Assignment 3A

Spark SQL Shell

Like other functionality in Spark, the pyspark shell can do Spark SQL stuff. There is already a `spark` variable (a SparkSession *(https://spark.apache.org/docs/2.2.0/api/python/pyspark.sql.html#pyspark.sql.SparkSession)* instance) defined when you start.

In pyspark, you can use `spark` to start creating DataFrame *(https://spark.apache.org/docs/2.2.0/api/python/pyspark.sql.html#pyspark.sql.DataFrame)* objects and using their methods. **Try this**:

```
>>> inputs = '/courses/732/reddit-1/' # or wherever
>>> comments = spark.read.json(inputs)
>>> averages = comments.select('subreddit', 'score').groupby('subreddit').avg()
>>> averages.show()
```

These are very SQL-like operations. You can also **do the same** thing with SQL syntax:

```
>>> comments.createOrReplaceTempView('comments')
>>> averages = spark.sql("""
    SELECT subreddit, AVG(score)
    FROM comments
    GROUP BY subreddit
""")
>>> averages.show()
```

The triple-quoted string syntax in Python (`"""..."""`) can be used to give a multi-line string literal, which is a convenient way to embed not-too-ugly SQL queries. The argument to `spark.sql` here is still just a string.

Aside: Spark SQL Programs

When using Spark SQL (or Spark DataFrames, if you prefer to call them that), our program template will change a little, to create a `SparkSession`, which is the entry point to this functionality. You won't *usually* need the `SparkContext` with Spark SQL programs, but can extract it if you do (commented-out line below).

```
import sys
from pyspark.sql import SparkSession, functions, types

spark = SparkSession.builder.appName('example application').getOrCreate()
# sc = spark.sparkContext
assert sys.version_info >= (3, 4) # make sure we have Python 3.4+
assert spark.version >= '2.2' # make sure we have Spark 2.2+

def main():
    # do things...

if __name__ == "__main__":
    main()
```

This uses the Python convention *(https://www.artima.com/weblogs/viewpost.jsp?thread=4829)* for a main function. With it, you can start a pyspark shell to test and experiment with your code, something like this:

```
>>> from reddit_average_sql import *
>>> main()
>>> any_other_function(8) # or whatever you want to test
```

That's probably a good starting point for a program using Spark DataFrames as you create them…

Standalone Reddit Averages

Use the above examples to **create a program** `reddit_average_sql.py` that behaves the same as the problem from Assignment 2A but uses Spark SQL to get its results.

To output to JSON file(s), you can use a DataFrame's `.write` property, which is a DataFrameWriter *(https://spark.apache.org/docs/2.2.0/api/python/pyspark.sql.html#pyspark.sql.DataFrameWriter)* :

```
df.write.save(path, format='json', mode='overwrite')
```

The output format won't be exactly like last week, but that's okay.

Specifying a Schema

The `spark.read` value that we used is a DataFrameReader *(https://spark.apache.org/docs/2.2.0/api/python/pyspark.sql.html#pyspark.sql.DataFrameReader)* . There is an optional argument when reading to specify a schema: what columns the dataframe will have, and their types.

If you don't specify a schema for JSON input, the **data must be read twice**: once to determine the schema, and again to actually load the data. That can be a big cost. **Add an explicit schema** to your `reddit_average_sql.py`.

The way a schema is constructed isn't obvious, but here's a hint:

```
schema = types.StructType([
    types.StructField('subreddit', types.StringType(), False),
])
```

If you don't specify fields in the schema, they will be ignored when the file is read, so you only have to worry about fields you need.

Race!

We have now written three Spark implementations to calculate the Reddit averages (core Spark in A2; Spark SQL without a schema; and Spark SQL with a schema specified). Do they really differ, or did we just do the same thing three times?

The `reddit-5` data set is a yet-larger subset of the Reddit data, and large enough to take at least some measurable time. **Try the three implementations**, with and without PyPy. Please limit yourself to four executors and two cores each for the runs. You can either use the Unix `time` command to measure the clock-time the command took, or look at the "elapsed" time in the YARN web frontend. [❓]

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

Again, feel free to do this in small groups: there's not much point to all of us grinding away at the cluster.

[My Spark RDD implementation triggers a bug in pypy *(https://bitbucket.org/pypy/pypy/issues/2659/typeerror-in-inspectgetattr_static)* : if yours does the same, feel free to skip that test.]

Actually, we have written four implementations: we did it in Java+MapReduce. I tried my implementation on the `reddit-5` data. It's hard to throttle MapReduce back to a certain number of processors, but with my best approximation of this situation, it took 1:28. [❓]

Server Logs (again)

Let's return to the NASA web server log data (`/courses/732/nasa-logs-1` and `/courses/732/nasa-logs-2`, or http://cmpt732.csil.sfu.ca/datasets/ *(http://cmpt732.csil.sfu.ca/datasets/)*).

Create a program `ingest_logs.py` with arguments for input and output files. We are essentially ingesting the data in a simple way: we will read and parse the lines, then store the data in a better format.

The format we will use is Parquet *(https://parquet.apache.org/)* , a columnar storage format designed for the kind of data analysis we would do with Spark SQL. (The video on that page is a good introduction to the internals if you're interested.)

As you did last week, parse the input lines. We will extract the requesting host, the datetime, the path, and the number of bytes. Each of these are captured by the regular expression from last week. We want to actually turn the timestamp into a Python datetime object *(https://docs.python.org/2/library/datetime.html#datetime-objects)* .

```
datetime.datetime.strptime(datestring, '%d/%b/%Y:%H:%M:%S')
```

The basic work of taking input lines and parsing it needs to be done with non-SQL Spark: Spark SQL can't really take over until we have tabular data. That will mean creating an RDD of Row objects *(https://spark.apache.org/docs/2.2.0/api/python/pyspark.sql.html#pyspark.sql.Row)* and then passing them to `spark.createDataFrame`.

Once you have a DataFrame, you can output to Parquet files easily:

```
df.write.format('parquet').save(outputdir)
```

Reading Parquet

Now that you have the Parquet file, read it with Spark SQL using `spark.read`. In the PySpark shell, calculate the total number of bytes transferred by all requests. [?]

[Yes, this is a trivial use: we want you to see that it's easy to reconstruct the DataFrame from Parquet files, but are not looking to add difficulty. If we tested, it would be somewhat faster than the same operation on JSON input.]

Takeaway message: if your project is going to use Spark SQL, consider an ETL phase that gets the data into Parquet, which should make everything easier after that.

Questions

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

1. What was the running time for your three reddit-averages implementations? How can you explain the differences?
2. How much difference did Python implementation make (PyPy vs the default CPython)? Why would it make so much/little difference sometimes?
3. How does your Spark implementation running time compare to my MapReduce time? What do you think accounts for the difference?
4. What statements did you execute to get the total number of bytes transferred (including reading the Parquet data)? What was the number?

Submission

Submit your files to the CourSys activity Assignment 3A.

Updated Fri Oct. 13 2017, 10:06 by ggbaker.