Assignment 3B

Weather Data 1: Spark SQL Python API

For this question, we will look at historical weather data from the Global Historical Climatology Network *(https://www.ncdc.noaa.gov/data-access/land-based-station-data/land-based-datasets/global-historical-climatology-network-ghcn)* . Their archives contain (among other things) lists of weather observations from various weather stations formatted as CSV files like this:

```
US1FLSL0019,20130101,PRCP,0,,,N,
US1FLSL0019,20130101,SNOW,0,,,N,
US1TXTV0133,20130101,PRCP,30,,,N,
USC00178998,20130101,TMAX,-22,,,7,0700
USC00178998,20130101,TMIN,-117,,,7,0700
USC00178998,20130101,TOBS,-28,,,7,0700
USC00178998,20130101,PRCP,0,T,,7,0700
USC00178998,20130101,SNOW,0,T,,7,
USC00178998,20130101,SNWD,0,,,7,
USC00242347,20130101,TMAX,6,,,7,0800
```

The fields are the weather station; the date (Jan 1 2013 in these lines); the observation (min/max temperature, amount of precipitation, etc); the value (an integer); and four more columns we won't use. The readme file in the archive explains the fields in detail. We will worry about the min and max temperature (TMIN, TMAX) which are given as °C × 10.

Data sets can be found on the cluster HDFS at `/courses/732/weather-1` (and `-2` and `-3`), and the first two can be downloaded from http://cmpt732.csil.sfu.ca/datasets/ *(http://cmpt732.csil.sfu.ca/datasets/)* . (I have taken the liberty of running the original files through the Unix split *(http://linux.die.net/man/1/split)* command to make input chunks of a million lines each. That will give us nicely partitioned RDDs/DataFrames.)

Create a program `temp_range.py` and take input and output directory arguments, as usual.

Reading CSV

This data is simple enough that we could just `.split(',')`, but let's do it properly: `spark.read.csv()` will parse CSV files. The CSV reader infer column names from a header row, but we don't have them so **specifying a schema is required**.

The Problem

What I want to know is: **what weather station had the largest temperature difference** on each day? That is, where was the largest difference between TMAX and TMIN?

We want final data like this (if we `.show()`):

```
+--------+-----------+-----+
|    date|    station|range|
+--------+-----------+-----+
|20130101|USR0000NTEX|  384|
|20130102|USS0006H27S|  730|
|20130103|USS0006H27S|  748|
|20130104|RSM00024763| 1002|
|20130105|USR0000CCMO|  500|
|20130106|USR0000CCMO|  628|
|20130107|RSM00024679|  779|
|20130108|USR0000AHOW|  584|
|20130109|USR0000COPA|  539|
|20130110|USR0000HKAN|  606|
|20130111|ASN00046043|  299|
+--------+-----------+-----+
```

Yes, it seems unlikely that RSM00024763 had a daily temperature swing of 100.2 C°. Let's fix that once we actually have the data calculated…

See below for notes on the final output format.

Getting There

For this question, I'm going to ask you to implement this with the **Python API** (that is, with methods on DataFrames, not `spark.sql()`), but the next part is to do the same thing with the SQL syntax: start with whichever you like, obviously.

You will need to get acquainted with the DataFrame methods *(https://spark.apache.org/docs/2.2.0/api/python/pyspark.sql.html#pyspark.sql.DataFrame)* .

Hint: first get the TMIN and TMAX for each day separately, and then join them back together to get the temperature range. Find the max. Then join back to find out which station(s) that range came from. If there's a tie, this method will get us both results, which seems reasonable.

Fix It

The completely-unreasonable temperature ranges are annoying. It turns out we can solve it with the rarely-employed technique "reading the documentation". The yearly readme says that the second column we ignored was a QFLAG, and the readme explains the values

Basically: if there's **any non-null QFLAG value**, we will ignore the row. (e.g. the first row in the sample data above shouldn't be used.) With this change, I get output like this, which is much more promising:

```
+--------+-----------+-----+
|    date|    station|range|
+--------+-----------+-----+
|20130101|ASN00012044|  307|
|20130102|USR0000CCRE|  339|
|20130103|USS0006L05S|  341|
|20130104|USR0000CCRE|  333|
|20130105|ASN00013012|  377|
|20130106|USW00053017|  351|
|20130107|USC00054054|  334|
|20130108|CA008204193|  316|
|20130109|USC00292608|  317|
|20130110|USC00425837|  344|
|20130110|USC00041072|  344|
|20130111|ASN00046043|  299|
+--------+-----------+-----+
```

Output

Of course, you can use `df.show()` when experimenting, but it's not very good "final" output. Our actual output should be **space-delimited fields**:

```
20130101 ASN00012044 307
20130102 USR0000CCRE 339
20130103 USS0006L05S 341
20130104 USR0000CCRE 333
20130105 ASN00013012 377
20130106 USW00053017 351
20130107 USC00054054 334
20130108 CA008204193 316
20130109 USC00292608 317
20130110 USC00425837 344
20130110 USC00041072 344
20130111 ASN00046043 299
```

To do this, you can access the DataFrame.rdd property *(https://spark.apache.org/docs/2.2.0/api/python/pyspark.sql.html#pyspark.sql.DataFrame.rdd)* , map the `Row` objects to strings in the right format, and `.saveAsTextFile`. Or you can use the DataFrame `.write.csv` method with a `sep=' '` argument. Your choice.

Weather Data 2: Spark SQL, SQL syntax

Create a `temp_range_sql.py` that does the same as the above, but **using the SQL syntax**. That is, most of the work should be done with calls to `sqlContext.sql()`, not by calling methods on DataFrame objects.

The logic should translate more-or-less directly. You just have to spell it with Spark's SQL dialect *(https://docs.databricks.com/spark/latest/spark-sql/index.html)* . [❓]

Most-Viewed Wikipedia Pages

As in assignment 1, let's look at the Wikipedia page view statistics *(https://dumps.wikimedia.org/other/pagecounts-raw/)* . Recall that the files are named like `pagecounts-20160801-120000.gz` and the format is like:

```
en Aaaah 20 231818
en Aaadonta 2 24149
en AaagHiAag 1 8979
```

Use Spark DataFrames to **finds the the most-viewed page each hour** and how many times it was viewed. The program should be called `wikipedia_popular.py` and (as usual) take input and output directories from the command line.

As before, we're interested only in: (1) English Wikipedia pages (i.e. language is `"en"`) only. (2) Not the page titled `'Main_Page'`. (3) Not the titles starting with `'Special:'`). [The smaller data sets with subsets of pages might not have the main page or special pages: that doesn't mean you aren't responsible for this behaviour.]

You can assume the filenames will be in the format `pagecounts-YYYYMMDD-HHMMSS*` (maybe with an extension). We want the `YYYYMMDD-HH` substring of that as a label for the day/hour.

Some hints:

- The DataFrame.read.csv function can read these space-delimited files with a `sep` argument. It can also have meaningful column names if you give a `schema` argument.
- The filename isn't obviously-visible, but can be retrieved with the `input_file_name` function if you use it right away: `spark.read.csv(…).withColumn('filename', functions.input_file_name())`
- Unless you're more clever than me, converting an arbitrarily-deep filename (that might be `file:///` or `hdfs:///` and `.gz` or `.lz4` or neither, or who-knows what else) is very hard in the DataFrame functions. I suggest writing a Python function that takes pathnames and

returns string like we need: `'20160801-12'`. Then turn it into a UDF: `path_to_hour = functions.udf(…, returnType=types.StringType())`.

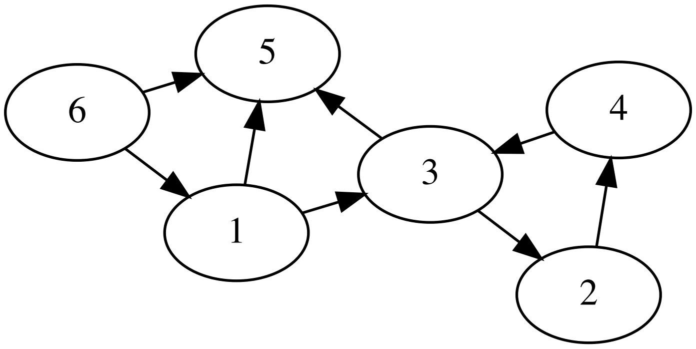
- To find the most-frequently-accessed page, you'll first need to find the largest number of page views in each hour.
- … then join that back to the collection of all page counts, so you keep only those with the `count == max(count)` for that hour. (That means if there's a tie you'll keep both, which is reasonable.)

Sort your results by date/hour (and page name if there's a tie) and output as a CSV. Part of my output on `pagecounts-1` looks like this. Note the tie in hour 6. [The `pagecounts-1` data set has been replaced since assignment 1, for something a little more interesting.]

```
20160801-03,Simon_Pegg,175
20160801-04,Simon_Pegg,135
20160801-05,Simon_Pegg,109
20160801-06,Simon_Cowell,96
20160801-06,Simon_Pegg,96
20160801-07,Simon_Pegg,101
20160801-08,Simon_Pegg,119
```

Shortest Paths in Graphs

Let's have a look at the problem of finding the shortest path in a graph. To start with, this graph:



We will represent the graph by listing the outgoing edges of each node (and all of our nodes will be labelled with integers):

```
1: 3 5
2: 4
3: 2 5
4: 3
5:
6: 1 5
```

If we would are asked to find the shortest path from 1 to 4, we should return 1, 3, 2, 4.

You can do this problem with Spark SQL, **or** core (non-SQL) Spark. (I will use a slightly more verbose description of the data below, to not bias towards one of the other.)

The Algorithm

We will use a modification of Dijkstra's algorithm *(https://en.wikipedia.org/wiki/Dijkstra's_algorithm)* that can be parallelized.

We will maintain an RDD/DataFrame containing the list of nodes we know how to reach with the shortest path, along with the node we reach them from (to reconstruct the path later) and their distance from the origin.

At the **end** of the algorithm, we hope to have found this:

```
node 1: no source node, distance 0
node 3: source 1, distance 1
node 5: source 1, distance 1
node 2: source 3, distance 2
node 4: source 2, distance 3
```

That is, node 4 can be reached from node 2, and is 3 steps from the origin (1). We can start the algorithm with one entry we know: the origin is 0 steps away from itself: (Represent the “no source” however you like: we won't be using the value.)

```
node 1: no source node, distance 0
```

After that, we can iterate: nodes in that list at distance n imply a path to their neighbours at distance $n + 1$. In this case, we have nodes of distance 0 (just node 1) and can now find all of their neighbours (3 and 5), so after one iteration, have:

```
node 1: no source node, distance 0
node 3: source 1, distance 1
node 5: source 1, distance 1
```

Now we can look at the nodes at distance 1 and their outgoing edges: those can be reached at distance 2. After another iteration:

```
node 1: no source node, distance 0
node 3: source 1, distance 1
node 5: source 1, distance 1
node 2: source 3, distance 2
node 5: source 3, distance 2
```

… and we see a problem. We already knew how to reach node 5 with a path of length 1: the newly-found path of length 2 doesn't help. We need to **keep only the shortest-length path** to each node:

```
node 1: no source node, distance 0
node 3: source 1, distance 1
node 5: source 1, distance 1
node 2: source 3, distance 2
```

With one more iteration:

```
node 1: no source node, distance 0
node 3: source 1, distance 1
node 5: source 1, distance 1
node 2: source 3, distance 2
node 4: source 2, distance 3
```

We can now use this to trace back the path used to get to 4: the last line says that we got to 4 from 2 at distance 3. We got to 2 from 3; and 3 from 1. That give the correct path 1, 3, 2, 4.

Implementation

Your program should be `shortest_path.py` and the command line should take four arguments: the input directory, an output path where we will produce a few things, and the source and destination nodes (both integers). That is, the command line will end up like:

```
spark-submit shortest_path.py /courses/732/graph-1 output 1 4
```

The input path will contain a file `links-simple-sorted.txt`, and you should consider **only** that file in the directory. The input file is in the format above. Each line of input is a node, colon, and a space-separated list of nodes it's adjacent to.

For each iteration of the algorithm, output the RDD or DataFrame of the found paths (in some sensible format) in a directory `iter-<N>`. When you are done, produce an output directory `path` containing the actual path you found, with one node per line:

```
1
3
2
4
```

It would be very easy to create an infinite loop while implementing this algorithm. Please implement like this, so there is an upper-bound on the number of iterations of the algorithm (assuming RDD data here; adapt appropriately for DataFrames):

```
for i in range(6):
    ...
    paths.saveAsTextFile(output + '/iter-' + str(i))
    if we_seem_to_be_done:
        break

finalpath = ...
finalpath.saveAsTextFile(output + '/path')
```

[This implies we can't find paths of length >6, but we will live with that restriction.]

Sample Data

On the cluster, you will find `/courses/732/graph-1` which is the 6-node graph from the examples above. The `-2` data set is a randomly generated graph with 100 nodes and 200 edges (with a path from 53 to 76 and 92 to 45, but none of length ≤6 from 1 to 51). The `-3` data set is randomly generated with 1M nodes and 8M edges (with paths from 1407 to 820198 and 3671 to 210435; none from 254907 to 180014).

These are also available at http://cmpt732.csil.sfu.ca/datasets/ *(http://cmpt732.csil.sfu.ca/datasets/)* .

On the cluster, there is also a `-4` data set which is not random *(http://haselgrove.id.au/wikipedia.htm)* . The data format for this question was borrowed from that data set.

Hints

Pay attention to what you cache here: not caching data you need to iterate on will turn a perfectly reasonable algorithm into something completely unworkable.

Overall efficiency in your loop is going to matter as the input graph grows: don't create candidate paths blindly and then reduce/filter them out if you can avoid it.

There are certainly many ways to proceed with the problem, but I have done these…

Hints: Spark + RDDs

You should create a key-value RDD of the **graph edges**: the node as your key, and list of nodes connected by outgoing edges as the value. Cache this, because you'll be using it a lot.

You'll use it by joining it to your working list of **known paths** in a format something like `(node, (source, distance))`. Then for each node, produce a new RDD with all known paths (probably with `flatMap` and a function that will…). Produce all currently-known paths, as well as any new ones found by going one more step in the graph.

That will produce the longer-than-necessary paths: reduce by key to keep only the shortest paths.

Hints: Spark SQL + DataFrames

Create a DataFrame of the **graph edges**: source and destination nodes for each edge. Cache this, because you'll be using it a lot.

Create a DataFrame to represent **known paths** (node, source, distance), starting with only the origin node (at distance 0 from itself). With each step, you will need to create a DataFrame in the same format with one-step-longer paths and use `.unionAll` to add it to the collection of paths you have.

That will produce the longer-than-necessary paths: subtract/filter/groupby to keep only the shortest.

Hints: path reconstruction

Once you have the destination node and its "source", you can start working backwards. You can do a `.lookup` on an RDD or a `.where`/`.filter` on a DataFrame to find the source and *its* source. Iterate until you get back to the original origin.

Since we know we will be finding paths with length at most 6, you can build the path in a Python list (and `.parallelize` to write).

Questions

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

1. For the weather data question, did you prefer writing the "DataFrames + Python methods" style, or the "temp tables + SQL syntax" style form solving the problem? Why?
2. Which do you think produces more readable code?

Submission

Submit your files to the CourSys activity Assignment 3B.

Updated Thu Oct. 19 2017, 16:06 by ggbaker.