Assignment 4

Course Rules

Spark is flexible enough to let you get yourself into trouble. In particular, you can easily destroy any parallelism and lose the ability to do actual big data.

If you use `.collect()` to turn an RDD into a Python list, you **must** have a comment nearby that gives a bound on the size of the data, to indicate that you have thought about the implications of turning it into a non-distributed collection.

Same rules for `.coalesce(1)` or similar operations that decrease the number of partitions.

```
# keys here are small integers; <=100 elements collected
data = rdd1.reduceByKey(f).collect()
# RDD has ~10k elements after grouping.
rdd3 = rdd2.groupByKey().coalesce(1)
```

Reddit ETL

It is common to have data arrive in a format that is not exactly what you want to work with. The common solution: do a quick extract, transform, load operation to transform the data into something you can more easily work with.

In this question, we will do a simple transformation on Reddit data, tidying up the data and keeping only the subset we need with a command like this:

```
time spark-submit reddit_etl.py reddit-2 output
```

In this artificial scenario, the transforms of the data we need will be:

- Keep only the fields subreddit, score, and author.
- Keep only data from subreddits that contain an `'e'` in their name (`'e' in comment['subreddit']`).
- Write the rows with score greater than zero in JSON format is a subdirectory `positive`: `.map(json.dumps).saveAsTextFile(output + '/positive')`
- Write the rows with score less-than-or-equal to zero in JSON format is a subdirectory `negative`: `.map(json.dumps).saveAsTextFile(output + '/negative')`

[This is very similar to the data that we would need for the next question, but we will work with the original data there.]

Caching

Since Spark evaluates RDDs (and as we'll see later, DataFrames) lazily and discards intermediate results, your code is likely reading the input and doing the first two steps above twice: once to write `positive` and then again to write `negative`.

The .cache() method on an RDD can overcome this: it's an instruction that when the RDD is calculated, it should be stored in memory in the executor process, so it's there when needed again.

Add a `.cache()` to the RDD *that you use twice* in your code and compare the running times with and without. The `reddit-2` data set should be large enough to see a small difference if you're running locally, and `reddit-3` or `-4` on the cluster. [❓]

Reddit Relative Scores

For this question, we want to know who made **the best** comment on Reddit. But some communities are friendlier than others: we will define "the best" as the comment with the highest score relative to the subreddit's average. That is, we are looking for the largest values of (comment score)/(subreddit average score).

Make a copy of your subreddit average code `relative_score.py`. As before, have input and output directories as arguments on the command line.

The first thing we need to do is to calculate the same RDD of pairs (subreddit, average score). Let's ignore any subreddits with averages 0 or negative.

Now we need the comment data again: make sure you `.cache()` the RDD after applying `json.loads` since that step is expensive and it's where we'll be starting the next calculation.

Form a pair RDD with the subreddit as keys and comment data as values (adjusting variable names as you like):

```
commentbysub = commentdata.map(lambda c: (c['subreddit'], c))
```

Use the .join() method to get the average scores together with the comments themselves.

We want to report the *author* of the comment, so create an RDD of `comment['score']/average` and `comment['author']`. Your output should be pairs of (relative_score, author).

Sort by the relative score (descending) and output to files. (Don't coalesce here: you have no reason to believe that the output will be small.)

Get this working on the same data sets as before: `/courses/732/reddit-1` and `/courses/732/reddit-2` or from https://ggbaker.ca/732-datasets/.

Go bigger!

Congratulations! You have been given your own cluster and real big data work to do. Your cluster has four nodes with quad core processors, and 2GB of memory per node. (It's not much of a cluster, but a bunch of them have to live inside our cluster so we'll make it work.)

We can simulate such a situation by invoking Spark on our Cluster like this:

```
spark-submit --conf spark.dynamicAllocation.enabled=false --num-executors=4 --executor-cores=4 --executor-memo
```

When you start the relative scores code like this, everything should go fine.

It's time for some big(ger) data. I have created another data set on the cluster's HDFS at `/courses/732/reddit-4/`. In small groups, run the above algorithm on it, with our cluster and the above `spark-submit` command. (If you all do this individually at the same time, you'll overwhelm our cluster.)

While you're waiting, have a look at the YARN frontend (port forwarded at http://localhost:8088/). Click on your job's ApplicationMaster link. [You'll have to edit the URL and change "controller.local" to "localhost".] You should be able to see the beautiful Spark UI. Have a look around: you've got some time to kill.

Depending on your implementation, your job may fail (running out of memory) or it may succeed after much longer than necessary.

We can do much better.

Smarter Join

What we're trying to do here is actually a little dumb: the data set we're failing on only contains data for a few subreddits: the RDD containing the averages-by-subreddit is tiny. Then we're joining it against a large RDD, making many copies of that information.

Make a copy of your `relative_score.py` called `relative_score_bcast.py` and make the changes below...

Since it's relatively small, could just turn it into a Python object (probably a dict with `dict(averages.collect())`) and send that to each executor. We can take the average scores RDD, `.collect()` it, and use sc.broadcast() to get it sent out (it returns a Broadcast object).

Once you broadcast a value, it is sent to each executor. You can access it by passing around the broadcast variable and any function can then get the `.value` property out of it.

To calculate the relative score, we map over the comments: write a function that takes the broadcast variable and comment as arguments, and returns the relative average as before. Make sure you're **actually using the broadcast object**: you should have to access it as `averages.value` inside the function that runs on each executor.

This technique is a *broadcast join*. Now you should actually be able to run it on the `reddit-4` data set on the cluster. [❓]

Let's be clear: on your little 4 × 2 GB cluster, you could have spent a bunch of money on more memory and made your original program run. Or you could have been just a little bit clever.

Spark SQL + DataFrames

Like other functionality in Spark, the pyspark shell can do Spark SQL/DataFrames stuff. There is already a `spark` variable (a SparkSession instance) defined when you start.

In pyspark, you can use `spark` to start creating DataFrame objects and using their methods. **Try this**:

```
>>> from pyspark.sql import functions, types
>>> inputs = '/courses/732/reddit-1/' # or other path on your computer
>>> comments = spark.read.json(inputs)
>>> averages = comments.groupby('subreddit').agg(functions.avg(comments['score']))
>>> averages.show()
```

Have a look at the output: this should be the same results as your previous Reddit averages implementations. This code could have been faster (if we specified a schema for the data) but we won't worry about that for now.

Weather ETL

Let's do another ETL task, this time with Spark DataFrames. We will work with data from the Global Historical Climatology Network, which contains about 180MB of data for each year: enough that big data tools are the best way to approach it.

The data is compressed CSV files. Conveniently, DataFrames can be read directly from CSV files. It's best to give an explicit schema for the file, so the types and column names are as you expect:

```
observation_schema = types.StructType([
    types.StructField('station', types.StringType()),
    types.StructField('date', types.StringType()),
    types.StructField('observation', types.StringType()),
    types.StructField('value', types.IntegerType()),
    types.StructField('mflag', types.StringType()),
    types.StructField('qflag', types.StringType()),
    types.StructField('sflag', types.StringType()),
    types.StructField('obstime', types.StringType()),
])

weather = spark.read.csv(inputs, schema=observation_schema)
```

Start with the SparkSkeleton for DataFrames, call it `weather_etl.py`, and add that code to read the original data, which is in the usual places as `weather-1`, `weather-2`, and `weather-3`. These are partitioned subsets of the original GHCN data, but the exact same file format.

Things that need to be done to get the data into a more usable shape (for some future work we imagine we might do):

1. Read the input files into a DataFrame as described above.
2. Keep only the records we care about:
   a. correct data: the field `qflag` (quality flag) is null; (Hint)
   b. Canadian data: the `station` starts with `'CA'`; (Hint option 1; Hint option 2)
   c. maximum temperature observations: the `observation` is `'TMAX'`.
3. Divide the temperature by 10 so it's actually in °C, and call the resulting column `tmax`.
4. Keep only the columns `station`, `date`, and `tmax`.
5. Write the result as a directory of JSON files GZIP compressed (in the Spark one-JSON-object-per-line way).

Code that will produce the desired output format:

```
etl_data.write.json(output, compression='gzip', mode='overwrite')
```

Here are two lines of correct output (after uncompressing):

```
{"station":"CA008206240","date":"20170520","tmax":17.6}
{"station":"CA001072692","date":"20170507","tmax":6.5}
```

Questions

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

1. How much of a difference did the `.cache()` make in your Reddit ETL code?
2. When would `.cache()` make code slower than without?
3. Under what conditions will the broadcast join be faster than an actual join?
4. When will the broadcast join be slower?

Submission

Python code should be beautiful and readable. There will be some "style" component to the marking from now on.

Submit your files to the CourSys activity Assignment 4.

Updated Mon Oct. 02 2023, 08:45 by ggbaker.