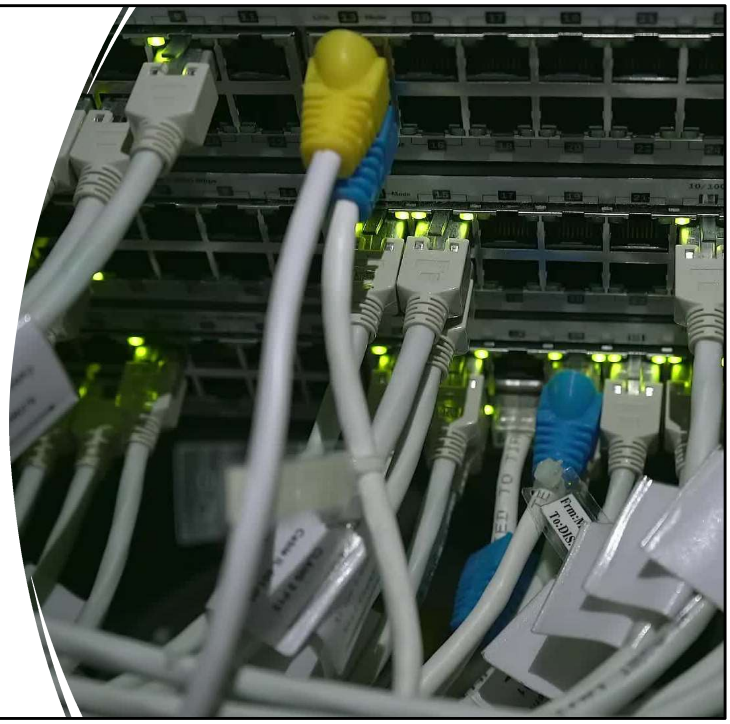
Data-Intensive Distributed Computing CS431/451/631/651

Module 3 – From MapReduce to Spark

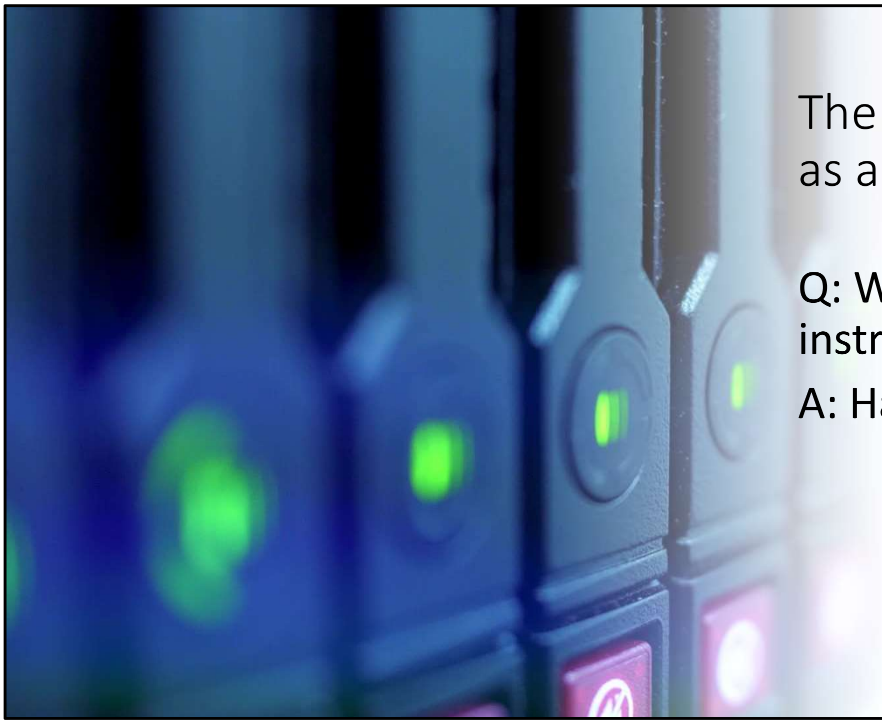


This Module’s Agenda

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

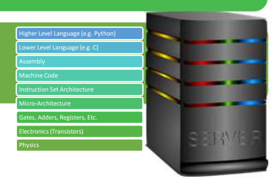
??? <TODAY'S TOPIC>

Hadoop Task

HDFS / Hadoop Framework

Cluster of Computers (Networking)

Individual Servers

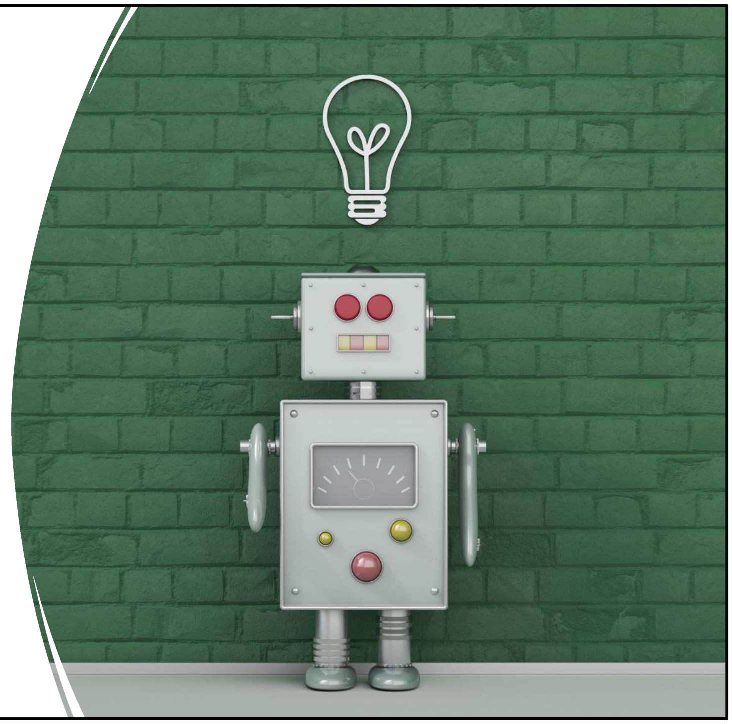


Do you like programming in assembly?

It’s OK if the answer is yes, there’s no judgement here

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

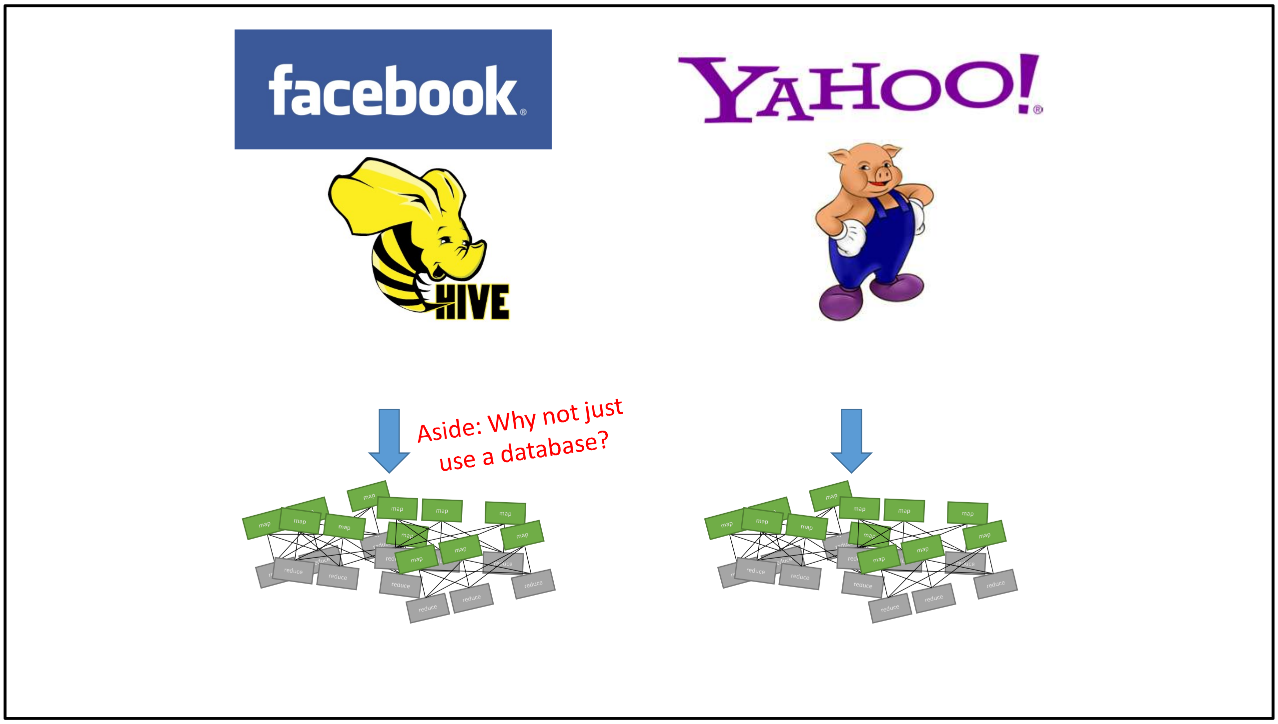
facebook

What we really need is SQL!

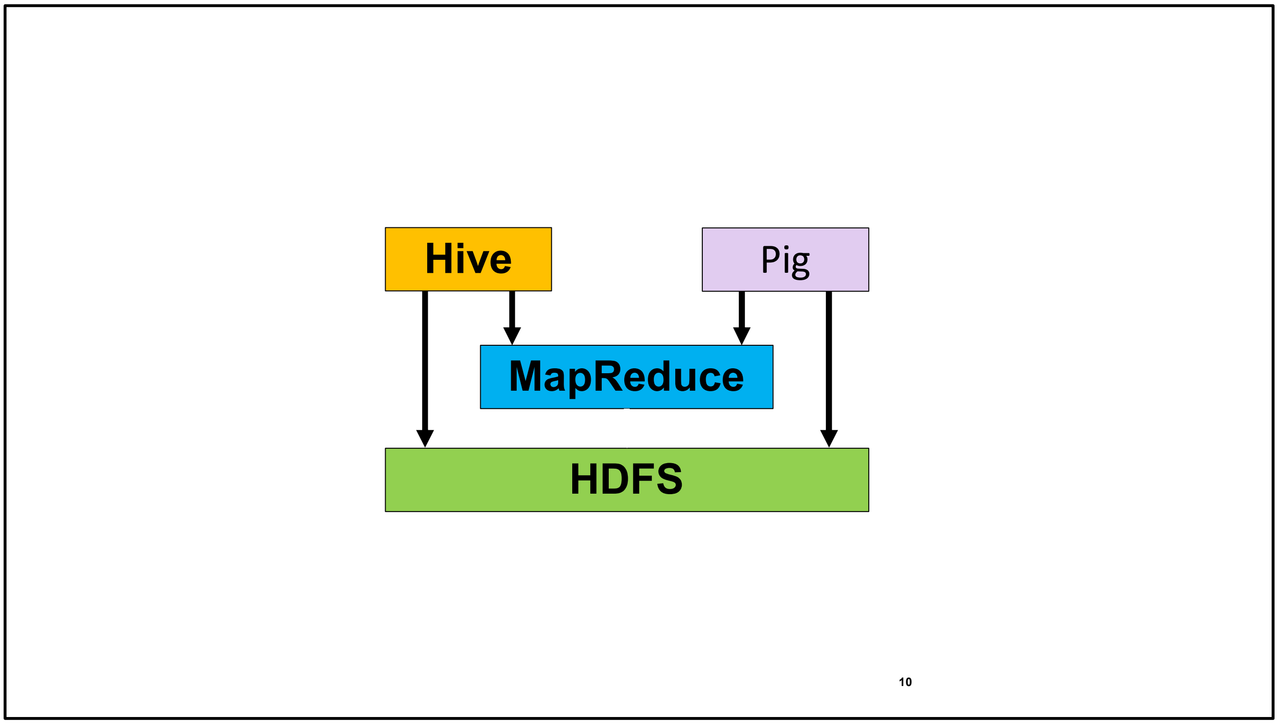
YAHOO!

What we really need is a scripting language!

* - not me personally, but it has been done. Several times.



Both have their place. Hive is on top of MapReduce. It's good for huge datasets that are accessed in a linear fashion. One read, one write. SQL requires lots of read/write access to the data.
SQL – You need OLTP and/or low latency. Less-complicated data sets that need frequent updates
Hive – You don't care about latency, or have huge amounts of data (which means it doesn't matter whether or not you care, you're going to have latency). Batch processing of complicated data sets



Pig and Hive programs are converted to MapReduce jobs at the end of the day.

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

⋮

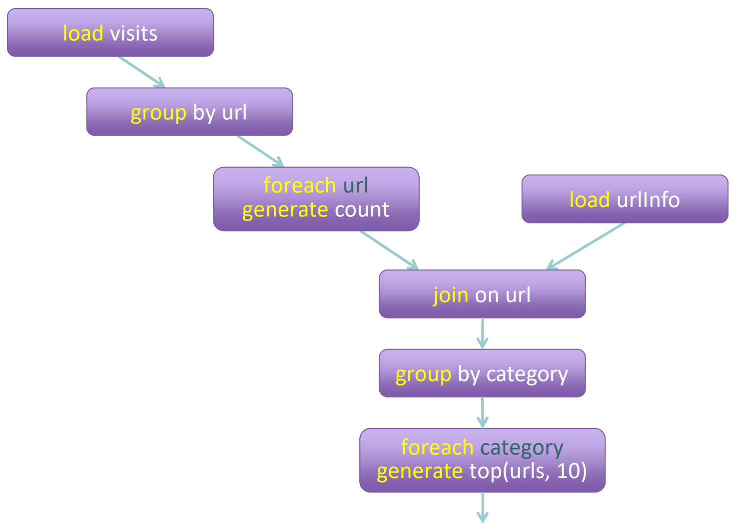
Pig Slides adapted from Olston et al. (SIGMOD 2008)

Pig: Example Script

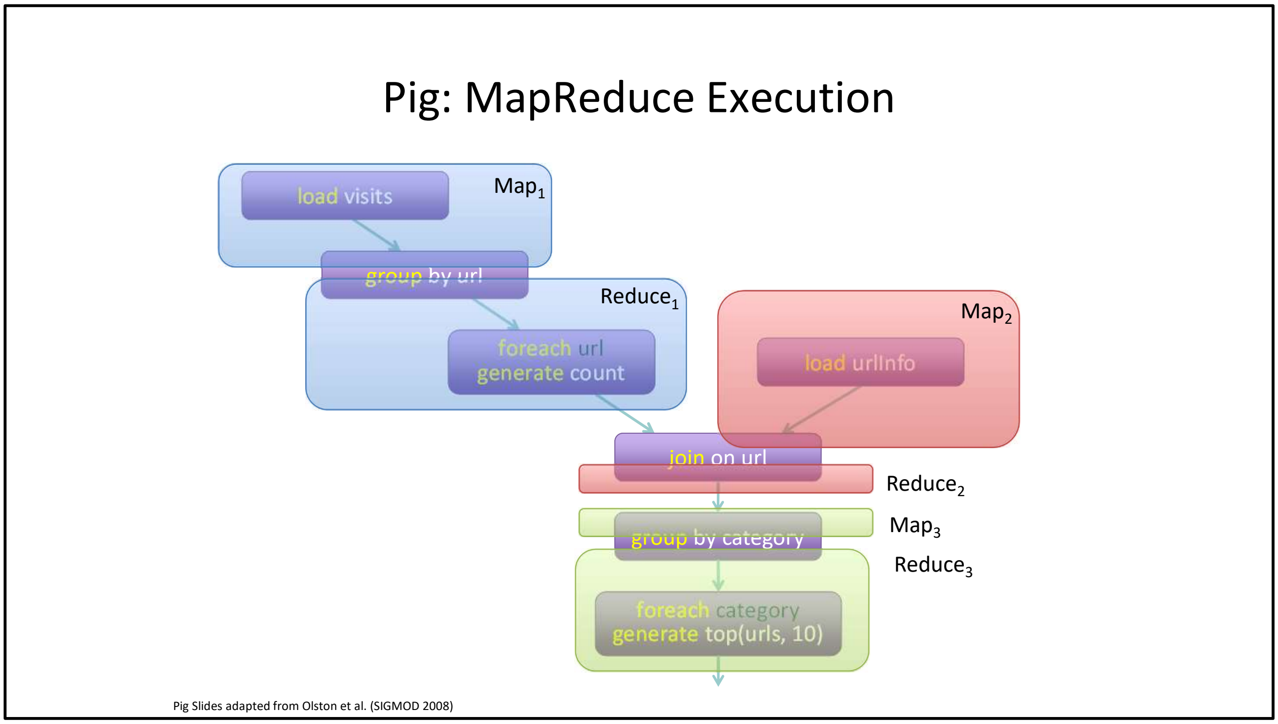
```
visits = load '/data/visits' as (user, url, time);
gVisits = group visits by url;
visitCounts = foreach gVisits generate url, count(visits);
urlInfo = load '/data/urlInfo' as (url, category, pRank);
visitCounts = join visitCounts by url, urlInfo by url;
gCategories = group visitCounts by category;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



YUP, you can do a map that takes multiple inputs. Neato!

```
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 (!)

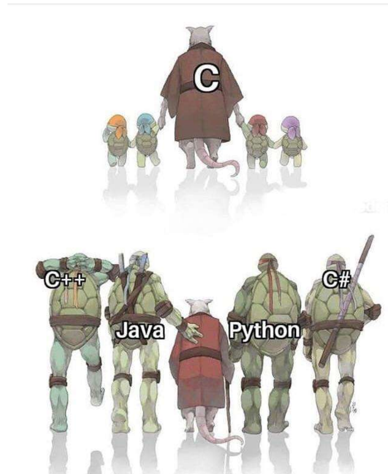
Pig Slides adapted from Olston et al. (SIGMOD 2008)

Isn’t Pig Slower than Hadoop?

Potentially.

Isn’t C slower than assembly?

Isn’t Python slower than C?



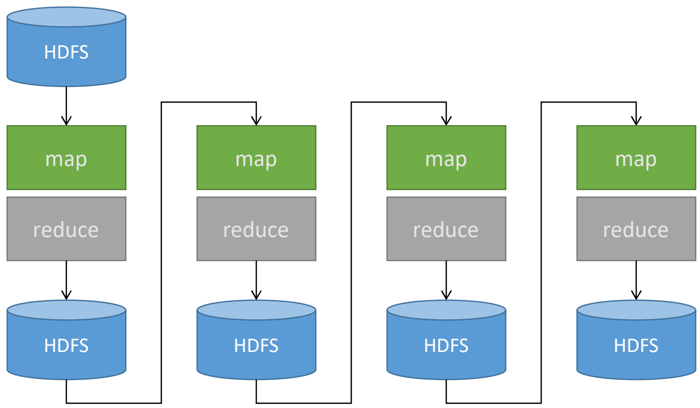
The Data Center as a Computer

So Hadoop is the Instruction Set, right?

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

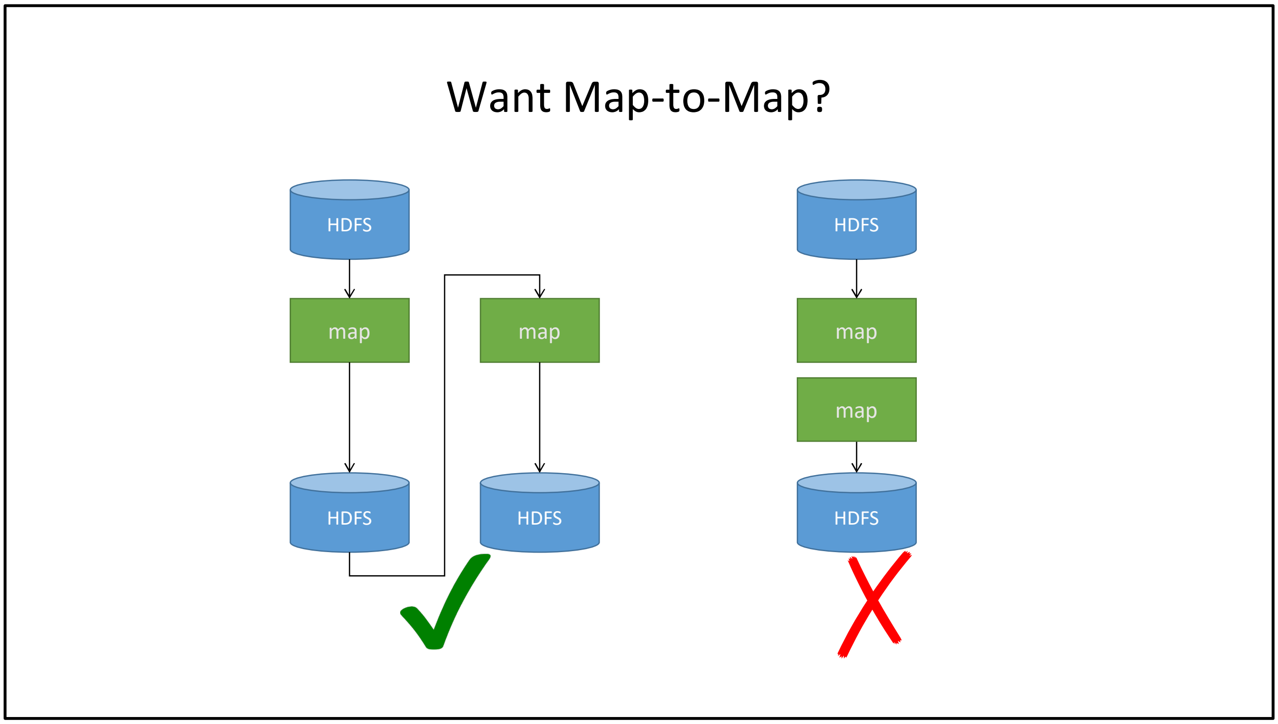
(On A1 yes, you do)

MapReduce Workflows



What's wrong?

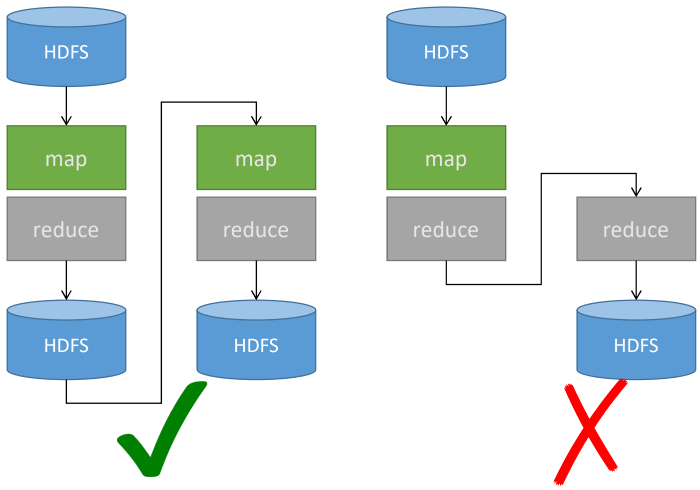
There is a lot of disk i/o involved which significantly reduces running MapReduce jobs like this.



It’s okay not to have reduce but the output of map cannot go to another map.

Why would you want to do this? Well, what if you have a two easily expressed functions – the combination might be a bit complicated. It’s admittedly not a very strong need.

Want Map-to-Reduce-to-Reduce?



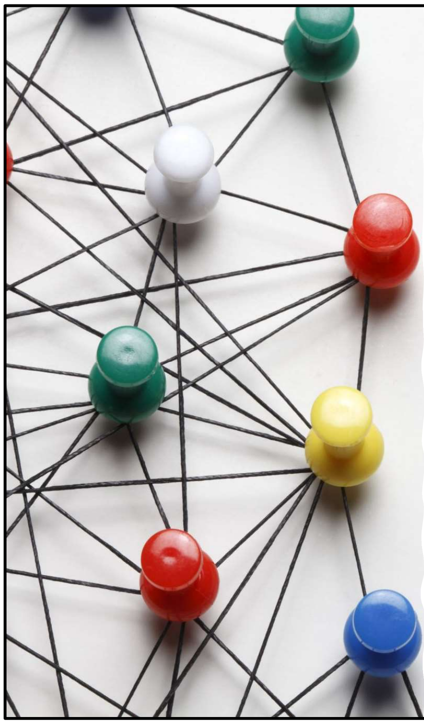
(Strictly speaking, there's FILE access between map and reduce tasks, too, just not HDFS, but we're keeping the slides simple)

Note: You can leave the second job's map function off and it will assume you want an identity function map. The job goes straight to a shuffle. I'm on the fence on if I should remove the second map, or just tell everyone that this will be a very trivial map, but still a map.

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

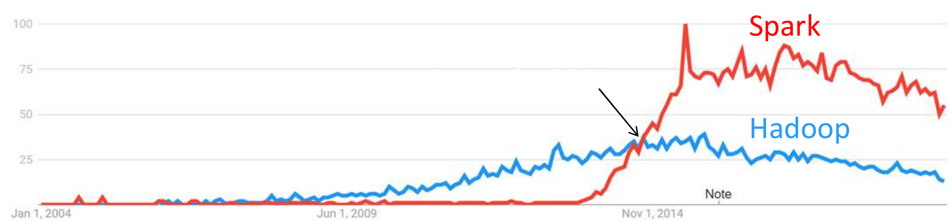
Brief history:

Developed at UC Berkeley AMPLab in 2009

Open-sourced in 2010

Became top-level Apache project in February 2014

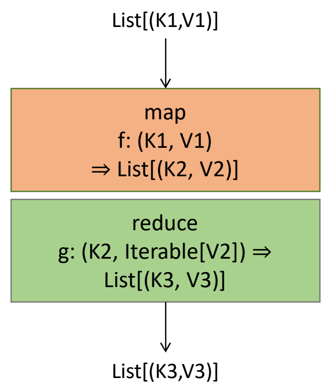
Spark vs. Hadoop



Spark is more popular than Hadoop today.


APACHE SPARK
SO HOT RIGHT NOW
memegenerator.net

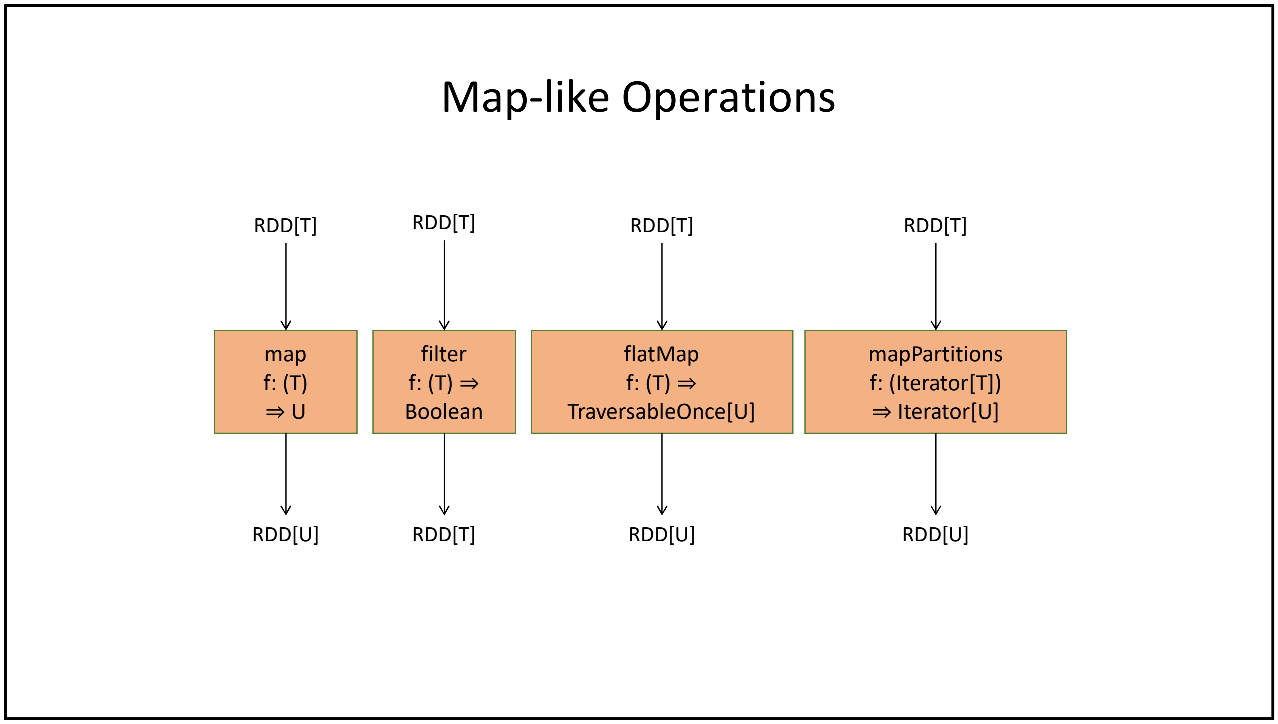
MapReduce



This is the only mechanism we had in MapReduce.

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.



But Spark provides many more operations (enriched instruction set).

Everyone always asks about the differences, so here you are!

Consider an RDD[T] with 4 partitions on 4 workers. The above operations return an RDD[U] with 4 partitions
Let's call the RDD[T] as RDD_{in} and the returned RDD[U] as RDD_{out}

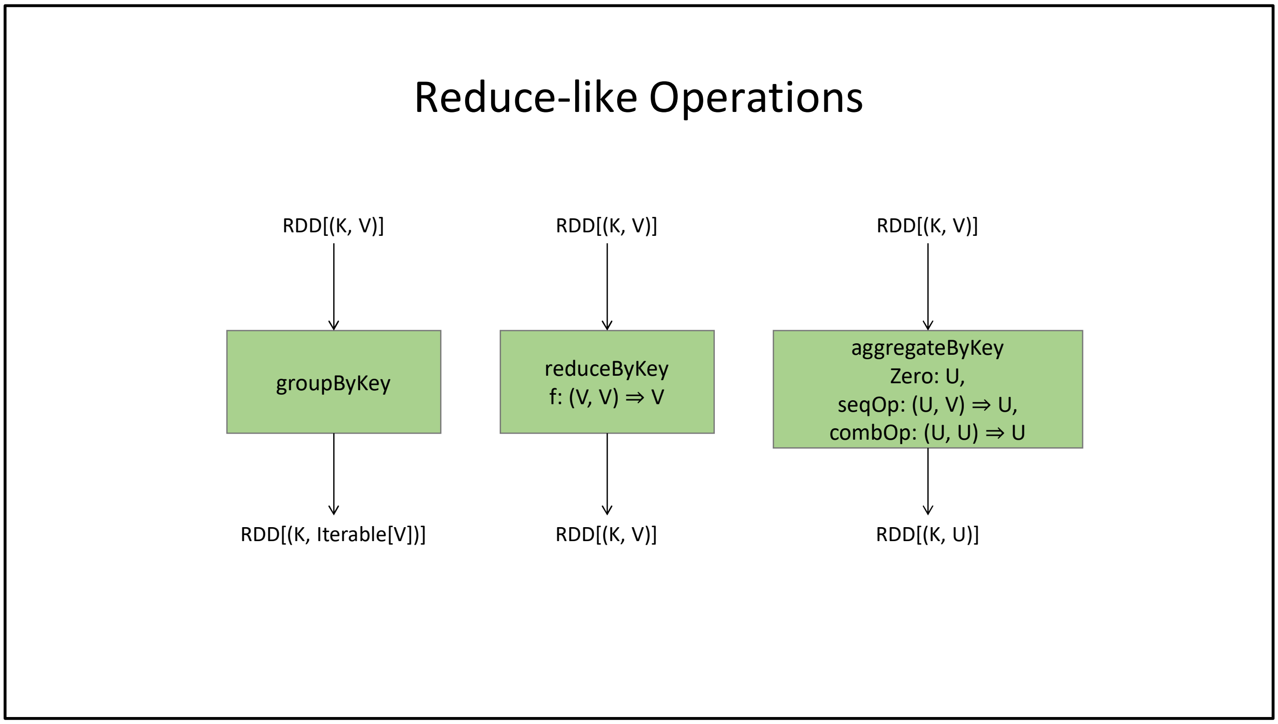
map(f) – f is given one value of type T, and returns one value of type U
Each worker will call f(x) on each item x from RDD_{in}
Each worker will put the value returned by f(x) into a partition of RDD_{out}
flatMap(f) – f is given one value of type T, and returns an iterator/iterable collection that produces values of type U
Each worker will call f(x) on each item x from RDD_{in}
Each worker will then traverse the iterable returned by f(x), and each value gets added to RDD_{out}
(If you think only in terms of lists, then instead of getting an RDD of lists, they are "flattened")
mapPartitions(f) – f is given an iterator that produces value of type T, and returns an iterator/iterable collection that produces values of type U

Each worker will call f(x) ONCE where x is an iterator that traverses all items in that worker's partition of RDD_{in}

Each worker will then traverse the iterable returned by f(x), and each value gets added to RDD_{out} just like flatMap

mapPartitions is handy when you want something like MapReduce's setup and cleanup.
You'd do:

```
def myFunction(values):
  setup things
  for x in values:
     something, probably accumulating values
  cleanup that returns accumulated values
```



Note that these do NOT sort, they use in-memory hash tables for the shuffle, not sorted files. (Spark's design assumes a lot more RAM than MapReduce does)

groupByKey – like MapReduces shuffle. NOT the reduce part, this is JUST the shuffle that brings the pairs to a single place. You'd then use map, flatMap, etc. to perform the reduce action itself.

reduceByKey – like MapReduce's shuffle + combine + reduce.
What does a worker do to perform reduceByKey(f(a,b)) ?

1. Create a Hash table called HT
2. For each (K,V) pair in RDD_{in}
   a. If k is not a key in HT, associate k with v in HT.
   b. Otherwise, retrieve the old value v_{old} from HT, and replace it with $f(v_{old},v)$
3. Perform a shuffle – each reducer-like-worker will receive key-value pairs. It will then repeat step 2 for all received pairs.

aggregateByKey – a more complicated reduceByKey
aggregateByKey(zero, insert, merge)

1. Create a Hash table called HT

2. For each (K,V) pair in RDD_{in}

a. If k is not a key in HT, associate k with insert(zero, v) in HT.

b. Otherwise, retrieve the accumulator u from HT, and replace it with insert(u,v)

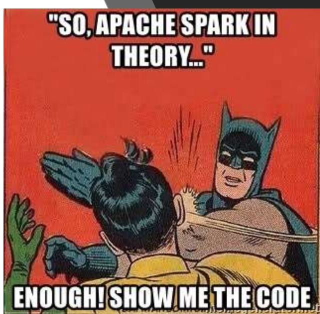
3. Perform a shuffle – each reducer-like-worker will receive key-value pairs. The third parameter, merge, is used to combine accumulators

(There's also combineByKey which is the same, except instead of a zero-value, you give it another function, one that creates an accumulator out of a single value V.
Technically combineByKey is the only "real" function – aggregateByKey(zero, insert, merge) calls combineByKey(lambda v: insert(zero, v), insert, merge), and reduceByKey(f) calls combineByKey(identity,f,f)

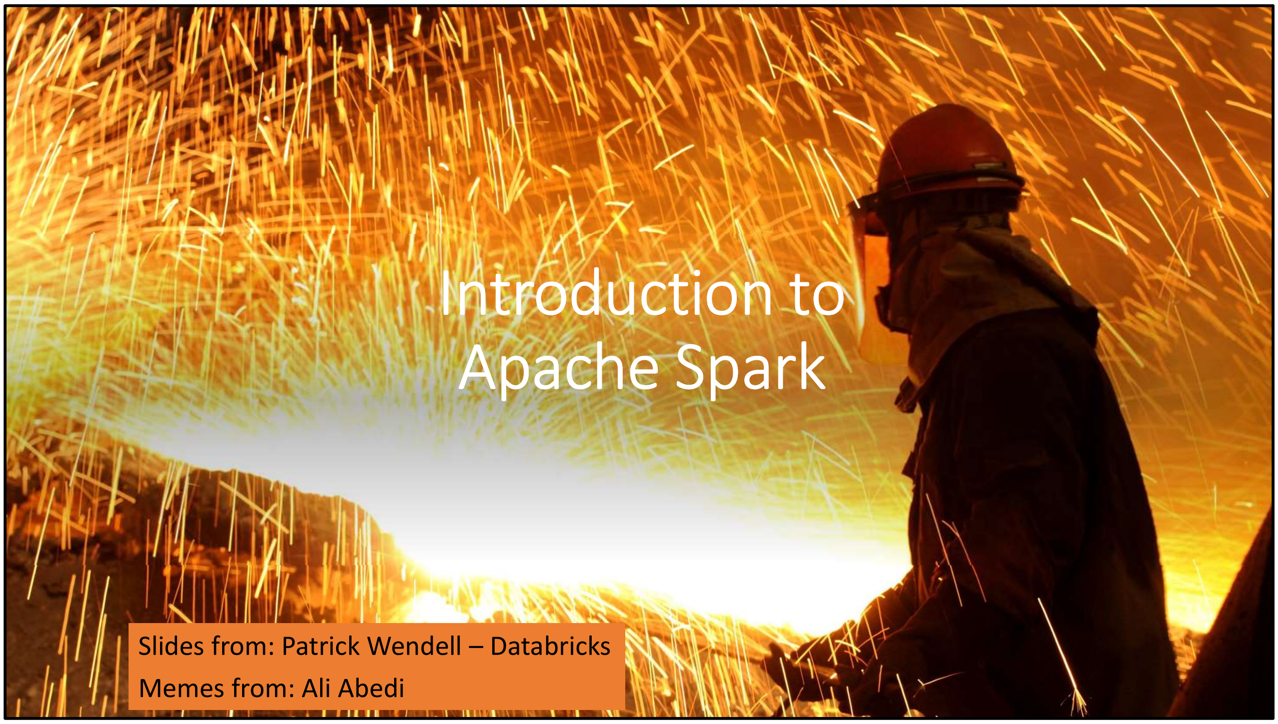
And many other operations!

31

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

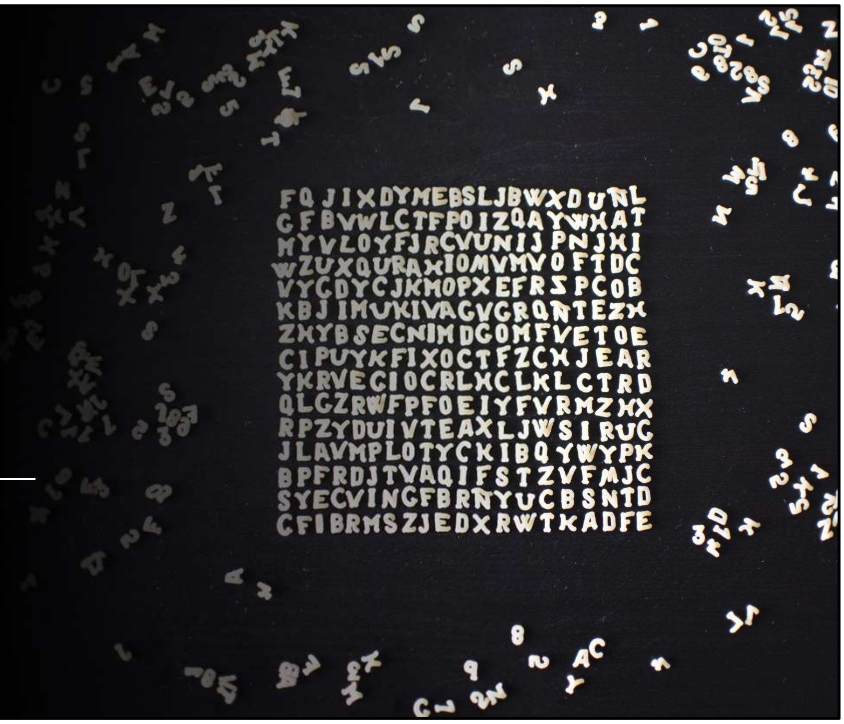
Efficient

- 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**

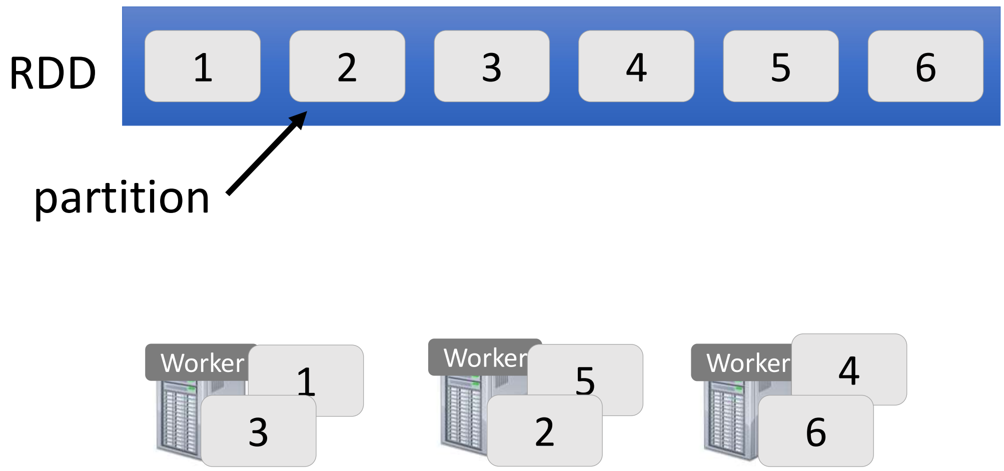
Resilient Distributed Datasets

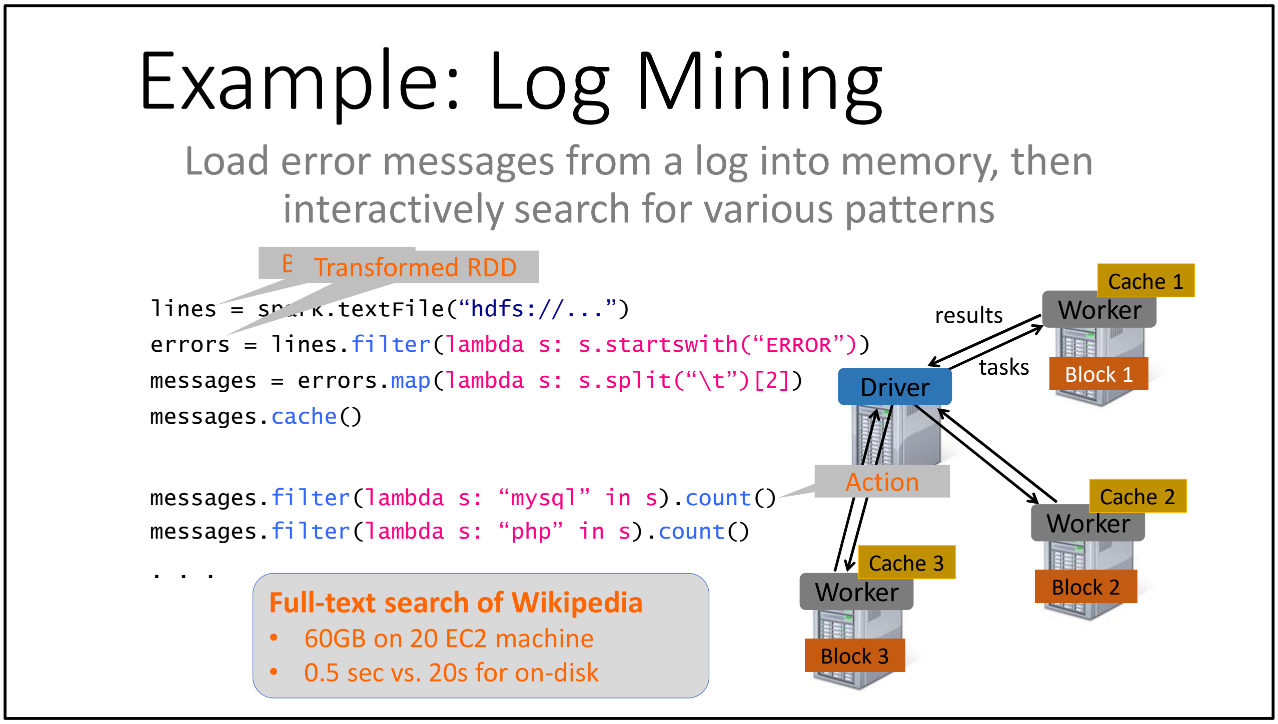
- Collections of objects spread across a cluster, stored in RAM or on Disk
- Built through parallel transformations
- Automatically rebuilt on failure

Operations

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

RDD structure

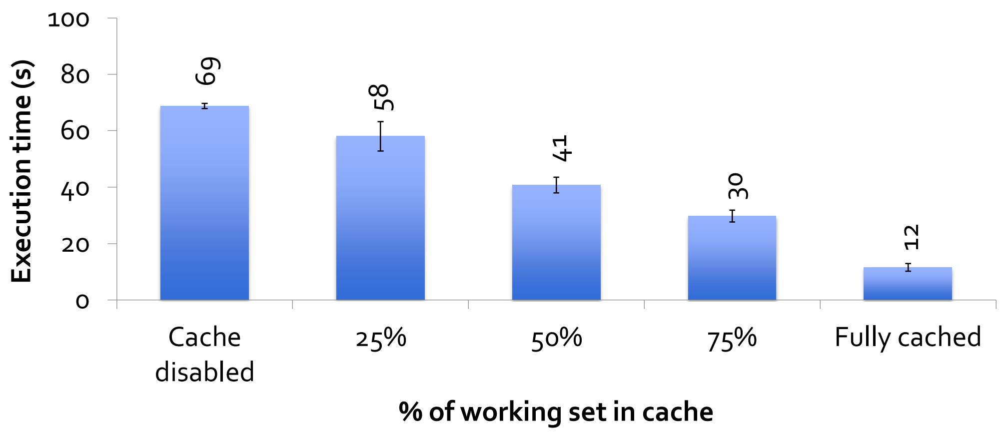




Lazy evaluation: Spark doesn't really do anything until it reaches an action! This helps Spark to optimize the execution and load only the data tat is really needed for evaluation.

Dan adds: If you branch, then you cache!

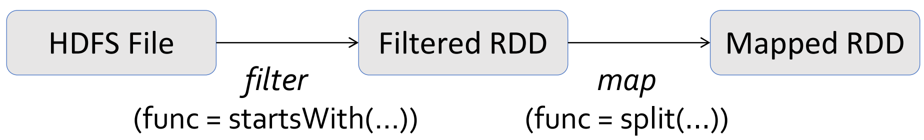
Impact of Caching on Performance



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

Poor font choice I think? Lowercase "sc"

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:

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

Scala:

```scala
val pair = (a, b)
pair._1 // => a
pair._2 // => b
```

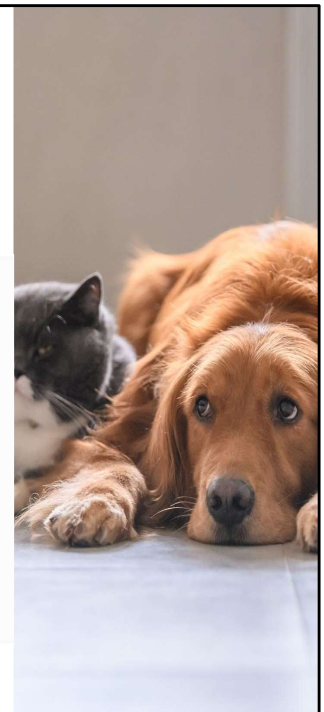
Java:

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

While this seems awful, you rarely actually need to deal with pairs in Scala by using _1 and _2, you can use pattern matching / case lambdas in a way that's not entirely unlike unpacking in Python

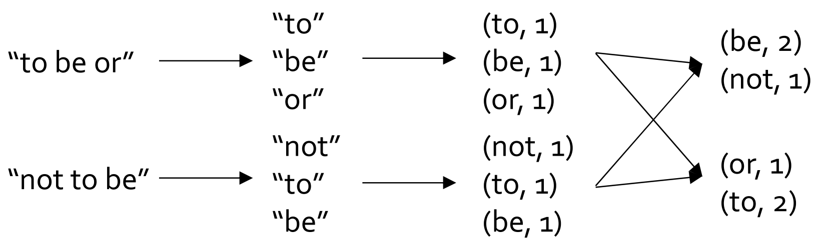
Some Key-Value Operations

```
> 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”

(_+_) looks like a butthole but we’re all going to just ignore that and be mature

I mean, word count, aka token frequency, is a building block for lots of text processing…just because it's easy doesn't mean it's not useful.

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)
```

Dan adds: So does scc.textFile (and other base RDDs). However, for these this is “minimum number of tasks”. E.g. if a file is split into 16 blocks on HDFS, and you open it with textFile(PathString, 10), you’ll still get 16 partitions, not 10.
If you’re submitting a job with a total of 8 vCores, you should always have 8 partitions if you can manage it. Otherwise a core will be idle. (In fact, it’s usually better to have more tasks than cores, so that tasks bottle necked on reading will be able to share a single core).

What’s the default?
It uses the same number of partitions for destination as the source has. Eg a reduceByKey on an RDD with 8 partitions will result in another RDD with 8 partitions.
For joins, it’s the minimum of the LHS and RHS RDDs. Eg join an RD with 3 parts to one with 8, you will get 3.

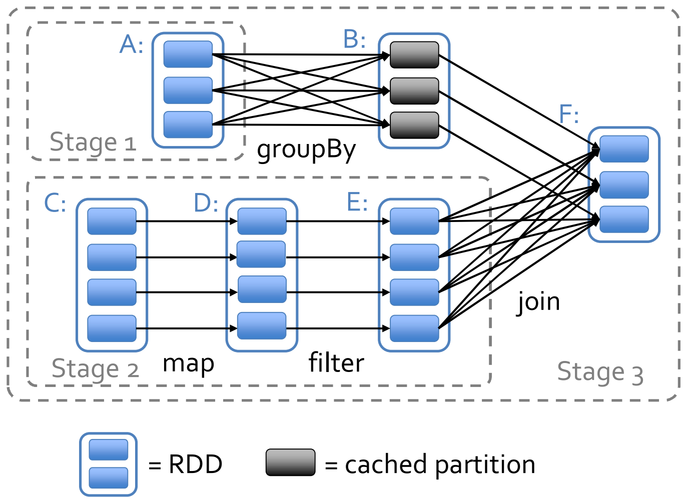
If you specify spark.default.parallelism it will use this as the default instead! (For shuffles only, not parallelize textFile or other base RDDs)

Please watch your jobs on datasci and kill things that seem to be stuck. Try on student.cs FIRST, only run on datasci when you're confident. If you need to make changes, rerun on student.cs first!!!

I've added a bonus slide at the end with some tips about viewing Spark jobs on the cluster. (A big reason for "runs forever" on datasci is a reducer that's O(n) – usually caused by stripes being merged inefficiently.)

Under The Hood: DAG Scheduler

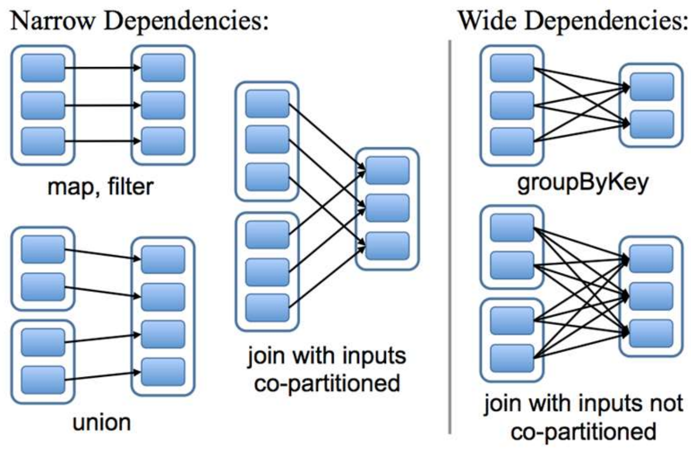
- General task graphs
- Automatically pipelines functions
- Data locality aware
- Partitioning aware to avoid shuffles



Directed Acyclic Graph (DAG)
A job is broken down to multiple stages that form a DAG.

You can get the DAG from an RDD using the toDebugString method. (print it, since it contains newlines and will be illegible as a string value)
It's also viewable through the Hadoop monitoring page.

Physical Operators



Narrow dependency is much faster than wide dependency because it does not require shuffling data between working nodes.

Also: reduceByKey, groupByKey, etc will also have narrow dependencies if the upstream RDD is already partitioned by key. Its less common but not unheard of.

More RDD Operators

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

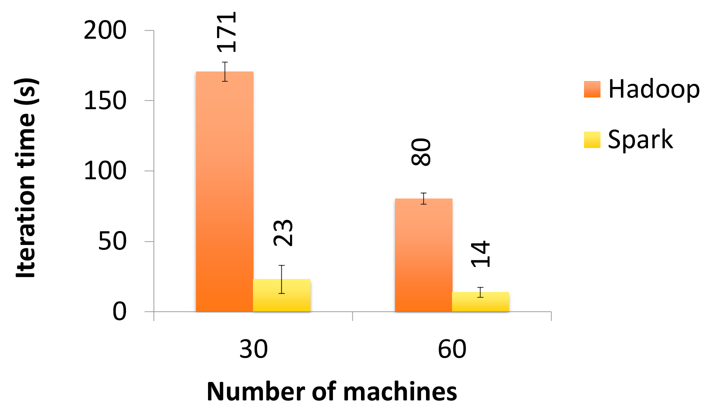
- `reduce`
- `count`
- `fold`
- `reduceByKey`
- `groupByKey`
- `cogroup`
- `cross`
- `zip`

`sample`
`take`
`first`
`partitionBy`
`mapWith`
`pipe`
`save` ...


Learn Apache Spark. You Must.

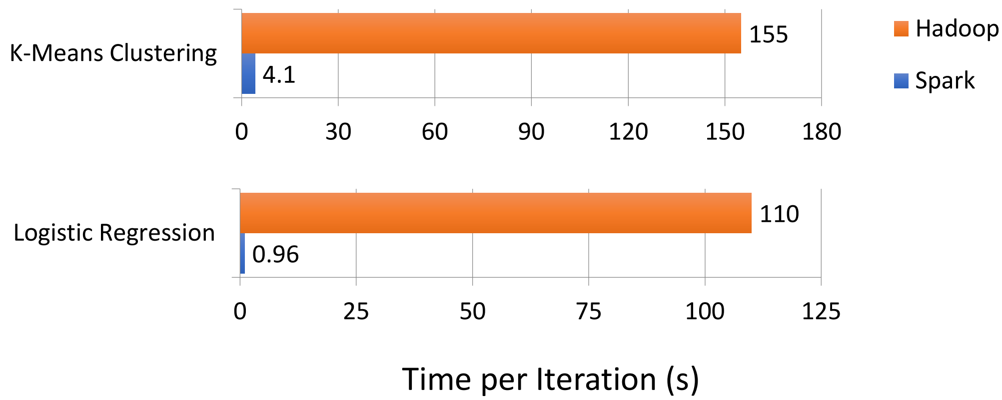

Performance
2,5
2,47

PageRank Performance



Since spark avoids heavy disk i/o, it significantly improves the performance.

Other Iterative Algorithms



Spark outperforms Hadoop in iterative programs because it tries to keep the data that will be used again in the next iteration in memory. In contrast with Hadoop which always read and write from/to disk.


Hadoop Ecosystem and Spark

YARN

Hadoop's (original) limitations:

Can only run MapReduce

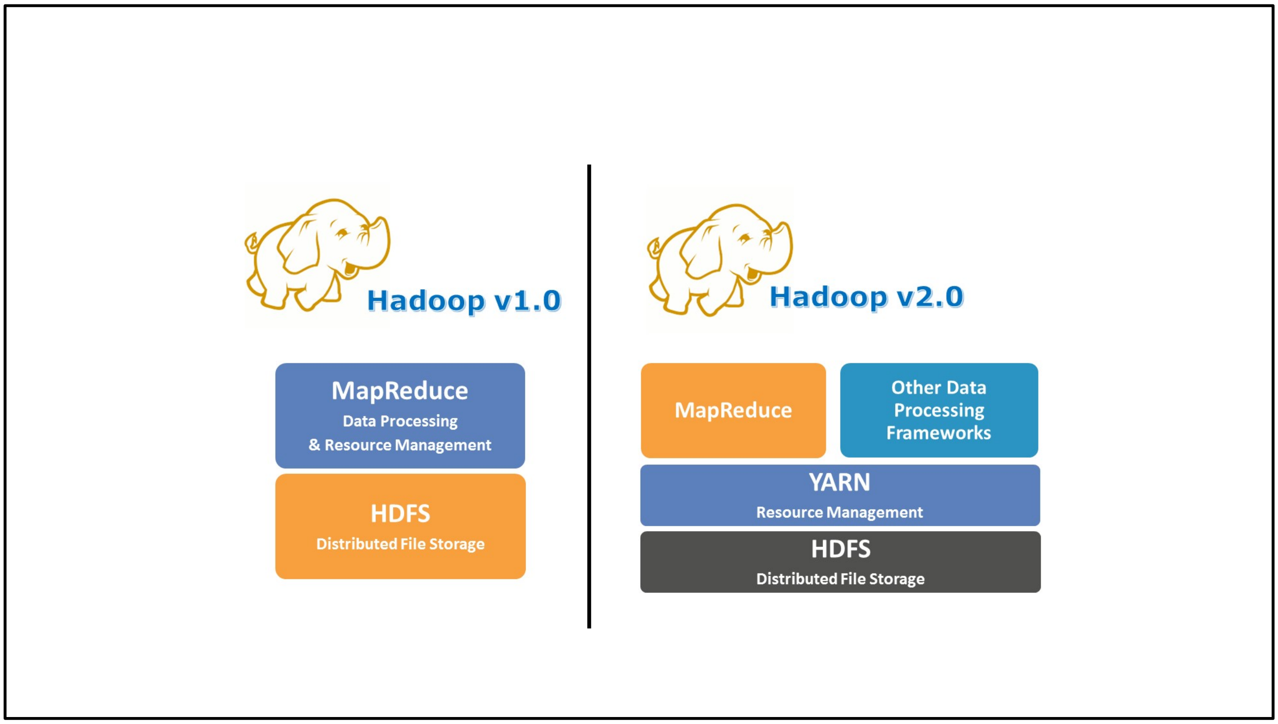
What if we want to run other distributed frameworks?

YARN = Yet-Another-Resource-Negotiator

Provides API to develop any generic distributed application

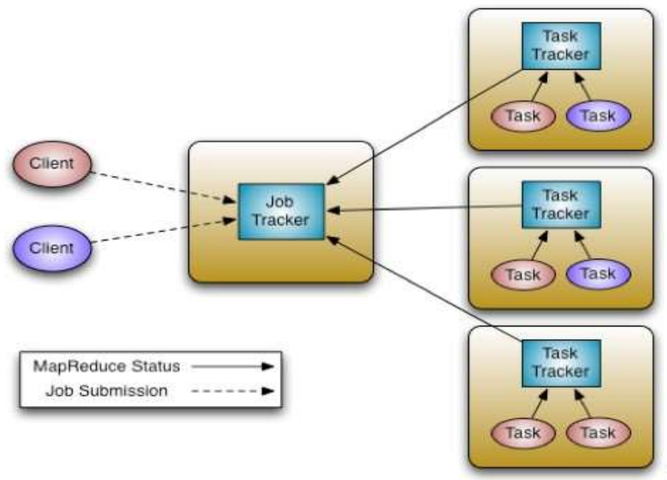
Handles scheduling and resource request

MapReduce (MR2) is one such application in YARN



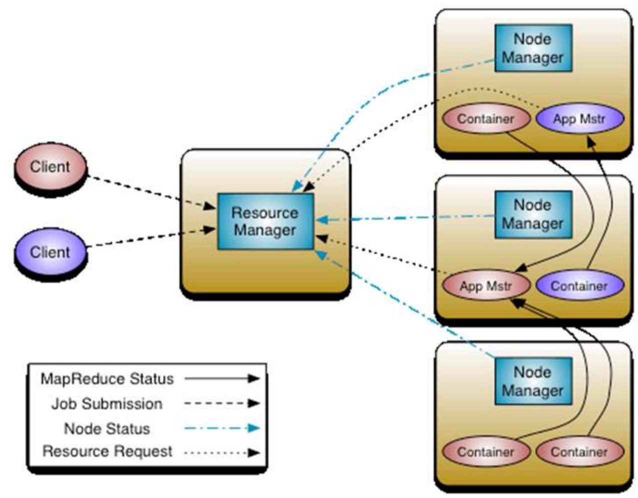
In Hadoop v1.0, the architecture was designed to support Hadoop MapReduce only. But later we realised that it is a good idea if other frameworks can also run on Hadoop cluster (rather than building a separate cluster for each framework). So in v2.0, YARN provides a general resource management system that can support different platforms on the same physical cluster.

Hadoop v1.0



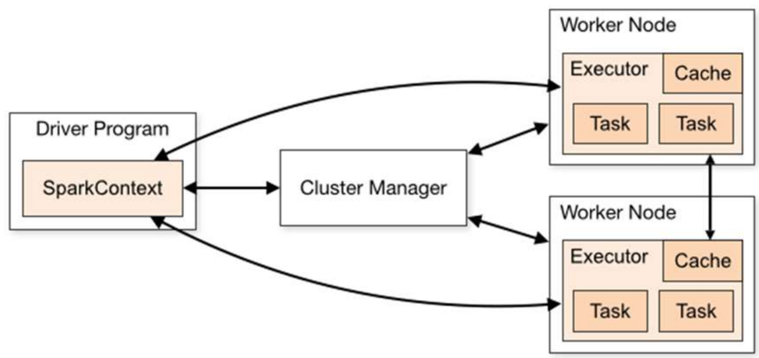
The Job tracker in v1.0 was specific to Hadoop jobs.

Hadoop v2.0



But the resource manager in v2.0 can support different types of jobs (e.g., Hadoop, Spark,…).

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)
```

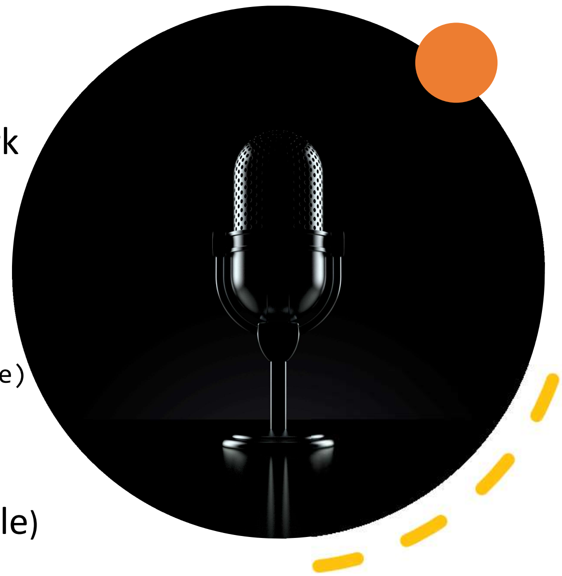
The lambda "captures" thresh, so it gets packaged up too. (That's bad if it's large)

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)

The errors are what you'd see in Scala or Python

Note that of course it's technically possible to make a broadcast variable where the value is a mutable type (easier in Python where that's most collection types, but still doable in Scala)
This will "work" in that it won't give you the above errors.
But it won't "work" in that each worker has its own copy of this value, so if one of them updates a dictionary, the other workers don't see that.

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

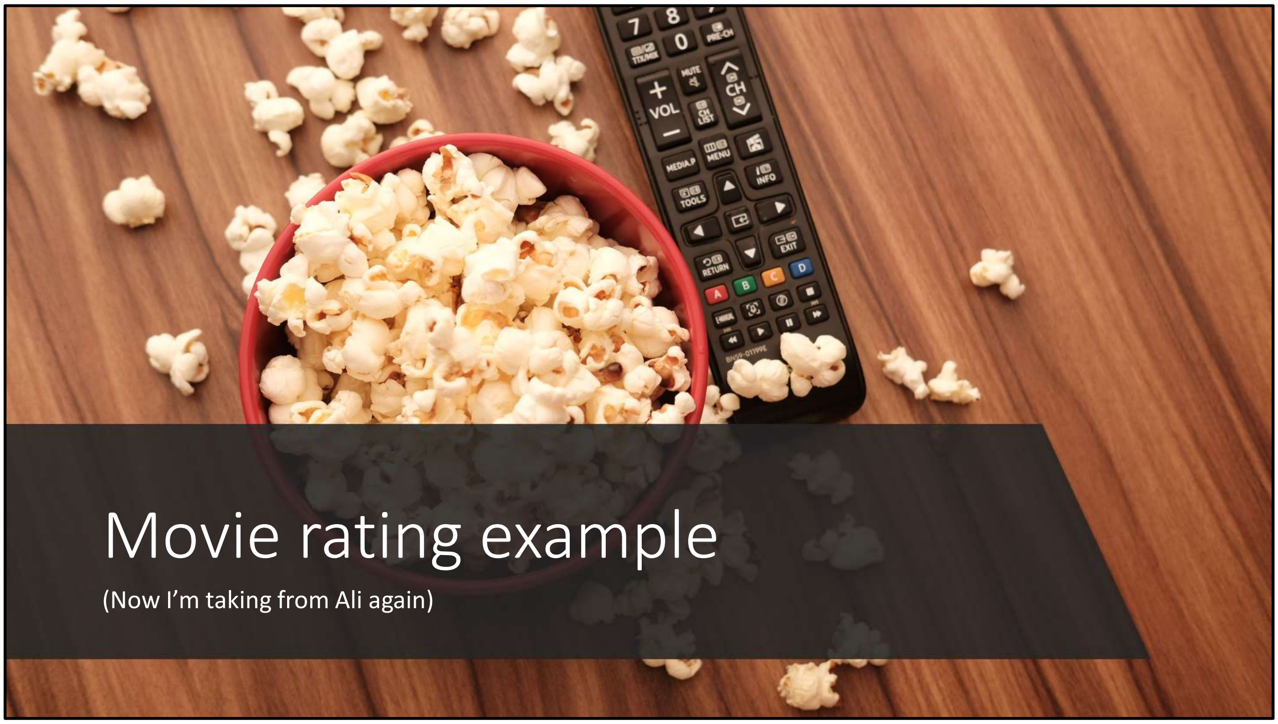
(In Python, they're just called accumulator)

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)

Also like MapReduce, can override.

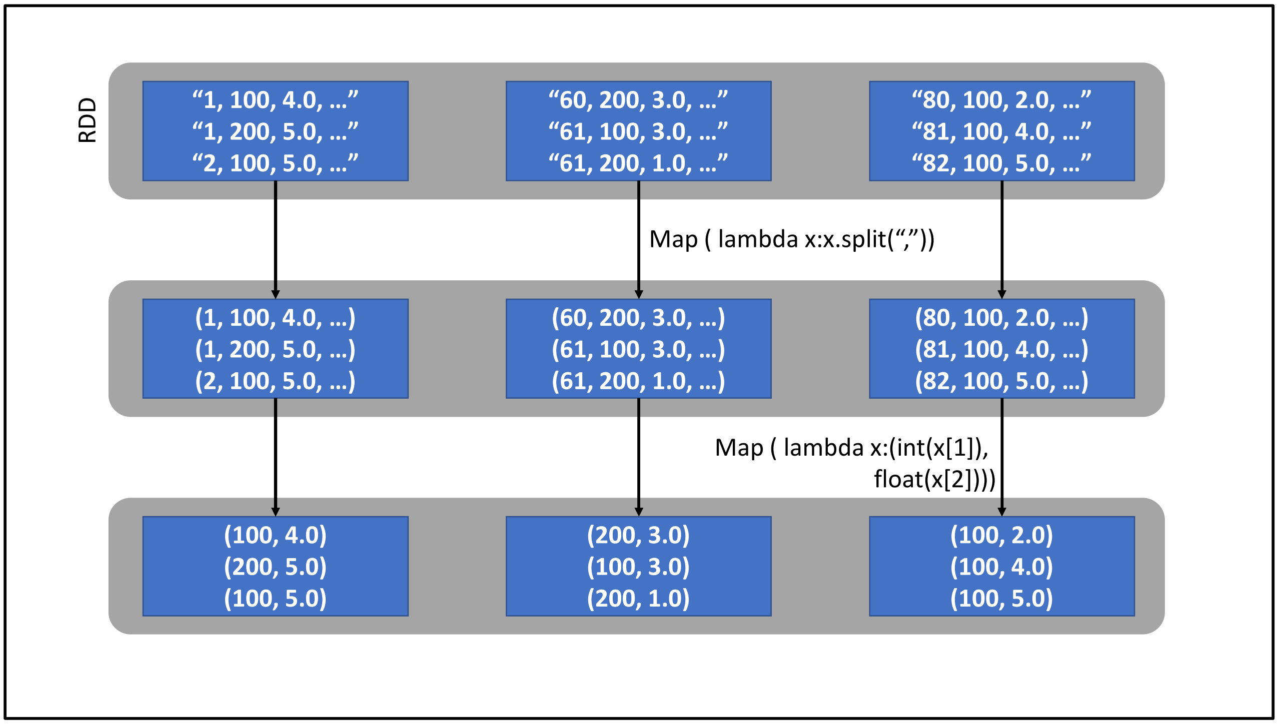


This example in particular is not very helpful in slide-only form. I alt-tab and do some goofing around in spark-shell or pyspark

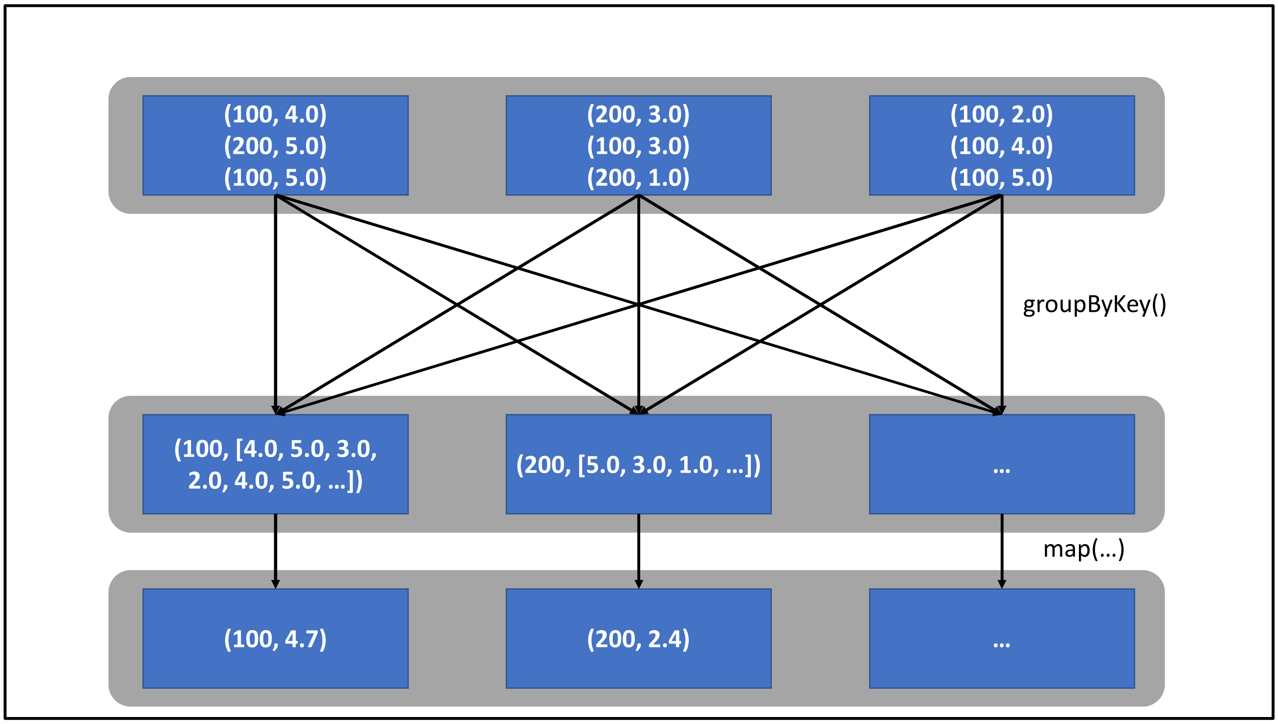
Input Format

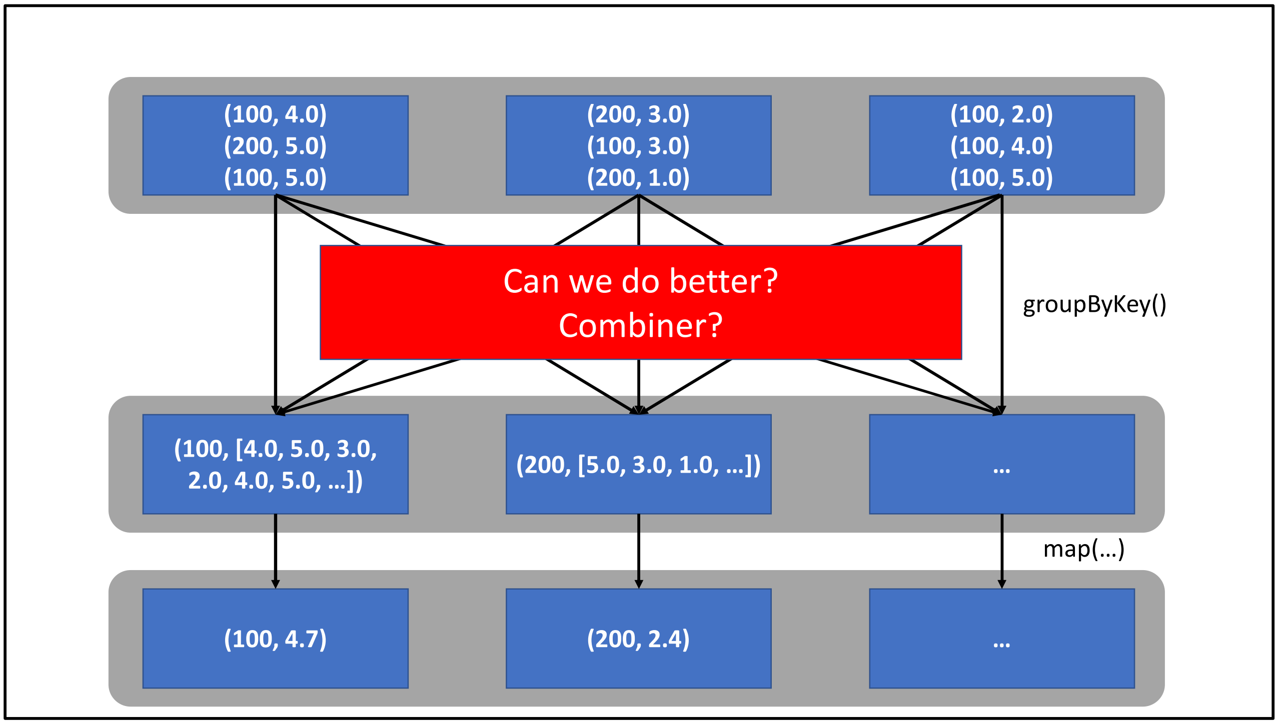
CSV file
Fields:
- User ID (unique key per user)
- Movie ID (unique key per movie)
- Rating (1-5 stars)
- Text of review (optional)

e.g.
“1, 100, 3.5, ‘s aight”



The “” are missing from the tuples in the middle because I’m not going to type that many “”””””! It’d be really hard to read I think


(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)
...



Avoid groupByKey if you can – MapReduce (without combiner) in Spark is essentially – flatmap -> groupByKey -> flatmap. We're trying to do better than MapReduce though.

Wait…converting MapReduce to Spark doesn't use the reduceByKey function??? That's right. MapReduce's reduce is more flexible.

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)] ⇒ 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

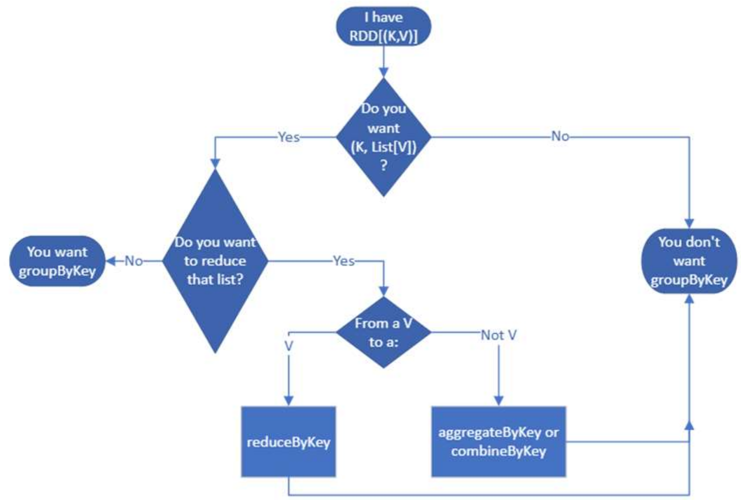
aggregateByKey is between reduceByKey and combineByKey

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

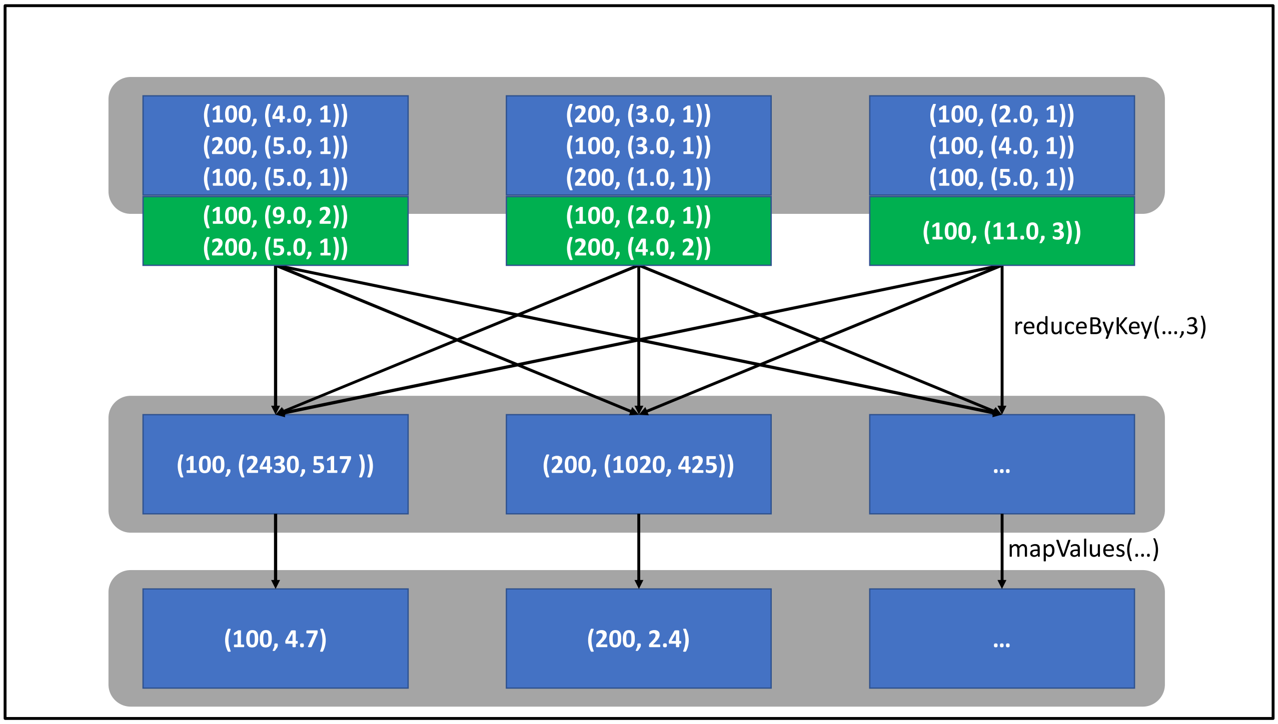
- init – initial (or zero) value [type C]
- append- take a C and add a V to it
- merge- combine two C

groupByKey

You (probably) don't want to use it

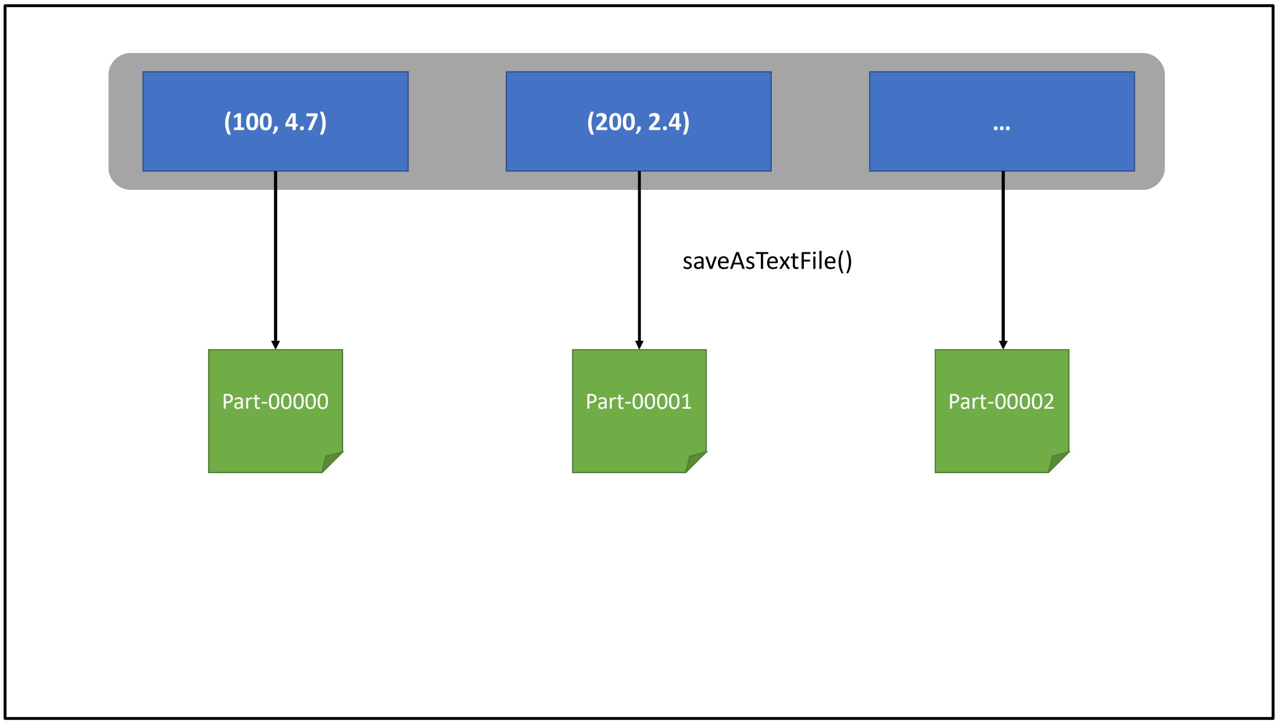


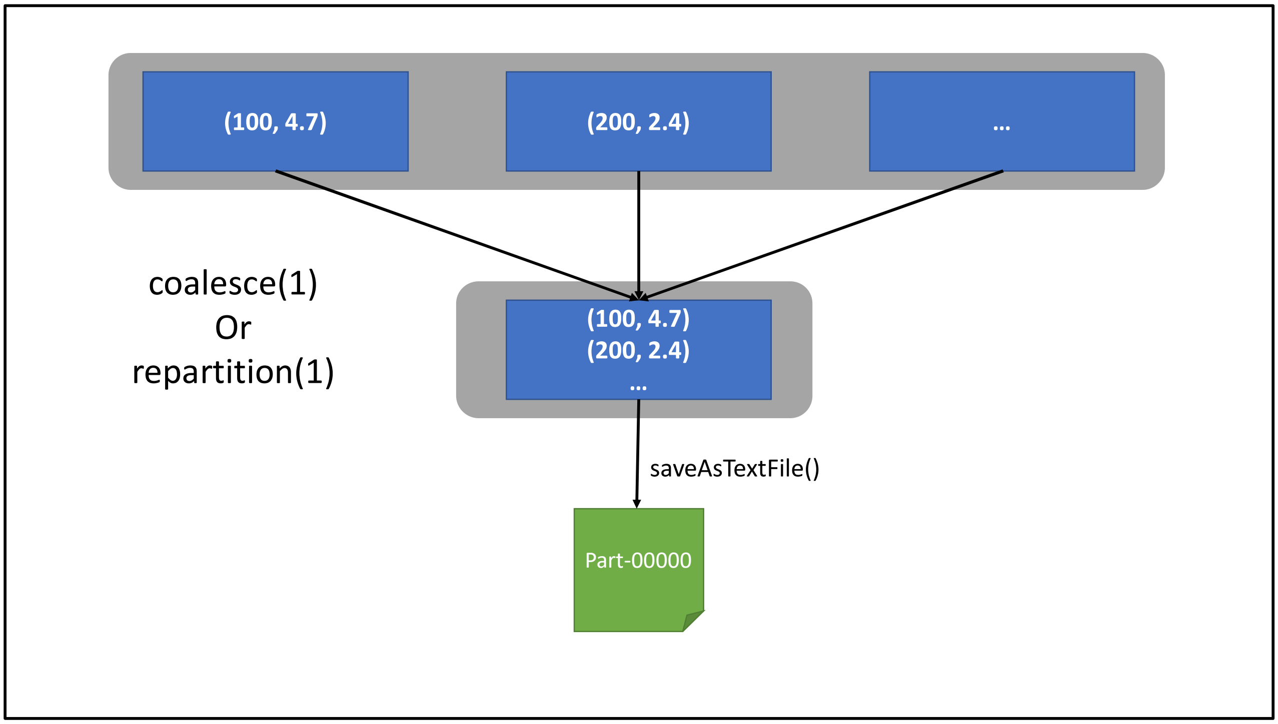
This is incomplete! If your reduce action needs to know what the key is (meaning, if some keys need to be treated differently) then groupByKey -> map or mapPartitions might be what you want.



Spark's reduceByKey is NOT like the Reduce phase of MapReduce!

reduceByKey – partitions the RDD, then reduces each partition, THEN shuffles for a final reduce.
The second parameter here is optional (the default number of partitions is a Spark configuration option)


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



Repartition triggers shuffling but it gives more balanced partitions. It can be used to increase or decrease the number of partitions.
Coalesce can be used to only reduce the number of partitions. It avoids full shuffling so it is faster than repartition but it may give unbalanced partitions.

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")
```

Behold the power of pattern matching anonymous functions! Pattern matching is one of several reasons to love Scala
I could have written (p1, p2) => (p1._1 + p2._1, p1._2 + p2._2)) but that's ugly!

Also…pro tip for live coding in front of an audience. Names like "count" and "cnt" are easy to typo. There was some scandalized gasps one lecture, let me tell you…

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 …
```

Read bottom-to-top

1. Text file loaded and partitioned (like MapReduce in Hadoop, this will try to allocate the jobs to workers that already have that chunk of HDFS data)
2. Map is applied to existing partitions (split the lines)
3. Map is applied to existing partitions (extract useful fields, convert to appropriate types, convert rating to (rating, 1) for averages
4. reduceByKey triggers a repartition based on the keys (movie IDs)
5. Map is applied to the new partitions (convert (sum , count) to sum / count)
6. Coalesce merges data into 1 partition

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