# Introduction to Spark with Python

## What is Spark

* Based on Resilient Distributed Datasets

## Resilient Dsitributed Datasets

* It is basically a huge list but distributed over many computers
* Immuatable! Not designed for read/write
  + Instead transform an existing one into a new one
* Divide the data in slices and keep each slice in a different node
  + Values are computed only when needed
  + To guarantee fault-tolerance, we also keep info about how we calculate each slice, so we can re-generate it if a node fails

## Shared Spark Variables

* Broadcast variables
  + Copy is kept at each node
* Accumulators
  + You can only add; main node can read

## Functional Programming in Python

### Map

* Apply an operation to each element in a list, return a new list with the results

### Filter

* Select only certain elements in a list

### Reduce

* Applies a function to all pairs of elements of a list; returns one value, not a list
* Better for functions that are commutative and associative

### Flatmap

* Sometimes we end up with a list of lists, and we want a ‘flat’ list
* Spark gives us flatMap, which flattens such a list
* Itertools.chain maps a list of iterables into a flat list and enables us to define our own flatmap

## Creating RDDs in Spark

* All spark commands operate on RDDs
* You can use sc.parallelize to go from list to RDD
* Later we will show how to read from files
* Many commands are lazy
* In pySpark, sc represents your SparkContext

## Transformations vs Actions

* RDD methods are divided into two kinds
  + Transformations
    - Return another RDD
    - Are not really performed until an action is called
  + Actions
    - Return a value other than an RDD
    - Are performed immediately

## Some RDD Methods

### Transformations

* .map(f) – returns anew 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

## Reading Files

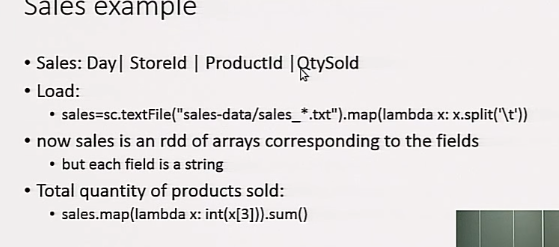
* Sc.textFile(urlOrPath, minPartitions,useUnicode=True)
  + Returns an RDD of strings (one per line)
  + Can read from many files, using wildcards (\*)
  + We normally use map right after and split/parse the lines

## 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

## Sending Programs within Shell

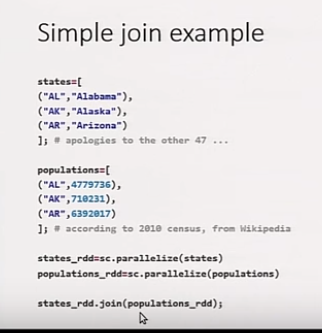
* You can use extra parameters to include python (or java) programs in your shell
* Get out of pyspark
  + Cntrl-D



## Joins

* Allows 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, but this is potentially very slow

### Simple Join Example



## Writing Spark Applications

* Need to obtain a SparkContext
  + From pyspark import SparkContext, SparkConf
  + Conf = SparkConf().setAppName(appName)
  + Sc = SparkContext(conf=conf)

## Data Tables

### New DataTable Functionality

* A datatable is like an RDD but with scheme information
  + Like a table in SQL, or datatable in pandas
  + Generic objects, know their fields
  + Database 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 Database 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)

### Database methods

* .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

## Performance Considerations

* Spark in python is slower than in scala due to translation
  + Spark processes are running in JVM
  + Need to send 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 com control
* Shuffling (join/reduce) is more expensive
  + Partitioning can help some

### RDD Performance

* RDD is Linage
  + Set of Partitions/splits
  + List of dependencies on parent RDDs
  + Function to compute ach partition given its parents
* RDD is optimized execution
  + Partitioner – which objects go on which partitions
    - Partitioning can help when suffling
  + Preferred location for each partition

# Machine Learning on Apache Spark MLib (Cloudera)