Resilient Distributed Datasets (RDDs)

Resilient Distributed Dataset (RDD)

- represents an immutable, partitioned collection of records that can be operated on in parallel
  - list of objects
- the records are just Java, Scala, or Python objects of the programmer's choosing.
  - Unlike DataFrames though, where each record is a structured row containing fields with a known schema
- RDDs provide transformations, which evaluate lazily, and actions, which evaluate eagerly, to manipulate data in distributed fashion.

Creating an RDD

- From local collection

```
myCollection = "Spark The Definitive Guide : Big Data Processing Made Simple"\
 .split(" ")
words = spark.sparkContext.parallelize(myCollection, 2)
```

- From data source

```
spark.sparkContext.wholeTextFiles("/some/path/withTextFiles")
```

Some examples

```
# Map (apply a function to a list)
rdd = sc.parallelize(range(10), 5)
rdd.map(lambda x: x*2).collect()
```

▸ (1) Spark Jobs

```
Out[7]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

```
words = "I like data analytics with PySpark".split(" ")  # List of words
wordsRDD = sc.parallelize(words)
wordsRDD.take(15)
```

▸ (3) Spark Jobs

```
Out[10]: ['I', 'like', 'data', 'analytics', 'with', 'PySpark']
```

```
# Add a suffix to each word
wordsRDD.map(lambda x: x + " um").take(15)
```

▸ (3) Spark Jobs

```
Out[11]: ['I um', 'like um', 'data um', 'analytics um', 'with um', 'PySpark um']
```

Examples using reduce

```
xRDD = sc.parallelize(range(0,10))
xRDD.collect()
```

▸ (1) Spark Jobs

```
Out[15]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
# Sum the list
# In the list, replace every pair of numbers with the sum of their numbers and repeat until the list has only one number.
xRDD.reduce(lambda x1,x2: x1+x2)
```

▸ (1) Spark Jobs

```
Out[16]: 45
```

Notice that RDD is the API for partitions

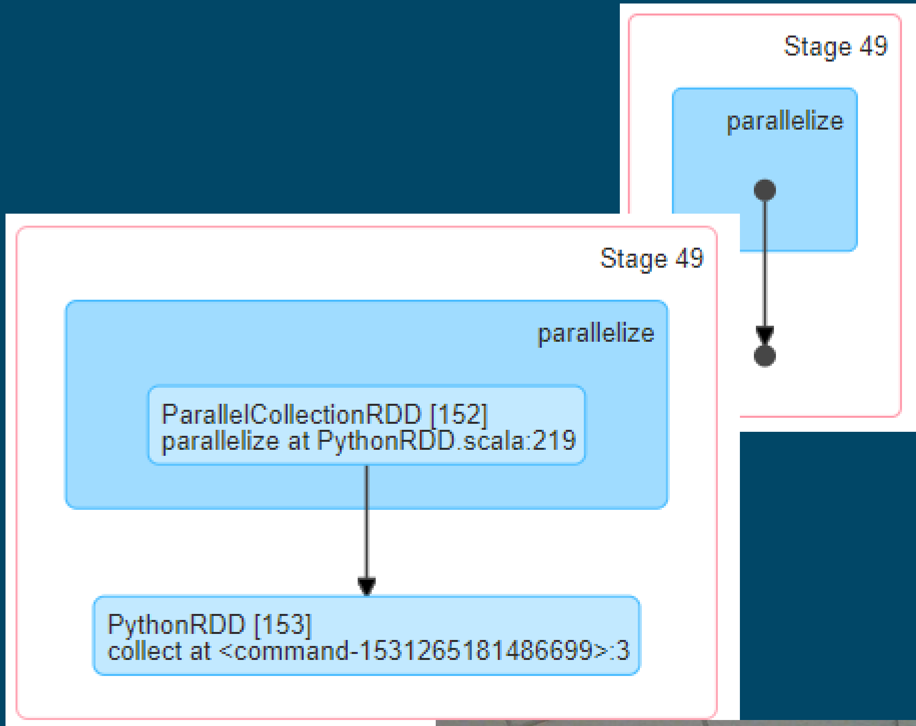
2 partitions -> 2 tasks

```
data = range(1,100)
rdd = sc.parallelize(data,2)
rdd.map(lambda x: x * x).collect()

```

(1) Spark Jobs

Job 36 View (Stages: 1/1)



Tasks (2)

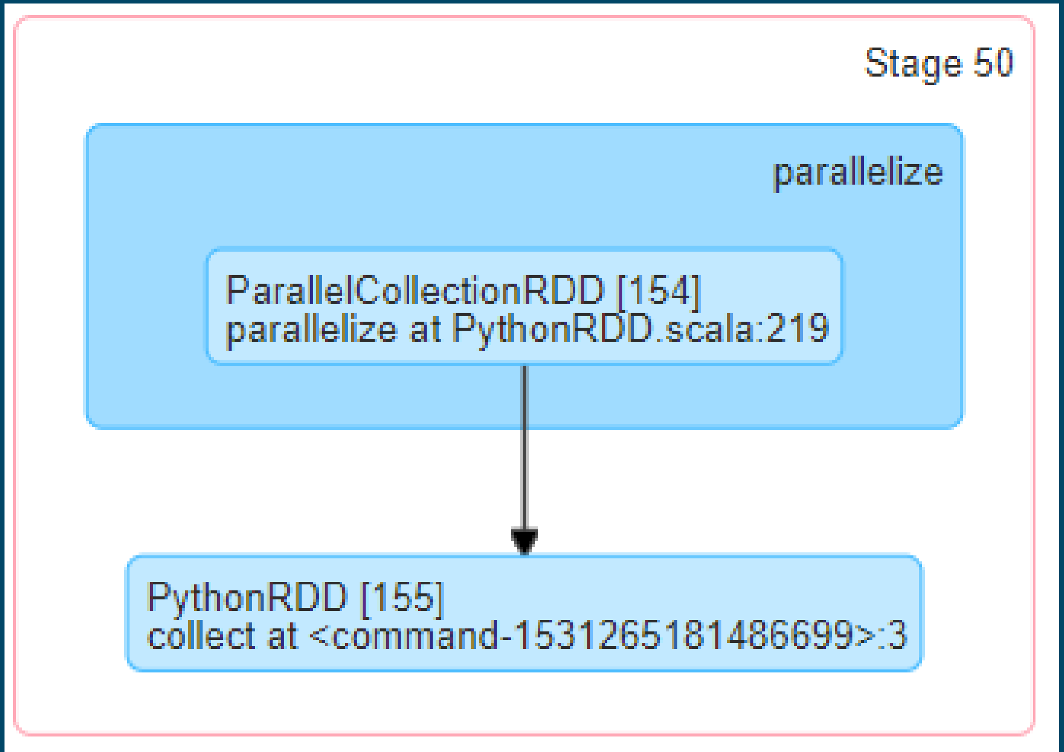
Index	ID	Attempt	Status	Locality Level	Executor ID	Host	Laun
0	206	0	SUCCESS	PROCESS_LOCAL	driver	localhost	2018
1	207	0	SUCCESS	PROCESS_LOCAL	driver	localhost	2018

5 partitions -> 5 tasks

```
data = range(1,100)
rdd = sc.parallelize(data,5)
rdd.map(lambda x: x * x).collect()
```

▾ (1) Spark Jobs

▸ Job 37 View (Stages: 1/1)



Tasks (5)

Index ▲	ID	Attempt	Status	Locality Level	Executor ID	Host	Launch Time	Duration
0	208	0	SUCCESS	PROCESS_LOCAL	driver	localhost	2018/11/02 16:58:40	0.2 s
1	209	0	SUCCESS	PROCESS_LOCAL	driver	localhost	2018/11/02 16:58:40	0.2 s
2	210	0	SUCCESS	PROCESS_LOCAL	driver	localhost	2018/11/02 16:58:40	0.3 s
3	211	0	SUCCESS	PROCESS_LOCAL	driver	localhost	2018/11/02 16:58:40	0.2 s
4	212	0	SUCCESS	PROCESS_LOCAL	driver	localhost	2018/11/02 16:58:40	0.2 s

RDD API functions

Some transformations

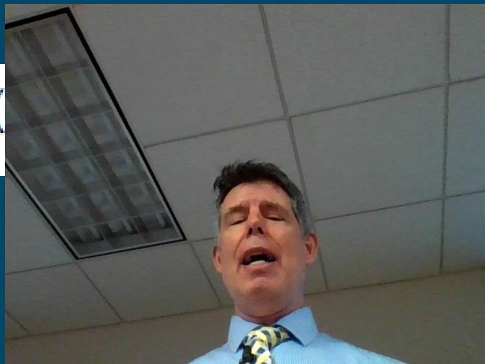
- Filter

```
def startsWithS(individual):
 return individual.startswith("S")
```

```
words.filter(lambda word: startsWithS(word)).collect()
```

- Map

```
words2 = words.map(lambda word: (word, word[0], word.startswith(
```



Some actions

- Count

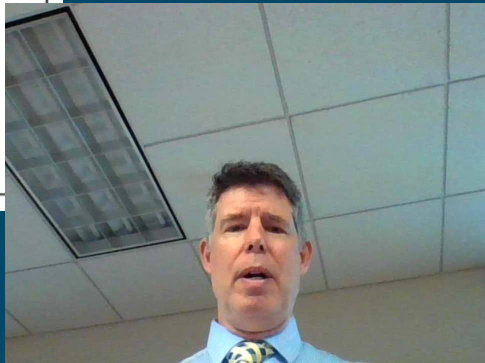
```
words.count()
```

```
words.cache()
```

- Cache
  - Subsequent access to words relies on temporary stored data rather than rerunning the computation

RDD Transformations and Actions

Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(*f* : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(*c* : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(*f* : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(*k* : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS



Key value pairs

- Many RDD are processed as key-value pairs
- Returns pair (word,1)
  - Word as key, with 1 as value

```
words.map(lambda word: (word.lower(), 1))
```

  - Can be used to count up word occurrences

- Map over pairs

```
keyword.mapValues(lambda word: word.upper()).collect()
```

```
[('s', 'SPARK'),
 ('t', 'THE'),
 ('d', 'DEFINITIVE'),
 ('g', 'GUIDE'),
 (':', ':'),
 ('b', 'BIG'),
 ('d', 'DATA'),
 ('p', 'PROCESSING'),
```

Count letter occurrence from words

- Collection of pairs (letter, 1)

```
chars = words.flatMap(lambda word: word.lower())
KVcharacters = chars.map(lambda letter: (letter, 1))
```

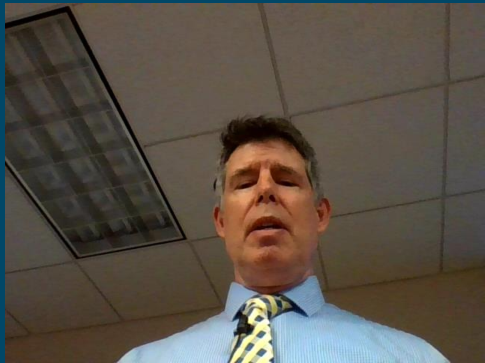
- groupByKey
  - First, sort letters into groups by their key (letter)
  - Then, using reduce, add up the occurrences

```
KVcharacters.groupByKey().map(lambda row: (row[0], reduce(addFunc, row[1])))\
 .collect()
```

- reduceByKey
  - Add occurrences that have the same key

```
KVcharacters.reduceByKey(addFunc).collect()
```

Class word-count program



Classic word-count with map-reduce

- Originally used to index web pages (by Google, Yahoo!)
  - Create a map of (word,1) pairs
  - Add up all numbers with for the same word

```
words = "I like data analytics with PySpark. It's data analytics are like Python.".split(" ")  # List of words
wordsRDD = sc.parallelize(words)
words_KV_RDD = wordsRDD.map(lambda w: (w.lower(), 1))

print("Each word (key) with an associated 1: {}".format(words_KV_RDD.collect()))

word_count_RDD = words_KV_RDD.reduceByKey(lambda left,right: left+right)
print("Now, reduce those pairs by summing the 1's when the keys match:\n {}".format(word_count_RDD.collect()))
# Notice the '2' values in the output
```

▸ (2) Spark Jobs

```
Each word (key) with an associated 1: [('i', 1), ('like', 1), ('data', 1), ('analytics', 1), ('with', 1), ('pyspark.', 1), ("it's", 1
1), ('are', 1), ('like', 1), ('python.', 1)]
Now, reduce those pairs by summing the 1's when the keys match:
 [('i', 1), ('like', 2), ('are', 1), ("it's", 1), ('python.', 1), ('data', 2), ('analytics', 2), ('pyspark.', 1), ('with', 1)]
```

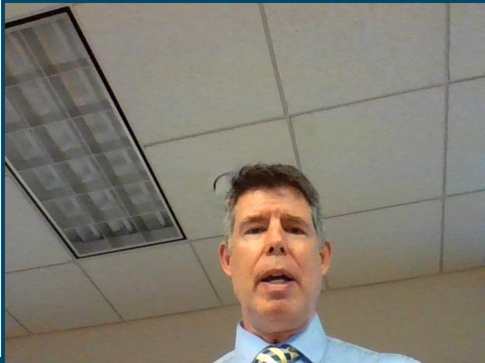
word-count with groupByKey

- Slower than reduceByKey
  - Create a map of (word,1) pairs
  - Group all words
  - Add numbers in group

```
# For every pair (key,valueList), return the key and the sum of the valueList
words_KV_RDD.groupByKey().map(lambda key_value: (key_value[0],sum(key_value[1]))).collect()
```

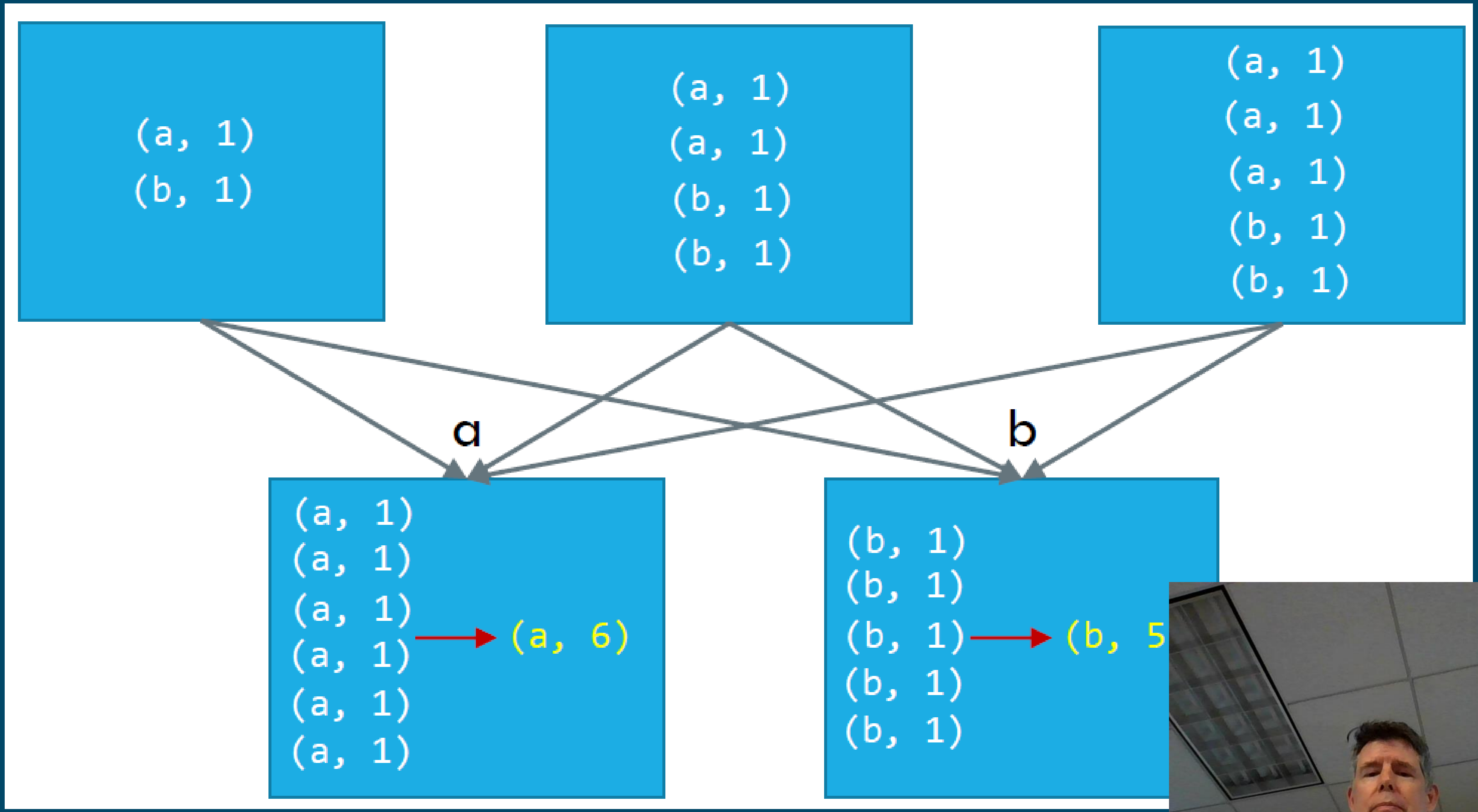
▸ (1) Spark Jobs

```
Out[4]: [('i', 1),
 ('like', 2),
 ('are', 1),
 ("it's", 1),
 ('python.', 1),
 ('data', 2),
 ('analytics', 2),
 ('pyspark.', 1),
 ('with', 1)]
```



Group By Key

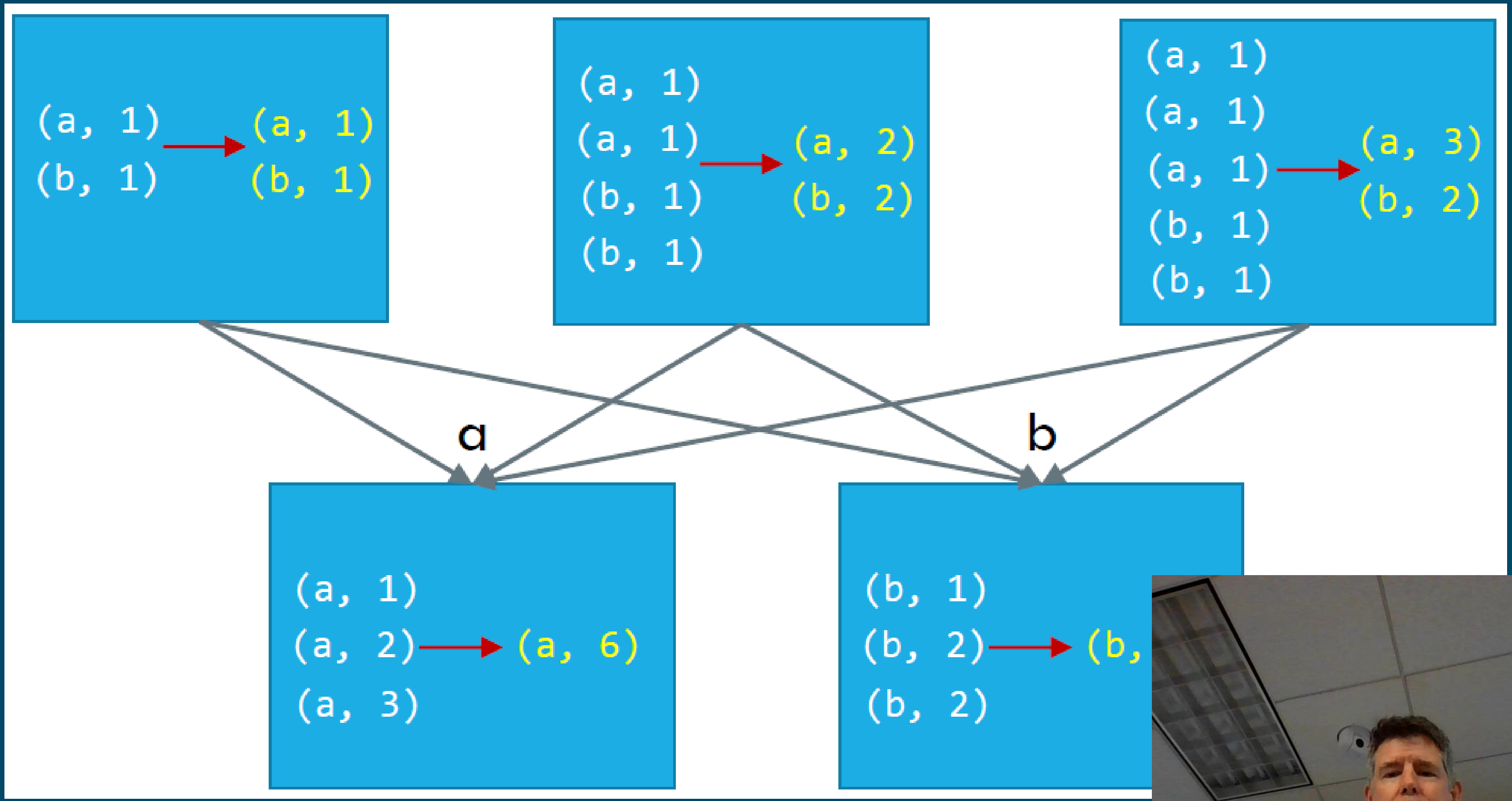
All the key-value pairs are shuffled around, to obtains the groups
More data transfer



Note data transferred…
- All simple <K,V>
- Sum only run in second node

Reduce By Key

Notice how pairs (e.g., <a, 3>) on the same machine with the same key are combined…
So **less data transfer**



Note data transferred…

- Only <K,V>
- Combiner (sum) run in first node
- Sum also run in second node

Prefer reduceByKey over groupByKey

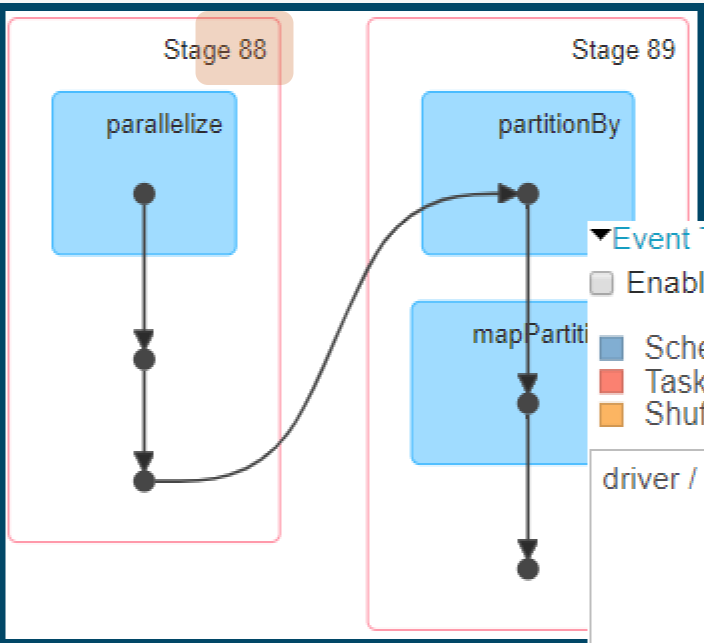
- Less data transit with reduceByKey

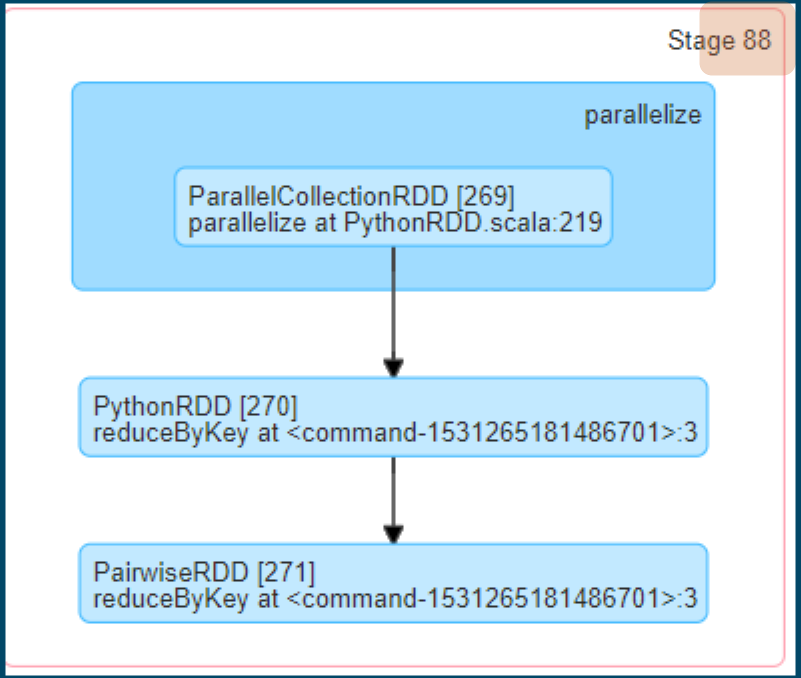
- In general, be aware of how each transformation works
  - Notice how much data is being moved
- Goal is to reduce data movement
  - Do computations local to the node that has the data
  - Don’t move the data to perform computations (unles
    necessary)

reduceByKey with 5 tasks

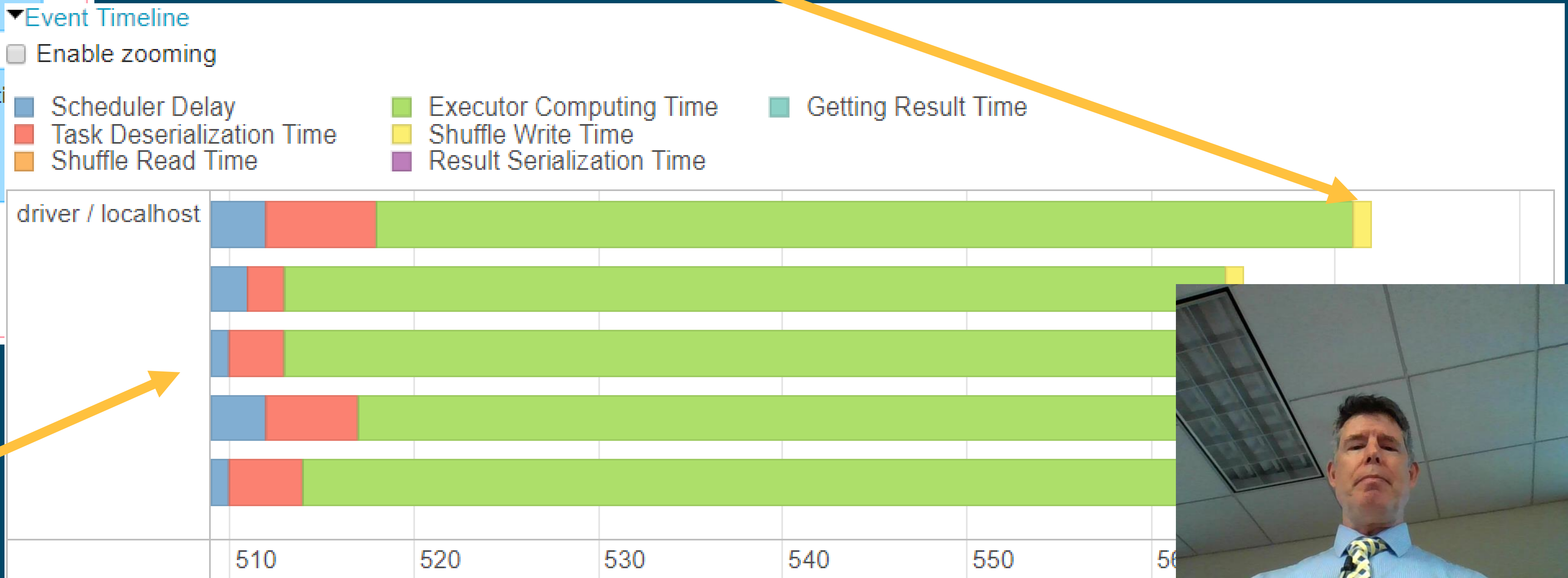
```
data = range(1,100)
rdd = sc.parallelize(data,5)
rdd.map(lambda x: (x%5, x)).reduceByKey(lambda x,y: x + y).collect()
```

(1) Spark Jobs
Job 59 View (Stages: 2/2)





Shuffle shown @ end



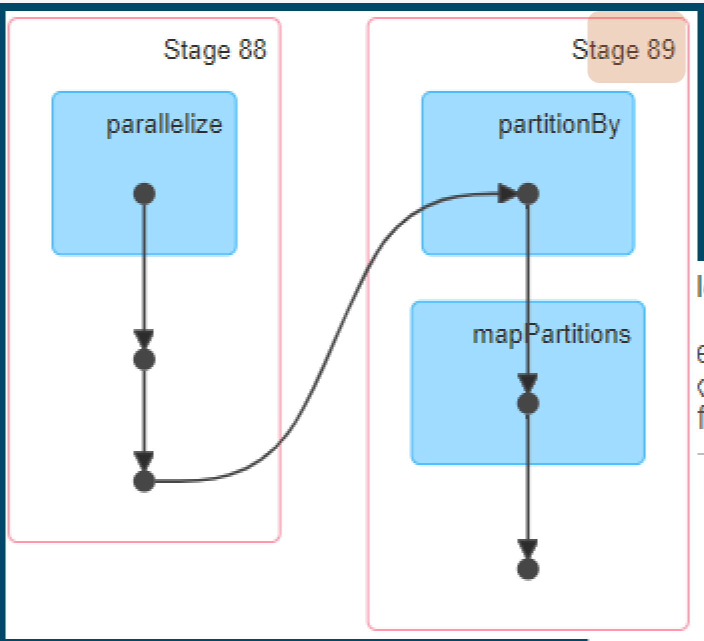
5 Tasks

reduceByKey with 5 tasks

```
data = range(1,100)
rdd = sc.parallelize(data,5)
rdd.map(lambda x: (x%5, x)).reduceByKey(lambda x,y: x + y).collect()
```

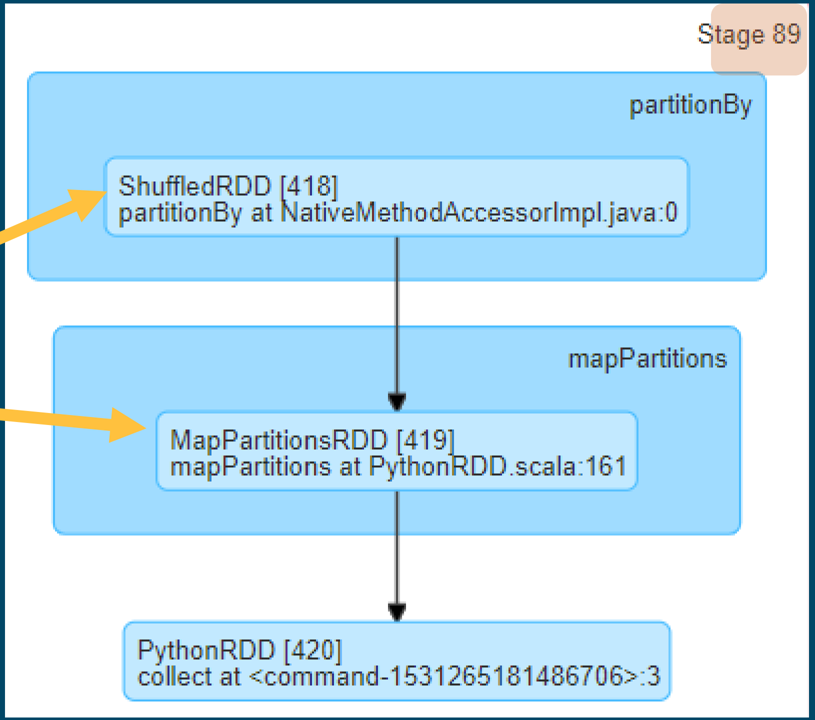
(1) Spark Jobs

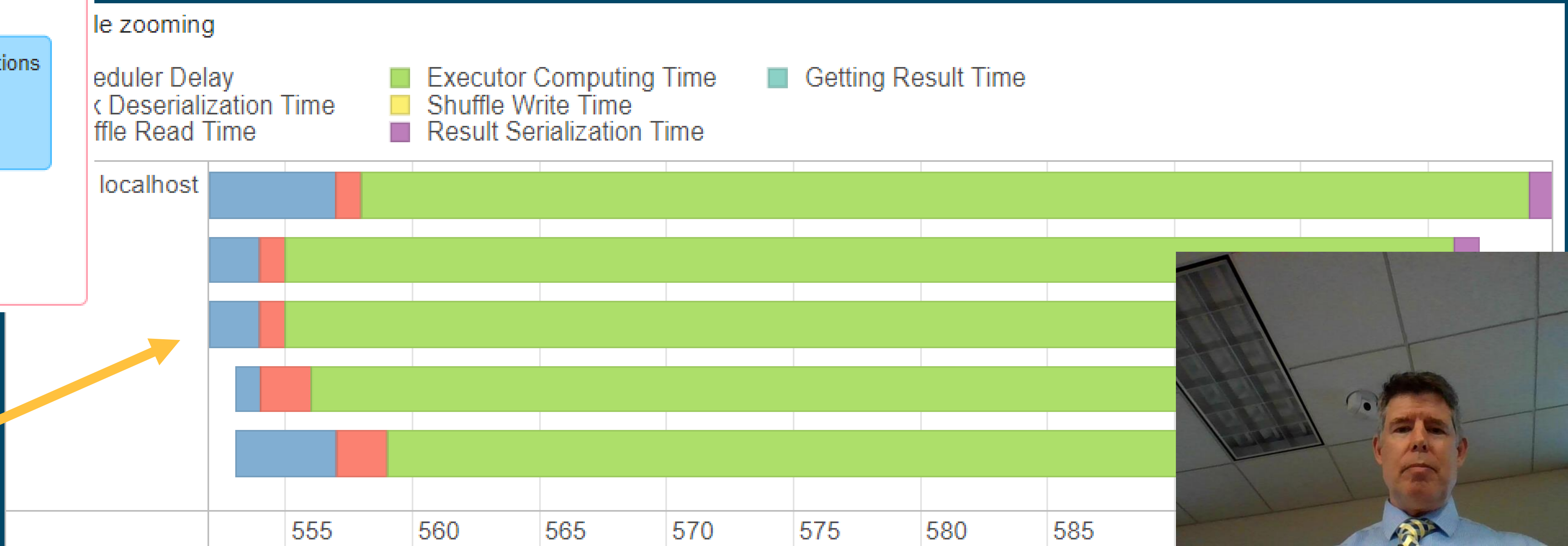
Job 59 View (Stages: 2/2)



Shuffled

Reduce



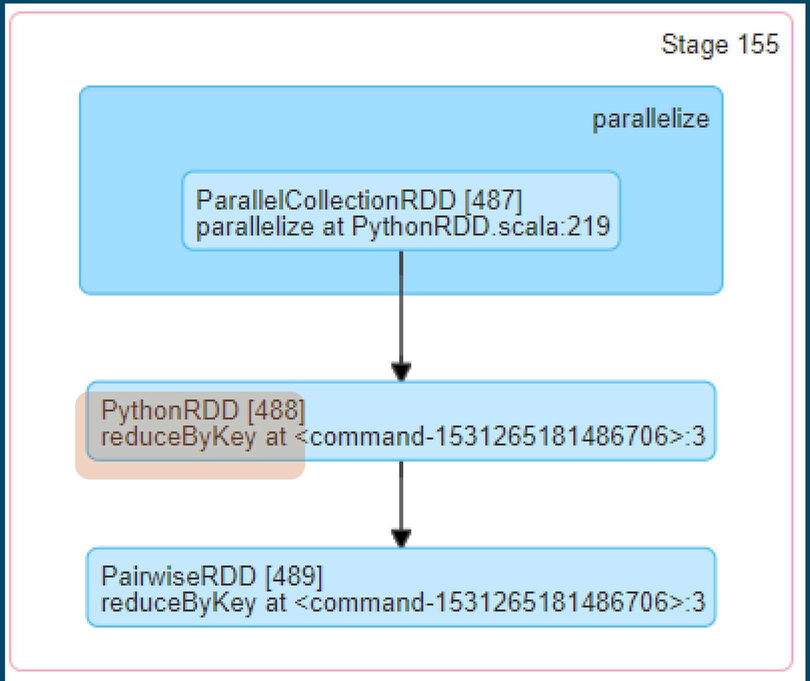


5 Tasks

Map Reduce with 5 tasks

Stage Id ▾	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total
156	8261552548926543378	data = range(1,100) rdd = sc.parallelize(data,5... collect at <command-1531265181486706>:3 +details	2018/11/02 21:09:34	50 ms	5/5
155	8261552548926543378	data = range(1,100) rdd = sc.parallelize(data,5... reduceByKey at <command-1531265181486706>:3 +details	2018/11/02 21:09:33	0.3 s	

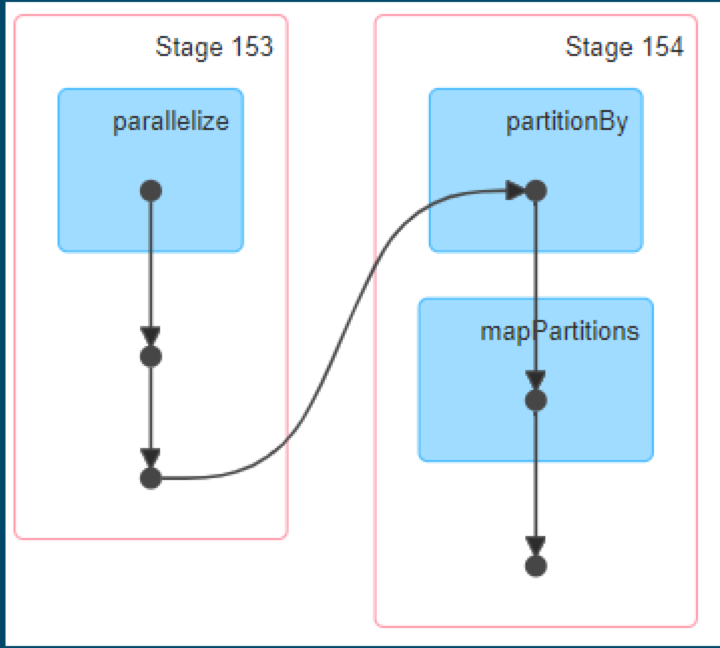
reduceByKey with 5 tasks



Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.2 s	0.2 s	0.2 s	0.2 s	0.3 s
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms
Shuffle Write Size / Records	305.0 B / 5	305.0 B / 5	305.0 B / 5	309.0 B / 5	310.0 B / 5

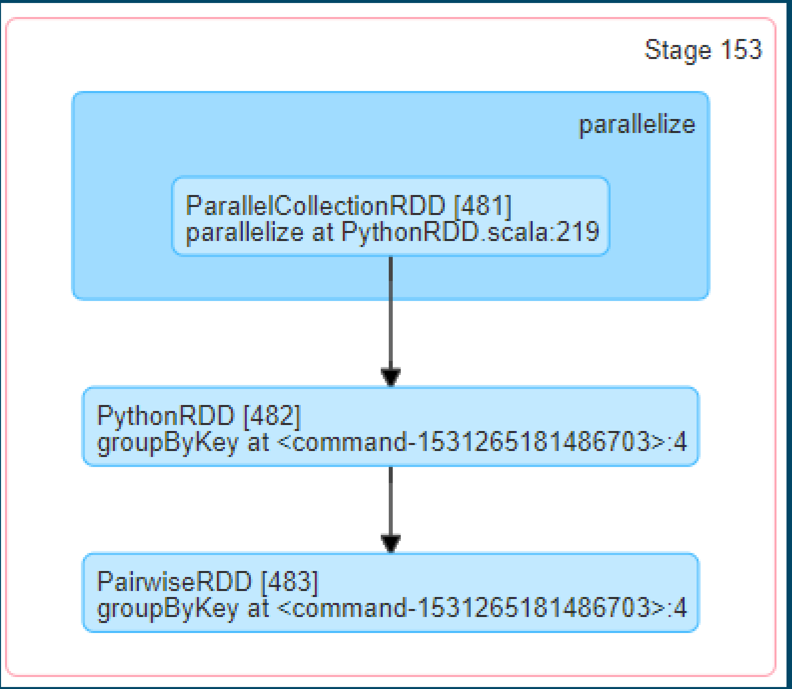
groupByKey with 5 tasks, using

```python
data = range(1,100)
rdd = sc.parallelize(data,5)
rdd.map(lambda x: (x%5, x)).groupByKey().map(lambda x: (x[0], sum(list(x[1])))).collect()
```



Stage Id ▾	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total
154	8261552548926543378	data = range(1,100) rdd = sc.parallelize(data,5... collect at <command-1531265181486703>:4 +details	2018/11/ 21:07:37		
153	8261552548926543378	data = range(1,100) rdd = sc.parallelize(data,5... groupByKey at <command-1531265181486703>:4 +details	2018/11/ 21:07:37		

groupByKey with 5 tasks

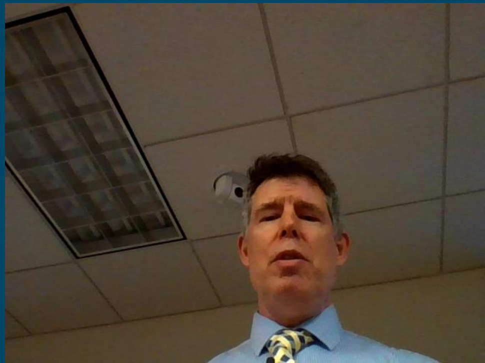


Slower than reduceByKey

Metric	Min	25th percentile	Median	75th perc
Duration	0.2 s	0.2 s	0.2 s	0.2 s
GC Time	0 ms	0 ms	0 ms	0 ms
Shuffle Write Size / Records	358.0 B / 5	358.0 B / 5	358.0 B / 5	358.0 B

reduceByKey shuffles smaller data set

- reduceByKey
  - Duration 50 ms
  - Shuffle min: 305 ms
- groupByKey
  - Duration 61 ms
  - Shuffle min: 358 ms

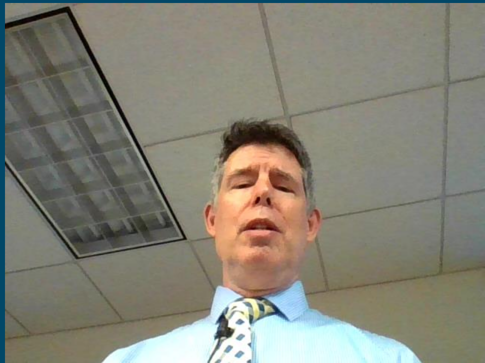


To reduce processing time,

Minimize the number of stages

(wide transformation, shuffle & sort)

a key to good Spark programming



Important to remember

- RDDs are the primitive structure that supports DataFrames
  - RDD data functions are simpler
  - RDD allows manipulation of the partitions
- When possible, minimize data tramission
  - Prefer reduceByKey over groupByKey
    - reduce aggregates local keys, and then send to combine results
    - group by sends all data to create groups, then aggregates