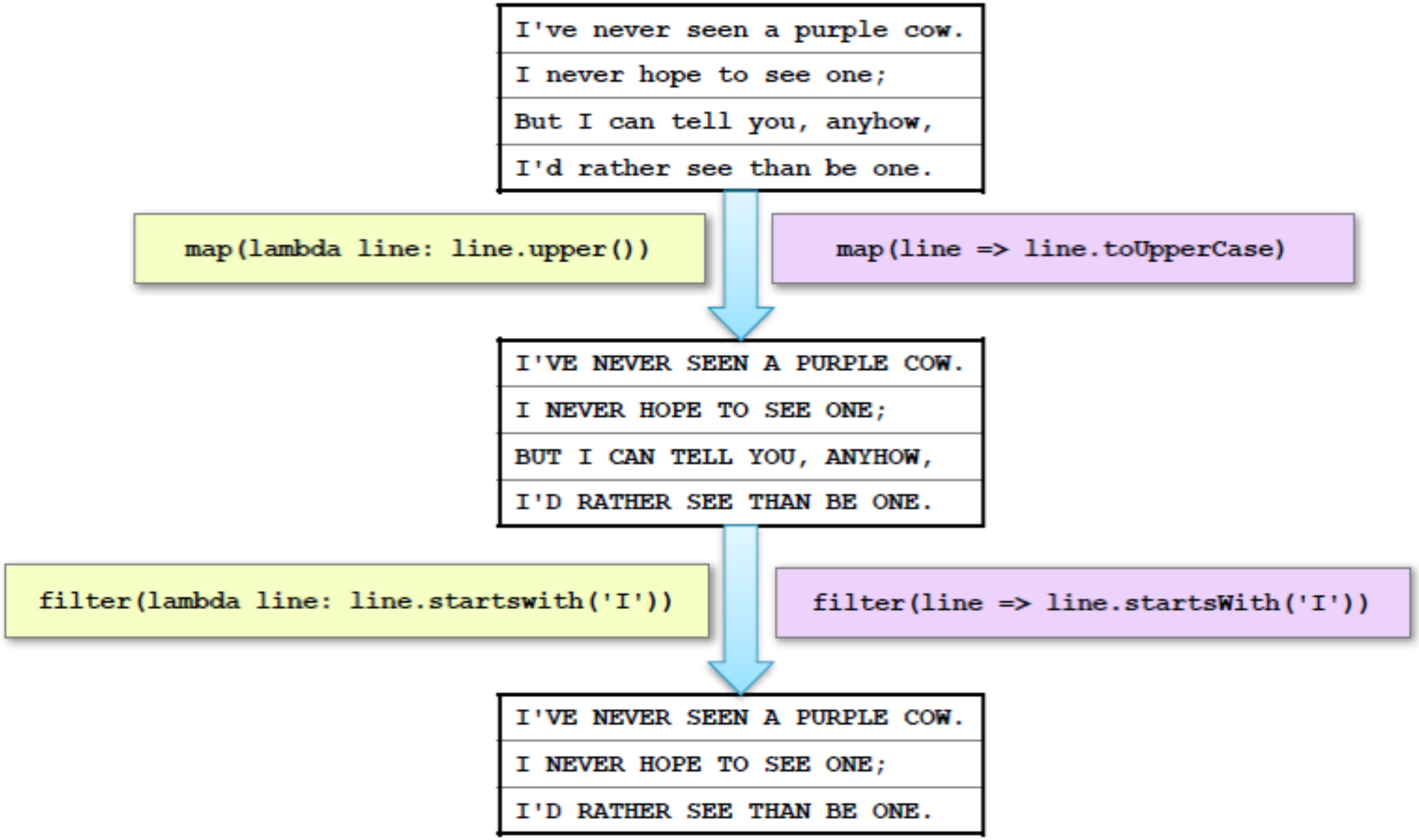
CSCI 381/780
Cloud Computing

Spark

Jun Li
Queens College

Example: map and filter Transformations



RDD Actions

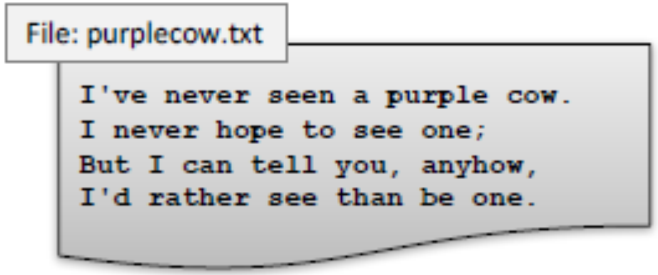
- Apply transformation chains on RDDs, eventually performing some additional operations (e.g., counting)
- Some actions only store data to an external data source (e.g. HDFS), others fetch data from the RDD (and its transformation chain) upon which the action is applied, and convey it to the driver
- Some common actions
  - count() – return the number of elements
  - take(*n*) – return an array of the first *n* elements
  - collect()– return an array of all elements
  - saveAsTextFile(*file*) – save to text file(s)

Graph of RDDs

- A collection of RDDs can be understood as a graph
- Nodes in the graph are the RDDs, which means the code but also the actual data object that will be create at runtime when executed on specific parameters + data. Reminder: RDD is a “read only” model, so we can “materialize” an RDD any time we like.
- Edges represent how data objects are accessed: RDD B might consume the object created by RDD A. This gives us a directed edge A → B

Lazy Execution of RDDs (1)

Data in RDDs is not processed until an action is performed


File: purplecow.txt

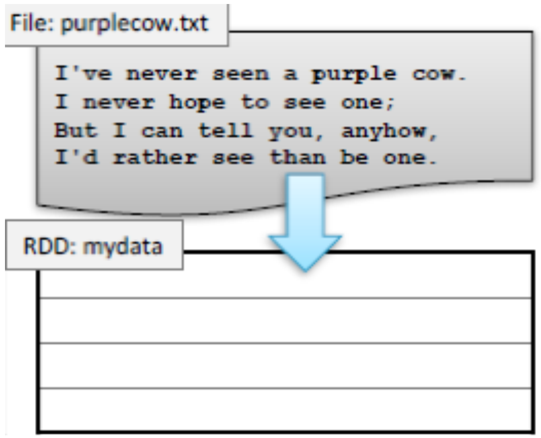
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.

```
>
```

Lazy Execution of RDDs (2)

Data in RDDs is not processed until an action is performed

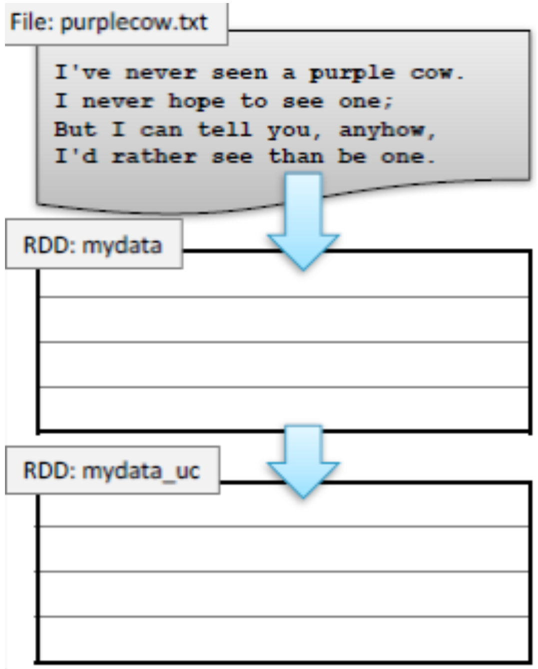
```
> val mydata = sc.textFile("purplecow.txt")
```



Lazy Execution of RDDs (3)

Data in RDDs is not processed until an action is performed

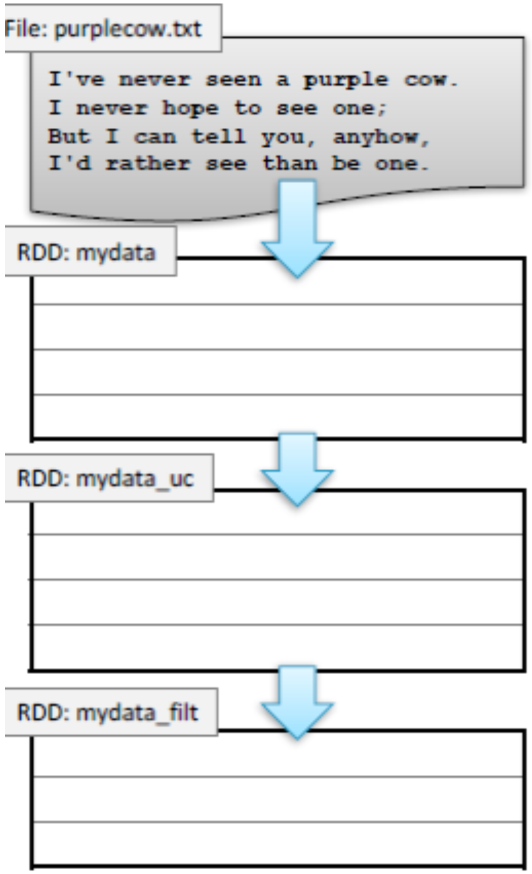
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
```



Lazy Execution of RDDs (4)

Data in RDDs is not processed until an action is performed

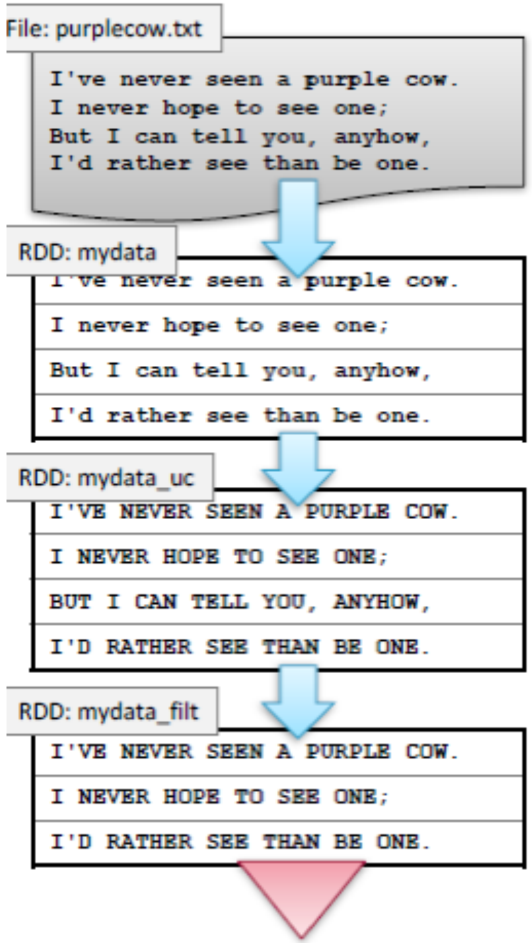
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
```



Lazy Execution of RDDs (5)

Data in RDDs is not processed until an action is performed

```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.count()
3
```



Output Action "triggers" computation, pull model

Opportunities This Enables

- **On-demand optimization**: Spark can behave like a compiler by first building a potentially complex RDD graph, but then trimming away unneeded computations that for today's purpose, won't be used.
- **Caching for later reuse**.
- **Graph transformations**: A significant amount of effort has been made in this area. It is a lot like compiler-managed program transformation and aims at simplifying and speeding up the computation that will occur.
- **Dynamic decisions about what to schedule and when**. Concept: *minimum adequate set* of input objects: RDD can run if *all* its inputs are ready

Example: Mine error logs

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

```
lines = spark.textFile("hdfs://...")    HadoopRDD
errors = lines.filter(lambda s: s.startswith("ERROR")) FilteredRDD
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()
messages.filter(lambda s: "foo" in s).count()
```

Result: full-text search of Wikipedia in 0.5 sec (vs 20 sec for on-disk data)

Key Idea: Elastic parallelism

RDDs operations are designed to offer embarrassing parallelism.

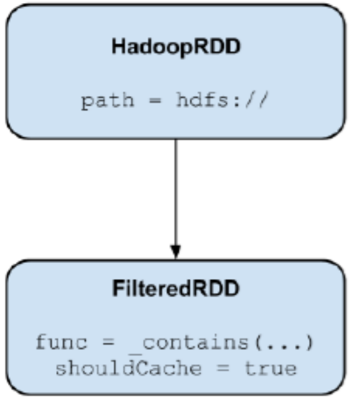
Spark will spread the task over the nodes where data resides, offers a highly concurrent execution that minimizes delays. Term: “partitioned computation” .

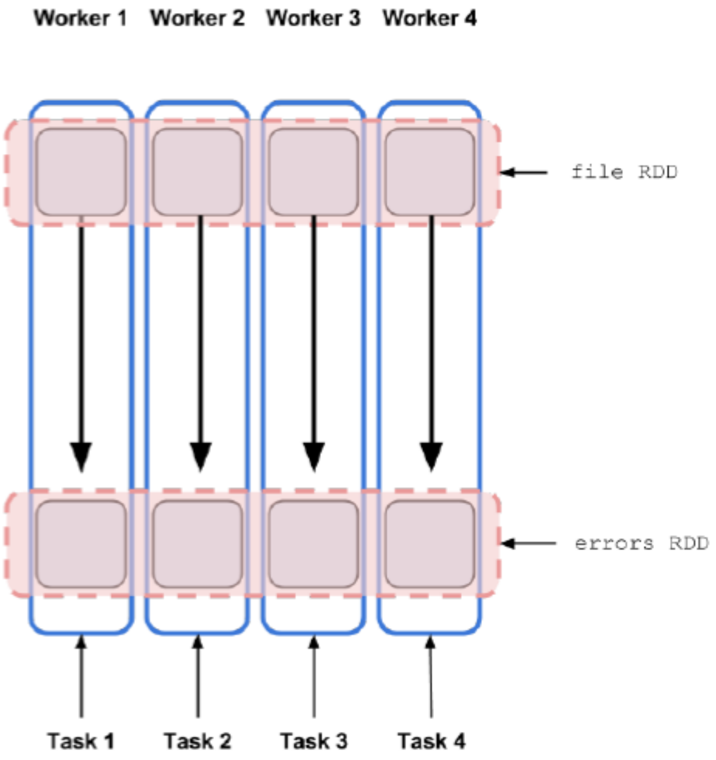
If some component crashes or even is just slow, Spark simply kills that task and launches a substitute.

RDD Graph: Data Set vs Partition Views

Much like in Hadoop MapReduce, each RDD is associated to (input) partitions

```
val sc = new SparkContext("spark://...", "MyJob", home,
    jars)

val file = sc.textFile("hdfs://...") // This is an RDD

val errors = file.filter(_.contains("ERROR")) // This is
    an RDD

errors.cache()

errors.count() // This is an action
```





RDDs: Data Locality

- Data Locality Principle
  - ➢ Keep high-value RDDs precomputed, in cache or SDD
  - ➢ Run tasks that need the specific RDD with those same inputs on the node where the cached copy resides.
  - ➢ This can maximize in-memory computational performance.

Requires cooperation between your hints to Spark when you build the RDD, Spark runtime and optimization planner, and the underlying YARN resource manager.

Typical RDD pattern of use

Instead of doing a lot of work in each RDD, developers split tasks into lots of small RDDs

These are then organized into a DAG.

Developer anticipates which will be costly to recompute and hints to Spark that it should cache those.

Why is this a good strategy?

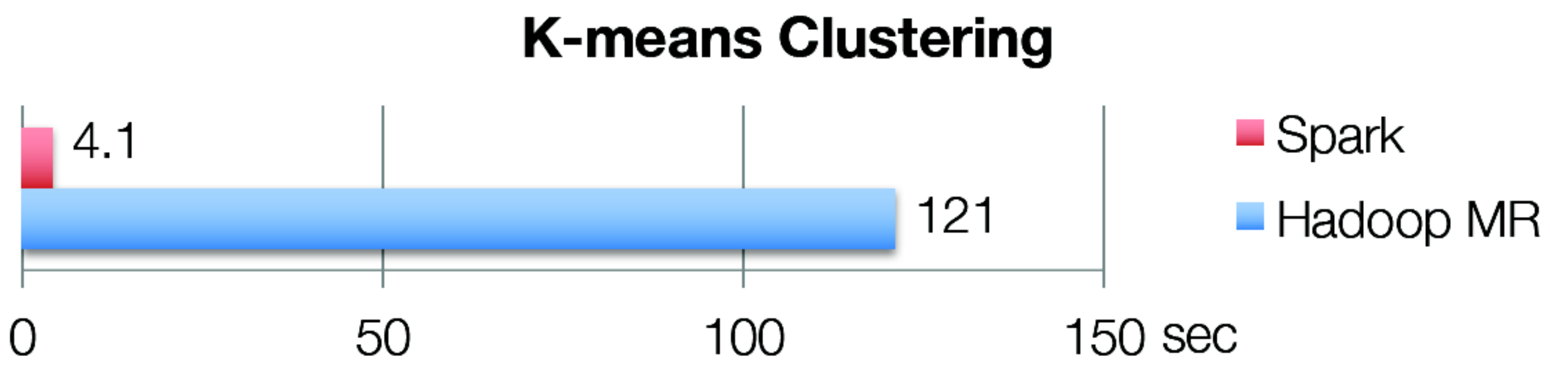
Spark tries to run tasks that will need the same intermediary data on the same nodes.

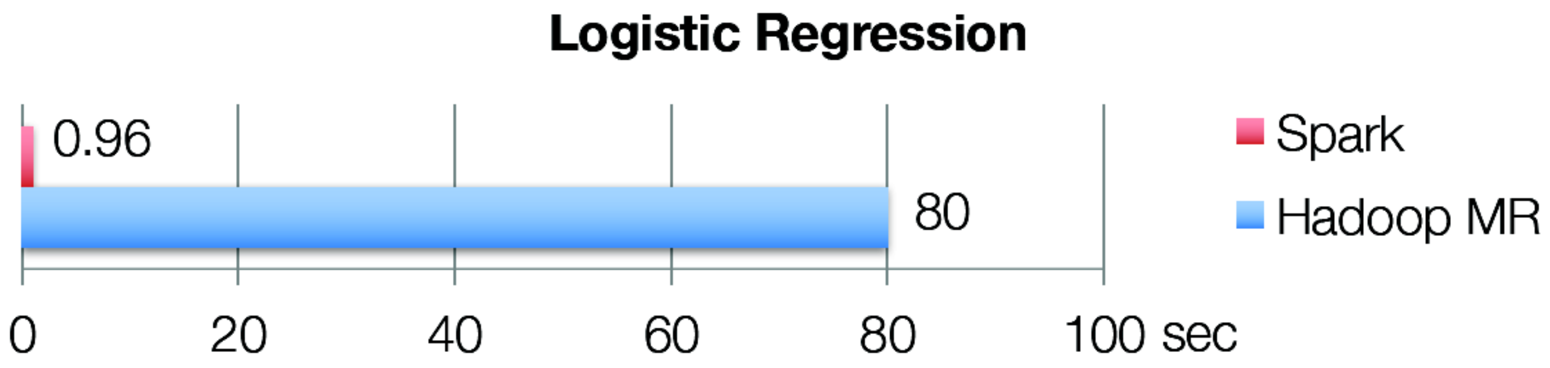
If MapReduce jobs were arbitrary programs, this wouldn't help because reuse would be very rare.

But in fact the MapReduce model is very repetitious and iterative, and often applies the same transformations again and again to the same input files.

- ➢ Those particular RDDs become great candidates for caching.
- ➢ MapReduce programmer may not know how many iterations will occur, but Spark itself is smart enough to evict RDDs if they don't actually get reused.

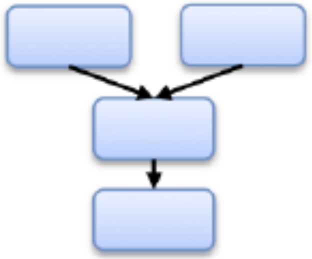
Iterative Algorithms: Spark vs MapReduce





Lifetime of a Job in Spark

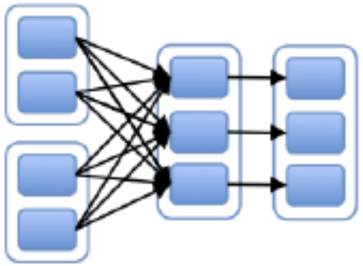
RDD Objects



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

Build the operator DAG

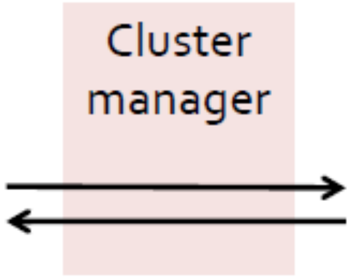
DAG Scheduler



Split the DAG into *stages of tasks*

Submit each stage and its tasks as ready

Task Scheduler



Launch tasks via Master

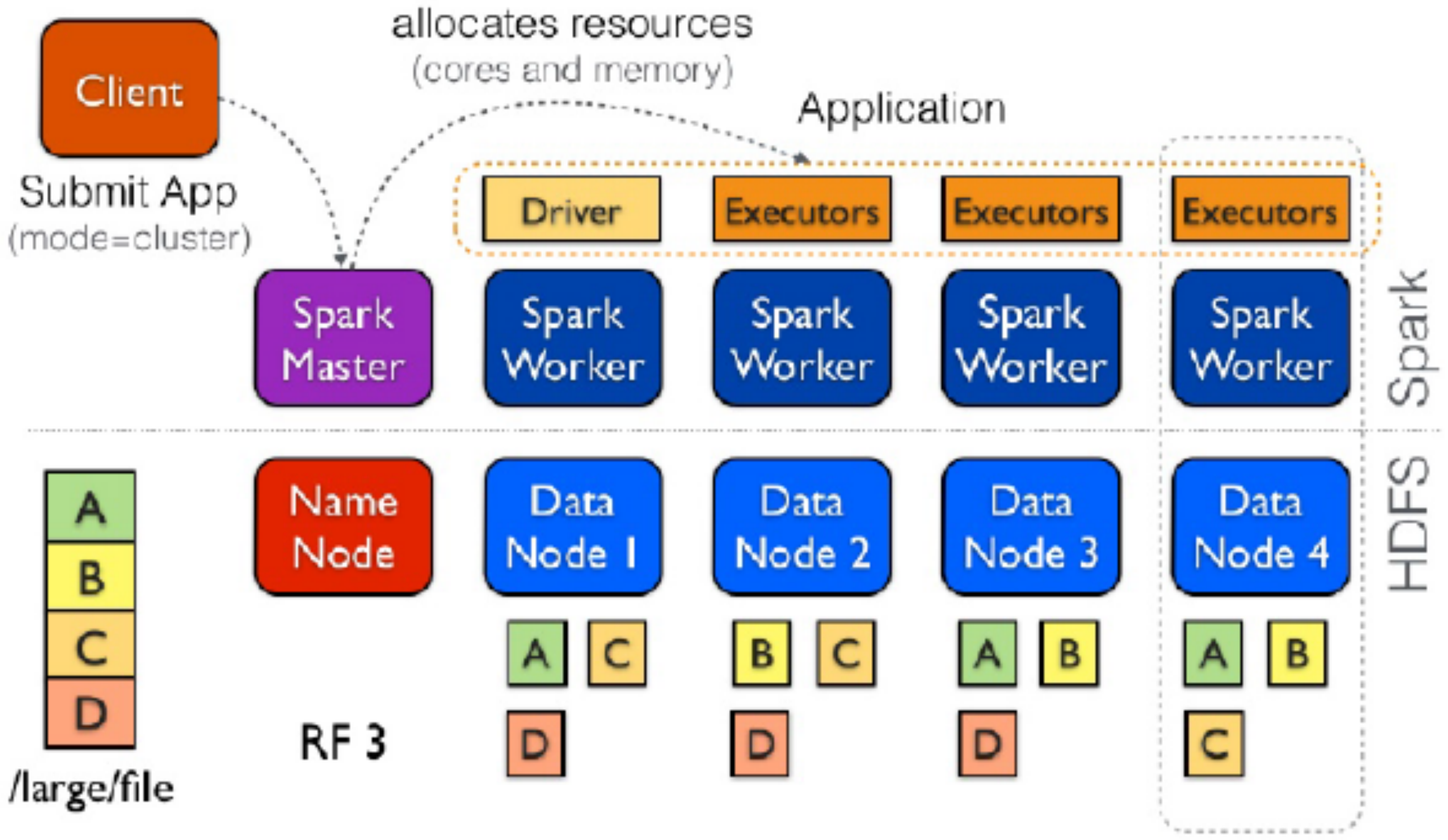
Retry failed and straggler tasks

Worker

Execute tasks

Store and serve blocks

Anatomy of a Spark Application



Cluster Manager
(YARN/Mesos)

Today’s Topics

- Motivation
- Spark Basics
- Spark Programming

Spark Programming (1)

Creating RDDs

```
# Turn a Python 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 existing Hadoop InputFormat (Java/Scala only)
sc.hadoopFile(keyClass, valClass, inputFmt, conf)
```

Basic Transformations

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

# Pass each element through a function
squares = nums.map(lambda x: x*x) // {1, 4, 9}

# Keep elements passing a predicate
even = squares.filter(lambda x: x % 2 == 0) // {4}
```

Spark Programming (3)

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(lambda x, y: x + y) # => 6
```

Spark Programming (4)

Working with Key-Value Pairs

Spark’s “distributed reduce” transformations operate on RDDs of key-value pairs

```
Python:  pair = (a, b)
          pair[0] # => a
          pair[1] # => b

Scala:   val pair = (a, b)
          pair._1 // => a
          pair._2 // => b

Java: Tuple2 pair = new Tuple2(a, b);
          pair._1 // => a
          pair._2 // => b
```

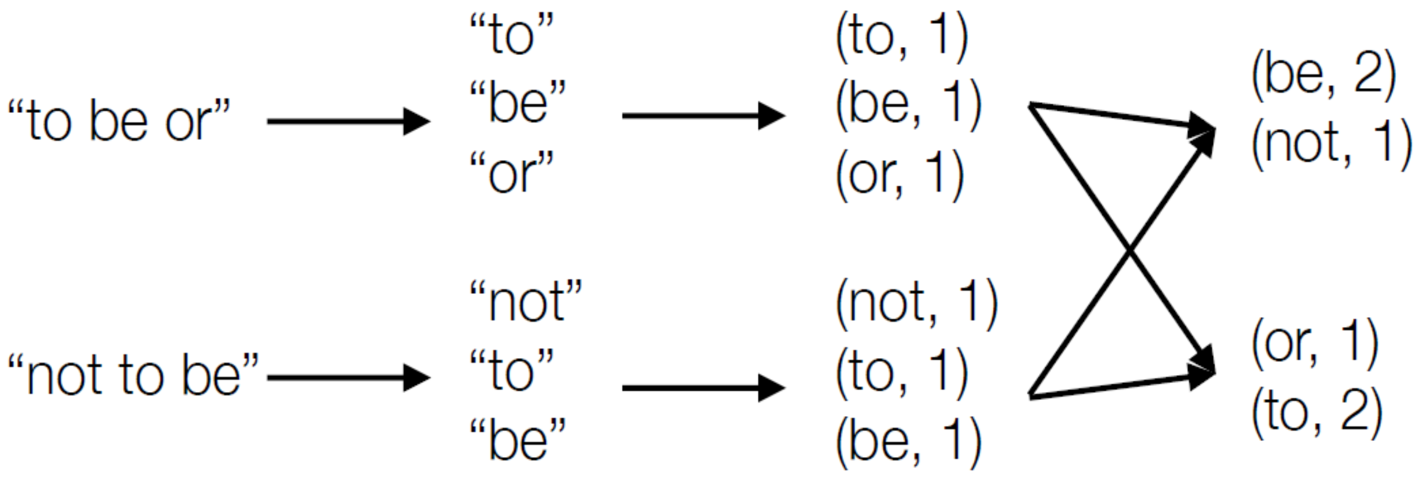
Spark Programming (5)

Some Key-Value Operations

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

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

pets.groupByKey()      # => {(cat, [1, 2]), (dog, [1])}

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

Example: Word Count

```
lines = sc.textFile("hamlet.txt")
counts = lines.flatMap(lambda line: line.split(" "))
              .map(lambda word: (word, 1))
              .reduceByKey(lambda x, y: x + y)
```



Spark: Setting the Level of Parallelism

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

```
words.reduceByKey(lambda x, y: x + y, 5)

words.groupByKey(5)

visits.join(pageViews, 5)
```