RESILIENT DISTRIBUTED DATASETS

Resilient Distributed Datasets

We've talked a little bit about RDDs before

Recap

Resilient Distributed Datasets

RDDs are the main programming abstraction in Spark

Recap

Resilient Distributed Datasets

RDDs are in-memory objects and all data is processed using them

Recap

Resilient Distributed Datasets

in-memory

In-memory, yet resilient!

RDDs are designed to be fault-tolerant while dealing with vast amounts of data

Recap

Resilient Distributed Datasets

There are a lot of complexities involved in dealing with vast amounts of data

Recap

Resilient Distributed Datasets

Lot of complexities

1. Distributing data across a cluster
2. Fault tolerance (if in-memory data is lost)
3. Efficiently processing billions of rows of data

Recap

Resilient Distributed Datasets

1. Distributing data across a cluster
2. Fault tolerance
3. Efficiently processing billions of rows of data

Think of other environments
like Java or Excel

It would be the responsibility of the
user to deal with most of these issues

Recap

Resilient Distributed Datasets

1. Distributing data across a cluster
2. Fault tolerance
3. Efficiently processing billions of rows of data

With RDDs, you can interact and play with billions of rows of data

...without caring about any of the complexities

Recap

Resilient Distributed Datasets

1. Distributing data across a cluster
2. Fault tolerance
3. Efficiently processing billions of rows of data

Spark takes care of all the complexities under the hood

The user is completely abstracted!

Resilient Distributed Datasets

RDDs have 3 defining characteristics

partitions

read-only

lineage

Resilient Distributed Datasets

partitions

read-only

lineage

Let's understand each of these

RDDs partitions

RDDs represent data in-memory

1	Swetha	30	Bangalore
2	Vitthal	35	New Delhi
3	Navdeep	25	Mumbai
4	Janani	35	New Delhi
5	Navdeep	25	Mumbai
6	Janani	35	New Delhi

RDDs partitions

When you have large amounts of data

1. It can take very long to process them on a single machine

2. The data may be too large to even be stored and processed on a single machine

1	Swetha	30	Bangalore
2	Vitthal	35	New Delhi
3	Navdeep	25	Mumbai
4	Janani	35	New Delhi
5	Navdeep	25	Mumbai
6	Janani	35	New Delhi

RDDs partitions

Instead you can divide the data into partitions

and distribute them to multiple machines

1	Swetha	30	Bangalore
2	Vitthal	35	New Delhi
3	Navdeep	25	Mumbai
4	Janani	35	New Delhi
5	Navdeep	25	Mumbai
6	Janani	35	New Delhi

RDDs partitions

These partitions are distributed to multiple machines/nodes

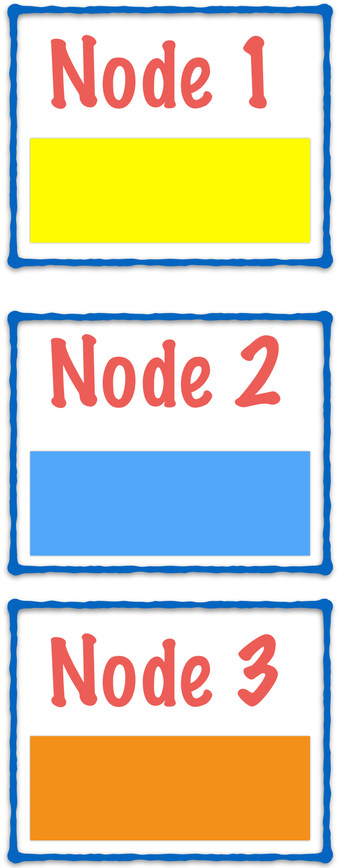
1	Swetha	30	Bangalore
2	Vitthal	35	New Delhi

3	Navdeep	25	Mumbai
4	Janani	35	New Delhi

5	Navdeep	25	Mumbai
6	Janani	35	New Delhi

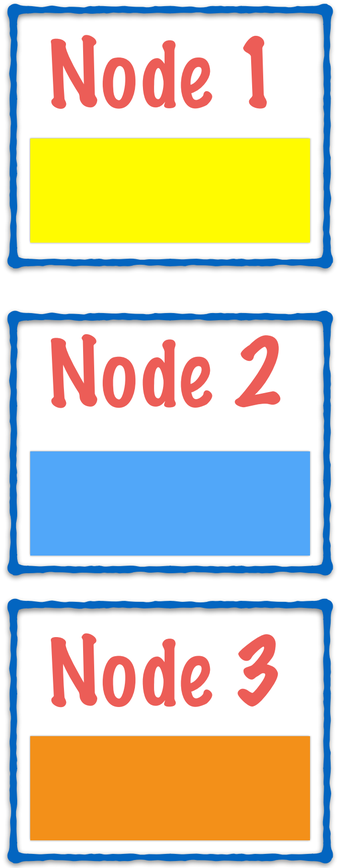
RDDs partitions

These partitions are distributed to multiple machines/nodes



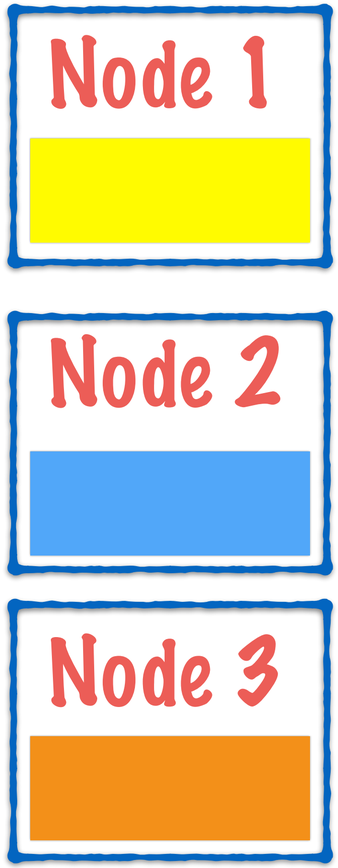
RDDs partitions

The nodes can work in parallel, making the processing much faster than if it were on a single machine



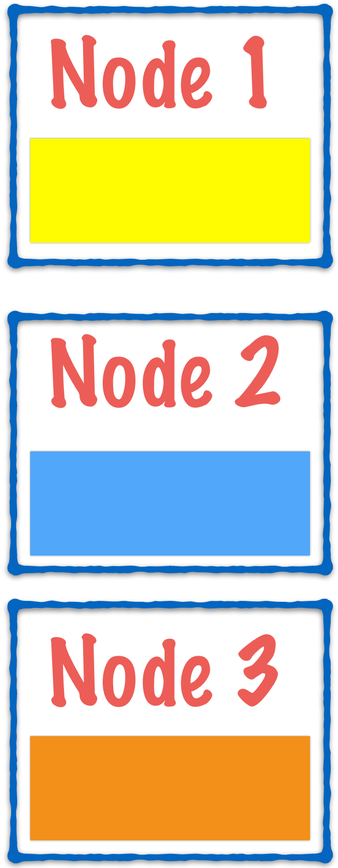
RDDs partitions

Each partition, is kept in-memory on a node in the cluster



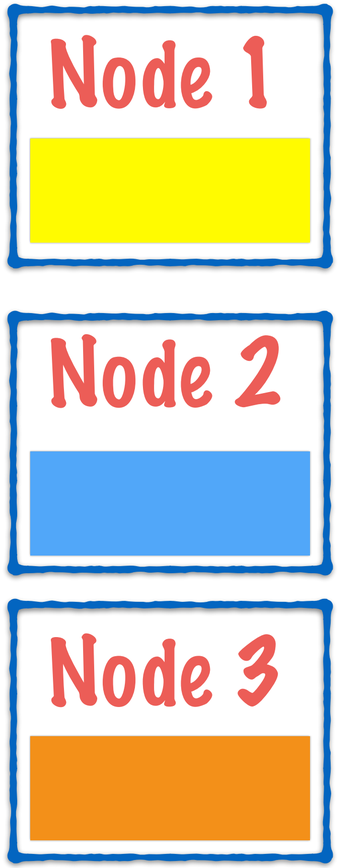
RDDs partitions

How is the data partitioned and distributed?



RDDs partitions

Usually the data comes from a distributed filesystem like HDFS where the data is already partitioned!

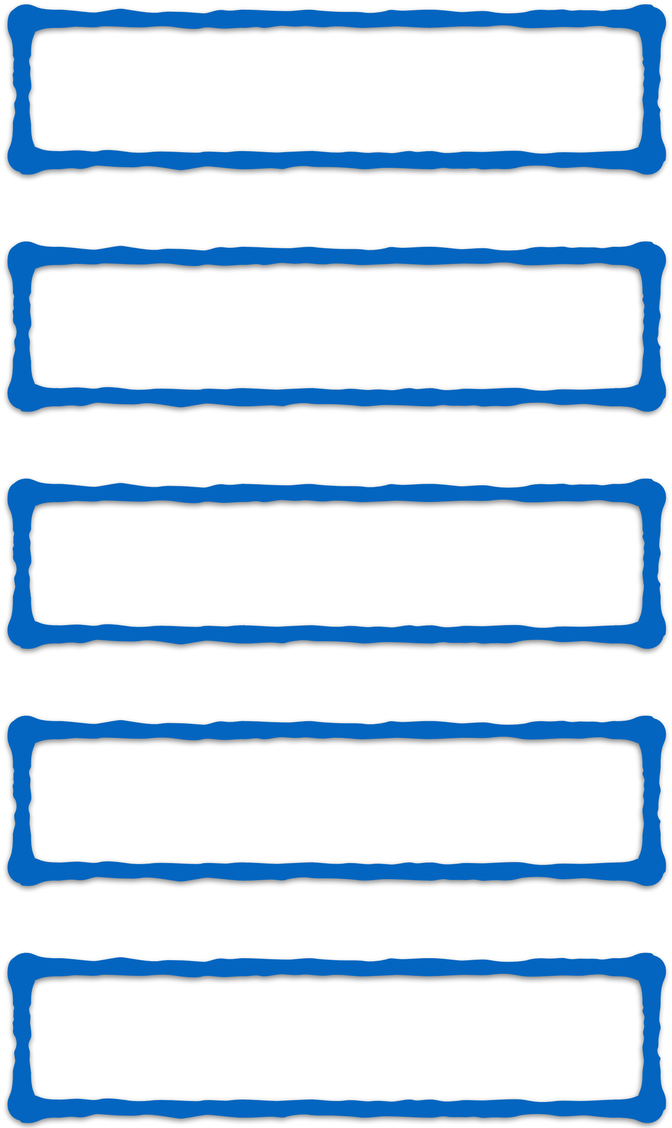


HDFS

The Hadoop Distributed File System

Hadoop uses this to store data across multiple disks

HDFS

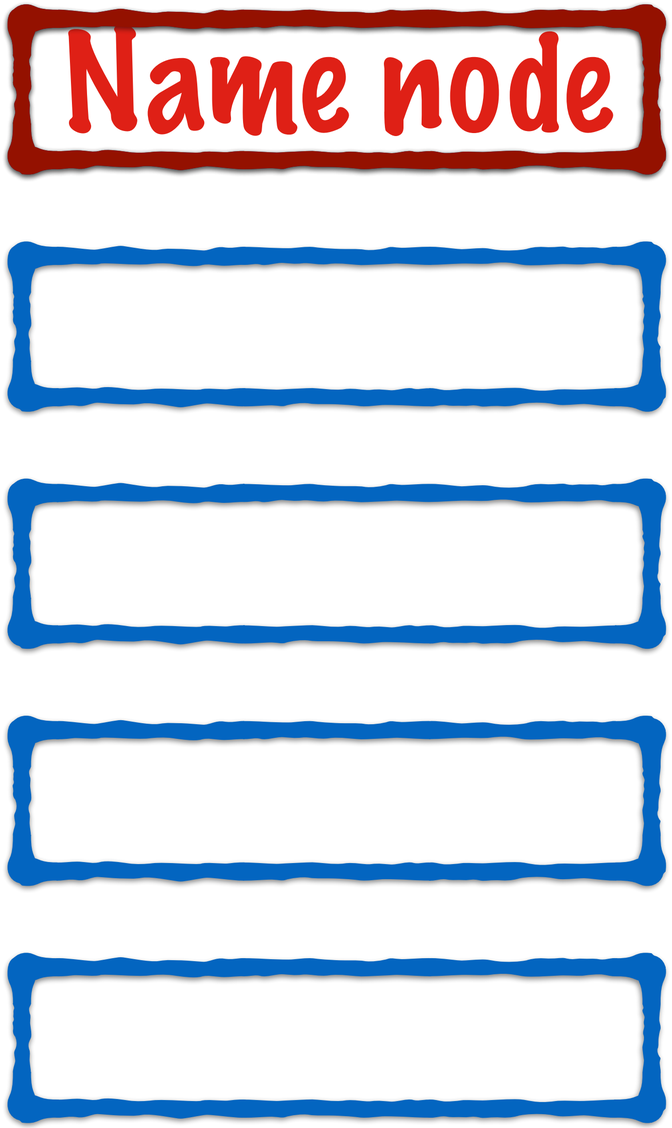


Hadoop is normally deployed on a group of machines

Cluster

Each machine in the cluster is a node

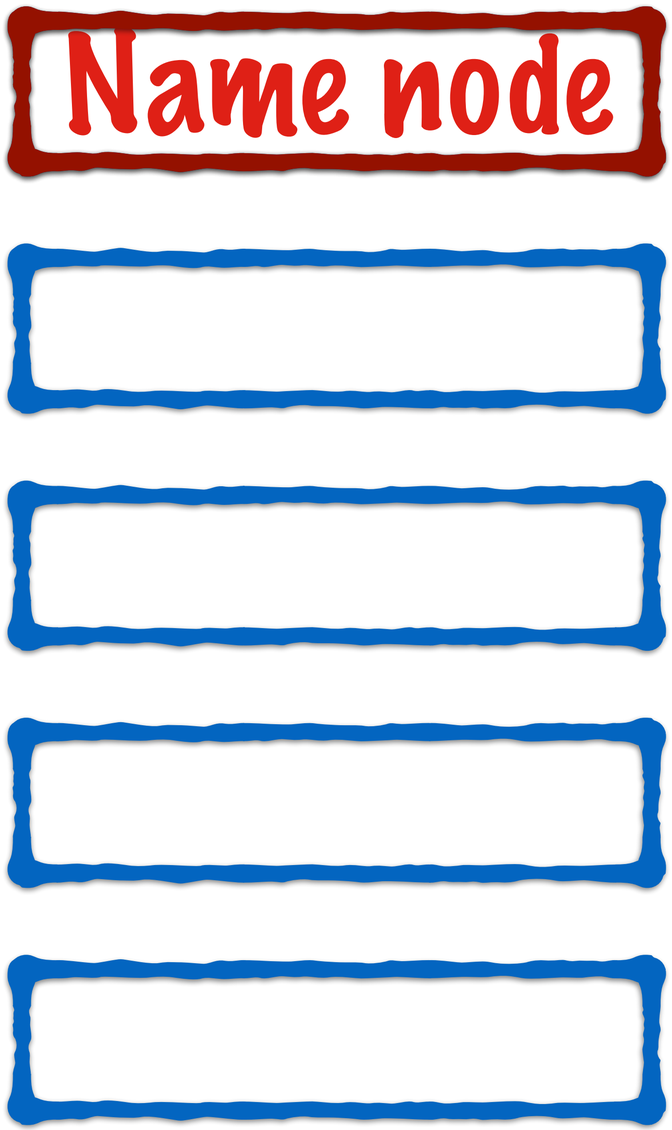
HDFS



One of the nodes acts as the **master node**

This node **manages** the overall file system

HDFS

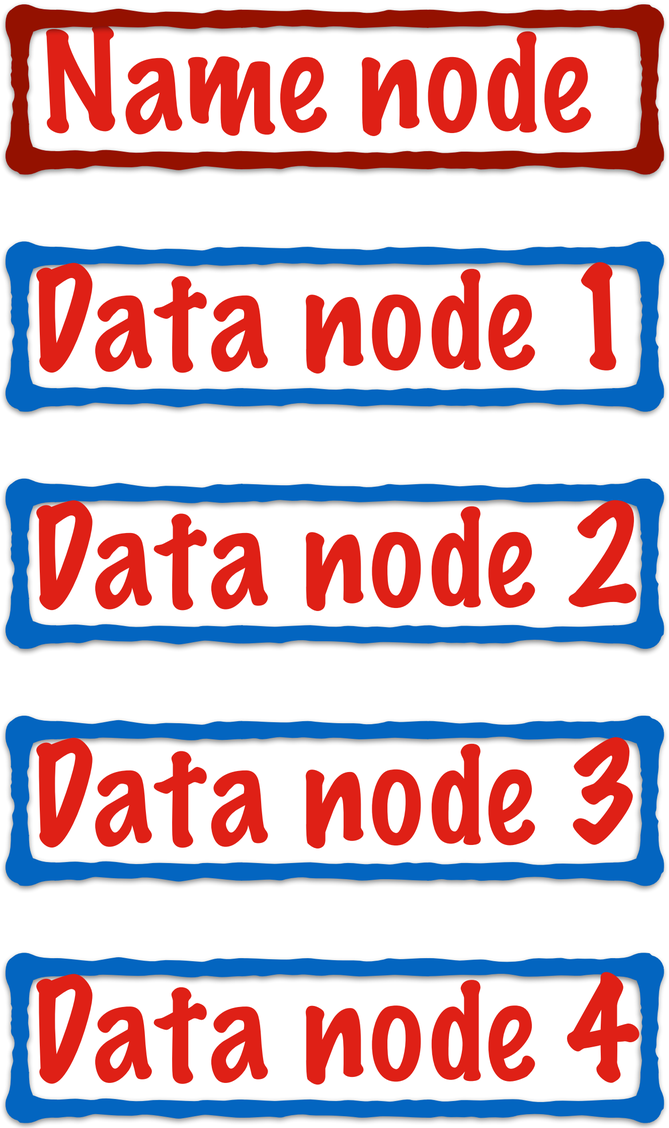


The name node stores

1. The directory structure

2. Metadata for all the files

HDFS



Other nodes are called data nodes

The data is physically stored on these nodes

HDFS

Here is a large text file

next up previous contents index
Next: Dynamic indexing Up: Index construction Previous: Single-pass in-memory indexing Contents Index

Distributed indexing

Collections are often so large that we cannot perform index construction efficiently on a single machine. This is particularly true of the World Wide Web for which we need large computer clusters [*]to construct any reasonably sized web index. Web search engines, therefore, use distributed indexing algorithms for index construction. The result of the construction process is a distributed index that is partitioned across several machines - either according to term or according to document. In this section, we describe distributed indexing for a
term-partitioned index . Most large search engines prefer a document-partitioned index (which can be easily generated from a term-partitioned index). We discuss this topic further in Section 20.3 (page [*]).

The distributed index construction method we describe in this section is an application of MapReduce , a general architecture for distributed computing. MapReduce is designed for large computer clusters. The point of a cluster is to solve large computing problems on cheap commodity machines or nodes that are built from standard parts (processor, memory, disk) as opposed to on a supercomputer with specialized hardware. Although hundreds or thousands of machines are available in such clusters, individual machines can fail at any time. One requirement for robust distributed indexing is, therefore, that we divide the work up into chunks that we can easily assign and - in case of failure - reassign. A master node directs the process of assigning and reassigning tasks to individual worker nodes.

The map and reduce phases of MapReduce split up the computing job into chunks that standard machines can process in a short time. The various steps of MapReduce are shown in Figure 4.5 and an example on a collection consisting of two documents is shown in Figure 4.6 . First, the input data, in our case a collection of web pages, are split into n splits where the size of the split is chosen to ensure that the work can be distributed evenly (chunks should not be too large) and efficiently (the total number of chunks we need to manage should not be too large); 16 or 64 MB are good sizes in distributed indexing. Splits are not preassigned to machines, but are instead assigned by the master node on an ongoing basis: As a machine finishes processing one split, it is assigned the next one. If a machine dies or becomes a laggard due to hardware problems, the split it is working on is simply reassigned to another machine.

Figure 4.5: An example of distributed indexing with MapReduce. Adapted from Dean and Ghemawat (2004).
\includegraphics[width=11.5cm]{art/mapreduce2.eps}
In general, MapReduce breaks a large computing problem into smaller parts by recasting it in terms of manipulation of key-value pairs . For indexing, a key-value pair has the form (termID,docID). In distributed indexing, the mapping from terms to termIDs is also distributed and therefore more complex than in single-machine indexing. A simple solution is to maintain a (perhaps precomputed) mapping for frequent terms that is copied to all nodes and to use terms directly (instead of termIDs) for infrequent terms. We do not address this problem here and assume that all nodes share a consistent term $\rightarrow$ termID mapping.

The map phase of MapReduce consists of mapping splits of the input data to key-value pairs. This is the same parsing task we also encountered in BSBI and SPIMI, and we therefore call the machines that execute the map phase parsers . Each parser writes its output to local intermediate files, the segment files (shown as \fbox{a-f\medstrut} \fbox{g-p\medstrut} \fbox{q-z\medstrut} in Figure 4.5).

For the reduce phase , we want all values for a given key to be stored close together, so that they can be read and processed quickly. This is achieved by partitioning the keys into j term partitions and having the parsers write key-value pairs for each term partition into a separate segment file. In Figure 4.5 , the term partitions are according to first letter: a-f, g-p, q-z, and $j=3$. (We chose these key ranges for ease of exposition. In general, key ranges need not correspond to contiguous terms or termIDs.) The term partitions are defined by the person who operates the indexing system (Exercise 4.6). The parsers then write corresponding segment files, one for each term partition. Each term partition thus corresponds to r segments files, where r is the number of parsers. For instance, Figure 4.5 shows three a-f segment files of the a-f partition, corresponding to the three parsers shown in the figure.

Collecting all values (here: docIDs) for a given key (here: termID) into one list is the task of the inverters in the reduce phase. The

Let's see how this file is stored in HDFS

HDFS

Block 1

Block 2

Block 3

Block 4

Block 5

Block 6

Block 7

Block 8

First the file is broken into blocks of size 128 MB

HDFS

Block 1

Block 2

Block 3

Block 4

Block 5

Block 6

Block 7

Block 8

First the file is broken into

blocks of size 128 MB

This size is chosen to minimize the time to seek to the block on the disk

HDFS

Block 1

Block 2

Block 3

Block 4

Block 5

Block 6

Block 7

Block 8

These blocks are then stored across the data nodes

HDFS

Data node 1

Block 1 | Block 2

Data node 3

Block 5 | Block 6

Data node 2

Block 3 | Block 4

Data node 4

Block 7 | Block 8

Name node

The name node stores metadata

HDFS

Block locations for each file are stored in the name node

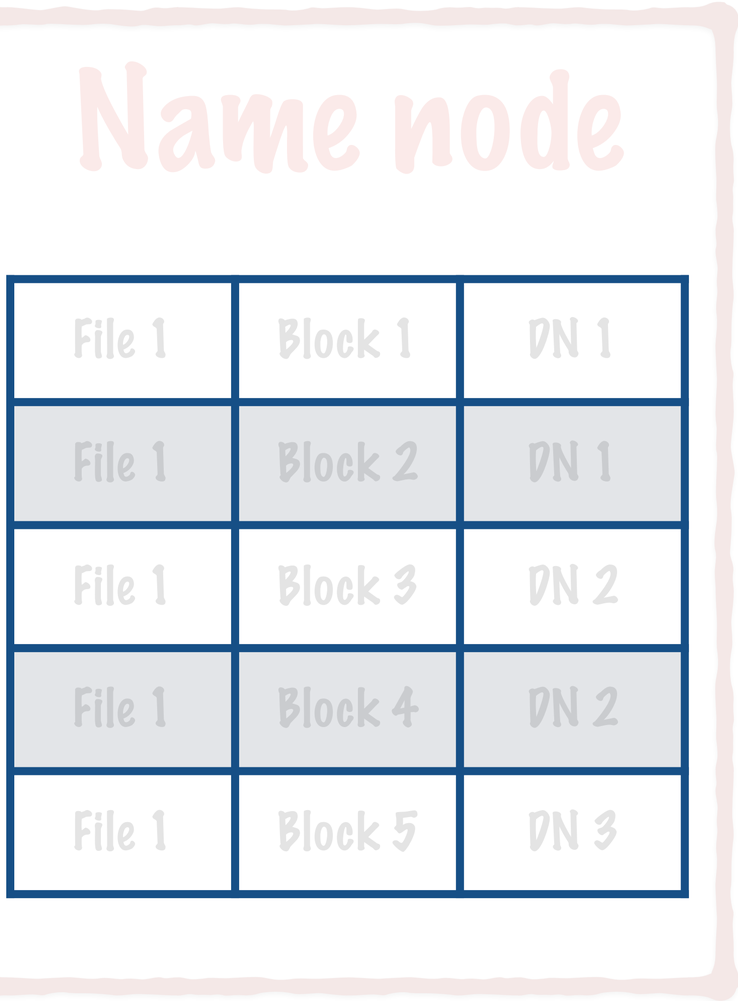
Name node

File 1	Block 1	DN 1
File 1	Block 2	DN 1
File 1	Block 3	DN 2
File 1	Block 4	DN 2
File 1	Block 5	DN 3

HDFS

A file is read using

1. The **metadata** in name node
2. The **blocks** in the data nodes



HDFS

Data node 1

Block 1 | Block 2

Data node 3

Block 5 | Block 6

Data node 2

Data node 4

Name node

File 1	Block 1	DN 1
File 1	Block 2	DN 1
File 1	Block 3	DN 2
File 1	Block 4	DN 2
File 1	Block 5	DN 3

What if one of the blocks gets corrupted?

HDFS

Data node 1

Block 1 | Block 2

Data node 3

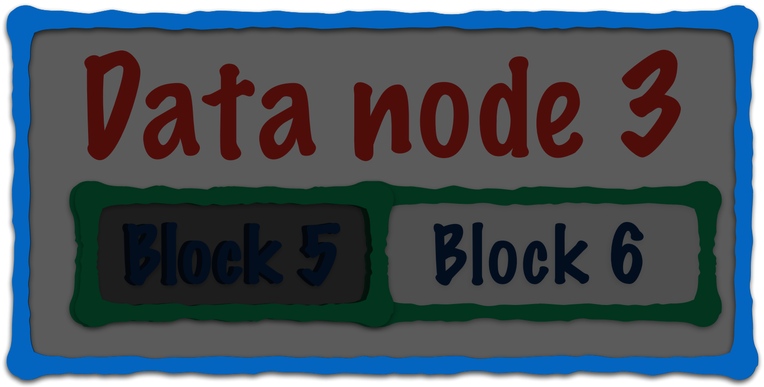
Block 5 | Block 6

Name node

File 1	Block 1	DN 1
File 1	Block 2	DN 1
File 1	Block 3	DN 2
File 1	Block 4	DN 2
File 1	Block 5	DN 3

Or one of the data nodes crashes?

HDFS



This is one of the key challenges in distributed storage

Name node

File 1	Block 1	DN 1
File 1	Block 2	DN 1
File 1	Block 3	DN 2
File 1	Block 4	DN 2
File 1	Block 5	DN 3

HDFS

Data node 1

Block 1

Data node 3

Block 6

Data node 2

Block 3

Block 4

Data node 4

Block 7

Block 8

You can define a replication factor in HDFS

Name node

File 1	Block 1	DN 1
File 1	Block 2	DN 1
File 1	Block 3	DN 2
File 1	Block 4	DN 2
File 1	Block 5	DN 3

HDFS

Data node 1

Block 1 | Block 2

Data node 2

Block 3 | Block 4

Block 1 | Block 2

Data node 3

Block 5 | Block 6

Name node

Each block is replicated, and the replicas are stored in different data nodes

HDFS

Data node 1

Block 1 | Block 2

Data node 3

Block 5 | Block 6

Name node

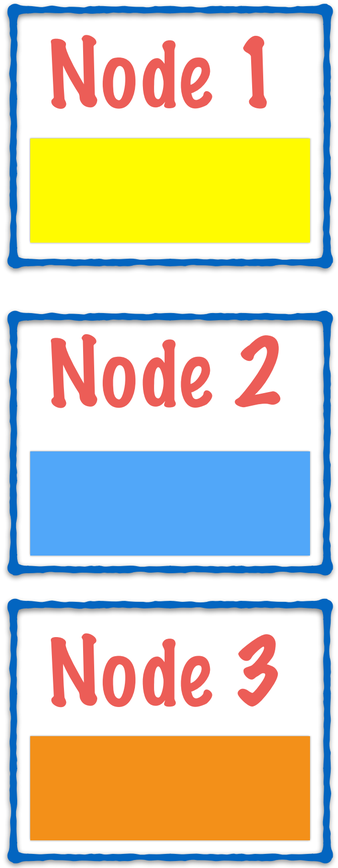
File 1	Block 1	Master	DN 1
File 1	Block 1	Replica	DN 2
..	..	..	..
..	..	..	..
..	..	..	..

The replica locations are also stored in the name node

RDDs partitions

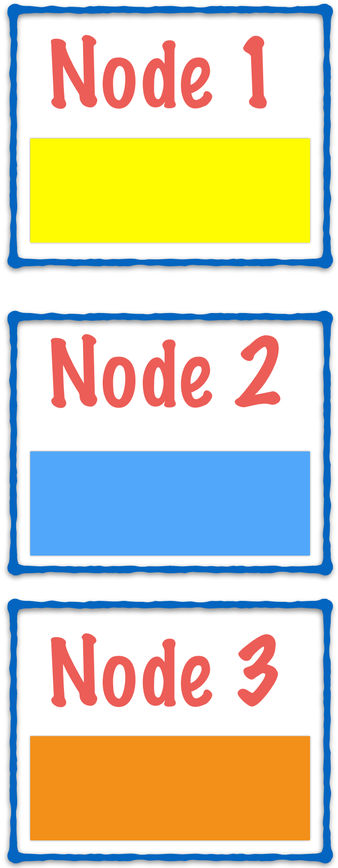
If Spark is reading data from HDFS
the data is already partitioned!

The file blocks in HDFS
are the partitions



RDDs partitions

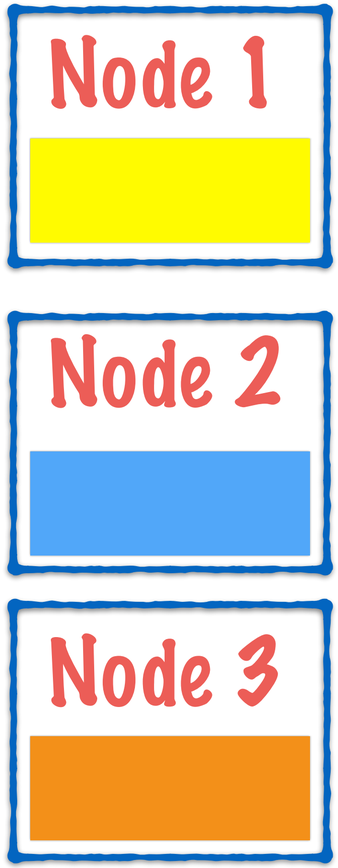
Each block is loaded into memory in the same node where it is stored



RDDs partitions

If the same node doesn't have enough free memory, then another node is chosen

The choice is made such that network transfer time for the data is minimized



Resilient Distributed Datasets

partitions

read-only

lineage

RDDs read-only

RDDs are immutable

RDDs read-only

Once an RDD is created, you can only do 2 things

Read data from it

Action

Transform it to another RDD

Transformation

Resilient Distributed Datasets

partitions

read-only

lineage

Recap

RDDs lineage

All operations on RDDs are either Transformations or Actions

Recap

RDDs lineage

When created, an RDD just holds metadata

1. A transformation
2. It's parent RDD

AirlineFiltered RDD

Recap

RDDs lineage

AirlineFiltered RDD

filter

airlines

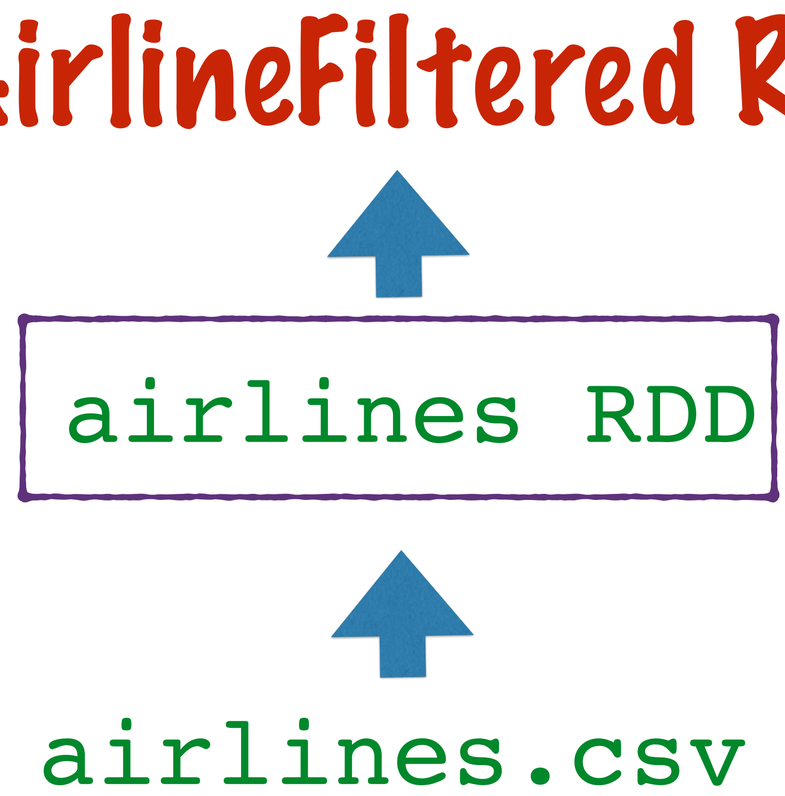
This way, every RDD knows where it came from

ie. RDDs know their lineage

Recap

RDDs lineage

This lineage can be traced back to the source (in this case, the original file that held the data)



RDDs lineage

lineage

is a really powerful concept!

lineage is the thing that

1. makes RDDs Resilient

2. makes lazy evaluation possible

Resilient

One of the key challenges
in distributed computing

How do you recover data if
one of the nodes crashes?

Resilient

In Hadoop for instance, this is managed by

A. Replication in HDFS

B. Writing data to disk
after every operation

Resilient

So, how do RDDs:

being in-memory

provide fault tolerance?

lineage

Resilient

lineage

Since RDDs know their lineage,
they can always be reconstructed
from the source

RDDs lineage

lineage is the thing that

1. makes RDDs Resilient

2. makes Lazy evaluation possible

Lazy evaluation

The other advantage of lineage is that it allows RDDs to be lazily evaluated

Lazy evaluation

When created, RDDs are not materialized

ie. they only consist of metadata

Lazy evaluation

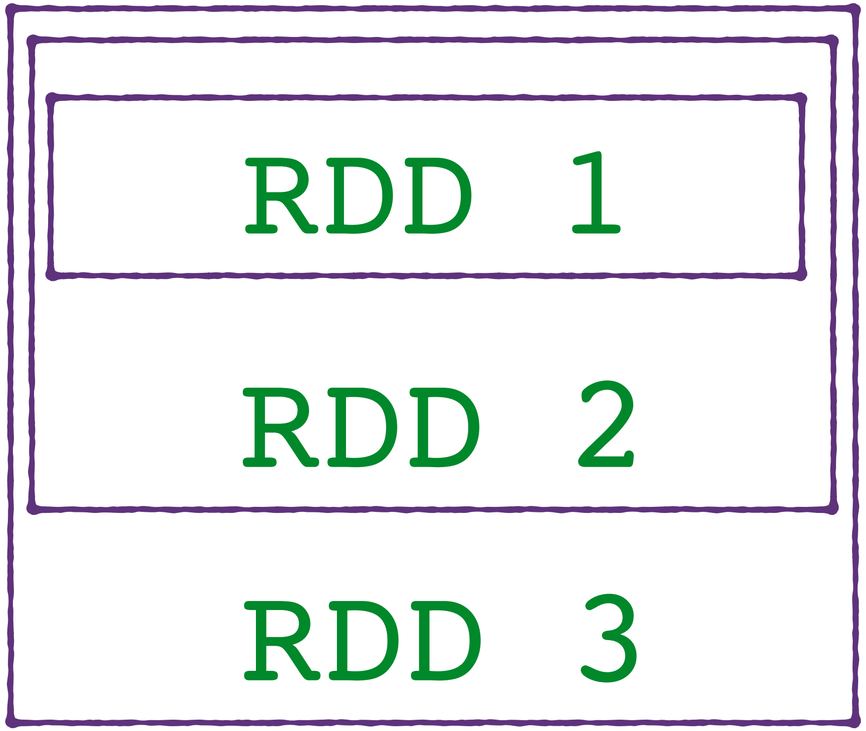
RDDs are only materialized when user requests a result using Actions

Lazy evaluation

Say you read data from a file

Filter the header row

Split the row by commas

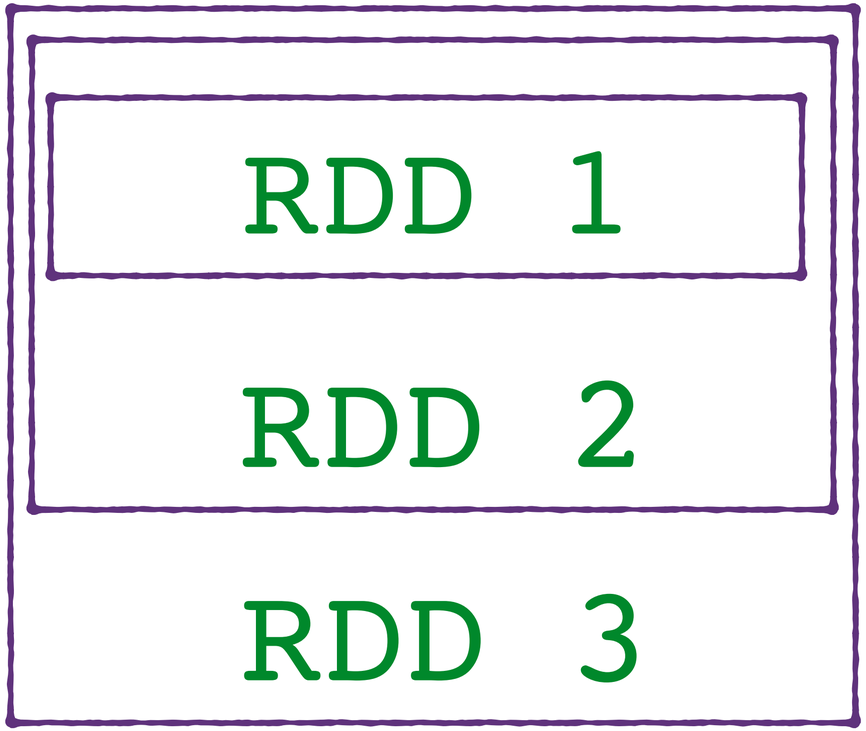


3 RDDs are created, none are materialized

Lazy evaluation

3 RDDs are created,
none are materialized

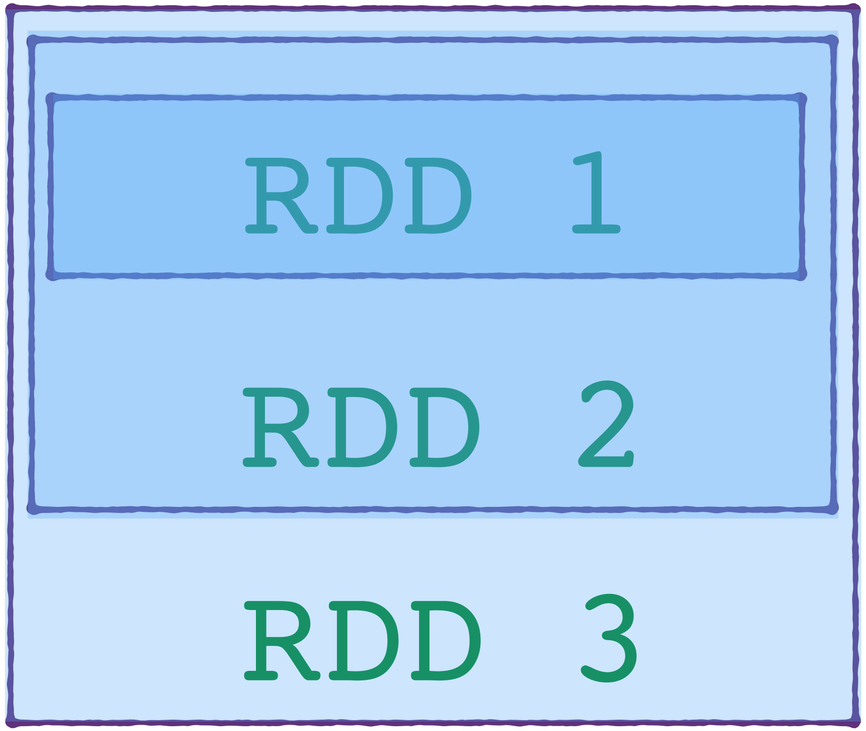
User asks for first
10 rows from RDD 3



Lazy evaluation

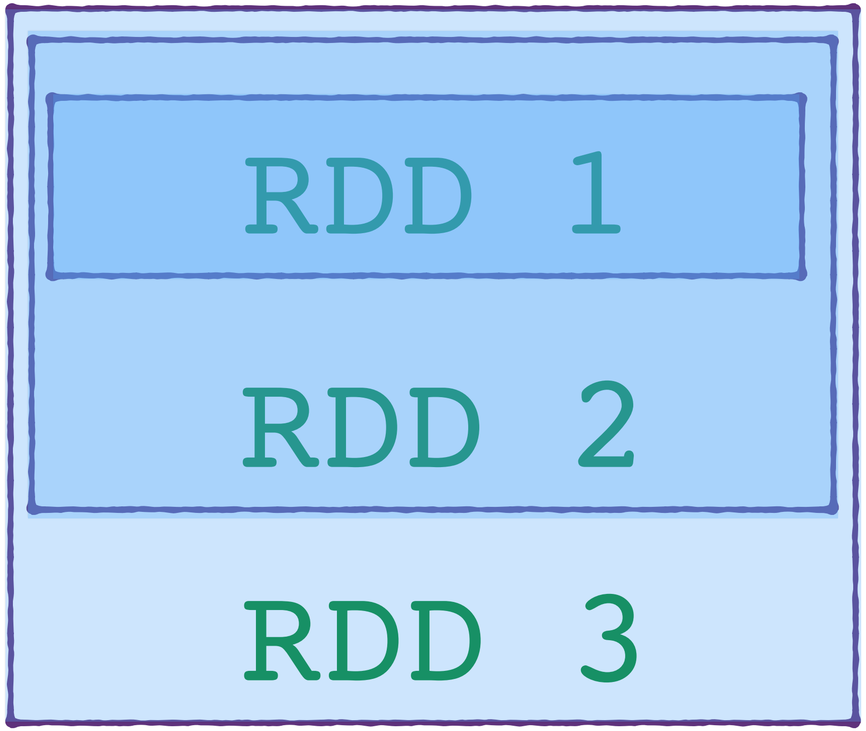
User asks for first 10 rows from RDD 3

Spark will now materialize all it's parent RDDs



Lazy evaluation

Lazy evaluation makes Spark very efficient



RDDs lineage

lineage is the thing that

1. makes RDDs Resilient

2. makes Lazy evaluation possible

Resilient Distributed Datasets

partitions

read-only

lineage

Things you can do with RDDs

Things you can do with RDDs

Create an RDD

Transformations

Actions

Persistence

Things you can do with RDDs

Create an RDD

Transformations

Actions

Persistence

Create an RDD

You can create an RDD in 2 ways

From a data source (usually on disk)

```
airlines=sc.textFile(airlinesPath)
```

From another RDD (using Transformations)

```
airlines.filter(lambda x:'Description' not in x)
```

Create an RDD

You can create an RDD in 2 ways

From a data source (usually on disk)

```
airlines=sc.textFile(airlinesPath)
```

The data source could also be a collection in your program

```
myList = [1,2,5,10]
myListRDD=sc.parallelize(myList)
```

Things you can do with RDDs

Create an RDD

Transformations

Actions

Persistence

Things you can do with RDDs

Create an RDD

Transformations

Actions

Persistence

Transformations

Transformations convert one RDD to another

```
airlines.filter(lambda x:'Description' not in x)
```

The most common transformations on an RDD are

filter

map

flatMap

Transformations

filter filters records from the RDD based on a condition

```
airlines.filter(lambda x:'Description' not in x)
```

map applies a function on each record

```
airlines.map(len)
```

flatMap takes each record and converts it to multiple records

Ex: from an RDD of lines of text to an RDD of words

Transformations

filter Takes a boolean function

map Any function that returns a single object

flatMap Any function that returns an iterator

Each of these take a **function** and **apply it on each** record in the RDD

Transformations

filter

map

flatMap

In addition to these, we have transformations that act on 2 RDDS

union

intersection

subtract

cartesian

Transformations

union union of 2 RDDs

intersection intersection of 2 RDDs

subtract Remove the contents of 1 RDD from the other

cartesian cartesian product : Creates an RDD of tuples

These act just like set operations

Transformations

Transformations are
lazily evaluated

ie. they are only executed
when an Action is performed

Things you can do with RDDs

Create an RDD

Transformations

Actions

Persistence

Things you can do with RDDs

Create an RDD

Transformations

Actions

Persistence

Actions compute a result from the RDD and return it to the user

```
# View a few lines
airlines.take(10)
```

```
[u'Code,Description',
 u'"19031","Mackey International Inc.: MAC"',
 u'"19032","Munz Northern Airlines Inc.: XY"',
 u'"19033","Cochise Airlines Inc.: COC"',
 u'"19034","Golden Gate Airlines Inc.: GSA"',
 u'"19035","Aeromech Inc.: RZZ"',
 u'"19036","Golden West Airlines Co.: GLW"',
 u'"19037","Puerto Rico Intl Airlines: PRN"',
 u'"19038","Air America Inc.: STZ"',
 u'"19039","Swift Aire Lines Inc.: SWT"']
```

Actions

collect
first
take
reduce
aggregate
count
countByValue

are some commonly used actions

Actions

first

take

collect

count

countByValue

reduce

aggregate

Collect will return all the elements of the RDD

Actions

first

take

count

countByValue

reduce

aggregate

collect

We usually don't use as it may overwhelm the system..

.....unless we know that the entire dataset is small enough fit onto 1 machine

collect

countByValue

reduce

aggregate

Actions

first The first row

take A specified number of rows

count The total number of rows

These are useful to get a quick sense of the data

collect

first

take

count

reduce

aggregate

Actions

countByValue

This will count the number of times a value appears in the RDD

collect

first

take

count

reduce

aggregate

Actions

countByValue

It can be used to summarize a dataset/ build a histogram

collect

first

take

count

countByValue

aggregate

Actions

reduce

This is one of the most common actions

collect

first

take

count

countByValue

aggregate

Actions

reduce

It will combine all the values in the RDD in a specified manner

collect

first

take

count

countByValue

aggregate

Actions

reduce

We can use this for instance to sum up all the values in an RDD of numbers

collect

first

take

count

countByValue

aggregate

Actions

reduce

Reduce takes a function
that acts on 2 elements
.. and returns another
element of the same type

collect

first

take

count

countByValue

reduce

Actions

aggregate

Similar to reduce

reduce will only take functions that return the same type as the RDD elements

Ex. Return a number using an RDD of numbers

collect

first

take

count

countByValue

reduce

Actions

aggregate

aggregate allows you to return elements of a different type

Ex. take an RDD of numbers and return a tuple

collect

first

take

count

countByValue

reduce

Actions

aggregate

We'll see some examples of reduce and aggregate later

Things you can do with RDDs

Create an RDD

Transformations

Actions

Persistence

Things you can do with RDDs

Create an RDD

Transformations

Actions

Persistence

Persistence

Let's say there is an RDD
that you need to use a lot

It will be **materialized**
from scratch every time
you perform an action on it

Persistence

Instead, you could persist the RDD in memory

This will force the RDD to be materialized

Things you can do with RDDs

Create an RDD

Transformations

Actions

Persistence

Let's see all of this in action

Exploring Airline delays data with PySpark

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
airlinesPath="hdfs:///user/swethakolalapudi/flightDelayData/airlines.csv"
airportsPath="hdfs:///user/swethakolalapudi/flightDelayData/airports.csv"
flightsPath="hdfs:///user/swethakolalapudi/flightDelayData/flights.csv"
```

airlines.csv
airports.csv
flights.csv

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

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

This loads the flights data from file to an RDD

```
flights
```

```
hdfs:///user/swethakolalapudi/flightDelaysData/flights.csv MapPartitionsRDD[7] at textFile at NativeMethodAccessorImpl.java:-2
```

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

This loads the flights data from file to an RDD

```
i/flightDelaysData/flights.csv MapPartitionsRDD[7] at textFile at NativeMetho
```

Let's get a quick sense of the data

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

```
476881
```

```
# The first 10 rows
flights.take(10)
```

```
[u'2014-04-01,19805,1,JFK,LAX,0854,-6.00,1217,2.00,355.00,2475.00',
 u'2014-04-01,19805,2,LAX,JFK,0944,14.00,1736,-29.00,269.00,2475.00',
 u'2014-04-01,19805,3,JFK,LAX,1224,-6.00,1614,39.00,371.00,2475.00',
 u'2014-04-01,19805,4,LAX,JFK,1240,25.00,2028,-27.00,264.00,2475.00',
 u'2014-04-01,19805,5,DFW,HNL,1300,-5.00,1650,15.00,510.00,3784.00',
 u'2014-04-01,19805,6,OGG,DFW,1901,126.00,0640,95.00,385.00,3711.00',
 u'2014-04-01,19805,7,DFW,OGG,1410,125.00,1743,138.00,497.00,3711.00',
 u'2014-04-01,19805,8,HNL,DFW,1659,4.00,0458,-22.00,398.00,3784.00',
 u'2014-04-01,19805,9,JFK,LAX,0648,-7.00,1029,19.00,365.00,2475.00',
 u'2014-04-01,19805,10,LAX,JFK,2156,21.00,0556,1.00,265.00,2475.00']
```

These are examples of Actions

Each row here has

```
[u'2014-04-01,19805,1,JFK,LAX,0854,-6.00,1217,2.00,355.00,2475.00',
 u'2014-04-01,19805,2,LAX,JFK,0944,14.00,1736,-29.00,269.00,2475.00',
 u'2014-04-01,19805,3,JFK,LAX,1224,-6.00,1614,39.00,371.00,2475.00',
 u'2014-04-01,19805,4,LAX,JFK,1240,25.00,2028,-27.00,264.00,2475.00',
 u'2014-04-01,19805,5,DFW,HNL,1300,-5.00,1650,15.00,510.00,3784.00',
 u'2014-04-01,19805,6,OGG,DFW,1901,126.00,0640,95.00,385.00,3711.00',
 u'2014-04-01,19805,7,DFW,OGG,1410,125.00,1743,138.00,497.00,3711.00',
 u'2014-04-01,19805,8,HNL,DFW,1659,4.00,0458,-22.00,398.00,3784.00',
 u'2014-04-01,19805,9,JFK,LAX,0648,-7.00,1029,19.00,365.00,2475.00',
 u'2014-04-01,19805,10,LAX,JFK,2156,21.00,0556,1.00,265.00,2475.00']
```

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

```
[u'2014-04-01,19805,1,JFK,LAX,0854,-6.00,1217,2.00,355.00,2475.00',
 u'2014-04-01,19805,2,LAX,JFK,0944,14.00,1736,-29.00,269.00,2475.00',
 u'2014-04-01,19805,3,JFK,LAX,1224,-6.00,1614,39.00,371.00,2475.00',
 u'2014-04-01,19805,4,LAX,JFK,1240,25.00,2028,-27.00,264.00,2475.00',
 u'2014-04-01,19805,5,DFW,HNL,1300,-5.00,1650,15.00,510.00,3784.00',
 u'2014-04-01,19805,6,OGG,DFW,1901,126.00,0640,95.00,385.00,3711.00',
 u'2014-04-01,19805,7,DFW,OGG,1410,125.00,1743,138.00,497.00,3711.00',
 u'2014-04-01,19805,8,HNL,DFW,1659,4.00,0458,-22.00,398.00,3784.00',
 u'2014-04-01,19805,9,JFK,LAX,0648,-7.00,1029,19.00,365.00,2475.00',
 u'2014-04-01,19805,10,LAX,JFK,2156,21.00,0556,1.00,265.00,2475.00']
```

We can take this RDD and parse each row into a list

- 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 a list

```
# Split the row into a list
flightsParsed=flights.map(lambda x: x.split(','))
```

The map operation is a Transformation

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

```
# Split the row into a list
flightsParsed=flights.map(lambda x: x.split(','))
```

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

```
# Split the row into a list
flightsParsed=flights.map(lambda x: x.split(','))
```

In this case, it will parse each row into a list of the values in the row

```
# Split the row into a list
flightsParsed=flights.map(lambda x: x.split(','))
```

flightsParsed is another RDD, but in this RDD, each row is a list instead of a string

```
# Split the row into a list
flightsParsed=flights.map(lambda x: x.split(','))
```

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

The list representing each record could be processed further

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

```
# Split the row into a list
flightsParsed=flights.map(lambda x: x.split(','))
```

Then we'll convert each list in flightsParsed to this class

Here's the code that will do all of this

```
from datetime import datetime
from collections import namedtuple

fields   = ('date', 'airline', 'flightnum', 'origin', 'dest', 'dep',
            'dep_delay', 'arv', 'arv_delay', 'airtime', 'distance')

Flight   = namedtuple('Flight', fields, verbose=True)

DATE_FMT = "%Y-%m-%d"

TIME_FMT = "%H%M"


def parse(row):
    row[0]  = datetime.strptime(
    row[5]  = datetime.strptime(
    row[6]  = float(row[6])
    row[7]  = datetime.strptime(
    row[8]  = float(row[8])
    row[9]  = float(row[9])
    row[10] = float(row[10])
    return Flight(*row[:11])
```

These are the field names we want to use

```python
from datetime import datetime
from collections import namedtuple

fields   = ('date', 'airline', 'flightnum', 'origin', 'dest', 'dep',
            'dep_delay', 'arv', 'arv_delay', 'airtime', 'distance')

Flight   = namedtuple('Flight', fields, verbose=True)

DATE_FMT = "%Y-%m-%d"

TIME_FMT = "%H%M"


def parse(row):
    row[0]  = datetime.strptime(row
    row[5]  = datetime.strptime(row
    row[6]  = float(row[6])
    row[7]  = datetime.strptime(row
    row[8]  = float(row[8])
    row[9]  = float(row[9])
    row[10] = float(row[10])
    return Flight(*row[:11])
```

namedtuple is a function that can create a class called Flight

```
from datetime import datetime
from collections import namedtuple

fields   = ('date', 'airline', 'flightnum', 'origin', 'dest', 'dep',
            'dep_delay', 'arv', 'arv_delay', 'airtime', 'distance')

Flight   = namedtuple('Flight', fields, verbose=True)

DATE

TIME

def
```

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

    __slots__ = ()

    _fields = ('date', 'airline', 'flightnum', 'origin', 'dest', 'dep', 'dep_delay', 'arv', 'arv_delay', 'air
 'distance')

    def __new__(_cls, date, airline, flightnum, origin, dest, dep, dep_delay, arv, arv_delay, airtime, distan
        'Create new instance of Flight(date, airline, flightnum, origin, dest, dep, dep_delay, arv, arv_delay
ime, distance)'
        return _tuple.__new__(_cls, (date, airline, flightnum, origin, dest, dep, dep_delay, arv, arv_delay,
e, distance))

    @classmethod
    def _make(cls, iterable, new=tuple.__new__, len=len):
        'Make a new Flight object from a sequence or iterable'
        result = new(cls, iterable)
        if len(result) != 11:
```

```
from datetime import datetime
from collections import namedtuple

fields   = ('date', 'airline', 'flightnum', 'origin', 'dest', 'dep',
            'dep_delay', 'arv', 'arv_delay', 'airtime', 'distance')

Flight   = namedtuple('Flight', fields, verbose=True)

DATE_FMT = "%Y-%m-%d"

TIME_FMT = "%H%M"


def parse(row):
    row[0]  = datetime.strptime(row
    row[5]  = datetime.strptime(row
    row[6]  = float(row[6])
    row[7]  = datetime.strptime(row
    row[8]  = float(row[8])
    row[9]  = float(row[9])
    row[10] = float(row[10])
    return Flight(*row[:11])
```

The class will be set up with member variables based on the fields tuple

```
from datetime import datetime
from collections import namedtuple

fields   = ('date', 'airline', 'flightnum', 'origin', 'dest', 'dep',
            'dep_delay', 'arv', 'arv_delay', 'airtime', 'distance')

Flight   = namedtuple('Flight', fields, verbose=True)

DATE_FMT = "%Y-%m-%d"

TIME_FMT = "%H%M"


def parse(row):
    row[0]  = datetime.strptime(row
    row[5]  = datetime.strptime(row
    row[6]  = float(row[6])
    row[7]  = datetime.strptime(row
    row[8]  = float(row[8])
    row[9]  = float(row[9])
    row[10] = float(row[10])
    return Flight(*row[:11])
```

We've used namedtuple to manufacture a class

```python
from datetime import datetime
from collections import namedtuple

fields   = ('date', 'airline', 'flightnum', 'origin', 'dest', 'dep',
            'dep_delay', 'arv', 'arv_delay', 'airtime', 'distance')

Flight   = namedtuple('Flight', fields, verbose=True)

DATE_FMT = "%Y-%m-%d"

TIME_FMT = "%H%M"


def parse(row):
    row[0]  = datetime.strptime(row
    row[5]  = datetime.strptime(row
    row[6]  = float(row[6])
    row[7]  = datetime.strptime(row
    row[8]  = float(row[8])
    row[9]  = float(row[9])
    row[10] = float(row[10])
    return Flight(*row[:11])
```

Such functions are called factory functions

```
from datetime import datetime
from collections import namedtuple

fields   = ('date', 'airline', 'flight
            'dep_delay', 'arv', 'arv_d

Flight   = namedtuple('Flight', fields

DATE_FMT = "%Y-%m-%d"

TIME_FMT = "%H%M"
```

```
def parse(row):
    row[0]  = datetime.strptime(row[0], DATE_FMT).date()
    row[5]  = datetime.strptime(row[5], TIME_FMT).time()
    row[6]  = float(row[6])
    row[7]  = datetime.strptime(row[7], TIME_FMT).time()
    row[8]  = float(row[8])
    row[9]  = float(row[9])
    row[10] = float(row[10])
    return Flight(*row[:11])
```

Now we have function to parse the row list and return a Flight object

```
from datetime import datetime
from collections import namedtuple

fields   = ('date', 'airline', 'flight
            'dep_delay', 'arv', 'arv_d

Flight   = namedtuple('Flight', fields

DATE_FMT = "%Y-%m-%d"

TIME_FMT = "%H%M"
```

```
def parse(row):
    row[0]  = datetime.strptime(row[0], DATE_FMT).date()
    row[5]  = datetime.strptime(row[5], TIME_FMT).time()
    row[6]  = float(row[6])
    row[7]  = datetime.strptime(row[7], TIME_FMT).time()
    row[8]  = float(row[8])
    row[9]  = float(row[9])
    row[10] = float(row[10])
    return Flight(*row[:11])
```

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

```
from datetime import datetime
from collections import namedtuple

fields   = ('date', 'airline', 'flight
            'dep_delay', 'arv', 'arv_d

Flight   = namedtuple('Flight', fields

DATE_FMT = "%Y-%m-%d"

TIME_FMT = "%H%M"
```

```
def parse(row):
    row[0]  = datetime.strptime(row[0], DATE_FMT).date()
    row[5]  = datetime.strptime(row[5], TIME_FMT).time()
    row[6]  = float(row[6])
    row[7]  = datetime.strptime(row[7], TIME_FMT).time()
    row[8]  = float(row[8])
    row[9]  = float(row[9])
    row[10] = float(row[10])
    return Flight(*row[:11])
```

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

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

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

Returns an RDD with a list which holds the details of one flight

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

Apply a map transformation to the list RDD

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

Get an RDD with a Flight object as every record

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

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

```
# Let's take a look at the parsed dataset
flightsParsed.first()
```

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

```
# Let's take a look at the parsed dataset
flightsParsed.first()
```

```
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)
```

Each record is now represented by a Flight object

```
# Let's take a look at the parsed dataset
flightsParsed.first()
```

```
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)
```

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

```
flightsParsed.map(lambda x:x.distance)
```

This will create an RDD which only has the distance field

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

Let's parse what happened here

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

We passed a function to map

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

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

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

The function uses the class definition for Flight

```
class Flight(tuple):
    'Flight(date, airline, flightnum, origin, dest, dep, dep_delay, arv, arv_delay, airtime, dista

    __slots__ = ()

    _fields = ('date', 'airline', 'flightnum', 'origin', 'dest', 'dep', 'dep_delay', 'arv', 'arv_d
 'distance')

    def __new__(_cls, date, airline, flightnum, origin, dest, dep, dep_delay, arv, arv_delay, airt
        'Create new instance of Flight(date, airline, flightnum, origin, dest, dep, dep_delay, arv
ime, distance)'
        return _tuple.__new__(_cls, (date, airline, flightnum, origin, dest, dep, dep_delay, arv,
e, distance))

    @classmethod
```

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

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

```
class Flight(tuple):
    'Flight(date, airline, flightnum, origin, dest, dep, dep_delay, arv, arv_delay, airtime, di

    __slots__ = ()

    _fields = ('date', 'airline', 'flightnum', 'origin', 'dest', 'dep', 'dep_delay', 'arv', 'ar
 'distance')

    def __new__(_cls, date, airline, flightnum, origin, dest, dep, dep_delay, arv, arv_delay, a
        'Create new instance of Flight(date, airline, flightnum, origin, dest, dep, dep_delay,
ime, distance)'
        return _tuple.__new__(_cls, (date, airline, flightnum, origin, dest, dep, dep_delay, ar
```

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

Such functions are called closure functions

Spark is built on Scala, which supports the use of closure functions

Spark is built on Scala, which supports the use of closure functions

We won't go further into closures here

Spark is built on Scala, which supports the use of closure functions

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

Spark is built on Scala, which supports the use of closure functions

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

Spark is built on Scala, which supports the use of closure functions

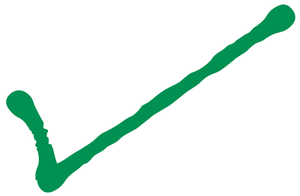
All you need to do is use the map operation!

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

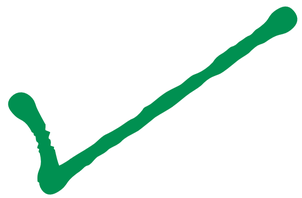
2. Compute the average distance travelled by a flight

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

```
totalDistance=flightsParsed.map(lambda x:x.distance).reduce(lambda x,y:x+y)
```

This extracts the distance field

The average distance travelled by a flight

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

Then we just need to divide it by the count of total number of flights

```
totalDistance=flightsParsed.map(lambda x:x.distance).reduce(lambda x,y:x+y)
```

This extracts the distance field

The average distance travelled by a flight

```
totalDistance=flightsParsed.map(lambda x:x.distance).reduce(lambda x,y:x+y)
```

This will sum up all the values in the field

The average distance travelled by a flight

```
totalDistance=flightsParsed.map(lambda x:x.distance).reduce(lambda x,y:x+y)
```

reduce is an Action

The average distance travelled by a flight

```
totalDistance=flightsParsed.map(lambda x:x.distance).reduce(lambda x,y:x+y)
```

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

The average distance travelled by a flight

```
totalDistance=flightsParsed.map(lambda x:x.distance).reduce(lambda 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

```
totalDistance=flightsParsed.map(lambda x:x.distance).reduce(lambda x,y:x+y)
```

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

Let's see this visually

```
totalDistance=flightsParsed.map(lambda x:x.distance).reduce(lambda x,y:x+y)
```

This is the distance RDD, partitioned among 3 nodes

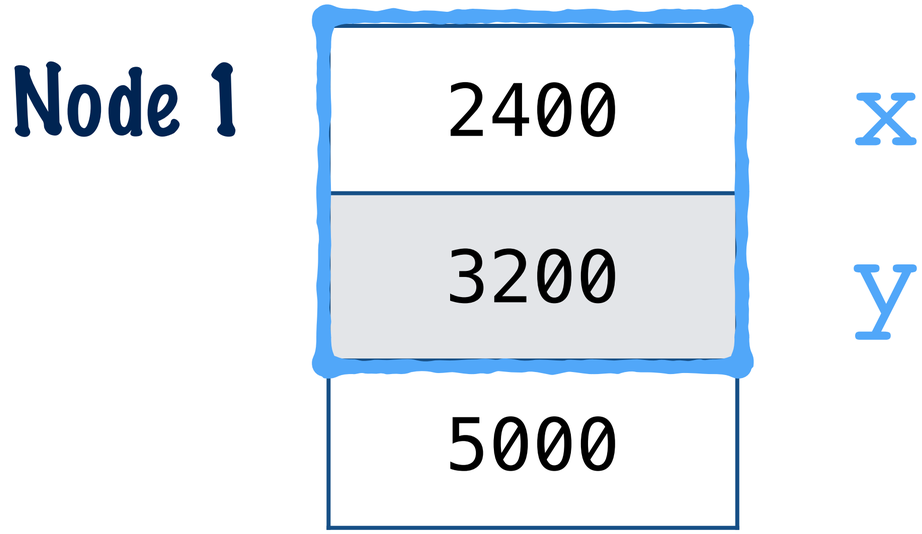
Node 1
2400
3200
5000

Node 2
2230
5400
4900

Node 3
4300
3600
2100

First, the reduce operation is performed on each node

```
totalDistance=flightsParsed.map(lambda x:x.distance).reduce(lambda x,y:x+y)
```

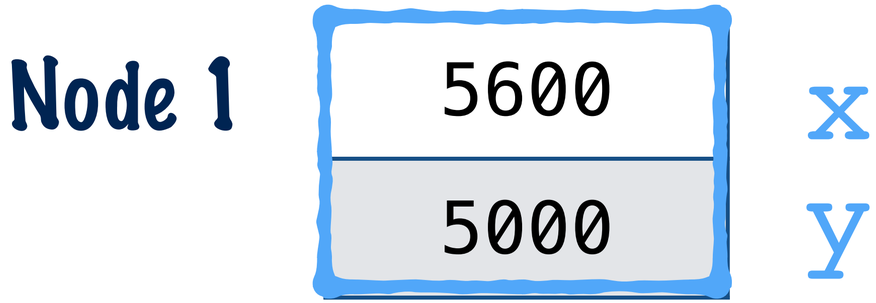


The function will start with the first 2 elements

```
totalDistance=flightsParsed.map(lambda x:x.distance).reduce(lambda x,y:x+y)
```

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

```
totalDistance=flightsParsed.map(lambda x:x.distance).reduce(lambda x,y:x+y)
```



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

```
totalDistance=flightsParsed.map(lambda x:x.distance).reduce(lambda x,y:x+y)
```

Node 1 | 10600 | x + y

The same thing is done in parallel on all the nodes

```
totalDistance=flightsParsed.map(lambda x:x.distance).reduce(lambda x,y:x+y)
```

Node 1 | 10600

Node 2 | 12530

Node 3 | 10000

The same thing is done in parallel on all the nodes

```
totalDistance=flightsParsed.map(lambda x:x.distance).reduce(lambda x,y:x+y)
```

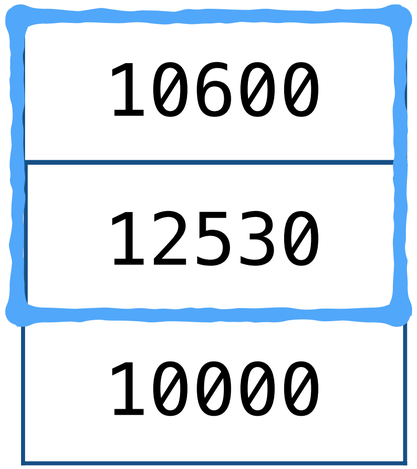
Node 1 | 10600

Node 2 | 12530

Node 3 | 10000

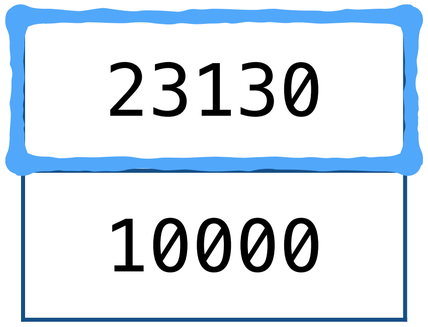
All the results from each node are brought over to a single node

```
totalDistance=flightsParsed.map(lambda x:x.distance).reduce(lambda x,y:x+y)
```



Again, reduce is applied iteratively on these results

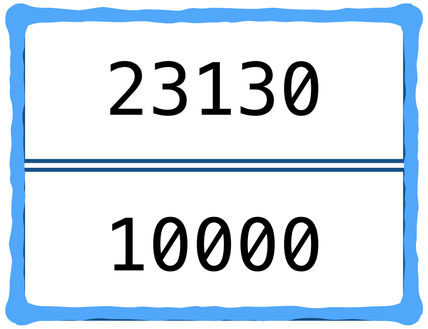
```
totalDistance=flightsParsed.map(lambda x:x.distance).reduce(lambda x,y:x+y)
```



x + y

Again, reduce is applied iteratively on these results

```
totalDistance=flightsParsed.map(lambda x:x.distance).reduce(lambda x,y:x+y)
```



Again, reduce is applied iteratively on these results

```
totalDistance=flightsParsed.map(lambda x:x.distance).reduce(lambda 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

```
totalDistance=flightsParsed.map(lambda x:x.distance).reduce(lambda x,y:x+y)
```

We need to divide this by the total number of flights

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

```
print 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(lambda x:x.dep_delay>0).count()
```

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

% of flights which had delays

```
flightsParsed.filter(lambda x:x.dep_delay>0).count()
```

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

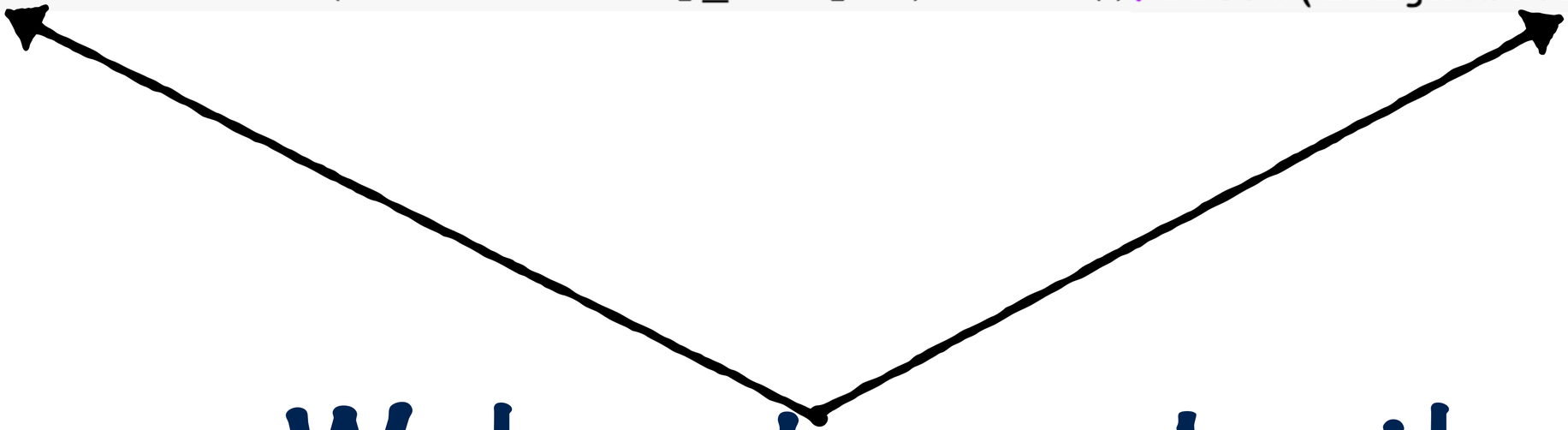
% of flights which had delays

```
flightsParsed.filter(lambda x:x.dep_delay>0).count()/float(flightsParsed.count())
```

We just need to divide by the total flight count

% of flights which had delays

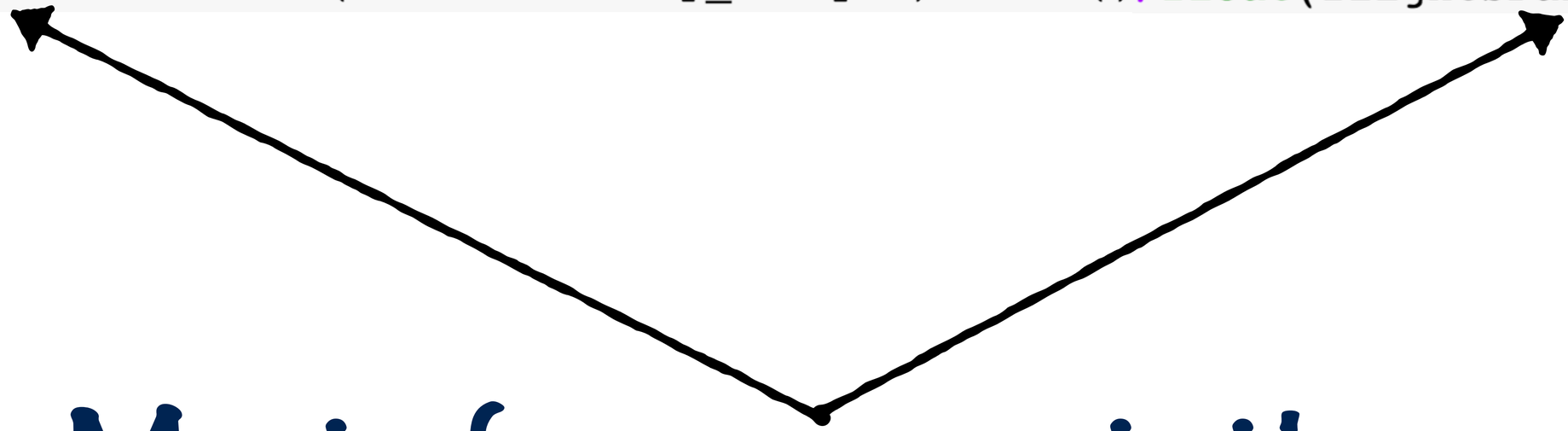
```
flightsParsed.filter(lambda x:x.dep_delay>0).count()/float(flightsParsed.count())
```



We have been using the flightsParsed RDD quite a lot

% of flights which had delays

```
flightsParsed.filter(lambda x:x.dep_delay>0).count()/float(flightsParsed.count())
```



Most of our computations need this RDD

% of flights which had delays

```
flightsParsed.filter(lambda x:x.dep_delay>0).count()/float(flightsParsed.count())
```

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

It is materialized again

% of flights which had delays

```
flightsParsed.filter(lambda x:x.dep_delay>0).count()/float(flightsParsed.count())
```

It is materialized again

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

% of flights which had delays

```
flightsParsed.filter(lambda x:x.dep_delay>0).count()/float(flightsParsed.count())
```

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

```
totalDistance=flightsParsed.map(lambda x:x.distance).reduce(lambda x,y:x+y)
avgDistance=totalDistance/flightsParsed.count()
```

We can just replace this with the delay field

Compute the average delay

```
totalDistance=flightsParsed.map(lambda x:x.distance).reduce(lambda x,y:x+y)
avgDistance=totalDistance/flightsParsed.count()
```

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

Compute the average delay

```
totalDistance=flightsParsed.map(lambda x:x.distance).reduce(lambda x,y:x+y)
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(lambda 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

```
sumCount=flightsParsed.map(lambda x:x.dep_delay).aggregate((0,0),
                                                           (lambda acc, value: (acc[0] + value, acc[1]+1)),
                                                           (lambda acc1,acc2: (acc1[0]+acc2[0],acc1[1]+acc2[1])))
```

Compute the average delay

aggregate

```
sumCount=flightsParsed.map(lambda x:x.dep_delay).aggr
```

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

Compute the average delay

aggregate

```
sumCount=flightsParsed.map(lambda x:x.dep_delay).aggr
```

The delay field

Compute the average delay

aggregate

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

The aggregate action

Compute the average delay

aggregate

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

The function to use on each node

Compute the average delay

aggregate

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

Calculate the sum portion of the tuple

Compute the average delay

aggregate

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

And the count portion of the tuple

Compute the average delay

aggregate

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

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

Compute the average delay

aggregate

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

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

Compute the average delay

aggregate

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

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

Compute the average delay

aggregate

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

The zero value

Compute the average delay

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

Let's see how this works visually

Compute the average delay

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

Node 1

14
5
10

Node 2

0
4
0

Node 3

3
15
7

This the delays RDD

Compute the average delay

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

Node 1

14
5
10

Node 2

0
4
0

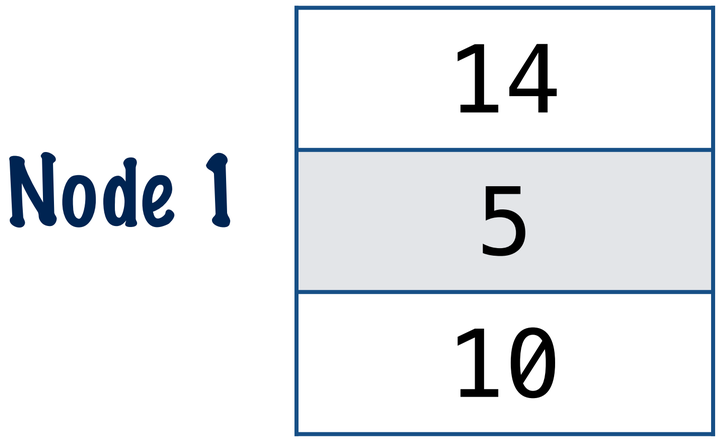
Node 3

3
15
7

First, the operation is performed on each node

Compute the average delay

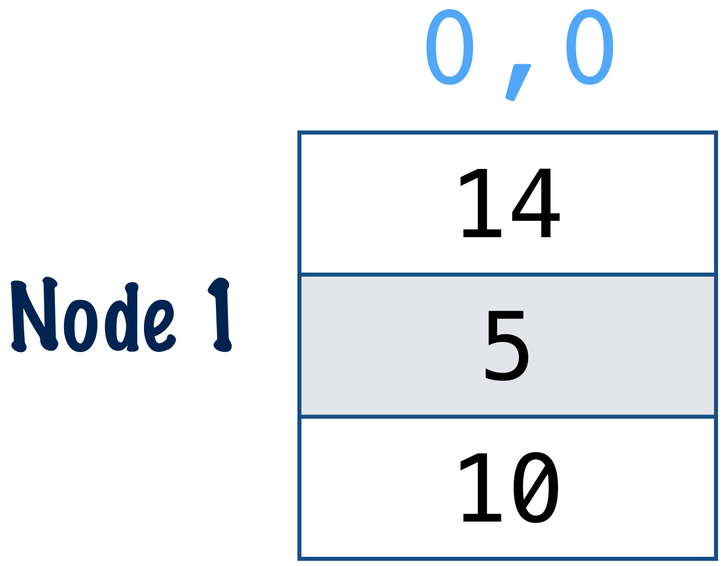
```
).aggregate((0,0),
            (lambda acc, value: (acc[0] + value, acc[1]+1)),
            (lambda acc1,acc2: (acc1[0]+acc2[0],acc1[1]+acc2[1])))
```



First, the operation is performed on each node

Compute the average delay

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

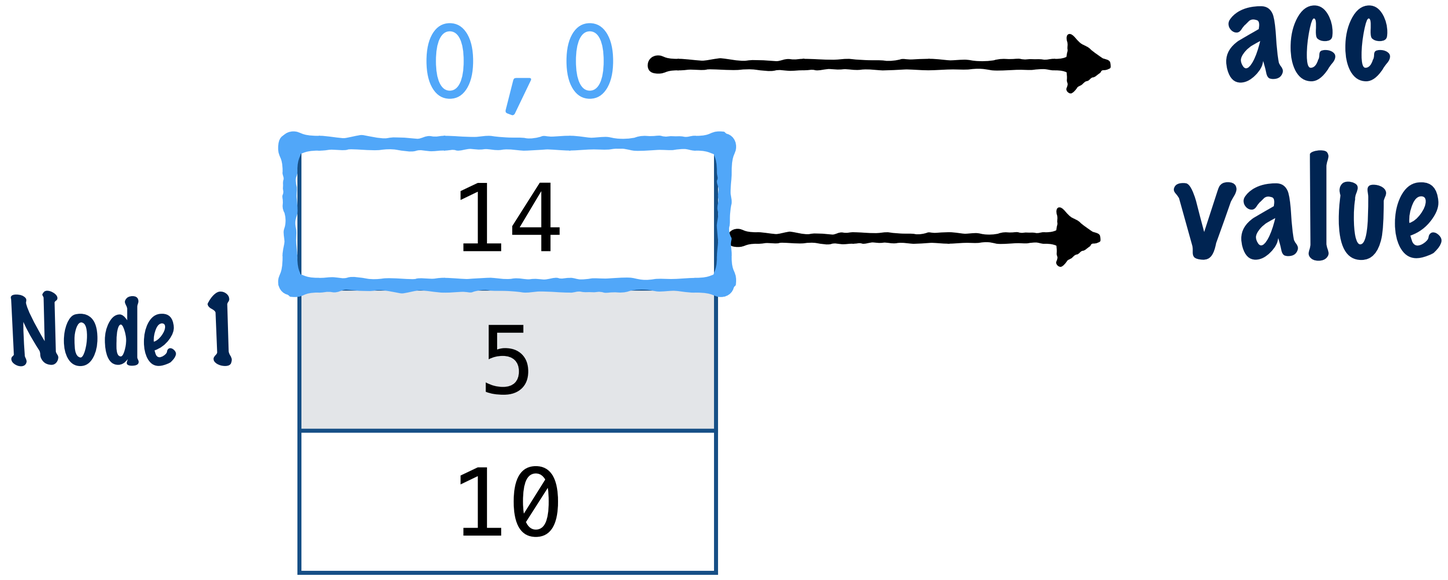


We start with the zero value

Compute the average delay

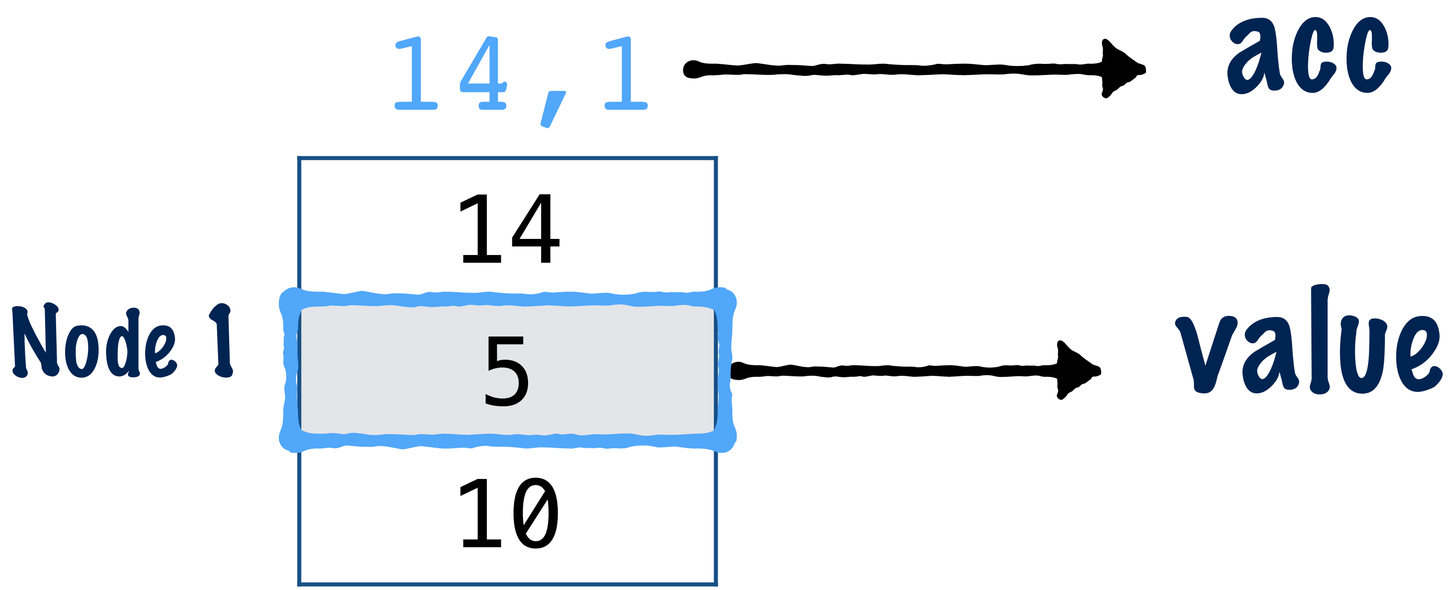
```
).aggregate((0,0),
            (lambda acc, value: (acc[0] + value, acc[1]+1)),
            (lambda acc1,acc2: (acc1[0]+acc2[0],acc1[1]+acc2[1])))
```

This function is applied iteratively on each value



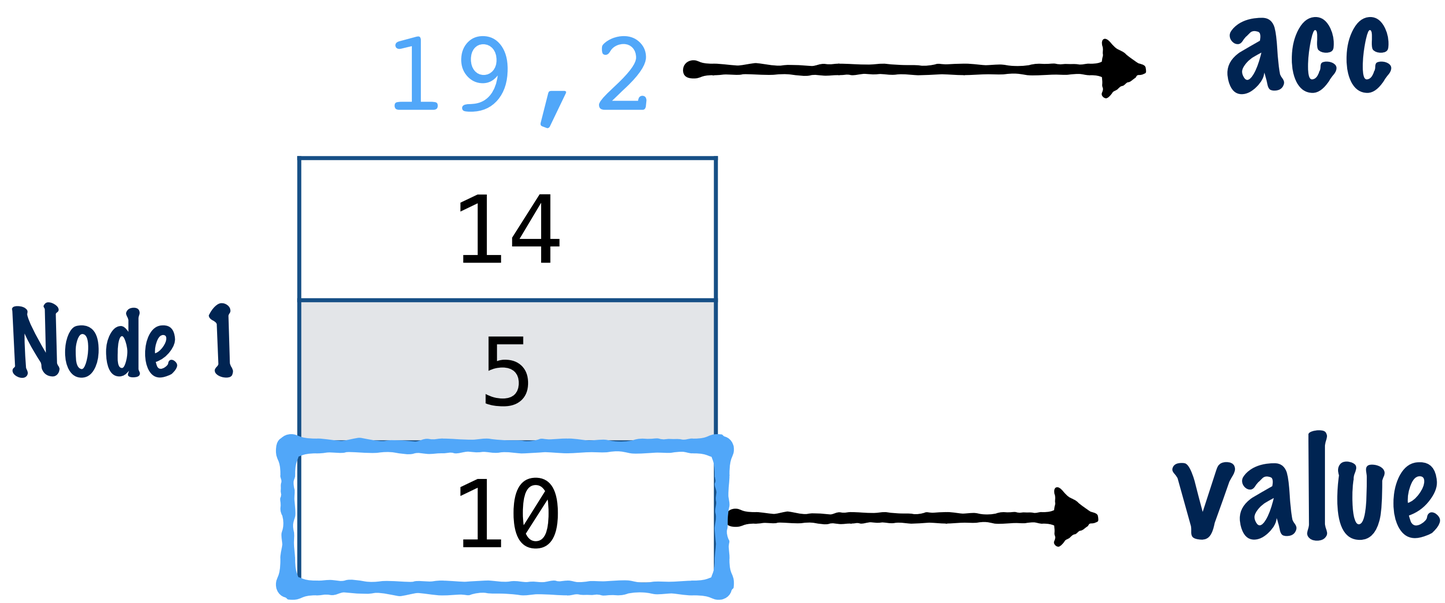
Compute the average delay

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



Compute the average delay

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



Compute the average delay

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

Node 1 29,3

The same thing is done on each node

Compute the average delay

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

Node 1 29,3

Node 2 4,3

Node 3 25,3

Now the second function is used to combine these tuples

Compute the average delay

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

0,0

Node 1 29,3

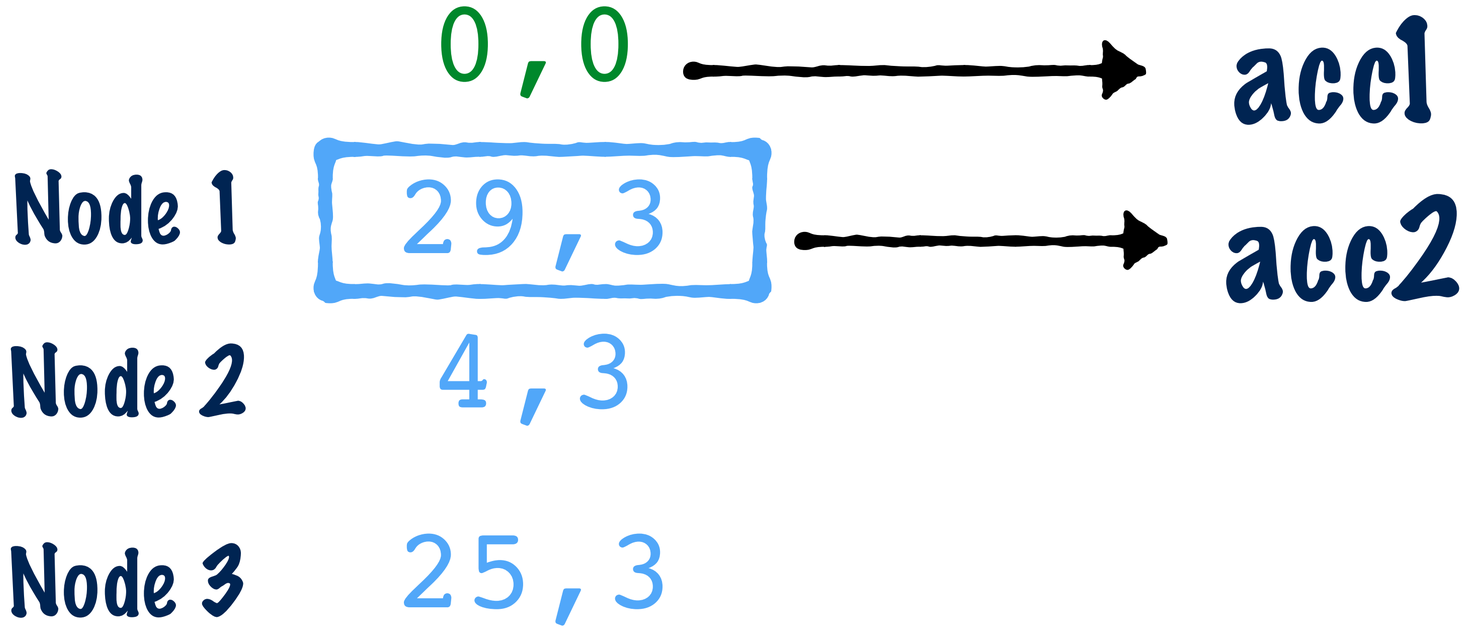
Node 2 4,3

Node 3 25,3

We'll start again with the zero value

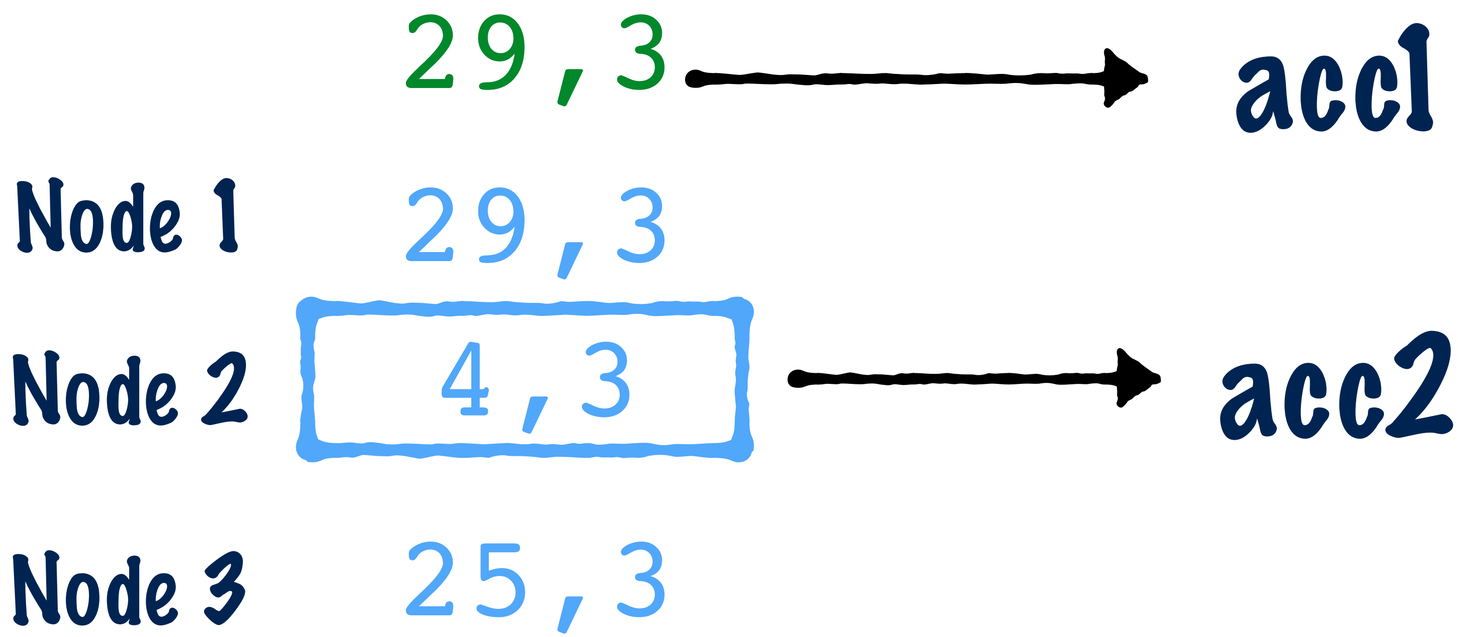
Compute the average delay

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



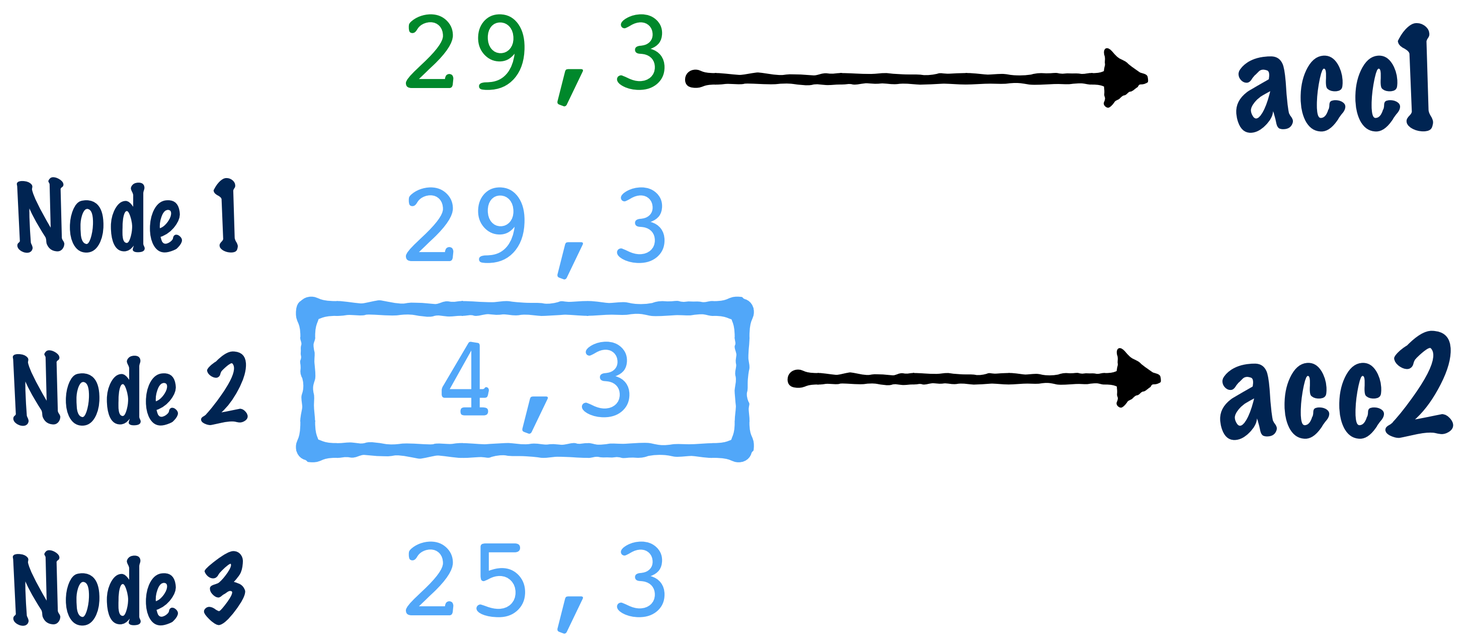
Compute the average delay

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



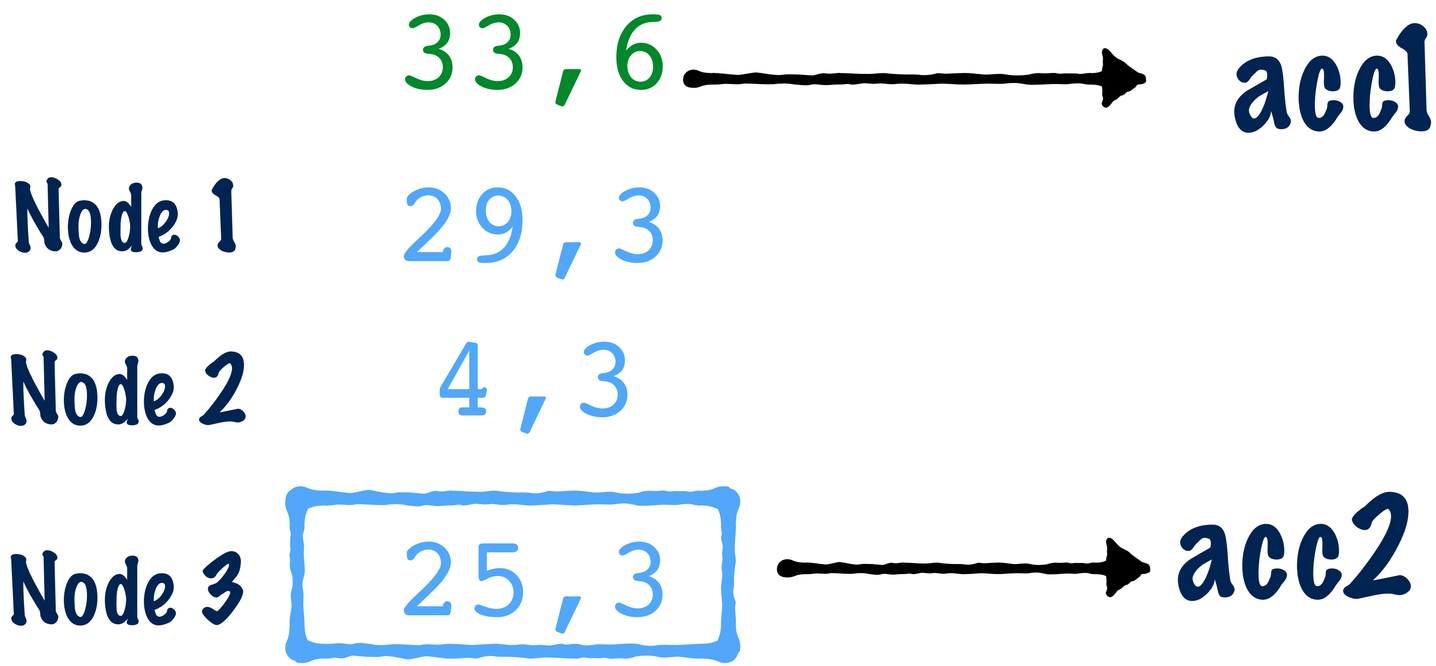
Compute the average delay

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



Compute the average delay

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



Compute the average delay

```
sumCount=flightsParsed.map(lambda x:x.dep_delay).aggreg
```

```
58,9
```

sum

count

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

Compute the average delay

```
sumCount=flightsParsed.map(lambda x:x.dep_delay).aggreg
```

```
sumCount[0]/float(sumCount[1])
```

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

Compute the average delay

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

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(lambda x: int(x.dep_delay/60)).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(lambda x: int(x.dep_delay/60)).countByValue()
```

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

This counts the number of times each value occurs

A frequency distribution

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

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

A frequency distribution

Num. Flights with Delay < 1hr

```
defaultdict(int,
            {0: 452963,
             1: 16016,
             2: 4893,
             3: 1729,
             4: 701,
             5: 249,
             6: 113,
             7: 66,
             8: 43,
             9: 26,
             10: 15,
             11: 12,
             12: 9,
             13: 15,
             14: 13,
             15: 4,
             17: 2,
```

A frequency distribution

Flights with delay between 1-2 hrs

..and so on

```
defaultdict(int,
            {0: 452963,
             1: 16016,
             2: 4893,
             3: 1729,
             4: 701,
             5: 249,
             6: 113,
             7: 66,
             8: 43,
             9: 26,
             10: 15,
             11: 12,
             12: 9,
             13: 15,
             14: 13,
             15: 4,
             17: 2,
```