Assignment 6

Standalone Reddit Averages

Use the SparkSkeleton and the example "Spark SQL + DataFrames" code from last week to **create a program** `reddit_average_df.py` that calculates the average score for each subreddit (as we have before), but using Spark DataFrames.

We will make a few additions to the code from last week...

When reading JSON into a DataFrame, specifying a schema makes reading much faster: without a schema, the files are read to infer the schema **and then again** to load the data. Specify the schema when reading the file. We really want to avoid that overhead. Big hint: RedditSchema.

We will write our output as CSV files. The output format won't be *exactly* like last week, but it will be similar. The code will be like: `averages.write.csv(output, mode='overwrite')`

Execution Plans

DataFrames have execution plans that describe how they will be computed. Have a look at the plan for this problem (assuming the DataFrame you write is `averages`):

```
averages.explain()
```

Some things to note from the output: when the JSON input is read, you can see which fields are loaded into memory; the average is calculated in three steps. [?]

Race!

We now have several implementations for the Reddit averages problem that we could compare: MapReduce, Spark RDDs, and Spark DataFrames.

For each of the Spark implementations, we could compare the usual CPython with PyPy and see if PyPy speeds up the code.

Do these really differ, or did we just waste our time reimplementing the same logic? Feel free to do these runs in small groups: there's not much point to all of us grinding away at the cluster. Write up your answers to the questions separately.

The `reddit-6` data set is a yet-larger subset of the Reddit data, and large enough to take at least some measurable time on the cluster. Again, we will limit ourselves to 8 executors to get consistent and fair comparisons.

These commands should get everything to run as described:

```
export HADOOP_CLASSPATH=./json-20180813.jar
# MapReduce
time yarn jar a1.jar RedditAverage -libjars json-20180813.jar,/opt/hadoop/share/hadoop/tools/lib/lz4-java-1.7.1
# Spark DataFrames (with CPython)
time spark-submit --conf spark.dynamicAllocation.enabled=false --num-executors=8 reddit_average_df.py /courses/
# Spark RDDs (with CPython)
time spark-submit --conf spark.dynamicAllocation.enabled=false --num-executors=8 reddit_averages.py /courses/73
# Spark DataFrames (with PyPy)
module load spark-pypy
time spark-submit --conf spark.dynamicAllocation.enabled=false --num-executors=8 reddit_average_df.py /courses/
# Spark RDDs (with PyPy)
time spark-submit --conf spark.dynamicAllocation.enabled=false --num-executors=8 reddit_averages.py /courses/73
```

For MapReduce, there's nothing on the command line that can limit the job to a certain number of cores, but the input is partitioned into 8 files, so we'll get 8 mappers and one reducer. That should be reasonably comparable to 8 executors and one driver for Spark.

If you want to see how important specifying a schema was above, you could do another run with the DataFrames implementation without the `schema=...` when reading the file. [For me, it was about 60 seconds extra.]

Have a look at the questions below: can you explain what you just observed? [?]

Most-Viewed Wikipedia Pages

As we did with MapReduce and RDDs, let's find the most popular page by hour from the Wikipedia page view statistics.

This time, we will work with the original data format: the hour is **not** in each line of the file, but must be inferred from the filename. 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
```

These files are in the `pagecounts-*` data sets, **not** the `pagecounts-with-time-*` sets.

Use Spark DataFrames to **find the the most-viewed page each hour** and how many times it was viewed. The program should be called `wikipedia_popular_df.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.
- When reading the file, give a `schema` argument so data is read correctly with meaningful column names.
- 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, and using it as a UDF. (Note below.)
- 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.)
- As always, use `.cache()` appropriately.

Sort your results by date/hour (and page name if there's a tie) and output as newline-delimited JSON. Part of my output on `pagecounts-1` looks like this. Note the tie in hour 6.

```
{"hour":"20160801-03","title":"Simon_Pegg","views":175}
{"hour":"20160801-04","title":"Simon_Pegg","views":135}
{"hour":"20160801-05","title":"Simon_Pegg","views":109}
{"hour":"20160801-06","title":"Simon_Cowell","views":96}
{"hour":"20160801-06","title":"Simon_Pegg","views":96}
{"hour":"20160801-07","title":"Simon_Pegg","views":101}
```

Your UDF should split the path string on `'/'`, take the last path component, and extract the string we want, like `'20160801-12'`. You can assume the filename (last path component) will start with `'pagecounts-'`.

```
@functions.udf(returnType=types.StringType())
def path_to_hour(path):
    ...
```

Broadcast Joins & DataFrames

The join operation that's necessary in this question combines every page and its count (big: there are many Wikipedia pages) with the max views for every hour (small: Wikipedia has only existed for a few thousand days × 24 hours). That seems like a candidate for the broadcast join we saw in the last exercise.

With DataFrames, we can get that optimization easily: either by the runtime noticing that one of the join inputs is small and automatically broadcasting, or by explicitly flagging a DataFrame as broadcast-able.

Try your code on a reasonably-large input set with and without the broadcast hint. Use an input set that's large enough to see what would happen on big data. That might be `pagecounts-3` on the cluster. [?]

There is an automatic detection of broadcast-able DataFrames, which you may have to disable on the command line to see the difference: `--conf spark.sql.autoBroadcastJoinThreshold=-1`

Also, compare the execution plans for your final result (from `.explain()`) for the join (1) with the broadcast hint and (2) without the broadcast hint and with `--conf spark.sql.autoBroadcastJoinThreshold=-1`. [?]

Weather and Temperature Ranges 1: DataFrame Methods

Let's look again at the Global Historical Climatology Network data. 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
```

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 several flags about the observation. Their readme file explains the fields in detail.

We will worry about the min and max temperature (TMIN, TMAX) which are given as °C × 10. As we did last week we only want to keep the values where the QFLAG field is null, indicating a "correct" observation.

Data sets are `weather-1` (and `-2` and `-3`), which are subsets of the full data, partitioned nicely. Create a program `temp_range.py` and take input and output directory arguments, as usual.

The CSV files should be read by using `spark.read.csv()` and specifying a schema, as before.

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(10)`):

```
+--------+-----------+-----+
|    date|    station|range|
+--------+-----------+-----+
|20170501|USC00410779| 26.1|
|20170502|USC00413411| 26.1|
|20170503|RSM00031939| 24.6|
|20170504|USS0013C39S| 22.5|
|20170505|USW00094012| 27.8|
|20170506|USC00243110| 31.1|
|20170507|RSM00030521| 27.1|
|20170508|RSM00030971| 25.2|
|20170509|USC00047109| 23.9|
|20170509|USW00094107| 23.9|
+--------+-----------+-----+
```

Output

The output should be sorted by date and by station if there is a tie. Our final output in this case should be the same as the input: CSV files in the default way that Spark produces.

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.

First get the TMIN and TMAX for each day separately, and then join them back together and subtract to get the temperature range. At some point, divide by 10 so we have actual °C values. Find the max. Then join the max values 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.

There are a few points where you should be caching DataFrames here: make sure you do.

It's possible that doing a broadcast join here would be faster than the standard join: you can include the broadcast hint if you want, but don't worry too much about it.

Weather and Temperature Ranges 2: 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 `spark.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. [?]

The output should be exactly the same as the DataFrames-methods implementation.

Questions

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

1. In the Reddit averages execution plan, which fields were loaded? How was the average computed (and was a combiner-like step done)?
2. What was the running time for your Reddit averages implementations in the five scenarios described above? How much difference did Python implementation make (PyPy vs the default CPython)? Why was it large for RDDs but not for DataFrames?
3. How much of a difference did the broadcast hint make to the Wikipedia popular code's running time (and on what data set)?
4. How did the Wikipedia popular execution plan differ with and without the broadcast hint?
5. 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? Which do you think produces more readable code?

Submission

Submit your files to the CourSys activity Assignment 6.

Updated Fri Aug. 25 2023, 15:55 by ggbaker.