wk4 Demo - Intro to Spark

`MIDS w261: Machine Learning at Scale | UC Berkeley School of Information | Spring 2023`

Last week we saw a number of design patterns in Hadoop MapReduce. This week we will look at the limitations of Hadoop MapReduce when it comes to running iterative jobs and preview the advantages of modern distributed compuation frameworks like Spark. By abstracting away many of the parallelization details Spark provides a flexible interface for the programmer. However a word of warning: don't let the ease of implementation lull you into complacency, scalable solutions still require attention to the details of smart algorithm design.

In class today we'll get some practice working with Spark RDDS. We'll use Spark to re-implement each of the tasks that you performed using the Command Line or Hadoop Streaming in weeks 1-3 of the course. Our goal is to get you up to speed and coding in Spark as quickly as possible; this is by no means a comprehensive tutorial. By the end of today's demo you should be able to:

- ... **initialize** a `SparkSession` in a local NB and use it to run a Spark Job.
- ... **access** the Spark Job Tracker UI.
- ... **describe** and **create** RDDs from files or local Python objects.
- ... **explain** the difference between actions and transformations.
- ... **decide** when to `cache` or `broadcast` part of your data.
- ... **implement** Word Counting, Sorting and Naive Bayes in Spark.

`NOTE:` Although RDD successor datatype, Spark dataframes, are becoming more common in production settings we've made a deliberate choice to teach you RDDs first beause building homegrown algorithm implementations is crucial to developing a deep understanding of machine learning and parallelization concepts -- which is the goal of this course. We'll still touch on dataframes in Week 5 when talking about Spark efficiency considerations and we'll do a deep dive into Spark dataframes and streaming solutions in Week 12.

`Additional Resources:` The offical documentation pages offer a user friendly overview of the material covered in this week's readings: Spark RDD Programming Guide.

RDD API docs: https://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD

Notebook Set-Up

In [5]:
```
#%cd /media/Instructors/LiveSessionMaterials/wk04Demo_IntroToSpark/master

import os
DATA_BUCKET = os.getenv('DATA_BUCKET','')[:-1] # our private storage bucket location
DEMO04_FOLDER = f"{DATA_BUCKET}/notebooks/jupyter/LiveSessionMaterials/wk04Demo_IntroToSpark"
#print(f"Personal Data bucket:  {DATA_BUCKET}")
#!gsutil ls -lh  {DEMO04_FOLDER}
```

In [6]:
```
# imports
import re
import numpy as np
import pandas as pd
```

In [7]:
```
# store path to notebook
PWD = !pwd
PWD = PWD[0]
PWD
```

Out[7]:
```
'/'
```

In [8]:
```
# make data directory if it doesn't already exist
!mkdir -p data
```

Load the data

Today we'll mostly be working with toy examples & data created on the fly in Python. However at the end of this demo we'll revisit Word Count & Naive Bayes using some of the data from weeks 1-3. Run the following cells to re-load the *Alice in Wonderland* text & the 'Chinese' toy example.

In [9]:
```
# (re)download alice.txt used in HW1
#!gsutil cp gs://w261-hw-data/main/Assignments/HW1/data/alice.txt data/alice.txt
#ALICE_TXT = PWD + "/data/alice.txt"
# (re)download alice.txt used in HW1
!gsutil cp gs://w261-hw-data/main/Assignments/HW1/data/alice.txt {DEMO04_FOLDER}/data/alice.txt
ALICE_TXT = DEMO04_FOLDER + "/data/alice.txt"
```

```
Copying gs://w261-hw-data/main/Assignments/HW1/data/alice.txt [Content-Type=text/plain]...
/ [1 files][170.2 KiB/170.2 KiB]
Operation completed over 1 objects/170.2 KiB.
```

In [10]:

```
#%%writefile data/chineseTrain.txt
chinese_train = '''D1	1		Chinese Beijing Chinese
D2	1		Chinese Chinese Shanghai
D3	1		Chinese Macao
D4	0		Tokyo Japan Chinese'''

!echo "{chinese_train}" | gsutil cp - {DEMO04_FOLDER}/data/chineseTrain.txt
```

```
Copying from <STDIN>...
/ [1 files][    0.0 B/    0.0 B]
Operation completed over 1 objects.
```

In [11]:

```
#%%writefile data/chineseTest.txt
chinese_test = '''D5	1		Chinese Chinese Chinese Tokyo Japan
D6	1		Beijing Shanghai Trade
D7	0		Japan Macao Tokyo
D8	0		Tokyo Japan Trade'''

!echo "{chinese_test}" | gsutil cp - {DEMO04_FOLDER}/data/chineseTest.txt
```

```
Copying from <STDIN>...
/ [1 files][    0.0 B/    0.0 B]
Operation completed over 1 objects.
```

In [12]:

```
# naive bayes toy example data paths - ADJUST AS NEEDED
TRAIN_PATH = DEMO04_FOLDER + "/data/chineseTrain.txt"
TEST_PATH = DEMO04_FOLDER + "/data/chineseTest.txt"
```

Exercise 1. Getting started with Spark.

For week 4 you read Ch 3-4 from *Learning Spark: Lightning-Fast Big Data Analysis* by Karau et. al. as well as a few blog posts that set the stage for Spark. From these readings you should be familiar with each of the following terms:

- **Spark session**
- **Spark context**

- **driver program**
- **executor nodes**
- **resilient distributed datasets (RDDs)**
- **pair RDDs**
- **actions** and **transformations**
- **lazy evaluation**

The first code block below shows you how to start a `SparkSession` in a Jupyter Notebook. Next we show a simple example of creating and transforming a Spark RDD. Let's use this as a quick vocab review before we dive into more interesting examples.

In [13]:

```
from pyspark.sql import SparkSession

try:
    spark
except NameError:
    print('starting Spark')
    app_name = 'wk4_demo'
    master = "local[*]"
    spark = SparkSession\
            .builder\
            .appName(app_name)\
            .master(master)\
            .getOrCreate()
sc = spark.sparkContext
```

```
starting Spark
:: loading settings :: url = jar:file:/usr/lib/spark/jars/ivy-2.4.0.jar!/org/apache/ivy/core/settings/ivysettings.xml
Ivy Default Cache set to: /root/.ivy2/cache
The jars for the packages stored in: /root/.ivy2/jars
graphframes#graphframes added as a dependency
org.apache.spark#spark-avro_2.12 added as a dependency
:: resolving dependencies :: org.apache.spark#spark-submit-parent-00545c7e-8477-4fd2-acd0-dde153aefa6d;1.0
        confs: [default]
        found graphframes#graphframes;0.8.2-spark3.1-s_2.12 in spark-packages
        found org.slf4j#slf4j-api;1.7.16 in central
        found org.apache.spark#spark-avro_2.12;3.1.3 in central
        found org.spark-project.spark#unused;1.0.0 in central
downloading https://repos.spark-packages.org/graphframes/graphframes/0.8.2-spark3.1-s_2.12/graphframes-0.8.2-spark3.1-s_
2.12.jar ...
        [SUCCESSFUL ] graphframes#graphframes;0.8.2-spark3.1-s_2.12!graphframes.jar (92ms)
downloading https://repo1.maven.org/maven2/org/apache/spark/spark-avro_2.12/3.1.3/spark-avro_2.12-3.1.3.jar ...
```

```
        [SUCCESSFUL ] org.apache.spark#spark-avro_2.12;3.1.3!spark-avro_2.12.jar (24ms)
downloading https://repo1.maven.org/maven2/org/slf4j/slf4j-api/1.7.16/slf4j-api-1.7.16.jar ...
        [SUCCESSFUL ] org.slf4j#slf4j-api;1.7.16!slf4j-api.jar (14ms)
downloading https://repo1.maven.org/maven2/org/spark-project/spark/unused/1.0.0/unused-1.0.0.jar ...
        [SUCCESSFUL ] org.spark-project.spark#unused;1.0.0!unused.jar (14ms)
:: resolution report :: resolve 2830ms :: artifacts dl 150ms
        :: modules in use:
        graphframes#graphframes;0.8.2-spark3.1-s_2.12 from spark-packages in [default]
        org.apache.spark#spark-avro_2.12;3.1.3 from central in [default]
        org.slf4j#slf4j-api;1.7.16 from central in [default]
        org.spark-project.spark#unused;1.0.0 from central in [default]
        ---------------------------------------------------------------------
        |                  |            modules            ||   artifacts   |
        |       conf       | number| search|dwnlded|evicted|| number|dwnlded|
        ---------------------------------------------------------------------
        |      default     |   4   |   4   |   4   |   0   ||   4   |   4   |
        ---------------------------------------------------------------------
:: retrieving :: org.apache.spark#spark-submit-parent-00545c7e-8477-4fd2-acd0-dde153aefa6d
        confs: [default]
        4 artifacts copied, 0 already retrieved (455kB/9ms)
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
23/01/31 17:44:03 INFO org.apache.spark.SparkEnv: Registering MapOutputTracker
23/01/31 17:44:03 INFO org.apache.spark.SparkEnv: Registering BlockManagerMaster
23/01/31 17:44:03 INFO org.apache.spark.SparkEnv: Registering BlockManagerMasterHeartbeat
23/01/31 17:44:03 INFO org.apache.spark.SparkEnv: Registering OutputCommitCoordinator
```

In [14]:

```
sc.getConf().getAll()
```

Out[14]:

```
[('spark.driver.port', '38805'),
 ('spark.eventLog.enabled', 'true'),
 ('spark.dynamicAllocation.minExecutors', '1'),
 ('spark.jars.packages',
  'graphframes:graphframes:0.8.2-spark3.1-s_2.12,org.apache.spark:spark-avro_2.12:3.1.3'),
 ('spark.dataproc.sql.parquet.enableFooterCache', 'true'),
 ('spark.files',
  'file:///root/.ivy2/jars/graphframes_graphframes-0.8.2-spark3.1-s_2.12.jar,file:///root/.ivy2/jars/org.apache.spark_sp
ark-avro_2.12-3.1.3.jar,file:///root/.ivy2/jars/org.slf4j_slf4j-api-1.7.16.jar,file:///root/.ivy2/jars/org.spark-projec
t.spark_unused-1.0.0.jar'),
 ('spark.sql.warehouse.dir', 'file:/spark-warehouse'),
 ('spark.dataproc.sql.joinConditionReorder.enabled', 'true'),
 ('spark.executor.memory', '5739m'),
 ('spark.yarn.am.memory', '640m'),
 ('spark.jars',
  'gs://spark-lib/bigquery/spark-bigquery-latest_2.12.jar,file:///root/.ivy2/jars/graphframes_graphframes-0.8.2-spark3.1
-s_2.12.jar,file:///root/.ivy2/jars/org.apache.spark_spark-avro_2.12-3.1.3.jar,file:///root/.ivy2/jars/org.slf4j_slf4j-a
pi-1.7.16.jar,file:///root/.ivy2/jars/org.spark-project.spark_unused-1.0.0.jar'),
 ('spark.ui.port', '4040'),
 ('spark.dataproc.sql.local.rank.pushdown.enabled', 'true'),
 ('spark.submit.pyFiles',
  '/root/.ivy2/jars/graphframes_graphframes-0.8.2-spark3.1-s_2.12.jar,/root/.ivy2/jars/org.apache.spark_spark-avro_2.12-
3.1.3.jar,/root/.ivy2/jars/org.slf4j_slf4j-api-1.7.16.jar,/root/.ivy2/jars/org.spark-project.spark_unused-1.0.0.jar'),
 ('spark.app.initial.jar.urls',
  'spark://w261-m.us-central1-a.c.w261-student-348823.internal:38805/jars/graphframes_graphframes-0.8.2-spark3.1-s_2.12.
jar,spark://w261-m.us-central1-a.c.w261-student-348823.internal:38805/jars/org.apache.spark_spark-avro_2.12-3.1.3.jar,sp
ark://w261-m.us-central1-a.c.w261-student-348823.internal:38805/jars/org.spark-project.spark_unused-1.0.0.jar,spark://w2
61-m.us-central1-a.c.w261-student-348823.internal:38805/jars/org.slf4j_slf4j-api-1.7.16.jar,gs://spark-lib/bigquery/spar
k-bigquery-latest_2.12.jar'),
 ('spark.yarn.historyServer.address', 'w261-m:18080'),
 ('spark.executor.instances', '2'),
 ('spark.repl.local.jars',
  'file:/tmp/spark-a3115df8-6f88-41f9-ba79-04dd62911830/spark-bigquery-latest_2.12.jar,file:///root/.ivy2/jars/graphfram
es_graphframes-0.8.2-spark3.1-s_2.12.jar,file:///root/.ivy2/jars/org.apache.spark_spark-avro_2.12-3.1.3.jar,file:///roo
t/.ivy2/jars/org.slf4j_slf4j-api-1.7.16.jar,file:///root/.ivy2/jars/org.spark-project.spark_unused-1.0.0.jar'),
 ('spark.serializer.objectStreamReset', '100'),
 ('spark.app.name', 'wk4_demo'),
 ('spark.master', 'local[*]'),
 ('spark.yarn.unmanagedAM.enabled', 'true'),
 ('spark.submit.deployMode', 'client'),
 ('spark.sql.autoBroadcastJoinThreshold', '43m'),
 ('spark.extraListeners',
  'com.google.cloud.spark.performance.DataprocMetricsListener'),
 ('spark.sql.cbo.joinReorder.enabled', 'true'),
 ('spark.driver.maxResultSize', '1920m'),
 ('spark.shuffle.service.enabled', 'true'),
 ('spark.metrics.namespace',
  'app_name:${spark.app.name}.app_id:${spark.app.id}'),
 ('spark.driver.host', 'w261-m.us-central1-a.c.w261-student-348823.internal'),
 ('spark.scheduler.mode', 'FAIR'),
 ('spark.dataproc.sql.optimizer.leftsemijoin.conversion.enabled', 'true'),
 ('spark.history.fs.logDirectory',
  'gs://dataproc-temp-us-central1-913378501339-xe7nshrz/aaff1f58-9f9c-4a18-9d7b-d88bc3aa972d/spark-job-history'),
 ('spark.sql.adaptive.enabled', 'true'),
 ('spark.yarn.jars', 'local:/usr/lib/spark/jars/*'),
 ('spark.eventLog.dir',
  'gs://dataproc-temp-us-central1-913378501339-xe7nshrz/aaff1f58-9f9c-4a18-9d7b-d88bc3aa972d/spark-job-history'),
 ('spark.scheduler.minRegisteredResourcesRatio', '0.0'),
 ('spark.executor.id', 'driver'),
 ('spark.hadoop.hive.execution.engine', 'mr'),
 ('spark.app.id', 'local-1675187044370'),
 ('spark.app.startTime', '1675187043129'),
 ('spark.hadoop.mapreduce.fileoutputcommitter.algorithm.version', '2'),
 ('spark.executor.cores', '2'),
 ('spark.dynamicAllocation.maxExecutors', '10000'),
 ('spark.sql.catalogImplementation', 'hive'),
 ('spark.rpc.message.maxSize', '512'),
 ('spark.rdd.compress', 'True'),
 ('spark.executorEnv.OPENBLAS_NUM_THREADS', '1'),
 ('spark.driver.memory', '3840m'),
 ('spark.dynamicAllocation.enabled', 'true'),
 ('spark.ui.showConsoleProgress', 'true'),
 ('spark.sql.cbo.enabled', 'true'),
 ('spark.app.initial.file.urls',
  'file:///root/.ivy2/jars/org.spark-project.spark_unused-1.0.0.jar,file:///root/.ivy2/jars/org.slf4j_slf4j-api-1.7.16.j
ar,file:///root/.ivy2/jars/graphframes_graphframes-0.8.2-spark3.1-s_2.12.jar,file:///root/.ivy2/jars/org.apache.spark_sp
ark-avro_2.12-3.1.3.jar')]
```

In [15]:

```
# a small example
myData_RDD = sc.parallelize(range(1,100))
squares_RDD = myData_RDD.map(lambda x: (x,x**2))
oddSquares = squares_RDD.filter(lambda x: x[1] % 2 == 1)
```

In [16]:

```
oddSquares.take(5)
```

Out[16]:

```
[(1, 1), (3, 9), (5, 25), (7, 49), (9, 81)]
```

> **DISCUSSION QUESTIONS:** For each key term from the reading, briefly explain what it means in the context of this demo code. Specifically:
>
> - *What is the 'driver program' here?*
> - *What does the spark context do? Do we have 'executors' per se?*
> - *List all RDDs and pair RDDs present in this example.*
> - *List all transformations present in this example.*
> - *List all actions present in this example.*
> - *What does the concept of 'lazy evaluation' mean about the time it would take to run each cell in the example?*
> - *If we were working on a cluster, where would each transformation happen? would the data get shuffled?*

INSTRUCTOR TALKING POINTS

- What is the 'driver program' here?

> The driver program is the code we write here in this Jupyter Notebook.

- What does the spark context do? Do we have 'executors' per se?

> Driver programs access Spark through a SparkContext object, which represents a connection to a computing cluster. Once you have a SparkContext, you can use it to build RDDs.
>
> The spark context is a local object that details a set of configuration parameters for running Spark jobs. Among other things it determines how many executors will run on each node in your cluster & what resources will be allotted to each one. In this case we're running Spark in local mode which means that we're using a pseudo cluster -- instead of executors each running with their own JVM (Java Virtual Machine) on various linked machines, in local mode Spark will use the same JVM for all the execution task & the driver itself.

In [1]:
```
# from IPython.display import Image
# Image(filename="spark-context.png")
```

- List all RDDs and pair RDDs present in this example.

> RDDs: `myData` ; Pair RDDs: `squares and oddSquares`

- List all transformations present in this example.

> `map() and filter()`

- List all actions present in this example.

> `take()`

- What does the concept of 'lazy evaluation' mean about the time it would take to run each cell in the example?

> Lazy evaluation means that Spark keeps track of a directed acyclic graph (DAG) that contains the list of transformations needed to recreate each RDD and only actually performs those transformations when an action is called. This means that when we run the cell with 3 lines of code, no transformations actually happen, whereas running the `take` action triggers the computation to start.

- If we were working on a cluster, where would each transformation happen? would the data get shuffled?

> There are two transformations in this code: map and filter. Both will happen on the nodes where the data are located. The data will not be shuffled in this code.

Exercise 2. RDD transformations warm ups.

Here are some more examples of Spark transformations and actions. For each task below, we've provided a few different implementations. Read each example and discuss the differences. Is one implementation better than the other or are the differences cosmetic? You may wish to discuss:

- the format of the data after each transformation
- memory usage (on executor nodes & in the driver)
- time complexity
- amount of network transfer
- whether or not the data will get shuffled
- coding efficiency & readability

Although we're working with tiny demo examples for now, try to imagine how the same code would operate if we were running a large job on a cluster. To aid in your analysis, navigate to the Spark UI (available at http://localhost:4040). To start, you should see a single job -- the job from Exercise 1. Click on the job description to view the DAG for that job. Check back with this UI as you run each version of the tasks below (**Note**: *the stages tab may be particularly helpful when making your comparisons*).

a) Multiples of 5 and 7

In [17]:
```python
# VERSION 1
dataRDD = sc.parallelize(range(1,100))
fivesRDD = dataRDD.filter(lambda x: x % 5 == 0)
sevensRDD = dataRDD.filter(lambda x: x % 7 == 0)
result = fivesRDD.intersection(sevensRDD)
result.collect()
```

Out[17]:
```
[35, 70]
```

In [18]:
```
# VERSION 2
dataRDD = sc.parallelize(range(1,100))
result = dataRDD.filter(lambda x: x % 5 == 0)\
                .filter(lambda x: x % 7 == 0)
result.collect()
```

Out[18]:
```
[35, 70]
```

In [19]:
```
# VERSION 3
dataRDD = sc.parallelize(range(1,100))
result = dataRDD.filter(lambda x: x % 7 == 0 and x % 5 == 0)
result.collect()
```

Out[19]:
```
[35, 70]
```

> **DISCUSSION QUESTION:**
>
> - What is the task here? Compare/contrast these three implementations.
> - Which of these versions require a shuffle? How do you know?

INSTRUCTOR TALKING POINTS

> **notes:** Version 1 requires a shuffle, the others don't.
> Versions 2 and 3 are exectly the same. Spark performs several optimizations, such as **"pipelining"** filter, map, and other such transformations together to merge them. Pipelining allows for a single pass over the data whenever possible instead of creating new RDDs. Pipelining occurs when RDDs can be computed from their parents without data movement.
>
> **debugging note:** *when you are first starting in Spark its a good idea to debug each step in isolation. If you are writing in the style of Version 1 you can do this by calling `collect()` on each RDD, if you are coding in the style of Version 2 you would comment out each line then uncomment one by one.*

b) Pig Latin Translator

In [20]:
```
poem = ["A bear however hard he tries",
        "Grows tubby without exercise",
        "said AA Milne"]
```

In [21]:

```
# VERSION 1
def translate(sent):
    words = [w[1:] + w[0] + '-ay' for w in sent.lower().split()]
    return ' '.join(words)

poemRDD = sc.parallelize(poem)
result = poemRDD.map(translate)\
                .reduce(lambda x,y: x + ' ' + y)
print(result)
```

```
a-ay earb-ay oweverh-ay ardh-ay eh-ay riest-ay rowsg-ay ubbyt-ay ithoutw-ay xercisee-ay aids-ay aa-ay ilnem-ay
```

In [22]:

```
# VERSION 2
def translate(wrd):
    return wrd[1:] + wrd[0] + '-ay'

poemRDD = sc.parallelize(poem)
result = poemRDD.flatMap(lambda x: x.lower().split())\
                .map(translate)\
                .reduce(lambda x,y: x + ' ' + y)
print(result)
```

```
a-ay earb-ay oweverh-ay ardh-ay eh-ay riest-ay rowsg-ay ubbyt-ay ithoutw-ay xercisee-ay aids-ay aa-ay ilnem-ay
```

> **DISCUSSION QUESTION:** What is the task here? Compare/contrast these two implementations.

INSTRUCTOR TALKING POINTS

> **main point:** `map()` vs. `flatMap()`
> **notes:** Version 1 is translating by phrase, Version 2 by word.
> The difference here is *mostly* cosmetic.
> V2 causes more records to be created, but ultimately the same amount of data has to be sent across the network either way. The extra map in V2 happens locally on each executor by default (yay Spark opimizations) & `flatMap` returns a generator so that extra step is not actually increasing the computational complexity of this solution. **syntax tip:** *If you are new to pyspark, take a moment to note the syntax in both versions where we pass our python translation function directly to the mapper.*

c) Average Monthly Purchases

In [30]:
```
shoppingList = ["JAN: 5 apples, 15 oranges",
                "FEB: 10 apples, 10 oranges",
                "MAR: 3 apples, 1 oranges",
                "APR: 6 apples, 2 oranges"]
```

In [31]:
```
# helper function
def parseShopping(line):
    """Parse each month's shopping list string into a key-value iterator."""
    month, items = line.split(':')
    items = [item.strip().split(' ') for item in items.split(',')] #can purchase many items per month
    return [(i[1], int(i[0])) for i in items]
```

In [32]:
```
# VERSION 1  (example 4-7 from Learning Spark)
shoppingRDD = sc.parallelize(shoppingList)
result = shoppingRDD.flatMap(lambda x: parseShopping(x))\
                    .mapValues(lambda x: (x,1))\
                    .reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))\
                    .mapValues(lambda x: x[0]/float(x[1]))
result.collect()
```

Out[32]:
```
[('avocados', 6.0), ('apples', 6.0), ('oranges', 7.0), ('grapes', 10.0)]
```

In [21]:
```
# VERSION 2 (example 4-12 from Learning Spark)
shoppingRDD = sc.parallelize(shoppingList)
result = shoppingRDD.flatMap(lambda x: parseShopping(x))\
                    .combineByKey(lambda x: (x,1),       # action for new key
                                  lambda x, y: (x[0] + y, x[1] + 1), # action for repeat key
                                  lambda x, y: (x[0] + y[0], x[1] + y[1]))\
                    .mapValues(lambda x: x[0]/float(x[1]))
result.collect()
```

Out[21]:
```
[('apples', 6.0), ('oranges', 7.0)]
```

In [22]:
```
# VERSION 3
shoppingRDD = sc.parallelize(shoppingList)
result = shoppingRDD.flatMap(lambda x: parseShopping(x))\
                    .groupByKey()\
                    .mapValues(lambda x: sum(x)/float(len(x)))
result.collect()
```

```
Out[22]: [('apples', 6.0), ('oranges', 7.0)]
```

Extra quiz: different items can be purchased every month

Here the number of unique items purchased per month is allowed to vary (above the same items were purchased per month). This will change the average calculations. Please adapt the above code to address this.

```
In [ ]:
shoppingList = ["JAN: 5 apples, 15 oranges",
                "FEB: 10 apples, 10 oranges, 10 grapes, 6 avocados",
                "MAR: 3 apples, 1 oranges",
                "APR: 6 apples, 2 oranges"]
```

```
In [ ]:
# VERSION 3: modify this version to work with the new dataset
shoppingRDD = sc.parallelize(shoppingList)
result = shoppingRDD.flatMap(lambda x: parseShopping(x))\
                    .groupByKey()\
                    .mapValues(lambda x: sum(x)/float(len(x)))
result.collect()
```

```
In [33]:
# VERSION 3: Solution: modify this version to work with the new dataset
shoppingRDD = sc.parallelize(shoppingList)
num_of_months  = shoppingRDD.count()
result = shoppingRDD.flatMap(lambda x: parseShopping(x))\
                    .groupByKey()\
                    .mapValues(lambda x: sum(x)/float(num_of_months))
result.collect()
```

```
Out[33]: [('avocados', 1.5), ('apples', 6.0), ('oranges', 7.0), ('grapes', 2.5)]
```

> **DISCUSSION QUESTION:** What is the task here? Compare/contrast these three implementations.

INSTRUCTOR TALKING POINTS

> **main point:** Combine early shuffle late.
> **notes:** V1 and V2 are equivalent. V3 is worse on large data.
> Spark automatically applies `reduceByKey()` and `combineByKey()` locally before shuffling anything.
> In V3 the groupByKey call would cause the items to be shuffled which is a lot more network traffic than the alternative solutions in which we aggregate by key on each executor before doing any global aggregations.
>
> Many sources—including the Spark documentation—warn against the scalability of the **groupByKey** function, which returns an iterator of each element by key. **groupByKey** is known to cause memory errors at scale. The reason is that the "groups" created by **groupByKey** are always iterators, which can't be distributed. This causes an expensive "shuffled read" step in which Spark has to read all of the shuffled data from disk and into memory. *From High Perfomance Spark, p.134*

Exercise 3. cache()-ing

In exercise 2 you saw how Spark builds an execution plan (DAG) so that transformations are evaluated lazily when triggerd by an action. In more complex DAGs you may need to reuse the contents of an RDD for multiple downstream operations. In such cases we'd like to avoid duplicating the computation of that intermediate result. Spark offers a few different options to persist an RDD in memory on the executor node where it is stored. Of these the most common is `cache()` (you'll read about others next week in ch 5 from *High Performance Spark*). Lets briefly look at how to `cache()` an RDD and discus when doing so is to your advantage.

In [34]:
```
# initialize data
dataRDD = sc.parallelize(np.random.random_sample(1000))
```

In [35]:
```
# perform some transformations
data2X= dataRDD.map(lambda x: x*2)
dataGreaterThan1 = data2X.filter(lambda x: x > 1.0)
cachedRDD = dataGreaterThan1.cache()
```

In [36]:
```
# count results less than 1
cachedRDD.filter(lambda x: x<1).count()
```

Out[36]:
```
0
```

In [37]:
```
# count results greater than 1
cachedRDD.filter(lambda x: x>1).count()
```

Out[37]:
```
522
```

In [38]:
```
# look at 10 results
for line in cachedRDD.take(10):
    print(line)
```

```
1.3542076638807559
1.958842491283623
1.7817491677259671
1.190189681491528
1.7047154547691594
1.2994010957787439
1.6596271948894428
1.3126646921992908
1.484598226302023
1.356892953132231
```

In [39]:
```
# look at top 10 results
for line in cachedRDD.top(10):
    print(line)
```

```
1.9989376577481401
1.997440066150761
1.9937366082397088
1.9933716179506606
1.9927211532857634
1.9920008858648233
1.9902955420424768
1.9898312696913971
1.9889284964789986
1.9855927510628304
```

In [40]:
```
# look at top 10 results
for line in cachedRDD.takeOrdered(10):
    print(line)
```

```
1.000451481529606
1.0008437606433485
1.0008657531387548
```

```
1.0030093660551767
1.005914458754175
1.0059833219762762
1.0061868559084255
1.0087988066366431
1.0110822315271633
1.012319625274343
```

> **DISCUSSION QUESTIONS:**
>
> - How many total actions are there in the 7 cells above?
> - If we hadn't cached the `dataGreaterThan1` RDD what would happen each time we call an action?
> - How does `cache()` change what the framework does?
> - When does it *not* make sense to `cache()` an intermediate result?

INSTRUCTOR TALKING POINTS

- How many total actions are there in the 7 cells above?

  > This code includes 4 transformations: `map()` , `filter()` , `filter()` , `filter()` , and 5 actions: `count()` , `count()` , `take()` , `top()` , `takeOrdered()`

- If we hadn't cached the dataGreaterThan1 RDD what would happen each time we call an action?

  > The DAG would re run from scratch.

- How does cache() change what the framework does?

  > It holds partial results in memory on the nodes where that data was computed so that subsequent transformations don't get recomputed from scratch.
  >
  > Persisting an RDD means materializing an RDD (usually by storing it in-memory onthe executors), for reuse during the current job. Spark remembers a persisted RDD's lineage so that it can recompute it for the duration of a Spark job if one of the persisted partitions is lost. (cache() is equivalent to per sist(), which is equivalent to persist("MEMORY_ONLY").)

- When does it not make sense to cache() an intermediate result?

  > Cache()-ing makes sense when an intermediate result will be used in multiple downstream transformations.

MORE INFO: *High Performance Spark, p. 116: Types of Reuse: Cache, Persist, Checkpoint, Shuffle Files*

```
In [45]:
# initialize data
dataRDD = sc.parallelize(np.random.random_sample(100_000))
#print(f"why is {dataRDD.mean()} not equal {dataRDD.mean()}")
dataRDD.mean()
```

```
Out[45]: 0.5007285612915016
```

```
In [46]:
dataRDD.mean()
```

```
Out[46]: 0.5007285612915016
```

Exercise 4. broadcast()-ing

Another challenge we faced when designing Hadoop MapReduce jobs was the challenge of making key pieces of information available to multiple nodes so that certain computations can happen in parallel. In Hadoop Streaming we resolved this challenge using custom partition keys and the order inversion pattern. In Spark we'll use broadcast variables -- read only objects that Spark will ship to all nodes where they're needed. Here's a brief example of how to create and access a broadcast variable.

Run the following cell to create our sample data files: a list of customers & a list of cities.

```
In [47]:
#%%writefile data/customers.csv
customers = '''Quinn Frank,94703
Morris Hardy,19875
Tara Smith,12204
Seth Mitchell,38655
Finley Cowell,10005
Cory Townsend,94703
Mira Vine,94016
Lea Green,70118
V Neeman,16604
Tvei Qin,70118'''

!echo "{customers}" | gsutil cp - {DEMO04_FOLDER}/data/customers.csv
```

```
Copying from <STDIN>...
/ [1 files][    0.0 B/    0.0 B]
Operation completed over 1 objects.
```

In [48]:

```
#%%writefile data/zipCodes.csv
zip_codes = '''94703,Berkeley,CA
94016,San Francisco,CA
10005,New York,NY
12204,Albany,NY
38655,Oxford,MS
70118,New Orleans,LA'''

!echo "{zip_codes}" | gsutil cp - {DEMO04_FOLDER}/data/zipCodes.csv
```

```
Copying from <STDIN>...
/ [1 files][    0.0 B/    0.0 B]
Operation completed over 1 objects.
```

Spark Job to count customers by state.

In [49]:

```
# load customers into RDD
dataRDD = sc.textFile(f'{DEMO04_FOLDER}/data/customers.csv')
```

In [51]:

```
# create a look up dictionary to map zip codes to state abbreviations

zipCodes = sc.textFile(f'{DEMO04_FOLDER}/data/zipCodes.csv')\
             .map(lambda l: tuple(l.split(',')))\
             .collect()
# zipCode-state map
zipCodes = {item[0]: f'{item[2]}' for item in zipCodes} # Dictionary comprehension
print(f"zipCodes: {zipCodes}")
zipCodes = sc.broadcast(zipCodes) # broadcast zipCode-state map
#access via zipCodes.value
```

```
zipCodes: {'94703': 'CA', '94016': 'CA', '10005': 'NY', '12204': 'NY', '38655': 'MS', '70118': 'LA'}
```

In [36]:

```
# Customer data record is name, zip
#   E.g., Morris Hardy,19875
# TASK count number of customers by state
result = dataRDD.map(lambda x: x.split(',')[1])\
                .map(lambda x: (zipCodes.value.get(x,'n/a'),1))\
                 .reduceByKey(lambda a, b: a + b)
# expected result [('CA', 3), ('NY', 2), ('LA', 2), ('n/a', 2), ('MS', 1)]
```

In [37]:
```
# take a look
result.collect()
```

Out[37]:
```
[('CA', 3), ('NY', 2), ('LA', 2), ('n/a', 2), ('MS', 1)]
```

> **DISCUSSION QUESTIONS:**
>
> - What does broadcasting achieve here?
> - Why not just encapsulate our variables in a function closure instead?
> - When would it be a bad idea to broadcast a supplemental table like our list of zip codes?
> - Note that we are working in local mode through out this notebook. What happens if you comment out the line where we broadcast the zip code dictionary? What would happen if you were working on a cluster?

INSTRUCTOR TALKING POINTS

- What does broadcasting achieve here?

  > Broadcasting makes a piece of information available to all the executors -- in this case it sends the look up dictionary of zipCodes to each node so that that dictionary can be used in the second map.

- Why not just encapsulate our variables in a function closure instead?

  > One way to use a variable in your driver node inside your tasks is to simply reference it in your function closures (e.g., in a `map` operation), but this can be inefficient, especially for large variables such as a lookup table or a machine learning model. The reason for this is that when you use a variable in a closure, it must be deserialized on the worker nodes many times (one per task). Moreover, if you use the same variable in multiple Spark actions and jobs, it will be re-sent to the workers with every job instead of once.

- When would it be a bad idea to broadcast a supplemental table like our list of zip codes?

  > We wouldn't want to do this for really large files/objects.

- Note that we are working in local mode through out this notebook. What happens if you comment out the line where we broadcast the zip code dictionary? What would happen if you were working on a cluster?

> In local mode the driver is running in the same JVM as the executors so the code actually runs fine even if we comment out the broadcast. On a cluster, the zipCodes would be sent to each task via encapsulation resulting in a less efficient implementation.

`Syntax Note:` *If you are new to pyspark pay careful attention to how we access the dictionary that is broadcast.*

Exercise 5. Accumulators

Accumulators are Spark's equivalent of Hadoop counters. Like broadcast variables they represent shared information across the nodes in your cluster, but unlike broadcast variables accumulators are *write-only* ... in other words **you can only access their values in the driver program and not on your executors** (where transformations are applied). As convenient as this sounds, there are a few common pitfalls to avoid. Let's take a look.

Run the following cell to create a sample data file representing a list of `studentID, courseID, final_grade` ...

In [55]:
```
#%%writefile data/grades.csv
# studentID, courseID, final_grade
grades = '''10001,101,98
10001,102,87
10002,101,75
10002,102,55
10002,103,80
10003,102,45
10003,103,75
10004,101,90
10005,101,85
10005,103,60'''

!echo "{grades}" | gsutil cp - {DEMO04_FOLDER}/data/grades.csv
```

```
Copying from <STDIN>...
/ [1 files][    0.0 B/    0.0 B]
Operation completed over 1 objects.
```

Detour EDA in Pandas

In [ ]:
```
# Some EDA in pandas just see what we are looking at
df = pd.DataFrame([x.split(',') for x in grades.split('\n')], columns=["studentID", "courseID", "final_grade"])
df['final_grade'] = df['final_grade'].astype(int)
df.loc[df.final_grade<65]
```

Suppose we want to compute the average grade by course and student while also tracking the number of failing grades awarded. We might try something like this:

In [53]:
```
# initialize an accumulator to track failing grades
nFailing = sc.accumulator(0)
```

In [54]:
```
# function to increment the accumulator as we read in the data
def parse_grades(line, accumulator):
    """Helper function to parse input & track failing grades."""
    student,course,grade = line.split(',')
    grade = int(grade)
    if grade < 65:
        accumulator.add(1)
    return(student,course, grade)
```

In [41]:
```
# compute averages in spark
gradesRDD = sc.textFile(f'{DEMO04_FOLDER}/data/grades.csv')\
              .map(lambda x: parse_grades(x, nFailing))
studentAvgs = gradesRDD.map(lambda x: (x[0], (x[2], 1)))\
                       .reduceByKey(lambda a, b: (a[0] + b[0], a[1] + b[1]))\
                       .mapValues(lambda x: x[0]/x[1])
courseAvgs = gradesRDD.map(lambda x: (x[1], (x[2], 1)))\
                      .reduceByKey(lambda a, b: (a[0] + b[0], a[1] + b[1]))\
                      .mapValues(lambda x: x[0]/x[1])
```

In [42]:
```
# take a look
print("===== average by student =====")
print(studentAvgs.collect())
print("===== average by course =====")
print(courseAvgs.collect())
print("===== number of failing grades awarded =====")
print(nFailing)
```

```
===== average by student =====
[('10001', 92.5), ('10004', 90.0), ('10002', 70.0), ('10003', 60.0), ('10005', 72.5)]
===== average by course =====
[('102', 62.333333333333336), ('101', 87.0), ('103', 71.66666666666667)]
===== number of failing grades awarded =====
6
```

> **DISCUSSION QUESTIONS:**
>
> - What is wrong with the results? (**`HINT:`** *how many failing grades are there really?*)
> - Why might this be happening? (**`HINT:`** *How many actions are there in this code? Which parts of the DAG are recomputed for each of these actions?*)
> - What one line could we add to the code to fix this problem?
>   - What could go wrong with our "fix"?
> - How could we have designed our parser differently to avoid this problem in the first place?

INSTRUCTOR TALKING POINTS

- What is wrong with the results? (HINT: how many failing grades are there really?)

  > There are actually only 3 failing grades but the accumulators seems to have found 6.

- Why might this be happening? (HINT: How many actions are there in this code? Which parts of the DAG are recomputed for each of these actions?)

  > We have two actions here -- one to print out `studentAvgs` and another to print `courseAvgs` . Since both of these RDDs are built from `gradesRDD` the code to create `gradesRDD` , which includes the map function where our accumulator is incremented, gets run twice. We're double counting the failing grades.

- What one line could we add to the code to fix this problem?

  > If we add a `cache()` statement to the end of the code for `gradesRDD` we could instruct Spark to only compute that intermediate RDD once which would prevent the double counting problem.

- What could go wrong with our fix?

> If the transformation task failed and had to be rerun, the accumulator would be incremented again. For accumulator updates performed inside **actions only**, Spark guarantees that each task's update to the accumulator will only be applied once, i.e. restarted tasks will not update the value. In transformations, users should be aware of that each task's update may be applied more than once if tasks or job stages are re-executed. For more on accumulators, see also the **foreach()** action.

- How could we have designed our parser differently to avoid this problem in the first place?

> Alternately, if we knew the two downstream tasks from the outset, we could have written a parser that emits two records per line: one for the student one for the course... that way we could use a composite key to compute the averages for courses and students in a single job and the accumulator would only get incremented once.

In [67]:

```
#reduce
my_count = sc.accumulator(0)
```

In [71]:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('detour').getOrCreate()
data=[("Z", 1),("A", 20),("B", 30),("C", 40),("B", 30),("B", 60)]
inputRDD = spark.sparkContext.parallelize(data)

listRdd = spark.sparkContext.parallelize([1,2,3,4,5,3,2])

#aggregate
seqOp = (lambda x, y: x + y)
combOp = (lambda x, y: x + y)
agg=listRdd.aggregate(0, seqOp, combOp)
print(agg) # output 20

#aggregate 2
seqOp2 = (lambda x, y: (x[0] + y, x[1] + 1))
combOp2 = (lambda x, y: (x[0] + y[0], x[1] + y[1]))
agg2=listRdd.aggregate((0, 0), seqOp2, combOp2)
print(agg2) # output (20,7)

agg2=listRdd.treeAggregate(0,seqOp, combOp)
print(agg2) # output 20

#fold
from operator import add
foldRes=listRdd.fold(0, add)
print(foldRes) # output 20
```

```
#reduce
#my_count = sc.accumulator(0)
def my_add(x, y):
    my_count.add(1)
    return x + y
redRes=listRdd.reduce(my_add)
print(f"my_count should be 7:{my_count} ") # output
print(f"my_add:{redRes}") # output 20
redRes=listRdd.reduce(add)
print(redRes) # output 20

#treeReduce. This is similar to reduce
add = lambda x, y: x + y
redRes=listRdd.treeReduce(add)
print(redRes) # output 20

#Collect
data = listRdd.collect()
print(data)

#count, countApprox, countApproxDistinct
print("Count : "+str(listRdd.count()))
#Output: Count : 20
print("countApprox : "+str(listRdd.countApprox(1200)))
#Output: countApprox : (final: [7.000, 7.000])
print("countApproxDistinct : "+str(listRdd.countApproxDistinct()))
#Output: countApproxDistinct : 5
print("countApproxDistinct : "+str(inputRDD.countApproxDistinct()))
#Output: countApproxDistinct : 5

#countByValue, countByValueApprox
print("countByValue :  "+str(listRdd.countByValue()))


#first
print("first :  "+str(listRdd.first()))
#Output: first :  1
print("first :  "+str(inputRDD.first()))
#Output: first :  (Z,1)

#top
print("top : "+str(listRdd.top(2)))
#Output: take : 5,4
```

```
print("top : "+str(inputRDD.top(2)))
#Output: take : (Z,1),(C,40)

#min
print("min :  "+str(listRdd.min()))
#Output: min :  1
print("min :  "+str(inputRDD.min()))
#Output: min :  (A,20)

#max
print("max :  "+str(listRdd.max()))
#Output: max :  5
print("max :  "+str(inputRDD.max()))
#Output: max :  (Z,1)

#take, takeOrdered, takeSample
print("take : "+str(listRdd.take(2)))
#Output: take : 1,2
print("takeOrdered : "+ str(listRdd.takeOrdered(2)))
#Output: takeOrdered : 1,2
print("take : "+str(listRdd.takeSample()))
```

```
20
(20, 7)
20
20
my_count should be 7:19
my_add:20
20
20
[1, 2, 3, 4, 5, 3, 2]
Count : 7
countApprox : 7
countApproxDistinct : 5
countApproxDistinct : 5
countByValue :  defaultdict(<class 'int'>, {1: 1, 2: 2, 3: 2, 4: 1, 5: 1})
first :  1
first :  ('Z', 1)
top : [5, 4]
top : [('Z', 1), ('C', 40)]
min :  1
min :  ('A', 20)
max :  5
max :  ('Z', 1)
take : [1, 2]
takeOrdered : [1, 2]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[71], line 91
     89 print("takeOrdered : "+ str(listRdd.takeOrdered(2)))
     90 #Output: takeOrdered : 1,2
---> 91 print("take : "+str(listRdd.takeSample()))

TypeError: takeSample() missing 2 required positional arguments: 'withReplacement' and 'num'
```

Exercise 6. WordCount & Naive Bayes Reprise

We'll wrap up today's demo by revisiting two tasks from weeks 1-2. Compare each of these Spark implementations to the approach we took when performing the same task in Hadoop MapReduce.

a) Word Count in Spark

In [72]:

```
# load the data into Spark
aliceRDD = sc.textFile(ALICE_TXT)
```

In [73]:

```
# perform wordcount
result = aliceRDD.flatMap(lambda line: re.findall('[a-z]+', line.lower())) \
                 .map(lambda word: (word, 1)) \
                 .reduceByKey(lambda a, b: a + b)\
                 .cache()
```

In [74]:

```
# take a look at the top 10 (by alphabet)
result.takeOrdered(10)
```

```
Out[74]: [('a', 695),
          ('abide', 2),
          ('able', 1),
          ('about', 102),
          ('above', 3),
          ('absence', 1),
          ('absurd', 2),
          ('accept', 1),
          ('acceptance', 1),
          ('accepted', 2)]
```

```
In [75]: # take a look at the top 10 (by count)
         result.takeOrdered(10, key=lambda x: -x[1])
```

```
Out[75]: [('the', 1839),
          ('and', 942),
          ('to', 811),
          ('a', 695),
          ('of', 638),
          ('it', 610),
          ('she', 553),
          ('i', 546),
          ('you', 486),
          ('said', 462)]
```

```
In [76]: # what does Spark consider the 'top'?
         result.top(10)
```

```
Out[76]: [('zip', 1),
          ('zigzag', 1),
          ('zealand', 1),
          ('youth', 6),
          ('yourself', 10),
          ('yours', 3),
          ('your', 71),
          ('young', 5),
          ('you', 486),
          ('yet', 25)]
```

b.1) Pipelining and Debugging in PySpark

pdb

- type q to quit the debugger

In [77]:
```
x = 1
import pdb; pdb.set_trace()  #type q to quit the debugger
print(x)
```

```
--Return--
None
> /tmp/ipykernel_16346/3694824765.py(2)<module>()
      1 x = 1
----> 2 import pdb; pdb.set_trace()  #type q to quit the debugger
      3 print(x)


1

1

1
```

In [110…
```
def split_into_words(line):
    """ mapper"""
    return re.findall('[a-z]+', line.lower())
split_into_words("this is a line of text") #unit for the mapper
```

Out[110…
```
['this', 'is', 'a', 'line', 'of', 'text']
```

In [79]:
```
# perform wordcount
result = aliceRDD.flatMap(split_into_words) \
                 .map(lambda word: (word, 1)).take(5) #\
                 #.reduceByKey(lambda a, b: a + b)\
                 #3.take(5)
result
```

Out[79]:
```
[('the', 1), ('project', 1), ('gutenberg', 1), ('ebook', 1), ('of', 1)]
```

In [113…
```
aliceRDD.count()
```

Out[113…
```
3761
```

In [119…

```
# perform wordcount
result = aliceRDD.map(lambda line: re.findall('[a-z]+', line.lower())).collect()
print(aliceRDD.count())
print(type(result))
result[:3]
```

```
3761
<class 'list'>
```

Out[119…

```
[['the',
  'project',
  'gutenberg',
  'ebook',
  'of',
  'alice',
  's',
  'adventures',
  'in',
  'wonderland',
  'by',
  'lewis',
  'carroll'],
 [],
 ['this',
  'ebook',
  'is',
  'for',
  'the',
  'use',
  'of',
  'anyone',
  'anywhere',
  'in',
  'the',
  'united',
  'states',
  'and']]
```

In [122…

```
# perform wordcount
result = aliceRDD.flatMap(lambda line: re.findall('[a-z]+', line.lower())).collect()
print(aliceRDD.count())
print(type(result))
result[:3]
```

```
3761
<class 'list'>
```

Out[122…
```
['the', 'project', 'gutenberg']
```

In [123…
```
# perform wordcount
result = aliceRDD.flatMap(lambda line: re.findall('[a-z]+', line.lower())) \
                 .map(lambda word: (word, 1)).take(5) #\
                 #.reduceByKey(lambda a, b: a + b)\
                 #3.take(5)
result
```

Out[123…
```
[('the', 1), ('project', 1), ('gutenberg', 1), ('ebook', 1), ('of', 1)]
```

In [126…
```
# perform wordcount
result = aliceRDD.flatMap(lambda line: re.findall('[a-z]+', line.lower())) \
                 .map(lambda word: (word, 1)) \
                 .reduceByKey(add).take(5)
                 #.reduceByKey(lambda a, b: a + b).take(5)
result
```

Out[126…
```
[('project', 88), ('gutenberg', 98), ('ebook', 13), ('of', 638), ('s', 222)]
```

In [127…
```
# perform wordcount
result = aliceRDD.flatMap(lambda line: re.findall('[a-z]+', line.lower())) \
                 .map(lambda word: (word, 1)) \
                 .reduceByKey(lambda a, b: a + b).take(5)
result
```

Out[127…
```
[('project', 88), ('gutenberg', 98), ('ebook', 13), ('of', 638), ('s', 222)]
```

In [81]:
```
# take a look at the top 10 (by alphabet)
result.takeOrdered(10) #what happened here?
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[81], line 2
      1 # take a look at the top 10 (by alphabet)
----> 2 result.takeOrdered(10)
```

```
AttributeError: 'list' object has no attribute 'takeOrdered'
```

In [95]:
```
type(result)
```

Out[95]:
```
list
```

In [82]:
```
# take a look at the top 10 (by count)
result.takeOrdered(10, key=lambda x: -x[1])
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[82], line 2
      1 # take a look at the top 10 (by count)
----> 2 result.takeOrdered(10, key=lambda x: -x[1])

AttributeError: 'list' object has no attribute 'takeOrdered'
```

In [ ]:
```
# what does Spark consider the 'top'?
result.top(10)
```

In [83]:
```
# perform wordcount top 10 most frequent
result = aliceRDD.flatMap(lambda line: re.findall('[a-z]+', line.lower())) \
                 .map(lambda word: (word, 1)) \
                 .reduceByKey(lambda a, b: a + b)\
                 .takeOrdered(10, key=lambda x: -x[1])
result
```

Out[83]:
```
[('the', 1839),
 ('and', 942),
 ('to', 811),
 ('a', 695),
 ('of', 638),
 ('it', 610),
 ('she', 553),
 ('i', 546),
 ('you', 486),
 ('said', 462)]
```

> **DICUSSION QUESTIONS:**

> - Compare/contrast this implementation to our Hadoop Streaming approach.
> - How many times does the data get shuffled?
> - What local aggregation will spark do?
> - What is the difference between `take()` and `top()` and `takeOrdered()` ? Is one more or less efficient than the others? Compare these actions to the work we had to do to sort and subset with multiple reducers in Hadoop MapReduce?
> - What would happen if we removed the `cache()` that follows the `reduceByKey()` ? [**`Hint:`** *this is kind of a trick question, but try rerunning the job & look at the Spark UI...*]

INSTRUCTOR TALKING POINTS

- Compare/contrast this implementation to our Hadoop Streaming approach.

  > The basic approach: tokenize, create counters, aggregate by key is identical to what we did in Hadoop MapReduce... but the code is dramatically shorter. Significantly, we don't need to specify any special configuration to access data with a secondary sort or to ensure load balancing, etc.

- How many times does the data get shuffled?

  > Once for the `reduceByKey()` operation. Note that this is different than the equivalent task in Hadoop - where word count w/ secondary sort required one shuffle for the aggregation and a second shuffle for the sorting by count -- to confirm this navigate to http://localhost:4040/jobs and look at the Shuffle Read/Write statistics.

- What local aggregation will Spark do?

  > When we use `reduceByKey()` Spark will automatically reduce locally before shuffling. This is one of the reasons we usually would rather use `reduceByKey` (or combine by key) instead of `groupByKey()` followed by a `map()` .

- What is the difference between `take()` and `top()` and `takeOrdered()` ? Is one more or less efficient than the others? Compare these actions to the work we had to do to sort and subset with multiple reducers in Hadoop MapReduce?

  > These are different ways to pull subsets of the results back into the driver program. `take()` returns the specified number of elements from the first RDD. `top()` returns the first 10 elemens using the default ordering (in this case reverse alphabetical by key). `takeOrdered()` has an optional second parameter that allows the user to define a custom ordering. What's interesting is that none of these actions trigger a shuffle (again, confirm using the UI)...

`takeOrdered()` gets the N elements from an RDD ordered in ascending order or as specified by the `optional` key function.

`RDD.takeOrdered(num, key=None)`

- What would happen if we removed the `cache()` that follows the `reduceByKey()`? [**`Hint:`** *this is kind of a trick question, but try rerunning the job & look at the Spark UI...*]

  > In theory removing the `cache()` would mean that each subsequent action causes the transformations to start from scratch - which would mean we do a shuffle each time which would be inefficient. However actually whenever you use a `ByKey()` transformation Spark automatically `caches()` the result (doing so is part of how it ensures the fault tolerance of the shuffle)... so actually removing the explicit instruction to `cache()` won't make a difference here.

b) Naive Bayes in Spark and broadcasting the model

Key takeaways

- Implement the algorithm (training & inference)
- Broadcasting the model

In [84]:
```
def parse(doc):
    """
    Helper Function to parse documents.
    # DocID  class   body_of_doc
    """
    docID, class_, subj, body = doc.lower().split('\t')
    return(class_, subj + " " + body)
```

In [85]:
```
def tokenize(class_, text):
    """
    Map text from a given class to word list with counts for each class.
    """
    # get words
    words = re.findall(r'[a-z]+', text)
    # emit a count for each class (0,1 or 1,0)
    class_counts = [1,0] if class_ =='0' else [0,1]
    return[(word, class_counts) for word in words] #emit (word, [1,0]) for each word
```

In [86]:
```
def NBtrain(dataRDD, smoothing = 1.0):
    """
    Function to train a Naive Bayes Model in Spark.
    Returns a dictionary.
    """
    # extract word counts
    docsRDD = dataRDD.map(parse) # (class_, subj+body)
    wordsRDD = docsRDD.flatMap(lambda x: tokenize(*x)).cache()\
                      .reduceByKey(lambda x,y: np.array(x) + np.array(y))\
                      .cache()
    # compute priors
    # RDD.countByKey: Count the number of elements for each key,
    # and return the result to the master as a dictionary
    docTotals = docsRDD.countByKey() #(class_, subj+body)
    priors = np.array([docTotals['0'], docTotals['1']])
    priors = priors/sum(priors)
    #import pdb; pdb.set_trace()

    # compute conditionals with smoothing =1 for laplace smoothing by default
    # smoothed word counts
    # for each class [tot1, tot2]
    wordTotals = sc.broadcast(wordsRDD.map(lambda x: x[1] + np.array([smoothing, smoothing]))\
                                      .reduce(lambda x,y: np.array(x) + np.array(y)))

    # smooth word counts again and then normalize to get Pr(word|class)
    cProb = wordsRDD.mapValues(lambda x: x + np.array([smoothing, smoothing]))\
                    .mapValues(lambda x: x/np.array(wordTotals.value))\
                    .collect()

    return dict([("ClassPriors", priors)] + cProb)
```

In [92]:
```
def NBclassify(document, model_dict):
    """
    Classify a document as ham/spam via Naive Bayes.
    Use logProbabilities to avoid floating point error.
    NOTE: this is just a python function, no distribution so
    we should expect our documents (& model) to fit in memory.
    """
    # get words
    words = re.findall(r'[a-z]+', document.lower())
    # compute log probabilities
    logProbs = [np.log(model_dict.get(wrd,[1,1])) for wrd in words]
    # return most likely class
    sumLogProbs = np.log(model_dict['ClassPriors']) + sum(logProbs)
    return np.argmax(sumLogProbs)
```

In [87]:

```
def evaluate(resultsRDD):
    """
    Compute accuracy, precision, recall an F1 score given a
    pairRDD of (true_class, predicted_class)
    """
    nDocs = resultsRDD.count()
    TP = resultsRDD.filter(lambda x: x[0] == '1' and x[1] == 1).count()
    TN = resultsRDD.filter(lambda x: x[0] == '0' and x[1] == 0).count()
    FP = resultsRDD.filter(lambda x: x[0] == '0' and x[1] == 1).count()
    FN = resultsRDD.filter(lambda x: x[0] == '1' and x[1] == 0).count()

    # report results
    print(f"Total # Documents:\t{nDocs}")
    print(f"True Positives:\t{TP}")
    print(f"True Negatives:\t{TN}")
    print(f"False Positives:\t{FP}")
    print(f"False Negatives:\t{FN}")
    print(f"Accuracy\t{(TP + TN)/(TP + TN + FP + FN)}")
    if (TP + FP) != 0:
        precision = TP / (TP + FP)
        print(f"Precision\t{precision}")

    if (TP + FN) != 0:
        recall = TP / (TP + FN)
        print(f"Recall\t{recall}")

    if TP != 0:
        f_score = 2 * precision * recall / (precision + recall)
        print(f"F-Score\t{f_score}")
```

Retrieve results.

In [88]:

```
# look at the raw training file
# DocID  class   body_of_doc
!gsutil cat {TRAIN_PATH}
```

```
D1      1               Chinese Beijing Chinese
D2      1               Chinese Chinese Shanghai
D3      1               Chinese Macao
D4      0               Tokyo Japan Chinese
```

In [89]:

```
# load data into Spark
trainRDD = sc.textFile(TRAIN_PATH)
testRDD = sc.textFile(TEST_PATH)
```

In [90]:

```
# train your model (& take a look)
NBmodel = NBtrain(trainRDD)
NBmodel
```

Out[90]:

```
{'ClassPriors': array([0.25, 0.75]),
 'chinese': array([0.22222222, 0.42857143]),
 'macao': array([0.11111111, 0.14285714]),
 'japan': array([0.22222222, 0.07142857]),
 'beijing': array([0.11111111, 0.14285714]),
 'shanghai': array([0.11111111, 0.14285714]),
 'tokyo': array([0.22222222, 0.07142857])}
```

In [93]:

```
# perform inference on a doc (just to test)
NBclassify("This Japan Tokyo Macao is Chinese", NBmodel)
```

Out[93]:

```
0
```

In [94]:

```
# evaluate your model
model_b = sc.broadcast(NBmodel)
resultsRDD = testRDD.map(parse)\
                    .mapValues(lambda x: NBclassify(x, model_b.value))
evaluate(resultsRDD)
```

```
Total # Documents:      4
True Positives: 2
True Negatives: 2
False Positives:        0
False Negatives:        0
Accuracy        1.0
Precision       1.0
Recall  1.0
F-Score 1.0
```

> **DICUSSION QUESTIONS:**
>
> - Compare/contrast this implementation to our Hadoop Streaming approach.

INSTRUCTOR TALKING POINTS

- Compare/contrast this implementation to our Hadoop Streaming approach.

  > Again, the basic architecture of the implementation should be recognizable... but the code itself is much more compact.

Example 6-14 in Learn Spark Book. Average with mapPartitions() in Python

mapPartitions

Similar to `map()` PySpark `mapPartitions()` is a **narrow** transformation operation that applies a function to each partition of the RDD.

NOTE: if you have a DataFrame, you need to convert to RDD in order to use it.

`mapPartitions()` is mainly used to initialize connections once for each partition instead of every row, this is the main difference between map() vs mapPartitions(). It is a narrow transformation as there will not be any data movement/shuffling between partitions to perform the function.

```
def f(partitionData):
  #perform heavy initializations like Databse connections
  for element in partitionData:
    # perform operations for element in a partition
  # return updated data
df.rdd.mapPartitions(f)
```

In [98]:
```
def combineCtrs(c1, c2):
    return (c1[0] + c2[0], c1[1] + c2[1])
def basicAvg(nums):
    """Compute the average"""
    nums.map(lambda num: (num, 1)).reduce(combineCtrs)

def partitionCtr(nums):
    """Compute sumCounter for partition"""
    sumCount = [0, 0]
    for num in nums:
        sumCount[0] += num
        sumCount[1] += 1
    return [sumCount]

def fastAvg(nums):
    """Compute the avg"""
    sumCount = nums.mapPartitions(partitionCtr).reduce(combineCtrs)
    return sumCount[0] / float(sumCount[1])

nums = fastAvg(sc.parallelize([1,2,3,4]))
print(nums)
```

```
2.5
```

In [103…

```
rdd = sc.parallelize(range(100))
print(rdd.mapPartitions(partitionCtr).collect())
print(rdd.mapPartitions(partitionCtr).reduce(combineCtrs))
sumCount = rdd.mapPartitions(partitionCtr).reduce(combineCtrs)
sumCount[0] / float(sumCount[1])
```

```
[[300, 25], [925, 25], [1550, 25], [2175, 25]]
(4950, 100)
```

Out[103…

```
49.5
```

In [108…

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('DataFrame').getOrCreate()
data = [('James','Smith','M',3000),
  ('Anna','Rose','F',4100),
  ('Robert','Williams','M',6200),
]

columns = ["firstname","lastname","gender","salary"]
df = spark.createDataFrame(data=data, schema = columns)
df.show()
```

```
+---------+--------+------+------+
|firstname|lastname|gender|salary|
+---------+--------+------+------+
|    James|   Smith|     M|  3000|
|     Anna|    Rose|     F|  4100|
|   Robert|Williams|     M|  6200|
|     Anna|    Rose|     F|  4100|
|   Robert|Williams|     M|  6200|
|     Anna|    Rose|     F|  4100|
|   Robert|Williams|     M|  6200|
+---------+--------+------+------+
```

Now use the PySpark mapPartitions() transformation to concatenate the firstname, lastname and calculate the bonus as 10% of the value of salary column.

In [109…

```
def reformat(partitionData):
    for row in partitionData:
        yield [row.firstname+","+row.lastname,row.salary*10/100]

df.rdd.mapPartitions(reformat).collect()
```

Out[109…

```
[['James,Smith', 300.0],
 ['Anna,Rose', 410.0],
 ['Robert,Williams', 620.0],
 ['Anna,Rose', 410.0],
 ['Robert,Williams', 620.0],
 ['Anna,Rose', 410.0],
 ['Robert,Williams', 620.0]]
```

In [106…

```
# This function calls for each partition
def reformat(partitionData):
    for row in partitionData:
        yield [row.firstname+","+row.lastname,row.salary*10/100]

df2=df.rdd.mapPartitions(reformat).toDF(["name","bonus"])
df2.show()
```

```
+---------------+-----+
|           name|bonus|
+---------------+-----+
|    James,Smith|300.0|
|      Anna,Rose|410.0|
|Robert,Williams|620.0|
+---------------+-----+
```

In [ ]:

Relative Frequencies example

pyspark.RDD.repartitionAndSortWithinPartitions

Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys.

- https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.RDD.repartitionAndSortWithinPartitions.html

```
rdd = sc.parallelize([(0, 5), (3, 8), (2, 6), (0, 8), (3, 8), (1, 3)])
rdd2 = rdd.repartitionAndSortWithinPartitions(2, lambda x: x % 2, True)
rdd2.glom().collect()
[[(0, 5), (0, 8), (2, 6)], [(1, 3), (3, 8), (3, 8)]]
```

Relative Frequencies

Data Intensive Text Processing

Lin and Dyer

3.3 COMPUTING RELATIVE FREQUENCIES

(ported to Spark RDD API)

In [1]:
```
import re
import ast
import time
import itertools
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

In [2]:
```
# start Spark Session (RUN THIS CELL AS IS)
from pyspark.sql import SparkSession
app_name = "relative_frequencies"
master = "local[*]"
spark = SparkSession\
        .builder\
        .appName(app_name)\
        .master(master)\
        .getOrCreate()
sc = spark.sparkContext
```

```
:: loading settings :: url = jar:file:/usr/lib/spark/jars/ivy-2.4.0.jar!/org/apache/ivy/core/settings/ivysettings.xml
Ivy Default Cache set to: /root/.ivy2/cache
The jars for the packages stored in: /root/.ivy2/jars
graphframes#graphframes added as a dependency
org.apache.spark#spark-avro_2.12 added as a dependency
:: resolving dependencies :: org.apache.spark#spark-submit-parent-dc8d1e2d-7271-4e3e-9891-0065c181304f;1.0
        confs: [default]
        found graphframes#graphframes;0.8.2-spark3.1-s_2.12 in spark-packages
        found org.slf4j#slf4j-api;1.7.16 in central
        found org.apache.spark#spark-avro_2.12;3.1.3 in central
        found org.spark-project.spark#unused;1.0.0 in central
:: resolution report :: resolve 350ms :: artifacts dl 9ms
        :: modules in use:
        graphframes#graphframes;0.8.2-spark3.1-s_2.12 from spark-packages in [default]
        org.apache.spark#spark-avro_2.12;3.1.3 from central in [default]
        org.slf4j#slf4j-api;1.7.16 from central in [default]
        org.spark-project.spark#unused;1.0.0 from central in [default]
        ---------------------------------------------------------------------
        |                  |            modules            ||   artifacts   |
        |       conf       | number| search|dwnlded|evicted|| number|dwnlded|
        ---------------------------------------------------------------------
        |      default     |   4   |   0   |   0   |   0   ||   4   |   0   |
        ---------------------------------------------------------------------
:: retrieving :: org.apache.spark#spark-submit-parent-dc8d1e2d-7271-4e3e-9891-0065c181304f
        confs: [default]
        0 artifacts copied, 4 already retrieved (0kB/9ms)
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
23/01/31 23:40:54 INFO org.apache.spark.SparkEnv: Registering MapOutputTracker
23/01/31 23:40:54 INFO org.apache.spark.SparkEnv: Registering BlockManagerMaster
23/01/31 23:40:54 INFO org.apache.spark.SparkEnv: Registering BlockManagerMasterHeartbeat
23/01/31 23:40:54 INFO org.apache.spark.SparkEnv: Registering OutputCommitCoordinator
23/01/31 23:40:54 WARN org.apache.spark.util.Utils: Service 'SparkUI' could not bind on port 4040. Attempting port 4041.
```

In [3]:

```
%%time
DATA_toy = sc.parallelize(['dog aardvark pig banana','bear zebra pig'])
DATA_toy.glom().collect()
# using glom, we can see that there were as many partitions created as there are cores on this machine
```

```
[Stage 0:>                                                          (0 + 4) / 4]
```

```
CPU times: user 13.1 ms, sys: 4.86 ms, total: 17.9 ms
Wall time: 1.99 s
```

```
Out[3]: [[], ['dog aardvark pig banana'], [], ['bear zebra pig']]
```

```
In [4]:
DATA_F1 = sc.textFile("/media/notebooks/Assignments/HW3/master/data/googlebooks-eng-all-5gram-20090715-0-filtered.txt")
            .map(lambda x: x.split('\t')[0]).cache()
DATA_F1.take(5)
```

```
23/01/31 23:41:25 WARN org.apache.hadoop.util.concurrent.ExecutorHelper: Thread (Thread[GetFileInfo #0,5,main]) interrup
ted:
java.lang.InterruptedException
        at com.google.common.util.concurrent.AbstractFuture.get(AbstractFuture.java:510)
        at com.google.common.util.concurrent.FluentFuture$TrustedFuture.get(FluentFuture.java:88)
        at org.apache.hadoop.util.concurrent.ExecutorHelper.logThrowableFromAfterExecute(ExecutorHelper.java:48)
        at org.apache.hadoop.util.concurrent.HadoopThreadPoolExecutor.afterExecute(HadoopThreadPoolExecutor.java:90)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1157)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
        at java.lang.Thread.run(Thread.java:750)
```

```
---------------------------------------------------------------------------
Py4JJavaError                             Traceback (most recent call last)
Cell In[4], line 3
      1 DATA_F1 = sc.textFile("/media/notebooks/Assignments/HW3/master/data/googlebooks-eng-all-5gram-20090715-0-filtere
d.txt")\
      2             .map(lambda x: x.split('\t')[0]).cache()
----> 3 DATA_F1.take(5)

File /usr/lib/spark/python/pyspark/rdd.py:1533, in RDD.take(self, num)
   1509 """
   1510 Take the first num elements of the RDD.
   1511
   (...)
   1530 [91, 92, 93]
   1531 """
   1532 items = []
-> 1533 totalParts = self.getNumPartitions()
   1534 partsScanned = 0
   1536 while len(items) < num and partsScanned < totalParts:
   1537     # The number of partitions to try in this iteration.
   1538     # It is ok for this number to be greater than totalParts because
   1539     # we actually cap it at totalParts in runJob.

File /usr/lib/spark/python/pyspark/rdd.py:2935, in PipelinedRDD.getNumPartitions(self)
   2934 def getNumPartitions(self):
```

```
-> 2935     return self._prev_jrdd.partitions().size()

File /opt/conda/miniconda3/lib/python3.8/site-packages/py4j/java_gateway.py:1304, in JavaMember.__call__(self, *args)
   1298 command = proto.CALL_COMMAND_NAME +\
   1299     self.command_header +\
   1300     args_command +\
   1301     proto.END_COMMAND_PART
   1303 answer = self.gateway_client.send_command(command)
-> 1304 return_value = get_return_value(
   1305     answer, self.gateway_client, self.target_id, self.name)
   1307 for temp_arg in temp_args:
   1308     temp_arg._detach()

File /usr/lib/spark/python/pyspark/sql/utils.py:111, in capture_sql_exception.<locals>.deco(*a, **kw)
    109 def deco(*a, **kw):
    110     try:
--> 111         return f(*a, **kw)
    112     except py4j.protocol.Py4JJavaError as e:
    113         converted = convert_exception(e.java_exception)

File /opt/conda/miniconda3/lib/python3.8/site-packages/py4j/protocol.py:326, in get_return_value(answer, gateway_client,
target_id, name)
    324 value = OUTPUT_CONVERTER[type](answer[2:], gateway_client)
    325 if answer[1] == REFERENCE_TYPE:
--> 326     raise Py4JJavaError(
    327         "An error occurred while calling {0}{1}{2}.\n".
    328         format(target_id, ".", name), value)
    329 else:
    330     raise Py4JError(
    331         "An error occurred while calling {0}{1}{2}. Trace:\n{3}\n".
    332         format(target_id, ".", name, value))

Py4JJavaError: An error occurred while calling o80.partitions.
: org.apache.hadoop.mapred.InvalidInputException: Input path does not exist: /media/notebooks/Assignments/HW3/master/dat
a/googlebooks-eng-all-5gram-20090715-0-filtered.txt
        at org.apache.hadoop.mapred.LocatedFileStatusFetcher.getFileStatuses(LocatedFileStatusFetcher.java:156)
        at org.apache.hadoop.mapred.FileInputFormat.listStatus(FileInputFormat.java:247)
        at org.apache.hadoop.mapred.FileInputFormat.getSplits(FileInputFormat.java:325)
        at org.apache.spark.rdd.HadoopRDD.getPartitions(HadoopRDD.scala:205)
        at org.apache.spark.rdd.RDD.$anonfun$partitions$2(RDD.scala:300)
        at scala.Option.getOrElse(Option.scala:189)
        at org.apache.spark.rdd.RDD.partitions(RDD.scala:296)
        at org.apache.spark.rdd.MapPartitionsRDD.getPartitions(MapPartitionsRDD.scala:49)
        at org.apache.spark.rdd.RDD.$anonfun$partitions$2(RDD.scala:300)
        at scala.Option.getOrElse(Option.scala:189)
        at org.apache.spark.rdd.RDD.partitions(RDD.scala:296)
        at org.apache.spark.api.java.JavaRDDLike.partitions(JavaRDDLike.scala:61)
        at org.apache.spark.api.java.JavaRDDLike.partitions$(JavaRDDLike.scala:61)
        at org.apache.spark.api.java.AbstractJavaRDDLike.partitions(JavaRDDLike.scala:45)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:498)
        at py4j.reflection.MethodInvoker.invoke(MethodInvoker.java:244)
        at py4j.reflection.ReflectionEngine.invoke(ReflectionEngine.java:357)
        at py4j.Gateway.invoke(Gateway.java:282)
        at py4j.commands.AbstractCommand.invokeMethod(AbstractCommand.java:132)
        at py4j.commands.CallCommand.execute(CallCommand.java:79)
        at py4j.GatewayConnection.run(GatewayConnection.java:238)
        at java.lang.Thread.run(Thread.java:750)
```

```
DATA = sc.textFile("/media/notebooks/Assignments/HW3/master/data").map(lambda x: x.split('\t')[0]).cache()
DATA.take(5)
----
['A CATALOGUE OF THE PATHOLOGICAL',
 'A CT scan shows a',
 'A Case of Frustrated Take',
 'A Catalogue of Books and',
 'A Celebration of the First']
```

Version 1

In [45]:

```python
from itertools import combinations
from operator import add
from collections import defaultdict

def makePairs(row):
    words = row.split(' ')
    for w1, w2 in combinations(words, 2):
        yield((w1,"*"),1)
        yield((w1,w2),1)


def partitionByWord(x):
    return hash(x[0][0])

def partMapper(seq):
    currPair, currWord, pairTotal, wordTotal = None, None, 0, 0
    for r in list(seq):
        w1, w2 = r[0][0], r[0][1]
        if w2 == "*":
            if w1 != currWord:
                wordTotal = 0
                currWord = w1
            wordTotal += r[1]
        else:
            pairTotal += r[1]

            if currPair != r[0]:
                yield(w1+" - "+w2, pairTotal/wordTotal)
                pairTotal = 0
                currPair = r[0]
```

In [36]:

```
RDD_v1_sm = DATA_toy.flatMap(makePairs)\
          .repartitionAndSortWithinPartitions(numPartitions=2,
                                              ascending=True,
                                              partitionFunc=partitionByWord,
                                              keyfunc=lambda x: (x[0],x[1]))\
          .mapPartitions(partMapper)
```

In [43]:

```
%%time
RDD_v1_sm.glom().collect()
```

```
CPU times: user 10 ms, sys: 0 ns, total: 10 ms
Wall time: 110 ms
```

Out[43]:

```
[[('bear - pig', 0.5),
  ('bear - zebra', 0.5),
  ('dog - aardvark', 0.3333333333333333),
  ('dog - banana', 0.3333333333333333),
  ('dog - pig', 0.3333333333333333),
  ('pig - banana', 1.0)],
 [('aardvark - banana', 0.5), ('aardvark - pig', 0.5), ('zebra - pig', 1.0)]]
```

In [46]:

```
RDD_v1_med = DATA_F1.flatMap(makePairs)\
          .repartitionAndSortWithinPartitions(numPartitions=2,
                                              ascending=True,
                                              partitionFunc=partitionByWord,
                                              keyfunc=lambda x: (x[0],x[1]))\
          .mapPartitions(partMapper)
```

In [47]:
```
%%time
RDD_v1_med.take(5)
```

```
CPU times: user 10 ms, sys: 10 ms, total: 20 ms
Wall time: 11.7 s
```

Out[47]:
```
[('C - a', 0.02),
 ('C - activation', 0.04),
 ('C - and', 0.02),
 ('C - by', 0.1),
 ('C - circumference', 0.02)]
```

Version 2

In [38]:
```
from itertools import combinations
from operator import add
from collections import defaultdict


def makePairsWithinPartition(seq):
    pairsDict = defaultdict(int)
    for row in seq:
        words = row.split(' ')
        for w1, w2 in combinations(words, 2):
            pairsDict[(w1,"*")]+=1
            pairsDict[(w1,w2)] += 1
    for k,v in pairsDict.items():
        yield(k,v)

def partitionByWord(x):
    return hash(x[0][0])


def calcRelFreq(seq):

    seq = sorted(seq, key=lambda tup: (tup[0][0], tup[0][1]))

    currPair, currWord, pairTotal, wordTotal = None, None, 0, 0
    for r in list(seq):
        w1, w2 = r[0][0], r[0][1]
        if w2 == "*":
            if w1 != currWord:
                wordTotal = 0
                currWord = w1
            wordTotal += r[1]
        else:
            pairTotal += r[1]

            if currPair != r[0]:
                yield(w1+" - "+w2, pairTotal/wordTotal)
                pairTotal = 0
                currPair = r[0]
```

In [39]:

```
RDD_sm = DATA_toy.mapPartitions(makePairsWithinPartition)\
              .reduceByKey(add, numPartitions=2, partitionFunc=partitionByWord)\
              .mapPartitions(calcRelFreq, True)\
```

In [40]:

```
%%time
RDD_sm.glom().collect()
```

```
CPU times: user 20 ms, sys: 0 ns, total: 20 ms
Wall time: 169 ms
```

Out[40]:

```
[[('bear - pig', 0.5),
  ('bear - zebra', 0.5),
  ('dog - aardvark', 0.3333333333333333),
  ('dog - banana', 0.3333333333333333),
  ('dog - pig', 0.3333333333333333),
  ('pig - banana', 1.0)],
 [('aardvark - banana', 0.5), ('aardvark - pig', 0.5), ('zebra - pig', 1.0)]]
```

In [41]:

```
RDD_med = DATA_F1.mapPartitions(makePairsWithinPartition)\
              .reduceByKey(add,numPartitions=2, partitionFunc=partitionByWord)\
              .mapPartitions(calcRelFreq, True)\
```

In [42]:

```
%%time
RDD_med.take(5)
```

```
CPU times: user 10 ms, sys: 0 ns, total: 10 ms
Wall time: 5 s
```

```
Out[42]: [('C - a', 0.04),
          ('C - activation', 0.02),
          ('C - and', 0.1),
          ('C - by', 0.02),
          ('C - circumference', 0.02)]
```

```
In [17]: RDD_lg = DATA.flatMap(makePairs)\
                     .reduceByKey(add,partitionFunc=partitionByWord)\
                     .mapPartitions(calcRelFreq, True)\
                     .cache()
```

```
In [18]: %%time
         RDD_lg.take(5)
```

```
CPU times: user 100 ms, sys: 50 ms, total: 150 ms
Wall time: 10min 49s
```

```
Out[18]: [('o - A', 0.0006691201070592171),
          ('o - ARNO', 0.00033456005352960856),
          ('o - Adrianne', 0.00033456005352960856),
          ('o - Affairs', 0.00033456005352960856),
          ('o - America', 0.00033456005352960856)]
```

Version 3

Can we get the best of both worlds? Using the framework to combine as well as do secondary sort for us?

TODO

```
In [ ]: composite keys!
```

```
In [ ]:
```

```
In [ ]: # TODO: calculate averages example with aggragateByKey
        aTuple = (0,0) # As of Python3, you can't pass a literal sequence to a function.
        rdd1 = rdd1.aggregateByKey(aTuple, lambda a,b: (a[0] + b,    a[1] + 1),
                                           lambda a,b: (a[0] + b[0], a[1] + b[1]))
```

```
finalResult = rdd1.mapValues(lambda v: v[0]/v[1]).collect()

'''
First lambda expression for Within-Partition Reduction Step::
a: is a TUPLE that holds: (runningSum, runningCount).
b: is a SCALAR that holds the next Value

Second lambda expression for Cross-Partition Reduction Step::
a: is a TUPLE that holds: (runningSum, runningCount).
b: is a TUPLE that holds: (nextPartitionsSum, nextPartitionsCount).
'''
```

Serialization

In [ ]:

```
```

In [57]:

```
from pyspark import SparkContext
from pyspark.serializers import BatchedSerializer, PickleSerializer, CloudPickleSerializer

# Default serializer: PickleSerializer

sc.stop()
sc = SparkContext(
    "local", "bar",
    serializer=CloudPickleSerializer(),
    # Unlimited batch size -> BatchedSerializer instead of AutoBatchedSerializer
    #batchSize=-1
)

sc.serializer
```

```
22/09/13 18:36:22 INFO org.apache.spark.SparkEnv: Registering MapOutputTracker
22/09/13 18:36:22 INFO org.apache.spark.SparkEnv: Registering BlockManagerMaster
22/09/13 18:36:22 INFO org.apache.spark.SparkEnv: Registering BlockManagerMasterHeartbeat
22/09/13 18:36:22 INFO org.apache.spark.SparkEnv: Registering OutputCommitCoordinator
```

Out[57]:

```
AutoBatchedSerializer(CloudPickleSerializer())
```

In [58]:

```
import numpy as np
import cloudpickle, pickle

data = np.random.randint(0, 255, dtype='u1', size=100000000)
```

In [59]:
```
%timeit len(pickle.dumps(data, protocol=pickle.HIGHEST_PROTOCOL))
#80.1 ms ± 722 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
37.1 ms ± 392 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

In [60]:
```
%timeit len(cloudpickle.dumps(data, protocol=pickle.HIGHEST_PROTOCOL))
#46.5 ms ± 483 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
50 ms ± 293 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

In [ ]:

In [ ]: