Assignment 3

In case there are more technical problems, remember PythonSpark for information working with Python and Spark code, and RunningSpark for information on getting your code to go: either on your machine or the cluster.

Reddit Average Scores

As before, we want to use data from the Complete Public Reddit Comments Corpus *(https://archive.org/details/2015_reddit_comments_corpus)* , and calculate the average score for each subreddit. You can use the same `reddit-*` data sets as in the last assignment.

Name your program `reddit_averages.py` and as before, take command-line arguments for the input and output directories. I suggest you start with the SparkSkeleton provided (for RDDs, which we are working with here).

Some hints:

- The built-in Python JSON module *(https://docs.python.org/3/library/json.html)* has a function `json.loads` that parses a JSON string to a Python object.
- Your keys and values should be as before: subreddit as key, and count, score_sum pair as value.
- Write a function `add_pairs` so `add_pairs((1,2), (3,4)) == (4, 6)`. That will be useful.
- There is no separate combiner/reducer here: just reduce to pairs and divide in another step. (Spark does the combiner-like step automatically as part of any reduce operation.)

Since we're taking JSON input, let's produce JSON as output this time. You can map the pairs through `json.dumps` to get output lines like this:

```
["xkcd", 4.489060773480663]
```

Improving Word Count

Nicer Code

Take a minute to rearrange your `wordcount-improved.py` code into the provided SparkSkeleton. This should help keep things organized, and make it possible to import the code in a shell or notebook.

Go Bigger

There is a data set on the cluster that is larger than we have tried in the past: `wordcount-5` at about 600MB.

Run your program for this part with this command: it will limit your job to 8 executor processes, so we can all get more consistent timings:

```
time spark-submit --conf spark.dynamicAllocation.enabled=false --num-executors=8 wordcount-improved.
```

While the program is running, have a look at the Spark web frontend for your job. It is likely taking much longer than necessary. (It can be done in about 3:30 on this input with these restrictions.) Have a look at the “Stages” tab and keep and eye on how it progresses. Also have a look at the input files you're getting from HDFS.

Why does this program take so long on this input? [?]

Update your wordcount program so the data is shaped better for as much of the work as possible. Big hint *(https://spark.apache.org/docs/2.4.4/api/python/pyspark.html#pyspark.RDD.repartition)* .

Parallel Computation

Sometimes the work you want to do on a cluster doesn't depend on a large set of input data, but is just a big calculation. Let's try that…

We can estimate Euler's consant, e *(https://en.wikipedia.org/wiki/E_(mathematical_constant))* using a stochastic representation *(https://en.wikipedia.org/wiki/E_%28mathematical_constant%29#Stochastic_representations)* . Given a sequence $X_1, X_2, \ldots$ of uniform random variables on the range $[0, 1]$, we can take the smallest number of these such that the total is greater than one:

$$V = \min\{n \mid X_1 + X_2 + \cdots + X_n > 1\}.$$

The expected value $E(V) = e$.

What that means to us: we can use the law of large numbers *(https://en.wikipedia.org/wiki/Law_of_large_numbers)* , sample the values of V a bunch of times, average, and use that as an estimate of e. All we need to do is generate random numbers in $[0, 1]$ until the total is more than 1: the number of random values we needed for that is an estimate of e. If we average over many samples, we will get a value close to e.

Pseudocode

The logic program is basically this:

```
samples = [integer from command line]
total_iterations = 0
repeat batches times:
    iterations = 0
    for i in range(samples/batches):
        sum = 0.0
        while sum < 1
            sum += [random double 0 to 1]
            iterations++
    total_iterations += iterations

print(total_iterations/samples)
```

… except we will be doing the work of the outer-most `for` loop in parallel across the cluster, and calculating `total_iterations` as a reduce operation.

Implementation

Write a program `euler.py` that takes one command line argument, which is the number of samples we will make of the random variable. Out output is simple: we will simply `print` the result at the end of the main function. The command line will be something like:

```
time spark-submit euler.py 1000000
```

In order to get working with Spark, you will need to create an RDD yourself. You may want `sc.parallelize` *(https://spark.apache.org/docs/2.4.4/api/python/pyspark.html#pyspark.SparkContext.parallelize)* or `sc.range` *(https://spark.apache.org/docs/2.4.4/api/python/pyspark.html#pyspark.SparkContext.range)* , depending on the strategy you're taking to get the work done.

You can use the Python random module *(https://docs.python.org/3/library/random.html)* to generate the random numbers, and count the number before the sum gets to 1.0: that will be one "sample"

In the function where you use `random`, make sure you call `random.seed()`. If you don't every instance will have the same random seed (because the random number generator is duplicated with its initial seed in each thread); calling `random.seed()` reseeds the random number generator from the system's random source.

Note: there is a small (but non-zero) cost to calling a Python function. Having a function called once for each sample is more expensive than once for each partition.

Add up the number of iterations used by the number of samples given on the command line. Eventually, you will do something like this to display the result:

```
print(total_iterations/samples)
```

Testing and Parallelism

Please run the program for this part mostly on your computer, **not the cluster**. We will try to reserve the cluster resources for the above question (but feel free to try on the cluster at less-busy times). This will provide a good example of how Spark can make use of all of the processing power you have on a multi-core desktop computer as well as a cluster.

When you created the RDD here, you had a choice of how many partitions it was divided into (with the `numSlices` argument). Experiment with values for the number of partitions and choose a good one. [?]

Choose the number of iterations you test with as appropriate to the computer you're working on. (The lab workstations are 4 core/8 thread processors.) You should probably have your fastest times in the 20–30 second range so you're sure you're measuring real computation, not just overhead.

Language and Implementation Comparison

Since most of the computing work here is being done in Python, we expect that our code is slowed down by the speed of executing Python code: Python isn't known as a language for fast numeric computations.

But, PyPy *(http://pypy.org/)* implements Python with a just-in-time compiler *(https://en.wikipedia.org/wiki/Just-in-time_compilation)* that is quite good. We can ask Spark to use that implementation of Python instead.

Since the logic is simple enough, let's also compare some simple single-threaded implementations of this same algorithm: in Python and in C. These will only use one processor core, but will give us some idea how much overhead Spark is adding.

These commands will work on the lab workstations. You can also download PyPy *(http://pypy.org/download.html)* (probably Python 3 compatible, Linux x86-64 binary) for yourself, unpack, and adjust the paths accordingly.

```
PYPY=/usr/shared/CMPT/big-data/pypy3.6-v7.1.1-linux64/bin/pypy3
# standard CPython implementation:
export PYSPARK_PYTHON=python3
time ${SPARK_HOME}/bin/spark-submit euler.py 1000000000
# Spark Python with PyPy:
export PYSPARK_PYTHON=${PYPY}
time ${SPARK_HOME}/bin/spark-submit euler.py 1000000000
# Non-Spark single-threaded PyPy:
time ${PYPY} euler_single.py 1000000000
# Non-Spark single-threaded C:
gcc -Wall -O2 -o euler euler.c && time ./euler 1000000000
```

Have a look at the relative running times and compare the overhead cause by the standard Python implementation vs PyPy and Spark vs standalone code. [?]

Questions

In a text file `answers.txt`, answer these questions:

1. What was wrong with the original `wordcount-5` data set that made repartitioning worth it? Why did the program run faster after?
2. The same fix does **not** make this code run faster on the `wordcount-3` data set. (It may be slightly slower?) Why? [For once, the answer is not “the data set is too small”.]
3. How could you modify the `wordcount-5` input so that the word count code can process it and get the same results as fast as possible? (It's possible to get about another minute off the running time.)
4. When experimenting with the number of partitions while estimating Euler's constant, you likely didn't see much difference for a range of values, and chose the final value in your code somewhere in that range. What range of partitions numbers was “good” (on the desktop/laptop where you were testing)?
5. How much overhead does it seem like Spark adds to a job? How much speedup did PyPy get over the usual Python implementation?

Submission

Submit your files to the CourSys activity Assignment 3.

Updated Wed Sept. 25 2019, 14:20 by ggbaker.