

Spark

MODIFIED BASED ON THE SLIDES FROM MATEI ZAHARIA (BERKELEY)
AND PATRICK WENDELL (FROM DATABRICKS)

Apache Spark Tutorial:

https://www.tutorialspoint.com/apache_spark/index.htm

The Google Pagerank Algorithm and How It Works, Ian Rogers

<https://www.cs.princeton.edu/~chazelle/courses/BIB/pagerank.htm>

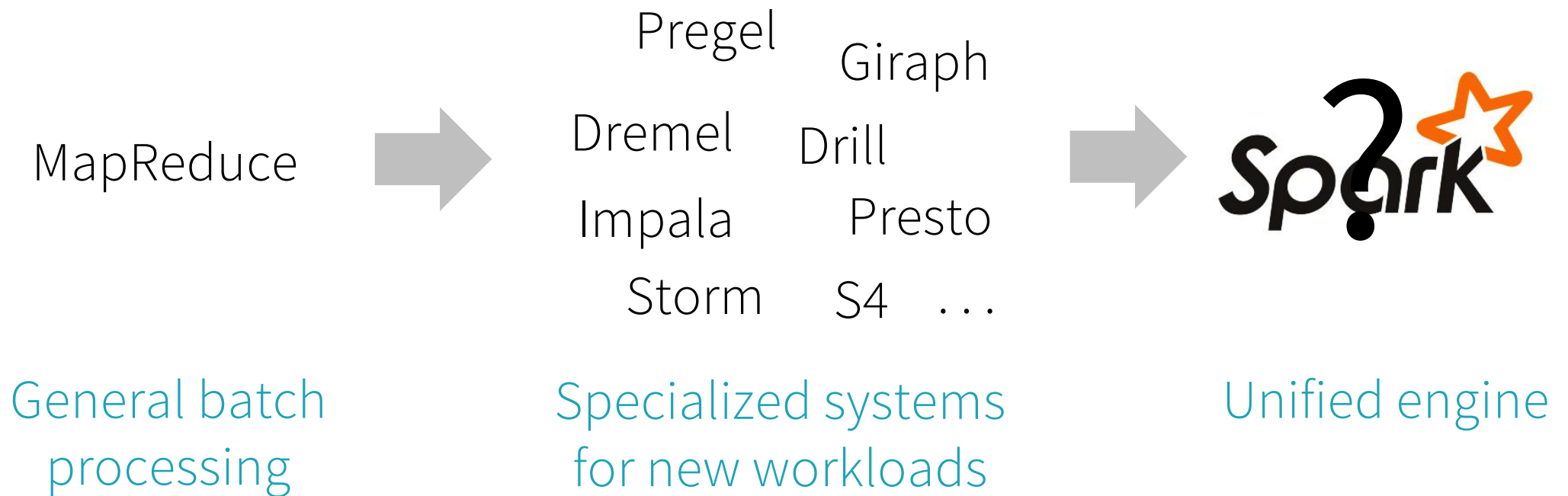
Section “Other Considerations” of Tuning Spark.

<https://spark.apache.org/docs/latest/tuning.html>

Understanding the Performance of Spark Applications. Patrick

Mendell. <https://databricks.com/session/understanding-the-performance-of-spark-applications>

Motivation: Unification



How to Run It

Local multicore: just a library in your program

EC2: scripts for launching a Spark cluster

Private cluster: Mesos, YARN, Standalone Mode

Languages

APIs in Java, Scala and Python

Interactive shells in Scala and Python

- Java developers: consider using Scala for console (to learn the API)

Performance:

- Java / Scala will be faster (statically typed), but Python can do well for numerical work with NumPy

Key Idea

Work with distributed collections as you would with local ones

Concept: resilient distributed datasets (RDDs)

- Immutable collections of objects spread across a cluster
- Built through parallel transformations (map, filter, etc)
- Automatically rebuilt on failure
- Controllable persistence (e.g. caching in RAM)

Operations

Transformations (e.g. map, filter, groupBy, join)

- Lazy operations to build RDDs from other RDDs

Actions (e.g. count, collect, save)

- Return a result or write it to storage

Spark Operations

Transformations (define a new RDD)	map filter sample groupByKey reduceByKey sortByKey	flatMap union join cogroup cross mapValues
Actions (return a result to driver program)	collect reduce count save lookupKey	

Example: Mining Console Logs

Load error messages from a log into memory, then interactively search for patterns

```
lines = spark.textFile("hdfs://...")  
errors = lines.filter(lambda s: s.startswith("ERROR"))  
messages = errors.map(lambda s: s.split('\t')[2])  
messages.cache()
```

Base RDD

Transformed RDD

Driver

Worker

Worker

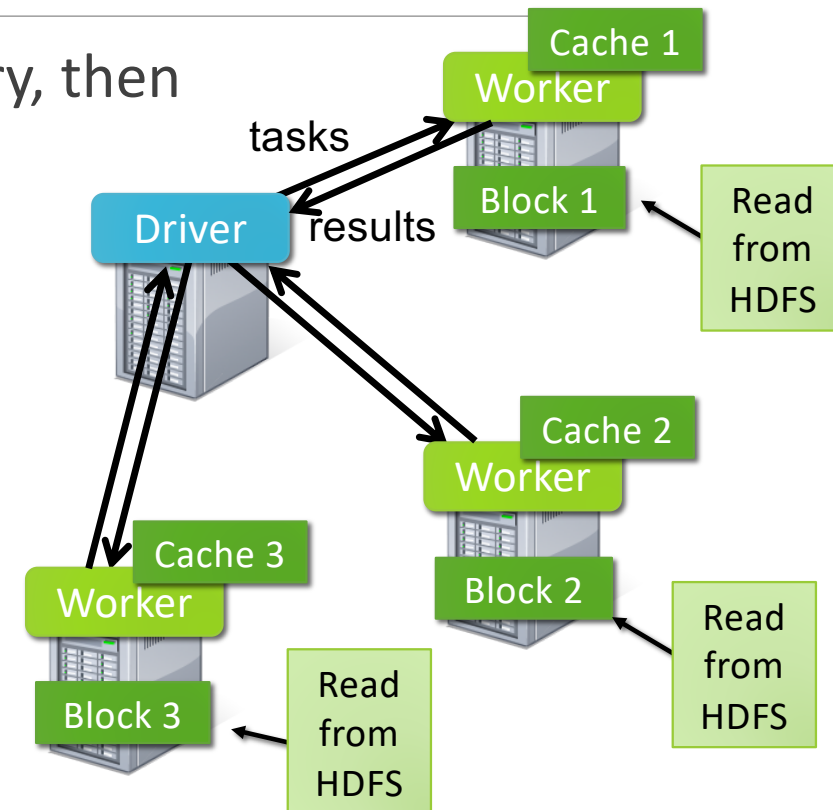
Worker

Example: Mining Console Logs

Load error messages from a log into memory, then interactively search for patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split('\t')[2])
messages.cache()

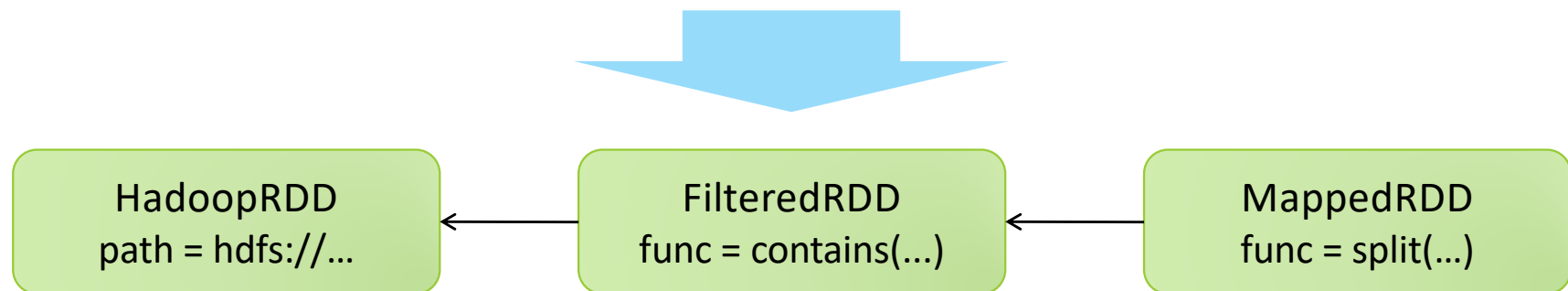
messages.filter(lambda s: "foo" in s).count()
messages.filter(lambda s: "bar" in s).count()
. . .
```



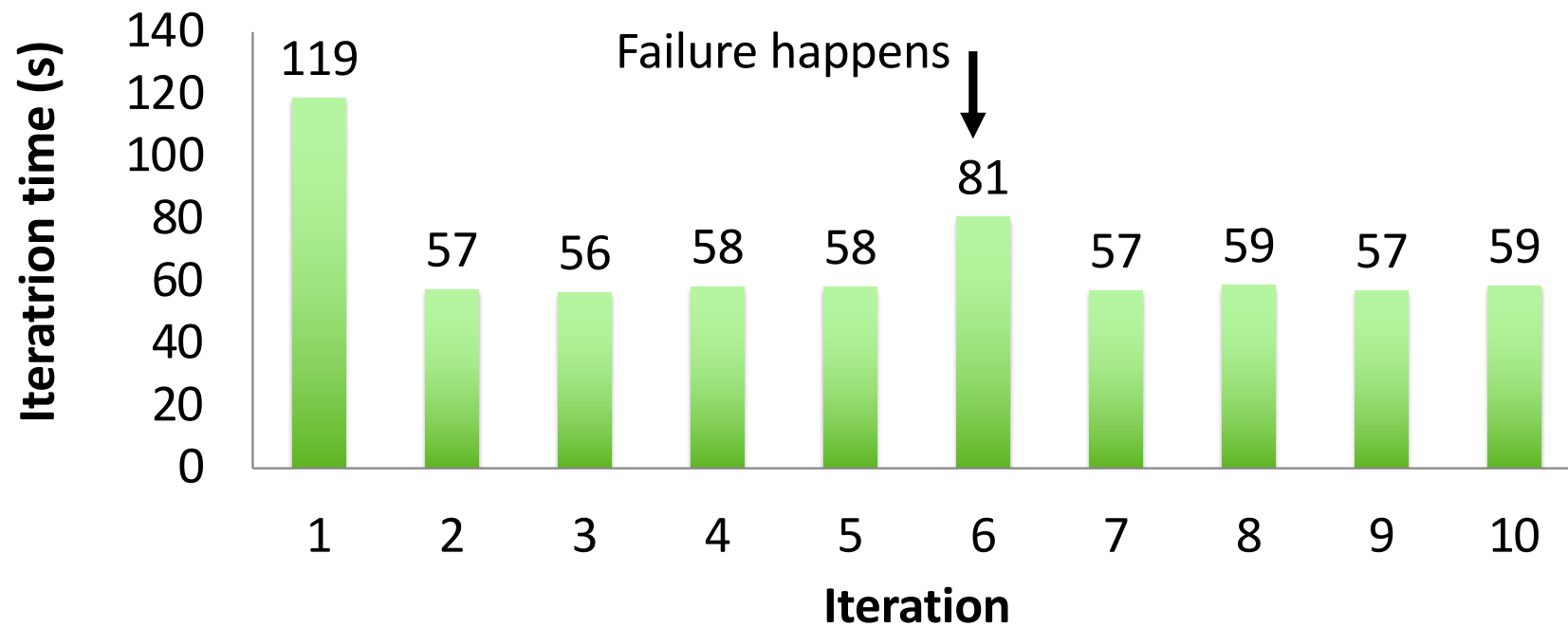
RDD Fault Tolerance

RDDs track the transformations used to build them (their lineage) to recompute lost data

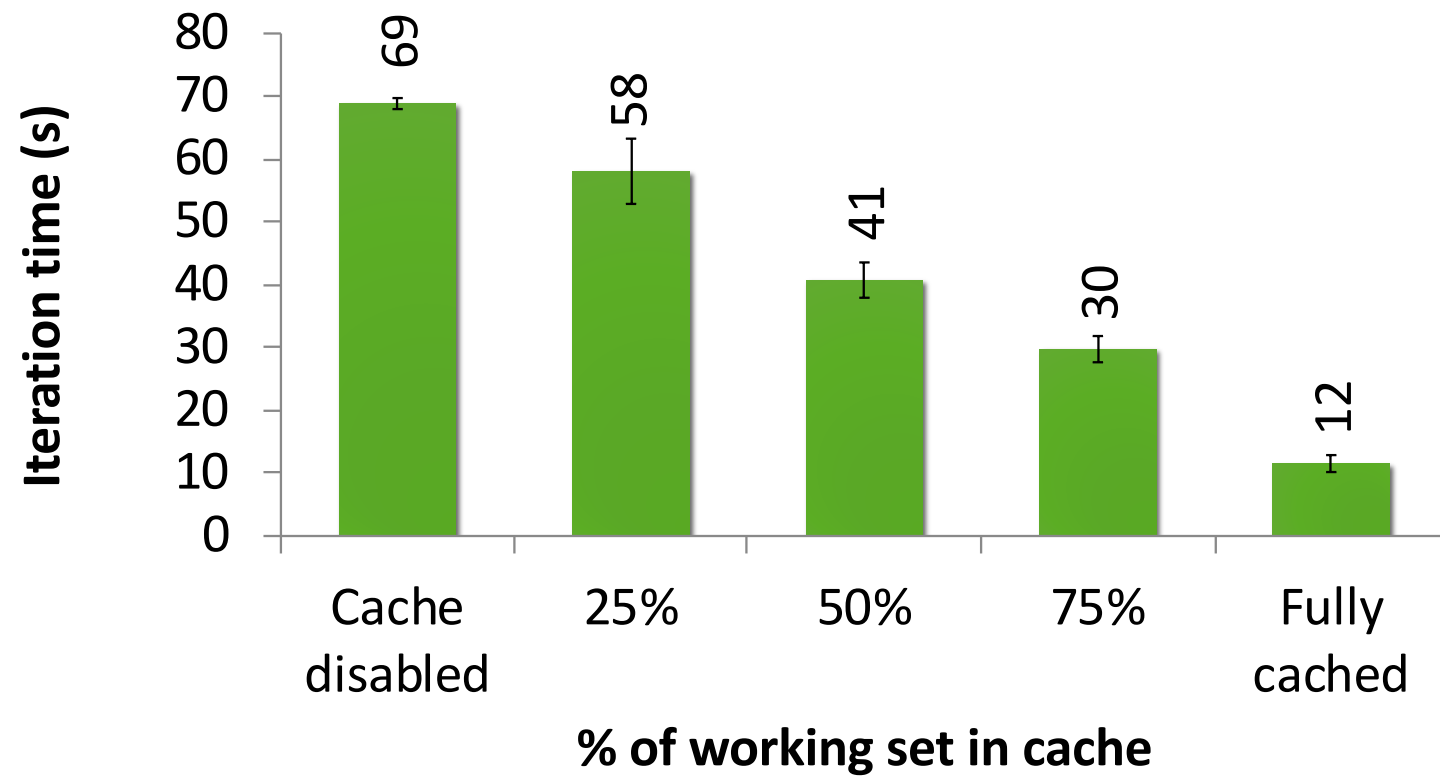
E.g: `messages = textFile(...).filter(lambda s: s.contains("ERROR"))
.map(lambda s: s.split('\t')[2])`



Fault Recovery Test



Behavior with Less RAM



Spark in Java

```
JavaRDD<String> lines = spark.textFile(...);  
errors = lines.filter(s ->s.contains("ERROR"));  
errors.count()
```

Deployment

Runs in local mode on 1 thread by default, but can control with MASTER environment var:

```
MASTER=local ./spark-shell      # local, 1 thread  
MASTER=local[2] ./spark-shell    # local, 2 threads  
MASTER=spark://host:port ./spark-shell # Spark standalone cluster
```

First Stop: SparkContext

Main entry point to Spark functionality

Created for you in Spark shells as variable `sc`

In standalone programs, you'd make your own (see later for details)

Creating RDDs

Turn a local collection into an RDD

```
sc.parallelize([1, 2, 3])
```

Load text file from local FS, HDFS, or S3

```
sc.textFile("file.txt")
```

```
sc.textFile("directory/*.txt")
```

```
sc.textFile("hdfs://namenode:9000/path/file")
```

Use any existing Hadoop InputFormat

```
sc.hadoopFile(keyClass, valClass, inputFmt, conf)
```

Basic Transformations

```
nums = sc.parallelize([1, 2, 3])
```

```
# Pass each element through a function
```

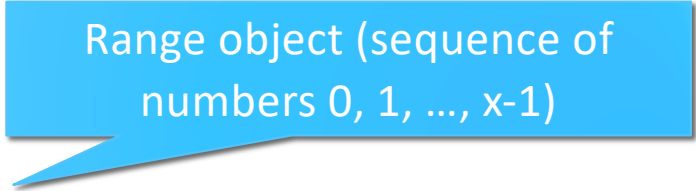
```
squares = nums.map(x -> x*x) # => {1, 4, 9}
```

```
# Keep elements passing a predicate
```

```
even = squares.filter(x -> x % 2 == 0) # => {4}
```

```
# Map each element to zero or more others
```

```
nums.flatMap(x -> range(0, x)) # => {0, 0, 1, 0, 1, 2}
```



Range object (sequence of numbers 0, 1, ..., x-1)

Basic Actions

```
nums = sc.parallelize([1, 2, 3])  
# Retrieve RDD contents as a local collection  
nums.collect() # => [1, 2, 3]  
# Return first K elements  
nums.take(2) # => [1, 2]  
# Count number of elements  
nums.count() # => 3  
# Merge elements with an associative function  
nums.reduce((x, y) -> x + y) # => 6  
# Write elements to a text file  
nums.saveAsTextFile("hdfs://file.txt")
```

Working with Key-Value Pairs

Spark's “distributed reduce” transformations act on RDDs of key-value pairs

Java: Tuple2 pair = new Tuple2(a, b); // class scala.Tuple2

- pair._1 // => a
- pair._2 // => b

Some Key-Value Operations

```
pets = sc.parallelize([("cat", 1), ("dog", 1), ("cat", 2)])
```

```
pets.reduceByKey((x, y) -> x + y)  
# => {(cat, 3), (dog, 1)}
```

```
pets.groupByKey()  
# => {(cat, Seq(1, 2)), (dog, Seq(1))}
```

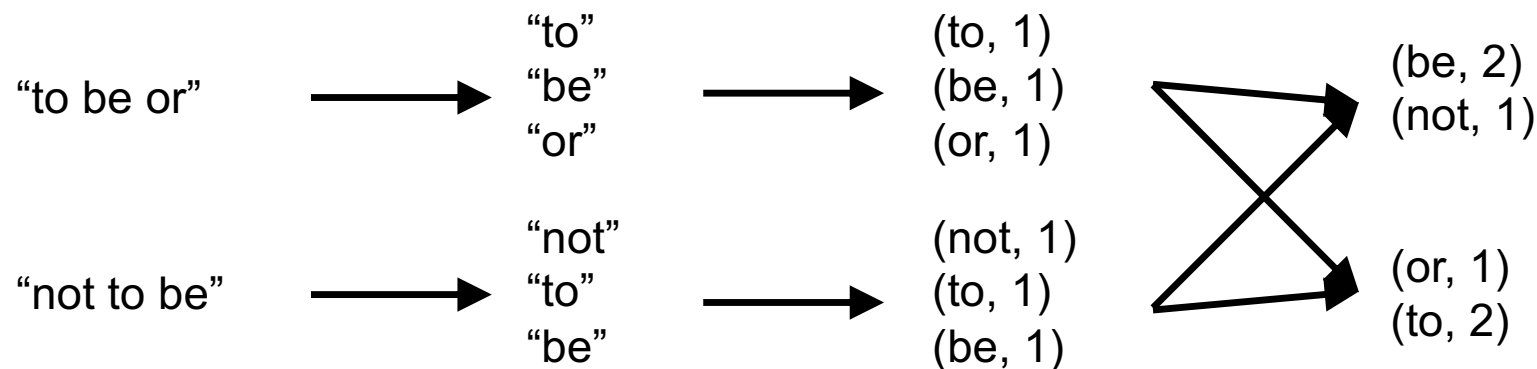
```
pets.sortByKey()  
# => {(cat, 1), (cat, 2), (dog, 1)}
```

reduceByKey also automatically implements combiners on the map side

Example: Word Count

```
lines = sc.textFile("hamlet.txt")
```

```
counts = lines.flatMap(s -> Arrays.asList(s.split(" ")).iterator())  
               .mapToPair(s -> new Tuple2<>(s, 1))  
               .reduceByKey((x, y) -> x + y)
```



Multiple Datasets

```
visits = sc.parallelize([("index.html", "1.2.3.4"),
                        ("about.html", "3.4.5.6"),
                        ("index.html", "1.3.3.1")])

pageNames = sc.parallelize([("index.html", "Home"), ("about.html", "About")])

visits.join(pageNames)
# ("index.html", ("1.2.3.4", "Home"))
# ("index.html", ("1.3.3.1", "Home"))
# ("about.html", ("3.4.5.6", "About"))

visits.cogroup(pageNames)
# ("index.html", (Seq("1.2.3.4", "1.3.3.1"), Seq("Home")))
# ("about.html", (Seq("3.4.5.6"), Seq("About")))
```

Controlling the Level of Parallelism

All the pair RDD operations take an optional second parameter for number of tasks

- `words.reduceByKey((x, y) -> x + y, 5)`
- `words.groupByKey(5)`
- `visits.join(pageViews, 5)`

Using Local Variables

External variables you use in a closure will automatically be shipped to the cluster:

- `query = raw_input("Enter a query:")`
- `pages.filter(x -> x.startswith(query)).count()`

Some caveats:

- Each task gets a new copy (updates aren't sent back)
- Variable must be Serializable (Java/Scala) or Pickle-able (Python)
- Don't use fields of an outer object (ships all of it!)

Closure Mishap Example

```
class MyCoolRddApp {  
  val param = 3.14  
  val log = new Log(...)  
  ...  
}
```

```
def work(rdd: RDD[Int]) {  
  rdd.map(x -> x + param)  
    .reduce(...)  
}
```

NotSerializableException:
MyCoolRddApp (or Log)

How to get around it:

```
class MyCoolRddApp {  
  ...  
}
```

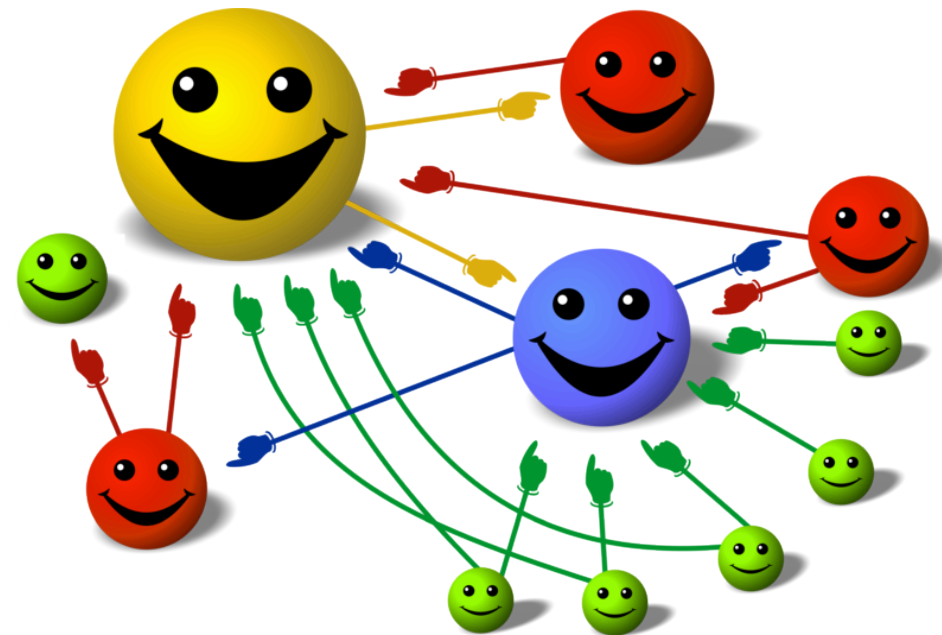
```
def work(rdd: RDD[Int]) {  
  val param_ = param  
  rdd.map(x -> x + param_)  
    .reduce(...)  
}
```

References only local variable
instead of this.param

An Example: Page Rank

Give pages ranks (scores) based on links to them

- Links from many pages → high rank
- Link from a high-rank page → high rank



Algorithm

Formula:

- $PR(A) = (1-d) + d (PR(T1)/C(T1) + \dots + PR(Tn)/C(Tn))$

Citations from Google's paper:

- *"We assume page A has pages T1...Tn which point to it (i.e., are citations).*
- *The parameter d is a damping factor which can be set between 0 and 1.*
 - *We usually set d to 0.85.*
- *Also C(A) is defined as the number of links going out of page A."*

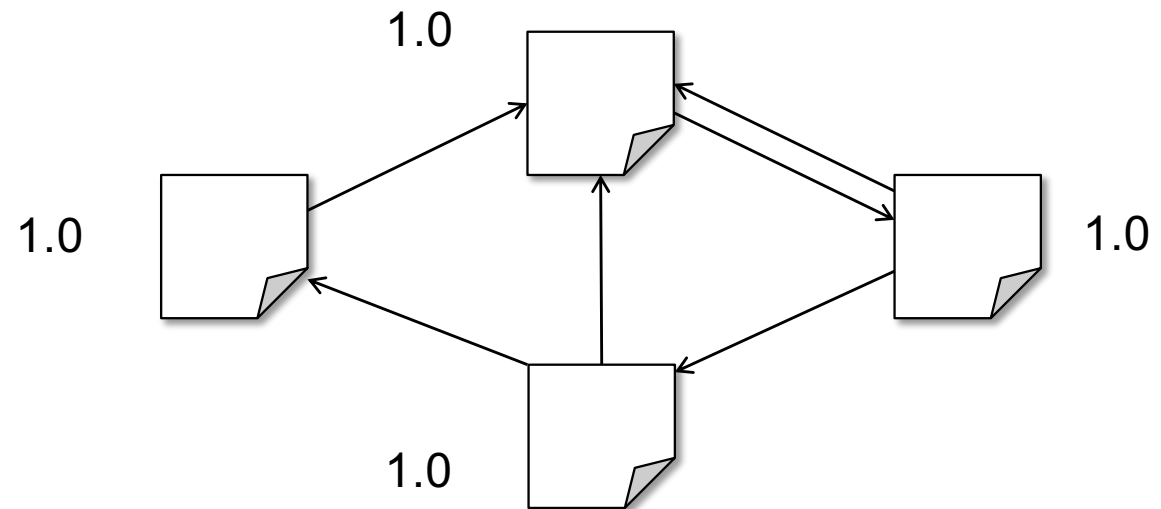
- *"The PageRanks form a probability distribution over web pages, so the sum of all web pages' PageRanks will be one."*

Algorithm

Start each page at a rank of 1

On each iteration, have page p contribute $\text{rank}_p / |\text{neighbors}_p|$ to its neighbors

Set each page's rank to $0.15 + 0.85 \times \text{contribs}$

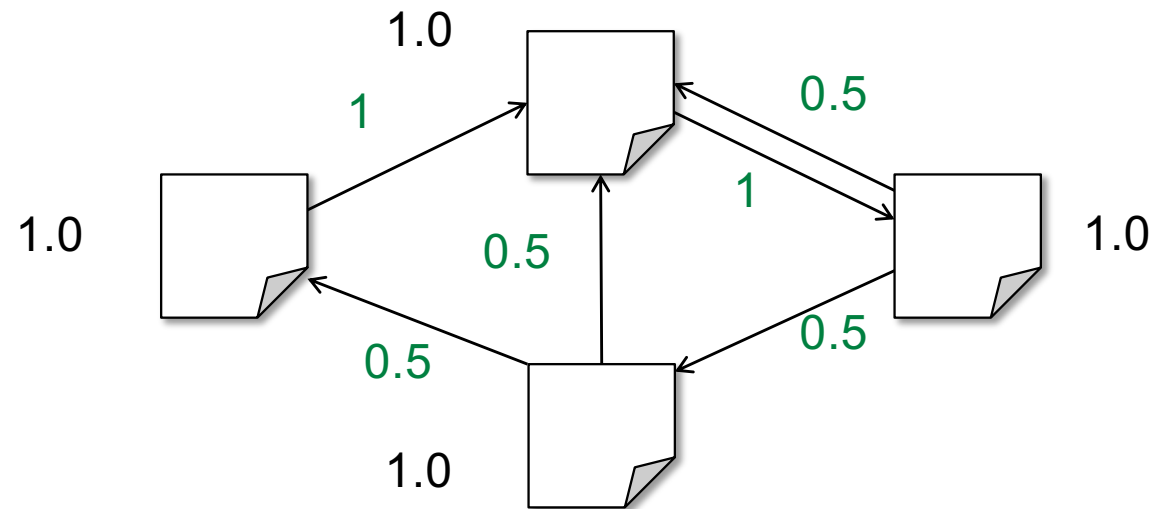


Algorithm

Start each page at a rank of 1

On each iteration, have page p contribute $\text{rank}_p / |\text{neighbors}_p|$ to its neighbors

Set each page's rank to $0.15 + 0.85 \times \text{contribs}$

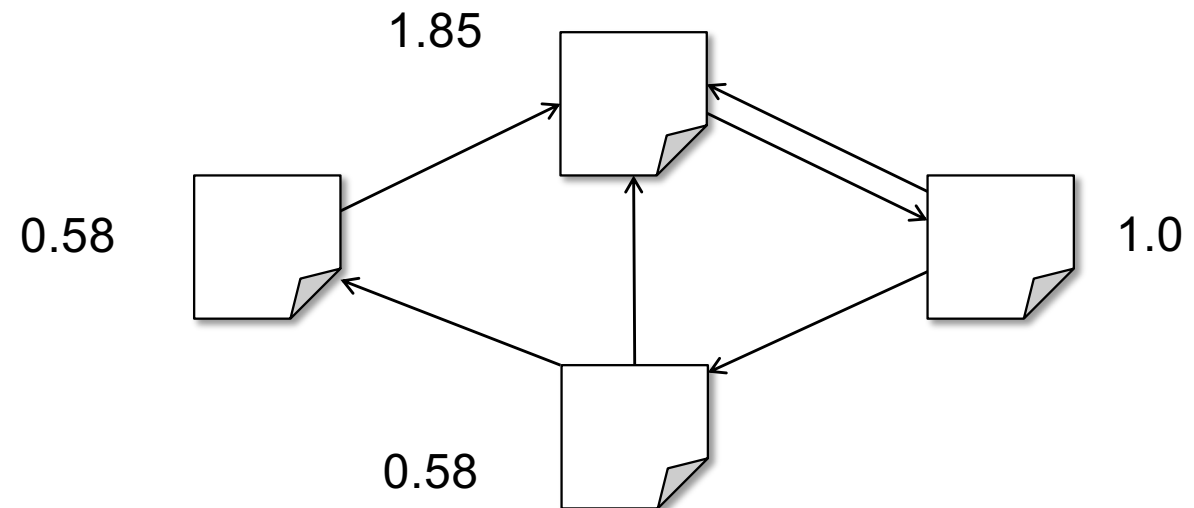


Algorithm

Start each page at a rank of 1

On each iteration, have page p contribute $\text{rank}_p / |\text{neighbors}_p|$ to its neighbors

Set each page's rank to $0.15 + 0.85 \times \text{contribs}$

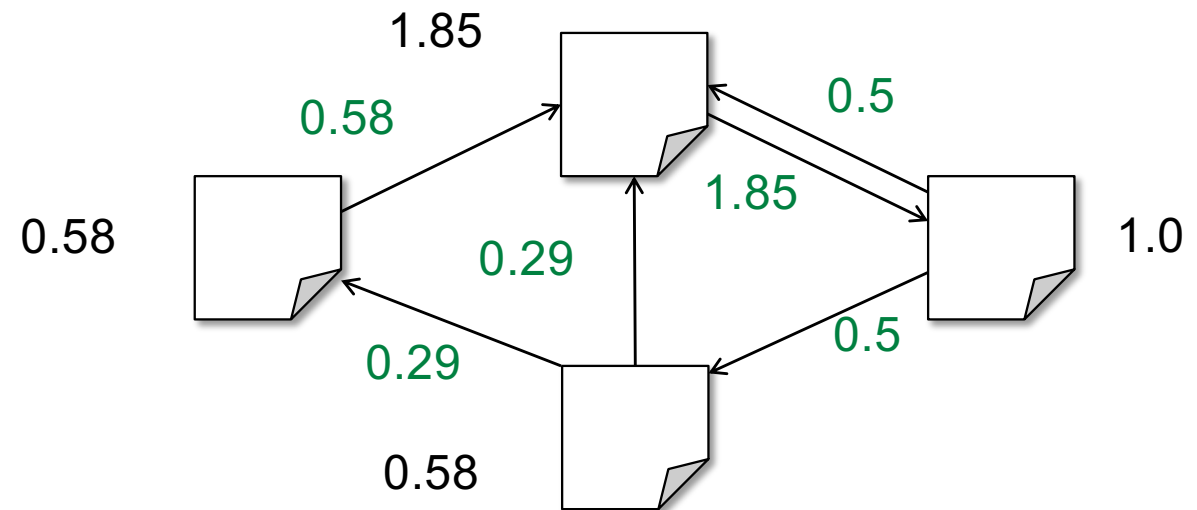


Algorithm

Start each page at a rank of 1

On each iteration, have page p contribute $\text{rank}_p / |\text{neighbors}_p|$ to its neighbors

Set each page's rank to $0.15 + 0.85 \times \text{contribs}$

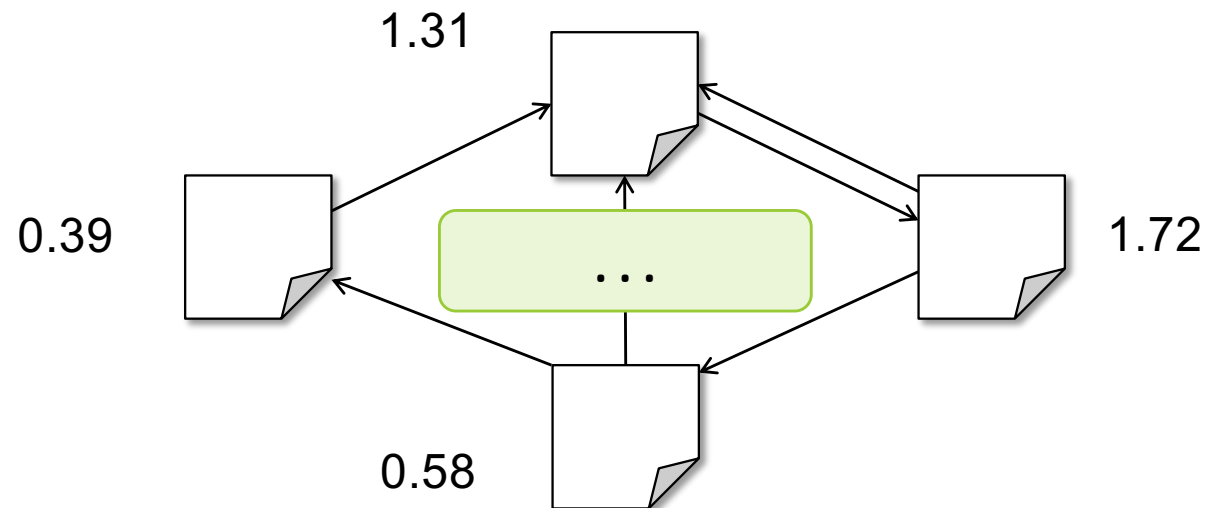


Algorithm

Start each page at a rank of 1

On each iteration, have page p contribute $\text{rank}_p / |\text{neighbors}_p|$ to its neighbors

Set each page's rank to $0.15 + 0.85 \times \text{contribs}$



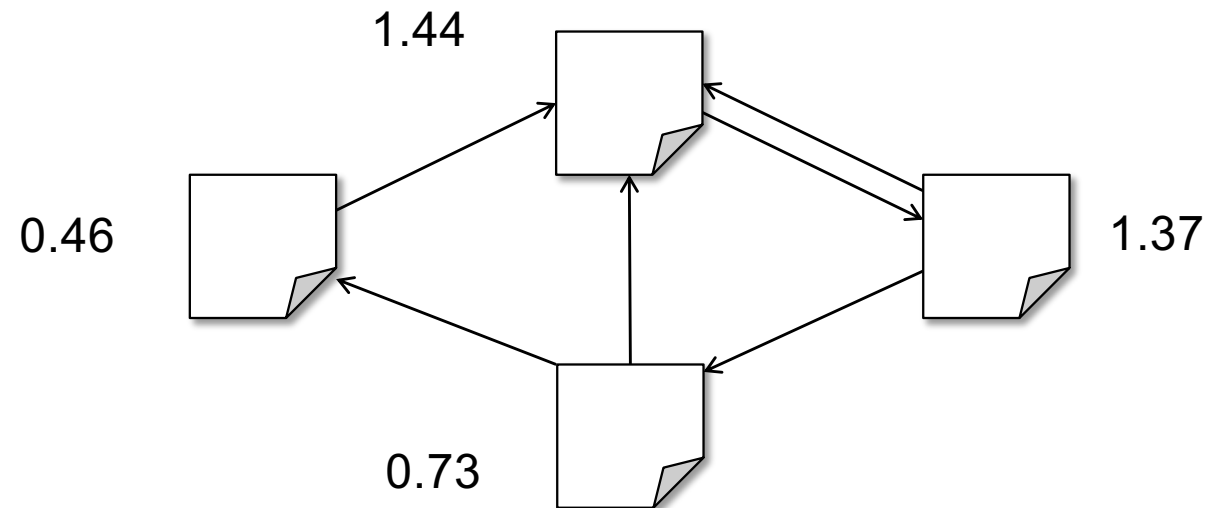
Algorithm

Start each page at a rank of 1

On each iteration, have page p contribute $\text{rank}_p / |\text{neighbors}_p|$ to its neighbors

Set each page's rank to $0.15 + 0.85 \times \text{contribs}$

Final state:



Scala Implementation

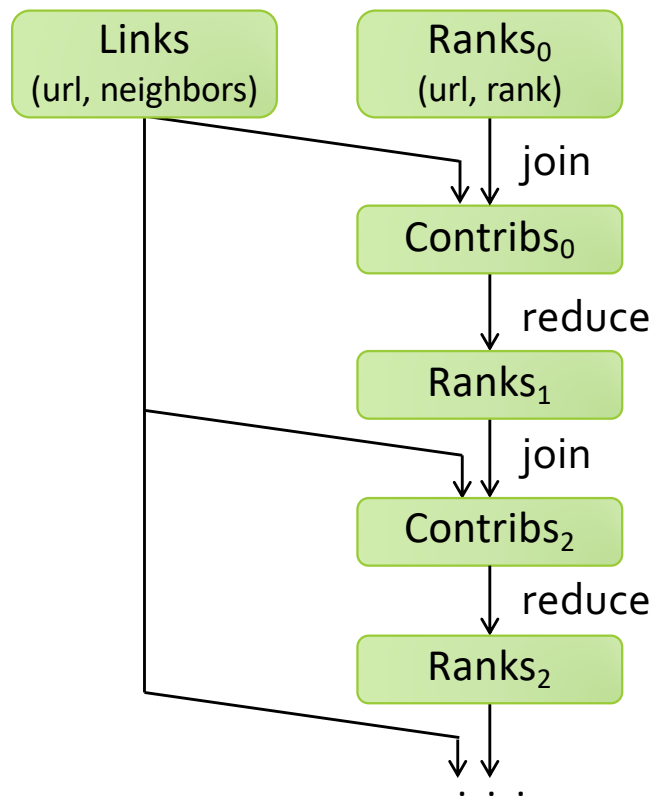
```
val links = // RDD of (url, neighbors) pairs
var ranks = // RDD of (url, rank) pairs

for (i <- 1 to ITERATIONS) {
  val contribs = links.join(ranks).flatMap {
    case (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }

  ranks = contribs.reduceByKey(_ + _).mapValues(0.15 + 0.85 * _)
}

ranks.saveAsTextFile(...)
```

Optimizing Placement



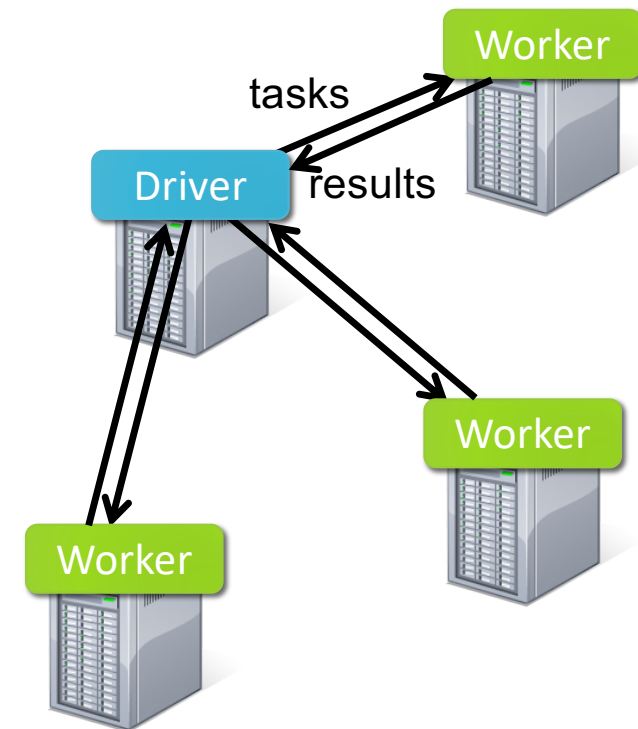
links & ranks repeatedly joined

Is this a problem?

- Yes, let's see why

Spark Execution Model

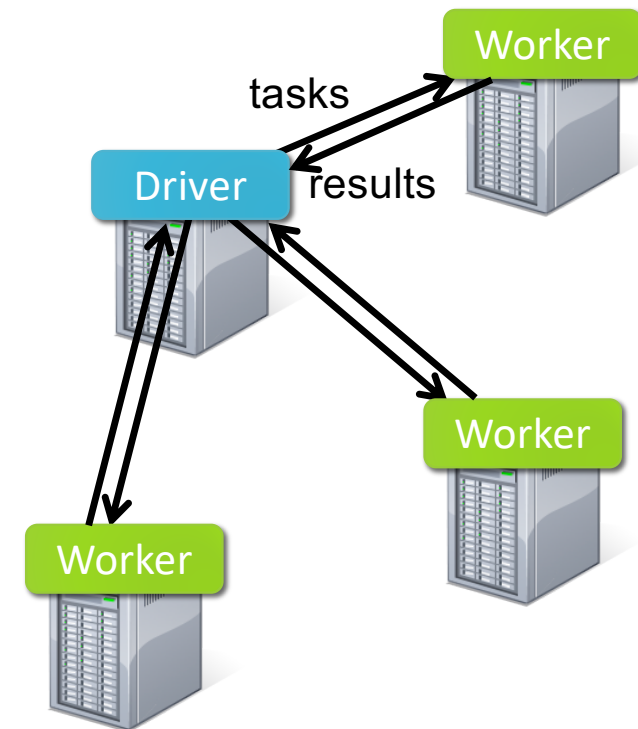
1. Driver program connects to a cluster manager to allocate resources
2. Acquires Executors (workers) on cluster nodes – run parallel tasks/cache data
3. Sends application code to workets
4. Sends tasks for executors to run



Spark Execution Model

RDDs are evaluated across the executors in partitions

Each worker can have multiple partitions, but a partition cannot be spread across multiple workers.



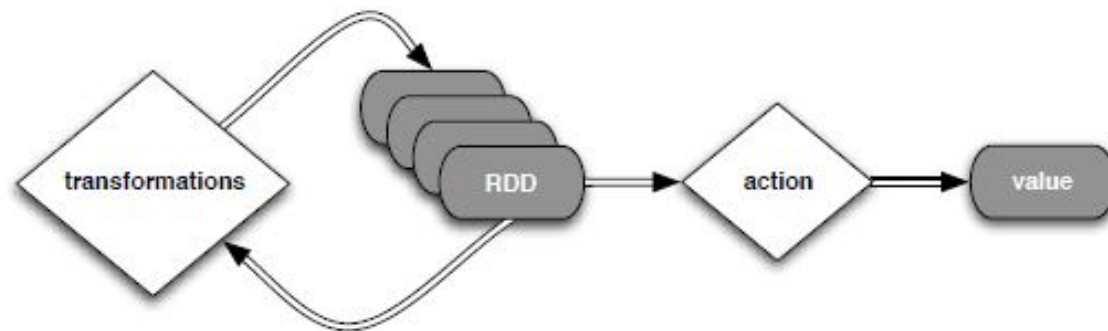
Spark Essentials – RDD Lineage

There are two types of operations on RDDs:

- Transformations (lazy – not computed immediately)
- Actions (return a value to app, or storage)

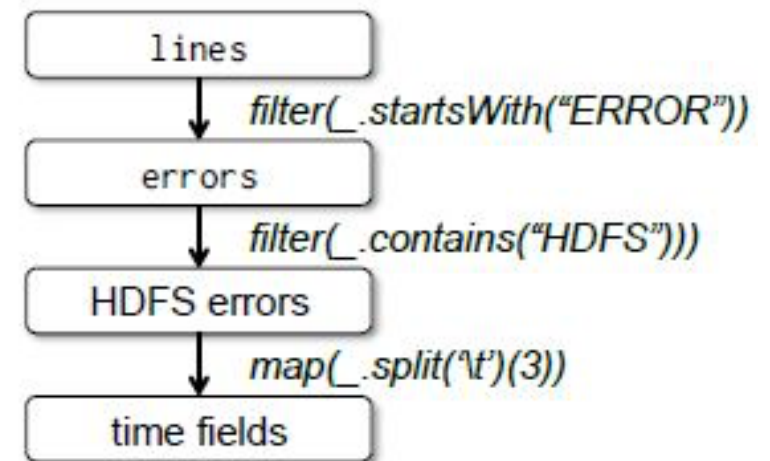
A transformed RDD is recomputed when an action runs on it.

But it can be persisted into memory or disk.



Spark Essentials – RDD Lineage

```
1. lines = spark.textFile("hdfs://..")
2. errors = lines.filter(_.startsWith("ERROR"))
3. errors.persist()
4. errors.count()
5. errors.filter(_.contains("HDFS"))
   .map(_.split('\t')(3))
   .collect()
```



Lineage is based on course-grained transformations

Lineage helps avoid the overhead from checkpointing RDDs

How to represent RDDs

Need to track lineage across a wide range of transformations

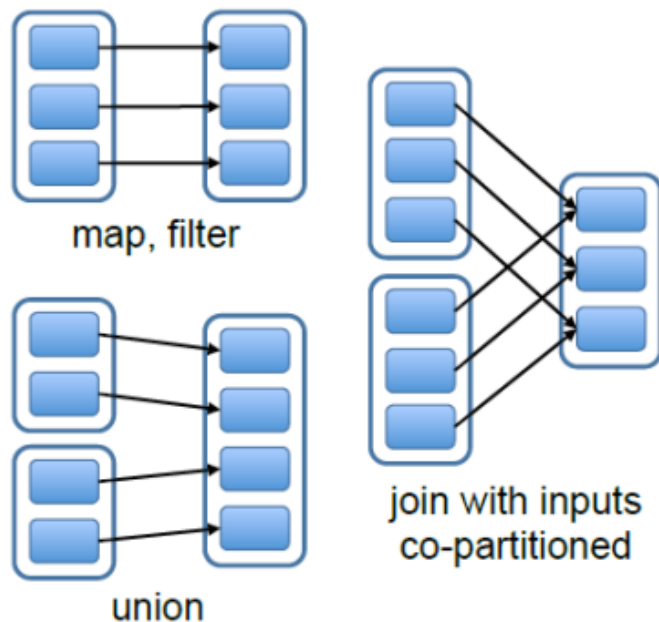
A graph-based representation

A common interface w/:

- a set of partitions: atomic pieces of the dataset
- a set of dependencies on parent RDDs
- a function for computing the dataset based on its parents
- metadata about its partitioning scheme and data placement

RDD Dependencies

Narrow dependencies: each partition of the parent RDD is used by at most one partition of the child RDD

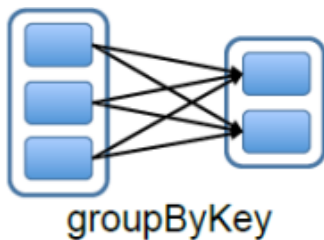


Good for pipelined execution on one cluster node, e.g., apply a map followed by a filter on an element-by-element basis.

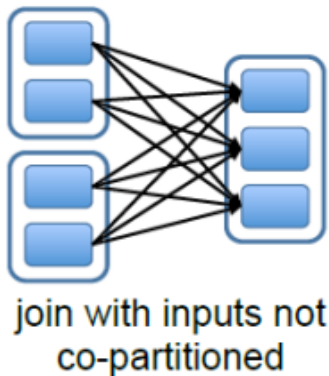
Efficient recovery as only the lost parent partitions need to be recomputed (in parallel on different nodes).

RDD Dependencies

Wide dependencies: each partition of the parent RDD is used by multiple child partitions

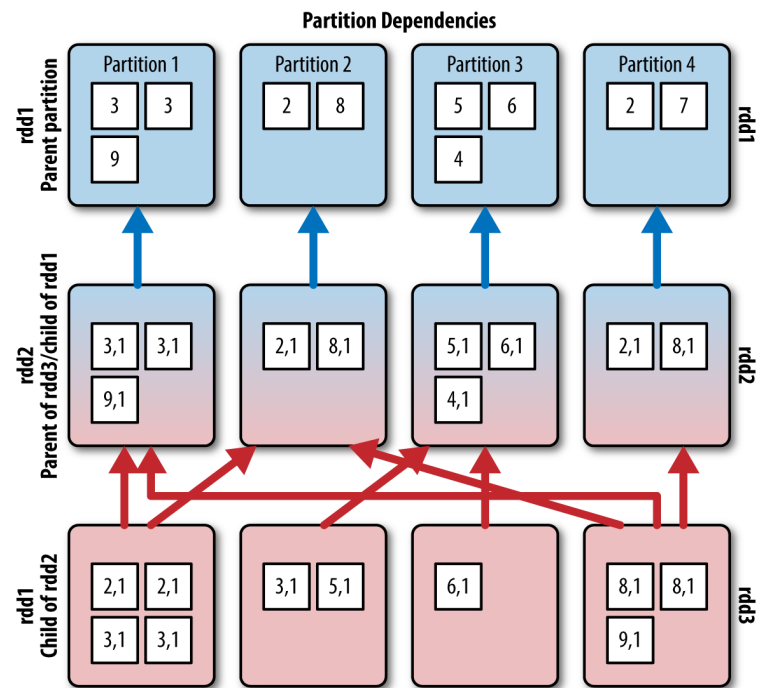


Require data from all parent partitions to be available and to be shuffled across the nodes using a MapReduce-like operation.



Expensive recovery, as a complete re-execution of all parent partitions is needed.

Narrow Versus Wide Dependencies



Narrow dependencies
`val rdd2=rdd1.map(
 x=(x,1))`

Wide dependencies
`val rdd3=
 rdd2.groupByKey`

Image taken from High Performance Spark by Holden Karau & Rachel Warren

How Spark works

RDD: a parallel collection w/ partitions

User application creates RDDs, transforms them, and runs actions

These result in a DAG of operators

DAG is compiled into stages

Each stage is executed as a series of tasks

Example

```
sc.textFile("/some-hdfs-data")  
  .map(line => line.split("\t"))  
  .map(parts =>  
    (parts[0], int(parts[1])))  
  .reduceByKey(_ + _, 3)  
  .collect()
```

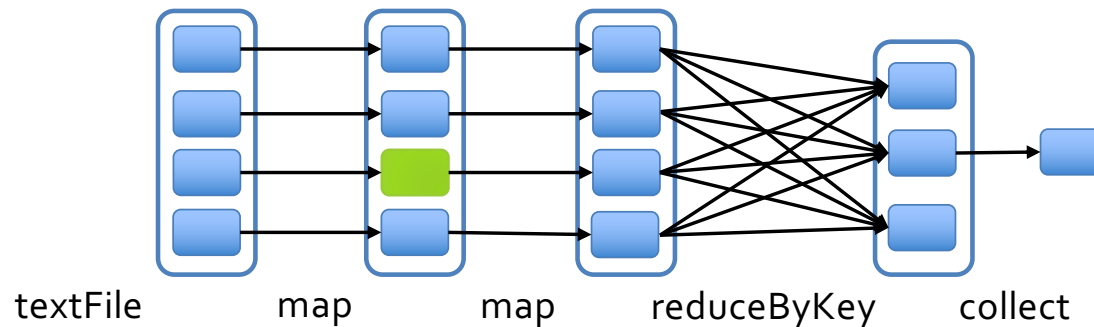
RDD[String]

RDD[List[String]]

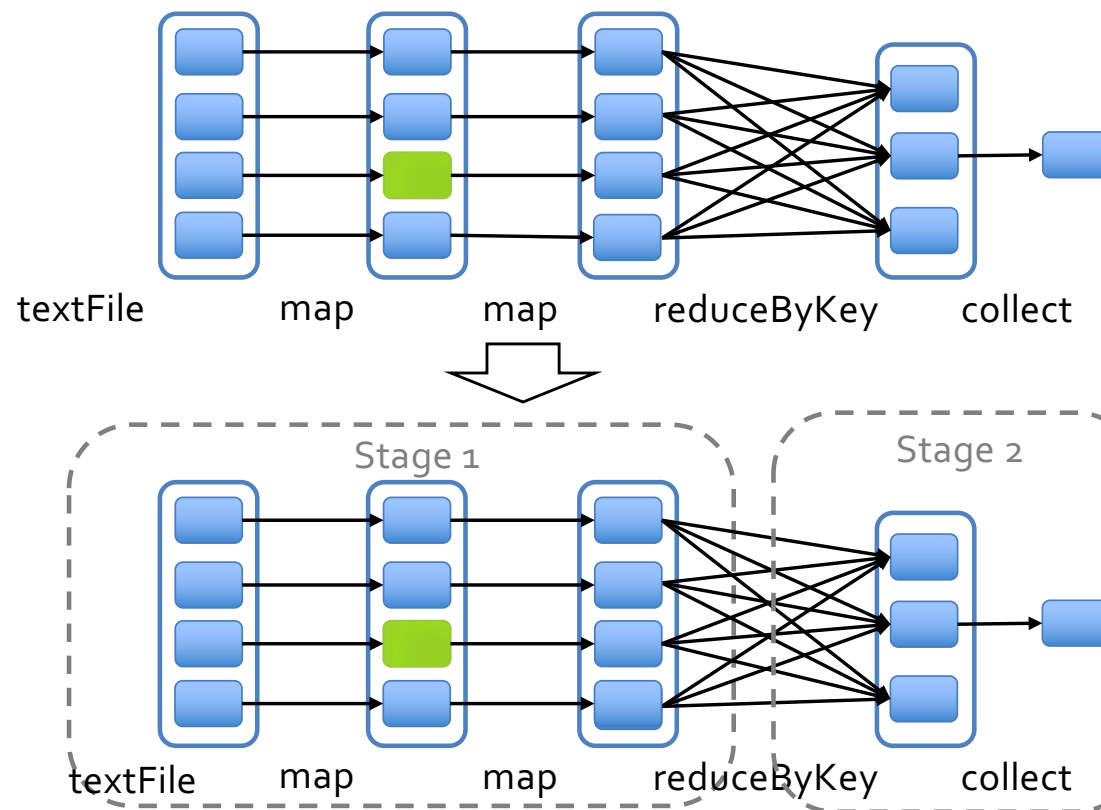
RDD[(String, Int)]

RDD[(String, Int)]

List[(String, Int)]



Execution Graph



Narrow Versus Wide Dependencies

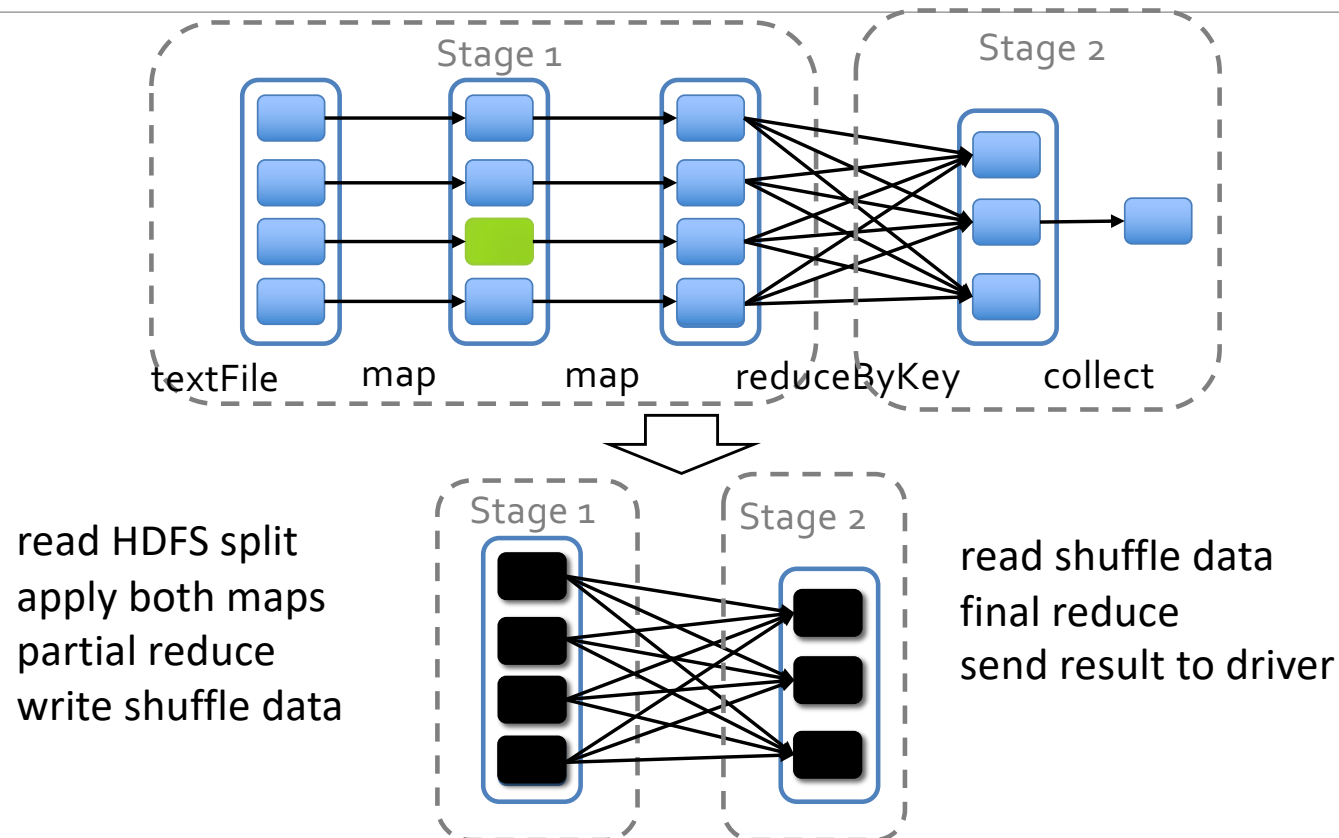
Narrow dependencies do not require data to be moved across partitions.

- narrow transformations don't require communication with the driver node

Each series of narrow transformations can be computed in the same “stage” of the query execution plan.

-

Execution Graph



Stage execution



Create a task for each partition in the new RDD

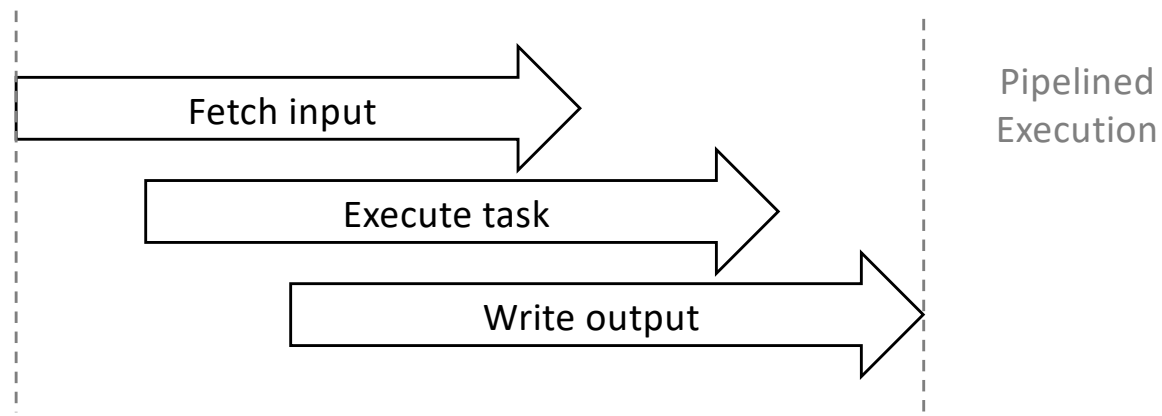
Serialize task

Schedule and ship task to slaves

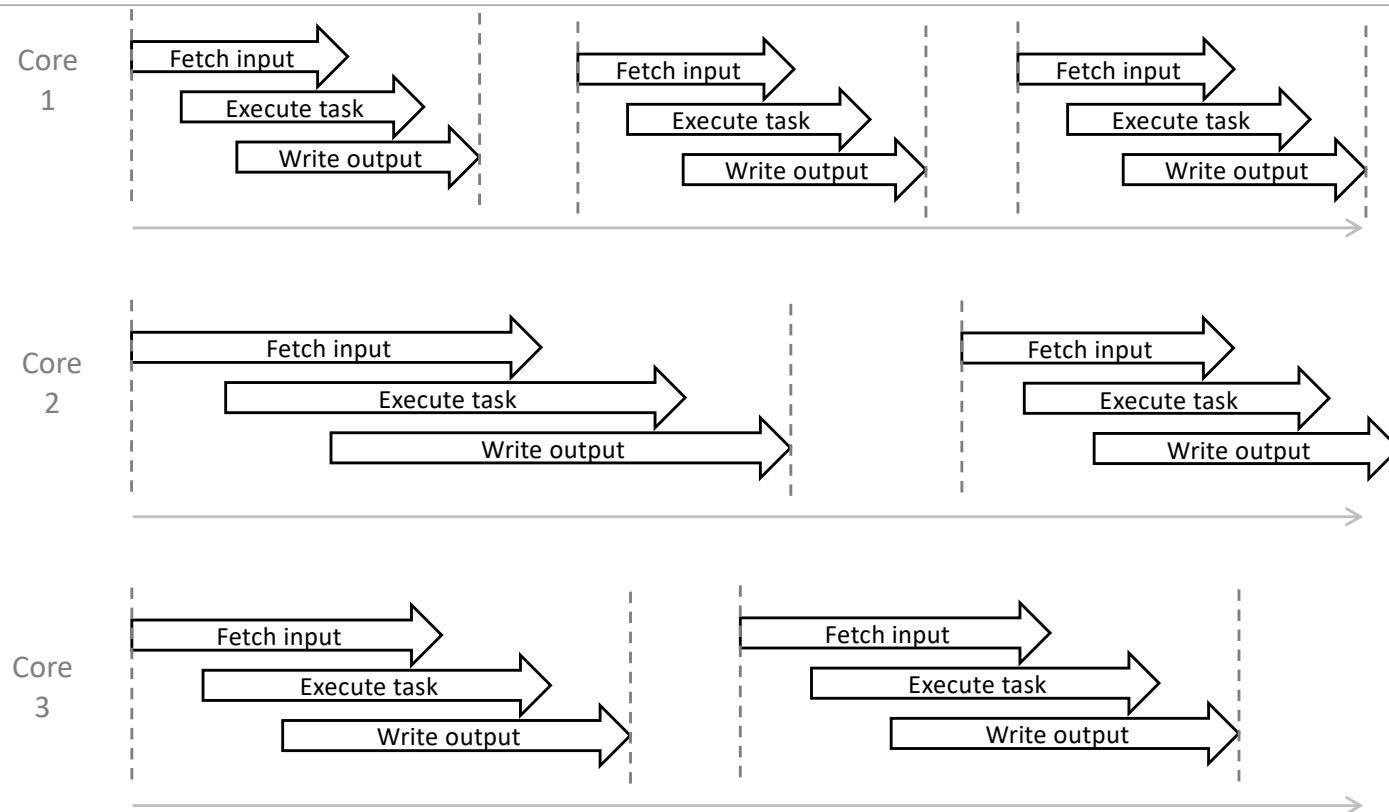
Task execution

Fundamental unit of execution in Spark

1. Fetch input from InputFormat or a shuffle
2. Execute the task
3. Materialize task output as shuffle or driver result



Spark Executor



Summary of Components

Tasks: Fundamental unit of work

Stage: Set of tasks that run in parallel

DAG: Logical graph of RDD operations

RDD: Parallel dataset with partitions

Why gain a deeper understanding?

(patrick, \$24), (matei, \$30), (patrick, \$1), (aaron, \$23),
(aaron, \$2), (reynold, \$10), (aaron, \$10).....

RDD

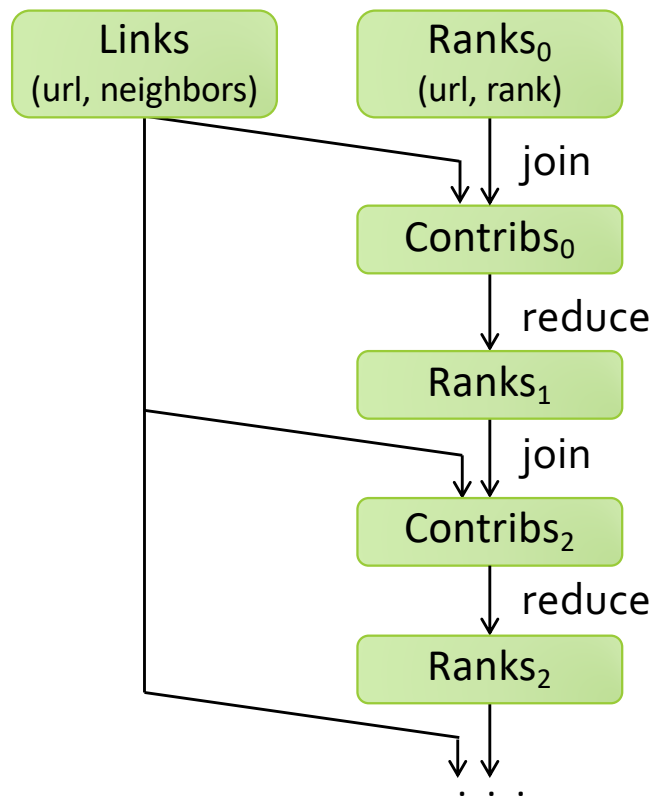
```
spendPerUser = rdd.groupByKey()  
                  .map(lambda pair: sum(pair[1]))  
                  .collect()
```

Copies all data over
the network

```
spendPerUser = rdd.reduceByKey(lambda x, y: x + y)  
                   .collect()
```

Reduces locally
before shuffling

PageRank Optimizing Placement



links & ranks repeatedly joined

Can co-partition them (e.g. hash both on URL) to avoid shuffles

```
links = links.partitionBy(  
    new URLPartitioner())
```

PageRank Performance

