PAIR RODS

There are 2 types of RDDs

Basic RDDs Each element is a single object

Pair RDDs Each element is a **Key/Value pair**

Basic RDDs

So far, we have only worked with Basic RDDs

ie. we treat each record in the RDD as a single object

Basic RDDs

All our transformations, actions act on each record as a whole

There are 2 types of RDDs

Basic RDDs Each element is a single object

Pair RDDs Each element is a **Key/Value pair**

Pair RDDs

Each element is a **Key/Value pair**

Many data processing tasks can be easily expressed using Key, Value pairs

Ex: Delays by Airline, Sales by City, Word Counts etc

Pair RDDs

Pair RDDs are special RDDs where each record is treated as tuple

Pair RDDs

All the basics RDD transformations and actions work for Pair RDDs too

Special Transformations and Actions exist for Pair RDDs

keys
values
mapValues
groupByKey
reduceByKey
combineByKey

A few transformations for pair RDDs

Pair RDDs

Transformations

keys
values
mapValues
groupByKey
reduceByKey
combineByKey

Return RDDs with only the keys or the values

Pair RDDs

Transformations

mapValues

keys
values

Takes a function and applies it on the values of the key, value pairs

groupByKey
reduceByKey
combineByKey

Pair RDDs

Transformations

groupByKey

Groups the values which have the same key into a list/collection

```
BLR,3        BLR,[3,2]
MUM,1        MUM,1
BLR,2
```

Pair RDDs

groupByKey

Transformations

cogroup is also like groupByKey

But it can group values across RDDS

Pair RDDs

Transformations

reduceByKey

This is like reduce on Basic RDDS

It takes a function to combine 2 values

It combines values with the same key

Pair RDDs

Transformations

keys

values

mapValues

groupByKey

combineByKey

reduceByKey

Using a Pair RDD of City, Sales

you can find the sum of sales for each city

Pair RDDs

Transformations

combineByKey

Just as for basic RDDs,
we have reduce and
aggregate

For Pair RDDs, we have
reduceByKey and
combineByKey

Pair RDDs

Transformations

combineByKey

reduce and aggregate

reduceByKey and combineByKey

Note one important difference!

Pair RDDs

Transformations

combineByKey

reduce and aggregate

Actions on basic RDDs

reduceByKey and combineByKey

Transformations on Pair RDDs

Pair RDDs

Transformations

One of the most common operations is

to **merge 2 Pair RDDs**

based on the keys

Pair RDDs

Transformations

Pair RDD1	Pair RDD2	Merge 2 Pair RDDs
BLR,3	BLR,"B"	BLR,[3,"B"]
MUM,1	MUM,"M"	MUM,[1,"M"]
DEL,2	DEL,"D"	DEL,[2,"D"]

Such operations are called joins

Pair RDDs

Transformations

joins

join

left outer join

right outer join

These are similar to their counter parts in SQL

Pair RDDs

Transformations

join

left outer join

right outer join

A join will return a new Pair RDD

Values from the input RDDs **whose keys match** are grouped together

Pair RDDs

Transformations

join

left outer join

right outer join

Only keys which exist in both RDDs are returned

Like an inner join in SQL

Pair RDDs

Transformations

join

Pair RDD1	Pair RDD2		
BLR,3	BLR,"B"		BLR,[3,"B"]
MUM,1	MUM,"M"	→	MUM,[1,"M"]
DEL,2			

Pair RDDs

Transformations

join

left outer join

right outer join

All keys from the left RDD are returned

Pair RDDs

Transformations

left outer join

Pair RDD1	Pair RDD2	
BLR,3	BLR,"B"	BLR,[3,"B"]
MUM,1	MUM,"M"	MUM,[1,"M"]
DEL,2	KOL,"D"	DEL,[2,None]

Pair RDDs

Transformations

join

left outer join

right outer join

All keys from the right RDD are returned

Pair RDDs

Transformations

right outer join

Pair RDD1	Pair RDD2	
BLR,3	BLR,"B"	BLR,[3,"B"]
MUM,1	MUM,"M"	MUM,[1,"M"]
DEL,2	KOL,"D"	KOL,[None,"D"]

Pair RDDs

Actions

countByKey

lookup

collectAsMap

A few special actions are available for pair RDDs

Pair RDDs

Actions

countByKey	count the number of values per key
lookup	returns all values for a specific key
collectAsMap	returns a dict with all the key value pairs

Exploring Airline delays data with PySpark

Part 3

We're back to our Flight related data from USDoT

Let's try doing a few more things with it

1. Compute the average delay per airport

2. Find the top 10 airports based on delay

1. Compute the average delay per airport

2. Find the top 10 airports based on delay

Average delay per airport

For each airport, we need

Sum of all delays

Count of number of flights

Option 1:

Compute these separately

Option 2:

Compute them in the same step

Average delay per airport

For each airport, we need

Sum of all delays

Count of number of flights

Option 1:

reduceByKey

Option 2:

combineByKey

Recap

We've already created an RDD with Flights data

```
flightsParsed=flights.map(lambda x: x.split(',')).map(parse)
```

Each record is represented as a Flight object

```
Flight(date=datetime.date(2014, 4, 1), airline=u'19805', flightnum=u'1', origin=u'JFK', dest=u'LAX', dep=datetim
e(8, 54), dep_delay=-6.0, arv=datetime.time(12, 17), arv_delay=2.0, airtime=355.0, distance=2475.0)
```

Recap

```
', flightnum=u'1', origin=u'JFK', dest=u'LAX', dep=dateti
rv_delay=2.0, airtime=355.0, distance=2475.0)
```

These represent the origin and destination airport codes

Average delay per airport

Option 1: reduceByKey

First let's create a Pair RDD with origin airport and delay for each flight

```
airportDelays = flightsParsed.map(lambda x: (x.origin,x.dep_delay))
```

Average delay per airport

Option 1: reduceByKey

```
airportDelays = flightsParsed.map(lambda x: (x.origin,x.dep_delay))
```

To create a Pair RDD, just make sure each record is a tuple

Average delay per airport

Option 1: reduceByKey

```
airportDelays = flightsParsed.map(lambda x: (x.origin,x.dep_delay))
```

To see if this worked , you can try to **access the keys and values** of the pair RDD

Average delay per airport

Option 1: reduceByKey

access the keys and values

```
airportDelays.keys().take(10)
```

```
[u'JFK',
 u'LAX',
 u'JFK',
 u'LAX',
 u'DFW',
 u'OGG',
 u'DFW',
 u'HNL',
 u'JFK',
 u'LAX']
```

keys() and values() are transformations

```
airportDelays.values().take(10)
```

```
[-6.0, 14.0, -6.0, 25.0, -5.0, 126.0, 125.0, 4.0, -7.0, 21.0]
```

Average delay per airport

Option 1: reduceByKey

```
airportDelays = flightsParsed.map(lambda x: (x.origin,x.dep_delay))
```

On this RDD, we can use reduceByKey twice, once for the sum and again for the count

Average delay per airport

Option 1: reduceByKey

```
airportTotalDelay=airportDelays.reduceByKey(lambda x,y:x+y)
```

First, let's compute the sum

Average delay per airport

Option 1:
reduceByKey

```
airportTotalDelay=airportDelays.reduceByKey(lambda x,y:x+y)
```

This is a new RDD

reduceByKey is a transformation

Average delay per airport

Option 1: reduceByKey

```
=airportDelays.reduceByKey(lambda x,y:x+y)
```

Similar to the reduce action

reduceByKey takes a function that combines 2 elements into 1

Average delay per airport

Option 1:
reduceByKey

```
=airportDelays.reduceByKey(lambda x,y:x+y)
```

The result is a Pair RDD

Keys = Airports
Values = Sum of Delays

Average delay per airport

Option 1: reduceByKey

```
=airportDelays.reduceByKey(lambda x,y:x+y)
```

reduceByKey effectively flattens the Pair RDD

Let's see this visually

```
alDelay=airportDelays.reduceByKey(lambda x,y:x+y)
```

P1

JFK	2
JFK	14
PPG	5
PPG	10
JFK	0
PPG	4

P2

JFK	3
JFK	0
LAX	6
LAX	11
JFK	0
PPG	7

This is the delays RDD

```
alDelay=airportDelays.reduceByKey(lambda x,y:x+y)
```

P1

JFK	2
JFK	14
PPG	5
PPG	10
JFK	0
PPG	4

P2

JFK	3
JFK	0
LAX	6
LAX	11
JFK	0
PPG	7

First, the operation is performed on each partition

```
alDelay=airportDelays.reduceByKey(lambda x,y:x+y)
```

p1

JFK	2
JFK	14
PPG	5
PPG	10
JFK	0
PPG	4

Values with the same key are grouped together

```
alDelay=airportDelays.reduceByKey(lambda x,y:x+y)
```

JFK	2
JFK	14
JFK	0

P1

PPG	5
PPG	10
PPG	4

The given function is applied **on each of these groups**

```
alDelay=airportDelays.reduceByKey(lambda x,y:x+y)
```

P1

JFK	2	x
JFK	14	y
JFK	0	

```
alDelay=airportDelays.reduceByKey(lambda x,y:x+y)
```

JFK	$\boxed{16}$
JFK	0

$x + y$

P1

```
alDelay=airportDelays.reduceByKey(lambda x,y:x+y)
```

P1

JFK	16	x
JFK	0	y

```
alDelay=airportDelays.reduceByKey(lambda x,y:x+y)
```

P1	JFK	16	x + y

```
alDelay=airportDelays.reduceByKey(lambda x,y:x+y)
```

P1

JFK	16

The same thing is done for each key

```
alDelay=airportDelays.reduceByKey(lambda x,y:x+y)
```

P1

JFK	16
PPG	5

The same thing is done for each key

```
alDelay=airportDelays.reduceByKey(lambda x,y:x+y)
```

P1

JFK	16
PPG	5

We have a set of Key value pairs from each node

```
alDelay=airportDelays.reduceByKey(lambda x,y:x+y)
```

P1

JFK	16
PPG	5

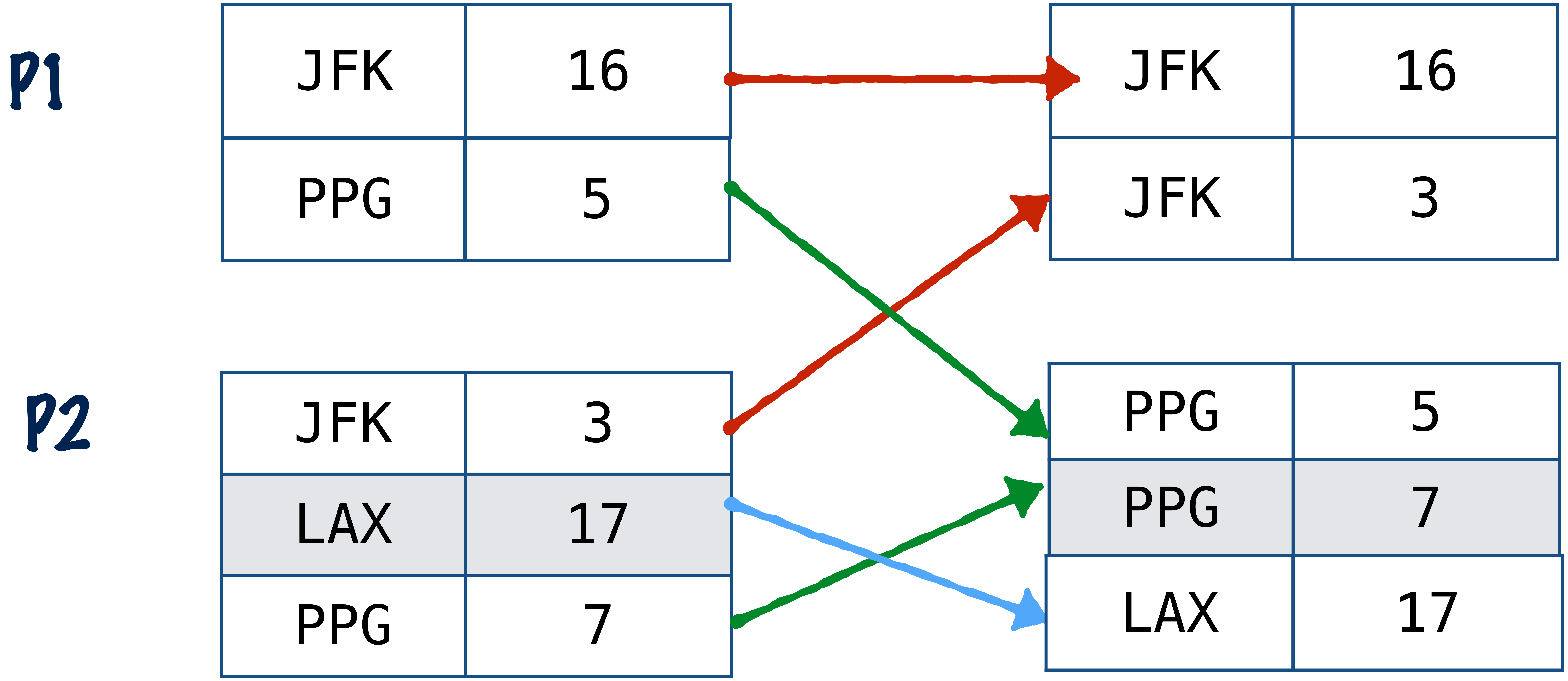
P2

JFK	3
LAX	17
PPG	7

These are shuffled so that all the values with same key are on a single partition

```
alDelay=airportDelays.reduceByKey(lambda x,y:x+y)
```

shuffle



```
alDelay=airportDelays.reduceByKey(lambda x,y:x+y)
```

P1

JFK	16
JFK	3

P2

PPG	5
PPG	7
LAX	17

On each partition, again the reduceByKey operation is applied

```
alDelay=airportDelays.reduceByKey(lambda x,y:x+y)
```

P1

JFK	19

P2

PPG	12
LAX	17

On each partition, again the reduceByKey operation is applied

```
alDelay=airportDelays.reduceByKey(lambda x,y:x+y)
```

Old RDD

P1

JFK	2
JFK	14
PPG	5
PPG	10
JFK	0
PPG	4

P2

JFK	3
JFK	0
LAX	6
LAX	11
JFK	0
PPG	7

New RDD

P1

JFK	19

P2

PPG	12
LAX	17

Average delay per airport

Option 1: reduceByKey

```
airportTotalDelay=airportDelays.reduceByKey(lambda x,y:x+y)
```

We can use the same idea to compute count

```
airportCount=airportDelays.mapValues(lambda x:1).reduceByKey(lambda x,y:x+y)
```

```
airportCount=airportDelays.mapValues(lambda x:1).reduceB
```

This is our RDD with

(origin airport, delay)

```
rtDelays.mapValues(lambda x:1).reduceByKey(lambda x,y:x+y)
```

(origin airport, delay)

mapValues will leave the key as is and apply a function on the value

```
rtDelays.mapValues(lambda x:1).reduceByKey(lambda x,y:x+y)
```

(origin airport, delay) → (origin airport, 1)

All the values are mapped to the constant 1

```
rtDelays.mapValues(lambda x:1).reduceByKey(lambda x,y:x+y)
```

(origin airport, 1)

This will sum all these 1s

effectively giving us count of flights by airport

Average delay per airport

Option 1: reduceByKey

```
airportTotalDelay=airportDelays.reduceByKey(lambda x,y:x+y)
airportCount=airportDelays.mapValues(lambda x:1).reduceByKey(lambda x,y:x+y)
```

The sum and count are in 2 separate RDDs

Average delay per airport

Option 1: reduceByKey

```
airportTotalDelay=airportDelays.reduceByKey(lambda x,y:x+y)
airportCount=airportDelays.mapValues(lambda x:1).reduceByKey(lambda x,y:x+y)
```

We can merge these using a join operation

Average delay per airport

Option 1: reduceByKey

```
airportSumCount=airportTotalDelay.join(airportCount)
```

This will merge the 2 RDDs by matching values with the same key

Average delay per airport

Option 1: reduceByKey

```
airportSumCount=airportTotalDelay.join(airportCount)
```

Sum

JFK	23
LAX	14
PPG	5

Count

JFK	4
LAX	2
PPG	1

sumCount

JFK	(23,4)
LAX	(14,2)
PPG	(5,1)

Average delay per airport

Option 1: reduceByKey

```
airportSumCount=airportTotalDelay.join(airportCount)
```

sumCount

JFK	(23,4)
LAX	(14,2)
PPG	(5,1)

We need to divide the sum by the count

Average delay per airport

Option 1: reduceByKey

```
airportAvgDelay=airportSumCount.mapValues(lambda x : x[0]/float(x[1]))
```

sumCount

JFK	(23,4)
LAX	(14,2)
PPG	(5,1)

Once again, we can use mapValues to do this

Average delay per airport

Option 1: reduceByKey

```
airportTotalDelay=airportDelays.reduceByKey(lambda x,y:x+y)
airportCount=airportDelays.mapValues(lambda x:1).reduceByKey(lam
airportSumCount=airportTotalDelay.join(airportCount)
```

```
airportAvgDelay=airportSumCount.mapValues(lambda x : x[0]/float(x[1]))
```

With reduceByKey, it took 3 steps to compute the average delay per airport

Average delay per airport

Option 1: reduceByKey

```
airportTotalDelay=airportDelays.reduceByKey(lambda x,y:x+y)
airportCount=airportDelays.mapValues(lambda x:1).reduceByKey(lam
airportSumCount=airportTotalDelay.join(airportCount)
```

1. Compute the sum in 1 RDD
2. Compute the count in another RDD
3. Join the 2 RDDs

Average delay per airport

Option 1: reduceByKey

```
airportTotalDelay=airportDelays.reduceByKey(lambda x,y:x+y)
airportCount=airportDelays.mapValues(lambda x:1).reduceByKey(lam
airportSumCount=airportTotalDelay.join(airportCount)
```

With combineByKey, we can do all this in one step

Average delay per airport

For each airport, we need

Sum of all delays

Count of number of flights

Option 1:

reduceByKey

Option 2:

combineByKey

Average delay per airport

Option 2: combineByKey

```
airportSumCount2=airportDelays.combineByKey((lambda value:(value,1)),
                                            (lambda acc, value: (acc[0]+value,acc[1]+1)),
                                            (lambda acc1, acc2: (acc1[0]+acc2[0],acc1[1]+acc2[1])))
```

This is a new RDD

combineByKey is a transformation

Average delay per airport

Option 2: combineByKey

```
airportSumCount2=airportDelays.combineByKey((lambda value:(value,1)),
                                            (lambda acc, value: (acc[0]+value,acc[1]+1)),
                                            (lambda acc1, acc2: (acc1[0]+acc2[0],acc1[1]+acc2[1])))
```

combineByKey is similar to aggregate

Average delay per airport

Option 2: combineByKey

```
airportSumCount2=airportDelays.combineByKey((lambda value:(value,1)),
                                             (lambda acc, value: (acc[0]+value,acc[1]+1)),
                                             (lambda acc1, acc2: (acc1[0]+acc2[0],acc1[1]+acc2[1])))
```

combineByKey requires 3 functions

Average delay per airport

Option 2: combineByKey

```
airportSumCount2=airportDelays.combineByKey((lambda value:(value,1)),
                                            (lambda acc, value: (acc[0]+value,acc[1]+1)),
                                            (lambda acc1, acc2: (acc1[0]+acc2[0],acc1[1]+acc2[1])))
```

createCombiner Function

This initializes a value when a key is first seen within a partition

Average delay per airport

Option 2: combineByKey

```
airportSumCount2=airportDelays.combineByKey((lambda value:(value,1)),
                                            (lambda acc, value: (acc[0]+value,acc[1]+1)),
                                            (lambda acc1, acc2: (acc1[0]+acc2[0],acc1[1]+acc2[1])))
```

merge Function

This specifies how values with the same key should be combined

within a partition

Average delay per airport

Option 2: combineByKey

```
airportSumCount2=airportDelays.combineByKey((lambda value:(value,1)),
                                            (lambda acc, value: (acc[0]+value,acc[1]+1)),
                                            (lambda acc1, acc2: (acc1[0]+acc2[0],acc1[1]+acc2[1])))
```

mergeCombiners Function

This specifies how the results from each partition should be combined

Average delay per airport

Option 2: combineByKey

```
.combineByKey((lambda value:(value,1)),
              (lambda acc, value: (acc[0]+value,acc[1]+1)),
              (lambda acc1, acc2: (acc1[0]+acc2[0],acc1[1]+acc2[1])))
```

createCombiner
merge
mergeCombiners

With combineByKey we have **very granular control** over how the computation should happen

Average delay per airport

Option 2: combineByKey

```
.combineByKey((lambda value:(value,1)),
              (lambda acc, value: (acc[0]+value,acc[1]+1)),
              (lambda acc1, acc2: (acc1[0]+acc2[0],acc1[1]+acc2[1])))
```

Let's see this visually

Average delay per airport

Option 2: combineByKey

```
.combineByKey((lambda value:(value,1)),
              (lambda acc, value: (acc[0]+value,acc[1]+1)),
              (lambda acc1, acc2: (acc1[0]+acc2[0],acc1[1]+acc2[1])))
```

P1

JFK	2
JFK	12
PPG	5
PPG	10
JFK	0
PPG	4

P2

JFK	3
JFK	0
LAX	6
LAX	11
JFK	0
PPG	7

This the delays RDD

Average delay per airport

Option 2: combineByKey

```
.combineByKey((lambda value:(value,1)),
              (lambda acc, value: (acc[0]+value,acc[1]+1)),
              (lambda acc1, acc2: (acc1[0]+acc2[0],acc1[1]+acc2[1])))
```

P1

JFK	2
JFK	12
PPG	5
PPG	10
JFK	0
PPG	4

P2

JFK	3
JFK	0
LAX	6
LAX	11
JFK	0
PPG	7

First, the operation is performed on each partition

Average delay per airport

Option 2: combineByKey

```
.combineByKey((lambda value:(value,1)),
              (lambda acc, value: (acc[0]+value,acc[1]+1)),
              (lambda acc1, acc2: (acc1[0]+acc2[0],acc1[1]+acc2[1])))
```

P1

JFK	2
JFK	12
PPG	5
PPG	10
JFK	0
PPG	4

We start with the first record

Average delay per airport

Option 2: combineByKey

```
.combineByKey((lambda value:(value,1)),
              (lambda acc, value: (acc[0]+value,acc[1]+1)),
              (lambda acc1, acc2: (acc1[0]+acc2[0],acc1[1]+acc2[1])))
```

P1

JFK	2
JFK	12
PPG	5
PPG	10
JFK	0
PPG	4

This is first time the key "JFK" is seen

Average delay per airport

Option 2: combineByKey

```
.combineByKey((lambda value:(value,1)),
              (lambda acc, value: (acc[0]+value,acc[1]+1)),
              (lambda acc1, acc2: (acc1[0]+acc2[0],acc1[1]+acc2[1])))
```

P1

JFK	2
JFK	12
PPG	5
PPG	10
JFK	0
PPG	4

JFK, (2,1)

The createCombiner function is used to initialize a value

Average delay per airport

Option 2: combineByKey

```
.combineByKey((lambda value:(value,1)),
              (lambda acc, value: (acc[0]+value,acc[1]+1)),
              (lambda acc1, acc2: (acc1[0]+acc2[0],acc1[1]+acc2[1])))
```

P1

JFK	2
JFK	12
PPG	5
PPG	10
JFK	0
PPG	4

JFK, (2,1)

The next step depends on whether it's a new key or a key that's already seen

Average delay per airport

Option 2: combineByKey

```
.combineByKey((lambda value:(value,1)),
              (lambda acc, value: (acc[0]+value,acc[1]+1)),
              (lambda acc1, acc2: (acc1[0]+acc2[0],acc1[1]+acc2[1])))
```

P1

JFK	2
JFK	12
PPG	5
PPG	10
JFK	0
PPG	4

JFK, (2,1)

The key "JFK" has already been seen

Average delay per airport

Option 2: combineByKey

```
.combineByKey((lambda value:(value,1)),
              (lambda acc, value: (acc[0]+value,acc[1]+1)),
              (lambda acc1, acc2: (acc1[0]+acc2[0],acc1[1]+acc2[1])))
```

P1

JFK	2
JFK	12
PPG	5
PPG	10
JFK	0
PPG	4

JFK, (2,1) → acc

→ value

The merge function is used

Average delay per airport

Option 2: combineByKey

```
.combineByKey((lambda value:(value,1)),
              (lambda acc, value: (acc[0]+value,acc[1]+1)),
              (lambda acc1, acc2: (acc1[0]+acc2[0],acc1[1]+acc2[1])))
```

P1

JFK	2
JFK	12
PPG	5
PPG	10
JFK	0
PPG	4

JFK, (14,2)

"PPG" is a new key

Average delay per airport

Option 2: combineByKey

```
.combineByKey((lambda value:(value,1)),
              (lambda acc, value: (acc[0]+value,acc[1]+1)),
              (lambda acc1, acc2: (acc1[0]+acc2[0],acc1[1]+acc2[1])))
```

P1

JFK	2
JFK	12
PPG	5
PPG	10
JFK	0
PPG	4

JFK, (14,2)

PPG, (5,1)

Continue until all records are processed

Average delay per airport

Option 2: combineByKey

```
.combineByKey((lambda value:(value,1)),
              (lambda acc, value: (acc[0]+value,acc[1]+1)),
              (lambda acc1, acc2: (acc1[0]+acc2[0],acc1[1]+acc2[1])))
```

P1

JFK	2
JFK	12
PPG	5
PPG	10
JFK	0
PPG	4

→ value

JFK, (14,2)

PPG, (5,1) → acc

Average delay per airport

Option 2: combineByKey

```
.combineByKey((lambda value:(value,1)),
              (lambda acc, value: (acc[0]+value,acc[1]+1)),
              (lambda acc1, acc2: (acc1[0]+acc2[0],acc1[1]+acc2[1])))
```

P1

JFK	2
JFK	12
PPG	5
PPG	10
JFK	0
PPG	4

JFK, (14,2) → acc

PPG, (15,2)

JFK 0 → value

Average delay per airport

Option 2: combineByKey

```
.combineByKey((lambda value:(value,1)),
              (lambda acc, value: (acc[0]+value,acc[1]+1)),
              (lambda acc1, acc2: (acc1[0]+acc2[0],acc1[1]+acc2[1])))
```

P1

JFK	2
JFK	12
PPG	5
PPG	10
JFK	0
PPG	4

JFK, (14,3)

PPG, (15,2) → acc

value

Average delay per airport

Option 2:
combineByKey

```
.combineByKey((lambda value:(value,1)),
              (lambda acc, value: (acc[0]+value,acc[1]+1)),
              (lambda acc1, acc2: (acc1[0]+acc2[0],acc1[1]+acc2[1])))
```

P1

JFK, (14,3)

PPG, (19,3)

The same thing is done on each Partition

Average delay per airport

Option 2: combineByKey

```
.combineByKey((lambda value:(value,1)),
              (lambda acc, value: (acc[0]+value,acc[1]+1)),
              (lambda acc1, acc2: (acc1[0]+acc2[0],acc1[1]+acc2[1])))
```

P1

```
JFK, (14,3)
PPG, (19,3)
```

P2

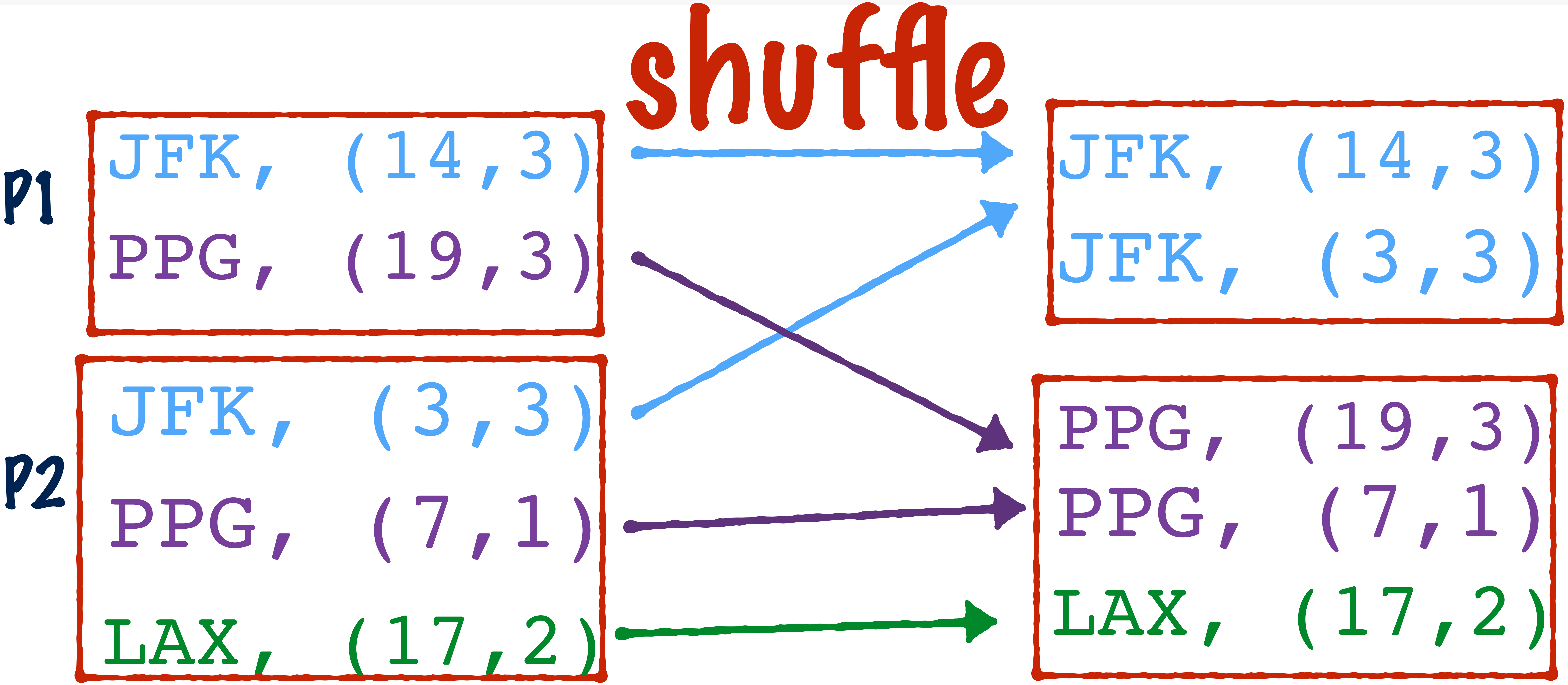
```
JFK, (3,3)
PPG, (7,1)
LAX, (17,2)
```

The records are shuffled until all records with the same key are on the same partition

Average delay per airport

Option 2: combineByKey

```
.combineByKey((lambda value:(value,1)),
              (lambda acc, value: (acc[0]+value,acc[1]+1)),
              (lambda acc1, acc2: (acc1[0]+acc2[0],acc1[1]+acc2[1])))
```



Average delay per airport

Option 2: combineByKey

```
.combineByKey((lambda value:(value,1)),
              (lambda acc, value: (acc[0]+value,acc[1]+1)),
              (lambda acc1, acc2: (acc1[0]+acc2[0],acc1[1]+acc2[1])))
```

P1

```
JFK, (14,3)
JFK, (3,3)
```

P2

```
PPG, (19,3)
PPG, (7,1)
LAX, (17,2)
```

The records in each partition are processed

Average delay per airport

Option 2: combineByKey

```
.combineByKey((lambda value:(value,1)),
              (lambda acc, value: (acc[0]+value,acc[1]+1)),
              (lambda acc1, acc2: (acc1[0]+acc2[0],acc1[1]+acc2[1])))
```

P1

```
JFK, (14,3)
JFK, (3,3)
```

Values with the same key are combined using the mergeCombiners function

Average delay per airport

Option 2: combineByKey

```
.combineByKey((lambda value:(value,1)),
              (lambda acc, value: (acc[0]+value,acc[1]+1)),
              (lambda acc1, acc2: (acc1[0]+acc2[0],acc1[1]+acc2[1])))
```

P1

JFK, (14,3) → acc1

JFK, (3,3) → acc2

Average delay per airport

Option 2: combineByKey

```
.combineByKey((lambda value:(value,1)),
              (lambda acc, value: (acc[0]+value,acc[1]+1)),
              (lambda acc1, acc2: (acc1[0]+acc2[0],acc1[1]+acc2[1])))
```

P1 JFK, (17,6)

Similarly, each partition is processed

Average delay per airport

Option 2: combineByKey

```
airportSumCount2=airportDelays.combineByKey((lambda value
                                            (lambda acc,
                                            (lambda acc1,
```

P1 JFK, (17,6)

P2 PPG, (26,4)
LAX, (17,2)

Using a single transformation, we could compute the sum and the count

1. Compute the average delay per airport

2. Find the top 10 airports based on delays

The top 10 airports based on delay

```
airportAvgDelay
```

We already have the average delay per airport

The top 10 airports based on delay

`airportAvgDelay`

It's a pair RDD with

keys=Airports

Values=Avg Delay per Airport

The top 10 airports based on delay

```
airportAvgDelay
```

We want to sort this RDD

In descending order of value

The top 10 airports based on delay

```
airportAvgDelay.sortBy(lambda x:-x[1])
```

Descending order of value

sortBy is a transformation

It can be used with both Basic and Paired RDDS

The top 10 airports based on delay

```
airportAvgDelay.sortBy(lambda x:-x[1])
```

This is the avg delay per airport ie. the value in the key value pair

The top 10 airports based on delay

```
airportAvgDelay.sortBy(lambda x: -x[1])
```

By reversing the sign, we are able to sort in descending order

The top 10 airports based on delay

```
airportAvgDelay.sortBy(lambda x:-x[1]).take(10)
```

```
[(u'PPG', 56.25),
 (u'EGE', 32.0),
 (u'OTH', 24.533333333333335),
 (u'LAR', 18.892857142857142),
 (u'RDD', 18.529411764705909),
 (u'MTJ', 18.363636363636363),
 (u'PUB', 17.54),
 (u'EWR', 16.478549005929544),
 (u'CIC', 15.931034482758621),
 (u'RST', 15.6993006993007)]
```

This will give us the top 10 airports and their avg delay

The top 10 airports based on delay

```
[(u'PPG', 56.25),
 (u'EGE', 32.0),
 (u'OTH', 24.533333333333335),
 (u'LAR', 18.892857142857142),
 (u'RDD', 18.55294117647059),
 (u'MTJ', 18.363636363636363),
 (u'PUB', 17.54),
 (u'EWR', 16.478549005929544),
 (u'CIC', 15.931034482758621),
 (u'RST', 15.6993006993007)]
```

These are the airport Codes

Which airports do these refer to?

The top 10 airports based on delay

```
[(u'PPG', 56.25),
 (u'EGE', 32.0),
 (u'OTH', 24.533333333333335),
 (u'LAR', 18.892857142857142),
 (u'RDD', 18.55294117647059),
 (u'MTJ', 18.363636363636363),
 (u'PUB', 17.54),
 (u'EWR', 16.478549005929544),
 (u'CIC', 15.931034482758621),
 (u'RST', 15.6993006993007)]
```

We can use the airports.csv file to find out

1. Compute the average delay per airport

2. Find the top 10 airports based on delays

Looking up Airport Descriptions

Recap

There are 3 files

flights.csv	Flight id, airline, airport, departure, arrival , delay
airlines.csv	airline id, airline name
airports.csv	airport id, airport name

Looking up Airport Descriptions

Let's load and parse the airports.csv file

```
import csv
from StringIO import StringIO
def split(line):

    reader = csv.reader(StringIO(line))
    return reader.next()

def notHeader(row):
    return "Description" not in row

|
airports=sc.textFile(airportsPath).filter(notHeader).map(split)
```

Looking up Airport Descriptions

```
import csv
from StringIO import StringIO
def split(line):

    reader = csv.reader(StringIO(line))
    return reader.next()

def notHeader(row):
    return "Description" not in row

airports=sc.textFile(airportsPath).filter(not
```

A function to filter out the header row

Looking up Airport Descriptions

```
import csv
from StringIO import StringIO
def split(line):

    reader = csv.reader(StringIO(line))
    return reader.next()

def notHeader(row):
    return "Description" not in row

airports=sc.textFile(airportsPath).filter(not
```

We take help from Python's csv module for a function to cleanly parse the rows

Looking up Airport Descriptions

```
import csv
from StringIO impor
def split(line):

    reader = csv.re
    return reader.n

def notHeader(row):
    return "Descrip

airports=sc.textFile(airportsPath).filter(notHeader).map(split)
```

We use the functions to load the data into an RDD and parse the rows

Looking up Airport Descriptions

```
airports=sc.textFile(airportsPath).filter(notHeader).map(split)
```

airports is also a Pair RDD

A list with 2 elements in each record, this Python treats exactly like a Pair RDD

Looking up Airport Descriptions

```
airports=sc.textFile(airportsPath).filter(notHeader).map(split)
```

airports is also a Pair RDD

Each pair has
(Airport Code,
Airport Description)

Looking up Airport Descriptions

```
airports=sc.textFile(airportsPath).filter(notHeader).map(split)
```

We have 3 options on how to use this RDD

lookup action

map

broadcast

Looking up Airport Descriptions lookup action

```
airports=sc.textFile(airportsPath).filter(notHeader).map(split)
```

Pair RDDs have an action called lookup

Looking up Airport Descriptions lookup action

```
[(u'PPG', 56.25),
 (u'EGE', 32.0),
 (u'OTH', 24.533333333333335),
 (u'LAR', 18.892857142857142),
 (u'RDD', 18.55294117647059),
 (u'MTJ', 18.363636363636363),
 (u'PUB', 17.54),
 (u'EWR', 16.478549005929544),
 (u'CIC', 15.931034482758621),
 (u'RST', 15.699300699300707)]
```

We can use lookup for each of these airport Codes

Looking up Airport Descriptions

```
[(u'PPG', 56.25),
 (u'EGE', 32.0),
 (u'OTH', 24.533333333333335),
 (u'LAR', 18.892857142857142),
 (u'RDD', 18.55294117647059),
 (u'MTJ', 18.363636363636363),
 (u'PUB', 17.54),
 (u'EWR', 16.478549005929544),
 (u'CIC', 15.931034482758621),
 (u'RST', 15.6993006993007)]
```

```
airports.lookup('PPG')
```

```
['Pago Pago, TT: Pago Pago International']
```

We can use lookup for each of these airport Codes

Looking up Airport Descriptions lookup action

```
airports.lookup('PPG')
```

```
['Pago Pago, TT: Pago Pago International']
```

That's an airport on a tiny little island in the South Pacific Ocean

Looking up Airport Descriptions lookup action

```
airports.lookup('PPG')
```

```
['Pago Pago, TT: Pago Pago International']
```

Did you know that the US has an island territory there in a little country called Samoa?

Huh!

You learn something new all the time :)

Looking up Airport Descriptions lookup action

```
airports.lookup('PPG')
```

```
['Pago Pago, TT: Pago Pago International']
```

Back to our example,

It's a little tedious to have to do this for each airport

Looking up Airport Descriptions

We have 3 options on how to use this RDD

lookup action ✓
dictionary
broadcast

Looking up Airport Descriptions

dictionary

We can **build a map** with all airports from the RDD and use that

```
airportLookup=airports.collectAsMap()
```

collectAsMap is an action

Looking up Airport Descriptions

dictionary

```
airportLookup=airports.collectAsMap()
```

```
airportLookup
```

```
{'SPY': "San Pedro, Cote d'Ivoire: San Pedro Airport",
 'SPZ': 'Springdale, AR: Springdale Municipal',
 'SPP': 'Menongue, Angola: Menongue Airport',
 'SPQ': 'San Pedro, CA: Catalina Air-Sea Terminal Heliport',
 'SPR': 'San Pedro, Belize: San Pedro Airport',
 'SPS': 'Wichita Falls, TX: Sheppard AFB/Wichita Falls Municipal',
 'SPU': 'Split, Croatia: Split Airport',
 'SPW': 'Spencer, IA: Spencer Municipal',
 'SPH': 'Sopu, Papua New Guinea: Sopu Airport',
 'SPI': 'Springfield, IL: Abraham Lincoln Capital',
 'SPK': 'Sapporo, Japan: Chitose AB',
 'SPM': 'Spangdahlem, Germany: Spangdahlem AB',
 'SPN': 'Saipan, TT: Francisco C. Ada Saipan International',
 'SPA': 'Spartanburg, SC: Spartanburg Downtown Memorial',
 'SPB': 'Charlotte Amalie, VI: Charlotte Amalie Harbor Seaplane Base',
 'SPC': 'Santa Cruz de la Palma, Spain: La Palma',
 'SPD': 'Saidpur, Bangladesh: Saidpur Airport',
 'SPE': 'Sepulot, Malaysia: Sepulot Airport',
 'SPF': 'Spearfish, SD: Black Hills Clyde Ice Field',
```

collectAsMap returns a dictionary with all the key value pairs in the RDD

Looking up Airport Descriptions

dictionary

```
airportLookup=airports.collectAsMap()
```

```
airportAvgDelay.map(lambda x: (airportLookup[x[0]],x[1]))
```

airportAvgDelay is a Pair RDD with

(Airport Code, Avg Delay)

For each record, this function **replaces the Airport Code** with corresponding Description

Looking up Airport Descriptions

dictionary

```
airportAvgDelay.map(lambda x: (airportLookup[x[0]],x[1]))
```

(Airport Code, Avg Delay)

It uses the airportLookup dict to do so

Looking up Airport Descriptions

dictionary

```
airportAvgDelay.map(lambda x: (airportLookup[x[0]],x[1]))
```

(Airport Code, Avg Delay)

(Airport Desc, Avg Delay)

airportLookup

```
{'SPY': "San Pedro, Cote d'Ivoire: San Pedro Airport",
 'SPZ': 'Springdale, AR: Springdale Municipal',
 'SPP': 'Menongue, Angola: Menongue Airport',
 'SPQ': 'San Pedro, CA: Catalina Air-Sea Terminal Heliport',
 'SPR': 'San Pedro, Belize: San Pedro Airport',
 'SPS': 'Wichita Falls, TX: Sheppard AFB/Wichita Falls Municipal',
 'SPU': 'Split, Croatia: Split Airport',
 'SPW': 'Spencer, IA: Spencer Municipal',
 'SPH': 'Sopu, Papua New Guinea: Sopu Airport',
 'SPI': 'Springfield, IL: Abraham Lincoln Capital',
 'SPK': 'Sapporo, Japan: Chitose AB',
 'SPM': 'Spangdahlem, Germany: Spangdahlem AB',
 'SPN': 'Saipan, TT: Francisco C. Ada Saipan International',
 'SPA': 'Spartanburg, SC: Spartanburg Downtown Memorial',
 'SPB': 'Charlotte Amalie, VI: Charlotte Amalie Harbor Seaplane Base',
 'SPC': 'Santa Cruz de la Palma, Spain: La Palma',
```

Looking up Airport Descriptions

dictionary

```
airportAvgDelay.map(lambda x: (airportLookup[x[0]],x[1]))
```

This is a function

Spark distributes this function to all the nodes with the partitions of the RDD

Looking up Airport Descriptions

dictionary

```
airportAvgDelay.map(lambda x: (airportLookup[x[0]],x[1]))
```

This function uses the airportLookup variable

It will carry it's own copy of this variable to each node

Looking up Airport Descriptions

dictionary

```
airportAvgDelay.map(lambda x: (airportLookup[x[0]],x[1]))
```

This is a property of closure functions

Closure functions are pretty complicated

Looking up Airport Descriptions

dictionary

```
airportAvgDelay.map(lambda x: (airportLookup[x[0]],x[1]))
```

Closure functions are pretty complicated

But they help Spark take complex operations defined by users and apply them on RDDs

Looking up Airport Descriptions

dictionary

```
airportAvgDelay.map(lambda x: (airportLookup[x[0]],x[1]))
```

But they help Spark take complex operations defined by users and apply them on RDDs

while keeping the user completely abstracted from the complexities of distributed computing

Looking up Airport Descriptions

dictionary

```
airportAvgDelay.map(lambda x: (airportLookup[x[0]],x[1]))
```

We mentioned that a copy of airportLookup variable is carried over to each node in the cluster

Looking up Airport Descriptions

dictionary

```
airportAvgDelay.map(lambda x: (airportLookup[x[0]],x[1]))
```

What if we had to use this lookup in multiple operations?

Looking up Airport Descriptions

dictionary

```
airportAvgDelay.map(lambda x: (airportLookup[x[0]],x[1]))
```

The airportLookup variable would be copied over to the nodes every time!

Looking up Airport Descriptions

dictionary

```
airportAvgDelay.map(lambda x: (airportLookup[x[0]],x[1]))
```

What if we could cache this variable on each of the nodes, so it can be reused?

Looking up Airport Descriptions

dictionary

```
airportAvgDelay.map(lambda x: (airportLookup[x[0]],x[1]))
```

This is exactly why Spark provides

Broadcast Variables

Looking up Airport Descriptions

We have 3 options on how to use this RDD

lookup action

map

broadcast

Spark allows you to define

Broadcast Variables

Looking up Airport Descriptions

broadcast

Broadcast Variables

These variables are 'broadcast' to all the nodes in the cluster

Looking up Airport Descriptions

broadcast

Broadcast Variables

They have 3 characteristics

Immutable

Distributed to all the nodes in the cluster

In-memory

Looking up Airport Descriptions

broadcast

Immutable

Broadcast variables are immutable

They cannot be changed after creation

Looking up Airport Descriptions

broadcast

Broadcast Variables

immutable

Distributed to all the nodes in the cluster

in-memory

Looking up Airport Descriptions

broadcast

Broadcast Variables

in-memory

Broadcast variables are cached in memory

So, they should not be too large

Looking up Airport Descriptions

broadcast

```
airportBC=sc.broadcast(airportLookup)
```

This is a broadcast variable

It has a method to lookup the value for a key

Looking up Airport Descriptions

broadcast

```
airportBC=sc.broadcast(airportLookup)
```

```
airportAvgDelay.map(lambda x: (airportBC.value[x[0]],x[1]))
```

This is pretty similar to how we used the dictionary

Looking up Airport Descriptions broadcast

```
lay.map(lambda x: (airportBC.value[x[0]],x[1]))
```

This is pretty similar to how we used the dictionary

But it's **much more efficient** because the broadcast variable is cached