Tuning and Debugging in Apache Spark

Patrick Wendell @pwendell

February 20, 2015

databricks

About Me

Apache Spark committer and PMC, release manager

Worked on Spark at UC Berkeley when the project started

Today, managing Spark efforts at Databricks

About Databricks

Founded by creators of Spark in 2013

Donated Spark to ASF and remain largest contributor

End-to-End hosted service: Databricks Cloud

Today's Talk

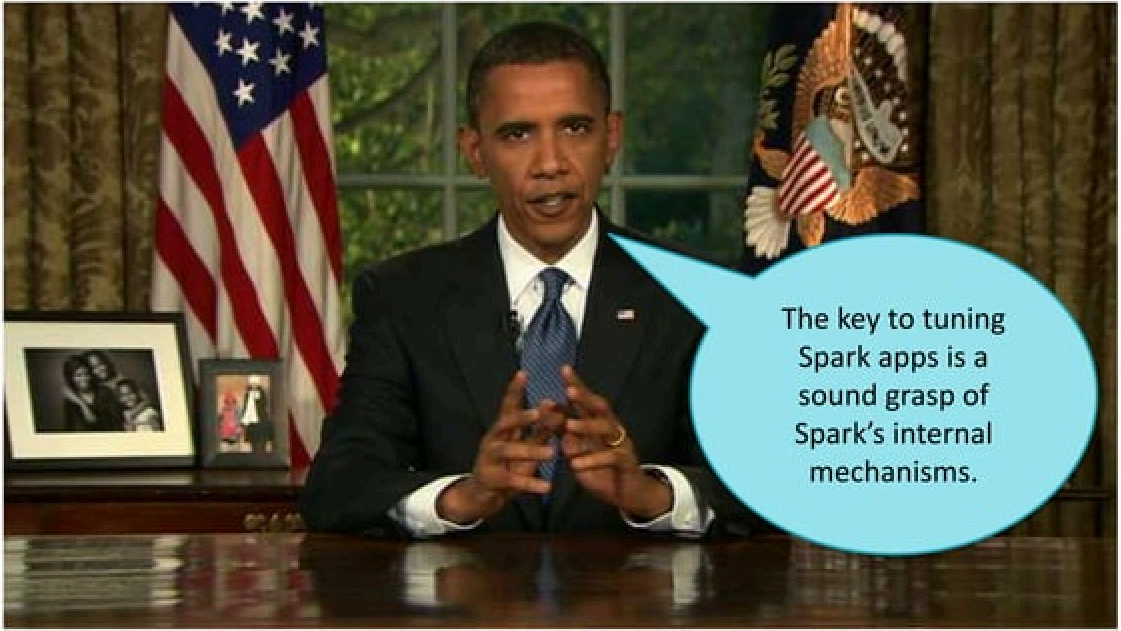
Help you understand and debug Spark programs

Related talk this afternoon:

Everyday I'm Shuffling - Tips for Writing Better Spark Programs

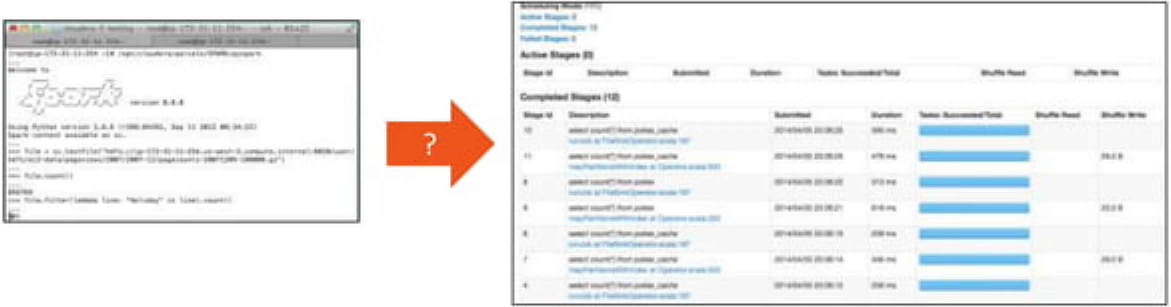
Assumes you know Spark core API concepts, focused on internals

Spark's Execution Model


The key to tuning Spark apps is a sound grasp of Spark's internal mechanisms.

Key Question

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



RDD API Refresher

RDDs are a distributed collection of records

```
rdd = spark.parallelize(range(10000), 10)
```

Transformations create new RDDs from existing ones

```
errors = rdd.filter(lambda line: "ERROR" in line)
```

Actions materialize a value in the user program

```
size = errors.count()
```

RDD API Example

```
// 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 }
```

input.txt

```
INFO Server started
INFO Bound to port 8080

WARN Cannot find srv.conf
```

databricks

RDD API Example

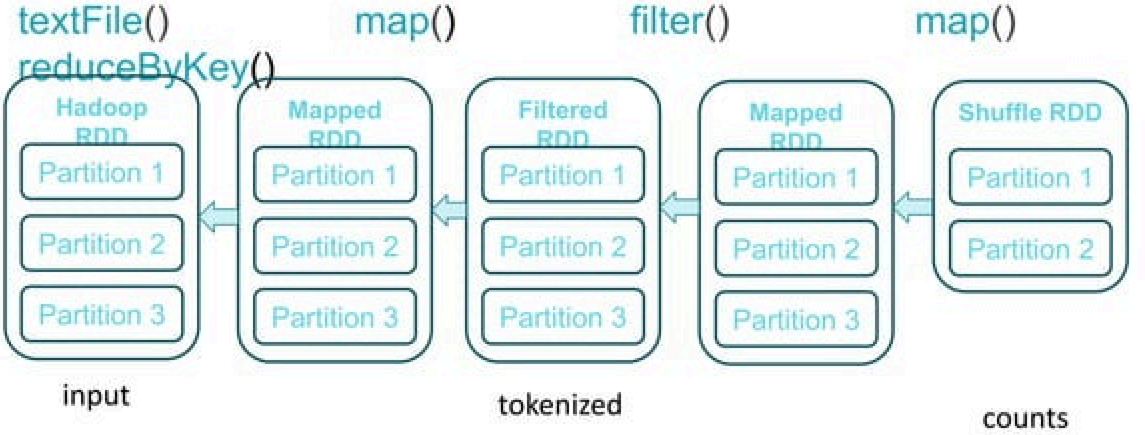
```
// Read input file
val input = sc.textFile(                )

val tokenized = input
  .map(                              )
  .filter(                                )

val counts = tokenized
  .map(                                ).
  .reduceByKey{                      }
```

databricks

Transformations

```
sc.textFile().map().filter().map().reduceByKey()
```

DAG View of RDD's



Transformations build up a DAG, but don’t “do anything”

Evaluation of the DAG

We mentioned "actions" a few slides ago. Let's forget them for a minute.

DAG's are materialized through a method sc.runJob:

```
def runJob[T, U](
        rdd: RDD[T],                          1. RDD to compute
        partitions: Seq[Int],                 2. Which partitions
        func: (Iterator[T]) => U))            3. Fn to produce results
    : Array[U]
```

databricks

Evaluation of the DAG

We mentioned "actions" a few slides ago. Let's forget them for a minute.

DAG's are materialized through a method sc.runJob:

```
def runJob[T, U](
        rdd: RDD[T],                          1. RDD to compute
        partitions: Seq[Int],                 2. Which partitions
        func: (Iterator[T]) => U))            3. Fn to produce results
```

databricks

Evaluation of the DAG

We mentioned "actions" a few slides ago. Let's forget them for a minute.

DAG's are materialized through a method sc.runJob:

```
def runJob[T, U](
        rdd: RDD[T],                              1. RDD to compute
        partitions: Seq[Int],                     2. Which partitions
        func: (Iterator[T]) => U))                3. Fn to produce results
```

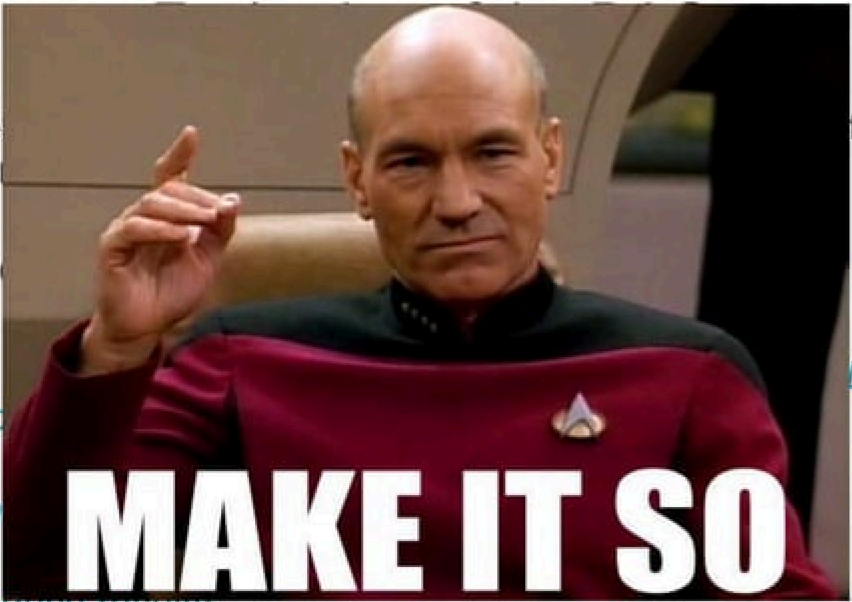
We t
the

DA

DD

to c

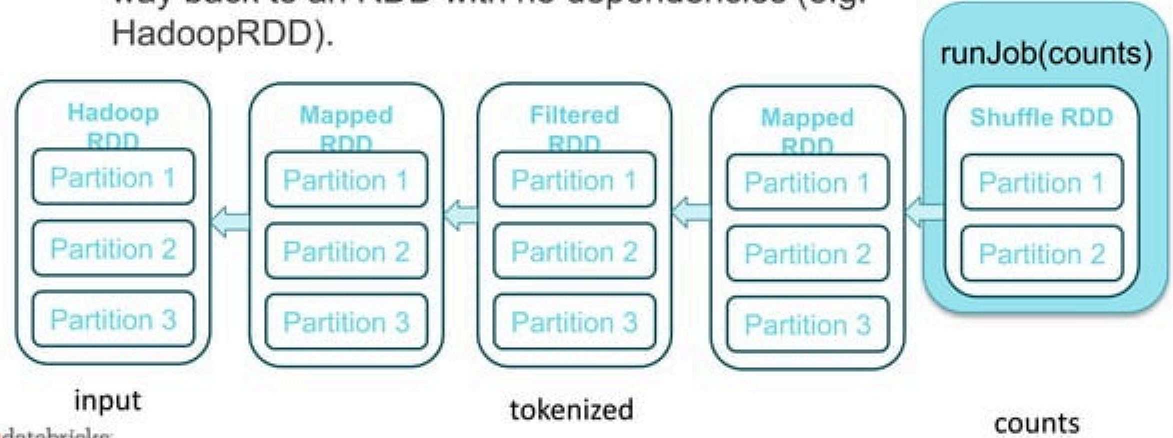
Wh



produce results

How runJob Works

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



Physical Optimizations

1. Certain types of transformations can be pipelined.

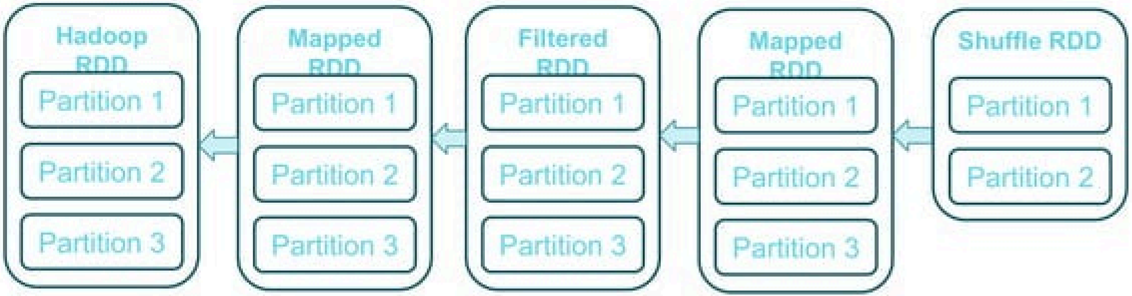
1. If dependent RDD's have already been cached (or persisted in a shuffle) the graph can be truncated.

Once pipelining and truncation occur, Spark produces a a set of *stages* each stage is composed of *tasks*

How runJob Works

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

runJob(counts)



input

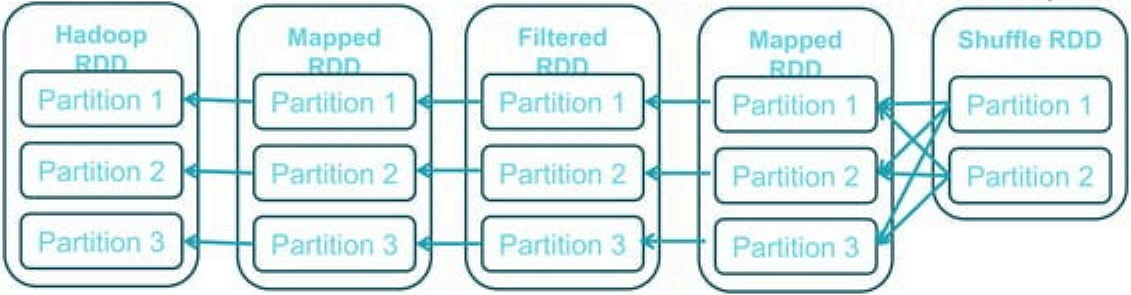
tokenized

counts

How runJob Works

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

runJob(counts)



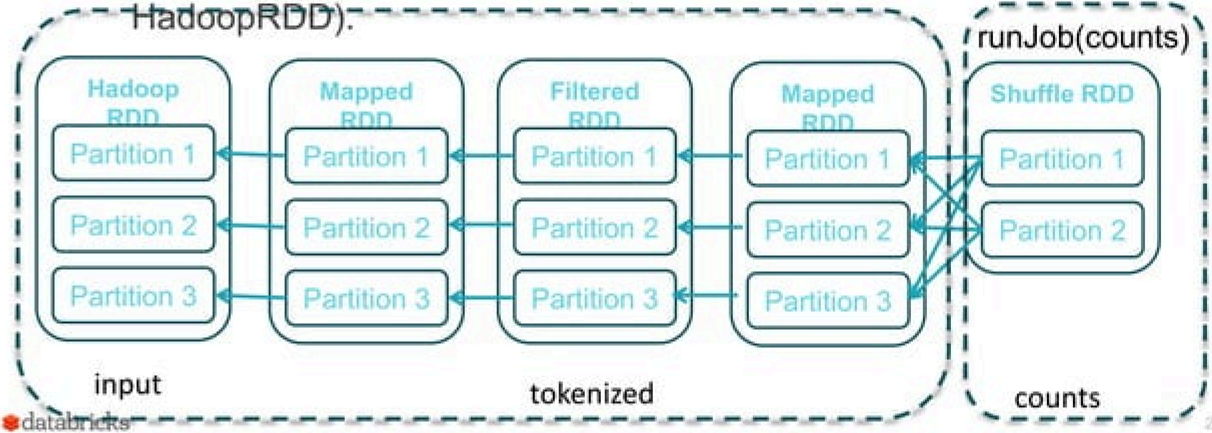
input

tokenized

counts

How runJob Works

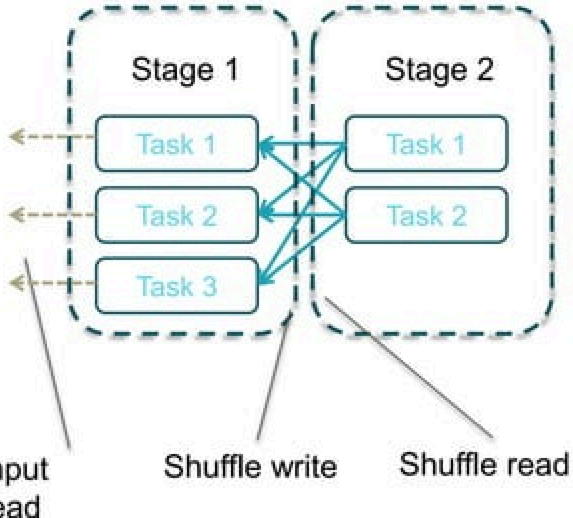
Needs to compute my parents, parents, parents, etc all the way back to an RDD with no dependencies (e.g. HadoopRDD).



Stage Graph

Each task will:

1. Read Hadoop input
2. Perform maps and filters
3. Write partial sums



Each task will:

1. Read partial sums
2. Invoke user function passed to runJob.

Units of Physical Execution

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.**

Seeing this on your own

```
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)

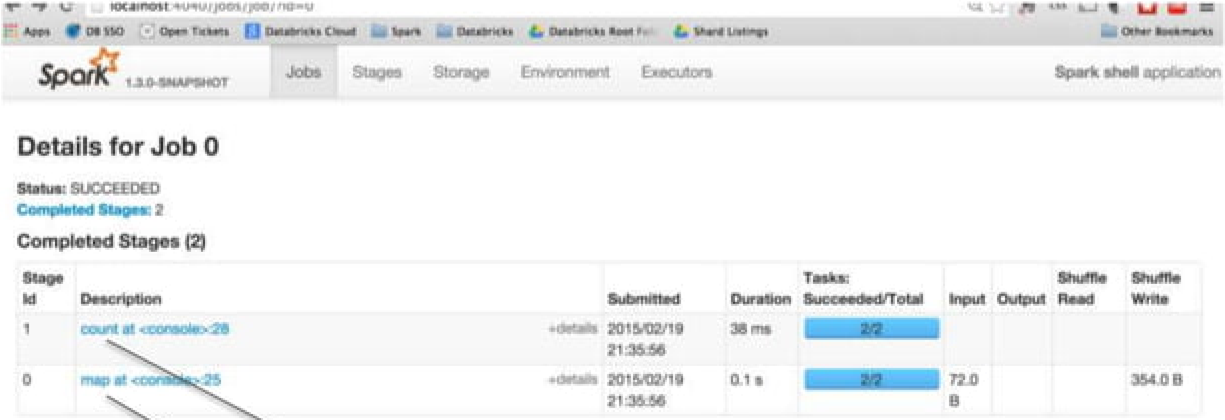
Example: *count()* action

```
class RDD {
  def count(): Long = {
     results = sc.runJob(
                this,                                   1. RDD = self
                0 until partitions.size,  2. Partitions = all partitions
                it => it.size()                         3. Function = size of the partition
     )
     return results.sum
  }
}
```

Example: *take(N)* action

```
class RDD {
 def take(n: Int) {
   val results = new ArrayBuffer[T]
   var partition = 0
   while (results.size < n) {
     result ++= sc.runJob(this, partition, it => it.toArray)
     partition = partition + 1
  }
  return results.take(n)
 }
}
```

Putting it All Together



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

Determinants of Performance in Spark

Quantity of Data Shuffled

In general, avoiding shuffle will make your program run faster.

1. Use the built in aggregateByKey() operator instead of writing your own aggregations.
2. Filter input earlier in the program rather than later.
3. Go to this afternoon's talk!

Degree of Parallelism

```
> input = sc.textFile("s3n://log-files/2014/*.log.gz") #matches thousands of files
> input.getNumPartitions()
35154

> lines = input.filter(lambda line: line.startswith("2014-10-17 08:")) # selective
> lines.getNumPartitions()
35154

> lines = lines.coalesce(5).cache() # We coalesce the lines RDD before caching
> lines.getNumPartitions()
5
>>> lines.count() # occurs on coalesced RDD
```

Degree of Parallelism

If you have a huge number of mostly idle tasks (e.g. 10's of thousands), then it's often good to coalesce.

If you are not using all slots in your cluster, repartition can increase parallelism.

Choice of Serializer

Serialization is sometimes a bottleneck when shuffling and caching data. Using the Kryo serializer is often faster.

```
val conf = new SparkConf()
conf.set("spark.serializer",
"org.apache.spark.serializer.KryoSerializer")

// Be strict about class registration
conf.set("spark.kryo.registrationRequired", "true")
conf.registerKryoClasses(Array(classOf[MyClass],
classOf[MyOtherClass]))
```

Cache Format

By default Spark will cache() data using MEMORY_ONLY level, deserialized JVM objects

MEMORY_ONLY_SER can help cut down on GC

MEMORY_AND_DISK can avoid expensive

recompuations

Hardware

Spark scales horizontally, so more is better

Disk/Memory/Network balance depends on workload: CPU intensive ML jobs vs IO intensive ETL jobs

Good to keep executor heap size to 64GB or less (can run multiple on each node)

Other Performance Tweaks

Switching to LZF compression can improve shuffle performance (sacrifices some robustness for massive shuffles):

```
conf.set("spark.io.compression.codec", "lzf")
```

Turn on speculative execution to help prevent stragglers

```
conf.set("spark.speculation", "true")
```

Other Performance Tweaks

Make sure to give Spark as many disks as possible to allow striping shuffle output

- SPARK_LOCAL_DIRS in Mesos/Standalone
- In YARN mode, inherits YARN's local directories

One Weird Trick for Great Performance

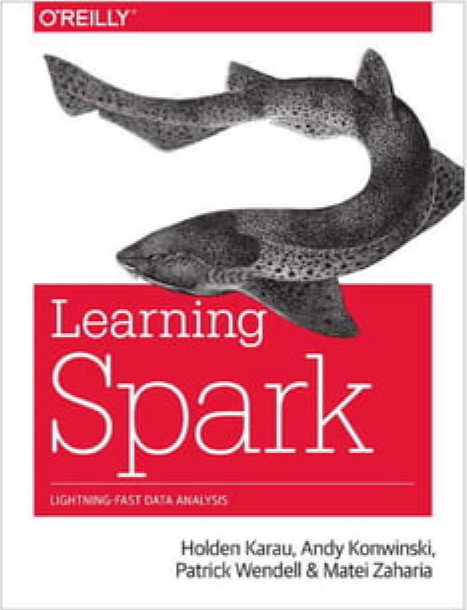
Use Higher Level API's!

DataFrame APIs for core processing

- Works across Scala, Java, Python and R

Spark ML for machine learning

Spark SQL for structured query processing



See also

Chapter 8: Tuning and Debugging Spark.

Come to Spark Summit 2015!

June 15-17 in San Francisco

Other Spark Happenings Today

Spark team “Ask Us Anything” at 2:20 in 211 B

Tips for writing better Spark programs at 4:00 in 230C

I’ll be around Databricks booth after this

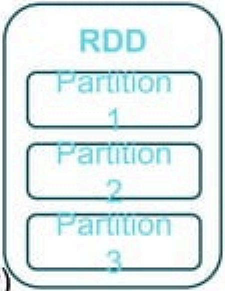
Thank you.
Any questions?

Extra Slides

databricks

Internals of the RDD Interface

1) List of partitions
2) Set of dependencies on parent RDDs
3) Function to compute a partition, given parents
4) Optional partitioning info for k/v RDDs (Partitioner)



Example: Hadoop RDD

Partitions = 1 per HDFS block

Dependencies = None

compute(partition) = read corresponding HDFS block

Partitioner = None

> rdd = spark.hadoopFile(“hdfs://click_logs/”)

Example: Filtered RDD

Partitions = parent partitions

Dependencies = a single parent

compute(partition) = call parent.compute(partition) and filter

Partitioner = parent partitioner

> filtered = rdd.filter(lambda x: x contains "ERROR")

Example: Joined RDD

Partitions = number chosen by user or heuristics

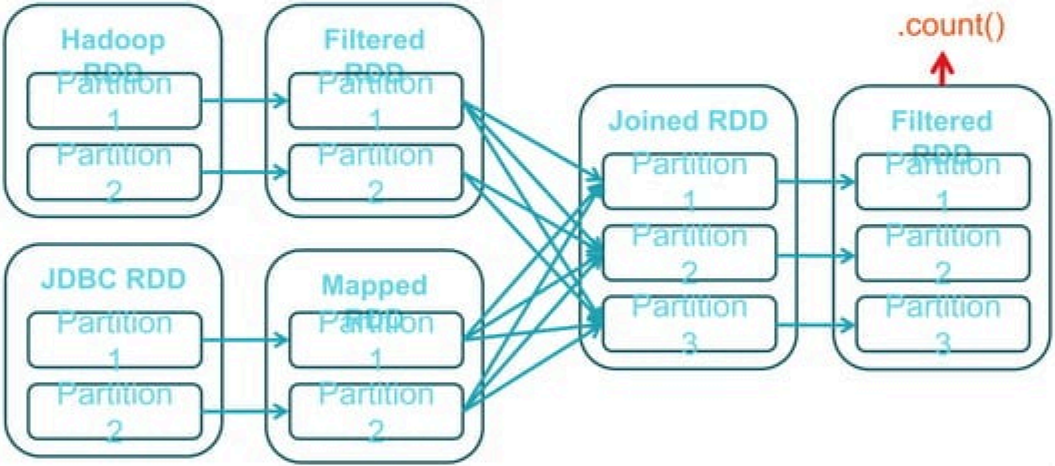
Dependencies = ShuffleDependency on two or more

parents

compute(partition) = read and join data from all parents

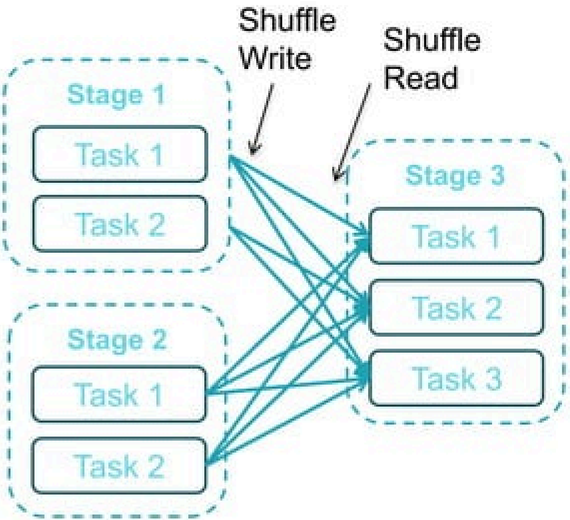
Partitioner = HashPartitioner(# partitions)

A More Complex DAG

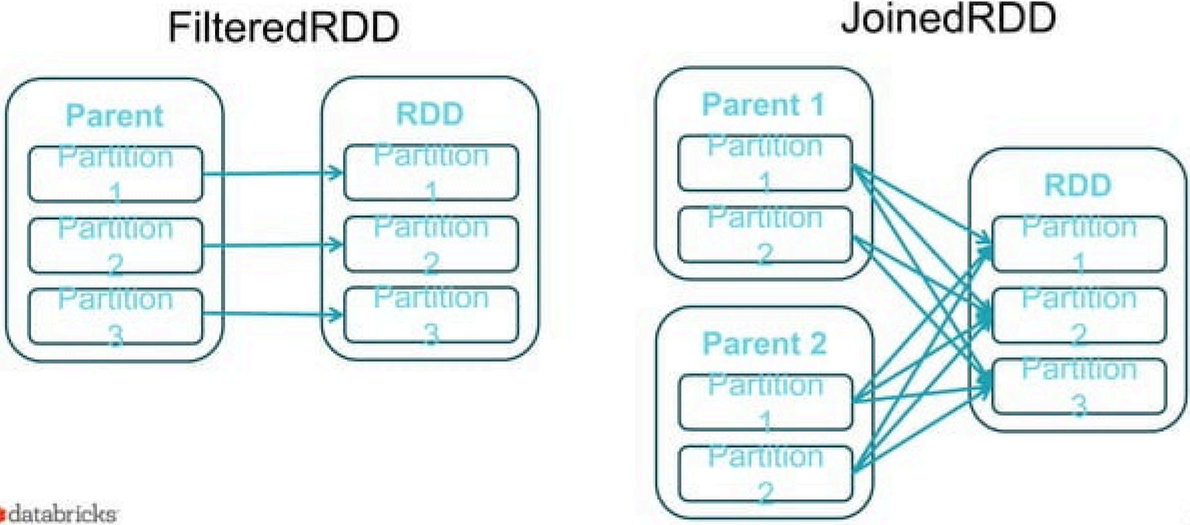


databricks

A More Complex DAG



Narrow and Wide Transformations



databricks