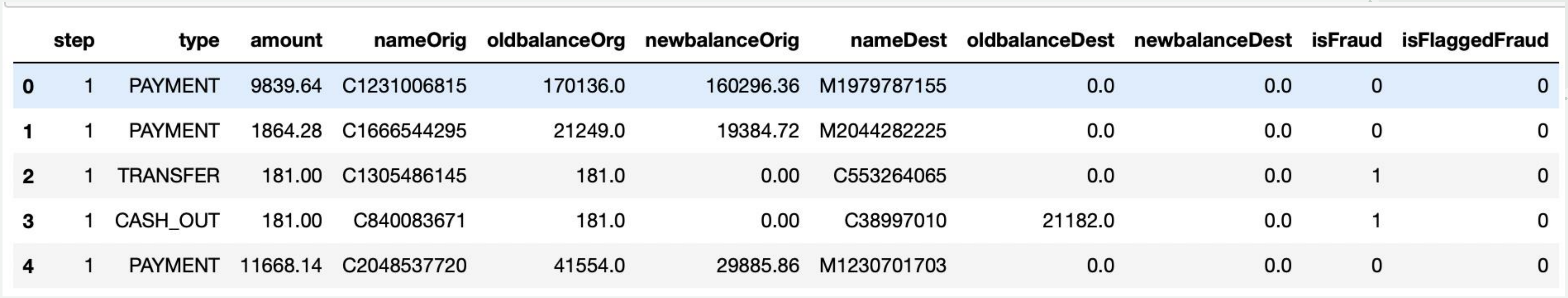
Streaming system using PySpark

BY PRATISHTHA SHARMA (IDS2022004) AND GOMTESH JAIN (IDS2022006)

About the data

- Paysim synthetic dataset of mobile money transactions. Each step represents an hour of simulation
- Shape of Data - (6362620, 11)
- there is **no missing** and **garbage value**, there is no need for data cleaning



	step	type	amount	nameOrig	oldbalanceOrg	newbalanceOrig	nameDest	oldbalanceDest	newbalanceDest	isFraud	isFlaggedFraud
0	1	PAYMENT	9839.64	C1231006815	170136.0	160296.36	M1979787155	0.0	0.0	0	0
1	1	PAYMENT	1864.28	C1666544295	21249.0	19384.72	M2044282225	0.0	0.0	0	0
2	1	TRANSFER	181.00	C1305486145	181.0	0.00	C553264065	0.0	0.0	1	0
3	1	CASH_OUT	181.00	C840083671	181.0	0.00	C38997010	21182.0	0.0	1	0
4	1	PAYMENT	11668.14	C2048537720	41554.0	29885.86	M1230701703	0.0	0.0	0	0

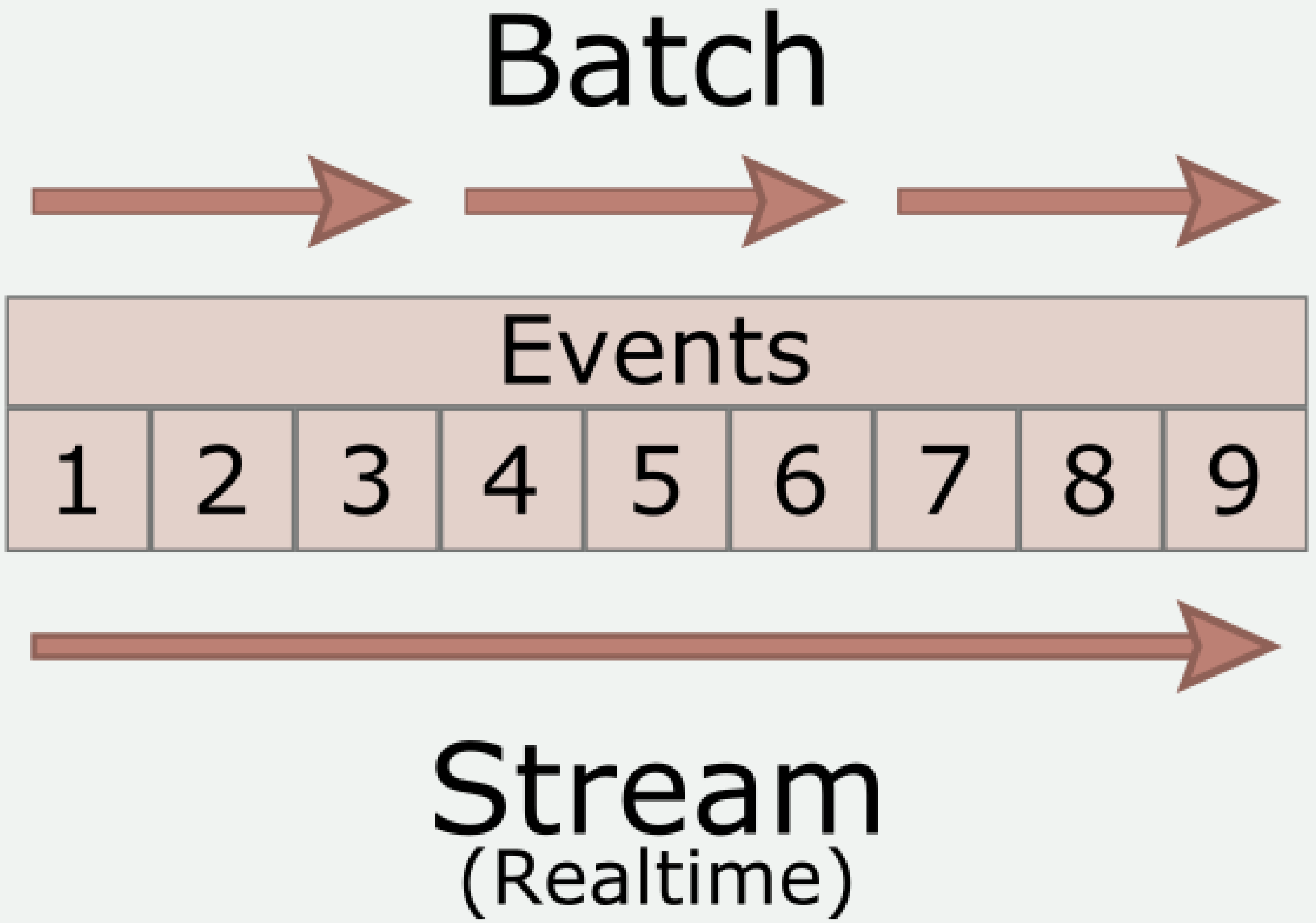
Technologies used are

- Python
- PySpark
- Jupiter notebook
- Pandas
- SQL

All About Streaming

- Also known as event stream processing, streaming data is the continuous flow of data generated by various sources.

- **1. Bounded Stream**
- The bounded stream will have a defined start and an end. While processing a bounded stream, we can ingest the entire data-set before starting any computation. Hence, we could perform operations such as sorting and summarize data. The processing of bounded stream is referred to as Batch processing.
- **2. Unbounded Stream** The unbounded stream will have a start but no end. Hence, the data needs to be continuously processed when it is generated. Data is processed based on the event time (Event time is the time that each individual event occurred on its producing device). The paradigm of processing unbounded stream is referred to as Stream processing.

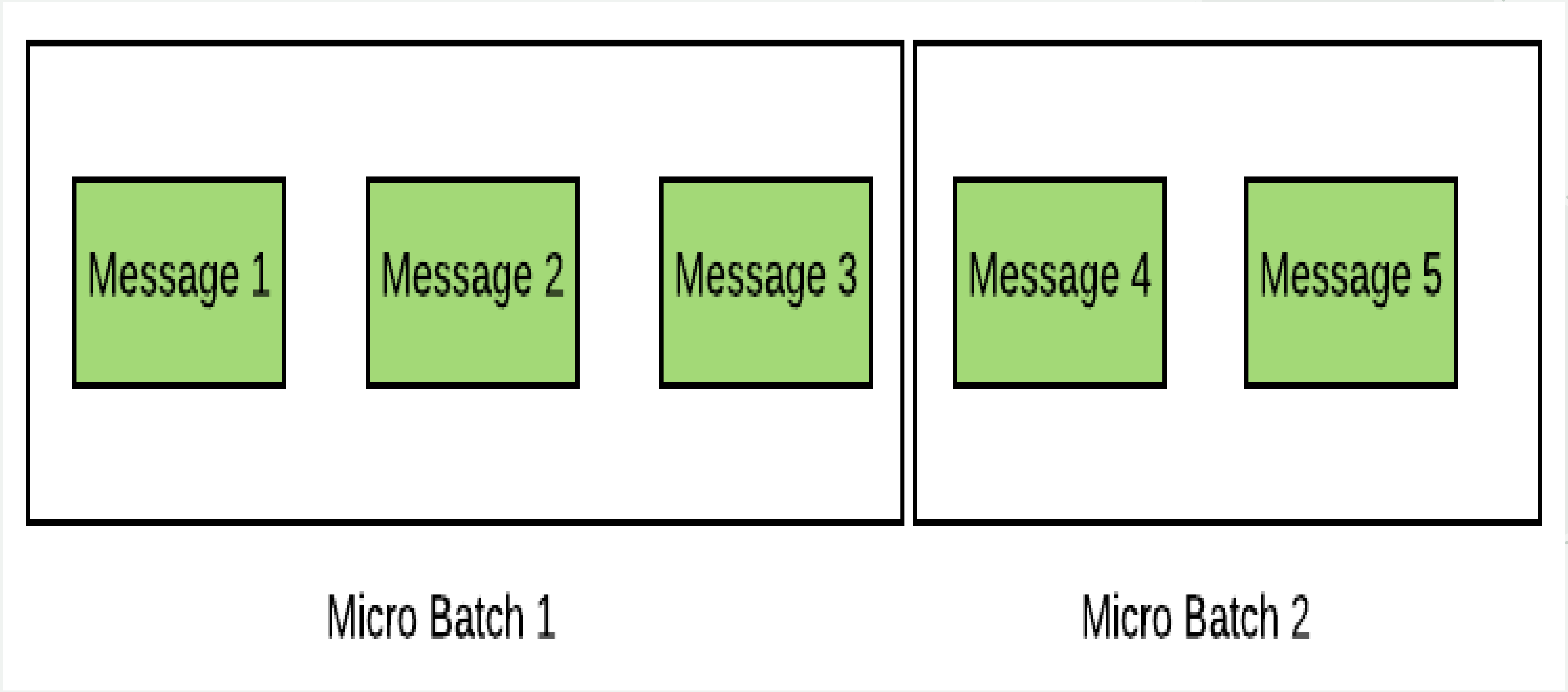

Batch
Events
1
2
3
4
5
6
7
8
9
Stream
(Realtime)

Spark Structured Streaming

- Spark structured streaming allows for near-time computations of streaming data over Spark SQL engine to generate aggregates or output as per the defined logic.
- This streaming data can be read from a file, a socket, or sources such as Kafka.
- the core logic of the implementation for processing is very closely related to how you would process that data in batch mode.

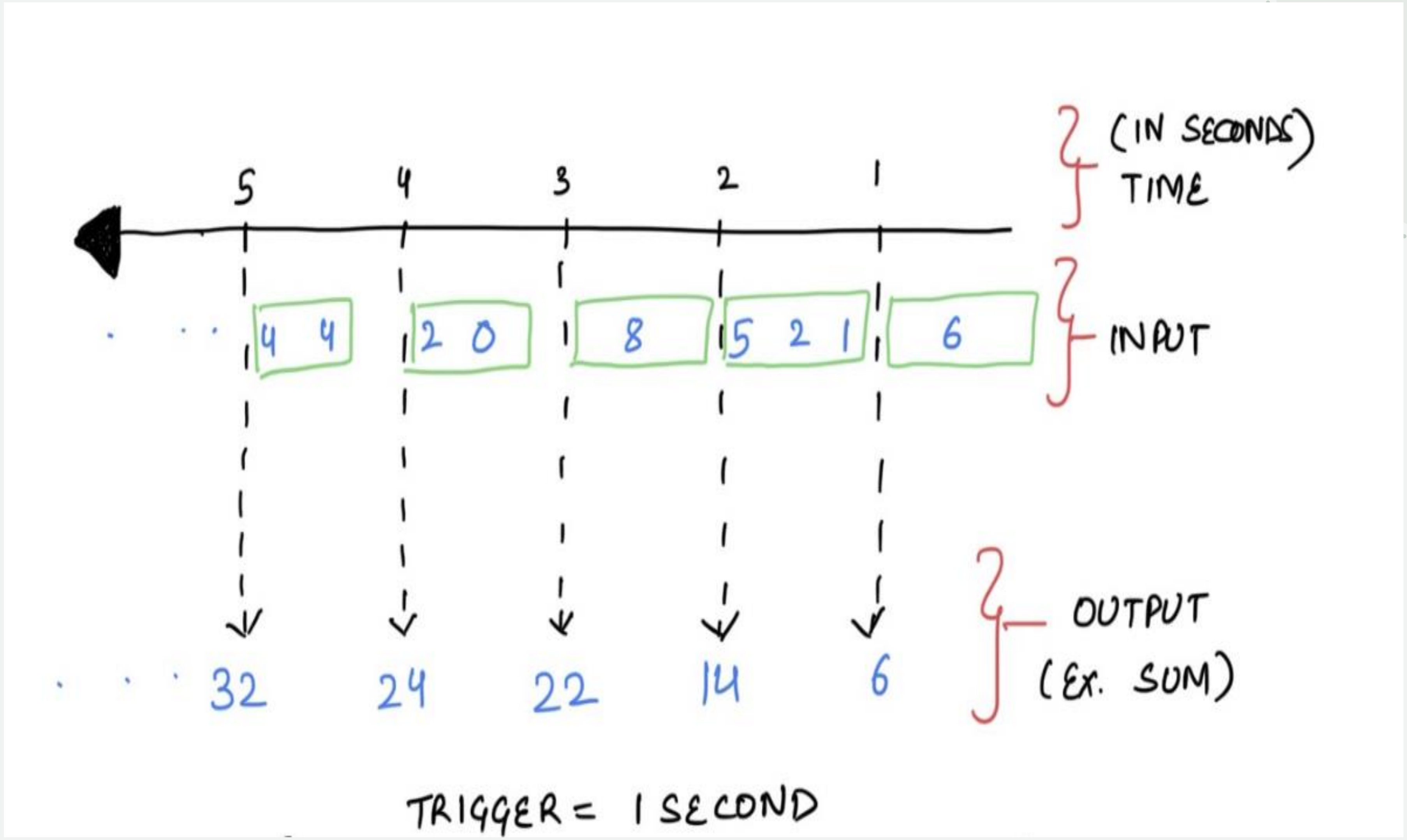
Micro batches: Structured Streaming

- Structured Streaming has a concept of micro-batches to process the data, meaning that not every record is processed as it comes. Instead, they are accumulated in small batches and these micro(tiny batches) are processed together i.e. near real-time. You can configure your micro-batch and can go as low as a few ms.


Message 1
Message 2
Message 3
Message 4
Message 5
Micro Batch 1
Micro Batch 2

Triggers

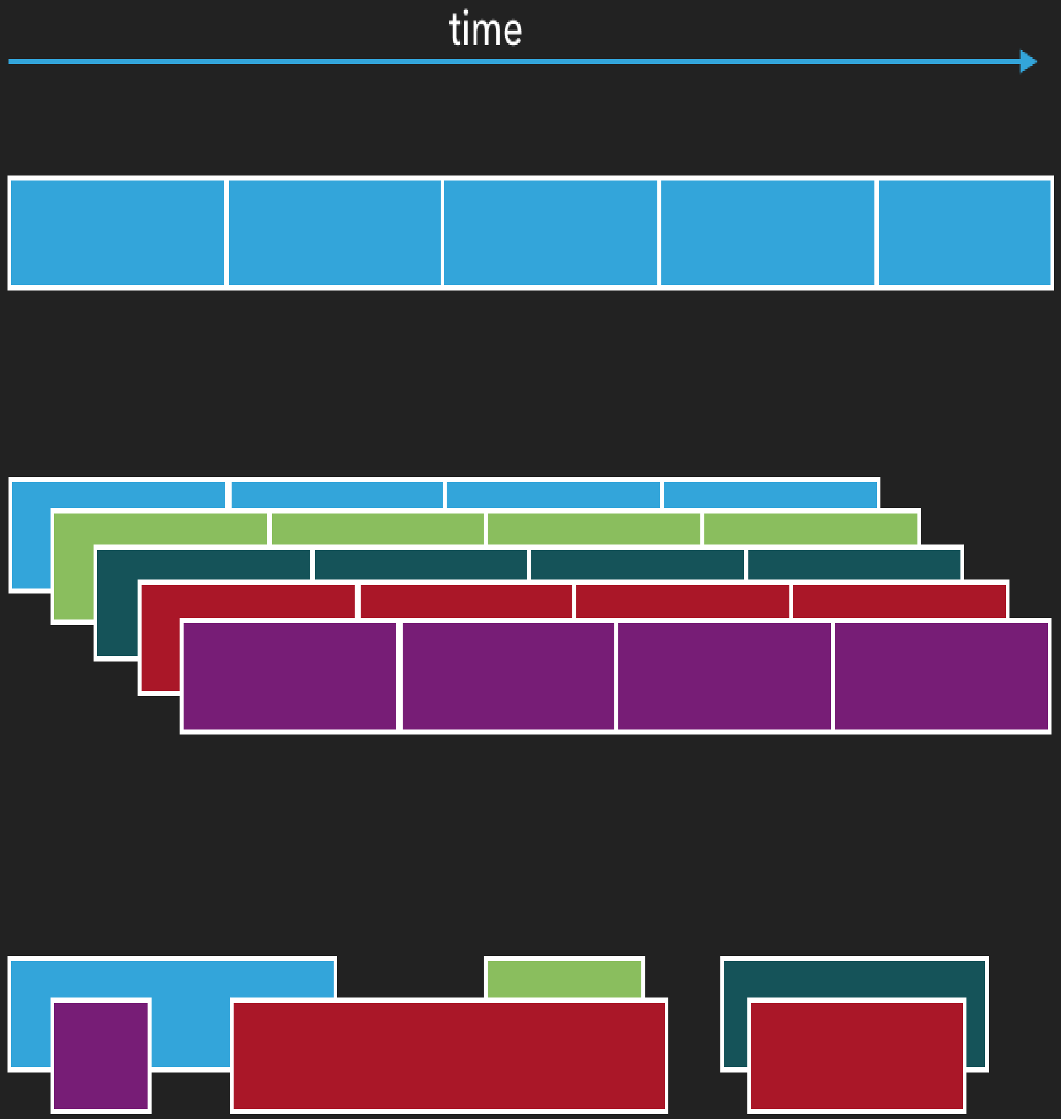
- Now how does Spark knows when to generate these micro-batches and append them to the unbounded table? This mechanism is called triggering. As explained, not every record is processed as it comes, at a certain interval, called the “trigger” interval, a micro-batch of rows gets appended to the table and gets processed. This interval is configurable, and has different modes, by default mode is start the next process as previous finishes.


(IN SECONDS)
TIME
INPUT
OUTPUT
(Ex. SUM)
5
4
3
2
1
4 4
2 0
8
5 2 1
6
32
24
22
14
6
TRIGGER = 1 SECOND

Streaming Windows

- **Fixed/Tumbling:** time is partitioned into same-length, non-overlapping chunks. Each event belongs to exactly one window
- **Sliding windows:** have fixed length, but are separated by a time interval (step) which can be smaller than the window length. Typically the window interval is a multiplicity of the step. Each event belongs to a number of windows ([window interval]/[window step]).
- **Session:** windows have various sizes and are defined basing on data, which should carry some session identifiers

- Time-based
  - fixed/tumbling
  - sliding
- Session-based



About the Project

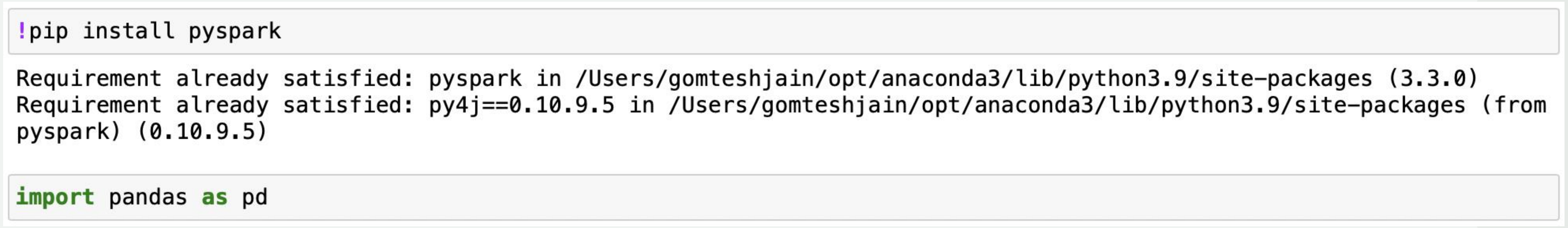
- Initially we had taken a Bank data of fraudulent transaction detection through credit cards which has around 63 Lakhs rows and 11 column and size is around 500 Mb. and divided them according to time stamps into various of .csv files which were around 800 in number using the GROUP BY function of SQL on Step Column. i.e. Maps a unit of time in the real world. In this case 1 step is 1 hour of time.
- Group by function help us to identify steps and its count which means number of rows or number of transitions on that particular step.
- We initially had 11 columns out of which 2 are found to be redundant which were removed during the data cleaning process to avoid crowding.

CONTINUE…

- In these particular files, records of data which only carried one transaction in a time stamp were further removed to further remove redundancy.
- After this step we had around 63 lakh rows and 9 columns.
- After that our main objective that was to design a streaming system using pyspark and perform aggregation analytics of SQL.
- In this structured streaming, we have used a function called as “maxFilesPerTrigger” and set it value to 1 but its value is variable and since we were getting best results keeping the value equal to 1 .
- We have used various kinds of streaming windows on self-made data set ans check the difference in results.

CODE

We Install the needful packages

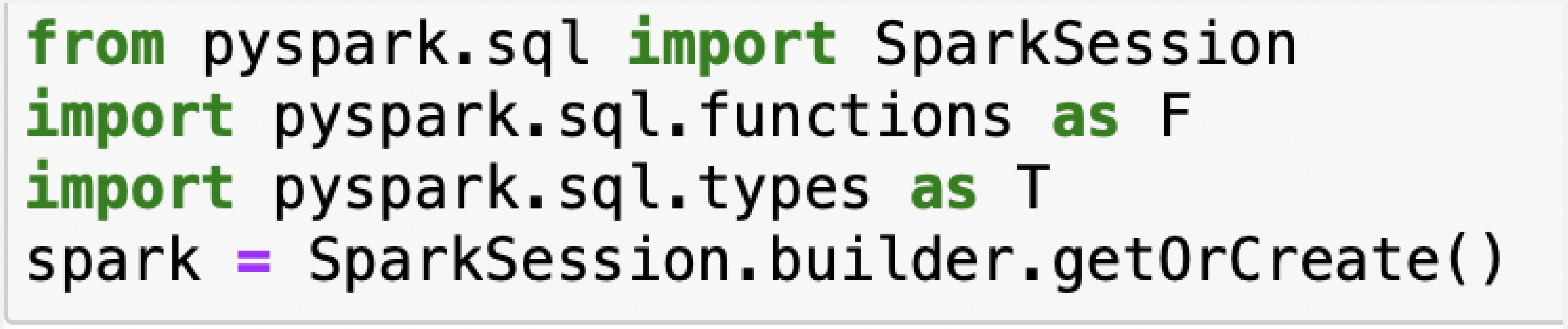


```
!pip install pyspark
```

```
Requirement already satisfied: pyspark in /Users/gomteshjain/opt/anaconda3/lib/python3.9/site-packages (3.3.0)
Requirement already satisfied: py4j==0.10.9.5 in /Users/gomteshjain/opt/anaconda3/lib/python3.9/site-packages (from
pyspark) (0.10.9.5)
```

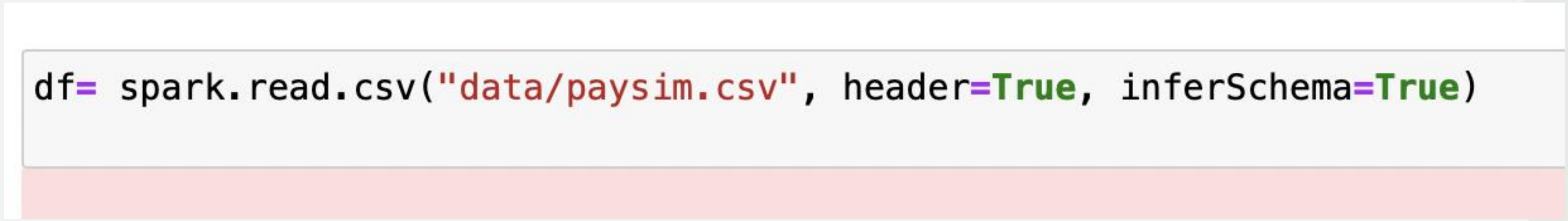
```
import pandas as pd
```

We import needful library



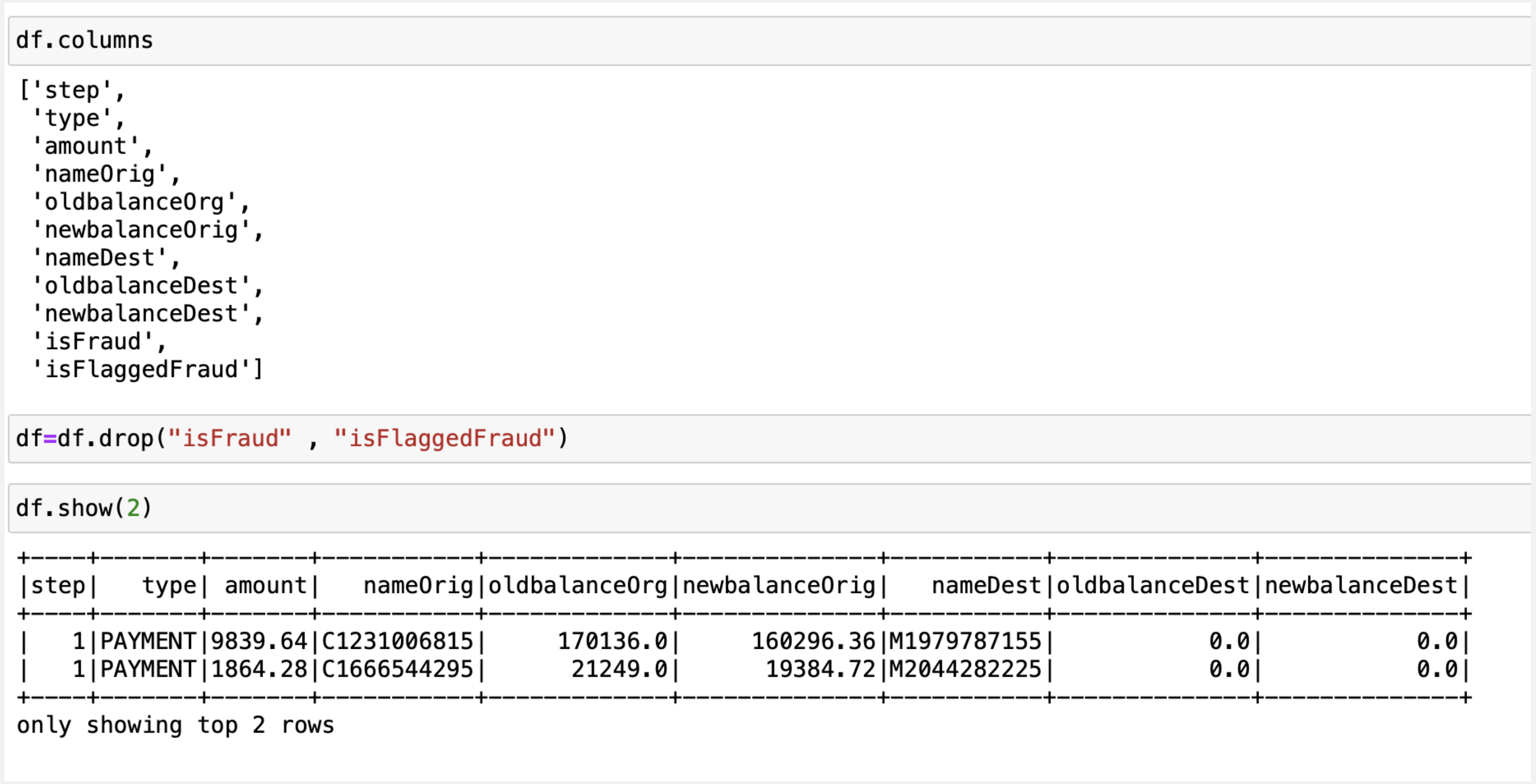
```
from pyspark.sql import SparkSession
import pyspark.sql.functions as F
import pyspark.sql.types as T
spark = SparkSession.builder.getOrCreate()
```

Read the CSV file



```
df= spark.read.csv("data/paysim.csv", header=True, inferSchema=True)
```

Drop unwanted columns and print the table



```
df.columns
```

```
['step',
 'type',
 'amount',
 'nameOrig',
 'oldbalanceOrg',
 'newbalanceOrig',
 'nameDest',
 'oldbalanceDest',
 'newbalanceDest',
 'isFraud',
 'isFlaggedFraud']
```

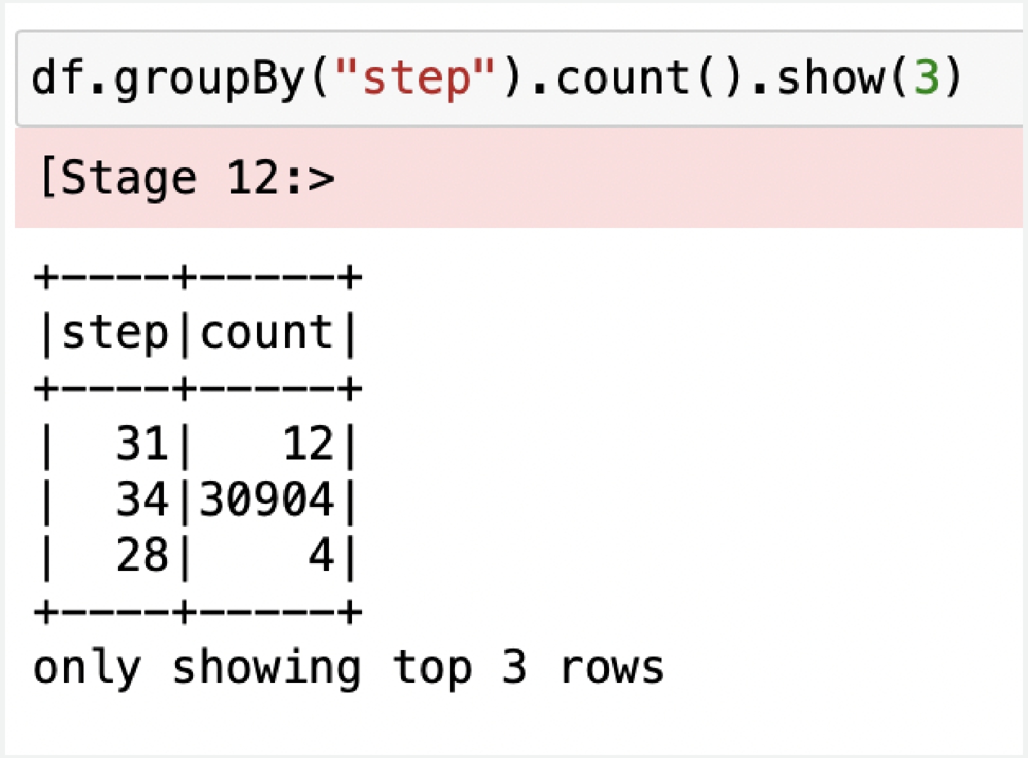
```
df=df.drop("isFraud" , "isFlaggedFraud")
```

```
df.show(2)
```

```
+----+-------+-------+-----------+-------------+--------------+-----------+--------------+--------------+
|step|   type| amount|   nameOrig|oldbalanceOrg|newbalanceOrig|   nameDest|oldbalanceDest|newbalanceDest|
+----+-------+-------+-----------+-------------+--------------+-----------+--------------+--------------+
|   1|PAYMENT|9839.64|C1231006815|     170136.0|     160296.36|M1979787155|           0.0|           0.0|
|   1|PAYMENT|1864.28|C1666544295|      21249.0|      19384.72|M2044282225|           0.0|           0.0|
+----+-------+-------+-----------+-------------+--------------+-----------+--------------+--------------+
only showing top 2 rows
```

GROUP BY

- Print group by on Step with count of transactions in that each step
- Step maps a unit of time in the real world. In this case 1 step is 1 hour of time.
- So we can assume for this example that we have another job that runs every hour and gets all the transactions in that time frame.

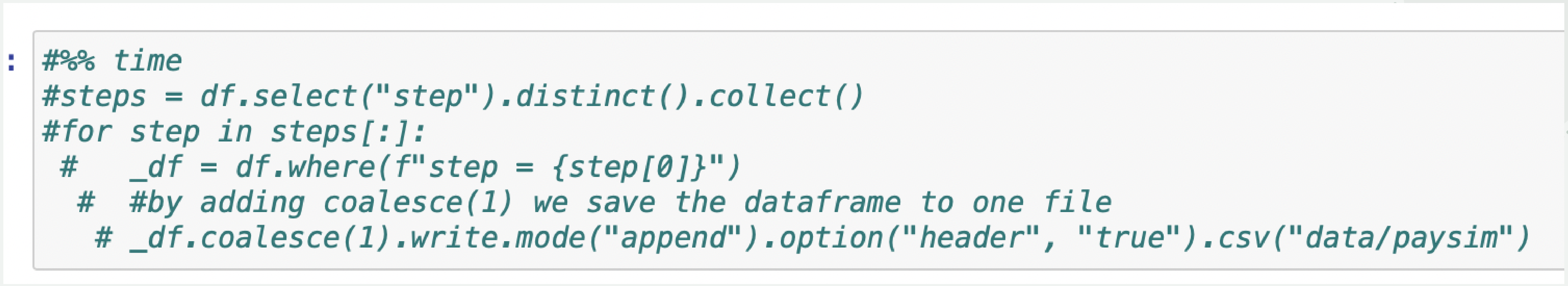


```
df.groupBy("step").count().show(3)
```

```
[Stage 12:>

+----+-----+
|step|count|
+----+-----+
|  31|   12|
|  34|30904|
|  28|    4|
+----+-----+
only showing top 3 rows
```

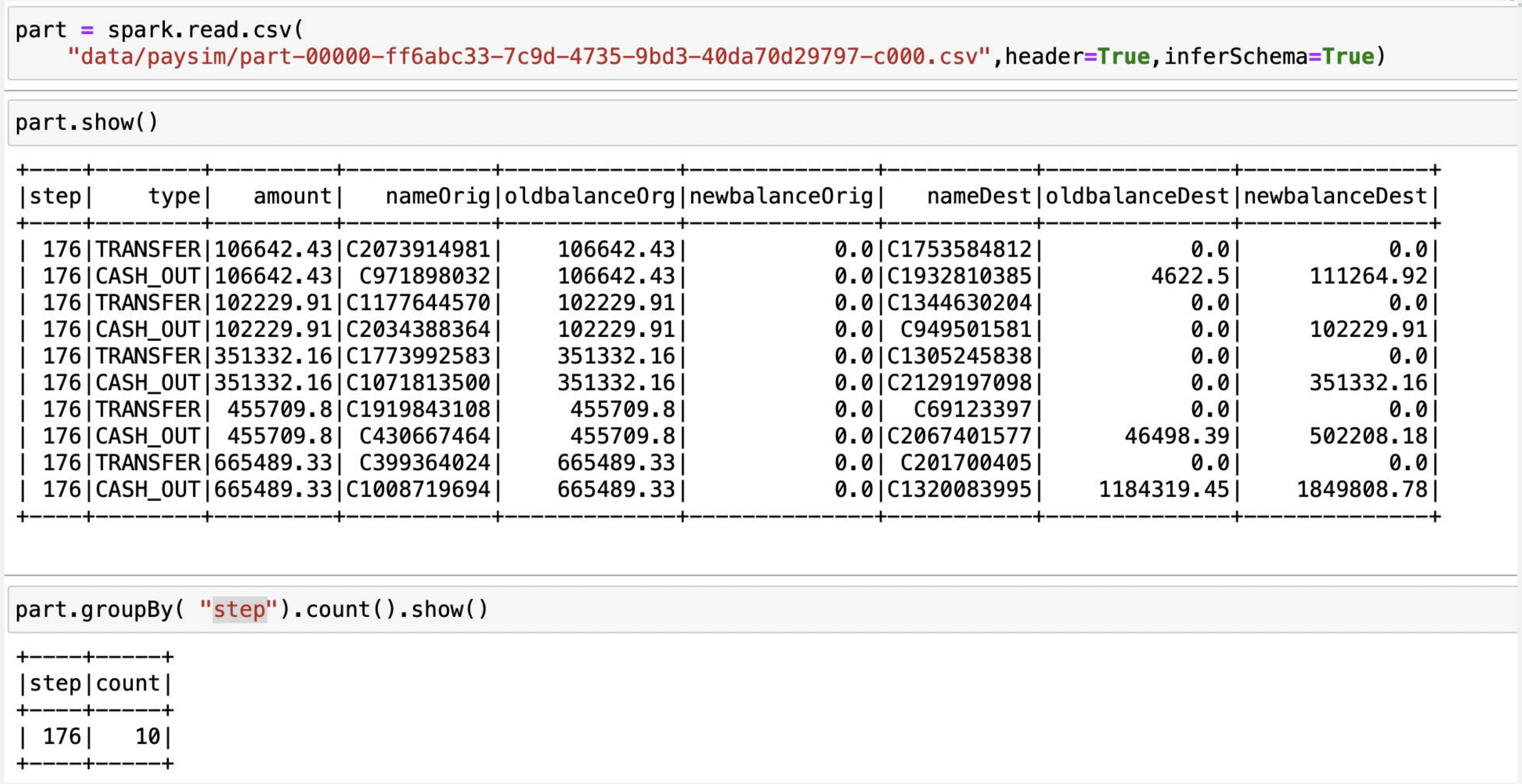
SPLIT CSV

- In this particular step , we have to break the csv into multiple csv which is broken on basis of different time
- We can therefore save the output of that job by filtering on each step and saving it to a separate file



```
:  #%% time
   #steps = df.select("step").distinct().collect()
   #for step in steps[:]:
    #   _df = df.where(f"step = {step[0]}")
     #  #by adding coalesce(1) we save the dataframe to one file
      # _df.coalesce(1).write.mode("append").option("header", "true").csv("data/paysim")
```

Print about csv names and 1 csv file



```
part = spark.read.csv(
    "data/paysim/part-00000-ff6abc33-7c9d-4735-9bd3-40da70d29797-c000.csv",header=True,inferSchema=True)
```

```
part.show()
```

```
+----+--------+---------+-----------+-------------+--------------+-----------+--------------+--------------+
|step|    type|   amount|   nameOrig|oldbalanceOrg|newbalanceOrig|   nameDest|oldbalanceDest|newbalanceDest|
+----+--------+---------+-----------+-------------+--------------+-----------+--------------+--------------+
| 176|TRANSFER|106642.43|C2073914981|    106642.43|           0.0|C1753584812|           0.0|           0.0|
| 176|CASH_OUT|106642.43| C971898032|    106642.43|           0.0|C1932810385|        4622.5|     111264.92|
| 176|TRANSFER|102229.91|C1177644570|    102229.91|           0.0|C1344630204|           0.0|           0.0|
| 176|CASH_OUT|102229.91|C2034388364|    102229.91|           0.0| C949501581|           0.0|     102229.91|
| 176|TRANSFER|351332.16|C1773992583|    351332.16|           0.0|C1305245838|           0.0|           0.0|
| 176|CASH_OUT|351332.16|C1071813500|    351332.16|           0.0|C2129197098|           0.0|     351332.16|
| 176|TRANSFER| 455709.8|C1919843108|     455709.8|           0.0|  C69123397|           0.0|           0.0|
| 176|CASH_OUT| 455709.8| C430667464|     455709.8|           0.0|C2067401577|      46498.39|     502208.18|
| 176|TRANSFER|665489.33| C399364024|    665489.33|           0.0| C201700405|           0.0|           0.0|
| 176|CASH_OUT|665489.33|C1008719694|    665489.33|           0.0|C1320083995|    1184319.45|    1849808.78|
+----+--------+---------+-----------+-------------+--------------+-----------+--------------+--------------+
```

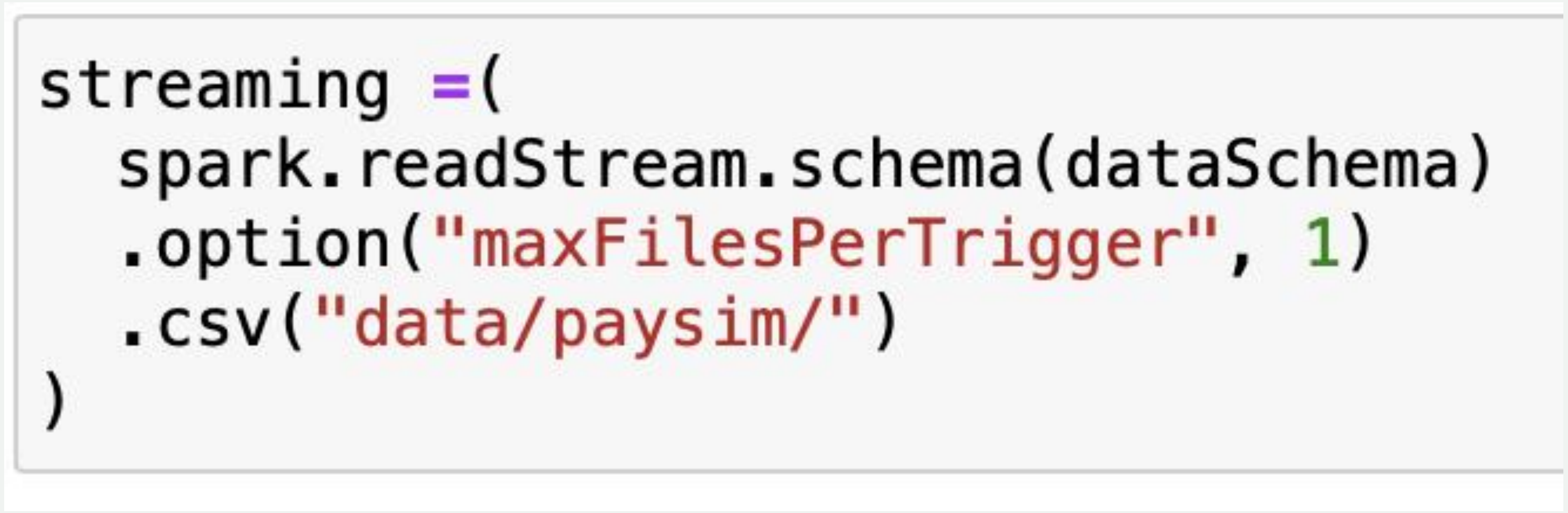
```
part.groupBy( "step").count().show()
```

```
+----+-----+
|step|count|
+----+-----+
| 176|   10|
+----+-----+
```

STREAMING

•A StreamingContext represents the connection to a Spark cluster, and can be used to create Data Stream various input sources. It can be from an existing SparkContext. After creating and transforming DStreams, the streaming computation can be started and stopped using context. start() and context.

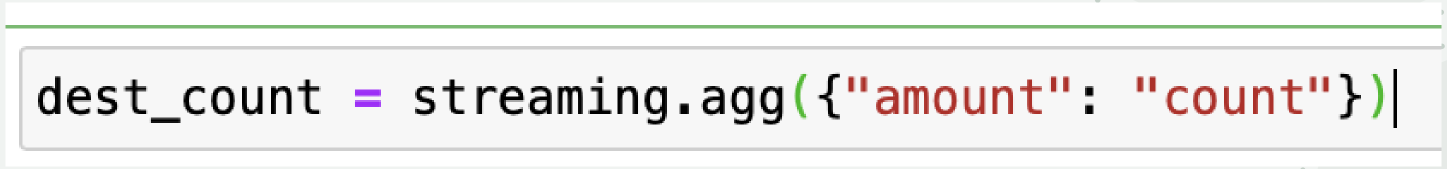
•maxFilesPerTrigger: maximum number of new files to be considered in every trigger



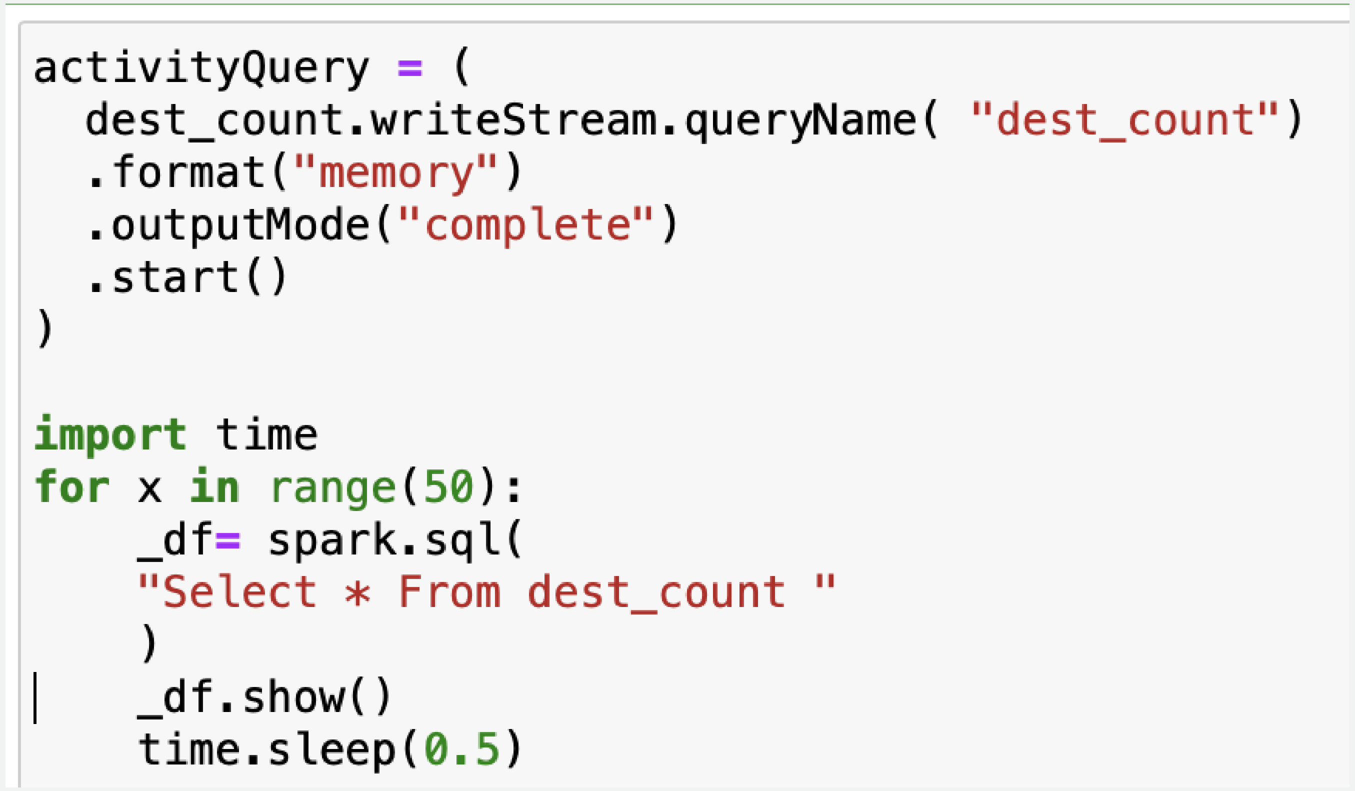
```
streaming =(
  spark.readStream.schema(dataSchema)
  .option("maxFilesPerTrigger", 1)
  .csv("data/paysim/")
)
```

COUNT

- Code of Count



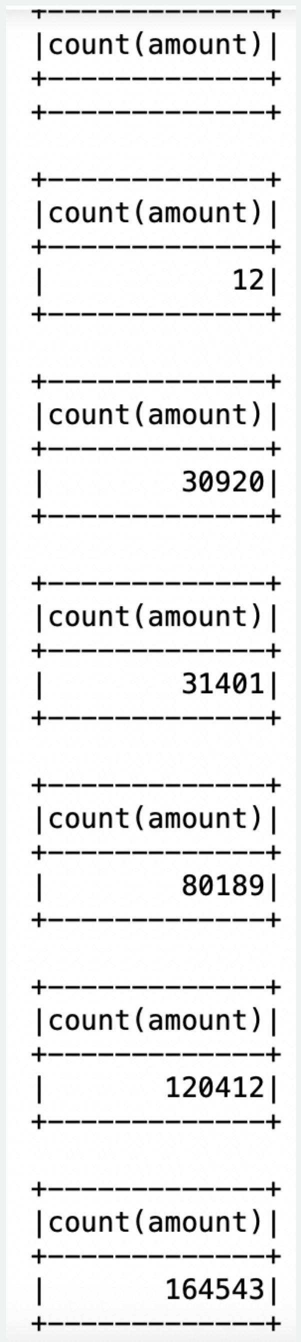
```
dest_count = streaming.agg({"amount": "count"})
```



```
activityQuery = (
  dest_count.writeStream.queryName( "dest_count")
  .format("memory")
  .outputMode("complete")
  .start()
)

import time
for x in range(50):
    _df= spark.sql(
    "Select * From dest_count "
    )
    _df.show()
    time.sleep(0.5)
```

Continue...

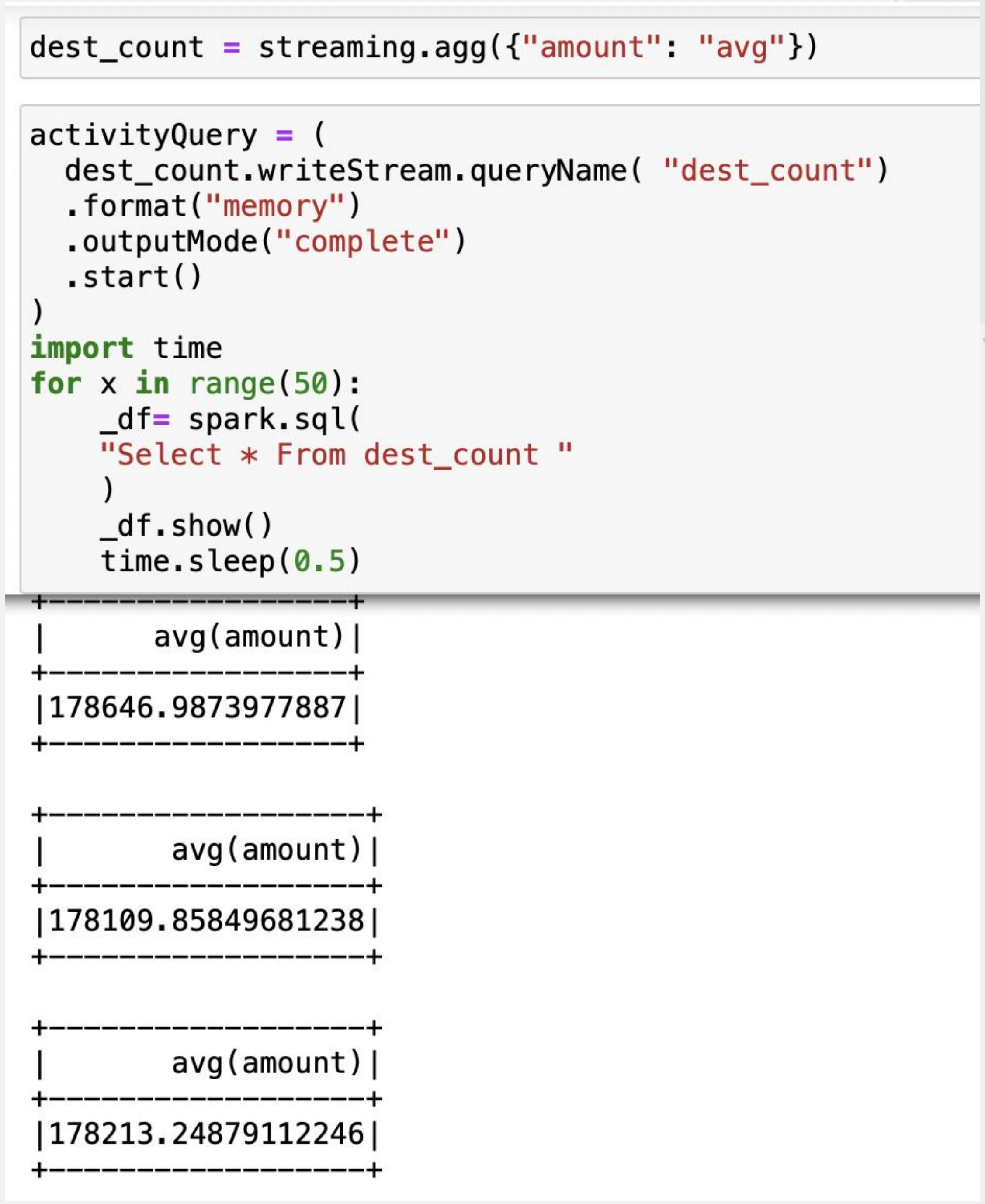
- Count Output



```
+-------------+
|count(amount)|
+-------------+
+-------------+

+-------------+
|count(amount)|
+-------------+
|           12|
+-------------+

+-------------+
|count(amount)|
+-------------+
|        30920|
+-------------+

+-------------+
|count(amount)|
+-------------+
|        31401|
+-------------+

+-------------+
|count(amount)|
+-------------+
|        80189|
+-------------+

+-------------+
|count(amount)|
+-------------+
|       120412|
+-------------+

+-------------+
|count(amount)|
+-------------+
|       164543|
+-------------+
```

AVERAGE

- Code of AVG



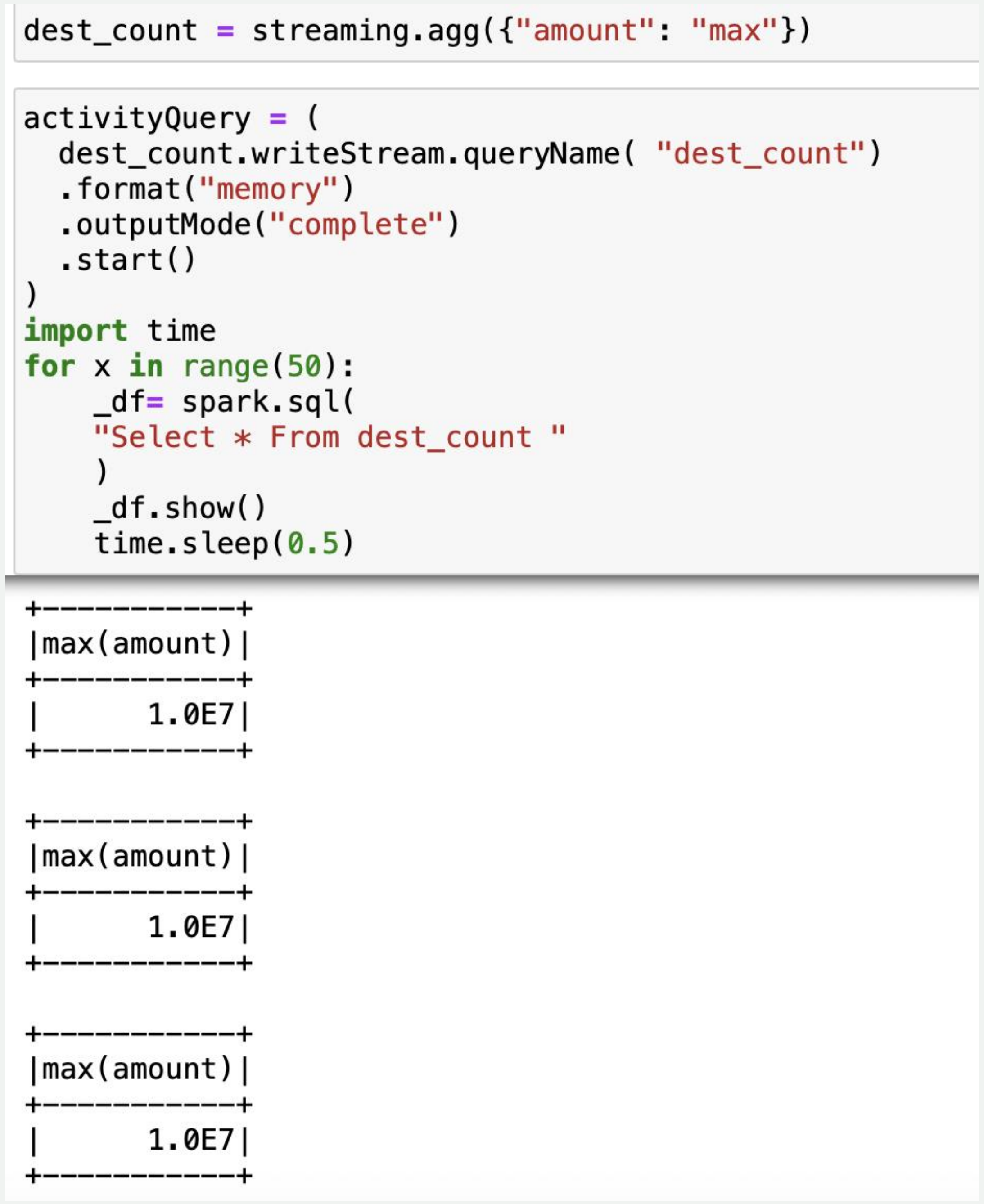
```
dest_count = streaming.agg({"amount": "avg"})
```

```
activityQuery = (
  dest_count.writeStream.queryName( "dest_count")
  .format("memory")
  .outputMode("complete")
  .start()
)
import time
for x in range(50):
    _df= spark.sql(
    "Select * From dest_count "
    )
    _df.show()
    time.sleep(0.5)
```

```
+-----------------+
|      avg(amount)|
+-----------------+
|178646.9873977887|
+-----------------+

+------------------+
|       avg(amount)|
+------------------+
|178109.85849681238|
+------------------+

+------------------+
|       avg(amount)|
+------------------+
|178213.24879112246|
+------------------+
```

MAXIMUM

•Code of MAX



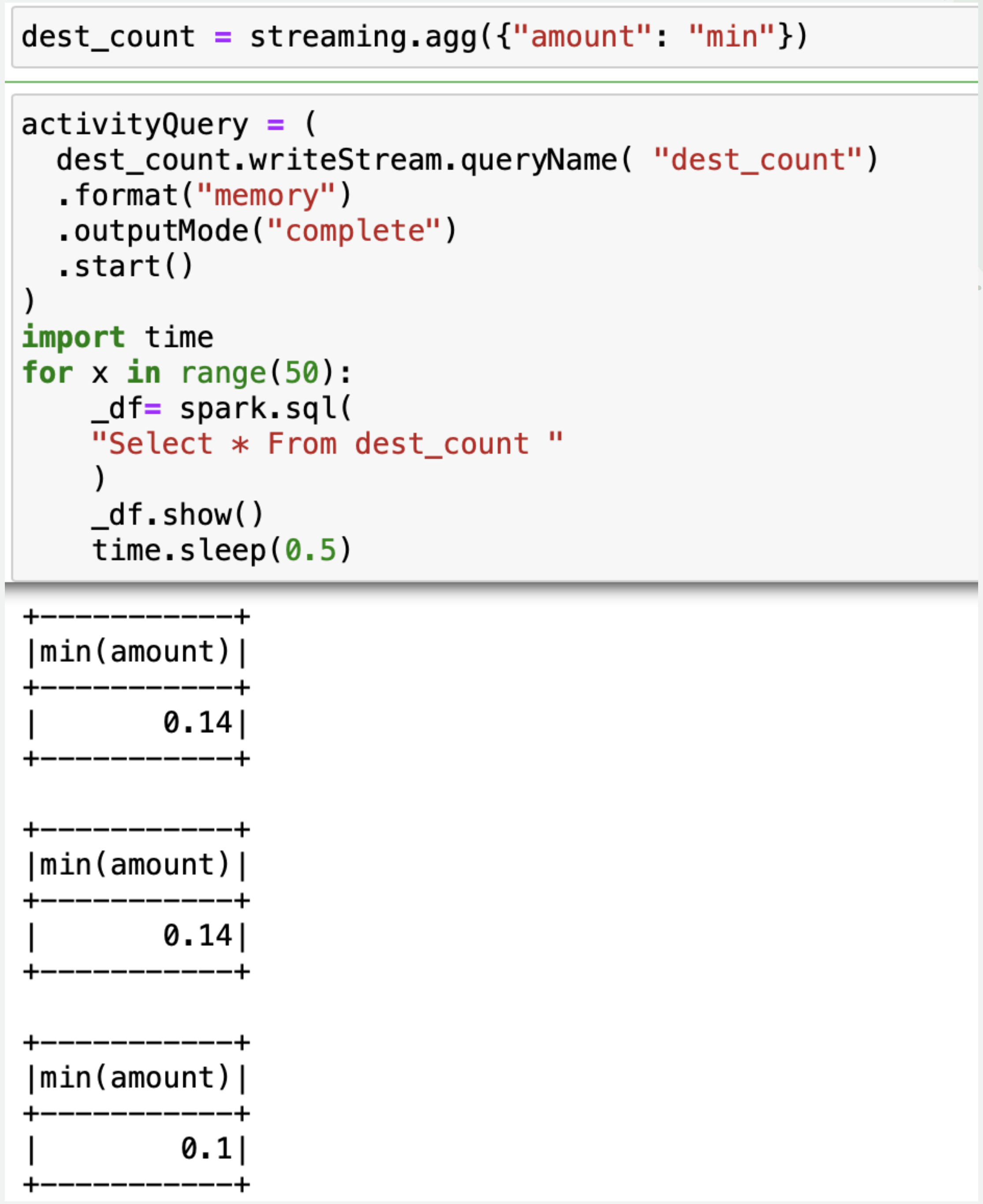
```
dest_count = streaming.agg({"amount": "max"})
```

```
activityQuery = (
  dest_count.writeStream.queryName( "dest_count")
  .format("memory")
  .outputMode("complete")
  .start()
)
import time
for x in range(50):
    _df= spark.sql(
    "Select * From dest_count "
    )
    _df.show()
    time.sleep(0.5)
```

```
+-----------+
|max(amount)|
+-----------+
|      1.0E7|
+-----------+

+-----------+
|max(amount)|
+-----------+
|      1.0E7|
+-----------+

+-----------+
|max(amount)|
+-----------+
|      1.0E7|
+-----------+
```

MINIMUM

- Code of MIN



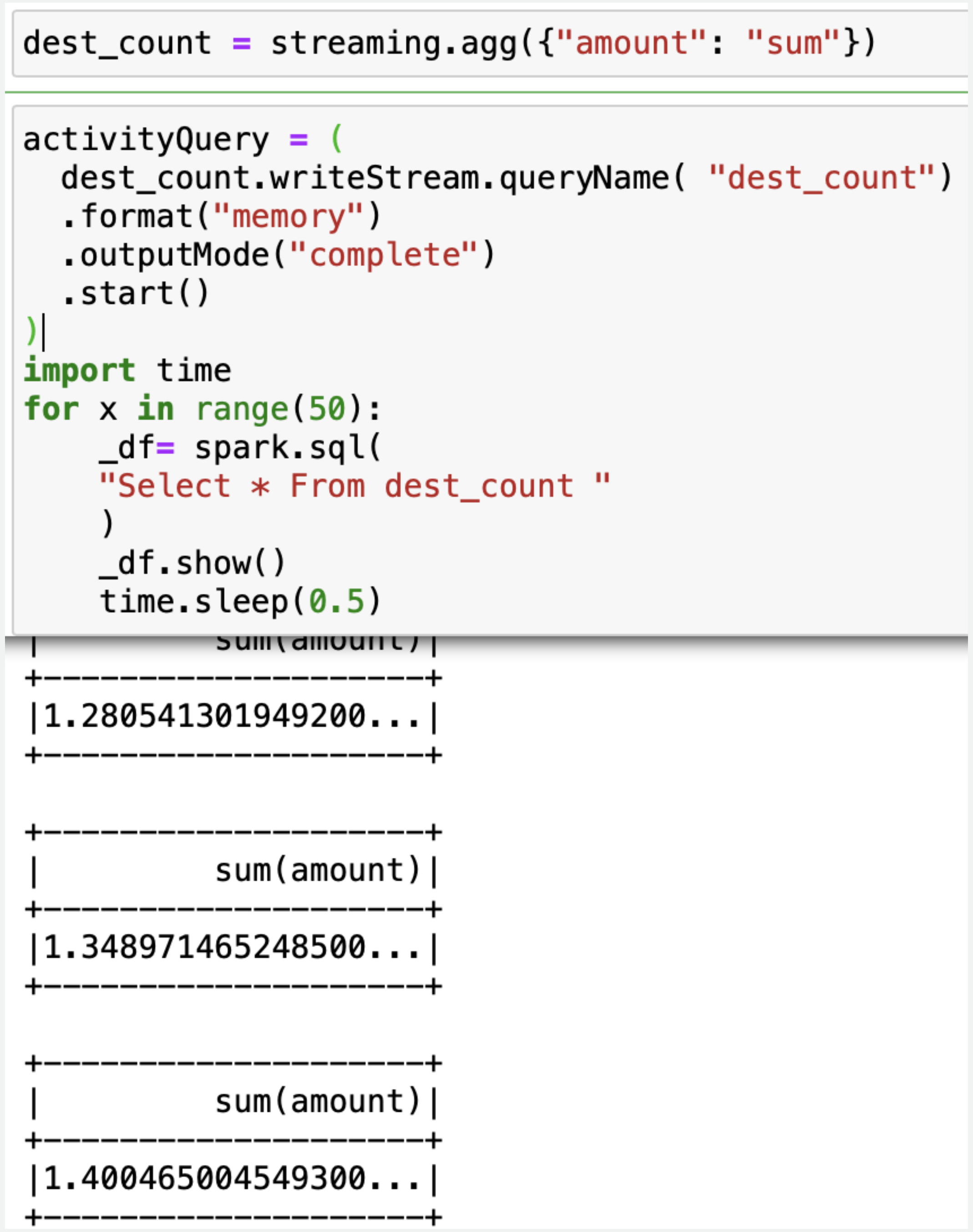
```
dest_count = streaming.agg({"amount": "min"})
```

```
activityQuery = (
  dest_count.writeStream.queryName( "dest_count")
  .format("memory")
  .outputMode("complete")
  .start()
)
import time
for x in range(50):
    _df= spark.sql(
    "Select * From dest_count "
    )
    _df.show()
    time.sleep(0.5)
```

```
+-----------+
|min(amount)|
+-----------+
|       0.14|
+-----------+

+-----------+
|min(amount)|
+-----------+
|       0.14|
+-----------+

+-----------+
|min(amount)|
+-----------+
|        0.1|
+-----------+
```

SUM

- Code of SUM

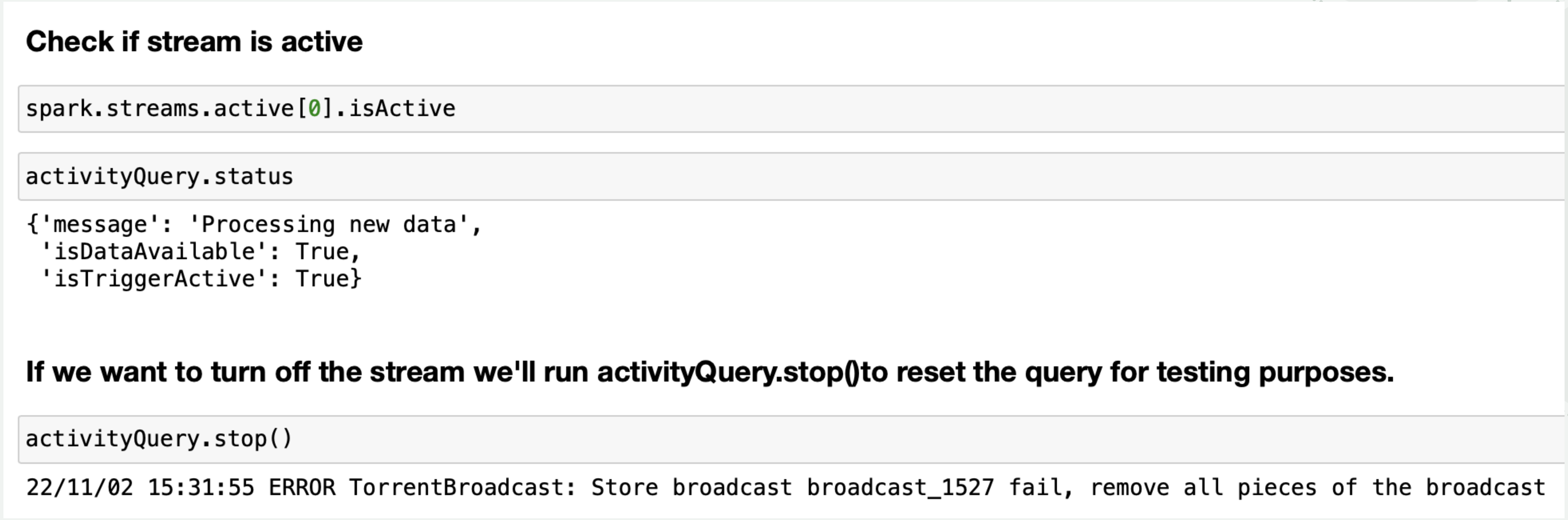


```
dest_count = streaming.agg({"amount": "sum"})
```

```
activityQuery = (
  dest_count.writeStream.queryName( "dest_count")
  .format("memory")
  .outputMode("complete")
  .start()
)
import time
for x in range(50):
    _df= spark.sql(
    "Select * From dest_count "
    )
    _df.show()
    time.sleep(0.5)
```

```
|         sum(amount)|
+--------------------+
|1.280541301949200...|
+--------------------+

+--------------------+
|         sum(amount)|
+--------------------+
|1.348971465248500...|
+--------------------+

+--------------------+
|         sum(amount)|
+--------------------+
|1.400465004549300...|
+--------------------+
```

Stopping the streaming



Check if stream is active

```
spark.streams.active[0].isActive
```

```
activityQuery.status
```

```
{'message': 'Processing new data',
 'isDataAvailable': True,
 'isTriggerActive': True}
```

If we want to turn off the stream we'll run activityQuery.stop()to reset the query for testing purposes.

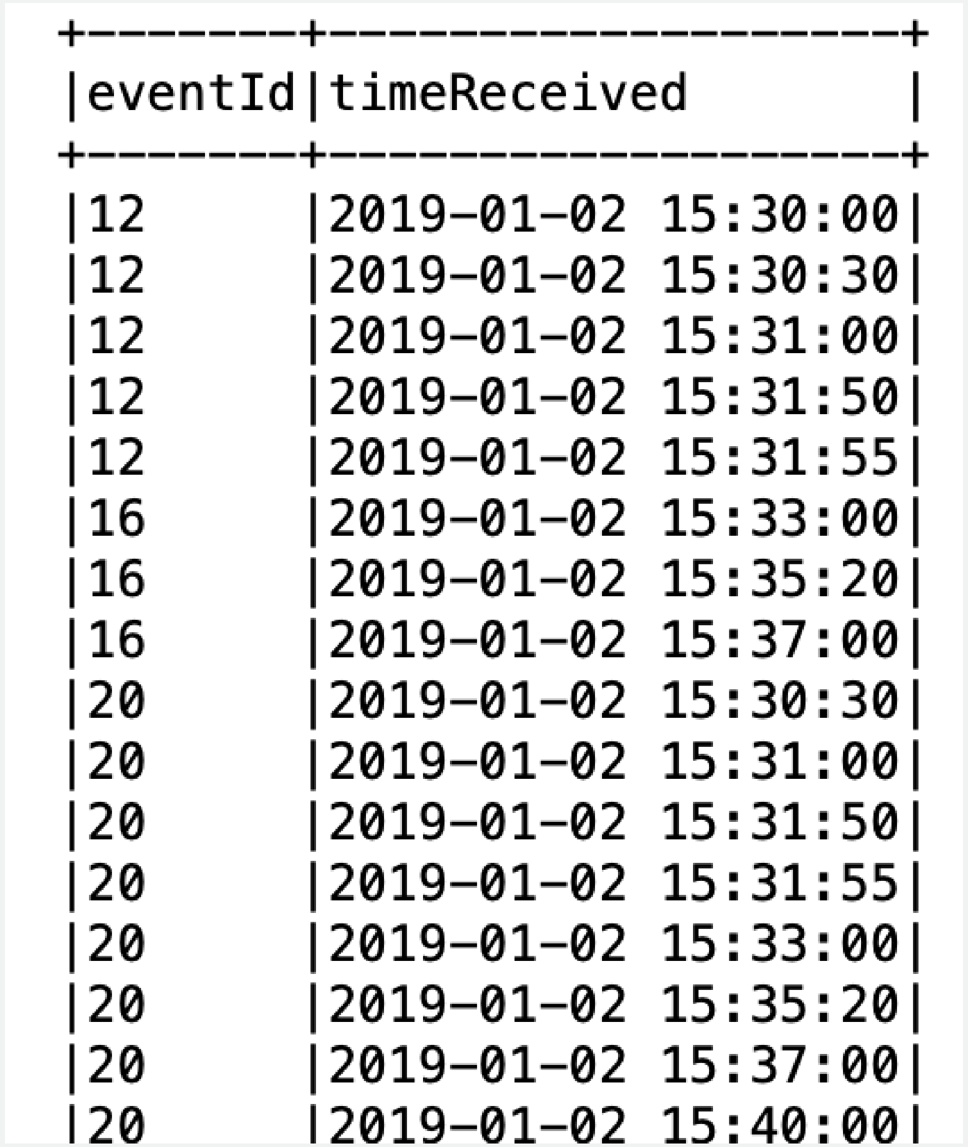
```
activityQuery.stop()
```

```
22/11/02 15:31:55 ERROR TorrentBroadcast: Store broadcast broadcast_1527 fail, remove all pieces of the broadcast
```

Window CODE

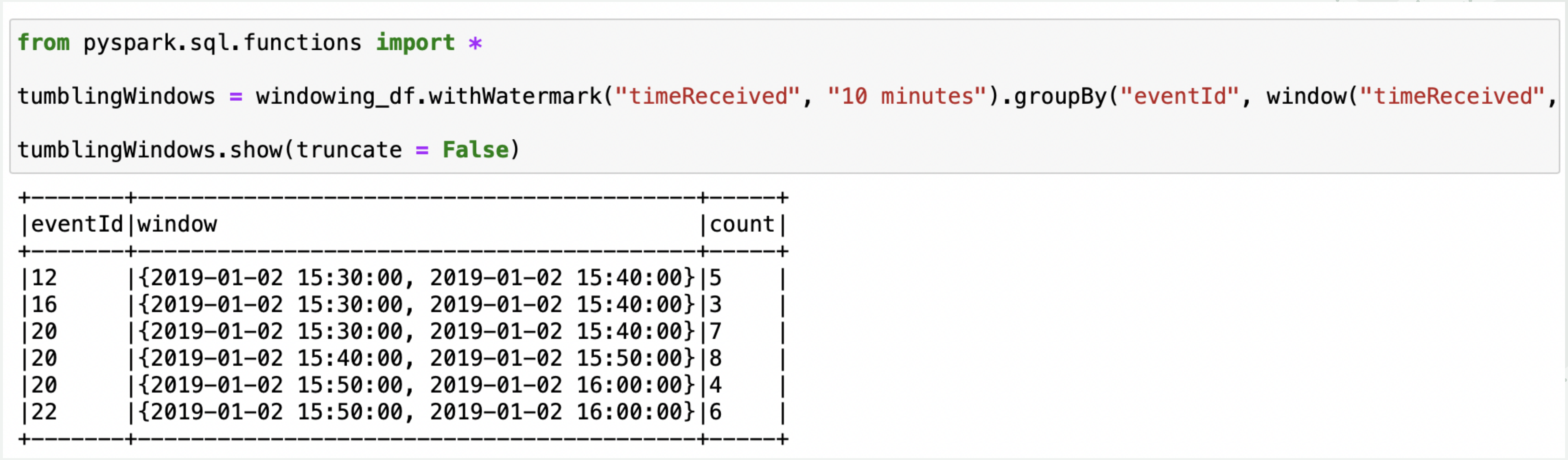
Window DATA

- •Created a data with column name event id and time received in which event_id is int and timeReceived is in DateTime.
- •We have created this to check behaviour of different windows



```
+-------+-------------------+
|eventId|timeReceived       |
+-------+-------------------+
|12     |2019-01-02 15:30:00|
|12     |2019-01-02 15:30:30|
|12     |2019-01-02 15:31:00|
|12     |2019-01-02 15:31:50|
|12     |2019-01-02 15:31:55|
|16     |2019-01-02 15:33:00|
|16     |2019-01-02 15:35:20|
|16     |2019-01-02 15:37:00|
|20     |2019-01-02 15:30:30|
|20     |2019-01-02 15:31:00|
|20     |2019-01-02 15:31:50|
|20     |2019-01-02 15:31:55|
|20     |2019-01-02 15:33:00|
|20     |2019-01-02 15:35:20|
|20     |2019-01-02 15:37:00|
|20     |2019-01-02 15:40:00|
```

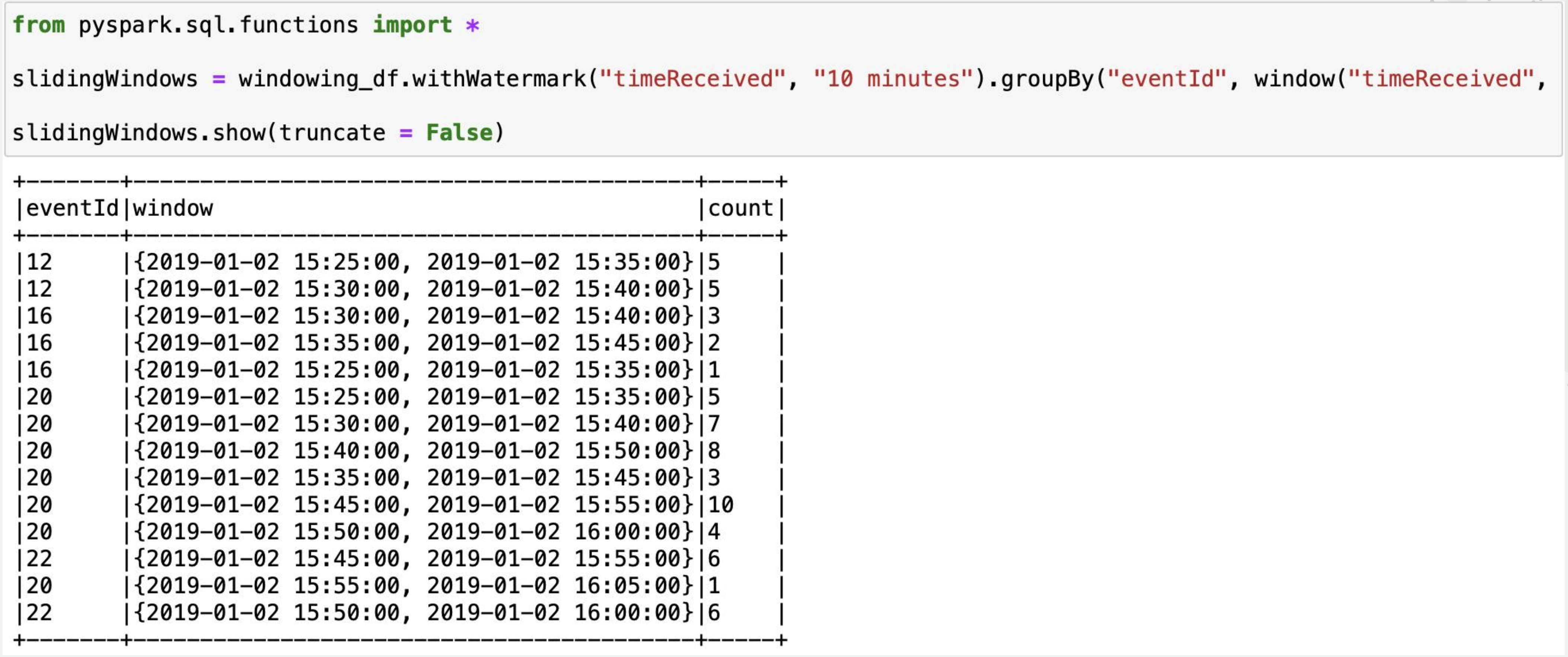
Tumbling window



```
from pyspark.sql.functions import *

tumblingWindows = windowing_df.withWatermark("timeReceived", "10 minutes").groupBy("eventId", window("timeReceived",

tumblingWindows.show(truncate = False)
```

```
+-------+------------------------------------------+-----+
|eventId|window                                    |count|
+-------+------------------------------------------+-----+
|12     |{2019-01-02 15:30:00, 2019-01-02 15:40:00}|5    |
|16     |{2019-01-02 15:30:00, 2019-01-02 15:40:00}|3    |
|20     |{2019-01-02 15:30:00, 2019-01-02 15:40:00}|7    |
|20     |{2019-01-02 15:40:00, 2019-01-02 15:50:00}|8    |
|20     |{2019-01-02 15:50:00, 2019-01-02 16:00:00}|4    |
|22     |{2019-01-02 15:50:00, 2019-01-02 16:00:00}|6    |
+-------+------------------------------------------+-----+
```

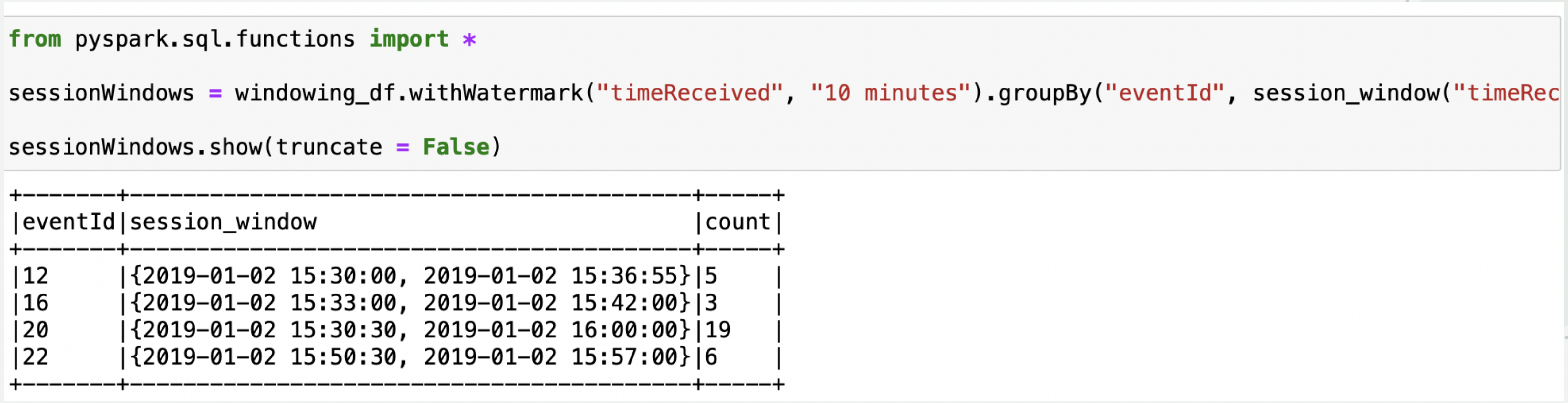
Sliding window



```
from pyspark.sql.functions import *

slidingWindows = windowing_df.withWatermark("timeReceived", "10 minutes").groupBy("eventId", window("timeReceived",

slidingWindows.show(truncate = False)
```

```
+-------+------------------------------------------+-----+
|eventId|window                                    |count|
+-------+------------------------------------------+-----+
|12     |{2019-01-02 15:25:00, 2019-01-02 15:35:00}|5    |
|12     |{2019-01-02 15:30:00, 2019-01-02 15:40:00}|5    |
|16     |{2019-01-02 15:30:00, 2019-01-02 15:40:00}|3    |
|16     |{2019-01-02 15:35:00, 2019-01-02 15:45:00}|2    |
|16     |{2019-01-02 15:25:00, 2019-01-02 15:35:00}|1    |
|20     |{2019-01-02 15:25:00, 2019-01-02 15:35:00}|5    |
|20     |{2019-01-02 15:30:00, 2019-01-02 15:40:00}|7    |
|20     |{2019-01-02 15:40:00, 2019-01-02 15:50:00}|8    |
|20     |{2019-01-02 15:35:00, 2019-01-02 15:45:00}|3    |
|20     |{2019-01-02 15:45:00, 2019-01-02 15:55:00}|10   |
|20     |{2019-01-02 15:50:00, 2019-01-02 16:00:00}|4    |
|22     |{2019-01-02 15:45:00, 2019-01-02 15:55:00}|6    |
|20     |{2019-01-02 15:55:00, 2019-01-02 16:05:00}|1    |
|22     |{2019-01-02 15:50:00, 2019-01-02 16:00:00}|6    |
+-------+------------------------------------------+-----+
```

Session window



```
from pyspark.sql.functions import *

sessionWindows = windowing_df.withWatermark("timeReceived", "10 minutes").groupBy("eventId", session_window("timeRec

sessionWindows.show(truncate = False)
```

```
+-------+------------------------------------------+-----+
|eventId|session_window                            |count|
+-------+------------------------------------------+-----+
|12     |{2019-01-02 15:30:00, 2019-01-02 15:36:55}|5    |
|16     |{2019-01-02 15:33:00, 2019-01-02 15:42:00}|3    |
|20     |{2019-01-02 15:30:30, 2019-01-02 16:00:00}|19   |
|22     |{2019-01-02 15:50:30, 2019-01-02 15:57:00}|6    |
+-------+------------------------------------------+-----+
```

Conclusion

- So in this project we have converted Static data into streaming data using the streaming function and then applying SQL through various aggregate functions to perform aggregation analytics.
- In this project we have converted time date column into a streaming event using different types of windows such as fixed, sliding , session.

Future Scope

- In Spark we can performing Data Analysis and later on for future prospects we can use machine learning algorithms to make this data streaming process more fool proof when it comes to Fraudulent Transaction Detection of Credit cards which is the basis of data we have used in data analysis.

REFERENCES

- https://www.kaggle.com/code/llabhishekll/fraud-transaction-detection/data
- www.infoworld.com
- www.careerride.com
- spark.apache.org
- www.confluent.io
- https://pandas.pydata.org/
- https://learn.microsoft.com/en-us/sql/t-sql/functions/aggregate-functions-transact-sql?view=sql-server-ver16
- https://docs.python.org/3/tutorial/index.html