# Introduction to Spark using Python

orlando.karam@gmail.com

Orlando.Karam@Microsoft.com

# Administrivia

- If you have a question, ask right away
- We're being recorded

# What is Spark

- Distributed data processing framework
    - Distributed – runs in several machines
        - We get more RAM, more processing power
    - Data processing
        - Read and process data
- Based on Resilient Distributed Datasets
- Used for ‘big data' processing
    - 'big' is something that doesn't fit on normal machines
        - Changes as machines become more powerful

# Resilient Distributed Datasets

- Imagine a big set of objects, and how we can distribute/parallelize
- We can divide in slices and keep each slice in a different node; this is the basic idea of an RDD
  - Values are computed only when needed
  - To guarantee fault-tolerance, we also keep info about how we calculated each slice, so we can re-generate it if a node fails
  - We can hint to keep in cache, or even save on disk
- Immutable ! not designed for read/write
  - instead, transform an existing one into a new one
- It is basically a huge list
  - But distributed over many computers

# Shared Spark Variables

- Broadcast variables
  - copy is kept at each node
- Accumulators
  - you can only add; main node can read

# Functional programming in python

- A lot of these concepts are already in python
  - But python community tends to promote loops
- Functional tools in python
  - map
  - filter
  - reduce
  - lambda
  - Itertools
    - Chain, flatmap

# Map in Python

- Python supports the map operation, over any list
- We apply an operation to each element of a list, return a new list with the results
  - a=[1,2,3]
  - def add1(x): return x+1
  - map(add1,a) => [2,3,4]
- We usually do this with a for loop, this is a slightly different way of thinking

# Filter

- Select only certain elements from a list
- Example:
    - a=[1,2,3,4]
    - def isOdd(x): return x%2==1;
    - filter(isOdd,a) => [1,3]

# reduce in python

- Applies a function to all pairs of elements of a list; returns ONE value, not a list
- Example:
  - a=[1,2,3,4]
  - def add(x,y): return x+y
  - reduce(add,a) => 10
    - add(1,add(2,add(3,4,)))
- Better for functions that are commutative and associative, so order doesn't matter

# lambdas

- When doing map/reduce/filter, we end up with many tiny functions
- Lambdas allow us to define a function as a value, without giving it a name
- example: lambda x: x+1
  - Can only have one expression
  - do not write return
  - I put parenthesis around it, usually not needed by syntax
- (lambda x: x+1)(3) => 4
- map(lambda x: x+1, [1,2,3])=> [2,3,4]

# Exercises

- (lambda x: 2*x)(3) => ?
- map(lambda x: 2*x, [1,2,3]) =>
- map(lambda t: t[0], [ (1,2), (3,4), (5,6) ] ) =>
- reduce(lambda x,y: x+y, [1,2,3]) =>
- reduce(lambda x,y: x+y, map(lambda t: t[0], [ (1,2), (3,4), (5,6) ] ))=>

# More exercises

- Given
  - a=[ (1,2), (3,4), (5,6)]
- Write an expression to get only the second elements of each tuple
- Write an expression to get the sum of the second elements
- Write an expression to get the sum of the odd first elements

# Flatmap

- Sometimes we end up with a list of lists, and we want a ‘flat’ list
- Many functional programming languages (and Spark) provide a function called flatMap, which flattens such a list
- Example:
  - Map(lambda t:range(t[0],t[1]), [ (1,5), (7,10)]) # returns list of lists
- Itertools.chain maps a list of iterables into a flat list
  - And so enables us to define our own flatmap

# Now let's do those with Spark

- Start the spark shell
  - run pyspark

# Creating RDDs in Spark

- All spark commands operate on RDDs (think big distributed list)
- You can use sc.parallelize to go from list to RDD
- Later we will see how to read from files
- Many commands are lazy (they don't actually compute the results until you need them)
- In pySpark, sc represents your SparkContext

# Simple example

- list1=sc.parallelize( range(1,1000) )
- list2=list1.map(lambda x: x*10) # notice lazy
- list2.reduce(lambda x,y: x+y)
- list2.filter(lambda x: x%100==0).collect()

# Transformations vs Actions

- We divide RDD methods into two kinds:
  - Transformations
    - return another RDD
    - are not really performed until an action is called (lazy)
  - Actions
    - return a value other than an RDD
    - are performed immediately

# Some RDD methods

- Transformations
  - .map( f ) – returns a new RDD applying f to each element
  - .filter( f ) – returns a new RDD containing elements that satisfy f
  - .flatmap(f) – returns a ‘flattened’ list
- Actions
  - .reduce( f ) – returns a value reducing RDD elements with f
  - .take( n ) – returns n items from the RDD
  - .collect() – returns all elements as a list
  - .sum() - sum of (numeric) elements of an RDD
    - max,min,mean …

# More examples

- rdd1=sc.parallelize( range(1,100) )
- rdd1.map(lambda x: x*x).sum()
- rdd1.filter(lambda x: x%2==0).take(5)

# Exercises

1. Get an RDD with number 1 to 10
2. Get all the elements in that RDD which are divisible by 3
3. Get the product of the elements in 2

# Reading files

- sc.textFile(urlOrPath,minPartitions,useUnicode=True)
  - Returns an rdd of strings (one per line)
  - Can read from many files, using wildcards (*)
  - Can read from hdfs, …
  - We normally use map right after and split/parse the lines
- Example:
  - people=sc.textFile(”../data/people.txt”)
  - people=sc.textFile(”../data/people.txt”).map(lambda  x: x.split(’\t’)

# Tuples and ReduceByKey

- Many times we want to group elements first, and then calculate values for each group
- In spark, we operate on tuples, <Key,Value> and we normally use reduceByKey to perform a reduce on the elements of each group

# People example/Exercises

- We have a people.txt file with following schema:
  - Name | Gender | Age | Favorite Language
- We can load with:
  - people=sc.textFile("../data/people.txt").map(lambda x: x.split('\t'))
- Find number of people by gender
  - first get tuples like: ('M',1),('F',1) ... then reduce by key
  - people.map(lambda t: (t[1],1)).reduceByKey(lambda x,y:x+y).collect()
- Let's find number of people by favorite programming language
- Example: youngest person per gender
  - people.map(lambda t: (t[3],int(t[2]) )).reduceByKey(lambda x,y:min(x,y)).collect()

# More people exercises

- Get number of people with age 40+
  - Using filter
  - Using map and reduceByKey to produce two groups <40, 40+

# Person example with objects

- Using tuples for everything is … *ok*, but sometimes we want nicer schema
  - We can use regular python objects
  - We still need to use tuples for joins, reduceByKey, since they operate on tuples
  - Can use x.name x.age etc which makes it slightly easier

# Person class

```
class Person:
	def parse(self,line):
		fields=line.split('\t')
		self.name=fields[0]
		self.gender=fields[1]
		self.age=int(fields[2])
		self.favorite_language=fields[3]
		return self

	def __repr__(self):
		return "Person( %s, gender=%s, %d years old, likes %s)"%
					(self.name,self.gender,self.age,self.favorite_language)
```

- people=sc.textFile("../../data/people.txt").map(Person().parse)

# Sending programs within shell

- You can use extra parameters to include python (or java) programs in your shell
  - --py-files (and list of files, separated with spaces)
    - Can use .py, .zip, .egg
  - --jars to include java jars
  - --packages, -- repositories to include maven packages (java)
  - --files to include arbitrary files in home folder of executor
- Get out of pyspark
  - Ctrl-D
- Run it again, including person.py in your --py-files

# Person with Objects

- Number of people by gender
  - people.map(lambda t: (t.gender,1)).reduceByKey(lambda x,y:x+y).collect()
- Let’s do number of people by programming language
- Youngest person by gender
  - people.map(lambda t: (t.gender,t.age )).reduceByKey(lambda x,y:min(x,y))

# More people exercises

- Get number of people with age 40+
  - Using filter
  - Using map and reduceByKey to produce two groups <40, 40+
- Get age of oldest person, by programming language

# Sales example

- Sales: Day| StoreId | ProductId |QtySold
- Load:
  - sales=sc.textFile("sales-data/sales_*.txt").map(lambda x: x.split('\t'))
- now sales is an rdd of arrays corresponding to the fields
  - but each field is a string
- Total quantity of products sold:
  - sales.map(lambda x: int(x[3])).sum()

# Grouping RDDs again

- Work on RDDs of *pairs, <key,value>*
- .reduceByKey(func)
  - groups based on the key
  - reduce values in each group using the passed function
    - function is same way as reduce
  - produces RDD <key, result>

# Example

- sales_by_store=sales.map( lambda t : (t[1], int(t[3])))
- sales_by_store.reduceByKey(lambda t1,t2: t1+t2).collect()

# Exercises

- Calculate the sales for each day
- Calculate the total sales for each day for store 1
- Calculate the total sales for each product

# Joins

- Joins allow us to combine 2 different RDDs
  - Each RDD is of the form <K,V> (key and value)
  - Result is of the form<K,<V1,V2>> (notice the nesting)
  - Joins only on equal keys (equijoin from db)
  - Also have leftOuterJoin, rightOuterJoin and fullOuterJoin
  - And cartesian, if you want the cartesian product, and other kinds of joins, but this is potentially very slow

# Simple join example

```
states=[
("AL","Alabama"),
("AK","Alaska"),
("AR","Arizona")
]; # apologies to the other 47 ...

populations=[
("AL",4779736),
("AK",710231),
("AR",6392017)
]; # according to 2010 census, from Wikipedia

states_rdd=sc.parallelize(states)
populations_rdd=sc.parallelize(populations)

states_rdd.join(populations_rdd);
```

# Sales and Objects

- Two other files, one for Products one for Stores
  - Classes: Store, Product, SaleRow, with parse method
- base_path="../data/sales“
- Sales_schema.py

```
stores=sc.textFile(base_path+"stores*.txt").map(lambda x:sales_schema.Store().parse(x))
products=sc.textFile(base_path+"products.txt").map(lambda x:sales_schema.Product().parse(x))

sales=sc.textFile(base_path+"sales_*.txt").map(lambda x:sales_schema.SaleRow().parse(x))
```

# Sales examples (with objects)

- sales_by_day=sales.map(lambda x : (x.day,x.quantity)).reduceByKey(lambda x,y:x+y)
- sales_by_store=sales.map(lambda x : (x.store_id,x.quantity) ).reduceByKey(lambda x,y:x+y)
- Now let’s do sales by product
- Get products with category stuff

# Sales and Joins (with objects)

- sales_by_store_joined=
  - sales_by_store.join(stores.map(lambda x: (x.id,x.name)))
- Now let’s do it with products

# Other joins

- Outer joins
  - Include the keys
  - .leftOuterJoin, .rightOuterJoin, .fullOuterJoin
- Cartesian Product
  - .cartesian

# Writing spark applications

- Need to obtain a SparkContext
  - from pyspark import SparkContext, SparkConf
  - conf = SparkConf().setAppName(appName) # appName not needed, but …
  - sc = SparkContext(conf=conf)
- Everything is the same after that !!
- You probably want to save your data … ☺
  - saveAsTextFile
  - Remember part files ☹
  - Can save in other ways

# Other functions

- Sample(withReplacement, fraction, seed)
- Union, intersection, distinct
- Coalesce, repartition
- aggregateByKey
- groupByKey
- repartitionAndSortWithinPartitions
- mapPartitions, mapPartitionsWithIndex

# New DataTable functionality

- A datatable is like an RDD but with schema information
  - Like a table in SQL, or datatable in pandas
  - Generic objects, know their fields
  - Datatable knows all its columns
  - All ‘rows’ are of the same kind (but there are nulls, and arrays etc)
- We need to either read from places with schemas, or add schema info
- We specify queries on them (similar to RDD, or through SQL), but there’s a query optimizer
  - Slightly harder to do general aggregates
- Much smaller python tax !

# Person datatable example

- Easiest way to get data with schema is from a ‘json’ file
  - Each line is a json object
  - { “field”:”value”, …}
- Need to use sqlCtx
  - people=sqlCtx.jsonFile("../../data/people1.json“)
- Notice how each element is a Row, knows its fields
- .show() displays in nice way (first 20 by default)

# Datatable

- .select – like map, can use strings or columns
  - people.select("name",people.age+1).show()
- .filter – filter certain rows
  - people.filter(people.age>30)
- .show – display nicely
- Pandas syntax for filter
  - people[people.gender=='F']
- GroupBy returns a grouped RDD
  - people.groupBy(people.gender).count()
- Join

# Group By

- .GroupBy creates a grouped RDD
  - Can specify several fields
  - Still need to specify aggregates
- Aggregates
  - Count, sum, …
- Can specify several with agg

# SQL

- Need to register the tables with the context
  - people.registerTempTable("people")
- Then can use .sql to do sql queries
  - sqlCtx.sql("select name, age FROM people").show()
  - sqlCtx.sql("select gender,avg(age) AS Av FROM people GROUP BY gender")

# Performance considerations

- Spark in python is slower than in scala due to translation
  - Spark processes are running in JVM
  - Need to seend objects back and forth between jvm and python
- Datatable avoids this translation, it all lives in JVM
  - Until last step to client ☺
- Datatable can optimize better
  - But you lose some control
- Shuffling (join/reduce) is more expensive
  - Partitioning can help some

# RDD Performance

- RDD is:
  - Lineage
    - Set of Partitions/splits
    - List of dependencies on parent RDDs
    - Function to compute each partition given its parents
  - Optimized execution
    - Partitioner – which objects go on which partitions
      - Partitioning can help when shuffling
    - Preferred location for each partition

# Execution

- Your program
- Spark driver (master)
  - Keeps track of RDD graph
  - Scheduler
  - Block tracker
  - Shuffle tracker
- Spark executors
  - Task threads
  - Block manager