Homework 7

For this homework you will create a `.py` file. This file should then be uploaded to wolfware in the assignment link!

The purpose of this homework is to practice analyzing streaming data and the use of Spark Structured Streaming. Most homework assignments will have a part that pushes you beyond what was in the lectures! Learning to search for the right questions and browsing stackoverflow are really life skills that we should hone :) Although you are turning in a `.py` file, you should still include comments that represent a brief narrative for what you are doing with the code.

Goal

Read in streams of different types using pyspark. Perform basic ETL operations (summarizations, parsing, etc) and write the output to the console or a file.

- You'll be creating one larger `.py` file, you should have sections corresponding to each task (separate by block comments (lots of comment characters).

- You should have any necessary python code to produce the data/populate it, as well as the pyspark code used to read and process the streams.

Task 1

Reading a Stream

- For this task, we'll just generate data using the `rate` source (see example in the notes).
  - Create an input stream from the `rate` format using 1 row per second
  - Recall that this outputs values with a timestamp. The values start at 0 and increase by 1 each time.

Transform/Aggregation Step

- Now, we'll do some basic practice with windowing summarizations! To do a window operation and outputting, we'll need to add a watermark and use `.groupBy()` to specify the windows.
  - Add a watermark that relies in the `timestamp` column that uses a five second watermark
  - Import the `window()` function and use it with `.groupBy()` to create windows that are 30 seconds long with no overlap
  - We'll simply sum the values within that window, so add the `.sum()` aggregation

Writing the Stream

- Lastly, we need to decide how to write the stream and where the results should go. We'll write the stream to `memory`. This will allow us to run the query (name the object `myquery`) for a while and then use SQL type commands to view the results.
  - Use the `memory` output `format`
  - Use the `update` `outputMode`
  - Add a `trigger` to slow down how often we do our computations. Make the trigger based on a `processingTime` of 20 seconds
  - You'll need to name the query. This is another method you add on to this part via `.queryName("name_goes_here")`

  - Lastly, start the query via `.start()`

Let `myquery` run for 2 minutes or so. Then type `myquery.stop()` and hit return.

Now What?

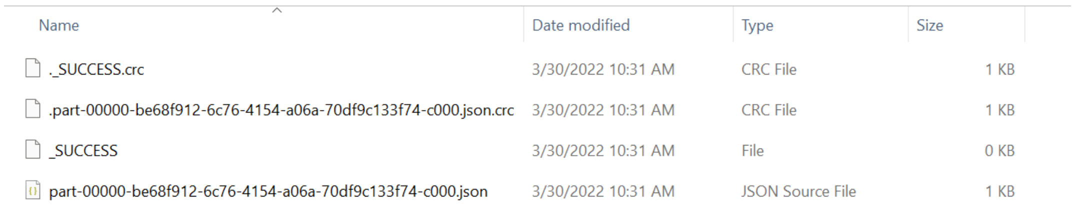
- The output is available to us in an in memory table called `name_goes_here`. View it in the pyspark console via

```
spark.sql("SELECT * FROM name_goes_here").show()
```

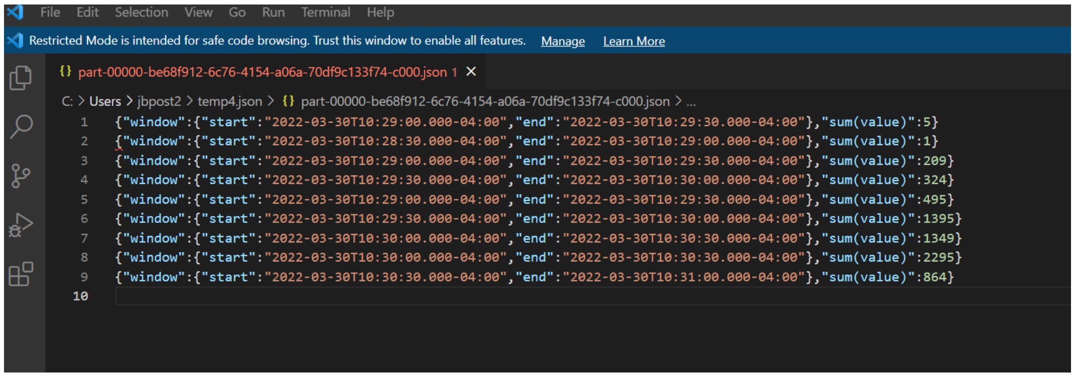
- Cool! But there are lots more rows than we would think should be there (we'd think one row for each 30 second interval, but it prints out intermediate results as well).
- Let's send all of this output to a file to investigate it further.
- Run a command similar to the one below to produce a `.json` file with the table's info in it:

```
spark.sql("SELECT * FROM name_goes_here") \
.coalesce(1) \
.write.format("json") \
.option("header", "false") \
.save("path_to_save/folder_name")
```

- We are selecting the table, using `.coalesce(1)` to take all of the pieces (recall things are stored in a distributed manner) and combine them into one file. Then we are writing the results to a `.json` file that will be created in the folder_name.
- This is what my `.json` file looks like in the folder (bottom most file) and when viewing it in Visual Studio Code:



Name	Date modified	Type	Size
._SUCCESS.crc	3/30/2022 10:31 AM	CRC File	1 KB
.part-00000-be68f912-6c76-4154-a06a-70df9c133f74-c000.json.crc	3/30/2022 10:31 AM	CRC File	1 KB
_SUCCESS	3/30/2022 10:31 AM	File	0 KB
part-00000-be68f912-6c76-4154-a06a-70df9c133f74-c000.json	3/30/2022 10:31 AM	JSON Source File	1 KB


File Edit Selection View Go Run Terminal Help
Restricted Mode is intended for safe code browsing. Trust this window to enable all features. Manage Learn More
part-00000-be68f912-6c76-4154-a06a-70df9c133f74-c000.json 1
C: > Users > jbpost2 > temp4.json > part-00000-be68f912-6c76-4154-a06a-70df9c133f74-c000.json > ...

```
{"window":{"start":"2022-03-30T10:29:00.000-04:00","end":"2022-03-30T10:29:30.000-04:00"},"sum(value)":5}
{"window":{"start":"2022-03-30T10:28:30.000-04:00","end":"2022-03-30T10:29:00.000-04:00"},"sum(value)":1}
{"window":{"start":"2022-03-30T10:29:00.000-04:00","end":"2022-03-30T10:29:30.000-04:00"},"sum(value)":209}
{"window":{"start":"2022-03-30T10:29:30.000-04:00","end":"2022-03-30T10:30:00.000-04:00"},"sum(value)":324}
{"window":{"start":"2022-03-30T10:29:00.000-04:00","end":"2022-03-30T10:29:30.000-04:00"},"sum(value)":495}
{"window":{"start":"2022-03-30T10:29:30.000-04:00","end":"2022-03-30T10:30:00.000-04:00"},"sum(value)":1395}
{"window":{"start":"2022-03-30T10:30:00.000-04:00","end":"2022-03-30T10:30:30.000-04:00"},"sum(value)":1349}
{"window":{"start":"2022-03-30T10:30:00.000-04:00","end":"2022-03-30T10:30:30.000-04:00"},"sum(value)":2295}
{"window":{"start":"2022-03-30T10:30:30.000-04:00","end":"2022-03-30T10:31:00.000-04:00"},"sum(value)":864}
```

- The last row that has a particular time window in it represent the sum of the values in that time interval. Recall that the sum of the first n numbers is n*(n+1)/2.
  - There are two observations in the 28:30-29:00 window (0 and 1), for a sum of 1
  - There are 30 observations in the 29:00-29:30 window, for a sum of 31*(31+1)/2 - 1 = 495 (subtract the 1 that in the previous window)
  - There are 30 observations in the 29:30-3:00 window, for a sum of 61*(61+1)/2 - 496 = 1395 (subtract off the previous sum)

Task 2

- Repeat task 1 but allow for overlapping windows. Have the windows overlap by 15 seconds. For example, windows from 28:00-28:30, 28:15-28:45, 28:30-29:00, ...

Task 3

- For this, we'll write a python script to send `.csv` files to a folder. You'll have pyspark monitoring that folder, reading in the data, transforming it, and writing the result out to a file.
- The data we'll use comes from the UCI machine learning repository. The study was about trying to detect heavy drinking during a bar crawl based on alcohol readings and accelerometer data from cell phones. We'll simply consider the accelerometer data for this example.
  - This data has five columns: time (in unicode seconds), pid (person id), x, y, and z information from the accelerometer.
  - We'll want to take in these values, convert the x, y, and z components to a magnitude (I think that makes sense to do here...)

Set up for Creating Files

- Read in the `all_accelerometer_data_pids_13.csv` file
- There are lots of records per user id. Let's just concern ourselves with person SA0297 and PC6771. Create two data frames, one for person SA0297's data and one for person PC6771's data.
- Set up a for loop to write 500 values at a time (not randomly, from first line) for SA0297 to a .csv file in a folder for that person's data. Similarly output values for PC6771 to another folder. The loop should then delay for 10 seconds after writing to the files.

Reading a Stream

- We're going to read in two streams (via two separate queries) for this part.
- Setup the schema (read all in as strings) and create an input stream from the `csv` folder for SA0297
- Setup the schema (read all in as strings) and create another input stream from the `csv` folder for PC6771

Transform/Aggregation Step

- Now, for each stream we'll do a basic transformation of the x, y, and z coordinates into a magnitude. This can be done via
$$mag = \sqrt{x^2 + y^2 + z^2}$$
- All of the necessary functions are avaible from the `spark.sql.functions` submodule. You'll need to cast the x, y, and z columns to type double and then apply the square and square root. I did this using the `.select()` method with the transformations done within `.select(transform here)`
- We want to keep the time and pid columns, this new column, and drop the original x, y, and z columns. This was easy to incorporate within the `.select()` above!

Writing the Streams

- Lastly, we'll write each stream out to their own csv file(s).
  - Use the `csv` output `format`
  - Use the `append` `outputMode`
  - You'll need to include an option for `checkpointlocation`.
    - The syntax is `.option("checkpointlocation","path")`
  - Lastly, start each query via the `.start()` method

Leave these queries running.

Now open a python console and submit the necessary code to run the loop that outputs data.

Let this all run for about 5 minutes. Stop both queries using the `.stop()` method.

Now What?

- The output is in `.csv` files for each user but not very easy to deal with because it is split up
- Using code from here, we can read in all the pieces and output each to their own single `.csv` file.

```
# Use PySpark to read in all "part" files
allfiles =  spark.read.option("header","false").csv("/destination_path/part-*.csv")
# Output as CSV file
allfiles \
.coalesce(1) \
.write.format("csv") \
.option("header", "false") \
.save("/destination_path/single_csv_file/")
```

You don't need to turn in these `.csv` files, just have the code for this part in there as well!

File to Submit

Upload the `.py` file with all the code (separated into clear sections) to the assignment link.