

IS 6493
Homework Exercise 6: Spark Structured Streaming
(Total: 40 points)

For this homework, you will perform stream processing using data from an IoT device. We will use Apache Spark's Structured Streaming API to write our code with Databricks Notebooks.

Prior to competing this homework, please read the following reference material:

- The slide set on Structured Streaming in Spark
 - This contains up-to-date material from the Spark Structured Streaming reference (to get more details, browse through the link below and go back to it as needed – there is a lot of additional advanced material there that we won't need):
<https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html>
 - The Python Structured Streaming documentation (classes and methods) for coding in Spark is here: <https://spark.apache.org/docs/3.0.0-preview/api/python/pyspark.sql.html#module-pyspark.sql.streaming>
- Optionally, skim through chapters 20, 21 and 22 from Spark: The Definitive Guide by Bill Chambers and Matei Zaharia.
 - We will use the same dataset used in these chapters with additional analyses

DATA

We will use The Heterogeneity Human Activity Recognition (HHAR) dataset from Smartphones and Smartwatches. This is a dataset devised to benchmark human activity recognition algorithms (classification, automatic data segmentation, sensor fusion, feature extraction, etc.) in real-world contexts; specifically, the dataset is gathered with a variety of different device models and use-scenarios, in order to reflect sensing heterogeneities to be expected in real deployments.

Take a look here to know more about the data we will use:

<https://archive.ics.uci.edu/ml/datasets/Heterogeneity+Activity+Recognition#>

This data measures the orientation of a sensor along x,y,z dimensions as 9 subjects perform various activities (gt). The device and model are recorded, as well as various timestamps and an index.

The file consists of the following columns:

- 'Arrival_Time' - The time the measurement arrived to the sensing application
- 'Creation_Time' - The timestamp the OS attaches to the sample
- 'Device' - The specific device this sample is from. prefixed with the model name and then the number, e.g., nexus4_1 or nexus4_2 (8 available)
- 'Index' - observation id/ row number
- 'Model' - smartphone/watch model (4 available)

- 'User' - user id
- 'gt' – activities carried out by participant (ground truth): bike sit, stand, walk, stairsup, stairsdown and null
- 'x' - sensor orientation along dimension x
- 'y' - sensor orientation along dimension y
- 'z' - sensor orientation along dimension z

This is a large dataset. We will only use a subset of the data that has already been created.

- Mainly our data contains only one Model: LG's nexus4 and two devices: nexus4_1 and nexus4_2
- This subset of data is already loaded on to DBFS in JSON format at
dbfs:/databricks-datasets/definitive-guide/data/activity-data/

Next you will perform a number of tasks. Before we use Spark Structured Streaming to read and process the data, we will first read it in using a static DataFrame. After obtaining some statistics on the data, we will then work with streaming operations.

SUBMISSION INSTRUCTIONS (PLEASE READ CAREFULLY)

- Create a Databricks notebook and name the file **Ex6_FirstLastName**
- Make the first cell a **Markdown cell** with L2 title TASK 0, followed by a L3 sub-title 'Exercise 6 Completed by Your name'
- Use **Markdown** (start cell with %md) to clearly label each task – TASK 1, TASK 2, in L2 headings.
- Use a **Python comment** to indicate each subtask using # 1a #1b ...
 - *Use a different cell for each subtask.*
 - Write all the code for each *subtask in one cell.*
 - This is really important to do, as it makes grading easier, so please comply.
- Some tasks have multiple outputs. **Every sub-task output should be visible**
 - if you produce more than one output per cell, make sure the **outputs asked for are visible** in the published notebook (use **print()**).
 - **If we can't see the output we can't grade it.**
IF THERE IS NO OUTPUT IN THE TABLES YOU WILL GET ZERO FOR THE TASK. IT IS YOUR RESPONSIBILITY TO RUN EACH QUERY AND WAIT UNTIL IT IS READY TO START PRINTING THE OUTPUTS USING THE FOR LOOPS. YES, THIS IS IN CAPS FOR A REASON!
- Please do not copy the instructions into the notebook – keep the file simple and clean (only include markdown for task numbering, code and outputs).
- For this exercise, DO NOT use **Run All** on your notebook – unfortunately doing so, will show empty outputs in tables that depends on streaming queries. For the streaming tasks (6-13), you need to carefully run the cells one by one and follow the instructions provided in each step.
- Submit two things:
 - publish your notebook and post the url in a **submission comment**.
 - export your notebook as **Ex6_FirstlastName.ipynb** and upload it.

SOME NOTES ON TASKS

- Only perform tasks marked as alphabets a, b, c....
- i, ii are sub-steps –should be combined to produce the output requested in each a, b, c ...
- Please show results in tabular form, unless requested otherwise. Refrain from using `DF.collect()` as this will load the entire distributed contents into the driver's memory – a very memory expensive task.

TASKS

Task 0. Housekeeping

- In the first cell, write some markdown as a L2 heading: Task 0
And a L3 sub-heading: **Exercise 6 completed by Your Name**

To enter Markdown, start the cell with **%md** leave a space or hit enter and write markdown.

- We will need the following import statements

```
import pyspark.sql.functions as F  
from pyspark.sql.types import *
```

Task 1. Explore the data folder

- View the contents of the folder that contain the data. The path for the folder is:
`"dbfs:/databricks-datasets/definitive-guide/data/activity-data/"`.
 - Call the `dbutils.fs.ls()` method with the above path to view the data files in that folder. Note that the output of this method is a list of many files, some of which are the data files we want (ending in .json).
 - Save this output in a variable, say *a*. Then use the `display()` function to view the output in a more readable format. Observe the column names.

- How many total files are in this directory? Use a command to obtain this.

HINT: What command will give you the size or number of elements in the output?
It's a standard Python function.

- How many .json files are in this directory? Use a command to obtain this.

HINT: there are many ways to solve this. I will explain how to do this with DataFrames.

- First create a new DataFrame from the output of 1a, saved in *a*. let's call this new dataframe *b*. Then use `filter()` method of DataFrame *b* to choose only the rows where the *name* column ends in .json. You will use a boolean expression with a method in the Column class (that works on strings) to only select rows that match this condition.

- ii. Finally, call the **count()** method of DataFrame.
- You can save intermediate outputs to new DataFrames (if you like), or you can specify all of the above operations as a chain of commands as a single line of code.
- d. What is the total size of the .json data files?
 - i. Do all the same steps as above (select only the rows that contain .json files), except calling the count() method. Instead select only the column 'size' that contains the bytes of each file.
 - ii. Then call **groupby().sum()** to get the sum of the values in that column.

Task 2. Read the data in as a static DataFrame

- a. First read the json files using **spark.read.format('json').load('path')** or **spark.read.json('path')** and save the static DataFrame as *staticDF*
 - i. Where path is the one shown above in Task 1.
 - ii. Unlike when reading .csv format files, there is no *inferSchema* parameter, so do not use it – schema is inferred for json automatically
- b. Obtain the schema of static DF using the **schema** attribute of DataFrame and save it as *dataSchema*. We will need to use this later
- c. Print the DataFrame's schema using the **printSchema()** method to see a pretty format.

Why read in the data as a static DataFrame?

We will see later that for streaming data, we have to specify the schema on read. This may not be easy to determine for streaming data. A trick is to obtain the schema by reading a part (or all) of the available streaming data first as a static DataFrame.

Task 3. View the data and obtain some descriptive stats

- a. View only the first 15 lines of staticDF DataFrame using **display()** function
- b. Count the number of observations/rows of the DataFrame using the **count()** method

Note: There are two count methods: **DataFrame.count()** and **GroupedData.count()**.

The DataFrame method is an action command and will output an int.

On the other hand, the GroupedData method is a transformation (and so produces no immediate output) and lazily produces a DataFrame. To see the output of transformations in Spark, we need call an action command after the transformation (such as **DataFrame.groupby().count().show()**).

- c. Determine if there are any duplicate observations using the *distinct()* and *count()* methods together.
 - i. Write down your answer as a comment.
- d. View some summary descriptive stats (mean, stddev, min, max) of all the columns in the staticDF by calling the *describe()* method of DataFrame.
 - i. This method will return null for the mean and stddev of non-numeric fields. Examine the distribution to know more about the data

Task 4. GroupedData operations

- a. How many different Model-Device combination values are there?
 - i. Call *groupby()* on 'Model' and 'Device' columns of the DataFrame and then call the *count()* method of GroupedData.
- b. Produce the number of counts for each combination of values in columns 'Device' and 'gt'. Print the result in descending sorted order of the column *count* created by calling *groupby()* followed by *count()*.
 - i. *gt* is short for ground truth and contains an activity performed by a subject that is measured by the sensors.
- c. Which combination of 'Device' and 'gt' columns has the highest count? Write a command to obtain just that row from the DataFrame produced in 4b.
 - i. Once the count column is sorted –think about whether the highest count value is at the beginning or at the end of the DataFrame rows. There is a DataFrame method that can give you just that one row.

NOTE: we typically do not save the outputs of each calculations in new named DataFrames. That is not the Spark way. The Spark way is to recompute answers when needed. So even though we have most of the code needed for part 4c in 4b, we will rewrite it again for this answer.

Task 5. Data partitions, repartitioning and shuffle partitions

- a. Determine the number of data partitions being used for the DataFrame.
 - i. To do this we need to first convert the DataFrame to an RDD by using the *rdd* attribute of DataFrame, and then call the *getNumPartitions()* method of rdd.
 - ii. Observe that the number of default partitions is greater than the number of worker nodes (8), so some workers may have more than one executor running on its cores.

- b. Change the number of partitions used by calling the ***repartition()*** method of DataFrames and give it a new value, say 100. Then call the ***getNumPartition()*** method again to verify the change
 - i. Since DataFrames are immutable, you have to save the result of ***repartition()*** back to staticDF to actually change the partitions that staticDF is distributed over
- c. Next, we will check and set the number of shuffle partitions.

Each partition is a unit of processing in Spark.

spark.sql.shuffle.partitions configures the number of partitions that are used when data is shuffled for joins or aggregations, e.g., groupby(). The default is 200. When a shuffle or join is called this number will be used to repartition the data.

In most of the cases, this number is too high for smaller data and too small for bigger data. If the data volume is not enough to fill all the partitions when there are 200 of them, this causes unnecessary map reduce operations, and ultimately causes the creation of very small files in DBFS, which is not desirable. If however you have too few partitions, and lots of data to process, each of your executor's memory might not be enough/available to process so much at one given time, causing errors like **OutOfMemoryError**.

- i. View the number of shuffle partitions by calling
spark.conf.get("spark.sql.shuffle.partitions")
- ii. Change the shuffle partitions by calling
spark.conf.set("spark.sql.shuffle.partitions", 100) and
- iii. print the value again to ensure that it is changed.

In recent versions of Spark, all of this tuning is done automatically using Adaptive Query Execution – so we will rarely need to (re-)partition directly. But now you know what to do if you want to customize it.

STREAMING TASKS

From here on, we will treat the data as a streaming source.

Task 6. Create a streaming DataFrame to read and output streaming data

To read in as a streaming input, we use the ***pyspark.sql.streaming.DataStreamReader*** class, which we obtain by calling the readStream property of Spark (***spark.readStream***). It has many methods to read from different types of data sources such as ***csv()***, ***json()***, ***orc()***, ***parquet()*** etc. It also has a method ***option()*** to specify extra options and ***schema()*** to specify the schema to apply to the incoming data.

See more here: <https://spark.apache.org/docs/3.0.0-preview/api/python/pyspark.sql.html#pyspark.sql.streaming.DataStreamReader>

- a. Read .json files
 - i. We can use the following chain of commands to read many .json files.

```
streamingDF = spark.readStream \  
    .option("maxFilesPerTrigger", 1) \  
    .json("dbfs:/databricks-datasets/definitive-guide/data/activity-  
    data/*.json")
```

Here we can mimic streaming data, by specifying that we read in only one file at a time each trigger. Trigger is used to specify the time intervals at which to (read in new input and) write out Streaming Queries. By default it is None, which means read the next after processing the previous event. So essentially one file at a time will be read and processed, and then the next file, and so on.

- ii. The above command will however produce an error, because when streaming from files, we have to specify the schema (and cannot infer it). How do we obtain the schema of this data? Recall that we saved it in Task 2.b. The correct code for reading streaming data is:

```
streamingDF = spark.readStream \  
    .schema(dataSchema) \  
    .option("maxFilesPerTrigger", 1) \  
    .json("dbfs:/databricks-datasets/definitive-guide/data/activity-data/*.json")
```

OR

```
streamingDF = spark.readStream \  
    .schema(dataSchema) \  
    .option("maxFilesPerTrigger", 1) \  
    .format('json') \  
    .load("dbfs:/databricks-datasets/definitive-guide/data/activity-data/*.json")
```

If you don't want to split your code across lines, you can write it as a single line as well. Remove the \ and extra white-space characters.

- b. Set up a streaming query to start processing the contents of the stream
 - i. We just read in some streaming data, but it is *not* possible to view the entire streaming DataFrame using an action command directly as we do with static DataFrames. `StreamingDF.show()` will not work.
 - ii. In order to view the streaming data as it is being read/processed, we have to set up a way to write the outputs to a sink: such as external *file* (used to persist the results), *memory* (used for testing purposes) or *console* (printed on the driver). This can be achieved by calling call the ***writeStream*** property of a streaming DataFrame.

streamingDF.writeStream

This creates a new instance of the *pyspark.sql.streaming.DataStreamWriter* class (this class allows us to write out a streaming DataFrame to an output sink)

- iii. We can set a few properties for the DataStreamWriter object to specify how we want to write the output. We can specify the output destination or sink using a method ***DataStreamWriter.format()***, and we can specify the output mode using a method ***DataStreamWriter.outputMode()***.

- 1. ***format()***

We will use output format (or sink) as 'memory'. When we write to 'memory' it will create a SQL table of outputs held in memory. This is used for testing and debugging purposes (so that we can view the output easily) and should not be used in production code. Other options for output format are 'console' (but will print at the driver not our local machines), 'kafka' (a stream processing framework) or a file (e.g., 'json', 'parquet'). When we write to 'memory', we also need to specify a name for this table using method ***DataStreamWriter.queryName()***

queryName() – this accepts one string parameter – the name of the intermediate sql table held in memory.

- 2. ***outputMode()***


This tells Spark how to create the intermediate results tables. Output mode = 'complete' will create a complete table each time. 'append' will add newly changed rows from the streaming DataFrame, and 'update' will only include rows that have changed.

Note: not all modes make sense for all streamingDataFrames – it will depend on the nature of transformation that are applied to obtain the streaming DataFrame. For example, when you use aggregations, use 'complete'. When viewing the DataFrame without any aggregation operations, you may use 'append', but cannot use 'complete'.

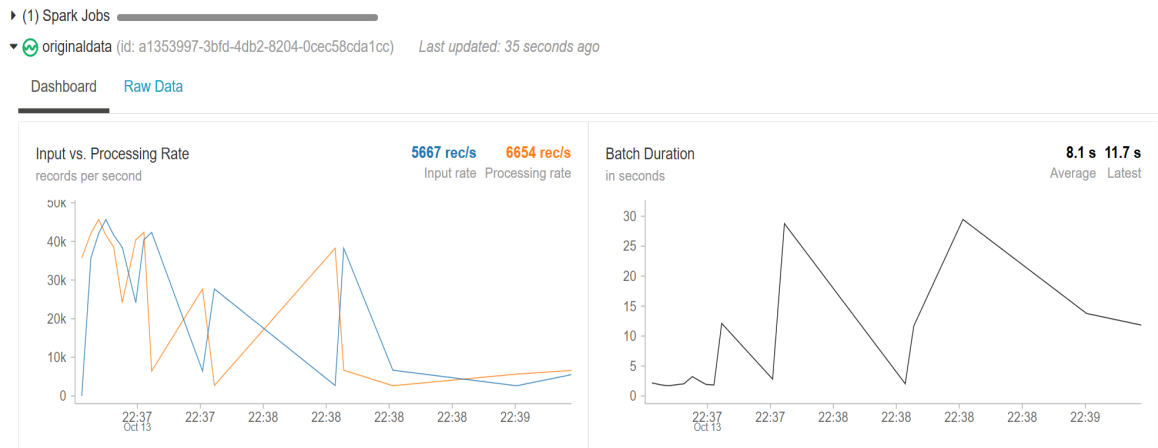
- iv. The last step is to start the streaming operation (reading in data as it arrives and write to a sink. To do this, we call the ***start()*** method of the ***DataStreamWriter*** object. This returns a ***pyspark.sql.streaming.StreamingQuery*** object.
- v. Putting it all together, we get the following chain of commands:
***streamingQuery = streamingDF ***

```
.writeStream \  
.queryName('OriginalData') \  
.format("memory") \  
.outputMode("append") \  
.start()
```


This produces streaming output. You will see the following in the output after the stream has initilaized.

►  OriginalData (id: 6985f3ed-9cbb-4783-8853-d2e2b907b495) Last updated: 5 seconds ago

If you click on the green symbol, you can then view the following dashboard.



IMPORTANT: If you try to write out streamingDF (the entire raw input) in *append* mode, you can run out of memory. You may then obtain an error – **QUOTA EXCEEDED**. Since we are using memory as our sink - it creates too many temporary files (each partition of each input file is a separate output file). Community Edition limits us to 10000 files and 15GB storage in DBFS.

Write the following clean up code at the beginning of your notebook in a cell, so that you can clean up each time before you run the notebook again, esp if you run your notebook many times and use output modes (like append) or write to external files stored in DBFS.

- call `dbutils.fs.ls` to view the path "`dbfs:/local_disk0/tmp`" folder to see all these files written out to memory. You can only view this after you have run some streaming queries and created a tmp file.
- Then call `dbutils.fs.rm(path, True)` to recursively remove all those files, if present.
- If this folder doesn't exist, you may get an error (just comment out these lines)

c. View the output of the streaming query 5 times

- In steps 6b, we are not doing any processing/ computation on the streaming data, we just read it and write it out. Since we are writing out to *memory* this will create an in-memory SQL Table called '*OriginalData*' as given in the parameter *queryName* above.
- If we want to view the output as streaming data accumulates, we can do so by converting this table into a DataFrame by calling `spark.sql("SELECT * FROM queryName")` *queryName* should be the name of the query that you used above.
- Then call `DataFrame.show()` on the static DataFrame that the above returns.

- iv. The above commands – each call to *spark.sql("SELECT...")* will show us the output as a DataFrame at one point in time. To view the output multiple times as the streaming progresses, we can call the above command inside a for loop.
- v. Additionally, we can add a *sleep(n)* function inside the loop to make the query wait n seconds before producing the next output.
You will need to import the sleep function from the time module.
from time import sleep
- vi. Wait until you see data is streaming in the dashboard above to call this (otherwise you will get empty rows), before calling the following command inside a loop.

Fill in the blanks below to print the streaming output **5 times** (you have learned how to count 5 times using a for loop earlier in this course) - showing **only 5 rows** each time, using **5 seconds of sleep** (you may need to adjust the sleep time if you like in order to see changing outputs).

The query name is the name given as *.queryName()* and now what the output has been saved as.

Write the code in a new cell

```
for _____:  
    spark.sql("SELECT * FROM _____").show(_)  
    sleep(_____)
```

Your output should show 5 different DataFrame tables – 5 snapshots of the streaming inputs. NOTE: the data seen will be the same – the top rows, but we cannot call *.tail()* on streaming data as of yet (since it is an unbounded table)

d. Stop the streaming query

- i. There are two ways in which queries can end
 1. by calling *query.stop()* or by an exception.
 2. To end a query, we have to explicitly ask it to *stop()*, otherwise the handle keeps waiting for more data, once all existing data has been read.
- ii. When the input Dashboard shows no more data (a flat line for input), OR when you have read enough to produce some output, stop the streaming query by calling the *stop()* method on the *StreamingQuery* object.

StreamingQuery.stop()

[Replace *StreamingQuery* with the name of your streaming query]

- iii. It may take a few seconds but watch the cell that is running the streaming query and you will see that it has stopped (the green streaming query icon is no longer green, and becomes grayed out).

NOTE: The above steps 6b, 6c and 6d – will be used every time we want to perform any transformations on streaming data and view some outputs as they are calculated.

In the next few tasks, we will first set up a transformation, and then write code similar to 6b and 6c to view streaming outputs, and code similar to 6d to stop the query after the desired outputs have been printed.

Task 7. Setting up a transformation on Streaming DataFrame: count the number of input rows

Many times, we will not just want to view the entire streaming input data, but we will want to perform transformations (calculations) on this data. Here, we will perform a simple count calculation that we set up as a transformation.

- a. Obtain a new DataFrame by specifying a transformation on *streamingDF* to count the rows.
 - i. The transformation we want to apply is ***count()***.
 - ii. We cannot apply ***count()*** directly to the whole streaming DataFrame. Why?
 - i. In order to use the *DataStreamWriter* to create a *streamingQuery*, we need to start with a streaming DataFrame; however calling ***DataFrame.count()*** would produce the answer as an int (which is not a streaming DF).
 - ii. There is a different form of *count()* that serves as a transformation and will give us a DataFrame (instead of a single int):
DataFrame.groupby().count()
That is what we want. When nothing is specified inside the parenthesis of ***groupby()***, the following method is applied to the whole data instead of sub-groups.
 - iii. Specify a transformation on the DataFrame that contains the raw data that we read in, as follows, and save it the resulting DF.
resultDF = streamingDF.groupby().count()

Task 8. Processing a streaming transformation: counts of rows

- a. Start streaming query by writing out to the counts of streaming data to *memory* as output sink.
 - i. Repeat steps similar to Task 6b.

- ii. Write outputs to *memory* sink – this will create an in-memory table (creates new temp files) that we can then view using SQL query as the streaming progresses. Alternatively, you can write the outputs out to an external file for persistence.
- iii. This time, use outputMode '*complete*', so that the whole table is rewritten after each micro-batch is processed. Aggregate values should change as more data is processed. '*complete*' is a good output mode to use when we use **groupby()** to perform aggregations on DataFrames.
- iv. Name the query as '*CountData*'.
- v. Save the resulting streaming query in a ***StreamingQuery*** object called *mystreamingQuery*, which serves as a handle to the query.

- vi. Below, I give you the skeleton command to start a streaming query with the properties described above. Please fill in the blanks

```
mystreamingQuery = _____.writeStream |  
                        .queryName(_____) |  
                        .format(_____) |  
                        .outputMode(_____) |  
                        .start()
```

While the query is still running (the query icon below is still green)
you can click on the streaming link with the same name as the query given above
(*'CountData'*) and view the streaming progress.

▶  CountData (id: 2f32a3f1-e281-427e-b351-06586e5410a3) Last updated: 20 seconds ago

- b. Awaiting termination for a streaming query (**DO NOT execute**)
 - i. This task is for information only (We will not be using this command for this exercise, so write the command in the same cell where the streaming query is created and started, but **comment it out**.)
Usually when we are not writing out to memory, but writing to a file for instance, we may not want to proceed to the next tasks until a streaming task has completed.
 - ii. You can achieve this by calling the **awaitTermination()** method of a ***StreamingQuery***.
mystreamingQuery.awaitTermination()
This will force the program to wait until all the streaming operations have completed before proceeding. Essentially the streaming process will block other commands from running.

Task 9. Viewing information about the streaming query and stopping it

- a. Print out some information about the streaming query while it is running (squiggly icon next to query name is green (active)).
 - i. Several properties or attributes of the ***StreamingQuery*** class can be used to obtain information.
StreamingQuery.status
StreamingQuery.id
StreamingQuery.name
StreamingQuery.isActive
NOTE: you have to run this command soon after you have started the streaming query in order to see the counts changing as records are being read in. Ensure that your output shows `.isActive = True`
Also you have to replace `StreamingQuery` above with the name of your ***StreamingQuery*** object created above in Task 8.
 - ii. Use **`print()`** function for each line to print all of the above outputs in the same cell.
- b. View the streaming output at five different times (5 seconds apart) using a loop
 - i. Use code similar to 6c
 - ii. Don't forget to write code using the correct `queryName` to obtain counts.
 - iii. Your output should show 5 different tables where the count column value is increasing (between the first and last outputs).
 1. If your query has finished running by the time you execute this loop, then all counts will be the same (the final full count because we are reading finite data). Restart the query, and run this command again
 - iv. Use **`show()`** not **`display()`** – to see changing output

c. Stop the streaming query after the loop has completed printing the desired outputs.

Task 10. Calculate aggregate counts of the various combination of values in Device and gt columns

In this question, we will compute the counts for the combinations found in the **Device** and **gt** columns using multiple methods. There are three types of aggregation methods available in Spark DataFrames- **`groupby()`**, **`cube()`** and **`rollup()`**.

- **`DataFrame.groupby(*.cols)`** produces counts for only the combinations of values actually present in the columns.
- **`DataFrame.cube(*.cols)`** – creates a multidimensional cube with all possible combinations of column values, with null in one column used to indicate the total counts in other columns (marginal counts). Nulls in all columns indicate overall total count of records.

- **DataFrame.rollup(*.cols)** - creates a hierarchy of subtotal values of GroupedData using the given columns starting from the first given column. This produces a subset of the rows given by DataFrame.cube().

Call them with the appropriate columns. We will use all three to see the differences.

We can use the streaming DataFrame streamingDF that we created above in Task 6a that contains the streaming input and then specify an aggregation transformation on *streamingDF*. As seen above in tasks 7, 8 and 9- we have to write code to perform several steps to view streaming outputs of the transformation.

- First, we have to set up a transformation to perform the computation that we want on streamingDF (as we did in Task 7). Here specify one of the above three aggregations as the transformation, and save the output to a new streamingDF.
 - Say *devActivityCount_groupbyDF* [then, *devActivityCount_cubeDF* and *devActivityCount_rollupDF*]
 - Set up a stream to write output to a sink (as in Task 8a). Since we are doing aggregations write to 'memory' as sink and use 'complete' as outputMode. You have to use a unique sql table name for each streamingQuery output table. This is specified using **queryName**. In order for queries to run simultaneously they have to have a *unique* queryName.
 - For example, you can use 'DAC_groupby', 'DAC_cube' and 'DAC_rollup'
 - Save the streaming query
 - you can save the output of the above step as: *DACQuery1*, *DACQuery2* and *DACQuery3*
 - Next, set up a loop to read from the streaming output **five (5) times** and view it, as we did in Task 9b. (write this code in a new cell)
 - Output will contain 3 columns
 - Ensure that the **output is changing between first and last tables. If not adjust your sleep time.**
 - Finally, **stop() each query** before starting the next one, as we did in Task 9c.
 - CAVEAT: you can run more than one query at the same time, but may run into out of memory issues on the Databricks Community Edition. So you may try it but if you run into memory issues, then just make sure to stop queries after you have obtained the desired results before starting another query.
- a. Perform the above steps for DataFrame method **groupby()** as the aggregation.
 - b. Perform the above steps for DataFrame method **cube()** as the aggregation.
 - c. Perform the above steps for DataFrame method **rollup()** as the aggregation.

NOTE:

At any time, you can check what streams are currently active by typing

spark.streams.active

- will show queries that aren't stopped.
- This may take a while to produce an output, if the streams are processing (ie.. the loops are still going)

Task 11. For each user 'a' and 'b', calculate the count of rows where the *gt* column contains 'stairs' and Device = 'nexus4_1'

- d. Write and run a streaming query called *stairsonlyDF* where the output contains a table with the streaming counts of the number of times the column named '**gt**' contains 'stairs' anywhere in the column value, and when the user used 'nexus4_1' as their '**Device**', only for users 'a' and 'b'
 - i. First specify the appropriate transformation on streamingDF for this task. You may use ***pyspark.sql.DataFrame.select()***, ***pyspark.sql.DataFrame.withcolumn()***, ***pyspark.sql.DataFrame.groupby()***, ***pyspark.sql.column.contains()***, ***pyspark.sql.DataFrame.where()*** or ***pyspark.sql.DataFrame.filter()*** as needed
 - ii. Set up a ***StreamingQuery*** and save it
 - Use 'update' as the output mode.
 - Write out to 'memory' as the sink.
 - Specify a unique ***queryName***
- e. Display the streaming output 5 times. The output will have 2 columns – '**User**' and '**count**'.
 - i. Notice how the output using 'update' is different from that using 'complete' for aggregations
 - ii. The output should show only two unique values for user - 'a' and 'b' with increasing number of rows in later outputs.
- f. Stop the query, after the loop has completed printing the desired outputs.
 - i. Ensure that the output displays changing results.

Task 12. Compute the averages of each column '**x**', '**y**', '**z**' and counts of Device column for each combination of values for '**Model**' and '**gt**' columns

- a. Write and run a streaming query called *gtmodelstatsQuery* where your streaming output is grouped by columns '**gt**' and '**Model**' columns of the streaming DataFrame *streamingDF* and computes 4 new columns –
 - The average of x in a column called '**x_avg**',

- the average of y in a column called '**y_avg**',
 - the average of z in a column called '**z_avg**', and
 - the count of the values in Device column in a column labeled '**devicecount**'.
- i. Once again, specify the appropriate transformations on streamingDF
You may use *pyspark.sql.DataFrame.groupby()*, *pyspark.sql.DataFrame.agg()*, *pyspark.sql.functions.avg()* and *pyspark.sql.functions.count()*, as you need.
 - ii. Setup the *StreamingQuery object* with '*complete*' as the output mode and '*memory*' as the sink format.
 - iii. Use a unique queryName
- b. Display /show the streaming output – but instead of the whole query table - apply a filter and only show the rows where the x_avg column is greater than 0. Use a loop to repeat this output 5 times.
- i. The output will have 6 columns – '**gt**', '**Model**', '**x_avg**', '**y_avg**', '**z_avg**' and '**devicecount**'.
 - ii. This last filter should be applied after writing out to a sink.

This Task shows you that we can also apply transformations on the intermediate streaming output table that is written to a sink (in our case '*memory*').

- c. Stop the query, after the loop has completed printing the desired outputs.

Task 13. Perform a join of a static DataFrame with a streaming DataFrame

Often we may have static data that is historical, and also have access to fresh streaming data. we may then want to join some stats from historical data with stats from new streaming data to easily see how data are changing over time, for instance. In this task, we will *join* the average values of the sensor orientation columns '**x**', '**y**', '**z**' calculated by grouping the columns '**gt**' and '**Model**', for the static data with that for the streaming data.

- a. First compute the historical outputs by specifying appropriate transformation on *staticDF*
 - i. We had created this earlier in the homework.
 - ii. Group by columns '**gt**' and '**Model**', and compute the average values of '**x**', '**y**', '**z**' columns for the static DataFrame, *staticDF*, and save the results in another DataFrame *historicalAvgDF*.
 - iii. Label the new columns as '**x_avg_h**', '**y_avg_h**', '**z_avg_h**' (may use *Column.alias()*)
- b. Calculate the averages for x, y, z for the streaming DataFrame, *streamingDF*
 - i. Label the streaming average columns as '**x_avg_s**', '**y_avg_s**', '**z_avg_s**'.

- ii. Then call the ***join()*** method to join the *historicalAvgDF* with the above streaming averages DataFrame on columns '**gt**' and '**Model**'.
 - iii. Finally, set up to write out the preceding streaming output (the joined table) to '*memory*' sink in '*complete*' output mode by calling ***.writeStream***. Provide a unique queryName for the sql output table.
 - iv. Start the query and save the results of the joined table in a new streaming query – give it a name of your choice.
 - v. The above commands can be specified as one long chain of commands
- c. Display /show the streaming output (the joined table) 5 times.
- i. You may need to adjust the sleep time in order to see one set of averages change.
 - ii. The output will have 8 columns – '**gt**', '**Model**', '**x_avg_h**', '**y_avg_h**', '**z_avg_h**', '**x_avg_s**', '**y_avg_s**', '**z_avg_s**'.
- d. Stop the query when 5 outputs have been produced.
- e. Which set of averages changes over time? Theoretically, one set will stay the same.
- i. Write down your answer in a new cell as a comment.

FINAL WORDS

For tasks 8-13, you can try different values of sleep to see changing outputs. Your saved MUST show that the output values in the table are changing results (or you will lose points).

So DO NOT use Run ALL – you have to run commands one by one for this assignment. Run the loop only after the stream has started (if you run the loop before that, you will get empty output tables). If you run the loop too later (after the stream has finished reading all data), the outputs will all be the same and wont change.

All done!