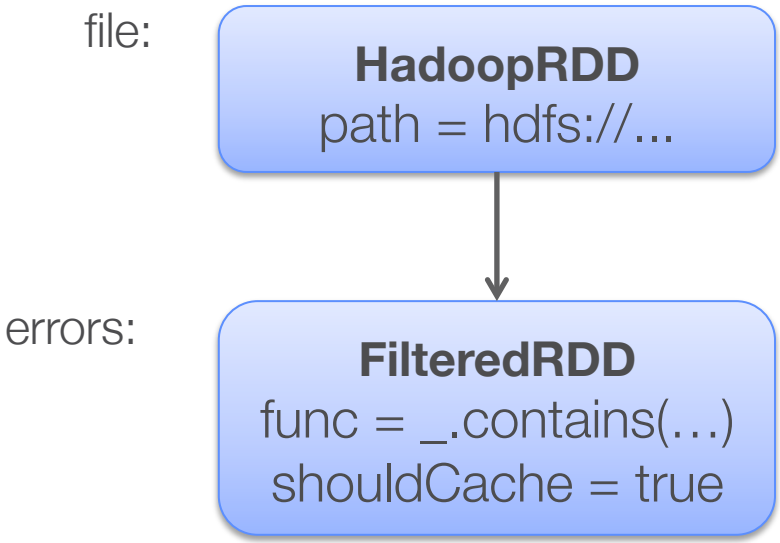
compute*(part)* = compute parent and filter it

preferredLocations*(part)* = none (ask parent)

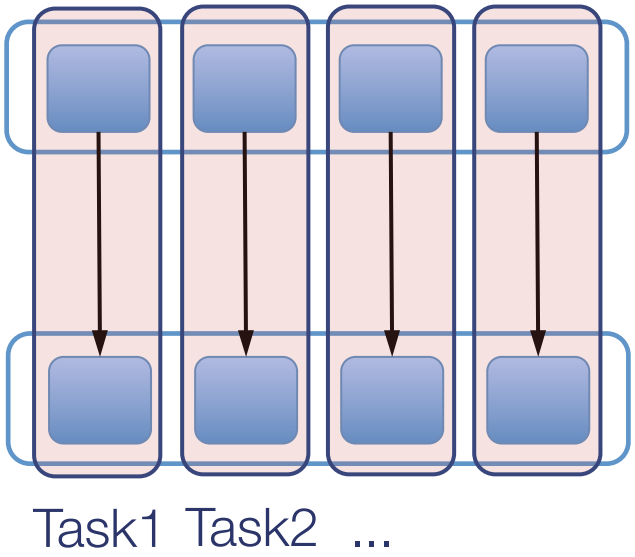
partitioner = none

RDD Graph (DAG of tasks)

Dataset-level view:



Partition-level view:



Example: JoinedRDD

partitions = one per reduce task

dependencies = “shuffle” on each parent

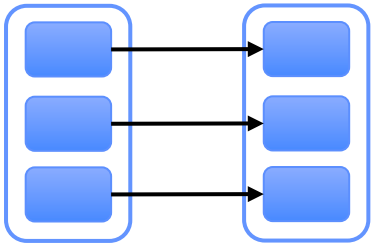
compute*(partition)* = read and join shuffled data

preferredLocations*(part)* = none

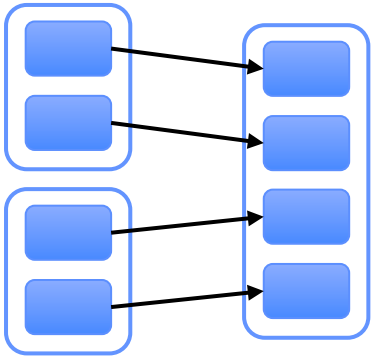
partitioner = HashPartitioner(numTasks)

Dependency Types

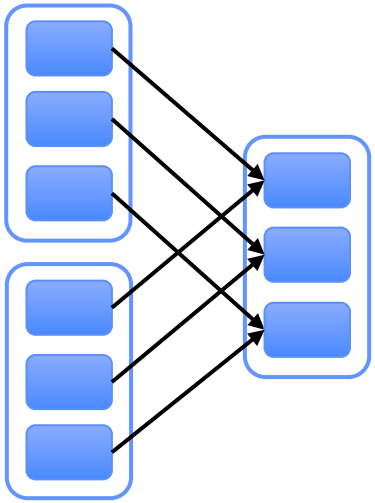
"Narrow" (pipeline-able)



map, filter

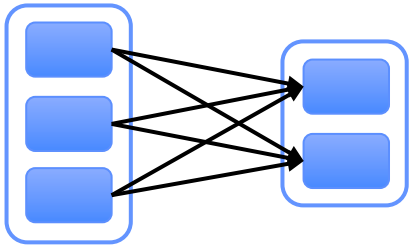


union

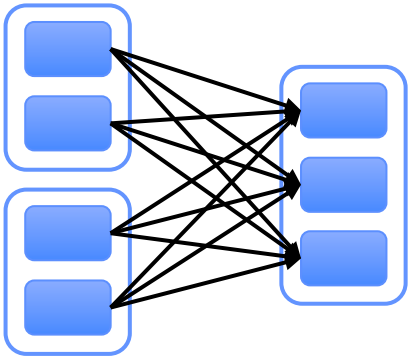


join with inputs co-partitioned

"Wide" (shuffle)



groupByKey on non-partitioned data



join with inputs not co-partitioned

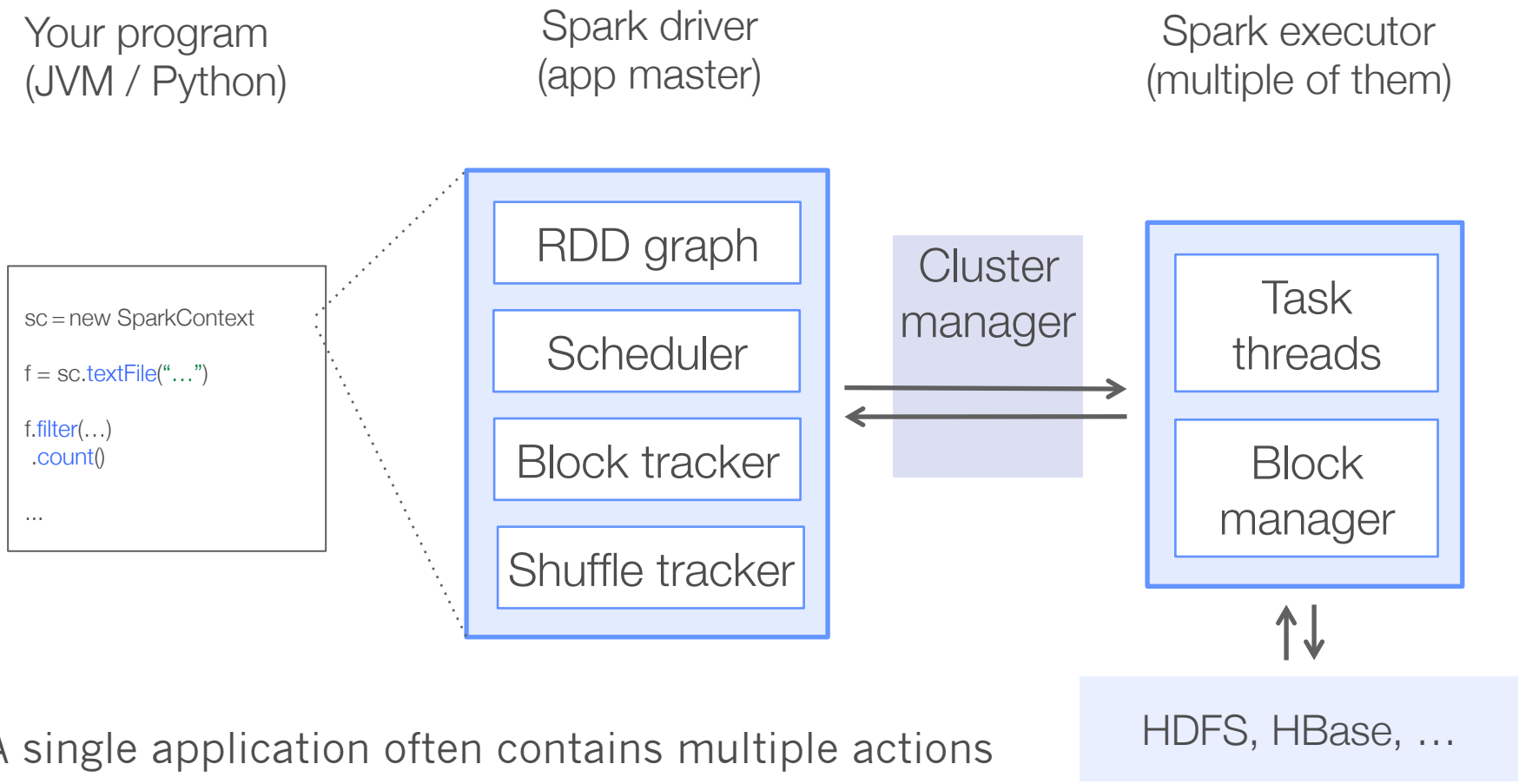
Recap

Each RDD consists of 5 properties:

1. partitions
2. dependencies
3. compute
4. (optional) partitioner
5. (optional) preferred locations

Life of a Spark Application

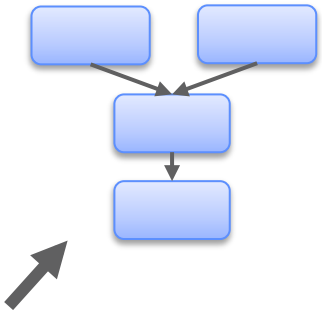
Spark Application



A single application often contains multiple actions

Job Scheduling Process

RDD Objects

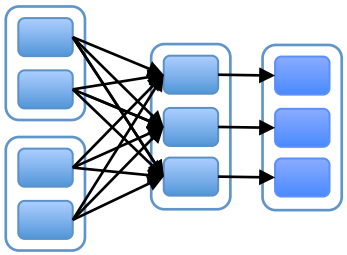


```
rdd1.join(rdd2)
    .groupBy(…)
    .filter(…)
    .count()
```

build operator DAG

DAG

Scheduler (DAGScheduler)



split graph into *stages* of tasks

submit each stage as ready

Task

Executors

execute tasks

store and serve blocks

DATABRICKS

DAG Scheduler

Input: RDD and partitions to compute

Output: output from actions on those partitions

Roles:

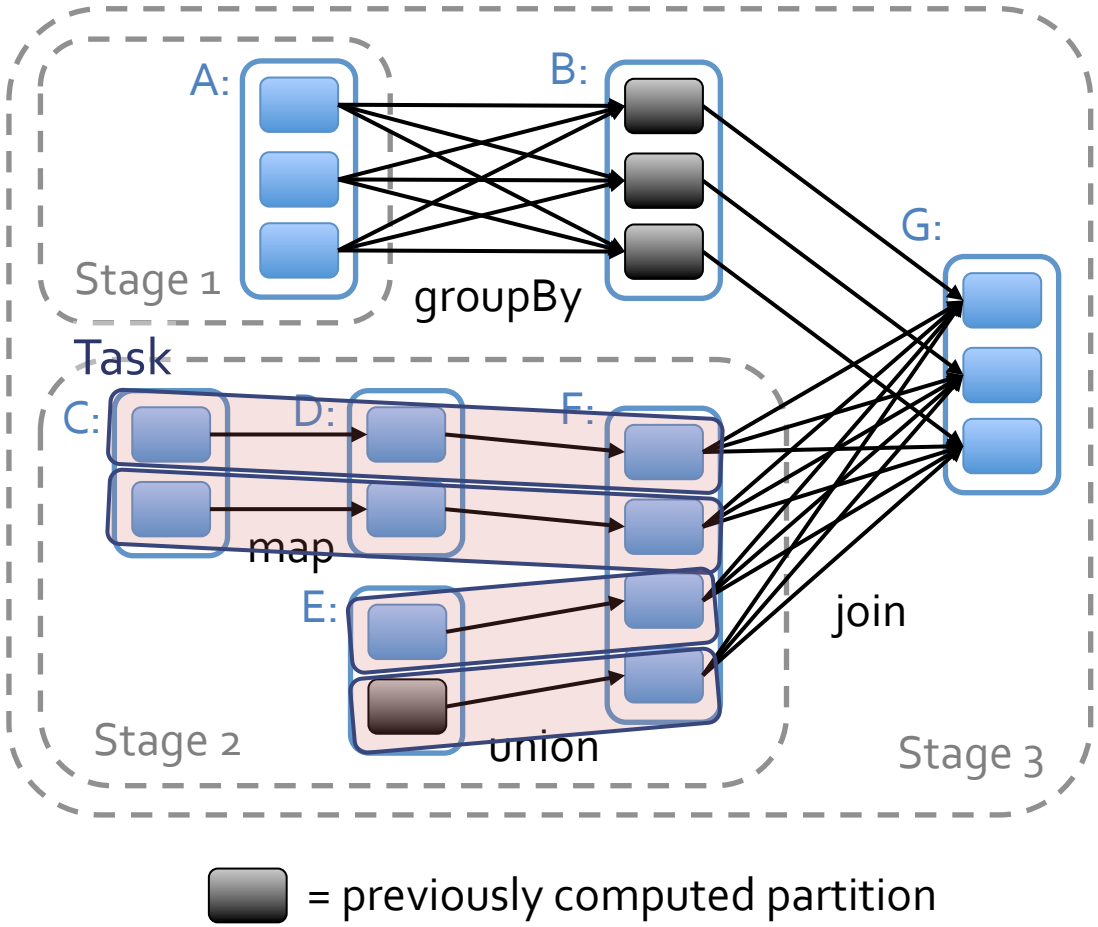
- Build stages of tasks
- Submit them to lower level scheduler (e.g. YARN, Mesos, Standalone) as ready
- Lower level scheduler will schedule data based on locality
- Resubmit failed stages if outputs are lost

Scheduler Optimizations

Pipelines operations within a stage

Picks join algorithms based on partitioning (minimize shuffles)

Reuses previously cached data



Task

Unit of work to execute on in an executor thread

Unlike MR, there is no “map” vs “reduce” task

Each task either partitions its output for “shuffle”, or send the output back to the driver

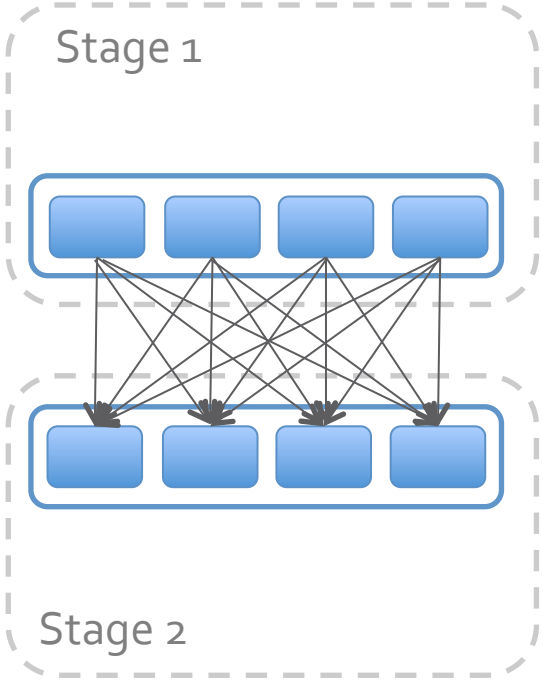
Shuffle

Redistributes data among partitions

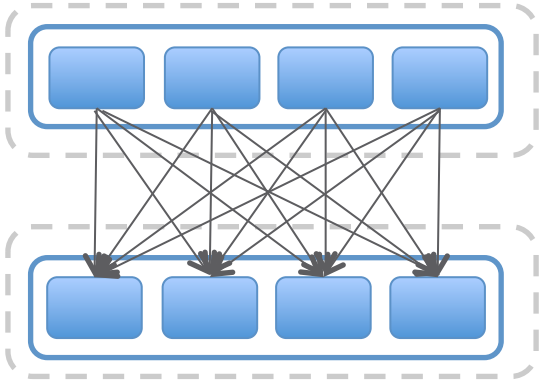
Partition keys into buckets
(user-defined partitioner)

Optimizations:

- Avoided when possible, if data is already properly partitioned
- Partial aggregation reduces data movement

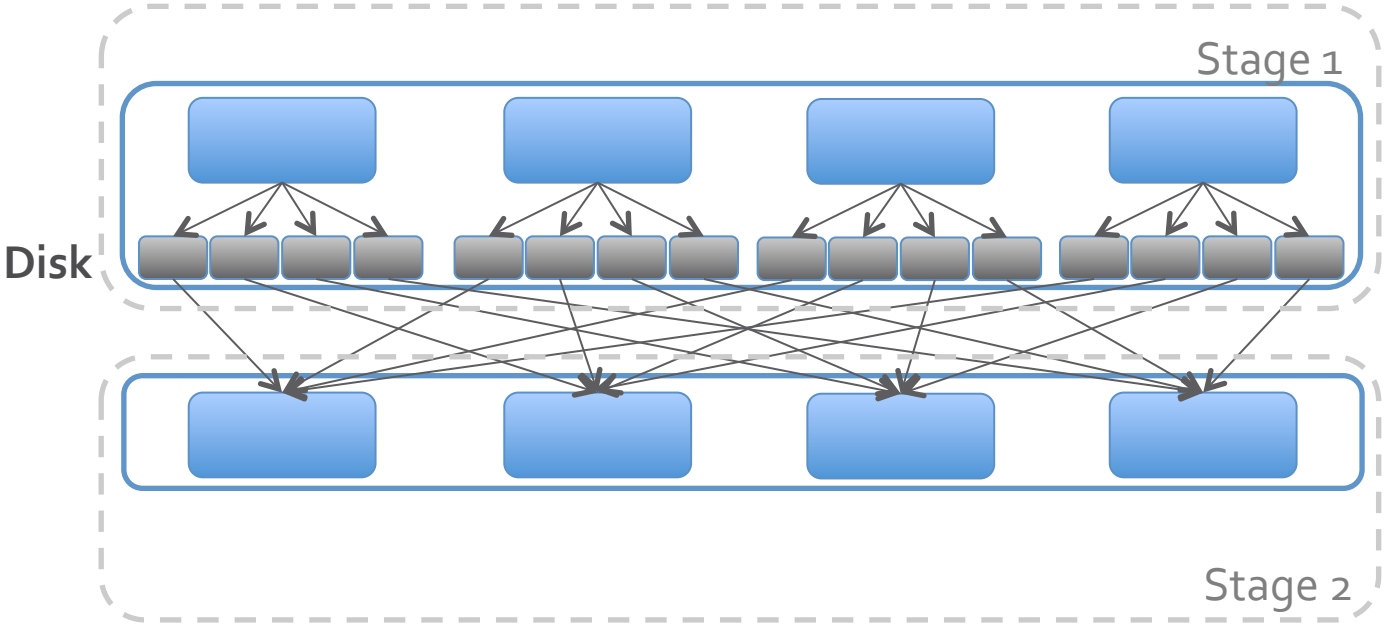


Shuffle



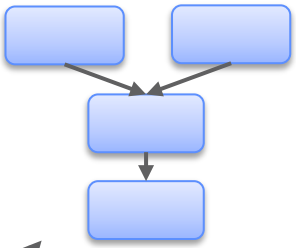
Write intermediate files to disk

Fetched by the next stage of tasks (“reduce” in MR)



Recap: Job Scheduling

RDD Objects

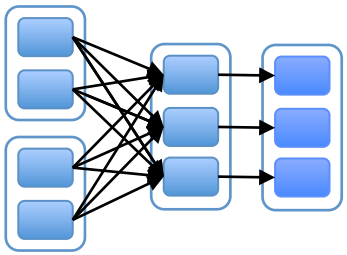


```
rdd1.join(rdd2)
    .groupBy(…)
    .filter(…)
    .count()
```

build operator DAG

DAG

Scheduler (DAGScheduler)



split graph into *stages* of tasks

submit each stage as ready

Task

Executors

execute tasks

store and serve blocks

Performance Debugging

Performance Debugging

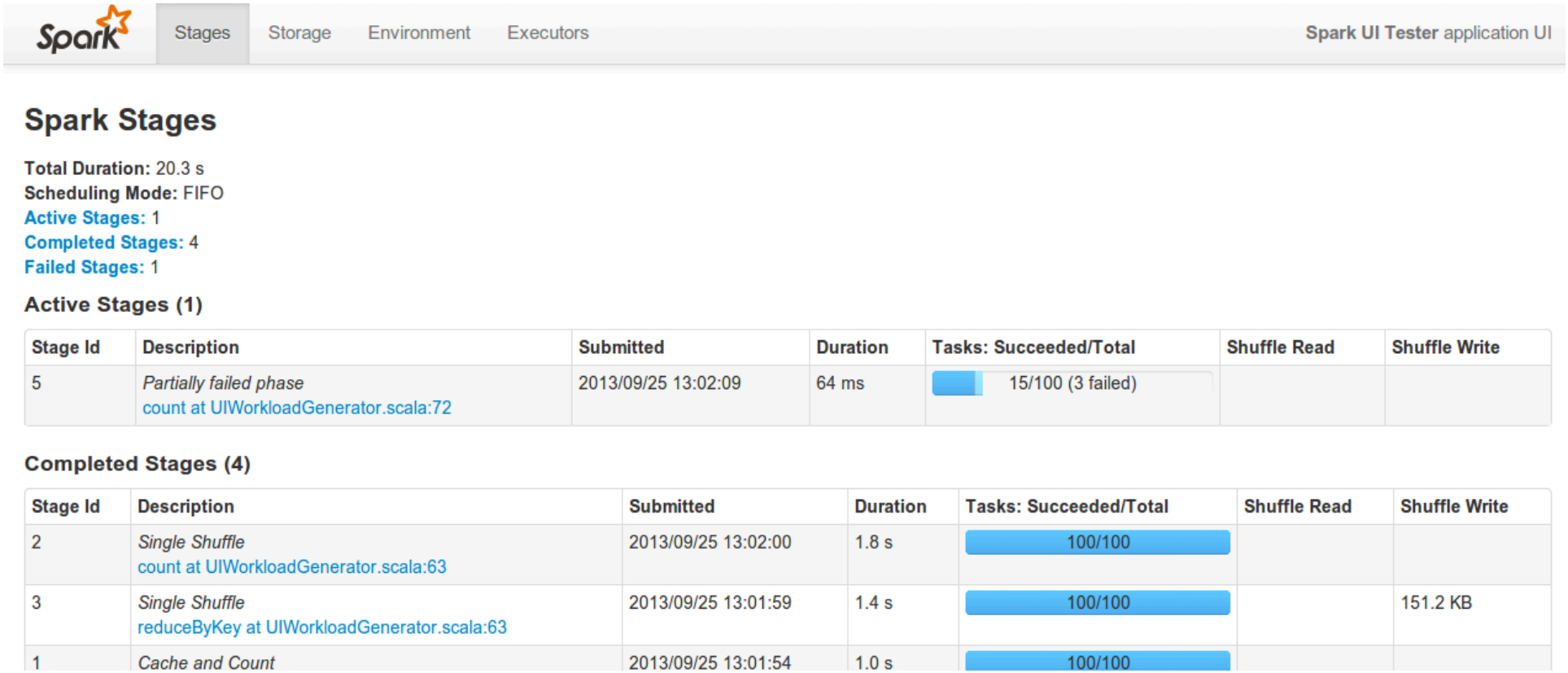
Distributed performance: program slow due to scheduling, coordination, or data distribution)

Local performance: program slow because whatever I'm running is just slow on a single node

Two useful tools:

- Application web UI (default port 4040)
- Executor logs (spark/work)

Find Slow Stage(s)



DATABRICKS

Stragglers?

Some tasks are just slower than others.

Easy to identify from summary metrics:

Summary Metrics for 19 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Result serialization time	0 ms	0 ms	0 ms	0 ms	
Duration	1 s	1 s	1 s	1 s	10 s
Time spent fetching task results	0 ms	0 ms	0 ms	0 ms	
Scheduler delay	14 ms	21 ms	37 ms	49 ms	58 ms

Stragglers due to slow nodes

```
sc.parallelize(1 to 15, 15).map { index =>
  val host = java.net.InetAddress.getLocalHost.getHostName
  if (host == "ip-172-31-2-222") {
    Thread.sleep(10000)
  } else {
    Thread.sleep(1000)
```

Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Shu Rea
0	ip-172-31-2-223.us-west-2.compute.internal:44092	8 s	8	0	8	0.0
1	ip-172-31-2-224.us-west-2.compute.internal:34889	7 s	7	0	7	0.0
2	ip-172-31-2-222.us-west-2.compute.internal:32996	40 s	4	0	4	0.0

DATABRICKS

Stragglers due to slow nodes

Turn speculation on to mitigates this problem.

Speculation: Spark identifies slow tasks (by looking at runtime distribution), and re-launches those tasks on other nodes.

spark.speculation true

Demo Time: slow node

Stragglers due to data skew

```
sc.parallelize(1 to 15, 15)
  .flatMap { i => 1 to i }
  .map { i => Thread.sleep(1000) }
  .count()
```

Speculation is not going to help because the problem is inherent in the algorithm/data.

Pick a different algorithm or restructure the data.

Demo Time

Tasks are just slow

Garbage collection

Performance of the code running in each task

Garbage Collection

Look at the “GC Time” column in the web UI

Tasks

Task Index	Task ID	Status	Locality Level	Executor	Launch Time	Duratio	GC Time	R T
0	96	SUCCESS	PROCESS_LOCAL	ip-172-31-2-222.us-west-2.compute.internal	2014/07/02 08:05:53	7 s	58 ms	

DATABRICKS

What if the task is still running?

To discover whether GC is the problem:

1. Set spark.executor.extraJavaOptions to include: “-XX:-PrintGCDetails -XX:+PrintGCTimeStamps”
2. Look at spark/work/app…/[n]/stdout on executors
3. Short GC times are OK. Long ones are bad.

```
root@ip-172-31-2-222 ~]$ cd spark
root@ip-172-31-2-222 spark]$ ls
bin  CHANGES.txt  conf  ec2  examples  lib  LICENSE  logs  NOTICE  python
root@ip-172-31-2-222 spark]$ cd work
root@ip-172-31-2-222 work]$ ls
app-20140702071428-0000  app-20140702072642-0002  app-20140702081339-0004
app-20140702072144-0001  app-20140702073903-0003
root@ip-172-31-2-222 work]$ ls -ltrh
total 20K
drwxr-xr-x 3 root root 4.0K Jul  2 07:14 app-20140702071428-0000
drwxr-xr-x 3 root root 4.0K Jul  2 07:21 app-20140702072144-0001
drwxr-xr-x 4 root root 4.0K Jul  2 07:38 app-20140702072642-0002
drwxr-xr-x 4 root root 4.0K Jul  2 08:13 app-20140702073903-0003
drwxr-xr-x 3 root root 4.0K Jul  2 08:13 app-20140702081339-0004
root@ip-172-31-2-222 work]$ cd app-20140702081339-0004/2/
root@ip-172-31-2-222 2]$ ls
stderr  stdout
```

```
root@ip-172-31-2-222 app-20140702081708-0005]$ less 2/stdout
8.939: [GC 3311616K->22889K(12694528K), 0.0450970 secs]
10.413: [GC 3334505K->6364K(12694528K), 0.0115100 secs]
24.277: [GC 3317980K->6492K(12694528K), 0.0095850 secs]
25.008: [GC 3318108K->6472K(12694528K), 0.0101310 secs]
26.603: [GC 3318088K->6476K(12694528K), 0.0096250 secs]
27.148: [GC 3318092K->6472K(13224960K), 0.0119190 secs]
29.607: [GC 4377928K->6511K(13223936K), 0.0128020 secs]
30.417: [GC 4377967K->6479K(13224960K), 0.0016880 secs]
31.827: [GC 4378959K->6495K(13224960K), 0.0016810 secs]
34.077: [GC 4378975K->6567K(13225472K), 0.0016340 secs]
34.833: [GC 4379559K->6599K(13224960K), 0.0015930 secs]
96.112: [GC 4379591K->7247K(13227008K), 0.0020790 secs]
97.292: [GC 4382799K->7303K(13225984K), 0.0021280 secs]
98.153: [GC 4382855K->7327K(13228544K), 0.0015220 secs]
123.293: [GC 4377491K->7375K(13227520K), 0.0016200 secs]
123.295: [Full GC 7375K->6842K(13227520K), 0.1671430 secs]
```

jmap: heap analysis

jmap -histo [pid]

Gets a histogram of objects in the JVM heap

jmap -histo:live [pid]

Gets a histogram of objects in the heap after GC (thus “live”)

```
root@ip-172-31-2-222 app-20140702081708-0005]$ jmap -histo:live 9845 | less

 num     #instances         #bytes  class name
----------------------------------------------
   1:         58569        7506000  <methodKlass>
   2:         58569        7425616  <constMethodKlass>
   3:          4580        5255440  <constantPoolKlass>
   4:          4580        4363632  <instanceKlassKlass>
   5:          3870        2799040  <constantPoolCacheKlass>
   6:         12428        2010560  [B
   7:            47        1540848  [Lscala.concurrent.forkjoin.ForkJoinTask;
   8:         14761        1223144  [C
   9:          4878         578408  java.lang.Class
  10:          8649         479208  [[I
  11:          7390         405504  [S
```

```
root@ip-172-31-2-222 app-20140702081708-0005]$ jmap -histo 9845 | less

 num     #instances         #bytes  class name
----------------------------------------------
   1:     134371218     2149939488  java.lang.Integer
   2:     134370876     2149934016  $line13.$read$$iwC$$iwC$$iwC$$iwC$DummyObject
   3:          1126      135873216  [I
   4:         58573        7506512  <methodKlass>
   5:         58573        7426048  <constMethodKlass>
   6:          4582        5257408  <constantPoolKlass>
   7:          4582        4364784  <instanceKlassKlass>
   8:          3872        2800576  <constantPoolCacheKlass>
   9:         12757        2261696  [B
  10:            49        1606416  [Lscala.concurrent.forkjoin.ForkJoinTask;
  11:         15689        1333976  [C
```

Demo: GC log & jmap

Reduce GC impact

```
class DummyObject(var i: Int) {
  def toInt = i
}

sc.parallelize(1 to 100 * 1000 * 1000, 1).map { i =>
  new DummyObject(i)  // new object every record
  obj.toInt
}


sc.parallelize(1 to 100 * 1000 * 1000, 1).mapPartitions { iter =>
  val obj = new DummyObject(0)  // reuse the same object
  iter.map { i =>
    obj.i = i
    obj.toInt
  }
}
```

Local Performance

Each Spark executor runs a JVM/Python process

Insert your favorite JVM/Python profiling tool

- jstack
- YourKit
- VisualVM
- println
- (sorry I don't know a whole lot about Python)
- …

Example: identify expensive comp.

```
def someCheapComputation(record: Int): Int = record + 1

def someExpensiveComputation(record: Int): String = {
  Thread.sleep(1000)
  record.toString
}

sc.parallelize(1 to 100000).map { record =>
  val step1 = someCheapComputation(record)
  val step2 = someExpensiveComputation(step1)
  step2
}.saveAsTextFile("hdfs:/tmp1")
```

Demo Time

jstack

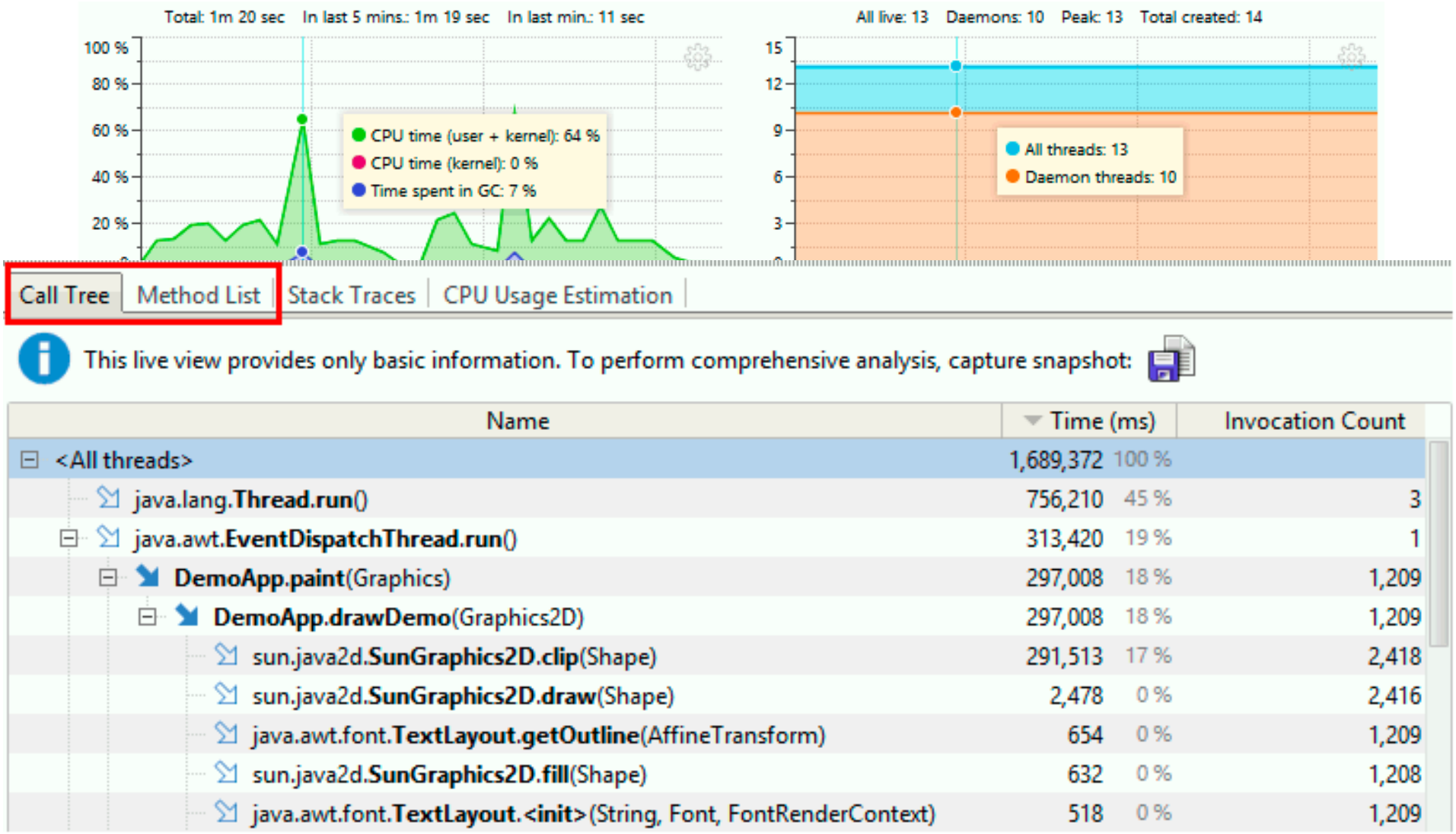
```
root@ip-172-31-2-222 ~]$ jps
2200 Worker
2489 Jps
1980 Worker
1814 DataNode
2330 CoarseGrainedExecutorBackend
root@ip-172-31-2-222 ~]$ jstack 2330 | less
```

jstack

```
"Executor task launch worker-7" daemon prio=10 tid=0x00007fdd801c2800 nid=0x9b1 waiting on condition [0x00007fddd52b8000
]
   java.lang.Thread.State: TIMED_WAITING (sleeping)
        at java.lang.Thread.sleep(Native Method)
        at $line5.$read$$iwC$$iwC$$iwC$$iwC.someExpensiveComputation(<console>:11)
        at $line14.$read$$iwC$$iwC$$iwC$$iwC$$anonfun$1.apply(<console>:18)
        at $line14.$read$$iwC$$iwC$$iwC$$iwC$$anonfun$1.apply(<console>:17)
        at scala.collection.Iterator$$anon$11.next(Iterator.scala:328)
        at scala.collection.Iterator$$anon$11.next(Iterator.scala:328)
        at org.apache.spark.rdd.PairRDDFunctions.org$apache$spark$rdd$PairRDDFunctions$$writeToFile$1(PairRDDFunctions.s
cala:777)
        at org.apache.spark.rdd.PairRDDFunctions$$anonfun$saveAsHadoopDataset$2.apply(PairRDDFunctions.scala:788)
        at org.apache.spark.rdd.PairRDDFunctions$$anonfun$saveAsHadoopDataset$2.apply(PairRDDFunctions.scala:788)
        at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:111)
        at org.apache.spark.scheduler.Task.run(Task.scala:51)
        at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:187)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
        at java.lang.Thread.run(Thread.java:744)
```

Can often pinpoint problems just by “jstack” a few times

YourKit (free for open source dev)



Debugging Tip

Local Debugging

Run in local mode (i.e. Spark master “local”) and debug with your favorite debugger

- IntelliJ
- Eclipse
- println

With a sample dataset

What we have learned?

RDD abstraction

- lineage info: partitions, dependencies, compute
- optimization info: partitioner, preferred locations

Execution process (from RDD to tasks)

Performance & debugging

DATABRICKS

Thank You!