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

```
val flightsParsed=flights.map(parse)
```

Each record is represented as a Flight object

```
Flight(2014-01-01,19805,1,JFK,LAX,08:54:00.000,-6.0,12:17:00.000,2.0,355.0,2475.0)
```

Recap

```
t(2014-01-01,19805,1,JFK,LAX,08:54:00.000,-6.0,12
```

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

```
val airportDelays = flightsParsed.map(x => (x.origin,x.dep_delay))
```

Average delay per airport

Option 1: reduceByKey

```
val airportDelays = flightsParsed.map(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

```
val airportDelays = flightsParsed.map(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)
```

```
Array(JFK, LAX, JFK, LAX, DFW, OGG, DFW, HNL, JFK, LAX)
```

keys() and values() are transformations

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

```
Array(-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

```
val airportDelays = flightsParsed.map(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

```
val airportTotalDelay=airportDelays.reduceByKey((x,y) => x+y)
```

First, let's compute the sum

Average delay per airport

Option 1: reduceByKey

```
val airportTotalDelay=airportDelays.reduceByKey((x,y) => x+y)
```

This is a new RDD

reduceByKey is a transformation

Average delay per airport

Option 1: reduceByKey

```
airportTotalDelay=airportDelays.reduceByKey((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

```
val airportTotalDelay=airportDelays.reduceByKey((x,y) => x+y)
```

The result is a Pair RDD

Keys = Airports
Values = Sum of Delays

Average delay per airport

Option 1: reduceByKey

```
val airportTotalDelay=airportDelays.reduceByKey((x,y) => x+y)
```

reduceByKey effectively flattens the Pair RDD

Let's see this visually

```
val airportTotalDelay=airportDelays.reduceByKey((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

```
val airportTotalDelay=airportDelays.reduceByKey((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

```
val airportTotalDelay=airportDelays.reduceByKey((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

```
val airportTotalDelay=airportDelays.reduceByKey((x,y) => x+y)
```

P1

JFK	2
JFK	14
JFK	0

PPG	5
PPG	10
PPG	4

The given function is applied on each of these groups

```
val airportTotalDelay=airportDelays.reduceByKey((x,y) => x+y)
```

P1

JFK	2
JFK	14
JFK	0

x

y

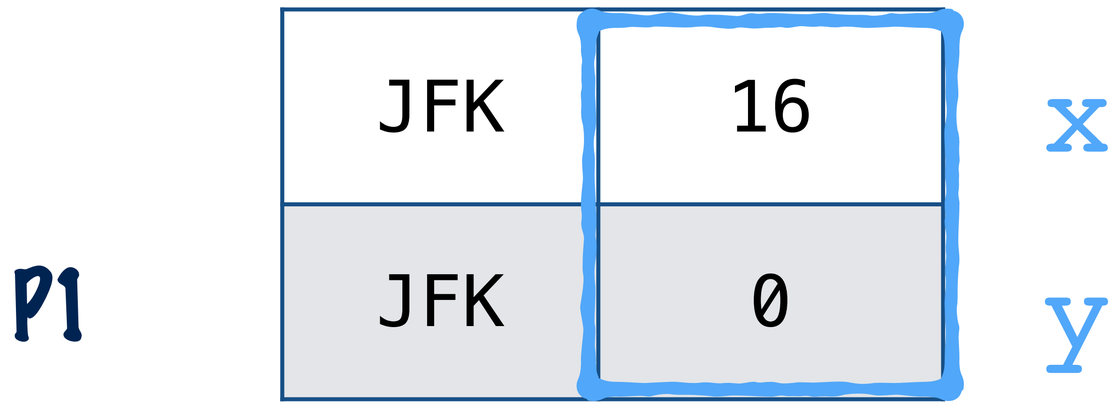
```
val airportTotalDelay=airportDelays.reduceByKey((x,y) => x+y)
```

P1

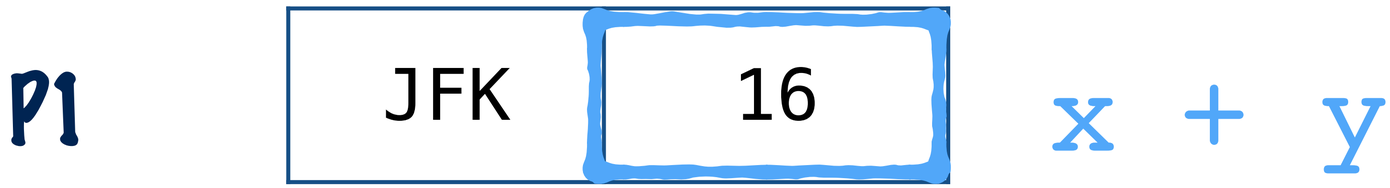
JFK	16
JFK	0

x + y

```
val airportTotalDelay=airportDelays.reduceByKey((x,y) => x+y)
```



```
val airportTotalDelay=airportDelays.reduceByKey((x,y) => x+y)
```



```
val airportTotalDelay=airportDelays.reduceByKey((x,y) => x+y)
```

P1

JFK	16

The same thing is done for each key

```
val airportTotalDelay=airportDelays.reduceByKey((x,y) => x+y)
```

P1

JFK	16
PPG	5

The same thing is done for each key

```
val airportTotalDelay=airportDelays.reduceByKey((x,y) => x+y)
```

P1

JFK	16
PPG	5

We have a set of Key value pairs from each node

```
val airportTotalDelay=airportDelays.reduceByKey((x,y) => x+y)
```

P1

JFK	16
PPG	5

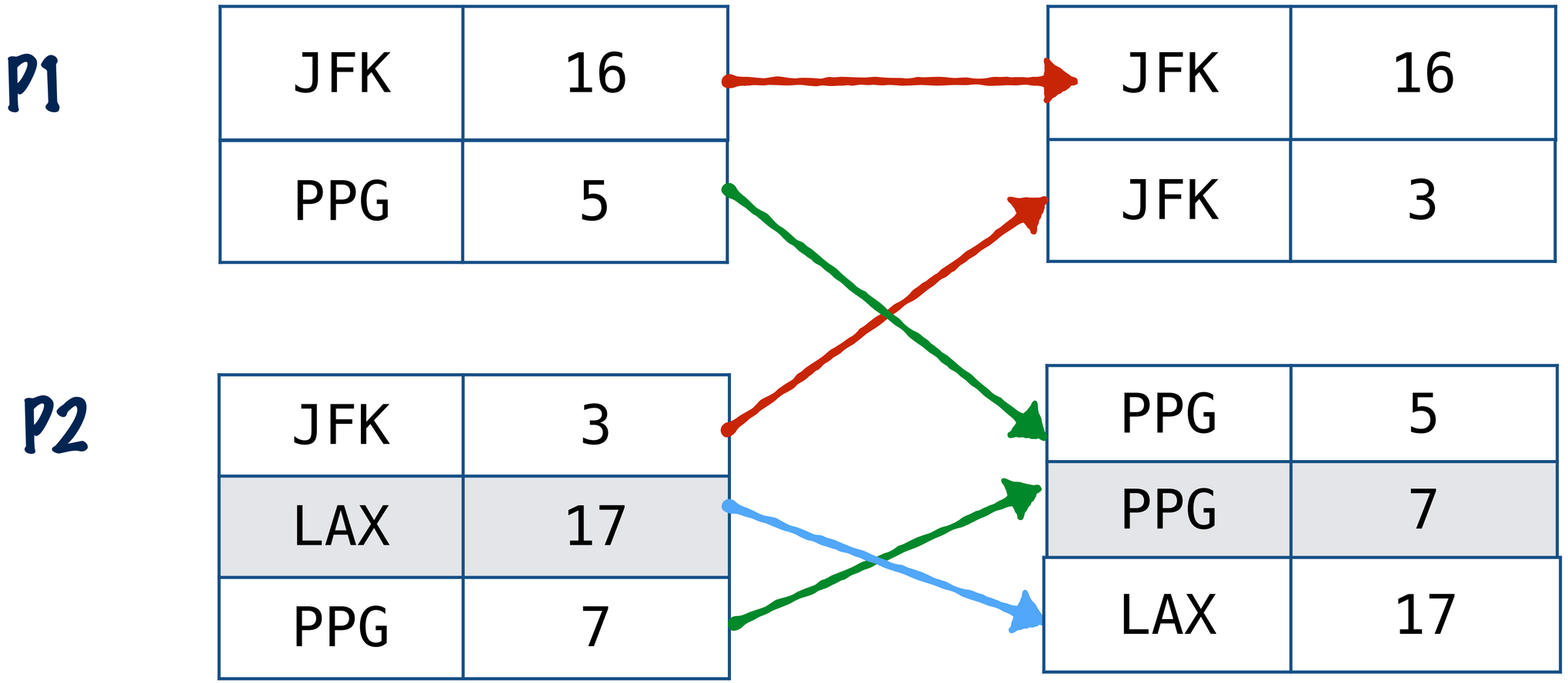
P2

JFK	3
LAX	17
PPG	7

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

```
val airportTotalDelay=airportDelays.reduceByKey((x,y) => x+y)
```

shuffle



```
val airportTotalDelay=airportDelays.reduceByKey((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

```
val airportTotalDelay=airportDelays.reduceByKey((x,y) => x+y)
```

P1

JFK	19

P2

PPG	12
LAX	17

On each partition, again the reduceByKey operation is applied

```
val airportTotalDelay=airportDelays.reduceByKey((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

```
val airportTotalDelay=airportDelays.reduceByKey((x,y) => x+y)
```

We can use the same idea to compute count

```
val airportCount=airportDelays.mapValues(x => 1).reduceByKey((x,y) => x+y)
```

```
val airportCount=airportDelays.mapValues(x => 1).reduceByKey((x,y) => x+y)
```

This is our RDD with
(origin airport, delay)

```
val airportCount=airportDelays.mapValues(x => 1).reduceByKey((x,y) => x+y)
```

(origin airport, delay)

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

```
val airportCount=airportDelays.mapValues(x => 1).reduceByKey((x,y) => x+y)
```

(origin airport, delay) ⟶ (origin airport, 1)

All the values are mapped to the constant 1

```
val airportCount=airportDelays.mapValues(x => 1).reduceByKey((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

```
val airportTotalDelay=airportDelays.reduceByKey((x,y) => x+y)
val airportCount=airportDelays.mapValues(x => 1).reduceByKey((x,y) => x+y)
```

The sum and count are in 2 separate RDDs

Average delay per airport

Option 1: reduceByKey

```
val airportTotalDelay=airportDelays.reduceByKey((x,y) => x+y)
val airportCount=airportDelays.mapValues(x => 1).reduceByKey((x,y) => x+y)
```

We can merge these using a join operation

Average delay per airport

Option 1: reduceByKey

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

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

Average delay per airport

Option 1: reduceByKey

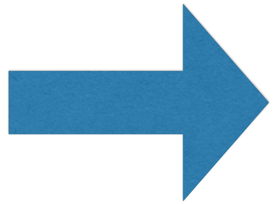
```
val airportSumCount=airportTotalDelay.join(airportCount)
```

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

```
val 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

```
val airportAvgDelay=airportSumCount.mapValues(x => x._1/x._2.toDouble)
```

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

```
val airportTotalDelay=airportDelays.reduceByKey((x,y) => x+y)
val airportCount=airportDelays.mapValues(x => 1).reduceByKey((x,y) => x+y)
val airportSumCount=airportTotalDelay.join(airportCount)
```

```
val airportAvgDelay=airportSumCount.mapValues(x => x._1/x._2.toDouble)
```

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

Average delay per airport

Option 1: reduceByKey

```
val airportTotalDelay=airportDelays.reduceByKey((x,y) => x+y)
val airportCount=airportDelays.mapValues(x => 1).reduceByKey((x,y) => x+y)
val 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

```
val airportTotalDelay=airportDelays.reduceByKey((x,y) => x+y)
val airportCount=airportDelays.mapValues(x => 1).reduceByKey((x,y) => x+y)
val 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

```
val airportSumCount2=airportDelays.combineByKey(
                                    value => (value,1),
                                    (acc: (Double,Int), value) =>  (acc._1 + value, acc._2+1),
                                    (acc1: (Double,Int), acc2: (Double,Int)) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

This is a new RDD

combineByKey is a transformation

Average delay per airport

Option 2: combineByKey

```
val airportSumCount2=airportDelays.combineByKey(
          value => (value,1),
          (acc: (Double,Int), value) =>  (acc._1 + value, acc._2+1),
          (acc1: (Double,Int), acc2: (Double,Int)) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

combineByKey is similar to aggregate

Average delay per airport

Option 2: combineByKey

```
val airportSumCount2=airportDelays.combineByKey(
                    value => (value,1),
                    (acc: (Double,Int), value) =>  (acc._1 + value, acc._2+1),
                    (acc1: (Double,Int), acc2: (Double,Int)) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

combineByKey requires 3 functions

Average delay per airport

Option 2: combineByKey

```
val airportSumCount2=airportDelays.combineByKey(
                                    value => (value,1),
                                    (acc: (Double,Int), value) =>  (acc._1 + value, ac
                                    (acc1: (Double,Int), acc2: (Double,Int)) => (acc1.
```

createCombiner Function

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

Average delay per airport

Option 2: combineByKey

```
airportDelays.combineByKey(
                        value => (value,1),
                        (acc: (Double,Int), value) =>  (acc._1 + value, acc._2+1),
                        (acc1: (Double,Int), acc2: (Double,Int)) => (acc1._1+acc2._1,acc1._2+a
```

merge Function

This specifies how values with the same key should be combined

within a partition

Average delay per airport

Option 2: combineByKey

```
ays.combineByKey(
              value => (value,1),
              (acc: (Double,Int), value) =>  (acc. 1 + value, acc. 2+1),
              (acc1: (Double,Int), acc2: (Double,Int)) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

mergeCombiners Function

This specifies how the results from each partition should be combined

Average delay per airport

Option 2: combineByKey

```
combineByKey(
        value => (value,1),
        (acc: (Double,Int), value) =>  (acc._1 + value, acc._2+1),
        (acc1: (Double,Int), acc2: (Double,Int)) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

createCombiner

merge

mergeCombiners

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

Average delay per airport

Option 2: combineByKey

```
combineByKey(
          value => (value,1),
          (acc: (Double,Int), value) =>  (acc._1 + value, acc._2+1),
          (acc1: (Double,Int), acc2: (Double,Int)) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

Let's see this visually

Average delay per airport

Option 2: combineByKey

```
combineByKey(
        value => (value,1),
        (acc: (Double,Int), value) =>  (acc._1 + value, acc._2+1),
        (acc1: (Double,Int), acc2: (Double,Int)) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

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(
        value => (value,1),
        (acc: (Double,Int), value) =>  (acc._1 + value, acc._2+1),
        (acc1: (Double,Int), acc2: (Double,Int)) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

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(
        value => (value,1),
        (acc: (Double,Int), value) =>  (acc._1 + value, acc._2+1),
        (acc1: (Double,Int), acc2: (Double,Int)) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

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(
        value => (value,1),
        (acc: (Double,Int), value) =>  (acc._1 + value, acc._2+1),
        (acc1: (Double,Int), acc2: (Double,Int)) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

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(
        value => (value,1),
        (acc: (Double,Int), value) =>  (acc._1 + value, acc._2+1),
        (acc1: (Double,Int), acc2: (Double,Int)) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

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(
        value => (value,1),
        (acc: (Double,Int), value) =>  (acc._1 + value, acc._2+1),
        (acc1: (Double,Int), acc2: (Double,Int)) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

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(
        value => (value,1),
        (acc: (Double,Int), value) =>  (acc._1 + value, acc._2+1),
        (acc1: (Double,Int), acc2: (Double,Int)) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

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(
        value => (value,1),
        (acc: (Double,Int), value) =>  (acc._1 + value, acc._2+1),
        (acc1: (Double,Int), acc2: (Double,Int)) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

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(
          value => (value,1),
          (acc: (Double,Int), value) =>  (acc._1 + value, acc._2+1),
          (acc1: (Double,Int), acc2: (Double,Int)) => (acc1._1+acc2._1,acc1._2+acc2._2)
```

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(
        value => (value,1),
        (acc: (Double,Int), value) =>  (acc._1 + value, acc._2+1),
        (acc1: (Double,Int), acc2: (Double,Int)) => (acc1._1+acc2._1,acc1._2+acc2._2)
```

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(
        value => (value,1),
        (acc: (Double,Int), value) =>  (acc._1 + value, acc._2+1),
        (acc1: (Double,Int), acc2: (Double,Int)) => (acc1._1+acc2._1,acc1._2+acc2._2)
```

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(
        value => (value,1),
        (acc: (Double,Int), value) =>  (acc._1 + value, acc._2+1),
        (acc1: (Double,Int), acc2: (Double,Int)) => (acc1._1+acc2._1,acc1._2+acc2._2)
```

P1

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

value

JFK, (14,2) → acc

PPG, (15,2)

Average delay per airport

Option 2: combineByKey

```
.combineByKey(
        value => (value,1),
        (acc: (Double,Int), value) =>  (acc._1 + value, acc._2+1),
        (acc1: (Double,Int), acc2: (Double,Int)) => (acc1._1+acc2._1,acc1._2+acc2._2)
```

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(
          value => (value,1),
          (acc: (Double,Int), value) =>  (acc._1 + value, acc._2+1),
          (acc1: (Double,Int), acc2: (Double,Int)) => (acc1._1+acc2._1,acc1._2+acc2._2)
```

P1

JFK, (14,3)

PPG, (19,3)

The same thing is done on each Partition

Average delay per airport

Option 2: combineByKey

```
.combineByKey(
        value => (value,1),
        (acc: (Double,Int), value) =>  (acc._1 + value, acc._2+1),
        (acc1: (Double,Int), acc2: (Double,Int)) => (acc1._1+acc2._1,acc1._2+acc2._2)
```

P1

JFK, (14,3)

PPG, (19,3)

P2

JFK, (3,3)

PPG, (7,1)

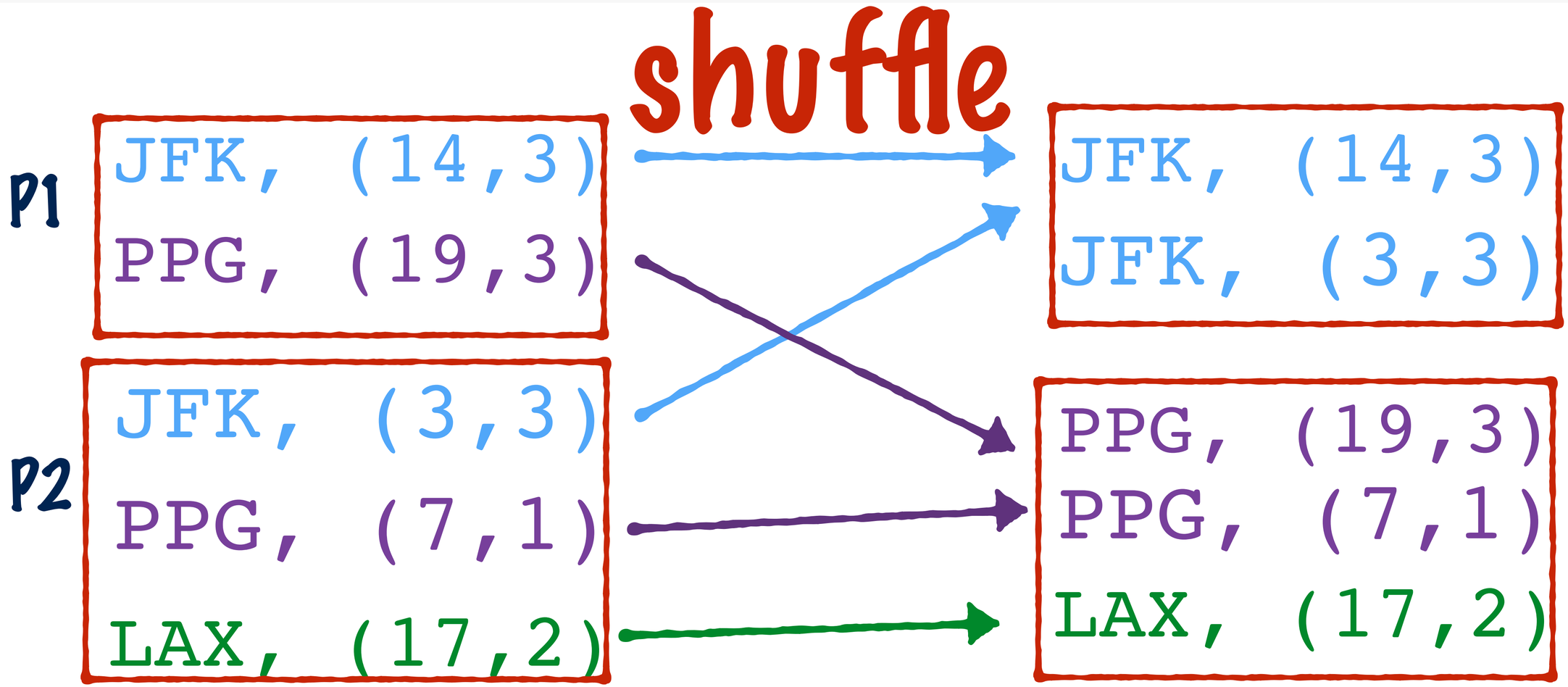
LAX, (17,2)

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

Average delay per airport

Option 2: combineByKey

```
.combineByKey(
        value => (value,1),
        (acc: (Double,Int), value) =>  (acc._1 + value, acc._2+1),
        (acc1: (Double,Int), acc2: (Double,Int)) => (acc1._1+acc2._1,acc1._2+acc2._2)
```



Average delay per airport

Option 2: combineByKey

```
.combineByKey(
        value => (value,1),
        (acc: (Double,Int), value) =>  (acc._1 + value, acc._2+1),
        (acc1: (Double,Int), acc2: (Double,Int)) => (acc1._1+acc2._1,acc1._2+acc2._2)
```

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

```
ombineByKey(
        value => (value,1),
        (acc: (Double,Int), value) =>  (acc. 1 + value, acc. 2+1),
        (acc1: (Double,Int), acc2: (Double,Int)) => (acc1._1+acc2._1,acc1._2+acc2._2)
```

P1

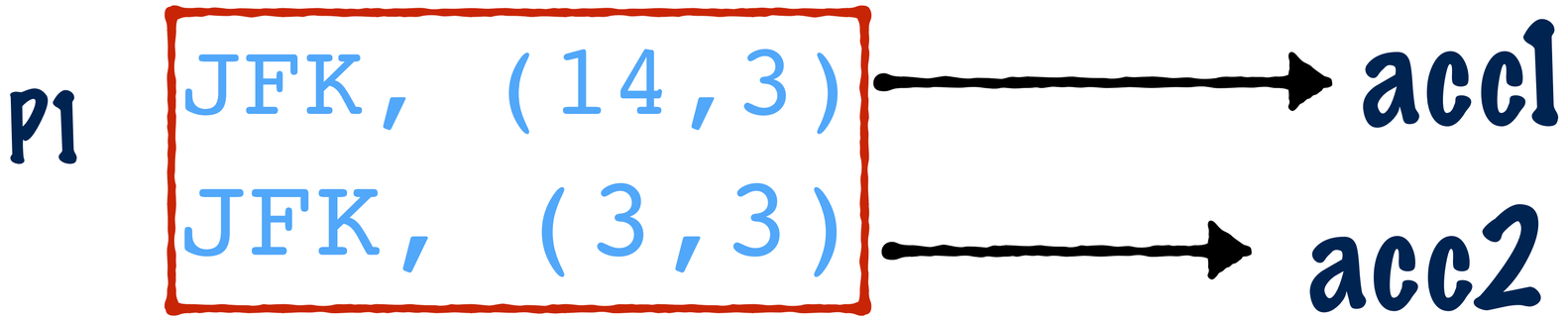
```
JFK, (14,3)
JFK, (3,3)
```

Values with the same key are combined using the mergeCombiners function

Average delay per airport

Option 2: combineByKey

```
ineByKey(
      value => (value,1),
      (acc: (Double,Int), value) =>  (acc._1 + value, acc._2+1),
      (acc1: (Double,Int), acc2: (Double,Int)) => (acc1._1+acc2._1,acc1._2+acc2._2))
```



Average delay per airport

Option 2: combineByKey

```
ineByKey(
      value => (value,1),
      (acc: (Double,Int), value) =>  (acc._1 + value, acc._2+1),
      (acc1: (Double,Int), acc2: (Double,Int)) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

P1 JFK, (17,6)

Similarly, each partition is processed

Average delay per airport

Option 2: combineByKey

```
val airportSumCount2=airportDelays.combineByKey(
                                    value => (value,1),
                                    (acc: (Double,Int), value
                                    (acc1: (Double,Int), acc2
```

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

```
val airportAvgDelay
```

We already have the average delay per airport

The top 10 airports based on delay

```
val airportAvgDelay
```

It's a pair RDD with

keys=Airports

Values=Avg Delay per Airport

The top 10 airports based on delay

```
val airportAvgDelay
```

We want to sort this RDD

In descending order of value

The top 10 airports based on delay

```
airportAvgDelay.sortBy(-_._2)
```

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(_._2)
```

This is a placeholder representing each key value pair

The top 10 airports based on delay

```
airportAvgDelay.sortBy(-_._2)
```

This is the avg delay per airport

ie. the value in the key value pair

The top 10 airports based on delay

```
airportAvgDelay.sortBy(-_._2)
```

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

The top 10 airports based on delay

```
airportAvgDelay.sortBy(-_._2).take(10)
```

```
Array((PPG,56.25), (EGE,32.0), (OTH,24.533333333333335), (LAR,18.892857142857142), (RDD,18.55294117647059), (MTJ,18.3
63636363636363), (PUB,17.54), (EWR,16.478549005929544), (CIC,15.931034482758621), (RST,15.6993006993007))
```

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

The top 10 airports based on delay

```
Array((PPG,56.25), (EGE,32.0), (OTH,24.533333333333335), (LAR,18.892857142857142)
636363636363), (PUB,17.54), (EWR,16.47854900592954), (CIC,15.931034482758621),
```

These are the airport Codes

Which airports do these refer to?

The top 10 airports based on delay

```
Array((PPG,56.25), (EGE,32.0), (OTH,24.533333333333335), (LAR,18.892857142857142)
636363636363), (PUB,17.54), (EWR,16.47854900592954), (CIC,15.931034482758621),
```

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

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

Each row has

airport Code, airport Description

Looking up Airport Descriptions

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

```
def notHeader(row: String): Boolean = {
    !row.contains("Description")
    }
```

Filter out the header row

Looking up Airport Descriptions

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

```
def notHeader(row: String): Boolean = {
    !row.contains("Description")
    }
```

This function returns a Boolean for each row

Looking up Airport Descriptions

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

```
def notHeader(row: String): Boolean = {
    !row.contains("Description")
    }
```

It returns True if the row does not contain the word "Description"

Looking up Airport Descriptions

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

```
def parseLookup(row: String): (String,String)={
 val x = row.replace("\"","").split(',')
 (x(0),x(1))
}
```

Parse the Airport code and Description into a tuple

Looking up Airport Descriptions

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

```
def parseLookup(row: String): (String,String)={
 val x = row.replace("\"","").split(',')
 (x(0),x(1))
}
```

Returns a tuple for each row

Looking up Airport Descriptions

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

```
def parseLookup(row: String): (String,String)={
 val x = row.replace("\"","").split(',')
 (x(0),x(1))
}
```

Remove " characters and split the row into an array

Looking up Airport Descriptions

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

```
def parseLookup(row: String): (String,String)={
 val x = row.replace("\"","").split(',')
 (x(0),x(1))
}
```

Return a tuple representing
Airport code and description

Looking up Airport Descriptions

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

airports is a Pair RDD

Each record is a tuple

(Airport Code, Description)

Looking up Airport Descriptions

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

We have 3 options on how to use this RDD

lookup action

map

broadcast

Looking up Airport Descriptions lookup action

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

Pair RDDs have an action called lookup

Looking up Airport Descriptions lookup action

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

We can use lookup to find the description for one airport code at a time

Looking up Airport Descriptions

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

```
WrappedArray(Pago Pago)
```

We can use lookup to find the description for one airport code at a time

Looking up Airport Descriptions lookup action

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

```
WrappedArray(Pago Pago)
```

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

Looking up Airport Descriptions lookup action

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

```
WrappedArray(Pago Pago)
```

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')
```

```
WrappedArray(Pago Pago)
```

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

```
val airportLookup=airports.collectAsMap
```

collectAsMap is an action

Looking up Airport Descriptions

dictionary

```
val airportLookup=airports.collectAsMap
```

```
airportLookup("CLI")
```

```
Clintonville
```

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

Looking up Airport Descriptions

dictionary

```
val airportLookup=airports.collectAsMap
```

```
airportAvgDelay.map(x=>(airportLookup(x._1),x._2))
```

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(x=>(airportLookup(x._1),x._2))
```

(Airport Code, Avg Delay)

It uses the airportLookup dict to do so

Looking up Airport Descriptions

dictionary

```
airportAvgDelay.map(x=>(airportLookup(x._1),x._2))
```

(Airport Code, Avg Delay)

(Airport Desc, Avg Delay)

Looking up Airport Descriptions

dictionary

```
airportAvgDelay.map(x=>(airportLookup(x._1),x._2))
```

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(x=>(airportLookup(x._1),x._2))
```

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(x=>(airportLookup(x._1),x._2))
```

This is a property of
closure functions

Closure functions are
pretty complicated

Looking up Airport Descriptions

dictionary

```
airportAvgDelay.map(x=>(airportLookup(x._1),x._2))
```

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(x=>(airportLookup(x._1),x._2))
```

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(x=>(airportLookup(x._1),x._2))
```

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

Looking up Airport Descriptions

dictionary

```
airportAvgDelay.map(x=>(airportLookup(x._1),x._2))
```

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

Looking up Airport Descriptions

dictionary

```
airportAvgDelay.map(x=>(airportLookup(x._1),x._2))
```

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

Looking up Airport Descriptions

dictionary

```
airportAvgDelay.map(x=>(airportLookup(x._1),x._2))
```

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

Looking up Airport Descriptions

dictionary

```
airportAvgDelay.map(x=>(airportLookup(x._1),x._2))
```

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

Looking up Airport Descriptions 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

```
val 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

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

```
airportAvgDelay.map(x => (airportBC.value(x._1),x._2)).
```

This is pretty similar to how we used the dictionary

Looking up Airport Descriptions broadcast

```
airportAvgDelay.map(x => (airportBC.value(x._1),x._2)).
```

This is pretty similar to how we used the dictionary

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