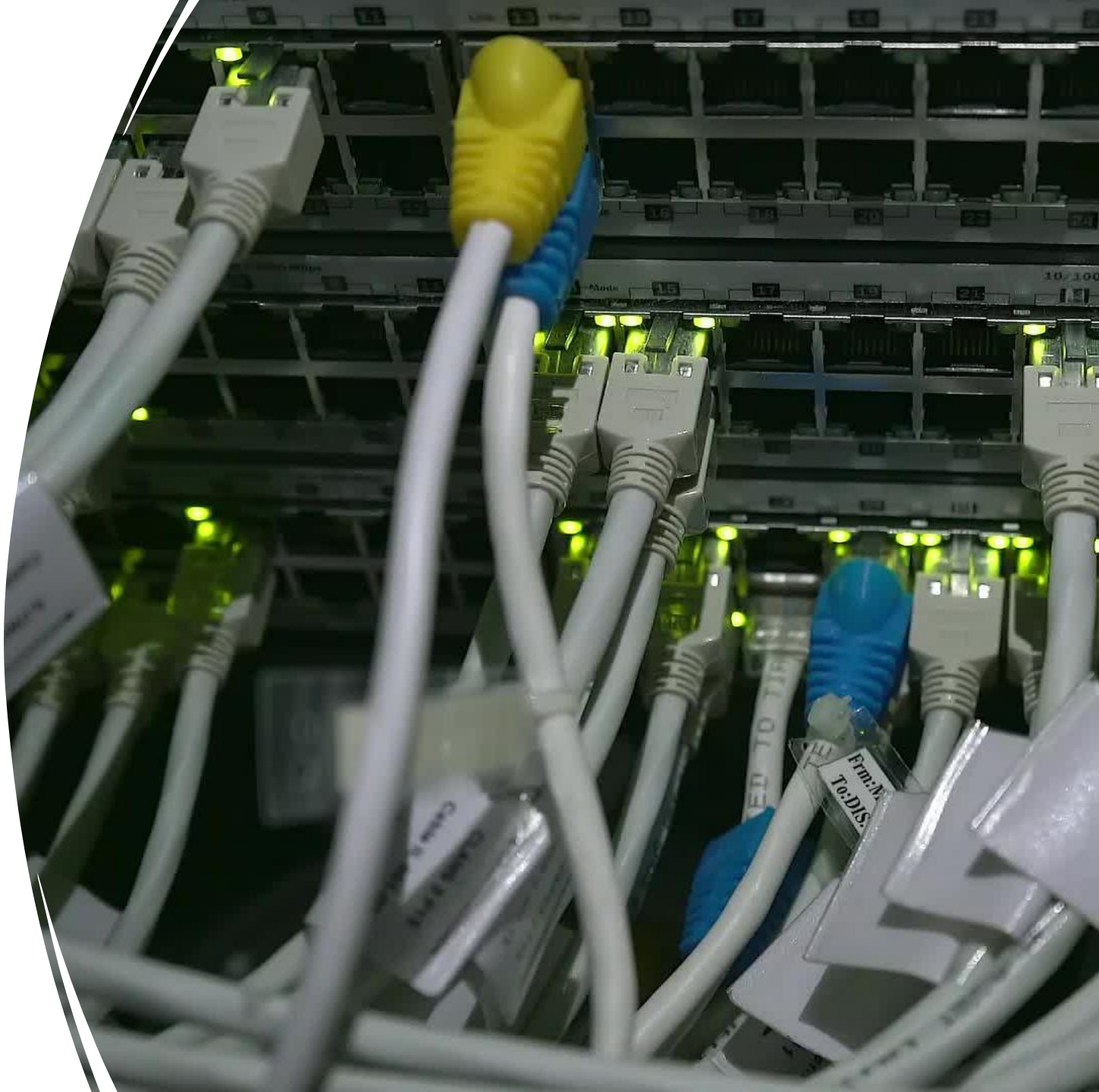
Data-Intensive Distributed Computing CS431/451/651

Module 3 – From MapReduce to Spark



This Module's Agenda

- Higher-Level Programming
- Spark
- Algorithm Design

The Datacenter as a Computer

Q: What's the instruction set?

A: Hadoop

Layers of Abstraction

- Higher Level Language (e.g. Python)
- Lower Level Language (e.g. C)
- Assembly
- Machine Code
- Instruction Set Architecture
- Micro-Architecture
- Gates, Adders, Registers, Etc.
- Electronics (Transistors)
- Physics

Data Center Abstraction

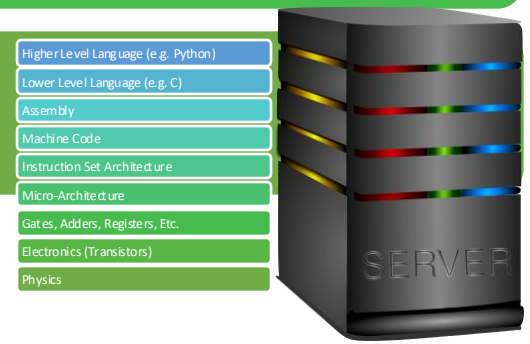
??? <TODAY'S TOPIC>

Hadoop Task

HDFS / Hadoop Framework

Cluster of Computers (Networking)

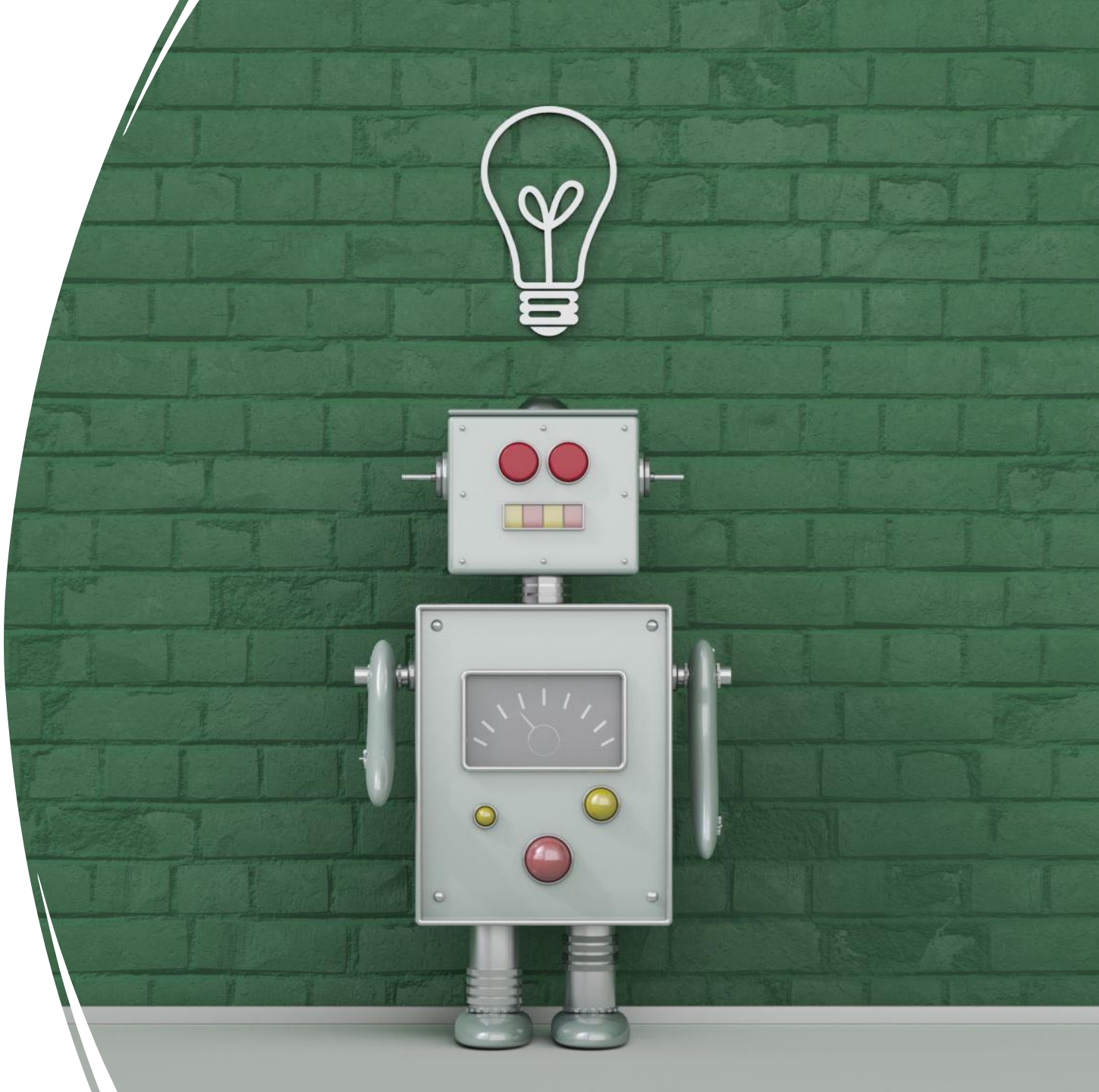
Individual Servers



Do you like programming in assembly?

What's the alternative?

- Hadoop is great, but has a lot of boilerplate and repetition
- It's also tedious to program
- Can we create a Distributed C (or Python) to Hadoop's Assembly?

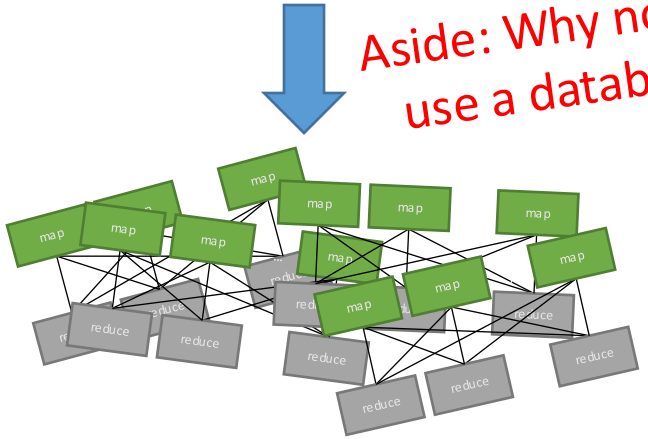


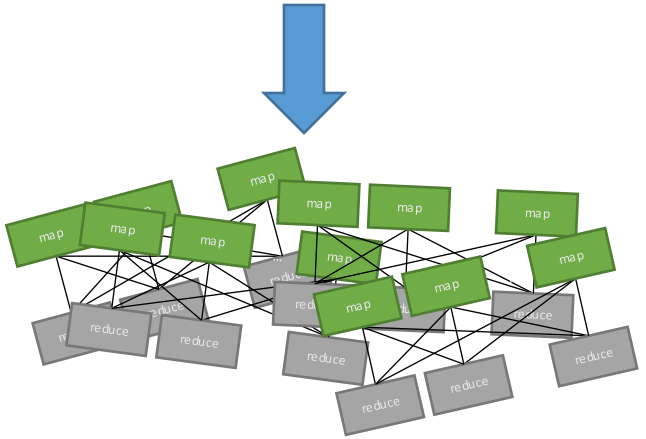
Yes We* Can

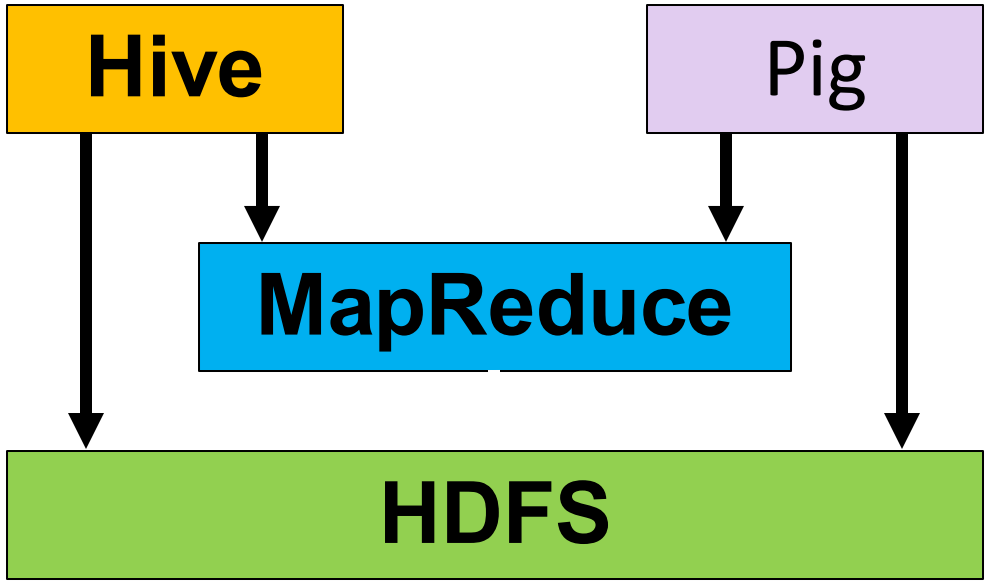
What we really need is SQL!

What we really need is a scripting language!

Aside: Why not just use a database?






Hive
Pig
MapReduce
HDFS

Pig Examples

Pig: Example

Task: Find the top 10 most visited pages in each category

Visits

User	Url	Time
Amy	cnn.com	8:00
Amy	bbc.com	10:00
Amy	flickr.com	10:05
Fred	cnn.com	12:00

URL Info

Url	Category	PageRank
cnn.com	News	0.9
bbc.com	News	0.8
flickr.com	Photos	0.7
espn.com	Sports	0.9

Pig Slides adapted from Olston et al. (SIGMOD 2008)

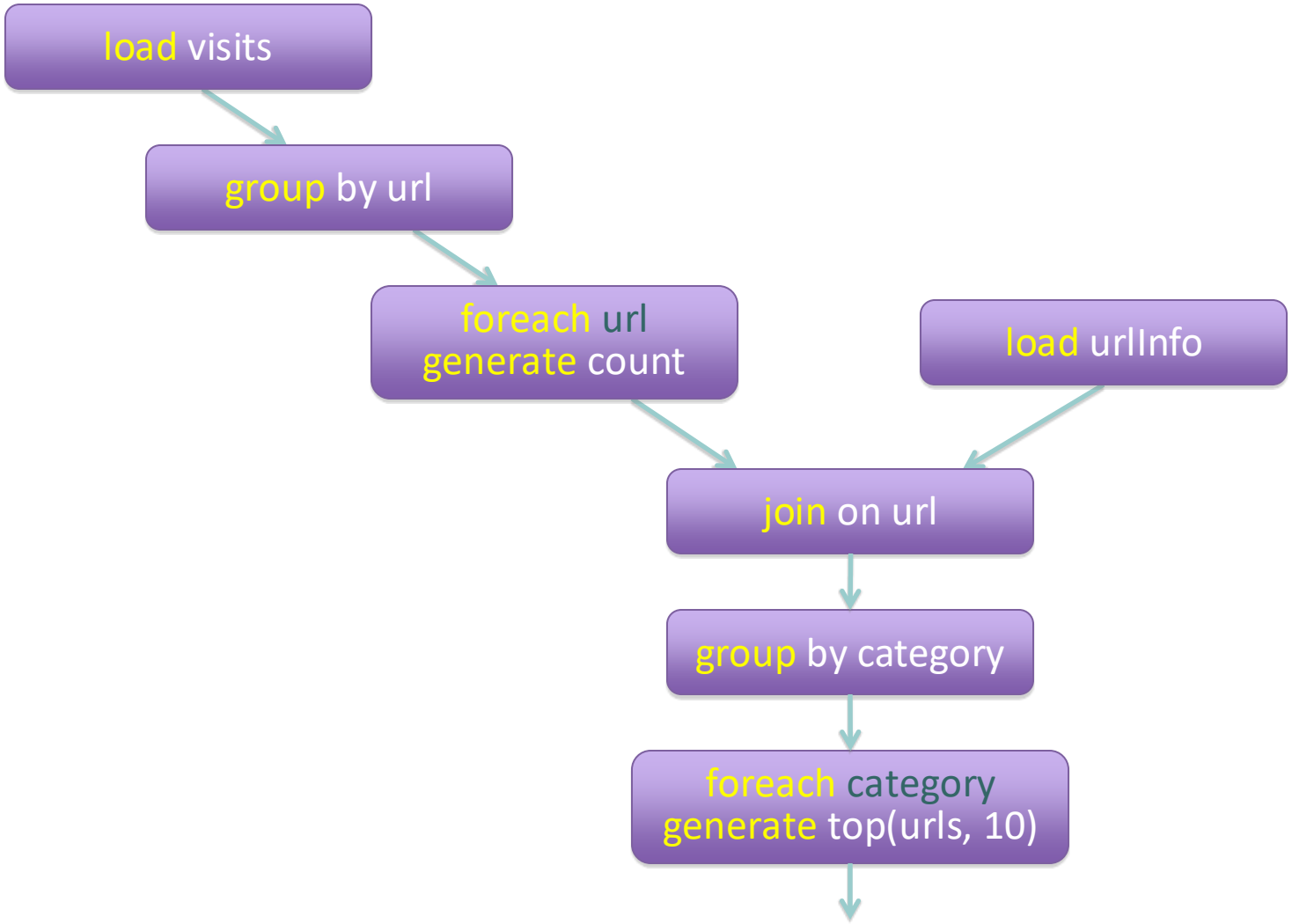
Pig: Example Script

```
visits = load ‘/data/visits’ as (user, url, time);
gVisits = group visits by url;
visitCounts = foreach gVisits generate url, count(visits);
urlInfo = load ‘/data/urlInfo’ as (url, category, pRank);
visitCounts = join visitCounts by url, urlInfo by url;
gCategories = group visitCounts by category;
topUrls = foreach gCategories generate
  top(visitCounts,10);

store topUrls into ‘/data/topUrls’;
```

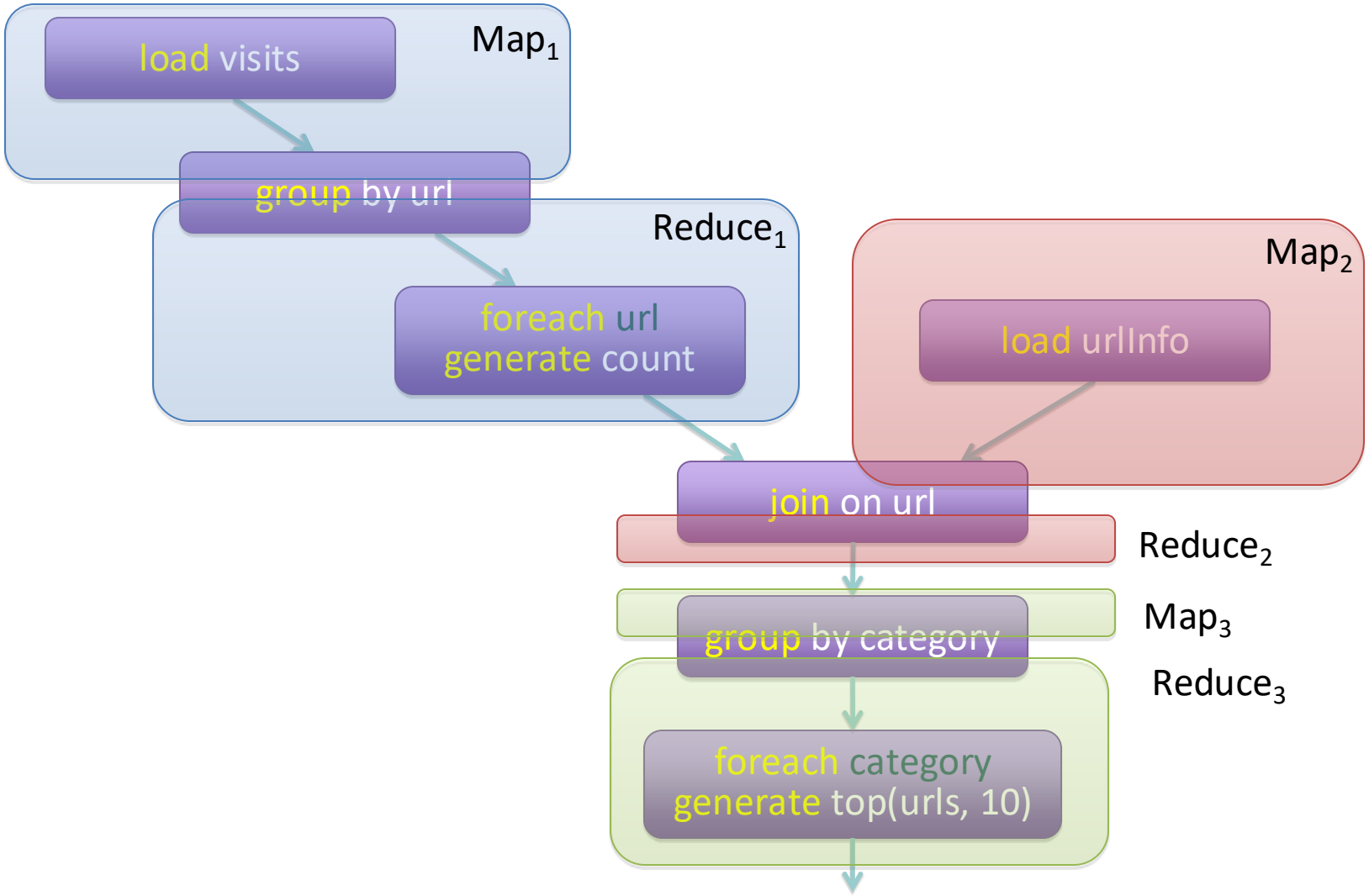
Pig Slides adapted from Olston et al. (SIGMOD 2008)

Pig Query Plan



Pig Slides adapted from Olston et al. (SIGMOD 2008)

Pig: MapReduce Execution



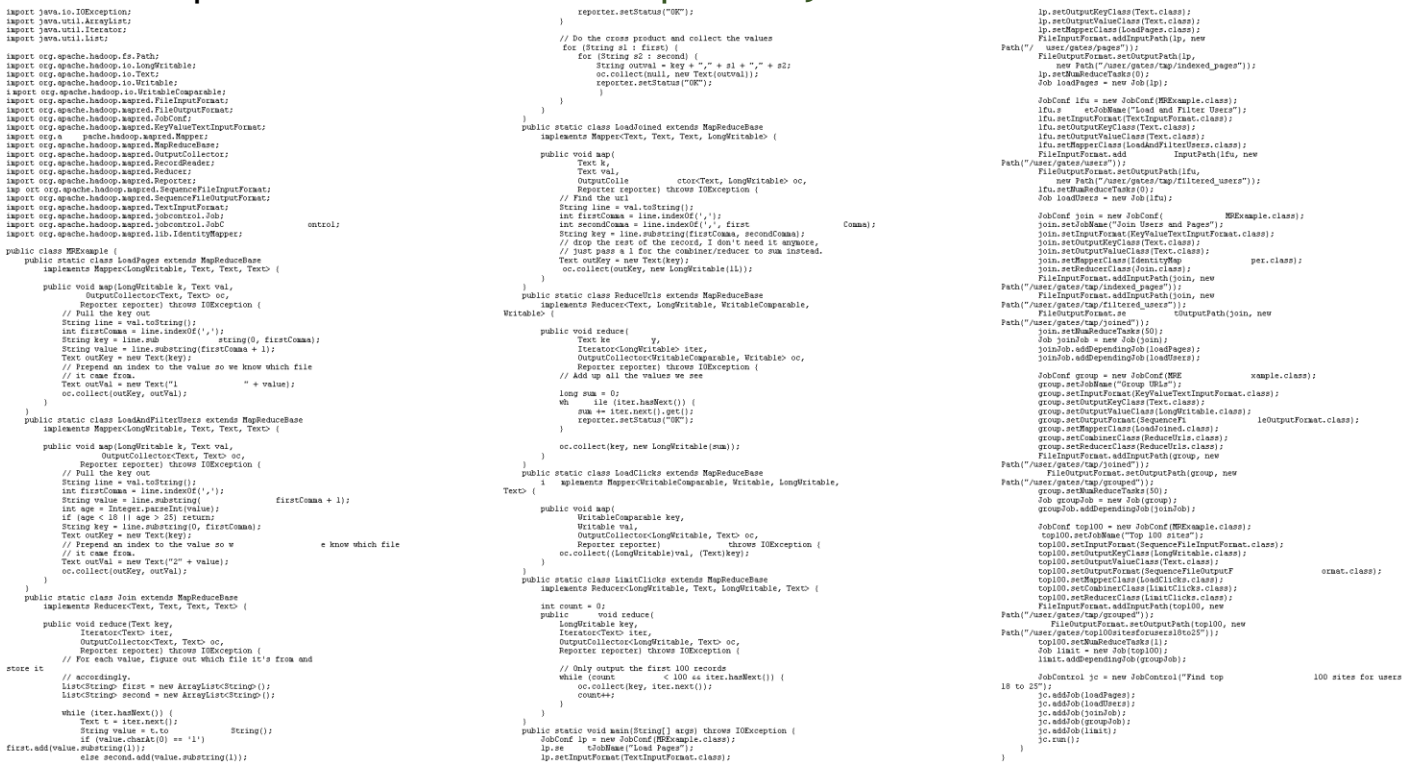
Pig Slides adapted from Olston et al. (SIGMOD 2008)

```
visits = load ‘/data/visits’ as (user, url, time);
gVisits = group visits by url;
visitCounts = foreach gVisits generate url, count(visits);
urlInfo = load ‘/data/urlInfo’ as (url, category, pRank);
visitCounts = join visitCounts by url, urlInfo by url;
gCategories = group visitCounts by category;
topUrls = foreach gCategories generate top(visitCounts,10);

store topUrls into ‘/data/topUrls’;
```

- Which would you rather:

- Read
- Write
- Debug (!)



Isn’t Pig Slower than Hadoop?

Potentially.

Isn’t C slower than assembly?

Isn’t Python slower than C?

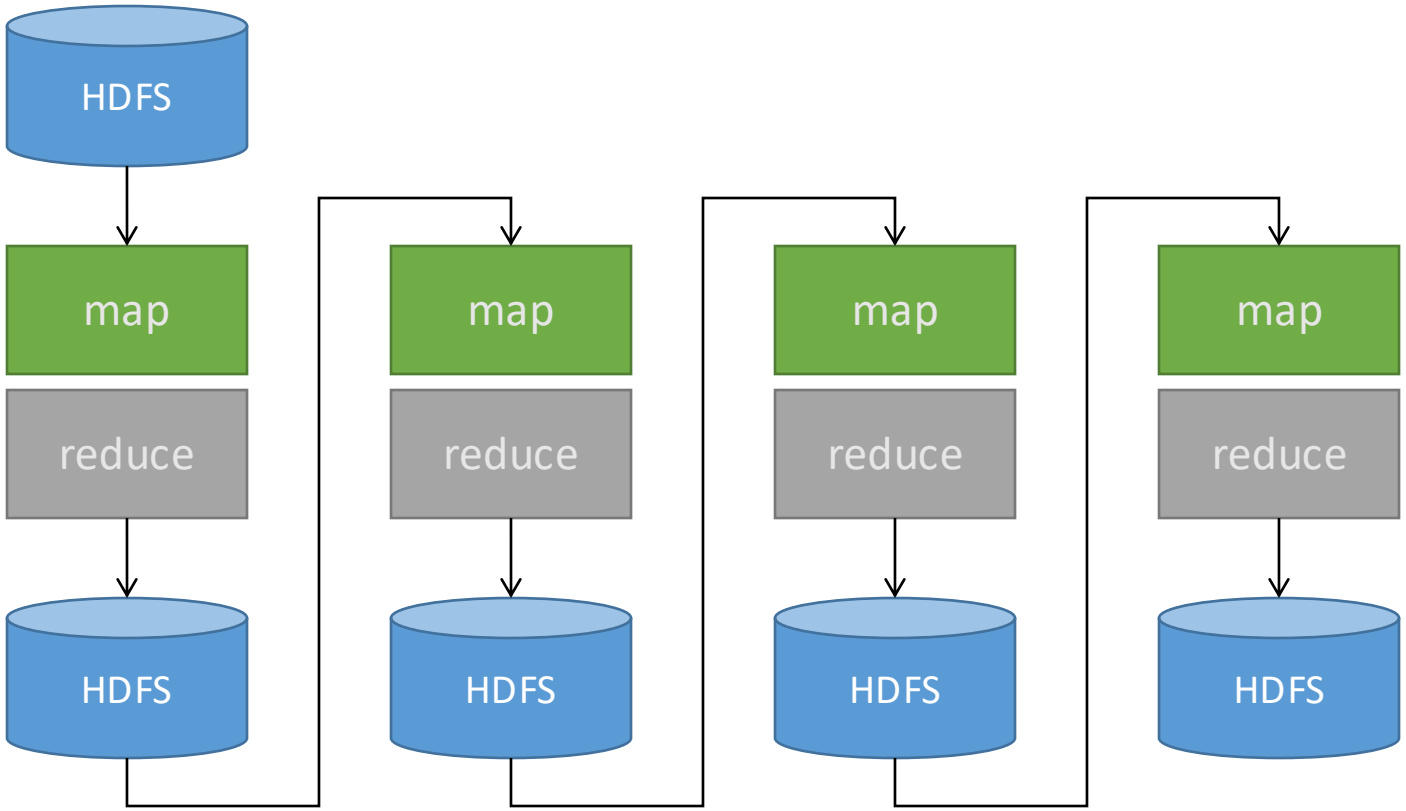
The Data Center as a Computer

So Hadoop is the Instruction Set, right?

What if I need two reduce passes. Do I really need two jobs?

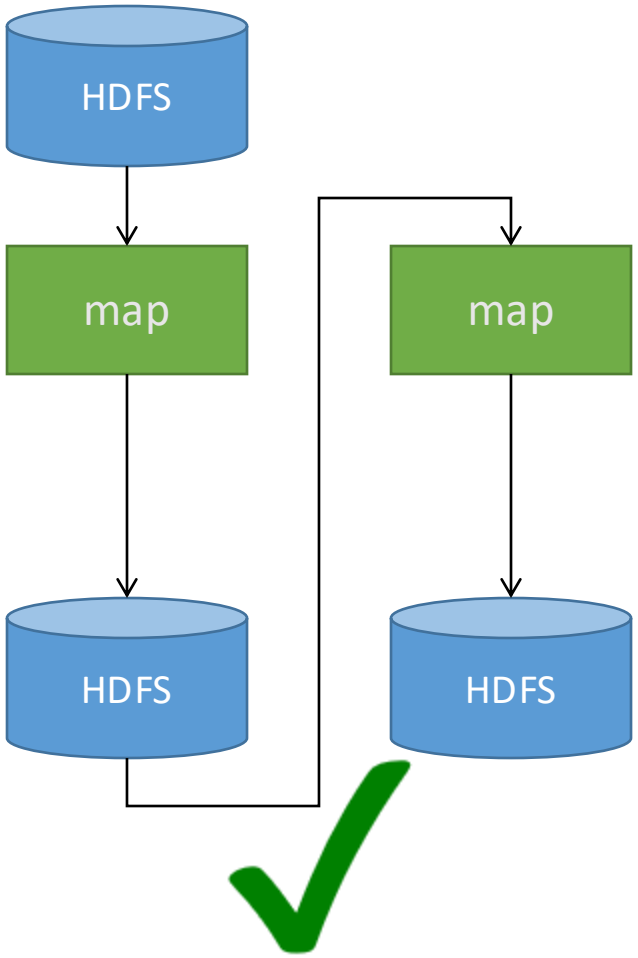
(On A1 yes, you do)

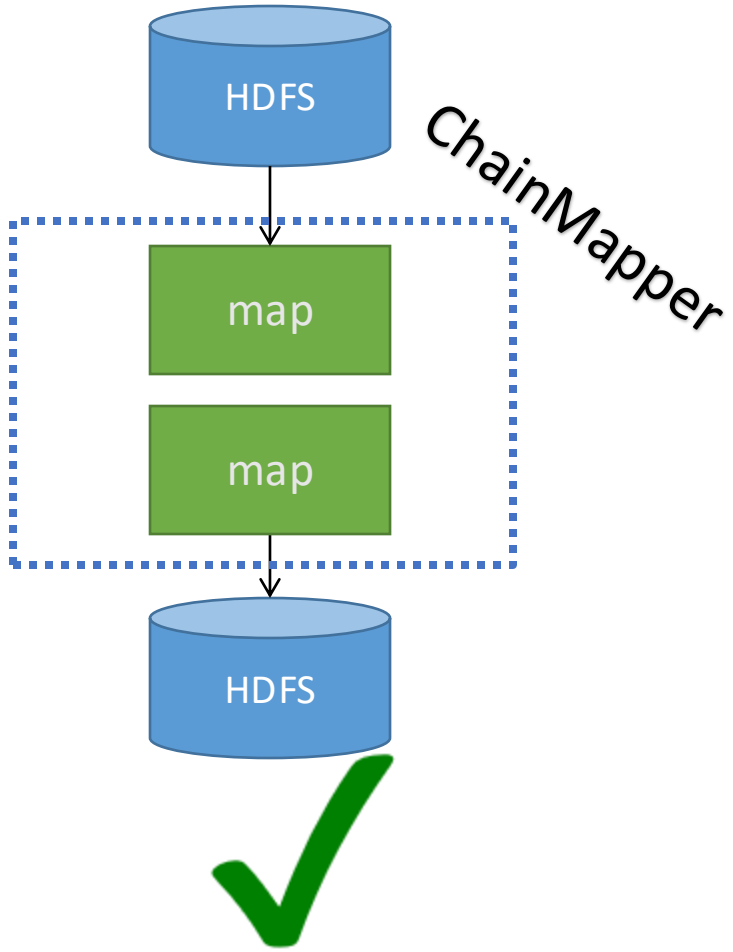
MapReduce Workflows



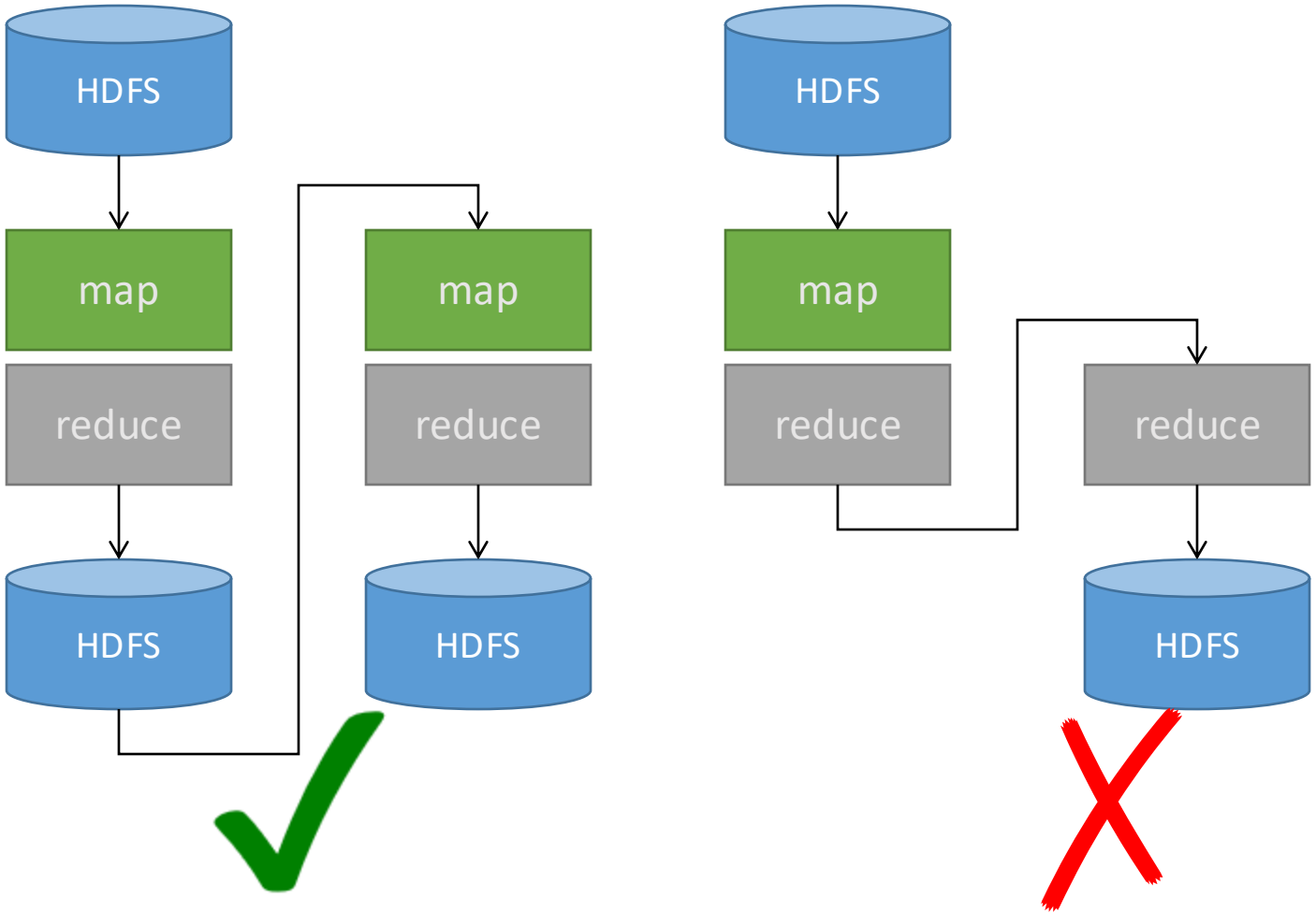
What's wrong?

Want Map-to-Map?





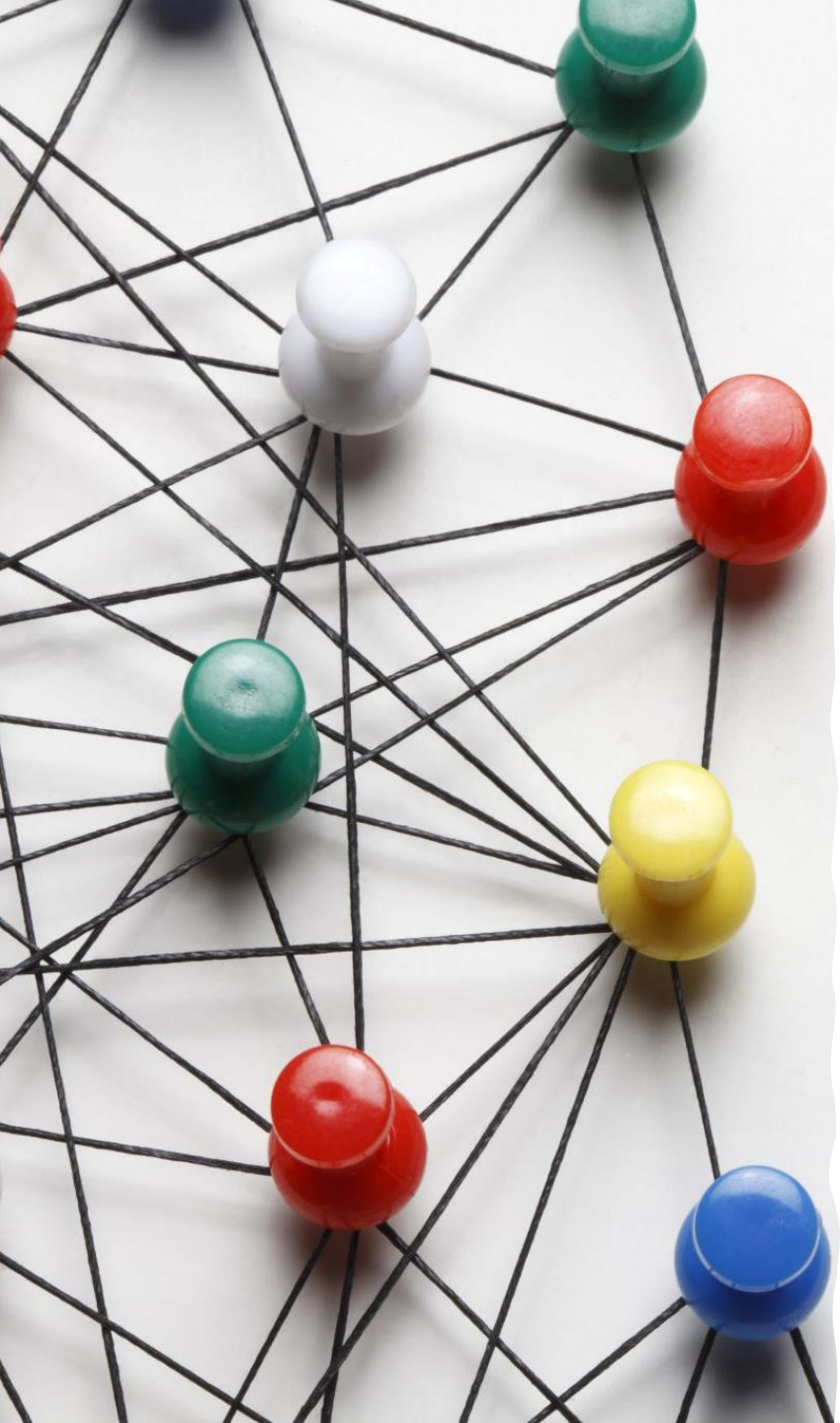
Want Map-to-Reduce-to-Reduce?



The Data Center as a Computer

Q: Is there a better instruction set?

A: Hadoop 2



Hadoop 2.0

Nodes are now resource managers

Can do MapReduce the same as always

Can also do other things

Other Things? Like What

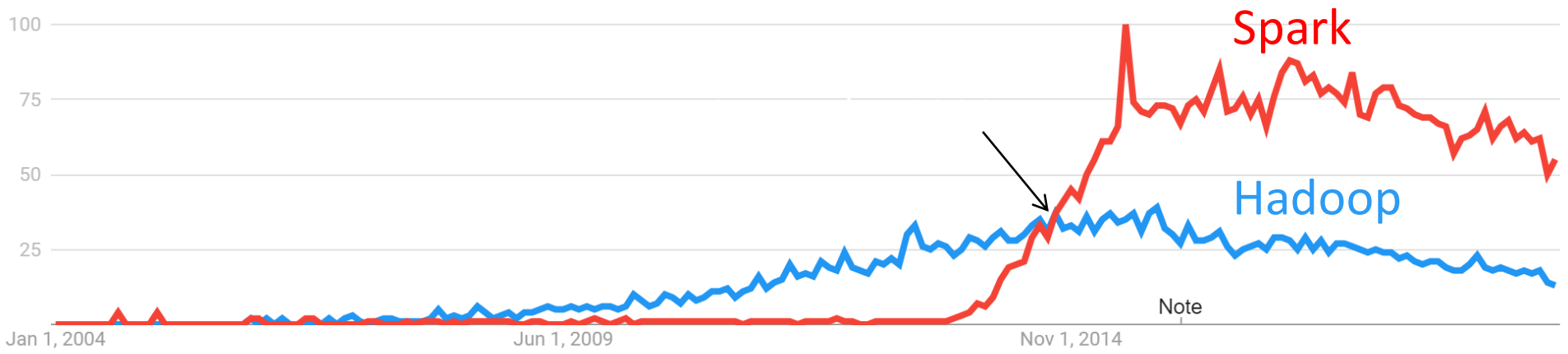
Brief history:

Developed at UC Berkeley AMPLab in 2009

Open-sourced in 2010

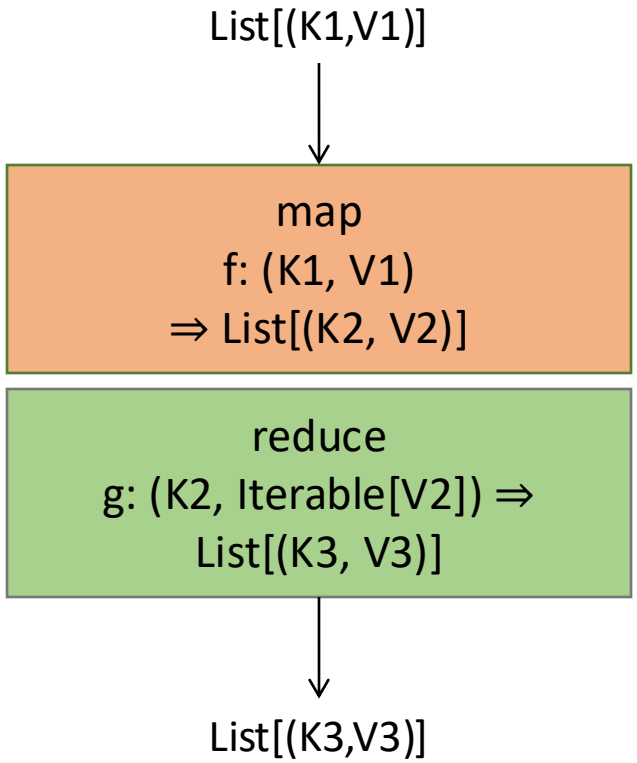
Became top-level Apache project in February 2014

Spark vs. Hadoop




APACHE SPARK
SO HOT RIGHT NOW
memegenerator.net

MapReduce


List[(K1,V1)]
map
f: (K1, V1)
⇒ List[(K2, V2)]
reduce
g: (K2, Iterable[V2]) ⇒
List[(K3, V3)]
List[(K3,V3)]

Important Term

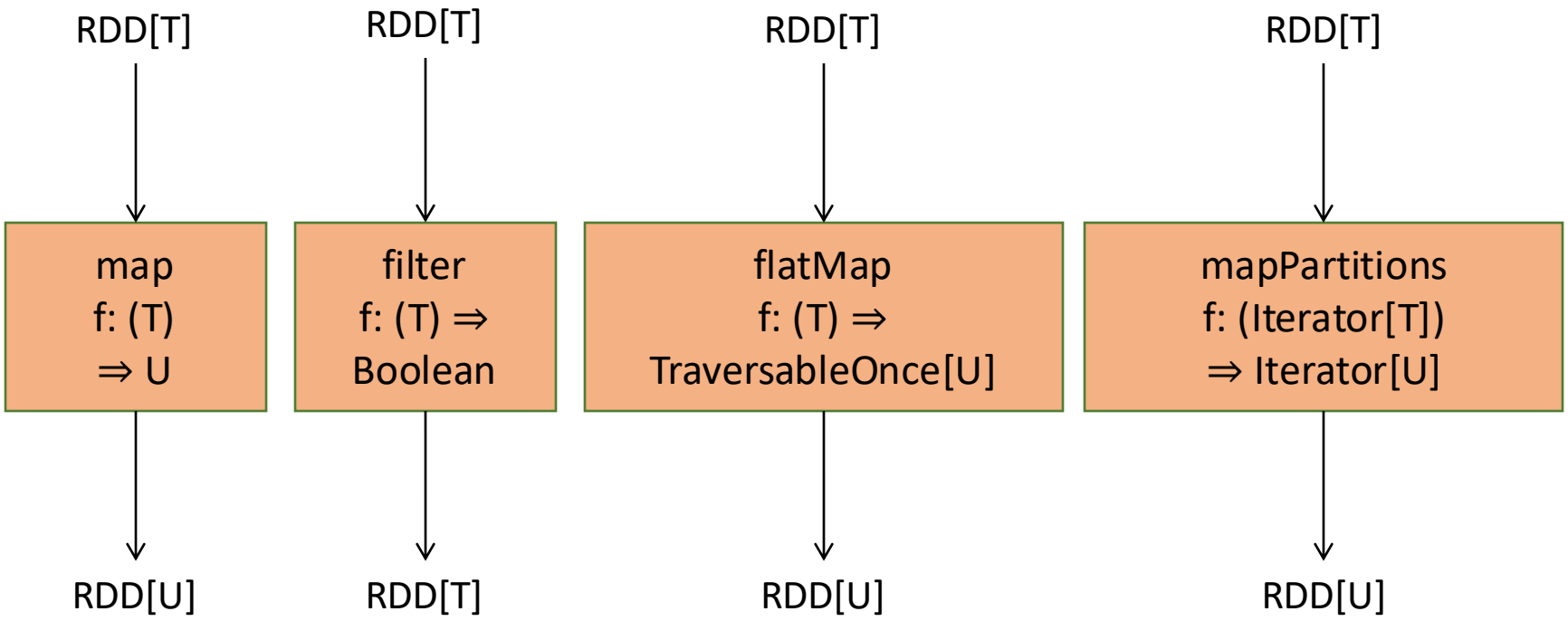
Resilient Distributed Dataset – RDD

RDD[T] – a collection of values of type T

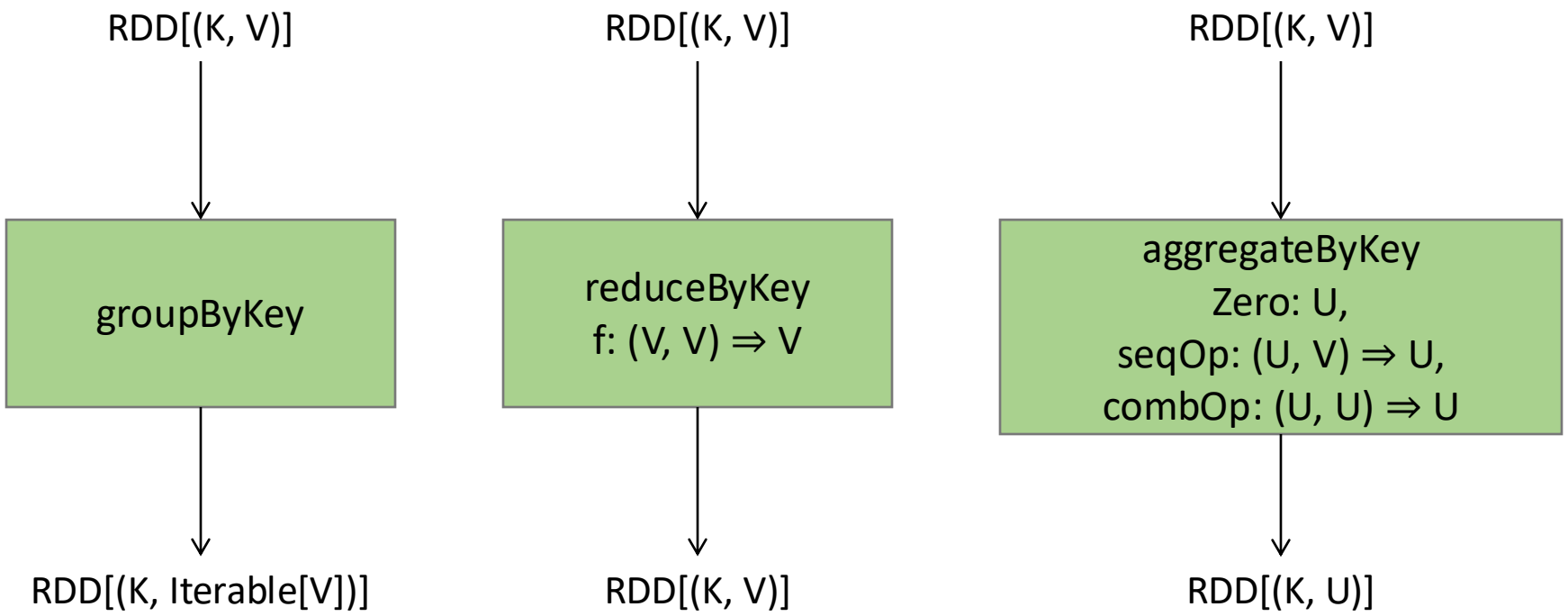
RDDs are divided into “partitions”

Workers operate on partitions independently.

Map-like Operations

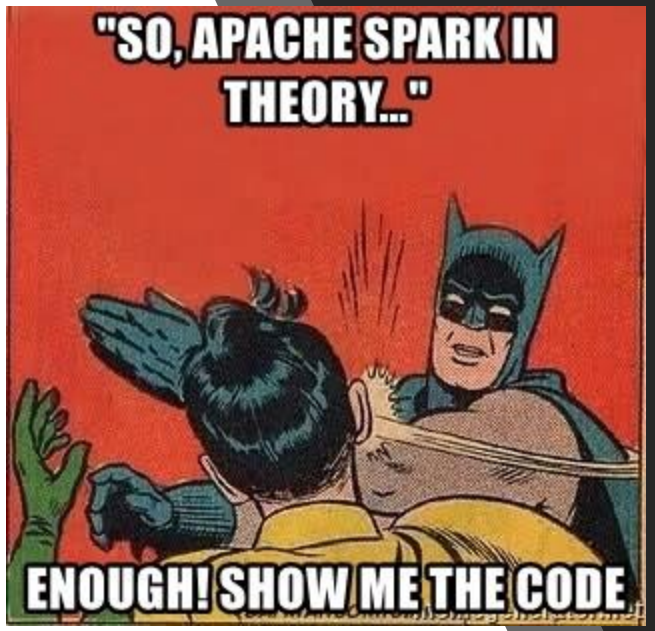


Reduce-like Operations



And many other operations!

Interactive Demo Time!



<Dan, showing off Spark Shell / PySpark>

Introduction to Apache Spark

Slides from: Patrick Wendell – Databricks

Memes from: Ali Abedi

What is Spark?

Fast and **Expressive** Cluster Computing Engine Compatible with Apache Hadoop

Up to **10×** faster on disk, **100×** in memory

2-5× less code

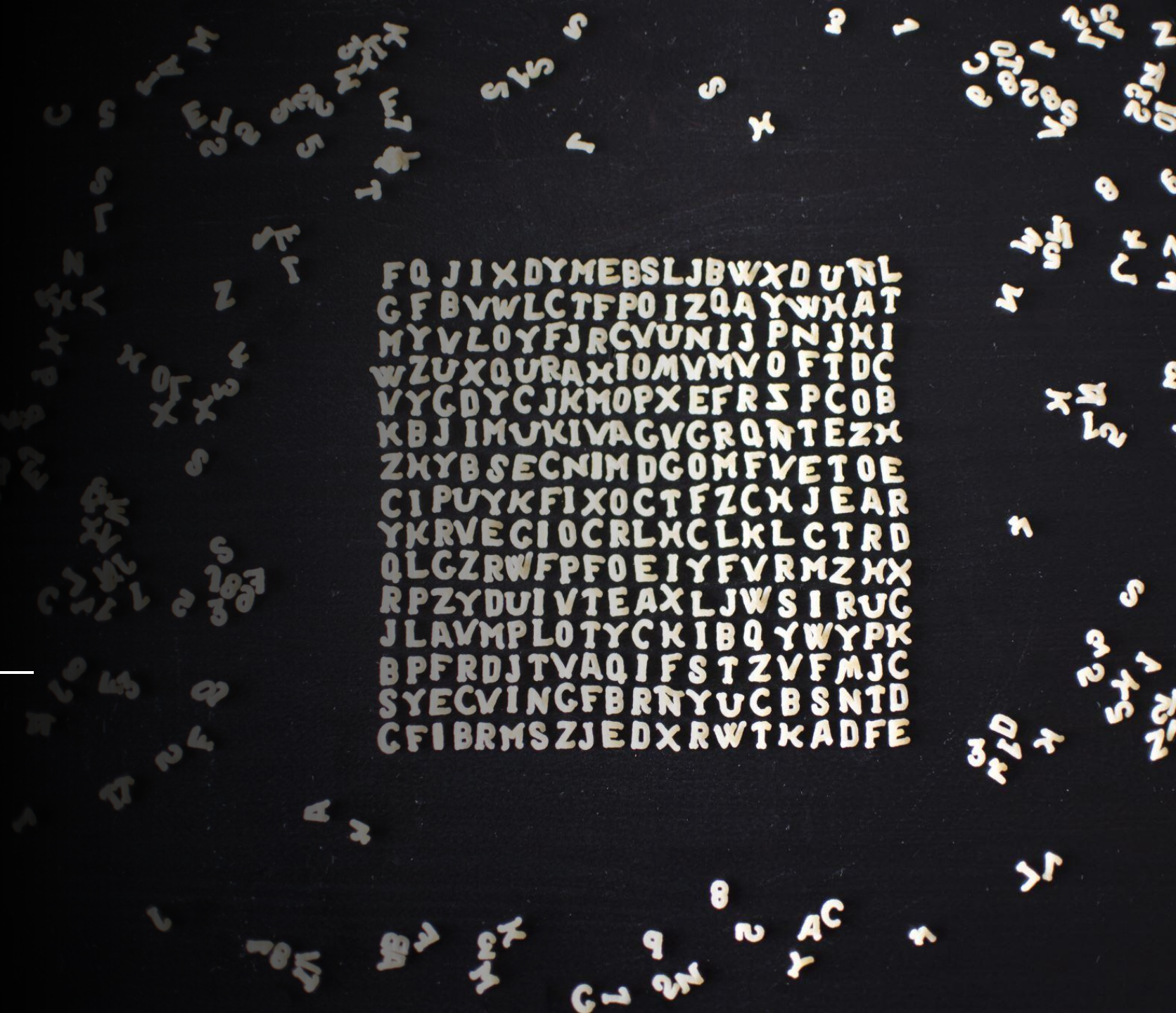
Efficient

- General execution graphs
- In-memory storage

Usable

- Rich APIs in Java, Scala, Python
- Interactive shell

Spark Programming Model



Key Concept: RDD's

Write programs in terms of **operations** on **distributed datasets**

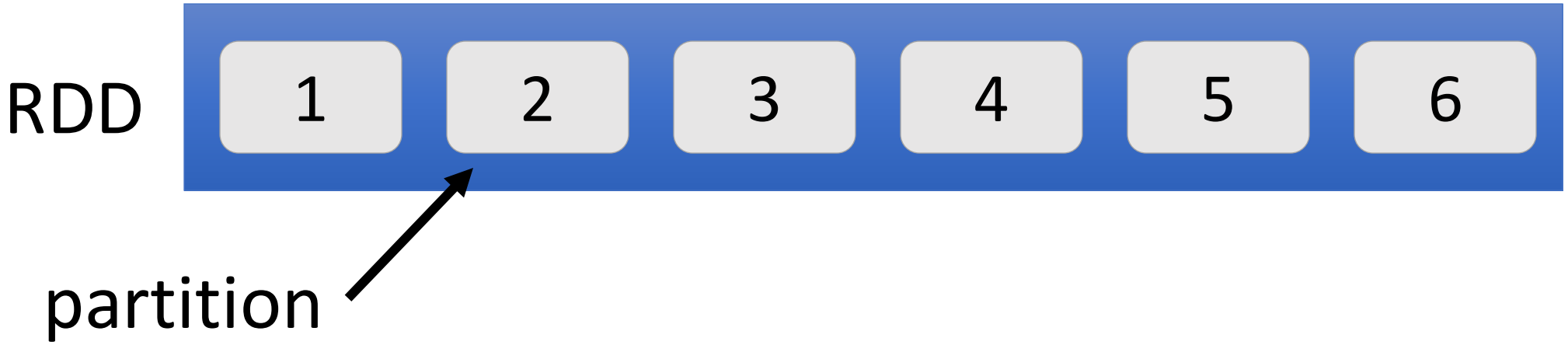
Resilient Distributed Datasets

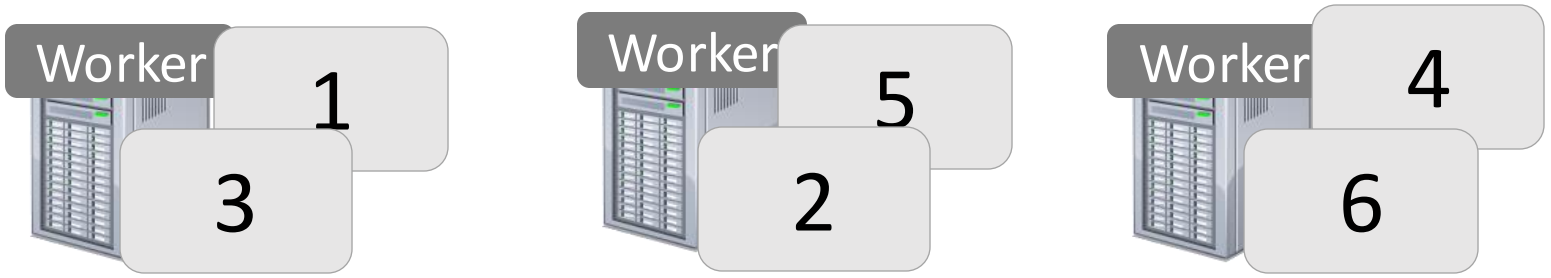
- Collections of objects spread across a cluster, stored in RAM or on Disk
- Built through parallel transformations
- Automatically rebuilt on failure

Operations

- Transformations (e.g. map, filter, groupBy)
- Actions (e.g. count, collect, save)

RDD structure





Example: Log Mining

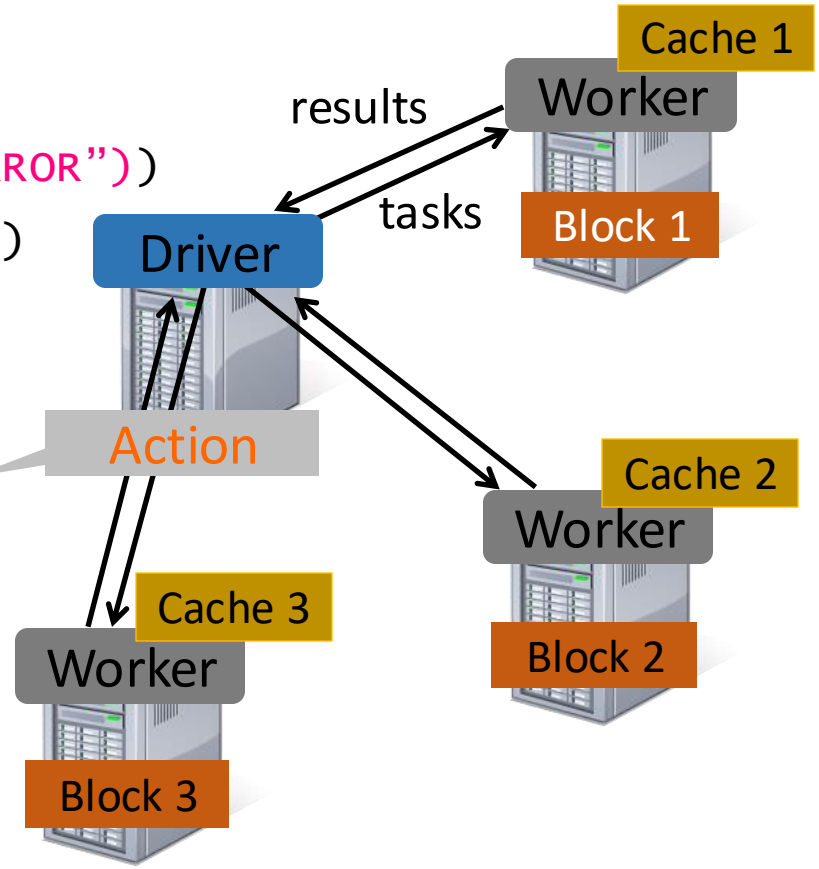
Load error messages from a log into memory, then interactively search for various patterns

Transformed RDD

```
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: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
. . .
```

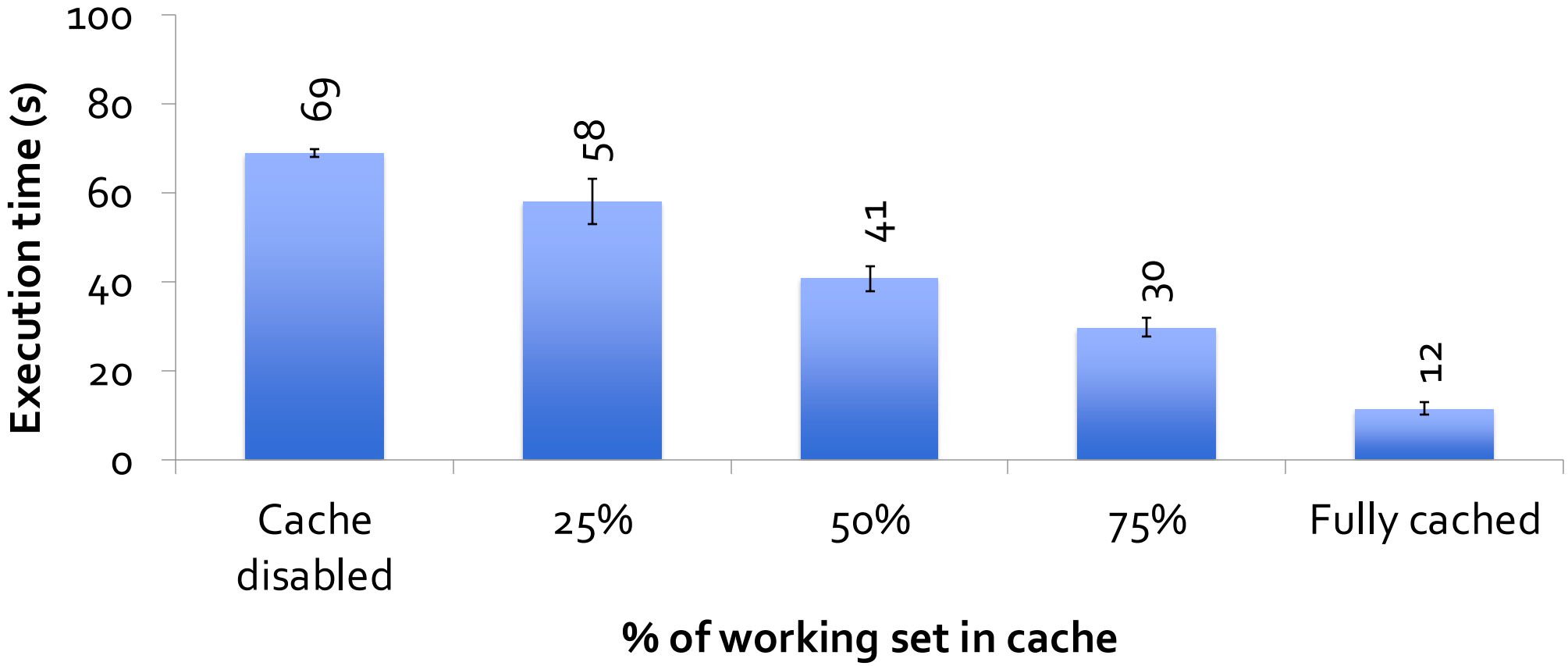
Action



Full-text search of Wikipedia

- 60GB on 20 EC2 machine
- 0.5 sec vs. 20s for on-disk

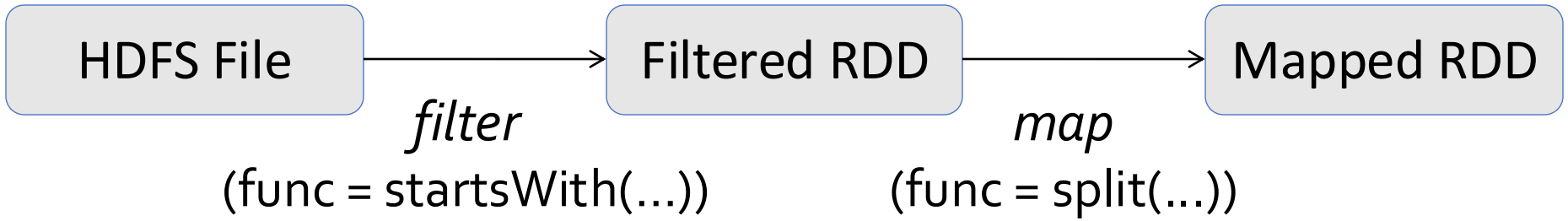
Impact of Caching on Performance



Fault Recovery

RDDs track *lineage* information that can be used to efficiently recompute lost data

```
msgs = textFile.filter(lambda s: s.startsWith("ERROR"))
               .map(lambda s: s.split("\t")[2])
```



Programming with RDD’s

SparkContext

- Main entry point to Spark functionality
- Available in shell as variable `sc`
- In standalone programs, you'd make your own

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")
```

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}

# Map each element to zero or more others
>nums.flatMap(lambda x: => range(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(lambda 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 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:

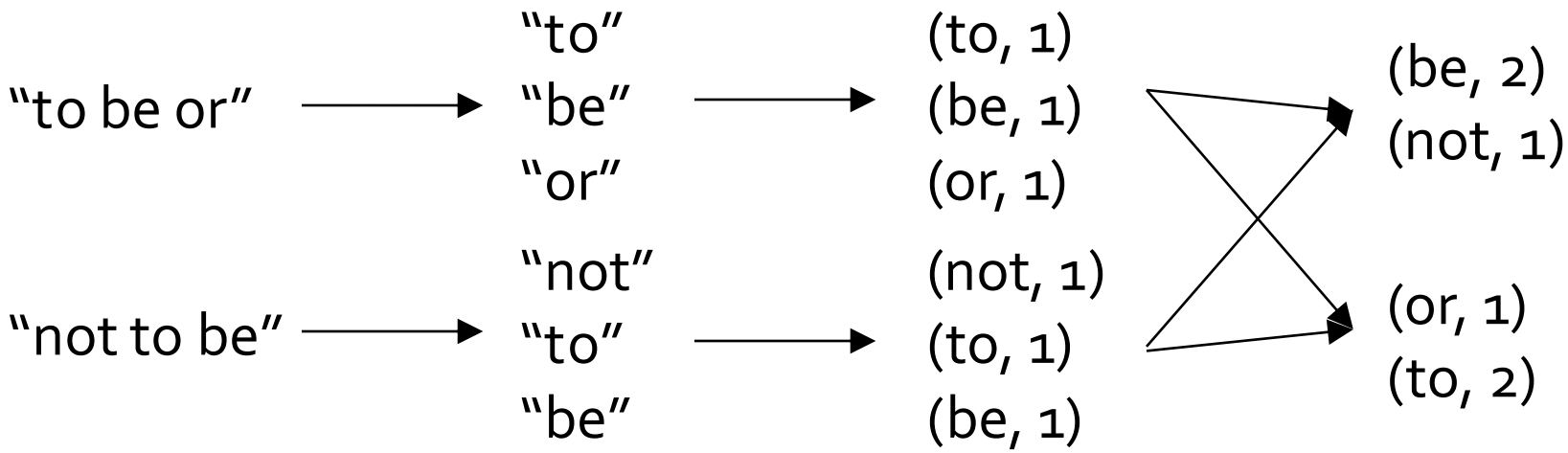
```
Tuple2 pair = new Tuple2(a, b);
pair._1 // => a
pair._2 // => b
```

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)}
```

Word Count (Python)

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



Word Count (Scala)

```
val textFile =
sc.textFile(“hamlet.txt”)

textFile
  .flatMap(line => line.split(“ “))
  .map(word => (word, 1))
  .reduceByKey((x, y) => x + y)
  .saveAsTextFile(“results”)
```

(Alternative Scala)

```
val textFile =
sc.textFile(“hamlet.txt”)

textFile
  .flatMap(_.split(“ “))
  .map((_, 1))
  .reduceByKey(_ + _)
  .saveAsTextFile(“results”)
```

In Scala, underscores mean “this expression is the body of an anonymous function”

“_ + _” means the same as “(x, y) => x + y”


SAY "WORD COUNT"
ONE MORE TIME...
imgflip.com

Other Key-Value Operations

```
> 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”, ([“1.2.3.4”, “1.3.3.1”], [“Home”]))
  # (“about.html”, ([“3.4.5.6”], [“About”]))
```

Setting the Level of Parallelism

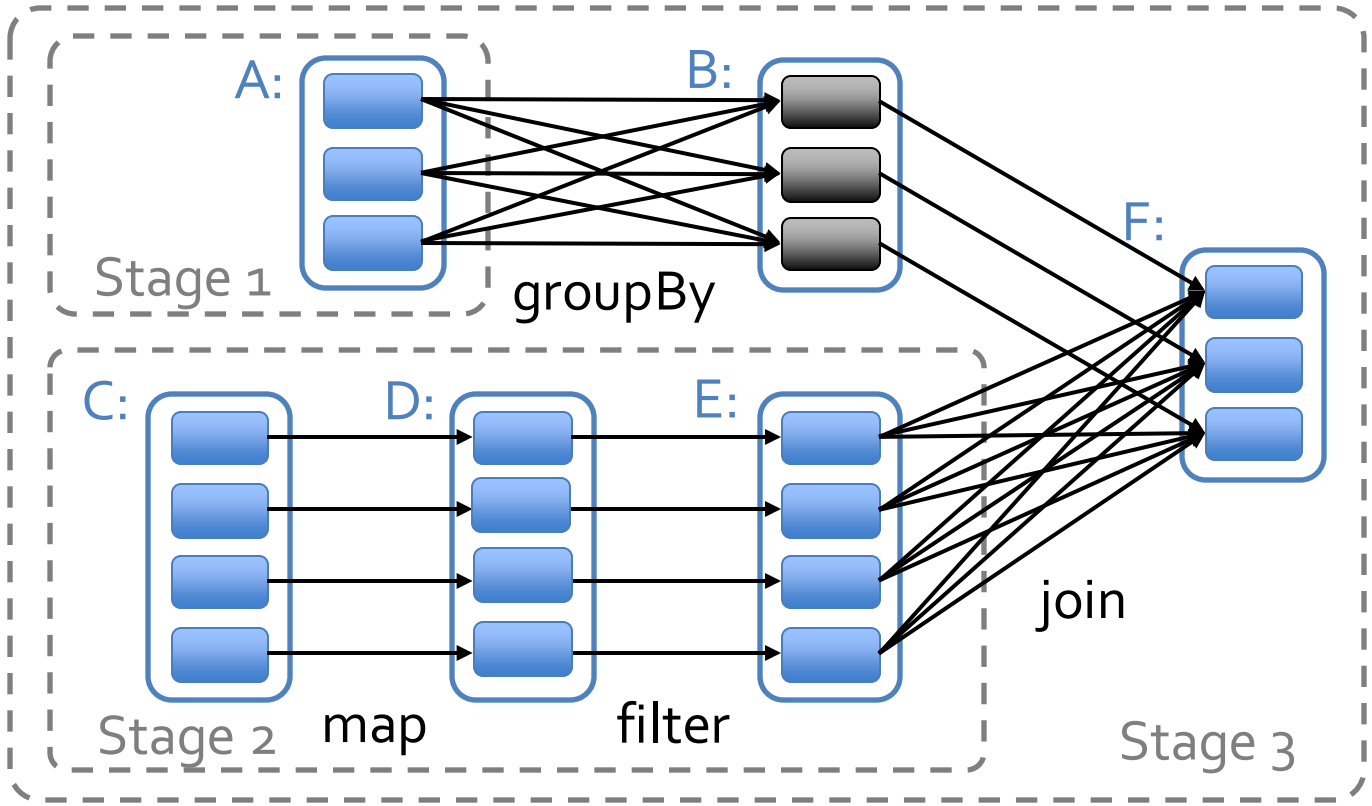
All the pair RDD operations take an optional second parameter for number of tasks

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

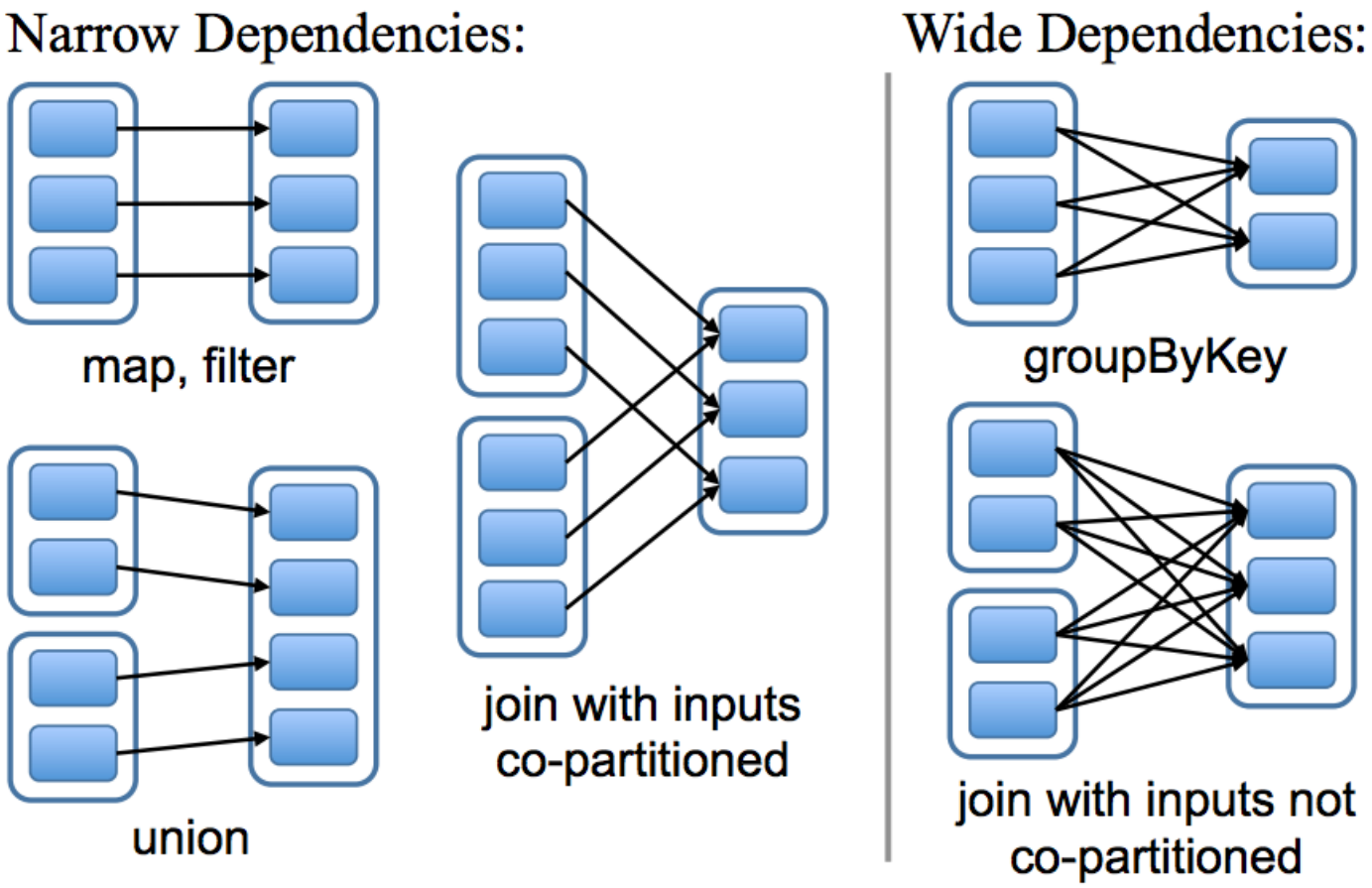

i have
a big
data problem
i
write
spark code
it
runs
and
runs...
imgflip.com

Under The Hood: DAG Scheduler

- General task graphs
- Automatically pipelines functions
- Data locality aware
- Partitioning aware to avoid shuffles



Physical Operators



More RDD Operators

- map
- filter
- groupBy
- sort
- union
- join
- leftOuterJoin
- rightOuterJoin

- reduce
- count
- fold
- reduceByKey
- groupByKey
- cogroup
- cross
- zip

sample

take

first

partitionBy

mapWith

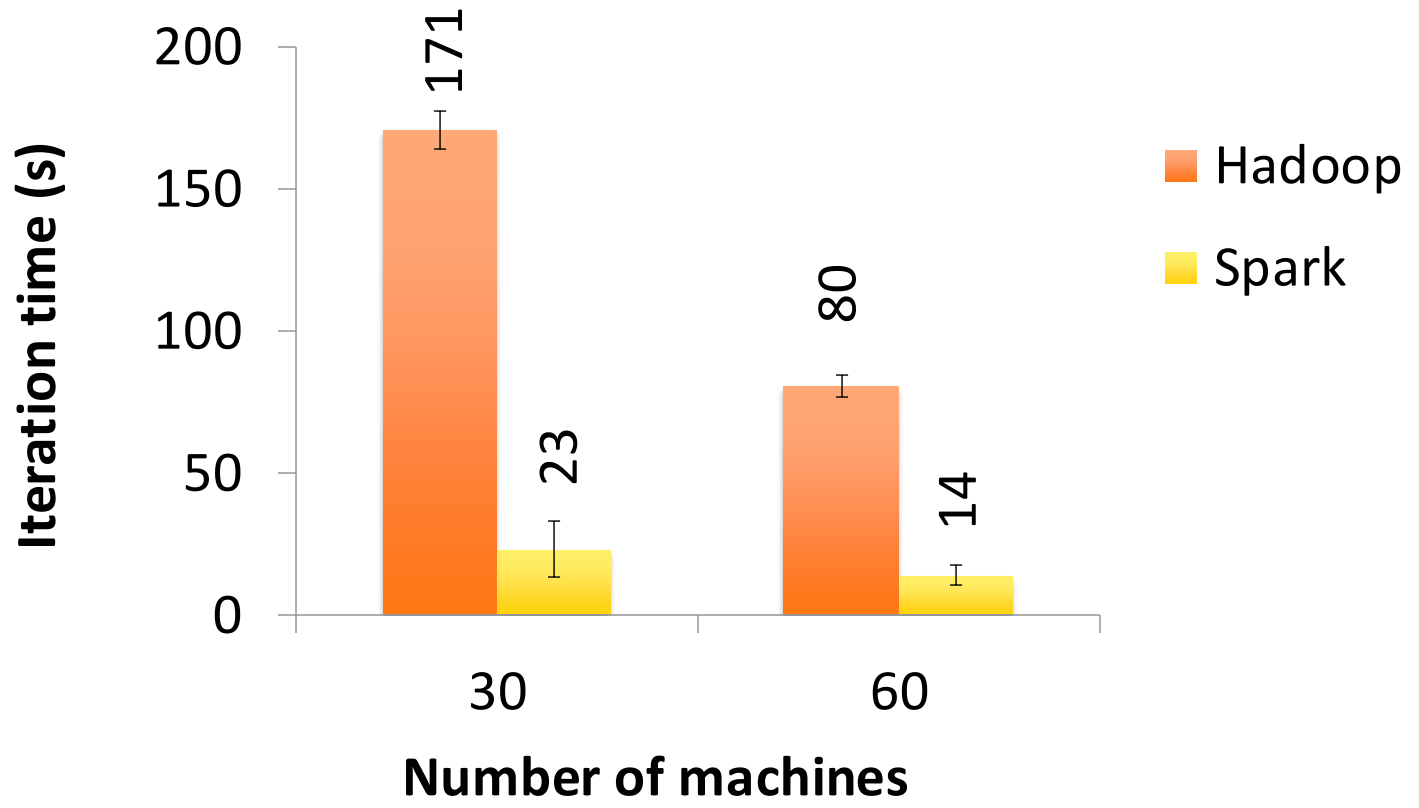
pipe

save ...

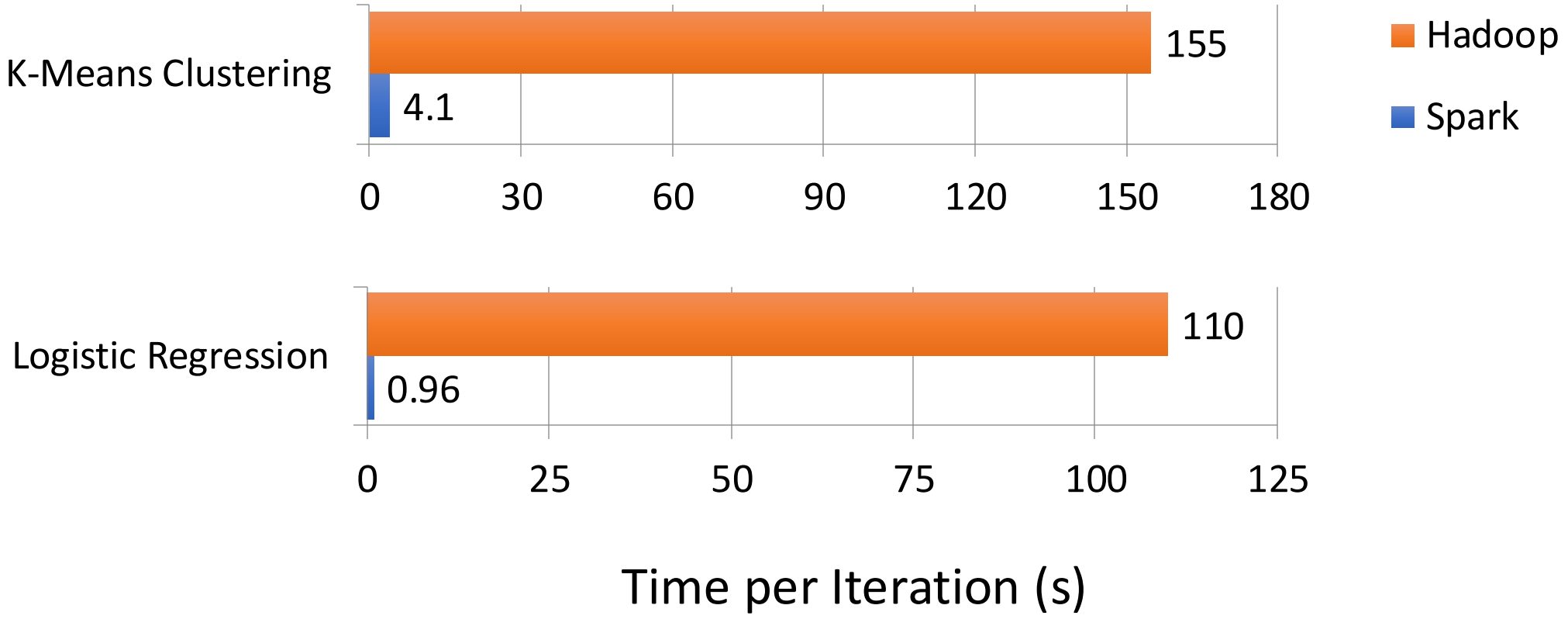

Learn Apache Spark. You Must.


Performance

PageRank Performance



Other Iterative Algorithms




Hadoop Ecosystem and Spark

YARN

Hadoop's (original) limitations:

Can only run MapReduce

What if we want to run other distributed frameworks?

YARN = Yet-Another-Resource-Negotiator

Provides API to develop any generic distributed application

Handles scheduling and resource request

MapReduce (MR2) is one such application in YARN

Hadoop v1.0

MapReduce
Data Processing
& Resource Management

HDFS
Distributed File Storage

Hadoop v2.0

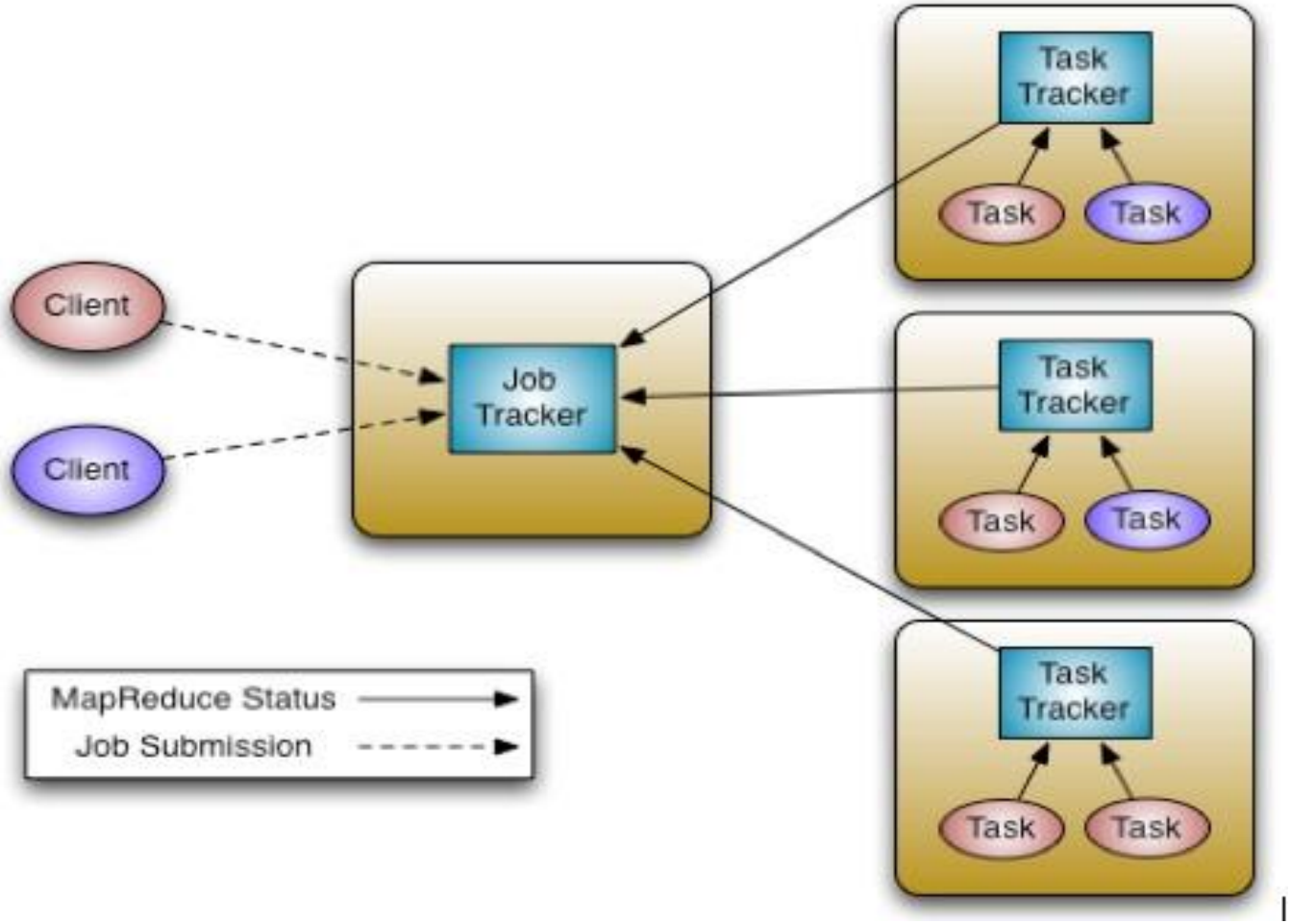
MapReduce

Other Data Processing Frameworks

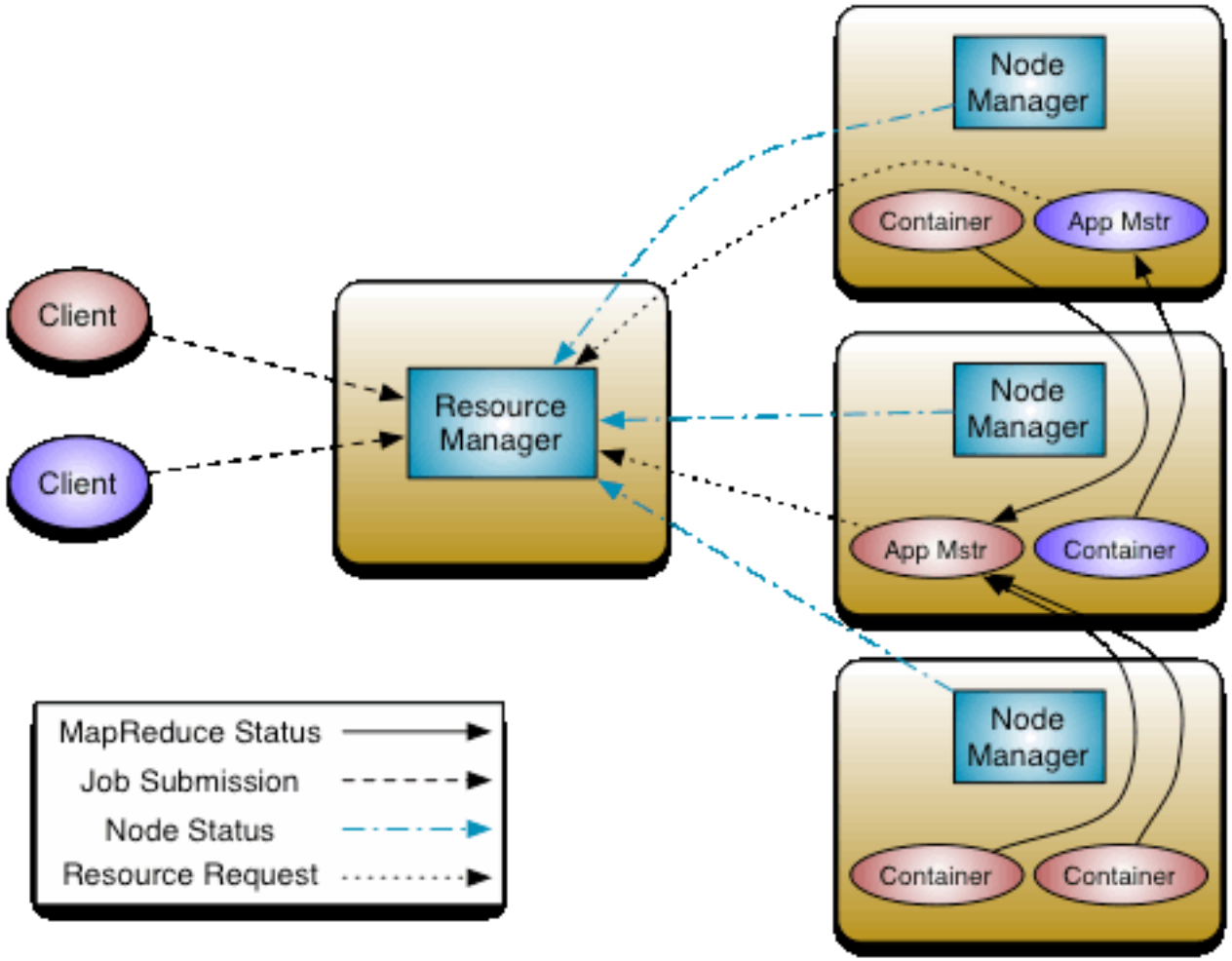
YARN
Resource Management

HDFS
Distributed File Storage

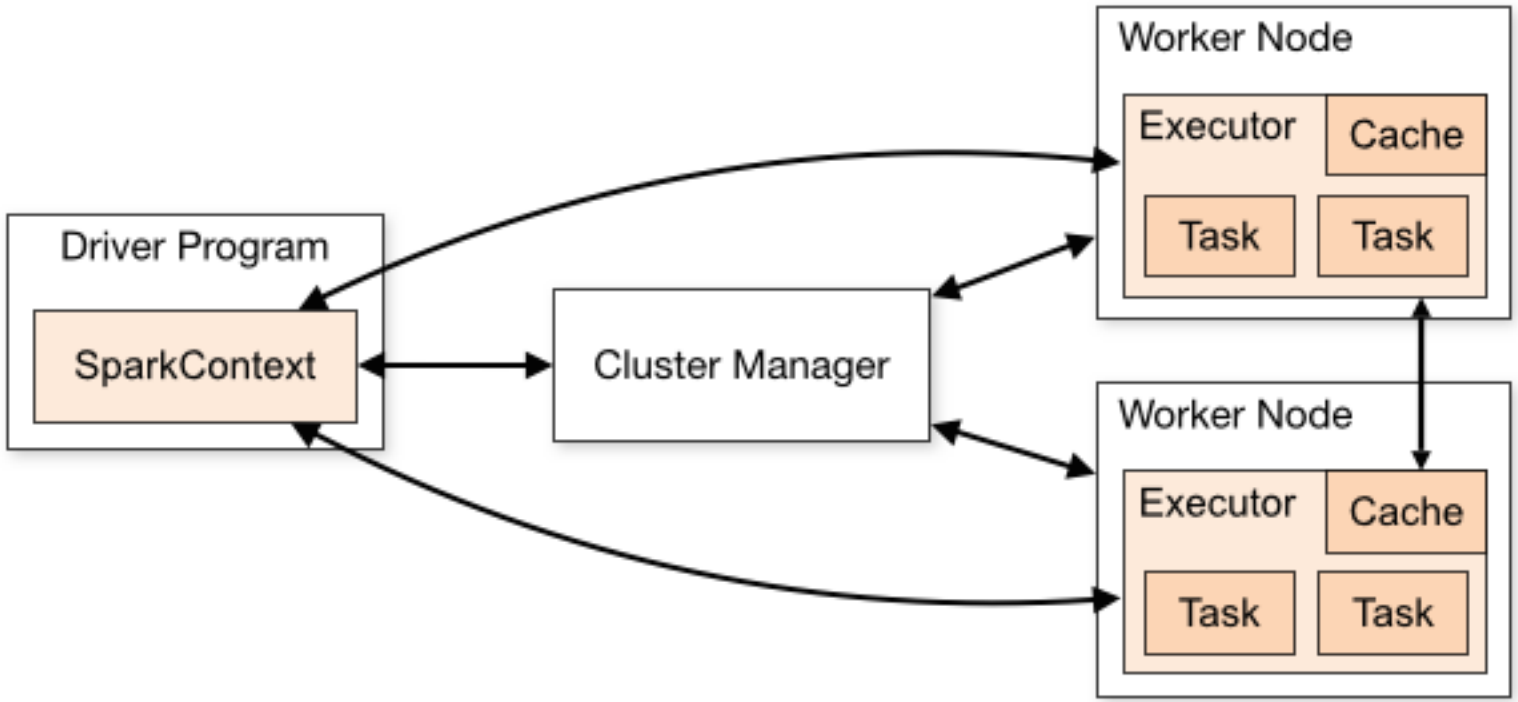
Hadoop v1.0


Client
Client
Job Tracker
Task Tracker
Task
Task
Task Tracker
Task
Task
Task Tracker
Task
Task
MapReduce Status
Job Submission

Hadoop v2.0


Client
Client
Resource Manager
Node Manager
Container
App Mstr
Node Manager
App Mstr
Container
Node Manager
Container
Container
MapReduce Status
Job Submission
Node Status
Resource Request

Spark Architecture



Important – There are multiple tasks per executor

Why is this important?

To work, the Spark driver must send relevant code (Scala or Python) to run each **task**.

```
thresh = 5
myRdd.map(lambda x: x >= thresh)
```

The lambda “captures” thresh, so it gets packaged up too. (That’s bad if it’s large)

Broadcast

If you Broadcast a value, then Spark only sends one copy per Executor (worker machine) not per Task

```
thresh = sc.broadcast(5)
myRdd.filter(lambda x: x > thresh.value)
```

(It makes no difference here, but would if broadcasting a lookup table)

Constant means Constant

Broadcast variables are read-only

```
thresh = sc.broadcast(5)
thresh.value = 6
Error: value is not a member of …Broadcast[int]
Error: value is not assignable
```

(Global variables are too, but will silently fail)

Accumulators

A Broadcast variable carries information from Driver to Executor

What if we want communication from Executor back to Driver?

A: Accumulator

Counter Accumulators (Python)

```
lineCounter = sc.accumulator(0)

def split_and_count(line):
    lineCounter.add(1)
    return line.split()

myRdd.map(split_and_count). …
lineCounter.value()
```

Counter Accumulators (Scala)

val lineCounter = sc.longAccumulator

```
def split_and_count(line : String) = {
   lineCounter.add(1)
   line.split()
}

myRdd.map(split_and_count). …
lineCounter.value
```

Types of Accumulator

longAccumulator, doubleAccumulator

(In Python, they’re just called accumulator)

Used for accumulating numerical values

Driver can inspect the value (and take average of values accumulated)

Workers can only write

Partitioners

By default Spark shuffles use a hash partitioner (just like MapReduce)

Or sometimes a range partitioner (used for sorting)

Also like MapReduce, can override.

Partitioners (Scala)

```
class myPartitioner(override val numPartitions : Int) {
  def getPartition(key : Any) : Int = key match {
    case x : Int => whatever logic you want % numPartitions
    case _ => throw an error
  }
}
```

rdd.reduceByKey(new myPartitioner(5), _ + _)

Partitioners (PySpark)

PySpark doesn't use Partitioner objects, just a partition function

This function returns an integer, Spark will take "X % numPartitions"

```
def myPartitionFunc(key):
  return {your logic here}

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

Movie rating example

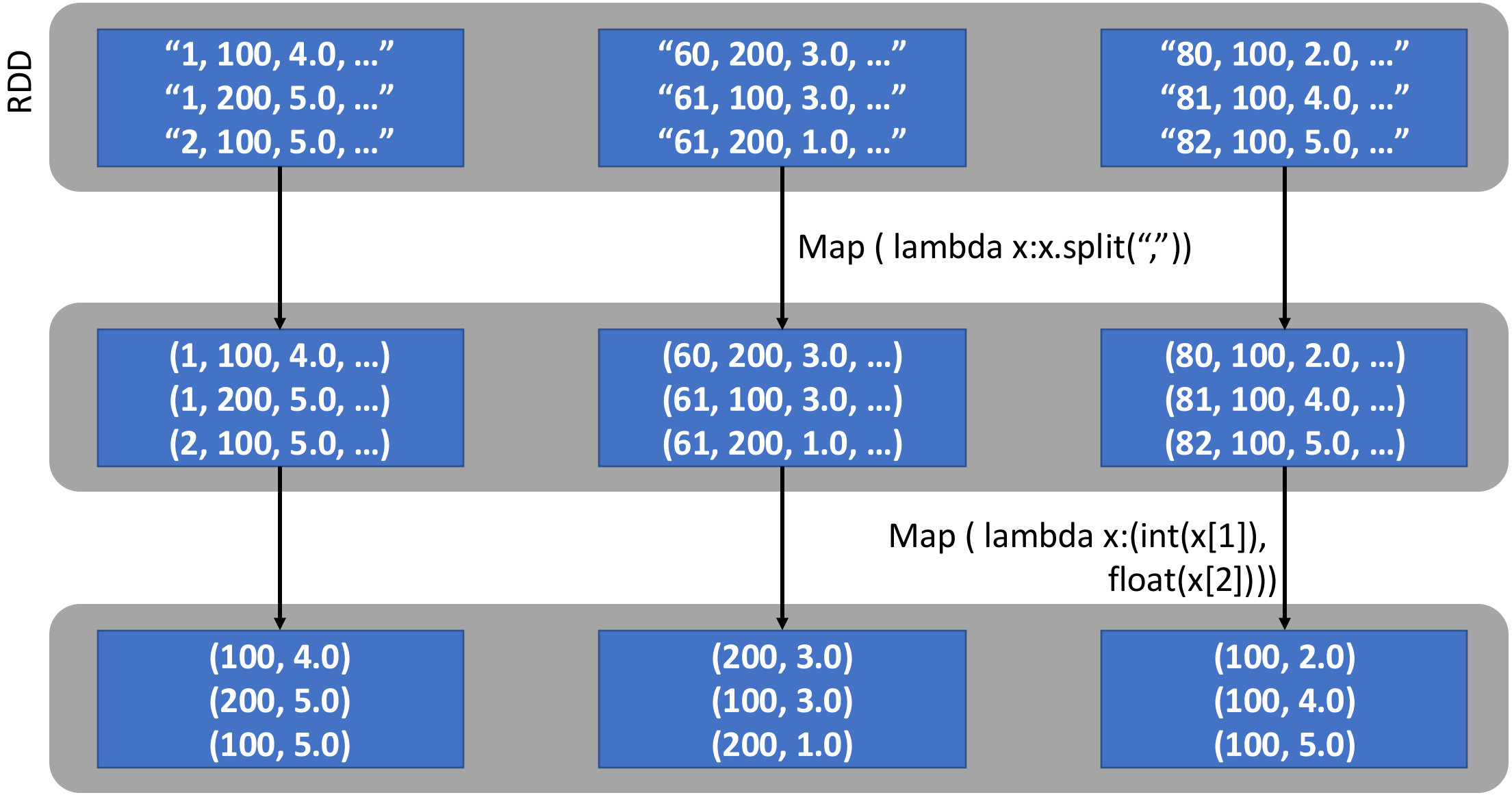
(Now I’m taking from Ali again)

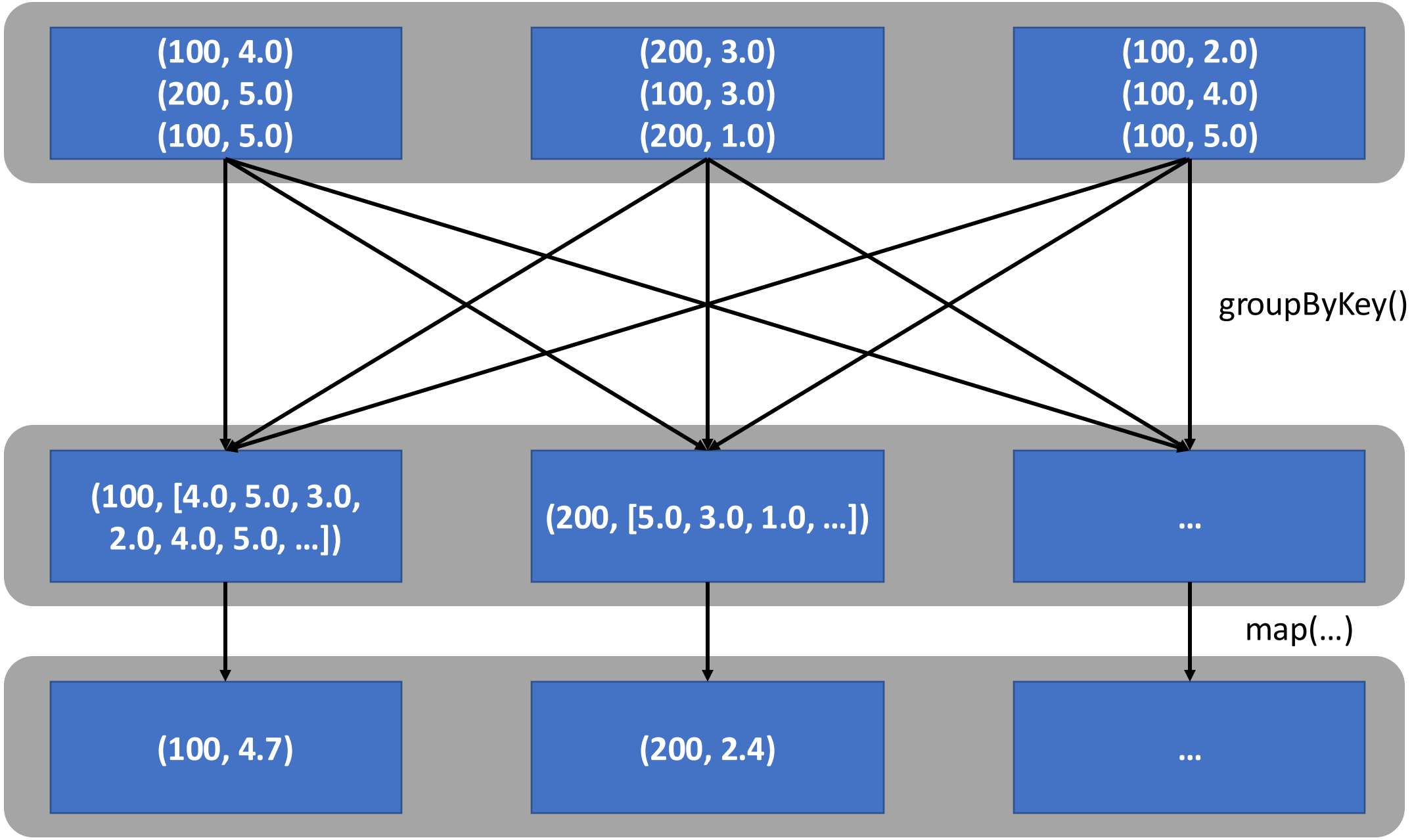
Input Format

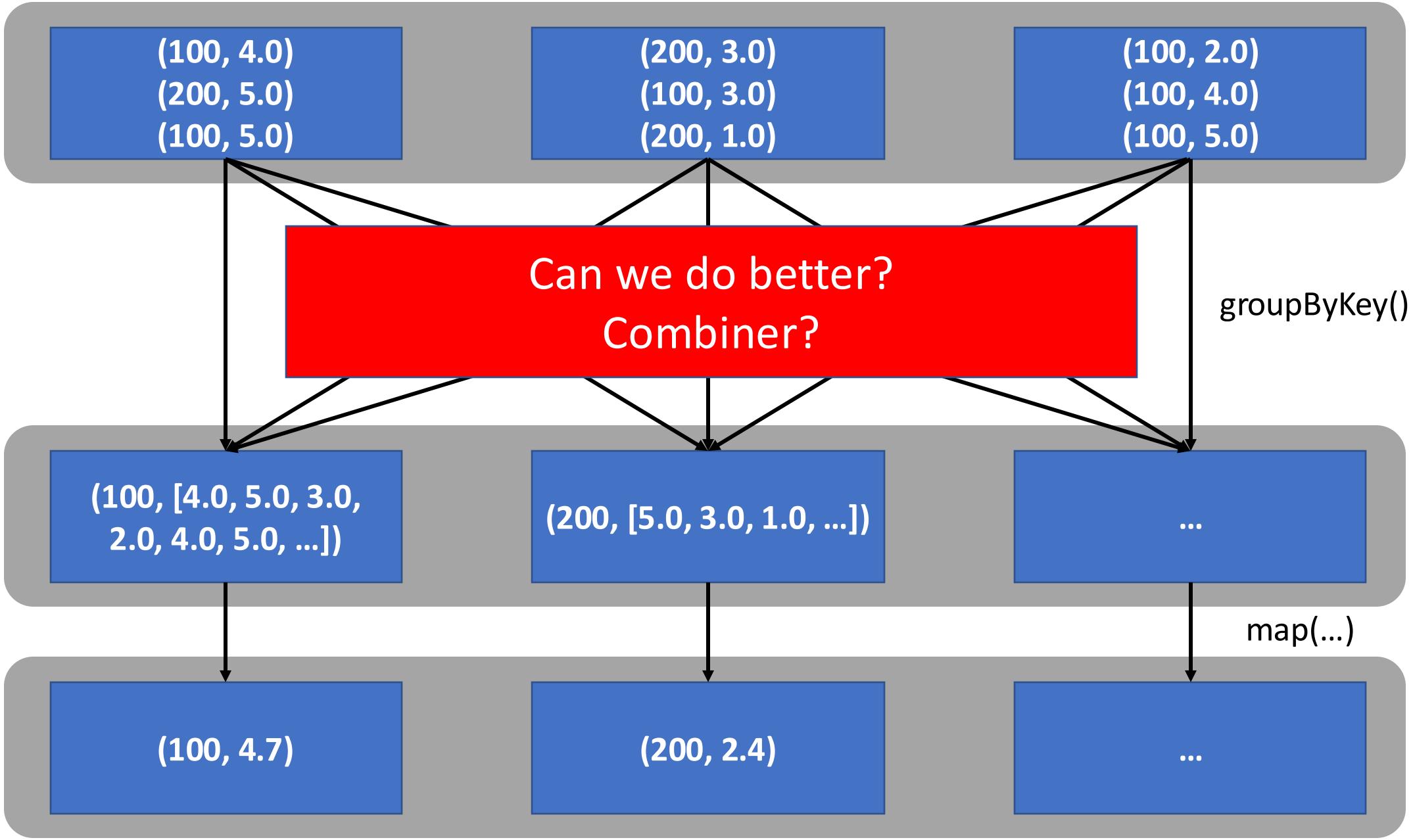
CSV file
Fields:

- User ID (unique key per user)
- Movie ID (unique key per movie)
- Rating (1-5 stars)
- Text of review (optional)

e.g.
“1, 100, 3.5, ‘s aight”


RDD
"1, 100, 4.0, …"
"1, 200, 5.0, …"
"2, 100, 5.0, …"
"60, 200, 3.0, …"
"61, 100, 3.0, …"
"61, 200, 1.0, …"
"80, 100, 2.0, …"
"81, 100, 4.0, …"
"82, 100, 5.0, …"
Map (lambda x:x.split(","))
(1, 100, 4.0, …)
(1, 200, 5.0, …)
(2, 100, 5.0, …)
(60, 200, 3.0, …)
(61, 100, 3.0, …)
(61, 200, 1.0, …)
(80, 100, 2.0, …)
(81, 100, 4.0, …)
(82, 100, 5.0, …)
Map (lambda x:(int(x[1]),
float(x[2])))
(100, 4.0)
(200, 5.0)
(100, 5.0)
(200, 3.0)
(100, 3.0)
(200, 1.0)
(100, 2.0)
(100, 4.0)
(100, 5.0)


(100, 4.0)
(200, 5.0)
(100, 5.0)
(200, 3.0)
(100, 3.0)
(200, 1.0)
(100, 2.0)
(100, 4.0)
(100, 5.0)
groupByKey()
(100, [4.0, 5.0, 3.0, 2.0, 4.0, 5.0, ...])
(200, [5.0, 3.0, 1.0, ...])
...
map(...)
(100, 4.7)
(200, 2.4)
...


(100, 4.0)
(200, 5.0)
(100, 5.0)
(200, 3.0)
(100, 3.0)
(200, 1.0)
(100, 2.0)
(100, 4.0)
(100, 5.0)
Can we do better?
Combiner?
groupByKey()
(100, [4.0, 5.0, 3.0, 2.0, 4.0, 5.0, …])
(200, [5.0, 3.0, 1.0, …])
…
map(…)
(100, 4.7)
(200, 2.4)
…

Reduce vs reduceByKey

Reduce (MapReduce)

- $(K_2,V_2) \Rightarrow List[K_3,V_3]$
- KVP are partitioned and shuffled by Partitioner
- Reduce job calls reduce on keys in sorted order

reduceByKey (Spark)

- $V \Rightarrow V$
- $RDD[(K,V)] \Rightarrow RDD[(K,V)]$
- Less flexible
  - But does what reduce should normally be used for
- Reduces before shuffle (combiner)
- Reduces after shuffle (reducer)

reduceByKey vs combineByKey

combineByKey gives more fine-grained control (if needed)

RDD[(K,V)].combineByKey(create, append, merge) ⇒ RDD[(K,C)]

create – make a C from a V

append – take a C and add a V to it

merge – combine two C

reduceByKey(reduce) calls combineByKey(identity, reduce, reduce)

reduceByKey vs aggregateByKey

aggregateByKey is between reduceByKey and combineByKey

RDD[(K, V)].aggregateByKey(zero, append, merge) => RDD[(K, C)]

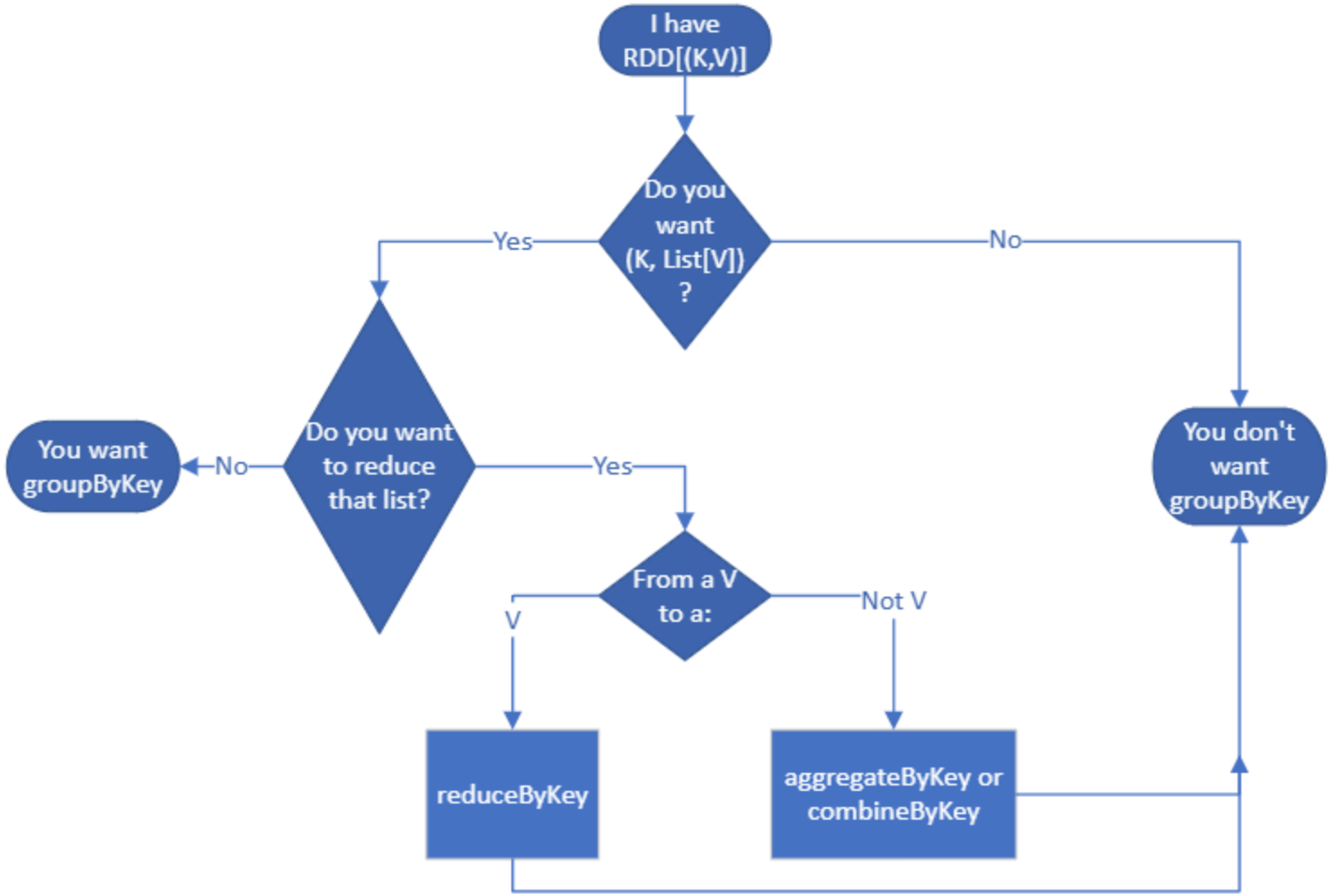
zero – initial (or zero) value [type C]

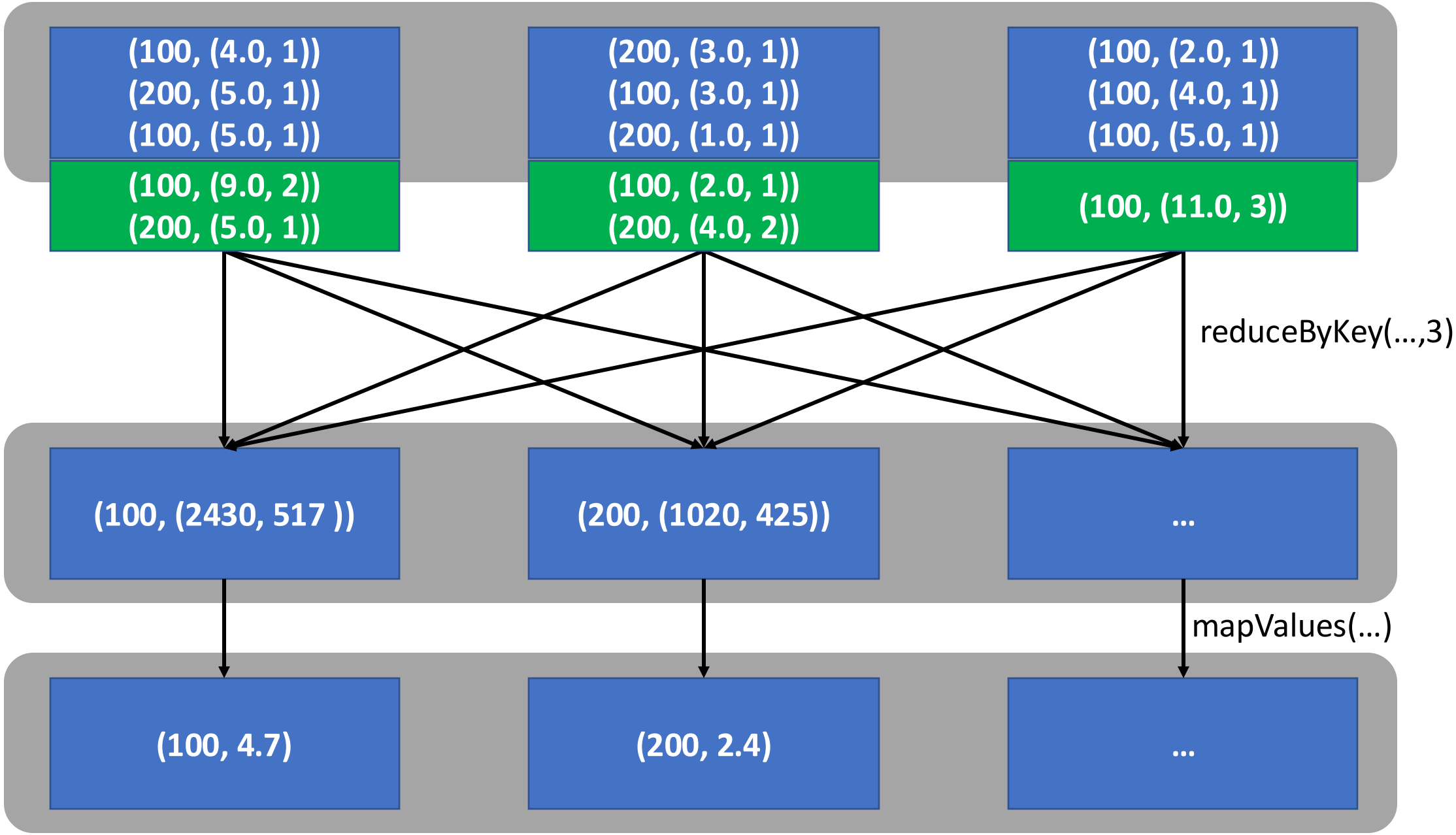
append- take a C and add a V to it

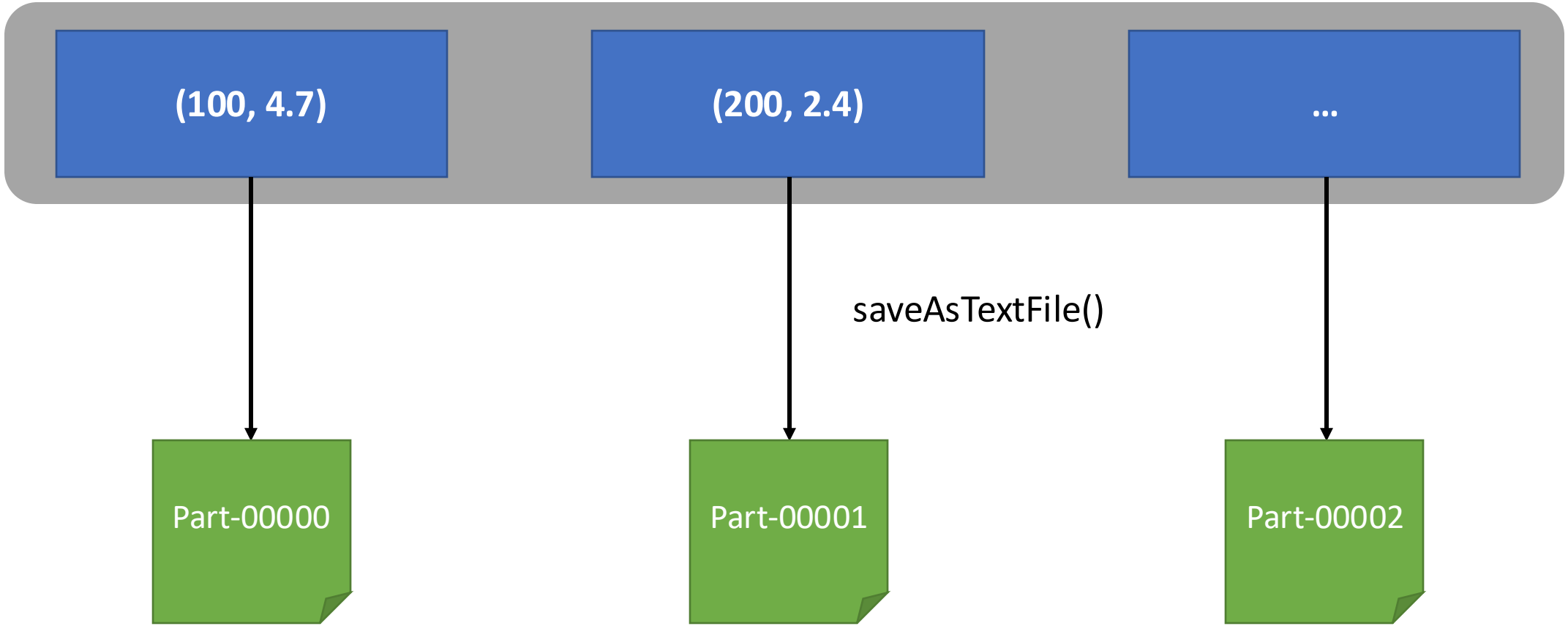
merge- combine two C

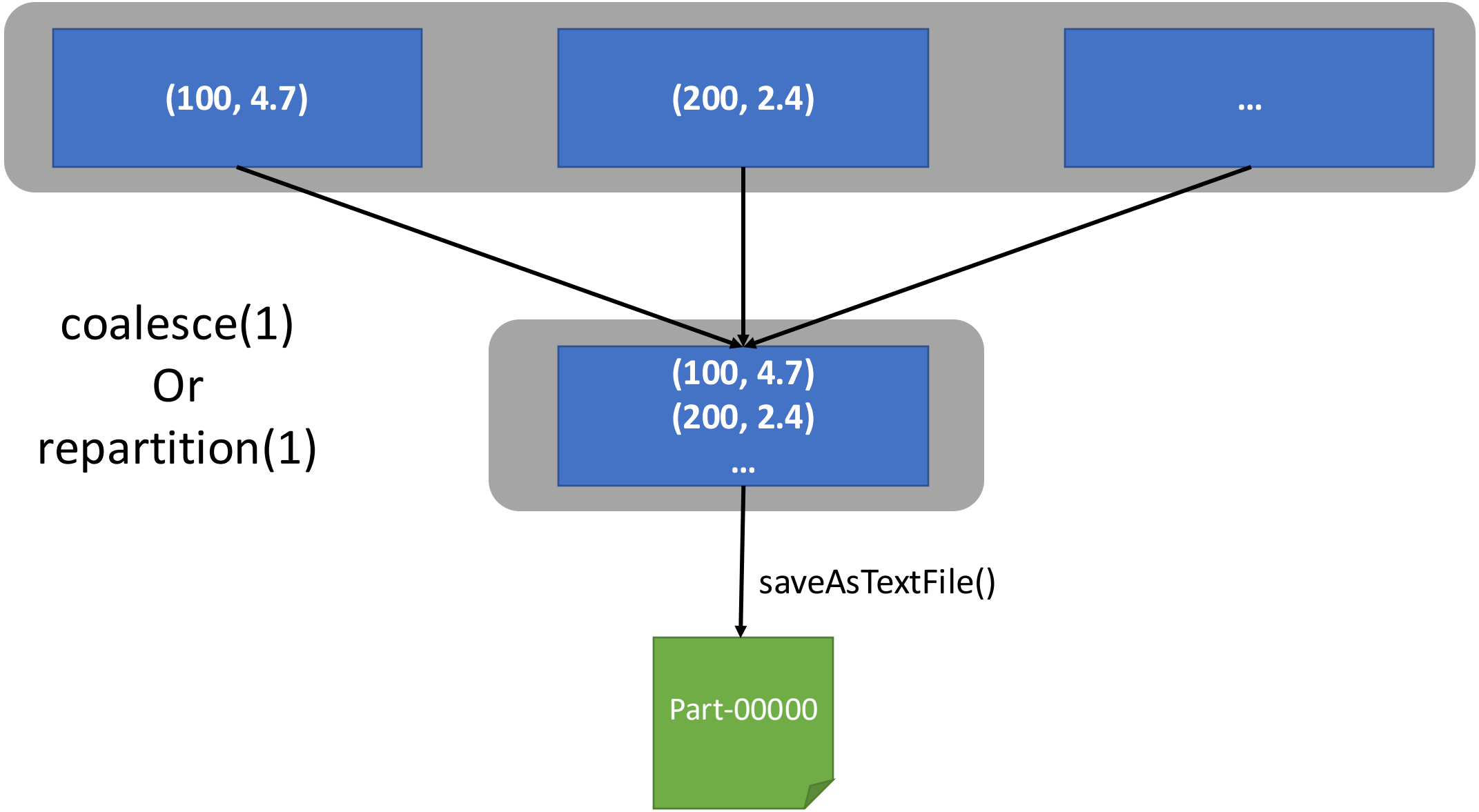
groupByKey

You (probably) don’t want to use it




(100, (4.0, 1))
(200, (5.0, 1))
(100, (5.0, 1))
(200, (3.0, 1))
(100, (3.0, 1))
(200, (1.0, 1))
(100, (2.0, 1))
(100, (4.0, 1))
(100, (5.0, 1))
(100, (9.0, 2))
(200, (5.0, 1))
(100, (2.0, 1))
(200, (4.0, 2))
(100, (11.0, 3))
reduceByKey(...,3)
(100, (2430, 517))
(200, (1020, 425))
...
mapValues(...)
(100, 4.7)
(200, 2.4)
...


(100, 4.7)
(200, 2.4)
...
saveAsTextFile()
Part-00000
Part-00001
Part-00002


(100, 4.7)
(200, 2.4)
...
coalesce(1)
Or
repartition(1)
(100, 4.7)
(200, 2.4)
...
saveAsTextFile()
Part-00000

Just the Code (Scala)

```
sc.textFile("movies.csv").
  map(_.split(",")).
  map(lst => (lst(1).toInt, (lst(2).toDouble,1))).
  reduceByKey({case ((s1,c1), (s2,c2)) =>
                    (s1 + s2, c1 + c2)}).
  mapValues({case (sum, cnt) =>
                  sum / cnt }).
  coalesce(1).
  saveAsTextFile("averages")
```

Just the Code (Python)

```
sc.textFile("movies.csv").\
  map(lambda line: line.split(",")).\
  map(lambda lst:
    (int(lst[1]), (float(lst[2]),1))).\
  reduceByKey(lambda p1, p2:
    (p1[0] + p2[0], p1[1] + p2[1])).\
  mapValues(lambda pair: pair[0] / pair[1])).\
  coalesce(1).\
  saveAsTextFile("averages")
```

D’ya like DAGs?

```
print(rdd.toDebugString())
(1) CoalescedRDD[13] at coalesce at …
 |  MapPartitionsRDD[12] at map at …
 |  ShuffledRDD[11] at reduceByKey at …
 +-(2) MapPartitionsRDD[10] at map at …
    |  MapPartitionsRDD[9] at map at …
    |  movies.csv MapPartitionsRDD[8] at textFile …
    |  movies.csv HadoopRDD[7] at textFile …
```

451 – A2 tips

- For CS451 students – the Hadoop cluster page you viewed on A0 is useful for figuring out what's going on with your Spark jobs!

- If you click "ApplicationManager" you can explore the DAG graphically, including seeing all of the individual tasks created.

- Caution – if your map / flatMap is slow…it might actually be the next stage that's inefficient:
  - RDD.flatMap(…).reduceByKey(…) – as the flatMap emits pairs, they'll be combined by reduceByKey's lambda (like a MapReduce combiner).
  - If this combiner is expensive, it'll look like flatMap is slow