Exploring Airline delays data with Spark Shell

Part 2

Let's go back to our Flight related data from USDoT

Let's go back to our Flight related data from USDoT

Here are a few things we would want to do

1. Parse the rows in the csv files
2. Compute the average distance travelled by a flight
3. Compute the % of flights which had delays
4. Compute the average delay
5. Compute a frequency distribution of delays

Let's go back to our Flight related data from USDoT

We'll do the following

1. Parse the rows in the csv files
2. Compute the average distance travelled by a flight
3. Compute the % of flights which had delays
4. Compute the average delay
5. Compute a frequency distribution of delays

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

Recap

```
// Data location
val airlinesPath="hdfs:///user/swethakolalapudi/flightDelaysData/airlines.csv"
val airportsPath="hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv"
val flightsPath="hdfs:///user/swethakolalapudi/flightDelaysData/flights.csv"
```

airlines.csv

airports.csv

flights.csv

We read these files from the local file system or from HDFS

```
val flights=sc.textFile(flightsPath)
```

This loads the flights data from file to an RDD

```
flights
```

```
hdfs:///user/swethakolalapudi/flightDelaysData/flights.csv MapPartitionsRDD[5] at textFile at <console>:22
```

```
val flights=sc.textFile(flightsPath)
```

This loads the flights data from file to an RDD

```
ata/flights.csv MapPartitionsRDD[5] at textFile at <consc
```

Let's get a quick sense of the data

```
// The total number of records
flights.count()
```

```
476881
```

```
// The first row
flights.first()
```

```
2014-04-01,19805,1,JFK,LAX,0854,-6.00,1217,2.00,355.00,2475.00
```

These are examples of Actions

Each row here has

```
2014-04-01,19805,1,JFK,LAX,0854,-6.00,1217,2.00,355.00,2475.00
```

All of this is contained in a single string

- Flight date
- Airline code
- Flightnum
- Source Airport
- Destination Airport
- Departure time
- Departure delay
- Arrival time
- Arrival delay
- Airtime
- Distance

Each row here has

```
2014-04-01,19805,1,JFK,LAX,0854,-6.00,1217,2.00,355.00,2475.00
```

Flight date

Airline code

Flightnum

Source Airport

Destination Airport

Departure time

Departure delay

Arrival time

Arrival delay

Airtime

Distance

We can take this RDD and parse each row into an Array[String]

We can take this RDD and parse each row into an Array[String]

```
flights.map(_.split(","))
```

The map operation is a Transformation

```
flights.map(_.split(","))
```

map will take a function and apply it to every record in the RDD

```
flights.map(_.split(","))
```

In this case, it will parse each record into an array

Flight date
Airline code
Flightnum
Source Airport
Destination Airport
Departure time
Departure delay
Arrival time
Arrival delay
Airtime
Distance

```
flights.map(_.split(","))
```

_ acts as a placeholder for each record

Flight date
Airline code
Flightnum
Source Airport
Departure time
e delay
Arrival time
Arrival delay
Airtime
Distance

```
flights.map(_.split(","))
```

At the end of this operation, we have an RDD of Arrays, but this could be processed further

Flight date
Airline code
Flightnum
Source Airport
Destination Airport
Departure time
Departure delay
Arrival time
Arrival delay
Airtime
Distance

We can set things up to reference these columns by name

Flight date

Airline code

Flightnum

Source Airport

Destination Airport

Departure time

Departure delay

Arrival time

Arrival delay

Airtime

Distance

We can convert these fields to relevant data types from string

Flight date

Airline code

Flightnum

Source Airport

Destination Airport

Departure time

Departure delay

Arrival time

Arrival delay

Airtime

Distance

We'll set up a class to represent 1 record

```
case class Flight(date: LocalDate,
                  airline: String ,
                  flightnum: String,
                  origin: String ,
                  dest: String ,
                  dep: LocalTime,
                  dep_delay: Double,
                  arv: LocalTime,
                  arv_delay: Double ,
                  airtime: Double ,
                  distance: Double
                   )
```

```
case class Flight(date: LocalDate,
                  airline: String ,
                  flightnum: String,
                  origin: String ,
                  dest: String ,
                  dep: LocalTime,
                  dep_delay: Double,
                  arv: LocalTime,
                  arv_delay: Double ,
                  airtime: Double ,
                  distance: Double
                   )
```

A case class is a special type of class in Scala

```
case class Flight(date: LocalDate,
                  airline: String ,
                  flightnum: String,
                  origin: String ,
                  dest: String ,
                  dep: LocalTime,
                  dep_delay: Double,
                  arv: LocalTime,
                  arv_delay: Double ,
                  airtime: Double ,
                  distance: Double
                   )
```

Case classes are used to setup plain objects that are just used to hold data with a certain structure

The parameters in the constructor automatically become public member variables

```
case class Flight(date: LocalDate,
                  airline: String ,
                  flightnum: String,
                  origin: String ,
                  dest: String ,
                  dep: LocalTime,
                  dep_delay: Double,
                  arv: LocalTime,
                  arv_delay: Double ,
                  airtime: Double ,
                  distance: Double
                 )
```

```
case class Flight(date: LocalDate,
                  airline: String ,
                  flightnum: String,
                  origin: String ,
                  dest: String ,
                  dep: LocalTime,
                  dep_delay: Double,
                  arv: LocalTime,
                  arv_delay: Double ,
                  airtime: Double ,
                  distance: Double
                   )
```

An instance of this class can be set up without using new

```
case class Flight(date: LocalDate,
                  airline: String ,
                  flightnum: String,
                  origin: String ,
                  dest: String ,
                  dep: LocalTime,
                  dep_delay: Double,
                  arv: LocalTime,
                  arv_delay: Double ,
                  airtime: Double ,
                  distance: Double
                   )
```

The constructor can be used like a function

```
Flight(date,airline,flightnum,origin,dest,dep,
       dep_delay,arv,arv_delay,airtime,distance)
```

new is not used

Scala automatically sets up equals and toString methods for a case class

```
case class Flight(date: LocalDate,
                  airline: String ,
                  flightnum: String,
                  origin: String ,
                  dest: String ,
                  dep: LocalTime,
                  dep_delay: Double,
                  arv: LocalTime,
                  arv_delay: Double ,
                  airtime: Double ,
                  distance: Double
                   )
```

```
case class Flight(date: LocalDate,
                  airline: String ,
                  flightnum: String,
                  origin: String ,
                  dest: String ,
                  dep: LocalTime,
                  dep_delay: Double,
                  arv: LocalTime,
                  arv_delay: Double ,
                  airtime: Double ,
                  distance: Double
                   )
```

We'll write a function to convert each String record in the flights RDD to this class

Here's the code that will do all of this

```
def parse(row: String): Flight={

  val fields = row.split(",")
  val datePattern = DateTimeFormat.forPattern("YYYY-mm-dd")
  val timePattern = DateTimeFormat.forPattern("HHmm")

  val date: LocalDate = datePattern.parseDateTime(fields(0)).toLocalDate()
  val airline: String = fields(1)
  val flightnum: String = fields(
  val origin: String = fields(3)
  val dest: String = fields(4)
  val dep: LocalTime = timePatter
  val dep_delay: Double = fields(
  val arv: LocalTime = timePatter
  val arv_delay: Double = fields(
  val airtime: Double = fields(9)
  val distance: Double = fields(1

  Flight(date,airline,flightnum,o
         dep_delay,arv,arv_delay,

    }
```

This function will parse the String representing each row, and return a Flight object

```
def parse(row: String): Flight={

  val fields = row.split(",")
  val datePattern = DateTimeFormat.forPattern("YYYY-mm-dd")
  val timePattern = DateTimeFormat.forPattern("HHmm")

  val date: LocalDate = datePattern.parseDateTime(fields(0)).toLocalDate()
  val airline: String = fields(1)
  val flightnum: String = fields(2)
  val origin: String = fields(3)
  val dest: String = fields(4)
  val dep: LocalTime = timePattern.pars
  val dep_delay: Double = fields(6).toD
  val arv: LocalTime = timePattern.pars
  val arv_delay: Double = fields(8).toD
  val airtime: Double = fields(9).toDou
  val distance: Double = fields(10).toD

  Flight(date,airline,flightnum,origin,
         dep_delay,arv,arv_delay,airtim

    }
```

We just need to use this function to process each row in the dataset

```
def parse(row: String): Flight={

  val fields = row.split(",")
  val datePattern = DateTimeFormat.forPattern("YYYY-mm-dd")
  val timePattern = DateTimeFormat.forPattern("HHmm")

  val date: LocalDate = datePattern.parseDateTime(fields(0)).toLocalDate()
  val airline: String = fields(1)
  val flightnum: String = fields(2)
  val origin: String = fields(3)
  val dest: String = fields(4)
  val dep: LocalTime = timePattern.parseD
  val dep_delay: Double = fields(6).toDou
  val arv: LocalTime = timePattern.parseD
  val arv_delay: Double = fields(8).toDou
  val airtime: Double = fields(9).toDoubl
  val distance: Double = fields(10).toDou

  Flight(date,airline,flightnum,origin,dest,dep,
         dep_delay,arv,arv_delay,airtime,distance)

    }
```

That's exactly what we have the **map operation** for!

```
def parse(row: String): Flight={

  val fields = row.split(",")
  val datePattern = DateTimeFormat.forPattern("YYYY-mm-dd")
  val timePattern = DateTimeFormat.forPattern("HHmm")

  val date: LocalDate = datePattern.parseDateTime(fields(0)).toLocalDate()
  val airline: String = fields(1)
  val flightnum: String = fields(2)
  val origin: String = fields(3)
  val dest: String = fields(4)
  val dep: LocalTime = timePattern.parseDateTime(fields(5)).toLocalTime()
  val dep_delay: Double = fields(6).
  val arv: LocalTime = timePattern.p
  val arv_delay: Double = fields(8).
  val airtime: Double = fields(9).to
  val distance: Double = fields(10).

  Flight(date,airline,flightnum,orig
         dep_delay,arv,arv_delay,air

    }
```

First, split the String into an Array

```
def parse(row: String): Flight={

  val fields = row.split(",")
  val datePattern = DateTimeFormat.forPattern("YYYY-mm-dd")
  val timePattern = DateTimeFormat.forPattern("HHmm")

  val date: LocalDate = datePattern.parseDateTime(fields(0)).toLocalDate()
  val airline: String = fields(1)
  val flightnum: String = fields(2)
  val origin: String = fields(3)
  val dest: String = fields(4)
  val dep: LocalTime = timePattern.parseDateTime(fields(5)).toLocalTime()
  val dep_delay: Double = fields(6).toDouble
  val arv: LocalTime = timePattern.parseDateTime(fields(7)).toLocalTime()
  val arv_delay: Double = fields(8).toD
  val airtime: Double = fields(9).toDoub
  val distance: Double = fields(10).toD

  Flight(date,airline,flightnum,origin,
         dep_delay,arv,arv_delay,airtime

    }
```

Convert each field to the relevant datatype

```
def parse(row: String): Flight={

  val fields = row.split(",")
  val datePattern = DateTimeFormat.forPattern("YYYY-mm-dd")
  val timePattern = DateTimeFormat.forPattern("HHmm")

  val date: LocalDate = datePattern.parseDateTime(fields(0)).toLocalDate()
  val airline: String = fields(1)
  val flightnum: String = fields(2)
  val origin: String = fields(3)
  val dest: String = fields(4)
  val dep: LocalTime = timePattern.parseDateTime(fields(5)).toLocalTime()
  val dep_delay: Double = fields(6).toDouble
  val arv: LocalTime = timePattern.parseDateTime(fields(7)).toLocalTime()
  val arv_delay: Double = fields(8).toD
  val airtime: Double = fields(9).toDoub
  val distance: Double = fields(10).toD

  Flight(date,airline,flightnum,origin,
         dep_delay,arv,arv_delay,airtime

    }
```

The Flight date is converted to an instance of LocalDate

```
def parse(row: String): Flight={

  val fields = row.split(",")
  val datePattern = DateTimeFormat.forPattern("YYYY-mm-dd")
  val timePattern = DateTimeFormat.forPattern("HHmm")

  val date: LocalDate = datePattern.parseDateTime(fields(0)).toLocalDate()
  val airline: String = fields(1)
  val flightnum: String = fields(2)
  val origin: String = fields(3)
  val dest: String = fields(4)
  val dep: LocalTime = timePattern.parseDateTime(fields(5)).toLocalTime()
  val dep_delay: Double = fields(6).toDouble
  val arv: LocalTime = timePattern.parseDateTime(fields(7)).toLocalTime()
  val arv_delay: Double = fields(8).toD
  val airtime: Double = fields(9).toDoub
  val distance: Double = fields(10).toD

  Flight(date,airline,flightnum,origin,
         dep_delay,arv,arv_delay,airtime

    }
```

LocalDate is a class from the joda time library

```scala
def parse(row: String): Flight={

  val fields = row.split(",")
  val datePattern = DateTimeFormat.forPattern("YYYY-mm-dd")
  val timePattern = DateTimeFormat.forPattern("HHmm")

  val date: LocalDate = datePattern.parseDateTime(fields(0)).toLocalDate()
  val airline: String = fields(1)
  val flightnum: String = fields(2)
  val origin: String = fields(3)
  val dest: String = fields(4)
  val dep: LocalTime = timePattern.parseDateTime(fields(5)).toLocalTime()
  val dep_delay: Double = fields(6).toDouble
  val arv: LocalTime = timePattern.parseDateTime(fields(7)).toLocalTime()
  val arv_delay: Double = fields(8).toD
  val airtime: Double = fields(9).toDoub
  val distance: Double = fields(10).toD

  Flight(date,airline,flightnum,origin,
         dep_delay,arv,arv_delay,airtime

    }
```

LocalTime is used to represent time of day

```
def parse(row: String): Flight={

  val fields = row.split(",")
  val datePattern = DateTimeFormat.forPattern("YYYY-mm-dd")
  val timePattern = DateTimeFormat.forPattern("HHmm")

  val date: LocalDate = datePattern.parseDateTime(fields(0)).toLocalDate()
  val airline: String = fields(1)
  val flightnum: String = fields(2)
  val origin: String = fields(3)
  val dest: String = fields(4)
  val dep: LocalTime = timePattern.parseDateTime(fields(5)).toLocalTime()
  val dep_delay: Double = fields(6).toDouble
  val arv: LocalTime = timePattern.parseDateTime(fields(7)).toLocalTime()
  val arv_delay: Double = fields(8).toD
  val airtime: Double = fields(9).toDoub
  val distance: Double = fields(10).toD

  Flight(date,airline,flightnum,origin,
         dep_delay,arv,arv_delay,airtime

    }
```

Airtime, delay and distance are represented using Doubles

```
def parse(row: String): Flight={

  val fields = row.split(",")
  val datePattern = DateTimeFormat.forPa
  val timePattern = DateTimeFormat.forPa

  val date: LocalDate = datePattern.pars
  val airline: String = fields(1)
  val flightnum: String = fields(2)
  val origin: String = fields(3)
  val dest: String = fields(4)
  val dep: LocalTime = timePattern.parse
  val dep_delay: Double = fields(6).toDo
  val arv: LocalTime = timePattern.parse
  val arv_delay: Double = fields(8).toDo
  val airtime: Double = fields(9).toDouble
  val distance: Double = fields(10).toDouble

  Flight(date,airline,flightnum,origin,dest,dep,
         dep_delay,arv,arv_delay,airtime,distance)

    }
```

Airtime, delay and distance are represented using Doubles

We use this function in a map operation

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

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

Get an RDD with a Flight object as every record

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

```
// Let's take a look at the data in the Parsed RDD
flightsParsed.first()
```

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

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

```
// Let's take a look at the data in the Parsed RDD
flightsParsed.first()
```

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

Each record is now represented by a Flight object

```
// Let's take a look at the data in the Parsed RDD
flightsParsed.first()
```

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

We can access the values in the Flight object using the field name

```
flightsParsed.map(_.distance)
```

This will create an RDD which only has the distance field

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

Let's parse what happened here

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

We passed a function to map

This function has to be applied on each record in the RDD

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

The RDD is distributed across different nodes in the cluster

A copy of the function will be sent to each of these nodes

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

The function uses the class definition for Flight

```
case class Flight(date: LocalDate,
                  airline: String ,
                  flightnum: String,
                  origin: String ,
                  dest: String ,
                  dep: LocalTime,
                  dep_delay: Double,
                  arv: LocalTime,
                  arv_delay: Double ,
                  airtime: Double ,
                  distance: Double
                   )
```

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

The function carries this definition along with it to all the nodes

```
case class Flight(date: LocalDate,
                  airline: String ,
                  flightnum: String,
                  origin: String ,
                  dest: String ,
                  dep: LocalTime,
                  dep_delay: Double,
                  arv: LocalTime,
                  arv_delay: Double ,
                  airtime: Double ,
                  distance: Double
                   )
```

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

Such functions are called
closure functions

Scala supports the use of
closure functions

Scala supports the use of closure functions

We won't go further into closures here

Scala supports the use of closure functions

Just know that closures are what makes working with Spark so cool!

Scala supports the use of closure functions

You can define functions with complex behavior in Scala/Python and Spark takes care of making sure they work across the cluster

Scala supports the use of closure functions

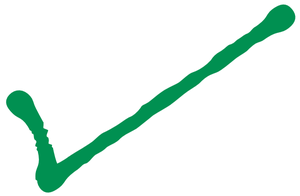
All you need to do is use the map operation!

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

Now we are done with parsing the rows, we can play with this dataset

Let's go back to our Flight related data from USDoT

We'll do the following

1. Parse the rows in the csv files 

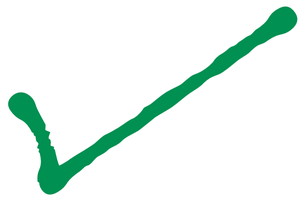
2. Compute the average distance travelled by a flight

3. Compute the % of flights which had delays

4. Compute the average delay

Let's go back to our Flight related data from USDoT

We'll do the following

1. Parse the rows in the csv files 
2. Compute the average distance travelled by a flight
3. Compute the % of flights which had delays
4. Compute the average delay

The average distance travelled by a flight

Let's start by computing the total distance travelled by all flights

```
val totalDistance=flightsParsed.map(_.distance).reduce((x,y) => x+y)
```

This extracts the distance field

The average distance travelled by a flight

```
val totalDistance=flightsParsed.map(_.distance).reduce((x,y) => x+y)
```

_ acts as a placeholder representing one object in the flightsParsed RDD

The average distance travelled by a flight

```
val totalDistance=flightsParsed.map(_.distance).reduce((x,y) => x+y)
```

This is a more concise way of representing

```
x => x.distance
```

The average distance travelled by a flight

```
val totalDistance=flightsParsed.map(_.distance).reduce((x,y) => x+y)
```

```
x => x.distance
```

Since each record in flightsParsed is a Flight object, it has a public distance attribute that can be accessed

The average distance travelled by a flight

```
val totalDistance=flightsParsed.map(_.distance).reduce((x,y) => x+y)
```

```
x => x.distance
```

The map takes each Flight object and returns the distance attribute of that object

The average distance travelled by a flight

```
val totalDistance=flightsParsed.map(_.distance).reduce((x,y) => x+y)
```

This will sum up all the values in the field

The average distance travelled by a flight

```
val totalDistance=flightsParsed.map(_.distance).reduce((x,y) => x+y)
```

reduce is an Action

The average distance travelled by a flight

```
val totalDistance=flightsParsed.map(_.distance).reduce((x,y) => x+y)
```

It will combine all the elements of the RDD in a specified way

The average distance travelled by a flight

```
val totalDistance=flightsParsed.map(_.distance).reduce((x,y) => x+y)
```

reduce takes a function that acts on two elements and returns an object of the same type

The average distance travelled by a flight

```
val totalDistance=flightsParsed.map(_.distance).reduce((x,y) => x+y)
```

This function will be iteratively applied on the elements of the RDD

Let's see this visually

```
val totalDistance=flightsParsed.map(_.distance).reduce((x,y) => x+y)
```

This is the distance RDD, partitioned among 3 nodes

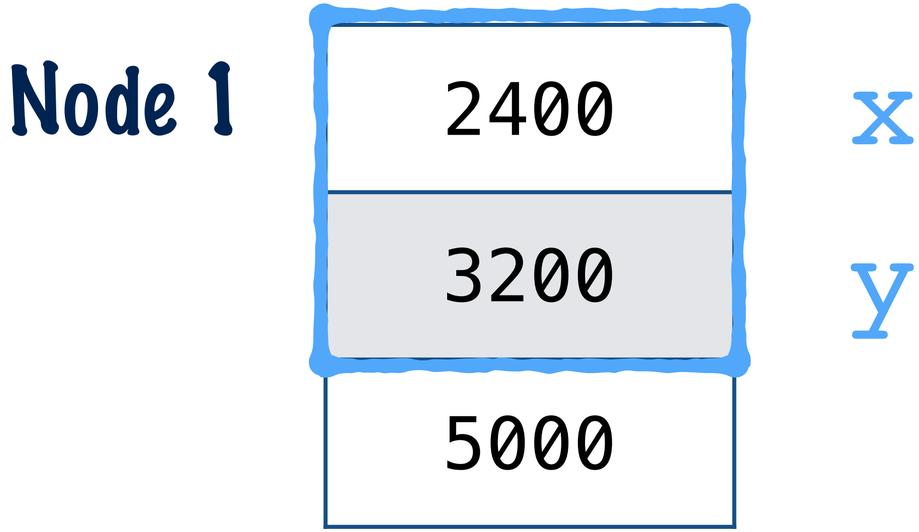
Node 1
2400
3200
5000

Node 2
2230
5400
4900

Node 3
4300
3600
2100

First, the reduce operation is performed on each node

```
val totalDistance=flightsParsed.map(_.distance).reduce((x,y) => x+y)
```

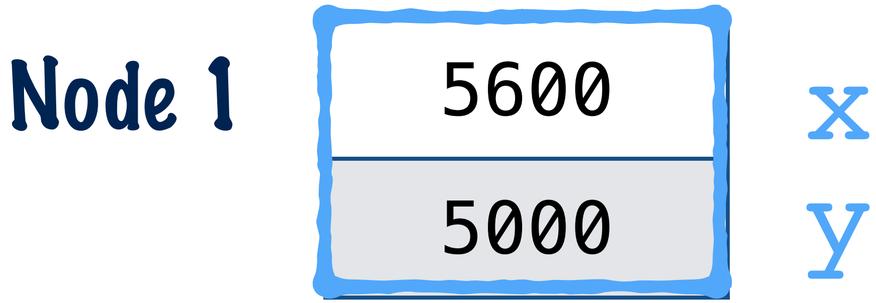


The function will start with the first 2 elements

```
val totalDistance=flightsParsed.map(_.distance).reduce((x,y) => x+y)
```

This will continue until all elements on the node have been added

```
val totalDistance=flightsParsed.map(_.distance).reduce((x,y) => x+y)
```



This will continue until all elements on the node have been added

```
val totalDistance=flightsParsed.map(_.distance).reduce((x,y) => x+y)
```

Node 1 | 10600 | x + y

The same thing is done in parallel on all the nodes

```
val totalDistance=flightsParsed.map(_.distance).reduce((x,y) => x+y)
```

Node 1 | 10600

Node 2 | 12530

Node 3 | 10000

The same thing is done in parallel on all the nodes

```
val totalDistance=flightsParsed.map(_.distance).reduce((x,y) => x+y)
```

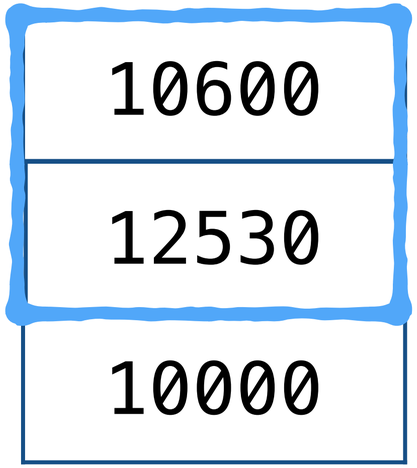
Node 1 | 10600

Node 2 | 12530

Node 3 | 10000

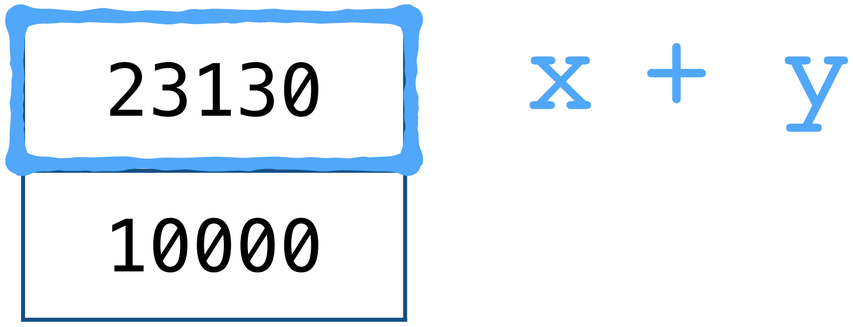
All the results from each node are brought over to a single node

```
val totalDistance=flightsParsed.map(_.distance).reduce((x,y) => x+y)
```



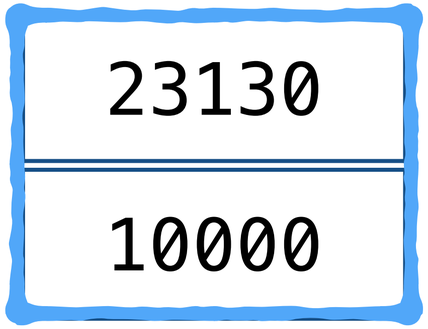
Again, reduce is applied iteratively on these results

```
val totalDistance=flightsParsed.map(_.distance).reduce((x,y) => x+y)
```



Again, reduce is applied iteratively on these results

```
val totalDistance=flightsParsed.map(_.distance).reduce((x,y) => x+y)
```



Again, reduce is applied iteratively on these results

```
val totalDistance=flightsParsed.map(_.distance).reduce((x,y) => x+y)
```

33130

x +y

You can give any function to reduce

as long it returns an object of the same type as the RDD elements

The average distance travelled by a flight

```
val totalDistance=flightsParsed.map(_.distance).reduce((x,y) => x+y)
```

We need to divide this by the total number of flights

```
val avgDistance=totalDistance/flightsParsed.count()
```

```
println(avgDistance)
```

```
794.858501387
```

Let's go back to our Flight related data from USDoT

We'll do the following

1. Parse the rows in the csv files ✓
2. Compute the average distance travelled by a flight ✓
3. Compute the % of flights which had delays
4. Compute the average delay

Let's go back to our Flight related data from USDoT

We'll do the following

1. Parse the rows in the csv files ✓
2. Compute the average distance travelled by a flight ✓
3. Compute the % of flights which had delays
4. Compute the average delay

% of flights which had delays

We'll start by counting the number of flights with delays

```
flightsParsed.filter(_.dep_delay>0).count().toDouble
```

% of flights which had delays

We'll start by counting the number of flights with delays

```
flightsParsed.filter(_.dep_delay>0).count().toDouble
```

This will filter out only those records where there was a delay at departure

% of flights which had delays

```
flightsParsed.filter(_.dep_delay>0).count().toDouble
```

This gives us the count of the number of flights with a delayed departure

% of flights which had delays

```
flightsParsed.filter(_.dep_delay>0).count().toDouble
```

Count is an int, convert to Double

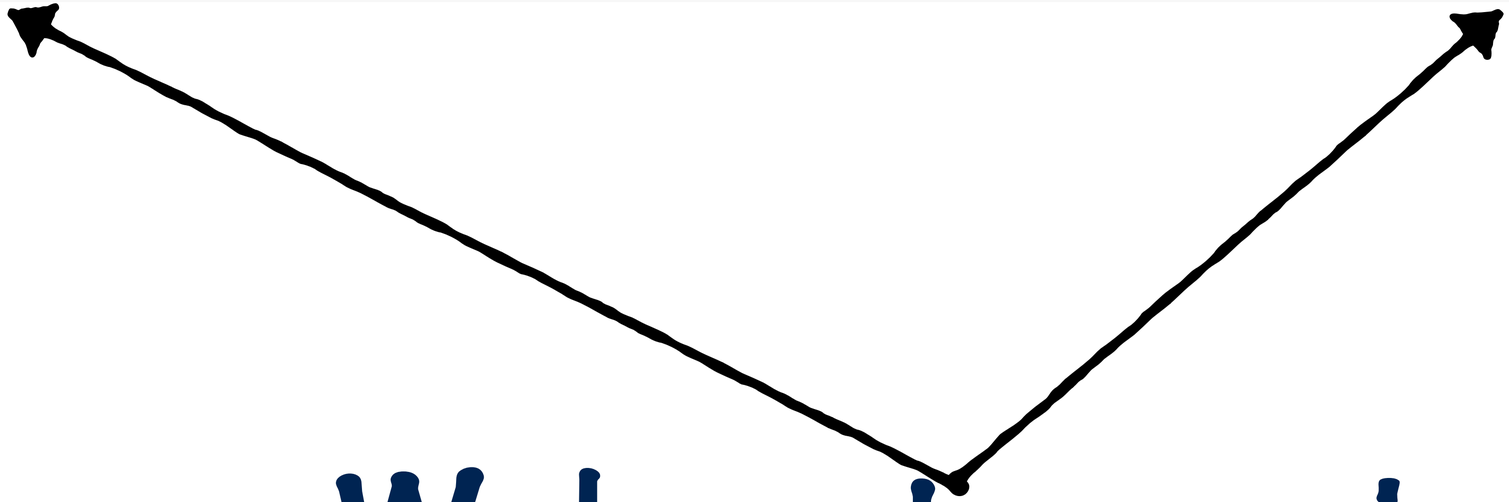
% of flights which had delays

```
flightsParsed.filter(_.dep_delay>0).count().toDouble/flightsParsed.count().toDouble
```

We just need to divide by the total flight count

% of flights which had delays

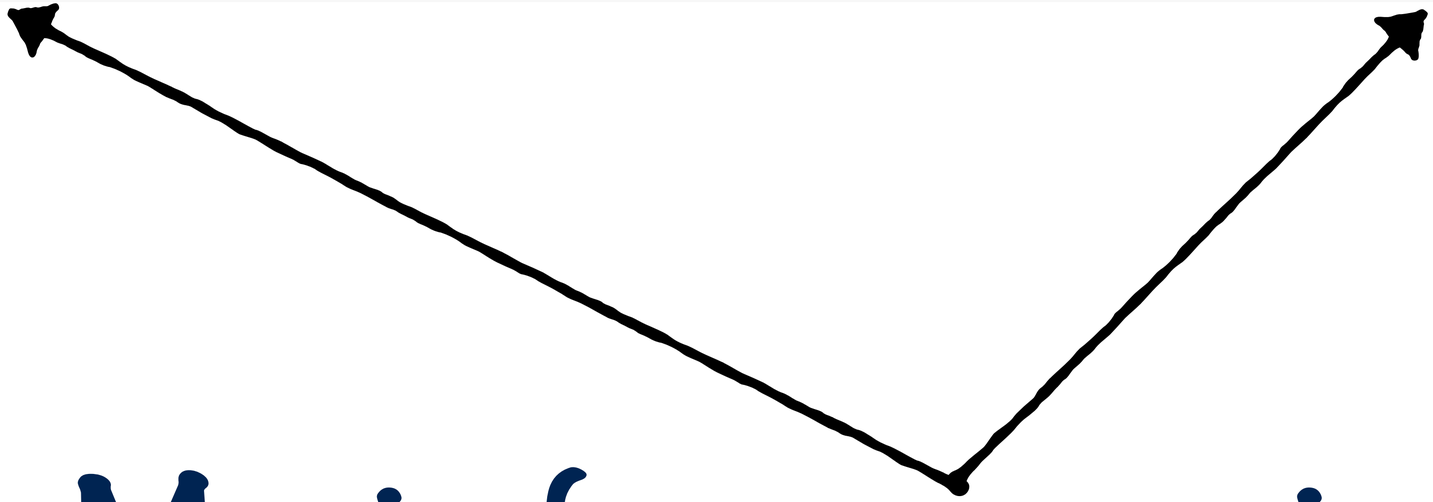
```
flightsParsed.filter(_.dep_delay>0).count().toDouble/flightsParsed.count().toDouble
```



We have been using the flightsParsed RDD quite a lot

% of flights which had delays

```
flightsParsed.filter(_.dep_delay>0).count().toDouble/flightsParsed.count().toDouble
```



Most of our computations need this RDD

% of flights which had delays

```
flightsParsed.filter(_.dep_delay>0).count().toDouble/flightsParsed.count().toDouble
```

Every time an action is computed on a child RDD of flightsParsed

It is materialized again

% of flights which had delays

```
flightsParsed.filter(_.dep_delay>0).count().toDouble/flightsParsed.count().toDouble
```

It is materialized again

i.e. the base data is loaded from file and parsed again

% of flights which had delays

```
flightsParsed.filter(_.dep_delay>0).count().toDouble/flightsParsed.count().toDouble
```

Instead, we can force the RDD to be materialized once

Then we can keep reusing it until we are done

% of flights which had delays

```
flightsParsed.persist()
```

The RDD has now been materialized

The data is cached in memory and can be reused

% of flights which had delays

```
flightsParsed.persist()
```

Once you are done with the RDD, you can discard it from memory

```
flightsParsed.unpersist()
```

Let's go back to our Flight related data from USDoT

We'll do the following

1. Parse the rows in the csv files ✓
2. Compute the average distance travelled by a flight ✓
3. Compute the % of flights which had delays ✓
4. Compute the average delay

Let's go back to our Flight related data from USDoT

We'll do the following

1. Parse the rows in the csv files ✓
2. Compute the average distance travelled by a flight ✓
3. Compute the % of flights which had delays ✓
4. Compute the average delay

Compute the average delay

We've already computed the average distance travelled

```
val totalDistance=flightsParsed.map(_.distance).reduce((x,y) => x+y)
val avgDistance=totalDistance/flightsParsed.count()
```

We can just replace this with the delay field

Compute the average delay

```
val totalDistance=flightsParsed.map(_.distance).reduce((x,y) => x+y)
val avgDistance=totalDistance/flightsParsed.count()
```

In this example, we computed the sum and count using separate actions

Compute the average delay

```
val totalDistance=flightsParsed.map(_.distance).reduce((x,y) => x+y)
val avgDistance=totalDistance/flightsParsed.count()
```

aggregate

Instead, we can use a single action to compute both sum and count

Compute the average delay

aggregate

Like reduce, aggregate will combine all the elements of the RDD in a specified manner

Compute the average delay

aggregate

There were 2 steps in reduce

1. Combining elements on individual nodes
2. Combining the results from all the nodes

Both steps use the same function

```
reduce((x,y) => x+y)
```

Compute the average delay

aggregate

1. Combining elements on individual nodes
2. Combining the results from all the nodes

With aggregate you can specify a separate function for each of these steps

Compute the average delay

aggregate

1. Combining elements on individual nodes
2. Combining the results from all the nodes

Aggregate will also need a zero value for these functions

Compute the average delay

aggregate

Let's use aggregate to compute the average

```
val sumCount=flightsParsed.map(_.dep_delay).aggregate((0.0,0))((acc, value) => (acc._1 + value, acc._2+1),
                                                               (acc1,acc2) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

Compute the average delay

aggregate

```
val sumCount=flightsParsed.map(_.dep_delay).
```

A tuple with both the sum of total delays and the count

Compute the average delay

aggregate

```
val sumCount=flightsParsed.map(_.dep_delay)
```

The delay field

Compute the average delay

aggregate

```
).aggregate((0.0,0))((acc, value) => (acc._1 + value, acc._2+1),
                     (acc1,acc2) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

The aggregate action

Compute the average delay

aggregate

```
regate((0.0,0))((acc, value) => (acc._1 + value, acc._2+1),
               (acc1,acc2) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

The function to use on each node

Compute the average delay

aggregate

```
regate((0.0,0))((acc, value) => (acc._1 + value, acc._2+1),
               (acc1,acc2) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

Represents 1 record in the RDD

Compute the average delay

aggregate

```
regate((0.0,0))((acc, value) => (acc._1 + value, acc._2+1),
               (acc1,acc2) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

We iteratively go through each value and perform some computation

Compute the average delay

aggregate

```
regate((0.0,0))((acc, value) => (acc._1 + value, acc._2+1),
                (acc1,acc2) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

The result of the computation from the previous record is in acc

Compute the average delay

aggregate

```
regate((0.0,0))((acc, value) => (acc._1 + value, acc._2+1),
                (acc1,acc2) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

A tuple representing (sum, count) till now

Compute the average delay

aggregate

```
regate((0.0,0))((acc, value) => (acc._1 + value, acc._2+1),
               (acc1,acc2) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

Calculate the sum portion of the tuple

Compute the average delay

aggregate

```
regate((0.0,0))((acc, value) => (acc._1 + value, acc._2+1),
               (acc1,acc2) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

And the count portion of the tuple

Compute the average delay

aggregate

```
.aggregate((0.0,0))((acc, value) => (acc._1 + value, acc._2+1),
                    (acc1,acc2) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

The function to use when combining the results from all the nodes

Compute the average delay

aggregate

```
aggregate((0.0,0))((acc, value) => (acc._1 + value, acc._2+1),
                   (acc1,acc2) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

We have a bunch of sum, count tuples from each node

Compute the average delay

aggregate

```
aggregate((0.0,0))((acc, value) => (acc._1 + value, acc._2+1),
                   (acc1,acc2) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

We iteratively go through these tuples

Compute the average delay

aggregate

```
aggregate((0.0,0))((acc, value) => (acc._1 + value, acc._2+1),
                   (acc1,acc2) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

acc2 represents the current tuple

Compute the average delay

aggregate

```
aggregate((0.0,0))((acc, value) => (acc._1 + value, acc._2+1),
                   (acc1,acc2) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

acc1 represents the result of the computation from the previous tuple

Compute the average delay

aggregate

```
aggregate((0.0,0))((acc, value) => (acc._1 + value, acc._2+1),
                   (acc1,acc2) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

acc1 represents the (sum,count) till now as we go through the tuples from each node

Compute the average delay

aggregate

```
aggregate((0.0,0))((acc, value) => (acc._1 + value, acc._2+1),
                   (acc1,acc2) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

Add up the sums from each node, first portion of tuple

Compute the average delay

aggregate

```
aggregate((0.0,0))((acc, value) => (acc._1 + value, acc._2+1),
                   (acc1,acc2) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

Add up the counts from each node, second portion of tuple

Compute the average delay

aggregate

```
p_delay).aggregate((0.0,0))((acc, value) => (acc._1 + valu
                           (acc1,acc2) => (acc1._1+acc2._1,ac
```

The zero value

Compute the average delay

aggregate

```
p_delay).aggregate((0.0,0))((acc, value) => (acc._1 + valu
                           (acc1,acc2) => (acc1._1+acc2._1,ac
```

This is the initial value of the (sum,count) tuple on each node

Compute the average delay

aggregate

```
p_delay).aggregate((0.0,0))((acc, value) => (acc._1 + valu
                            (acc1,acc2) => (acc1._1+acc2._1,ac
```

It is also the initial value of the (sum,count) tuple when the results from all nodes are combined

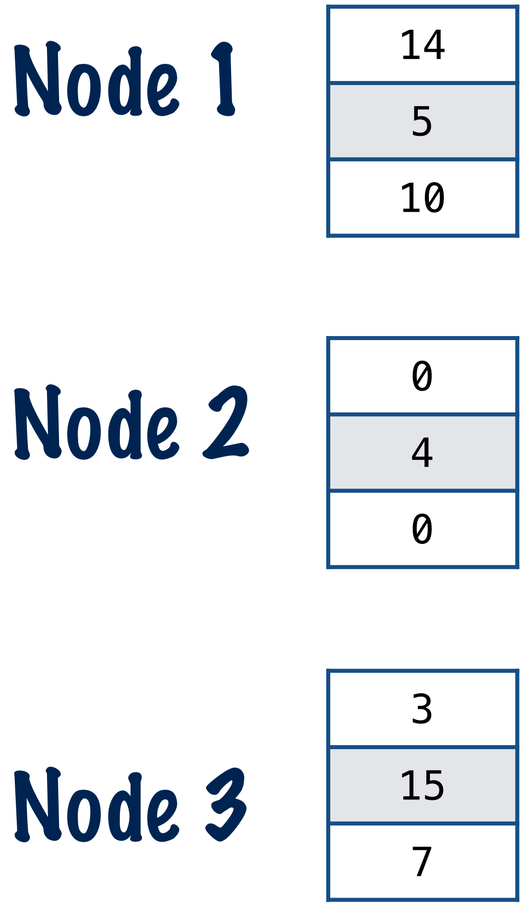
Compute the average delay

```
.aggregate((0.0,0))((acc, value) => (acc._1 + value, acc._2+1),
                    (acc1,acc2) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

Let's see how this works visually

Compute the average delay

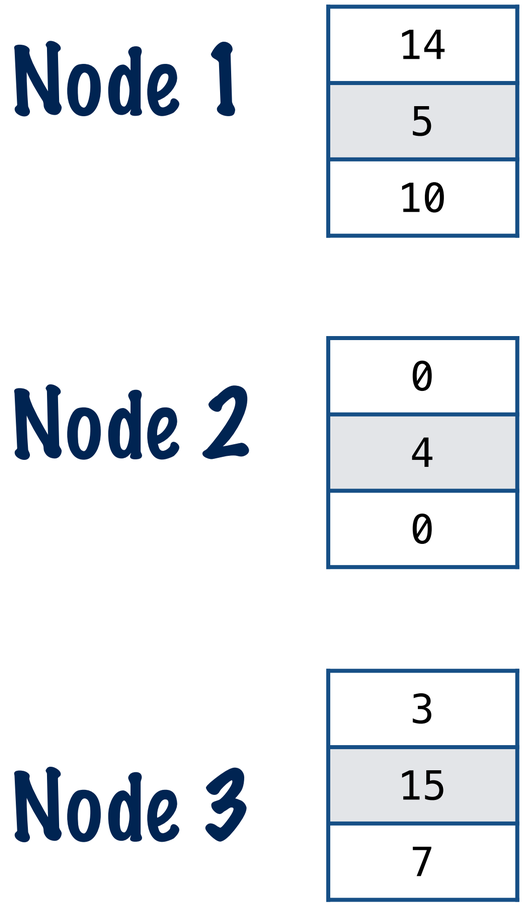
```
.aggregate((0.0,0))((acc, value) => (acc._1 + value, acc._2+1),
                    (acc1,acc2) => (acc1._1+acc2._1,acc1._2+acc2._2))
```



This the delays RDD

Compute the average delay

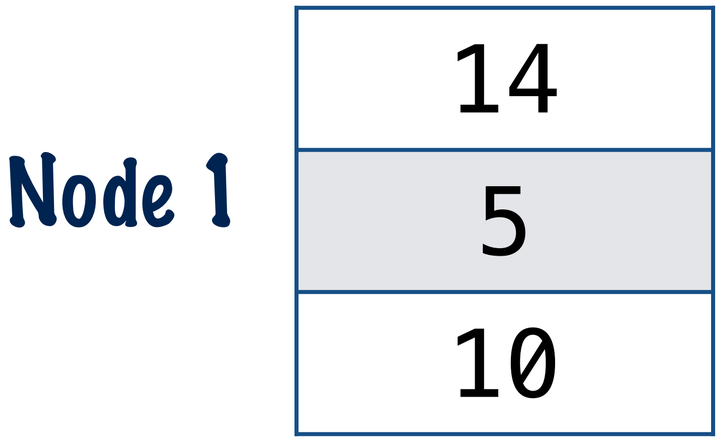
```
.aggregate((0.0,0))((acc, value) => (acc._1 + value, acc._2+1),
                    (acc1,acc2) => (acc1._1+acc2._1,acc1._2+acc2._2))
```



First, the operation is performed on each node

Compute the average delay

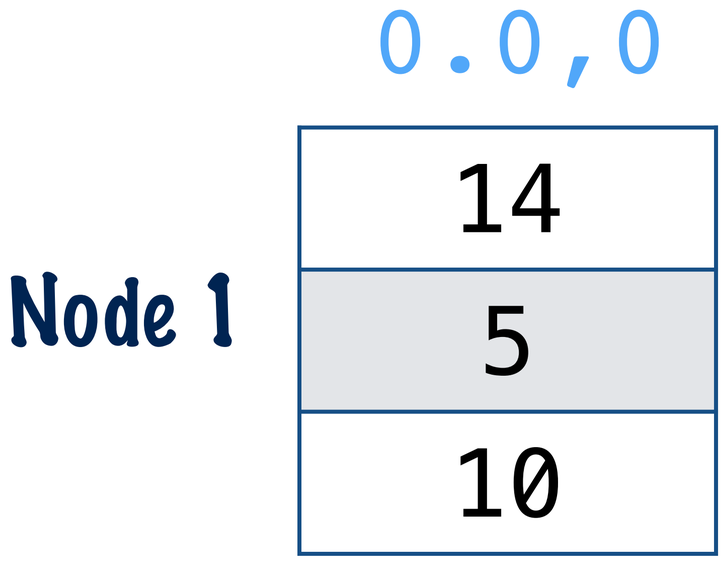
```
.aggregate((0.0,0))((acc, value) => (acc._1 + value, acc._2+1),
                    (acc1,acc2) => (acc1._1+acc2._1,acc1._2+acc2._2))
```



First, the operation is performed on each node

Compute the average delay

```
.aggregate((0.0,0))((acc, value) => (acc._1 + value, acc._2+1),
                    (acc1,acc2) => (acc1._1+acc2._1,acc1._2+acc2._2))
```



We start with the zero value

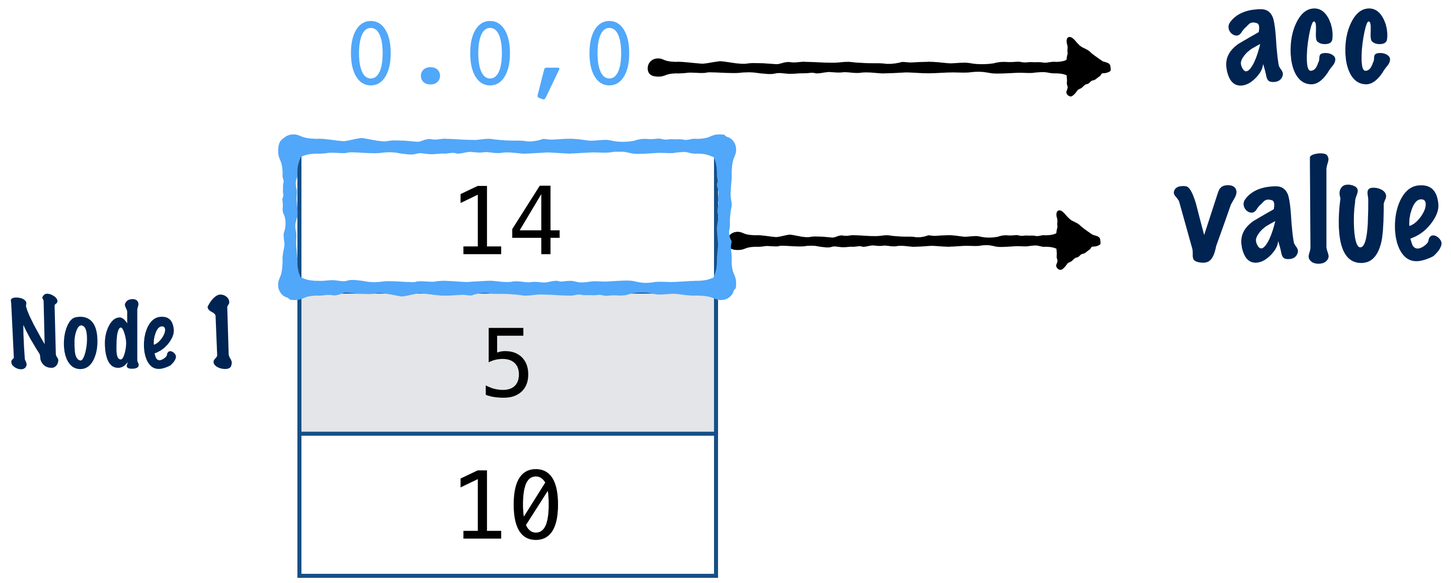
0.0 represents the sum of values, which is a Double

0 represents the count of values, which is an Int

Compute the average delay

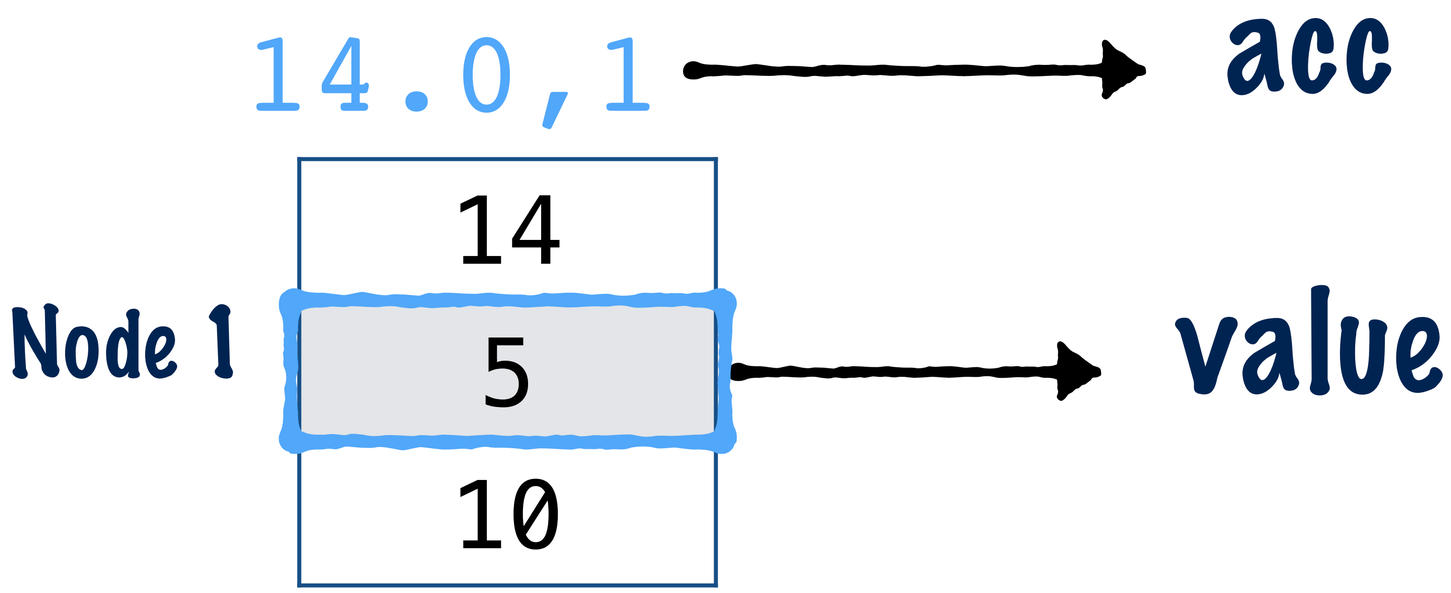
```
.aggregate((0.0,0))((acc, value) => (acc._1 + value, acc._2+1),
                    (acc1,acc2) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

This function is applied iteratively on each value



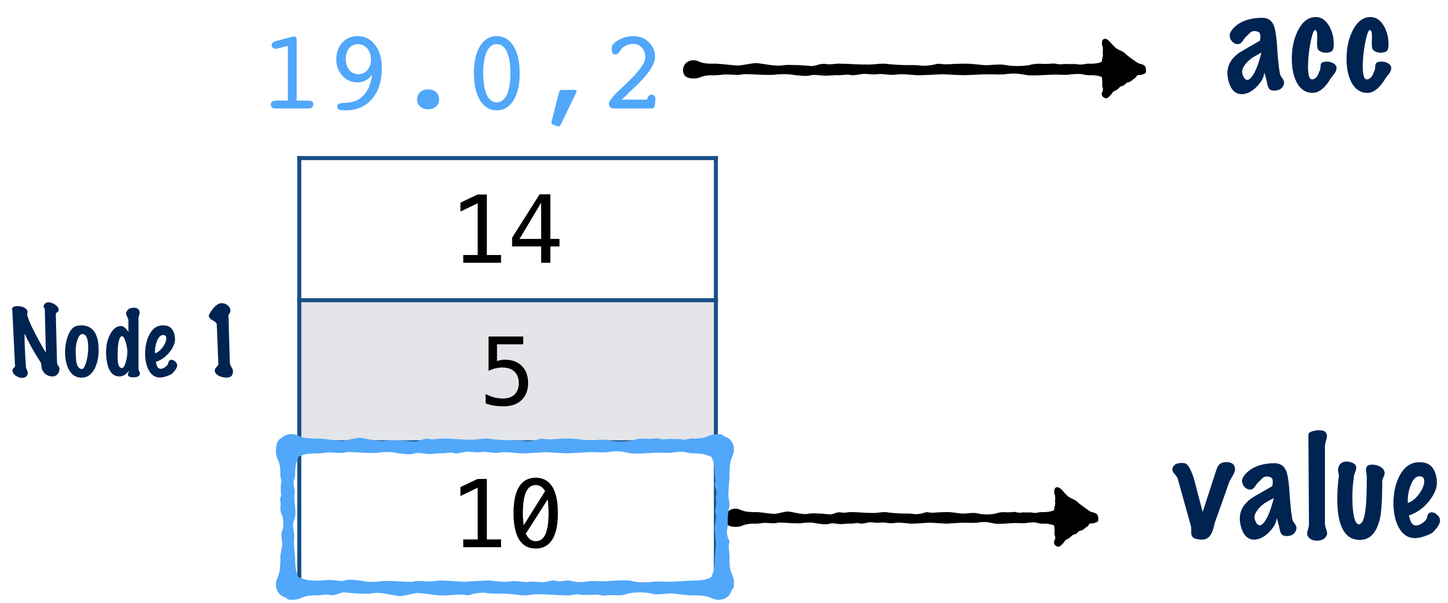
Compute the average delay

```
.aggregate((0.0,0))((acc, value) => (acc._1 + value, acc._2+1),
                    (acc1,acc2) => (acc1._1+acc2._1,acc1._2+acc2._2))
```



Compute the average delay

```
.aggregate((0.0,0))((acc, value) => (acc._1 + value, acc._2+1),
                    (acc1,acc2) => (acc1._1+acc2._1,acc1._2+acc2._2))
```



Compute the average delay

```
.aggregate((0.0,0))((acc, value) => (acc._1 + value, acc._2+1),
                    (acc1,acc2) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

Node 1 29.0,3

The same thing is done on each node

Compute the average delay

```
.aggregate((0.0,0))((acc, value) => (acc._1 + value, acc._2+1),
                    (acc1,acc2) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

Node 1 29.0,3

Node 2 4.0,3

Node 3 25.0,3

Now the second function is used to combine these tuples

Compute the average delay

```
.aggregate((0.0,0))((acc, value) => (acc._1 + value, acc._2+1),
                    (acc1,acc2) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

0.0,0

Node 1 29.0,3

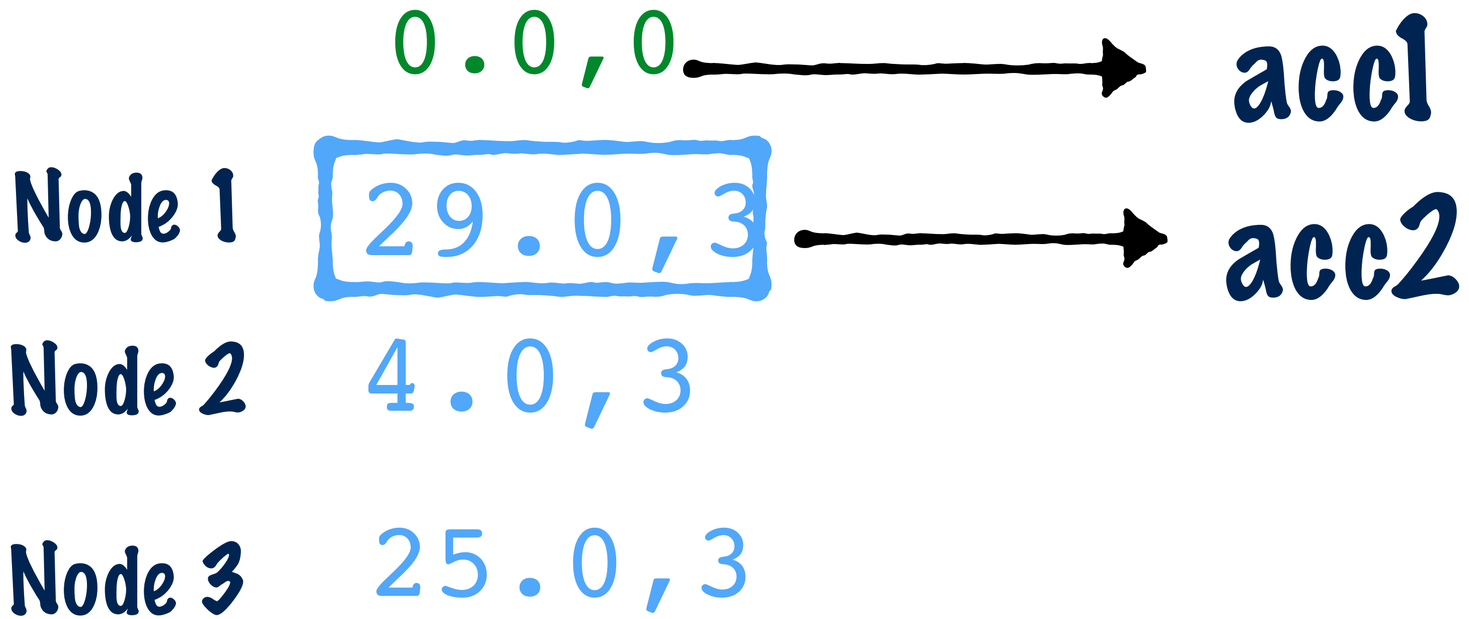
Node 2 4.0,3

Node 3 25.0,3

We'll start again with the zero value

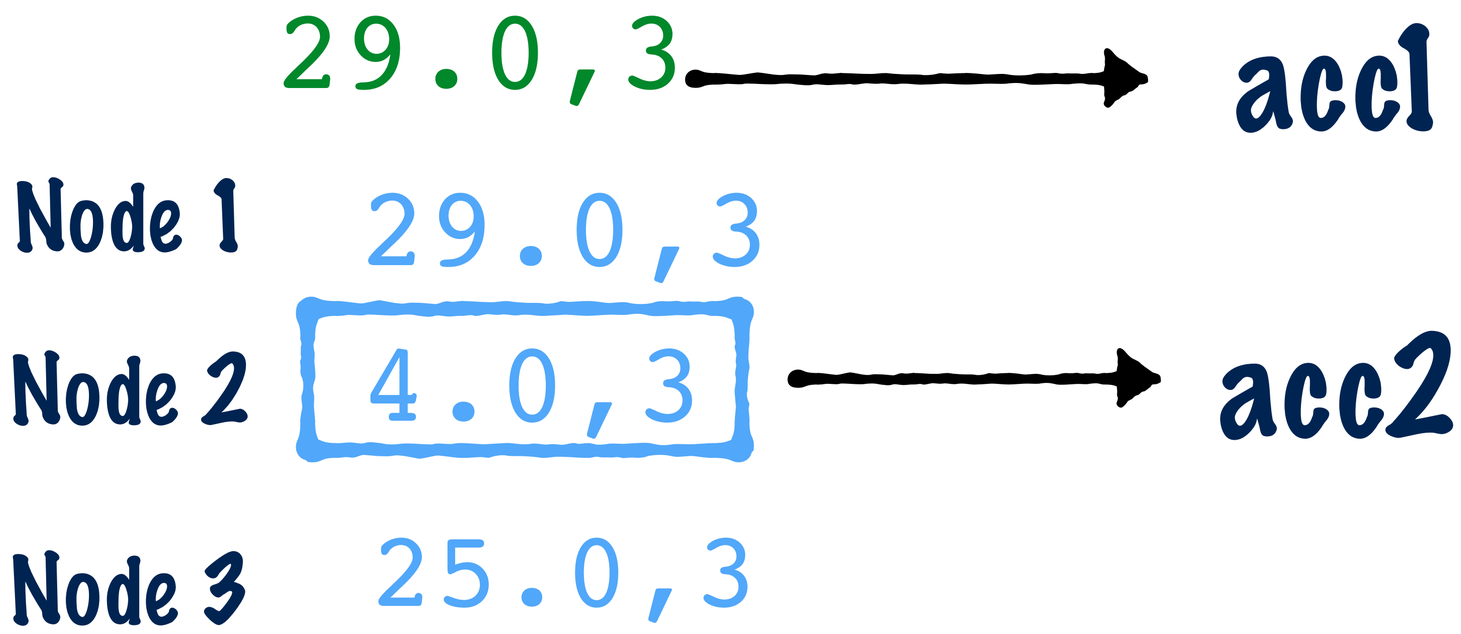
Compute the average delay

```
.aggregate((0.0,0))((acc, value) => (acc._1 + value, acc._2+1),
                    (acc1,acc2) => (acc1._1+acc2._1,acc1._2+acc2._2))
```



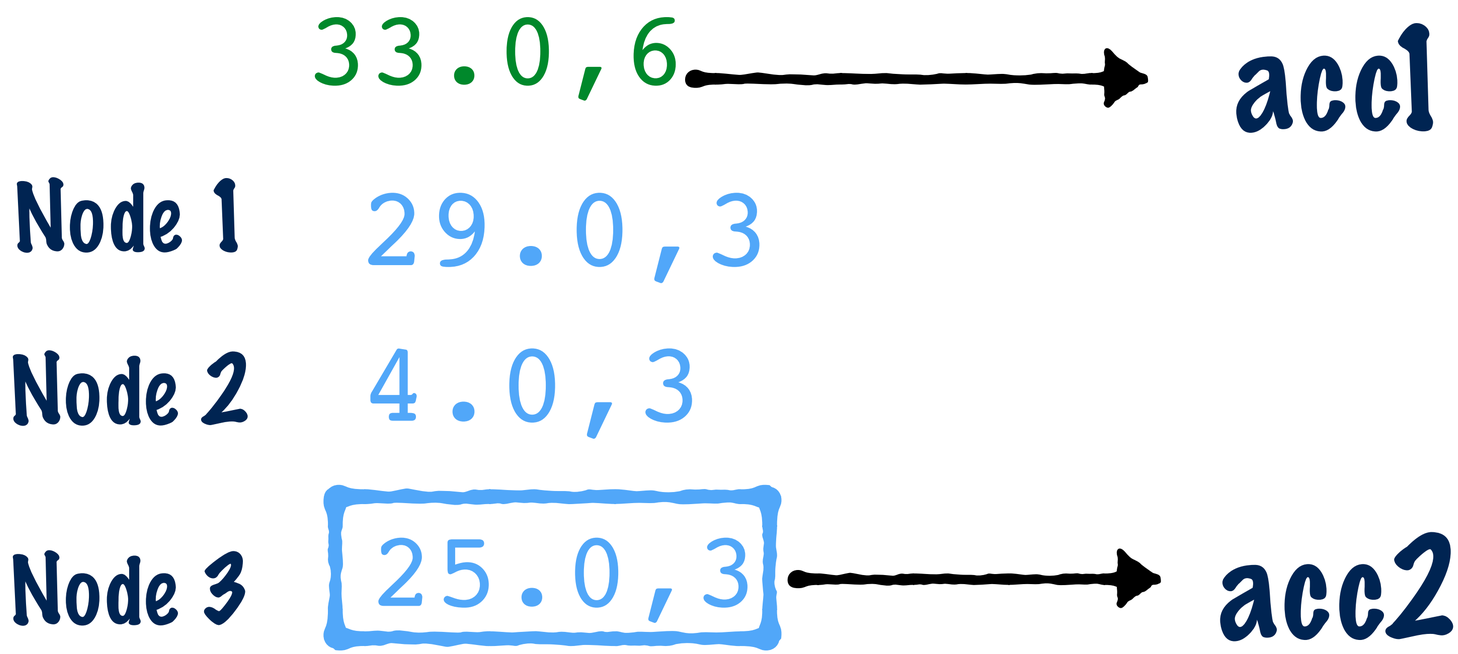
Compute the average delay

```
.aggregate((0.0,0))((acc, value) => (acc._1 + value, acc._2+1),
                    (acc1,acc2) => (acc1._1+acc2._1,acc1._2+acc2._2))
```



Compute the average delay

```
.aggregate((0.0,0))((acc, value) => (acc._1 + value, acc._2+1),
                    (acc1,acc2) => (acc1._1+acc2._1,acc1._2+acc2._2))
```



Compute the average delay

```
val sumCount=flightsParsed.map(_.dep_delay).aggregate(
```

```
58.0,9
```

sum

count

We just need to divide the 2 to get the average delay

Compute the average delay

```
val sumCount=flightsParsed.map(_.dep_delay).aggregate(
```

```
sumCount._1/sumCount._2
```

We just need to divide the 2 to get the average delay

Compute the average delay

aggregate

```
p_delay).aggregate((0.0,0))((acc, value) => (acc._1 + valu
                            (acc1,acc2) => (acc1._1+acc2._1,ac
```

aggregate is a curried function

Compute the average delay

aggregate

```
p_delay).aggregate((0.0,0))((acc, value) => (acc._1 + valu
                           (acc1,acc2) => (acc1._1+acc2._1,ac
```

In scala, curried functions are functions that take their parameters separately

Compute the average delay

aggregate

```
p_delay).aggregate((0.0,0))((acc, value) => (acc._1 + valu
                           (acc1,acc2) => (acc1._1+acc2._1,ac
```

sum(x,y) vs sum(x)(y)

Non-curried vs curried

Compute the average delay

aggregate

```
p_delay).aggregate((0.0,0))((acc, value) => (acc._1 + valu
                           (acc1,acc2) => (acc1._1+acc2._1,ac
```

aggregate(zero)(fn1, fn2)

aggregate is curried

Compute the average delay

aggregate

```
p_delay).aggregate((0.0,0))((acc, value) => (acc._1 + valu
                           (acc1,acc2) => (acc1._1+acc2._1,ac
```

Functions in scala are first class citizens

Compute the average delay

aggregate

```
p_delay).aggregate((0.0,0))((acc, value) => (acc._1 + valu
                           (acc1,acc2) => (acc1._1+acc2._1,ac
```

You can create a function and treat it like a variable / object

Compute the average delay

aggregate

```
p_delay).aggregate((0.0,0))((acc, value) => (acc._1 + valu
                            (acc1,acc2) => (acc1._1+acc2._1,ac
```

With curried functions, you can create a variable which is the function with few parameters specified

Ex. just the zero value is defined at first

Compute the average delay

aggregate

```
p_delay).aggregate((0.0,0))((acc, value) => (acc._1 + valu
                    (acc1,acc2) => (acc1._1+acc2._1,ac
```

Then you can reuse the function + zero value, with different values for the rest of the parameters

Compute the average delay

```
y).aggregate((0.0,0))((acc, value) => (acc._1 + value, acc._2+1),
                      (acc1,acc2) => (acc1._1+acc2._1,acc1._2+acc2._2))
```

With the aggregate function we were able to

1. Specify different ways to accumulate values on the individual nodes and on the final node

2. Take an RDD of numbers and return a tuple

Let's go back to our Flight related data from USDoT

We'll do the following

1. Parse the rows in the csv files ✓
2. Compute the average distance travelled by a flight ✓
3. Compute the % of flights which had delays ✓
4. Compute the average delay ✓

Let's go back to our Flight related data from USDoT

We'll do the following:

1. Parse the rows in the csv files ✓
2. Compute the average distance travelled by a flight ✓
3. Compute the % of flights which had delays ✓
4. Compute the average delay ✓
5. Compute a frequency distribution of delays

A frequency distribution

We want to compute a frequency distribution of Flight delays

Delay in hrs	Number of Flights
0–1	1000
1–2	453
2–3	20
>3	2

A frequency distribution

```
flightsParsed.map(x => (x.dep_delay/60).toInt).countByValue()
```

This takes the departure delay field bins it into 1 hr intervals

Delay in hrs
0–1
0–1
2–3
1–2

A frequency distribution

```
flightsParsed.map(x => (x.dep_delay/60).toInt).countByValue()
```

```
Map(0 -> 452963, 5 -> 249, 10 -> 15, 24 -> 3, 25 -> 1, 14 -> 13, 20 -> 4, 1 -> 16016, 6 -> 113, 28 -> 1, 21 -> 3, 9 -
> 26, 13 -> 15, 2 -> 4893, 17 -> 2, 12 -> 9, 7 -> 66, 3 -> 1729, 11 -> 12, 8 -> 43, 4 -> 701, 15 -> 4)
```

This counts the number of times each value occurs

A frequency distribution

```
Map 0 -> 452963  5 -> 249, 10 -> 15, 24 -> 3, 25 -> 1, 14 -> 13, 20 -> 4, 1 -> 16
> 26, 13 -> 15, 2 -> 4893, 17 -> 2, 12 -> 9, 7 -> 66, 3 -> 1729, 11 -> 12, 8 -> 4
```

This result is a map with the values and the number of times they occur

A frequency distribution

```
Map(0 -> 452963, 5 -> 249, 10 ->
> 26, 13 -> 15, 2 -> 4893, 17 ->
```

Num. Flights with Delay < 1hr

A frequency distribution

```
Map(0 -> 452963, 5 -> 249, 10 -> 15, 24 -> 3, 25 -> 1, 1
> 26, 13 -> 15, 2 -> 4893, 17 -> 2, 12 -> 9, 7 -> 66, 3
```

Flights with delay between 1-2 hrs

..and so on