Module: Big Data Analytics – 1 Credit / 15 hours
Competency: Big Data
Author: Jack Pope

Instructor overview & objective
This course module is about data analytics in a distributed computing environment.

Given that this module is to be worth 1 credit, or 15 hours of coursework, the instructor should prepare three to five subtopic discussions. These should account for at least 5 hours of instructional material and might be in the form of lecture notes or lecture videos. The additional hours should be comprised of student labs or assignments.

The lesson below use the Python (PySpark) and SQL programming languages on distributed instances of Apache Spark. Installation and configuration of required software is covered in the Big Data Systems course module.

Students should have prior experience in data structures in the Python programming language. Experience with SQL and Functional Programming will also be helpful.

Schedule

Unit	Topic
1	Overview
2	The map-reduce paradigm
3	Resilient Distributed Datasets (RDD)
	RDD partitioning
	Python dictionaries versus RDDs
4.	Functional programming
	Higher order functions
	Exercise
5	PySpark
	Exercise
	RDD Persistence
	Shared variables
	Exercise
6	Spark SQL
	Exercise
7	Spark applications

Overview
The problem, or What is Big Data?

One definition of Big Data, according to Jack Pope: *when the value of your data needs exceeds the value of your local processing resources.*

By this definition, Big Data processing cannot take place on a single machine. Instead, many machines, a "cluster," must be used in tandem. These machines can be local or virtual machines in the cloud.

For our big data analytics in this course module we will emphasize Apache Spark. Spark makes for distributed in-memory computing and can write to local as well as networked file systems. A distributed file system called the Hadoop Distributed File System (HDFS) is typically installed with Apache Spark in the enterprise. We will not use HDFS in this module.

Spark essentially allows us to process large data sets faster by employing more than one machine. Spark also easily integrates with popular programming languages, databases and other tools for ensuring high availability.

A few points of information need to be made to distinguish Big Data Systems from Big Data Analytics. Big Data Systems are platforms. Before installing and configuring a Big Data System, the analytical problem should be understood. This is because a lot of effort and time can go into building a particular Big Data System that is not suitable for the problem, and sometimes we just don't need a sledgehammer to drive a tack. We cover Big Data Systems in another course module.

Big Data Analytics deals with the commands and logic that we can employ, often in the form of a functional programming language, to extract intelligence from large data sets. The analytics yielded may be in text of graphical format.

We are using PySpark (Python) for programming applications for a Spark cluster. PySpark is included in the installation of Apache Spark 2.3.1 and it works with GraphX, MLlib and Spark SQL libraries that come with Apache Spark. You must separately install Python (version 2). Please see "installation and configuration" in the Big Data Systems module.

PySpark is part of the Spark Core library in Apache Spark.

Spark libraries include:
- Spark Core – provides task distributions, scheduling, basic I/O, and an API for Java, Python, Scala and R.
- Spark SQL – relies on JDBC and treats tables as RDDs so that queries are Spark operations.
- MLlib (deprecated to ML) – machine learning algorithms.
- GraphX – graph algorithms.
- Spark Streaming – real-time stream data processing.

The map-reduce paradigm
When faced with huge data sets we are motivated to avoid serial in design and execution, as it can tax patience. Instead, we try to break a workload into parts that can be processed independently using multiple machines. Such parallelization may or may not be feasible given the problem at hand, and we have to think about what problems can be solved in asynchronous task parallel and data parallel fashion.

Processing of partitioned data on cluster nodes takes place independently and concurrently. Results can be consolidated on a master node according to the keys of key-value pairings in the data.

While the map reduce concept goes back a to the time when Lisp was at is zenith, Google managed to win a patent for its version, spelled MapReduce (USPTO patent #7,650,331).

The map-reduce paradigm supports parallel computing. The map function as applied to each value of a data set is readily performed in parallel, as the the value and the function are independent. However, in the context of distributed processing, a reduce operation will depend on a completed map operation.

map and reduce definitions:
Fundamentally, the map and reduce functions each take as parameters a data set and a function, which is then applied to each element of data. The map function then returns a transformed data set. The reduce function returns an aggregate of the the data set.

- map definition 2: index and partition input data into a subset for each compute node. In tree structured clusters, worker nodes are masters of subordinate nodes.

- reduce definition 2: the master node consolidates compute node output into one data set. Reduction may entail sorting, averaging, filtering, etc.

In Apache Spark, map and reduce work on local data, and the map-reduce paradigm is injected with “shuffle” and “sort” operations. We have the following definitions:

- map definition 3: Worker nodes apply the map function to their respective local data, maintaining results in temporary storage or memory. Map operations occur on each partition element and include transforming or filtering.

- shuffle: The transfer of values from the mapper to the reducer. Worker nodes are assigned a partition of data associated with a sorted range of keys. The keys are generated by a dedicated key-value map function.

- sort: a dependency for shuffle, with ordering by keys, then value.

- reduce definition 3: Worker nodes aggregate (sum, count) all partition values by key.

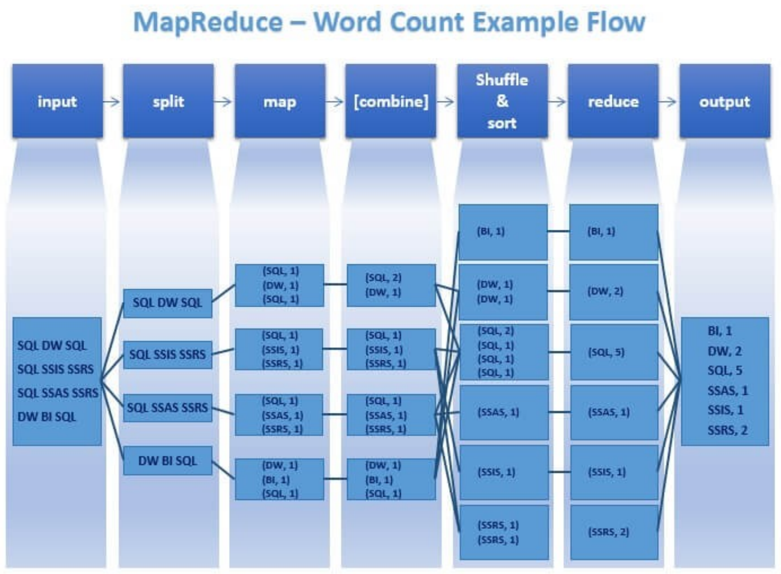
The pseudo code for a map:

```
map(key, value) → list(key, value')
```

The pseudo code for a map:

```
reduce(key, list(value')) → (key, value")
```

Apache Hadoop's implementation of Google's MapReduce entails additional operations such as split and combine. The following illustration provides some clarity (Source: https://www.mssqltips.com/sqlservertip/3222/big-data-basics--part-5--introduction-to-mapreduce/)



In the past decade, the general map-reduce concept has been convoluted by various implementations, including complex ones like that of MapReduce for Hadoop. Apache Spark does not require Hadoop Google's MapReduce. Spark employs the map-reduce paradigm in partitioning RDDs and applying transformation to data in dedicated areas of distributed memory.

Resilient Distributed Dataset (RDD)
Spark's architecture is designed around a data structure called the Resilient Distributed Dataset (RDD).

RDDs may be considered as specialized collections in the Java sense. Like a hash map in Java, RDDs can store primitive and object data, including other lists, and can be stored to disk as serialized object files.

As a key-value data structure, the RDDs are partitioned according to the number of distribution targets. You can think of each partition as representing a sublist from a master list of data. If you are unfamiliar with HashMap or key-value oriented data structures, consider these like parallel arrays (associative arrays) with one array used to stored the index (or key) and the other array used to store the value per the same index.

Each data set in the RDD may be considered a record, or it might be another object.

The RDD is immutable. That means that any transformation entails a new RDD; the original is maintained. And since each node maintains the full original RDD, the failure of one node means that its RDD partition can be applied to other nodes. Hence, the notion of “Resilient.”

RDDs accommodate map and reduce operations. The an RDD key can be used for a search or to accumulate a series of values from worker nodes.

For reduce operations, Spark has combine functions. Combine functions include sum, but exclude other that do not correctly evaluate mathematics after data is aggregated from nodes.

RDD operations:
- Transformations − create a new RDD from an existing RDD. Ex: map, filter, groupBy, join
- Action − effect RDD elements and return result to driver. Ex: collect

Here is an example of join:

```
a = sc.parallelize([("goat", 1), ("pig", 3), ("cow", 5)])
b = sc.parallelize([("goat", 2), ("pig", 4), ("cow", 6)])

ab = a.join(b)
ab_here = ab.collect()
print "Set a joined to set b: %s" % (ab_here)
→ ('Set a joined to set b: %s', [('goat', (1, 2)), ('cow', (5, 6)), ('pig', (3, 4))])
```

Important notes regarding these RDDs:
- Have a key-value structure.
- Join occurs on common keys.

Therefore, if the sets have no commond keys, the RDD resulting from a join will be empty. For example, if we had a third set with only one common key, observe the join:

```
c = sc.parallelize([("goat", 7), ("lamb", 1), ("chicken", 3)])
a.join(c).collect()        →        [('goat', (1, 7))]
```

Check that the RDD is cached:

```
ab.persist().is_cached     →        True
```

More:
https://spark.apache.org/docs/latest/api.html
https://spark.apache.org/docs/latest/api/java/org/apache/spark/rdd/RDD.html

RDD partitioning
Partitioning of RDDs can be done using HashPartitioner or CustomPartitioner.

HashPartitioner:
- Spark's defaut partitioner
- The key is from Java’s Object.hashcode() method. This will provide each element of an RDD with a distinct hash if the associated object is distinct. Objects that are the same will have the same hash (Java example: “key”.hashCode().equals(“key”.hashCode()) → true)

RangePartitioner:
- Creates equally sized partitions on sortable records.
- Initially sorts according key.

CustomPartitioner:
- Just extend Spark's Partitioner class.

Python dictionaries versus RDDs
An efficient way to compute modes is using associative-arrays, also known as dictionaries. Dictionaries are associated (key : value) pairs. Unlike strings and lists, dictionaries are a group of Python collections known as non-sequential collections (as opposed to sequential). Like lists, dictionaries are mutable.

For example:

```
times = {'Owens' : 10.1, 'Smith' : 10.5, 'Franks' : 10.7 }
```

Or a string that concatenates two strings for a string-int pair:

```
>>> prices[“Fall” + “, Boston”] = 21
>>> prices[“Spring” + “, Chicago”] = 23
>>> prices[“Summer” + “, NYC”] = 22
>>> prices[“Fall” + “, Boston”]
21
```

RDDs, like Python dictionaries, have their own methods. Of particular note are the methods to get the keys and values. In Python there is the form dName.keys() and dName.values(). [equiv in Spark]

The method items() returns a tuple from a dictionary. Tuples are similar to lists, but are immutable. They can have various data types.

```
>>>  times.items()
dict_items([('Franks', 10.7), ('Owens', 10.1), ('Smith', 10.5)])

>>> times.get("Owens")
10.1
```

Suppose there are multiple values in a dictionary having the same key. Can you devise a way to compute the average of these values? Here is an example that combines two “associative arrays” with dictionaries. Based on this example, it may come as no surprise that dictionaries are also referred to as associative arrays.

```
import decimal

# some times
times =  [ "Owens:10.1", "Smith:10.5", "Owens:10.8", \
           "Frank:10.7", "Owens:10.2", "Lewis:10.4" ]

# define some more dictionaries for holding stats
total = { }
mode = { }

# useful key and value arrays
key = []
value = []

# initialize the keys of the stat dictionaries
for i in times:
  pair = i.split(":")
  key = pair[0]
  total[key] = 0
  mode[key] = 0

for i in times:
  pair = i.split(":")
  key = pair[0]
  value = pair[1]

  total[key] += decimal.Decimal(value)
  mode[key] = mode[key] + 1

# output average time for Owens only
mean = total['Owens'] / mode['Owens']
print(mean)

print "Owen's average time is: %0.2f seconds over %i races." % (mean, mode['Owens'])
→ Owen's average time is: 10.37 and number of races is 3
```

Converting the above program to PySpark, we use map and groupByKey operations. For example:

```
times = [('Owens',10.1), ('Smith',10.5), ('Owens',10.8), ('Frank',10.7),
              ('Owens',10.2), ('Lewis',10.4)]

sc.parallelize(times).groupByKey().mapValues(lambda a: sum(a) / len(a)).collect()
→
[('Smith', 10.5), ('Frank', 10.7), ('Lewis', 10.4), ('Owens', 10.36)]
```

As you can see, the PySpark code is brief relative to regular Python.

RDD APIs:
https://spark.apache.org/docs/latest/api/
https://spark.apache.org/docs/latest/rdd-programming-guide.html

Functional programming in PySpark
Various programming languages implement the functional paradigm, including Java via Lambdas, D, Python and Scala.

Relative to standard procedural languages such as C and Java, functional programming allows to program with relatively concise statements that reflect mathematical expressions (accordance with formal mathematical provability). This is because functional programming breaks a problem into specialized functions connected by input and output.

Modularity is a byproduct of breaking a problem into is smallest parts, each represented by a function. Programs that are comprised of many distinct small functions, each performing a single task, are more maintainable than a large complicated function (effectively a program). In turn, a benefit of the modularity afforded by functional constraints includes quicker prototyping.

Formally, functional programming entails immutable objects, and if a language is “purely functional,” then there are no side effects (that alter some system state apart from the function itself). Hence, a function's output always depends on an explicit input. This also means that objects are not changed by a function. Instead, a new object is created for output, as is the case with Spark's central data structure, the Resilient Distributed Dataset or RDD.

Functions in a functional programming language may be “higher order functions,” meaning that they accept other functions as arguments and are able to return a function. Typical examples of higher order functions include map, reduce and filter. Such higher order functions exist in PySpark.

Because functional languages put constraints on side effects, functions and their processes can operate independently, which is required for asynchronous parallel operations. That way the processes of various functions do not contest the same objects or system resources, which would slow processing.

Lazy evaluation:
Lazy evaluation is another characteristic of functional programming. Lazy evaluation reserves the evaluation of an expression until it is needed. This facilitates modularization since it may not be known which function or expression should ultimately be engaged. This goes hand-in-hand with being side-effect free, so that program behavior is not affected by the order of evaluation.

Lambda expressions:
In several languages that support functional programming, a facility called lambdas that accommodates small user defined, or anonymous, functions that can be passed as parameters to other functions.

Python has a functional programming library called functools. PySpark readily accommodates functional programming.

In Python, a lambda expression combines parameters into an “anonymous function” such as:

```
lambda x, y: x * y
```

We can assign this lambda to a variable such as:

```
product = lambda x, y: x * y
```

… and affect the lambda as:

```
print(product(2,3))          →          6
```

In other words, we assigned the anonymous function to “product” and passed it as a parameter to the print() function. However, we need not even name the function. For example:

```
result = otherFunction(lambda x, y: x * y)
```

This arrangement enforces narrow functionality. This is unlike defining a using the “def” keyword in Python, which encourages line-after-line of alternative actions for the function to take.

To use conditional expressions withs, we can have:

```
fn = lambda L, R: ("Go Left", L, R) if L > 5 and R < 10 else ("Go Right", L, R)
print(fn(5,15))              →          ('Go Right', 5, 15)
```

Higher order functions
Higher order functions (aka “functors”) accept other functions as arguments and are able to return a function. Examples of higher order functions include map, filter, flatMap and reduce.

map:
The map function applies another function to a list of values. Its output is a new list of transformed elements.

map takes two arguments:

```
map(function, list)
```

For example, suppose we have a list of exam grades:

```
grades = [30, 35, 55, 70, 45, 60]
```

To scale all exam grades up by 10 points, map a lambda as follows:

```
grades_adj = list(map(lambda a: a + 10, grades))
list(grades_adj)             →          [40, 45, 65, 80, 55, 70]
```

reduce: Like map, reduce applies a function to a list of values. Unlike map, the output of reduce is an aggregate of the list. The operation of reduce is pairwise such that for elements a, b, the pairing “a + b, b” creates a sum aggregate. For example:

```
ints = [2, 4, 6, 8]
sum = lambda a, b: a + b
avg = reduce(sum, ints)/len(ints)
print(avg)        →                  20
```

filter: used to filter data, conditionally allowing some data to pass without modifying it. Hence, the resulting list may be shorter than the input list.

```
ok = lambda a: a > 0
ints = [2, -4, 6, -8]
pos_ints = filter(ok, ints)
print(pos_ints)                  →                  [2, 6]
```

flatMap: the map function may yield multiple sub lists. If a single “flattened” list is preferred, then use flatMap.

For example, in PySpark suppose the following

```
sc.parallelize(grades).map(lambda a: [a,  a + 10]).collect()
→ [[30, 40], [35, 45], [55, 65], [70, 80], [45, 55], [60, 70]]
```

Using flatMap we get:

```
sc.parallelize(grades).flatMap(lambda a: [a,  a + 10]).collect()
→ [30, 40, 35, 45, 55, 65, 70, 80, 45, 55, 60, 70]
```

… with results “flattened” into one single list. So, given a couple sets of grade provided as:

Similarly, given nested lists such as:

```
grades = [[20, 40, 60, 80], [10, 30, 50, 70]]
```

We can flatten the list into a single list as follows:

```
sc.parallelize(grades).flatMap(lambda a: a).collect()
→ [20, 40, 60, 80, 10, 30, 50, 70]
```

That is enough of a functional programming overview for this module. For more functional programming material and Python basics, please visit the other Python and Functional Programming modules.

The concepts of map and reduce as seen in Python evolve in the distributed context of the Spark environment, where map performs transformations while another function “collect” is responsible for retrieving transformed elements.

Related Internet resources:
https://en.wikipedia.org/wiki/Functional_programming
https://en.wikipedia.org/wiki/Currying
https://docs.python.org/2.7/library/functools.html
https://en.wikipedia.org/wiki/Higher-order_function

Exercises
1. Using Python, given a list of pets:

```
pets = ['dogs', 'cats', 'birds', 'fish']
```

Use the map function to print the character count comprising each pet type.

2. Given the list:

```
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Devise a lambda function to yield a list of only the odd integers.

3. Using Python, given a list of pairs, pets with their associated count, as (pet, count):

```
pets = [('dogs', 5), ('cats', 3), ('birds', 7), ('fish', 4)]
```

Devise lambda functions to do the following:
A. Sort the pets alphabetically by type.
B. Sort the pets numerically by count.
C. Retrieve the total count of all pets.

Hint: Lists contain a sort function.

Submit your code and output from executing your programs.

PySpark
PySpark is the API for Python programming in the Spark environment.

The interactive PySpark shell session can be launched from the command-line:

```
pyspark
```

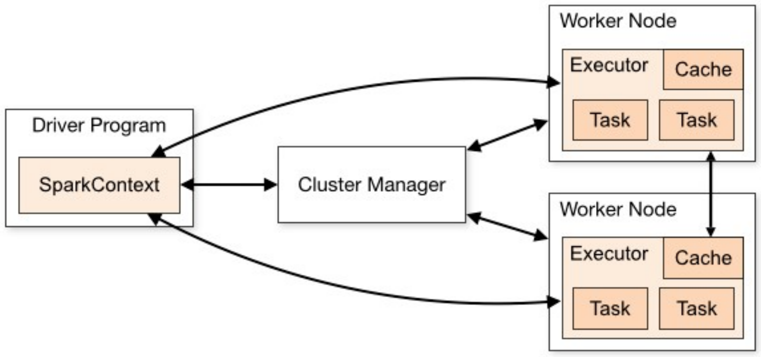
Alternatively, PySpark applications are typically run using $SPARK_HOME/bin/spark-submit. For example:

```
$SPARK_HOME/bin/spark-submit /path/to/my/program.py
```

Use of PySpark automatically imports required libraries (no explicit import statement required). For example, SparkContext uses the Py4J library to launch the JVM, creating a JavaSparkContext.

With respect to PySpark, a SparkContext instance named “sc” is automatically created. Therefore, it is useless to try to create a new SparkContext unless you want to change its constructor parameters.

Your Spark context defines how your application's processes run independently across a cluster. This is illustrated as follows (source: at https://spark.apache.org/docs/latest/cluster-overview.html):



With the initialization of a SparkContext, each worker node starts an executor program which eventually sends information back to to the driver on the master. The driver can be considered a local parent process of the remote executors. When the driver stops so do the executors.

The driver will serialize function objects for tasks on an executor. The executor then deserializes the the function and executes it on a partition of an RDD.

To check the Spark Context of a session, start a PySpark session and enter:

```
spark.sparkContext.getConf().getAll()
```

This will return a dictionary of settings. These setting can be changed by providing arguments to the SparkContext constructor. The argument choices are here:
https://spark.apache.org/docs/2.3.1/api/java/org/apache/spark/SparkContext.html

The first argument in the SparkContex is the cluster type. This can include:

- local – single machine (desktop, notebook, ...)
- stand-along – a group pre-defined machines
- yarn – a scheduler
- cloud – such as EC2

We could define a single task for a single “local” machine as follows:

```
sc = SparkContext('local', 'testing...')
```

Suppose we want to use all cores on a local machine, then we can specify:

```
sc = SparkContext('local[*]', 'testing...')
```

RDDs in the context of PySpark:
To distribute data around the disparate machines comprising a cluster, Spark uses a data structure called a Resilient Distributed Dataset or RDD.

RDDs include methods that works with the PySpark API's lambda syntax. A function can be passed to an RDD method, as we saw with the above example with sc.parallelize(grades).map(lambda a: [a, a + 10]).collect().

Unlike with Scala, Python RDDs can hold objects of multiple types. RRD methods can also take functions and return Python collection types.

The Spark API includes Spark Core for basic functionality such as RDDs and related map, reduce and filter operations.

Lets convert our earlier Python example to PySpark we use RDDs instead of Python lists. We now have:

```
grades = [30, 35, 55, 70, 45, 60]
rdd_grades = sc.parallelize(grades)
rdd_grades_adj = rdd_grades.map(lambda a: a + 10).collect()
print(rdd_grades_adj)          →          [40, 45, 65, 80, 55, 70]
```

When dealing with large data sets, the accumulation of output can push local resources. This is a reason to limit some of the collection by using the take() method instead of collect.

```
rdd_grades_adj = rdd_grades.map(lambda a: a + 10).take(2)
print(rdd_grades_adj)          →          [40, 45]
```

For an example of reduce, lets find the average grade.

```
rdd_grades_avg = rdd_grades.reduce(lambda a, b: a + b)
rdd_grades_avg = rdd_grades.reduce(lambda a, b: a + b)/rdd_grades.count()
print(rdd_grades_avg)          →          49
```

Exercise
Repeat the Python exercises in the Functional Programming section using PySpark.

Submit your code and output from executing your program.

RDD persistence
Redundancies such as RDD duplication in memory and storage to disk (serialization).

Serialization is useful for data transmission as well as storage and can speed up operations.

Serialization can be accommodated by file format such as XML, JSON or binary (really Java byte code for Spark).

PySpark has a variety of serializers, including:
- AutoSerializer – Automatically select Marshal or Pickle serializer protocol.
- CompressedSerializer – Serialize and unserialize objects with GZIP compression.
- MarshalSerializer – faster than PickleSerializer, but has fewer data types.
- PickleSerializer – accommodates most Python objects.

More here: https://spark.apache.org/docs/2.3.1/api/python/_modules/pyspark/serializers.html

Seven RDD storage levels are outline here:
https://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-persistence

For example, lets serialize an RDD for storage to both memory and disk:

```
a.persist(pyspark.StorageLevel.MEMORY_AND_DISK_SER)
```

Verify that storage:

```
print(a.getStorageLevel())      →      Disk Memory Serialized 1x Replicated
```

Another example (If within the pyspark interactive shell, first stop the prior Spark Context):

```
sc.stop()
from pyspark.serializers import PickleSerializer
sc = SparkContext("local", "serialization app", serializer = PickleSerializer())
print(sc.parallelize(list(range(1000))).map(lambda a: a * 2).take(10))
sc.stop()
```

Shared variables
Types of variables that are shared across nodes:
- Accumulator – stores of updated aggregated value.
- Broadcast – sent to all nodes where they are cached.

In the following example, the variable 'a' is an accumulator, as it is modified such that a = a + b.

```
ints = [2, 4, 6, 8]
sum = lambda a, b: a + b
avg = reduce(sum, ints)/len(ints)
print(avg)       →             20
```

Example broadcast:

```
more_animals = sc.broadcast([("rooster", 1), ("ox", 2), ("horse", 5)])
```

Now verify oxen in new set:

```
animal = more_animals.value[1]
print ("Oxen? ... %s", animal)       →       ('Oxen? ... %s', ('ox', 2))
```

Exercise
Devise an algorithm that Create an RDD_alpa1 consisting of the letters of the alphabet. Serialize the RDD to disk using CompressedSerializer. Devise a second algorithm that create RDD_alpha2 from the serialized RDD_alpa1 that is on disk. Write test code that validates the two RDDs are the same.

Upload your code and test output.

Spark SQL
Spark SQL allows RDD transforms using Structured Query Language (SQL). Ironically, it does not require a database engine such as SQL Server or Oracle be accessed, though it can do that as well. The flexibility of Spark SQL is that data can reside in semi-structured text files, such as JSON.

The Spark SQl API: https://spark.apache.org/docs/2.3.1/api/python/pyspark.sql.html

Given the file SPY.csv (5000 records with 7 fields each), I ran the following tasks:

```
import csv
aFile = "/home/.../SPY.csv"
rdd = sc.textFile(aFile)
rdd_result = rdd.mapPartitions(lambda fields: csv.reader(fields))
```

The following example program wordcount.py ships with Spark. It is modified here for style.

```
from __future__ import print_function
import sys
from operator import add
from pyspark.sql import SparkSession

if __name__ == "__main__":
   if len(sys.argv) != 2:
      print("Usage: wordcount <file>", file=sys.stderr)
      exit(-1)

   spark = SparkSession.builder.appName("PythonWordCount").getOrCreate()

   aFile = sys.argv[1]
   lines = spark.read.text(aFile).rdd.map(lambda r: r[0])
   counts = lines.flatMap(lambda x: x.split(',')).map(lambda x: (x, 1))
                 .reduceByKey(add)
   output = counts.collect()

   for (word, count) in output:
      print("%s: %i" % (word, count))

   spark.stop()
```

That program was run on a 3-node cluster from the command-line as:

```
$SPARK_HOME/bin/spark-submit \
$SPARK_HOME/examples/src/main/python/wordcount.py \
                  /home/.../SPY.csv
```

Here is another example that reads in a csv file:

```
from pyspark import SparkContext
from pyspark.sql import SQLContext

aFile = "/home/.../data/SPY.txt"
dat = spark.read.csv(aFile)
# dat.dtypes
# dat.printSchema()
dat.select("*").show(10)

+----------+------+------+------+------+---------+------+
|       _c0|   _c1|   _c2|   _c3|   _c4|      _c5|   _c6|
+----------+------+------+------+------+---------+------+
|2014-06-13|193.92|194.32|193.30|194.13| 81991900|194.13|
|2014-06-12|194.69|194.80|193.11|193.54|106350000|193.54|
|2014-06-11|194.90|195.12|194.48|194.92| 68772000|194.92|
|2014-06-10|195.34|195.64|194.92|195.60| 57129000|195.60|
|2014-06-09|195.35|196.05|195.17|195.58| 65119000|195.58|
|2014-06-06|194.87|195.43|194.78|195.38| 78696000|195.38|
|2014-06-05|193.41|194.65|192.70|194.45| 92103000|194.45|
|2014-06-04|192.47|193.30|192.27|193.19| 55529000|193.19|
|2014-06-03|192.43|192.90|192.25|192.80| 65047000|192.80|
|2014-06-02|192.95|192.99|191.97|192.90| 64656000|192.90|
+----------+------+------+------+------+---------+------+
only showing top 10 rows
```

A query using SQL's “where” clause to select all data in the year 2012:

```
dat.where(s._c0.rlike('2012')).show()

+----------+------+------+------+------+---------+------+
|       _c0|   _c1|   _c2|   _c3|   _c4|      _c5|   _c6|
+----------+------+------+------+------+---------+------+
|2012-12-31|139.66|142.56|139.54|142.41|243935200|138.98|
|2012-12-28|140.64|141.42|139.87|140.03|148806700|136.66|
|2012-12-27|141.79|142.08|139.92|141.56|167920600|138.15|
|2012-12-26|142.64|142.71|141.35|141.75|106947700|138.33|
|2012-12-24|142.48|142.56|142.19|142.35| 53874600|138.92|
|2012-12-21|142.17|144.09|141.94|142.79|245883800|139.35|
|2012-12-20|144.38|145.14|143.98|145.12|168487000|140.63|
|2012-12-19|145.53|145.58|144.24|144.29|150895400|139.82|
|2012-12-18|144.00|145.50|143.79|145.37|177762800|140.87|
|2012-12-17|142.47|143.85|142.43|143.77|143238200|139.32|
|2012-12-14|142.32|142.58|141.88|142.10|137701700|137.70|
|2012-12-13|143.42|143.83|142.27|142.63|135715000|138.21|
|2012-12-12|144.00|144.55|143.31|143.51|145880100|139.06|
|2012-12-11|143.06|144.11|142.99|143.44|152570400|139.00|
|2012-12-10|142.21|142.81|142.15|142.47| 98840700|138.06|
|2012-12-07|142.53|142.69|141.67|142.41|108726400|138.00|
|2012-12-06|141.37|142.04|141.16|141.98|103220600|137.58|
|2012-12-05|141.37|142.16|140.37|141.50|147300500|137.12|
|2012-12-04|141.44|141.87|140.87|141.25|127512200|136.88|
|2012-12-03|142.80|142.92|141.34|141.45|124656300|137.07|
+----------+------+------+------+------+---------+------+
only showing top 20 rows
```

In order to use standard SQL query strings, first define a temporary table:

```
dat.registerTempTable("SPY")
```

Now query as follows:

```
sql("select sum(_c5) from SPY where _c0 like '2012%'").show()

+------------------------+
|sum(CAST(_c5 AS DOUBLE))|
+------------------------+
|          3.58327342E10|
+------------------------+
```

Spark SQL can connect to to a database via the OBDC library. Keep in mind that if all compute nodes connect to a single database server, that server may become a bottleneck. Alternatively, you can use serverless SQLite (all nodes will need their own installation of SQLite).

Invoke PySpark session with SQLite:

```
pyspark --driver-class-path .:sqlite-jdbc-3.8.11.2.jar
```

Create a data frame:

```
df = sqlContext.read.format('jdbc'). \
      options(url='jdbc:sqlite:/home/.../data/Chinook_Sqlite.sqlite', \
      dbtable='employee',driver='org.sqlite.JDBC').load()
```

Test:

```
df.printSchema()
→
root
 |-- EmployeeId: integer (nullable = false)
 |-- LastName: string (nullable = true)
 |-- FirstName: string (nullable = true)
 |-- Title: string (nullable = true)
 |-- ReportsTo: integer (nullable = true)
 |-- BirthDate: date (nullable = true)
 |-- HireDate: date (nullable = true)
 |-- Address: string (nullable = true)
 |-- City: string (nullable = true)
 |-- State: string (nullable = true)
 |-- Country: string (nullable = true)
 |-- PostalCode: string (nullable = true)
 |-- Phone: string (nullable = true)
 |-- Fax: string (nullable = true)
 |-- Email: string (nullable = true)
```

Exercise
Note that Spark and SQL have some common methods. Using both Spark's RDD API methods and Spark SQL to create standard SQL Query strings, write PySpark code for the following operations: group by, count, sort, filter, union, intersection.

Apply your code to the Chinook database at
https://github.com/lerocha/chinook-database/blob/master/ChinookDatabase/DataSources/Chinook_Sqlite.sqlite

See also the Spark method APIs at:
https://spark.apache.org/docs/latest/api/java/org/apache/spark/rdd/RDD.html

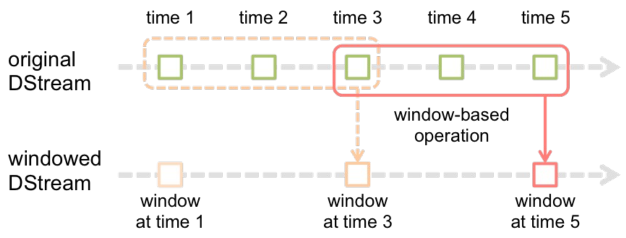
Based on this exercise, you should answer the following questions:
- Which approach is more costly in terms of coding times?
- Which approach is likely to be more time consuming in processing Big Data?

Upload your code and the first 10 lines of test output for each RDD and SQL method.

Applications
There are plenty of examples and data that you can test at https://github.com/apache/spark . We will comment here on Spark Streaming, Machine Learning and Graph processing in the context of PySpark.

Spark Streaming:
Spark Streaming enables high-throughput stream processing of live data streams. Data can be ingested from TCP sockets. Continuous streams are discretized into DStreams, which is internally represented as a sequence of RDDs. Text files can also be read as streams, which can allow programs to automatically adapt to local data sources.

Windowing: Discretization can be expanded over a sliding window of time periods, so that RDDs that fall within the window are combined into a windowed DStream.



Streams can also be joined. This may be useful for providing a seamless data feed when multiple sources are available (think IOT). Example:

```
joinedStream = windowedStream.transform(lambda rdd: rdd.join(dataset))
```

More details: https://spark.apache.org/docs/latest/streaming-programming-guide.html

Spark Machine Learning (MLib):
MLlib's APIs cover algorithms for classification, regression, clustering, dimensionality reduction, linear algebra and basic statistics. More details: https://spark.apache.org/docs/latest/ml-guide.html

Lets consider an application of binary classification. The code we will use is here: https://github.com/apache/spark/blob/master/examples/src/main/python/mllib/svm_with_sgd_example.py , modified for style and data paths:

```
if __name__ == "__main__":
    sc = SparkContext(appName="PythonSVMWithSGDExample")

    # Load and parse the data
    def parsePoint(line):
        values = [float(x) for x in line.split(' ')]
        return LabeledPoint(values[0], values[1:])

    data = sc.textFile("data/sample_svm_data.txt")
    parsedData = data.map(parsePoint)

    # Build the model
    model = SVMWithSGD.train(parsedData, iterations=100)

    # Evaluating the model on training data
    labelsAndPreds = parsedData.map(lambda p: (p.label, model.predict(p.features)))
    trainErr = labelsAndPreds.filter(lambda lp: lp[0] != lp[1]).count() /
                    float(parsedData.count())

    print("Training Error = " + str(trainErr))

    # Save and load model
    model.save(sc, "tmp/pythonSVMWithSGDModel")
    sameModel = SVMModel.load(sc, "tmp/pythonSVMWithSGDModel")
```

Some data we will test is here:
https://github.com/apache/spark/blob/master/data/mllib/sample_svm_data.txt

Binary classification divides data into two categories: positive and negative. This could reflect any number of states such as up/down, on/off, true/false, yes/no, etc (MLib denotes positive as 1 and negative as 0).

Classification can be done by Support Vector Machine (SVM) or logistic Regression. The example uses the former. An SVM separates data points into classes by a hyperplane. During training, the best hyperplane is sought with a certain maximum margin of distance from the hyperplane in separating the classes.

When run, a model is generated that has a certain training error. (For the given data, the training error is Training Error = 0.38198757764)

Graphs processing in Spark:
The premiere graphing program in Apache Spark is GraphX. However, GraphX is not compatible with PySpark at this time. An alternative project called GraphFrames provides a wrapper to GraphX objects by using Spark's DataFrames API. However, Spark 2x is not quickly compatible with GraphFrames either. A work around is to use the Networkx library for graph processing, converting graph data to RDDs. Another possibility for parallelization of graphs in Python/Networkx is a platform called Parallel Python: https://www.parallelpython.com

Other Internet resources

https://www.usenix.org/legacy/event/hotpar10/tech/full_papers/McCool.pdf
https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf
https://spark.apache.org/docs/latest/api/java/org/apache/spark/rdd/RDD.html
https://spark.apache.org/docs/latest/api
PySpark API docs: https://spark.apache.org/docs/2.3.1/api/python/index.html