

# lab3

May 16, 2024

## 1 Stream processing with Spark

Rémi Pépin, Arthur Katosky, Ludovic Deneuville

### 1.1 Before you start

- ☐ Download this Jupyter Notebook
- ☐ Follow these instructions

### 1.2 Outline

This tutorial will teach you the basic of **stream processing with Spark**. As soon as an application compute something with business value (for instance customer activity), and new inputs arrive continuously, companies will want to compute this result continuously too. Spark makes possible to process stream with the **Structured Streaming API**. This lab will teach you the basics of this Spark's API. Because the Structured Streaming API is based on the DataFrame API most syntaxes of tutorial 1 are still relevant.

### 1.3 1 Create a Spark session

- ☐ Depending on the **chosen platform**, initialize the Spark session

#### 1.3.1 1.1 Only on SSPCloud

See default configuration on the datalab :

```
[1]: ! cat /opt/spark/conf/spark-defaults.conf
```

```
spark.driver.extraJavaOptions -Dcom.amazonaws.sdk.disableCertChecking=false
-Dhttp.nonProxyHosts=localhost -Dhttps.nonProxyHosts=localhost
spark.executor.extraJavaOptions -Dcom.amazonaws.sdk.disableCertChecking=false
-Dhttp.nonProxyHosts=localhost -Dhttps.nonProxyHosts=localhost
spark.kubernetes.authenticate.driver.serviceAccountName jupyter-pyspark-456456
spark.kubernetes.container.image inseeifrlab/onyxia-jupyter-
pyspark:py3.12.2-spark3.5.1
spark.kubernetes.driver.pod.name jupyter-pyspark-456456-0
spark.kubernetes.namespace user-yann223
spark.master k8s://https://kubernetes.default.svc:443
spark.driver.memory 2g
spark.dynamicAllocation.enabled true
```

```

spark.dynamicAllocation.executorAllocationRatio 1
spark.dynamicAllocation.initialExecutors 1
spark.dynamicAllocation.maxExecutors 10
spark.dynamicAllocation.minExecutors 1
spark.dynamicAllocation.shuffleTracking.enabled true
spark.executor.memory 2g
spark.hadoop.fs.s3a.bucket.all.committer.magic.enabled true

```

To modify the config :

```

[2]: import os
from pyspark.sql import SparkSession

spark = (SparkSession
        .builder
        # default url of the internally accessed Kubernetes API
        # (This Jupyter notebook service is itself a Kubernetes Pod)
        .master("k8s://https://kubernetes.default.svc:443")
        # Executors spark docker image: for simplicity reasons, this jupyter_
↳notebook is reused
        .config("spark.kubernetes.container.image", os.environ['IMAGE_NAME'])
        # Name of the Kubernetes namespace
        .config("spark.kubernetes.namespace", os.
↳environ['KUBERNETES_NAMESPACE'])
        # Allocated memory to the JVM
        # Stay careful, by default, the Kubernetes pods has a higher limit_
↳which depends on other parameters.
        .config("spark.executor.memory", "4g")
        .config("spark.kubernetes.driver.pod.name", os.
↳environ['KUBERNETES_POD_NAME'])
        # dynamic allocation configuration
        .config("spark.dynamicAllocation.enabled", "true")
        .config("spark.dynamicAllocation.initialExecutors", "1")
        .config("spark.dynamicAllocation.minExecutors", "1")
        .config("spark.dynamicAllocation.maxExecutors", "5")
        .getOrCreate()
        )

```

```

[3]: # See the current number of executors (one for now)
!kubectl get pods -l spark-role=executor

```

NAME	READY	STATUS	RESTARTS	AGE
pyspark-shell-2060788f7c34cc98-exec-1	1/1	Running	0	9s

### 1.3.2 1.2 Only on AWS

```
[4]: #Spark session
spark

# Configuraion
spark._jsc.hadoopConfiguration().set("fs.s3.useRequesterPaysHeader","true")
```

### 1.3.3 1.3 Check spark session

```
[4]: spark
```

```
[4]: <pyspark.sql.session.SparkSession at 0x7f0b4745df10>
```

#### Explanation:

- `spark._jsc.hadoopConfiguration().set("fs.s3.useRequesterPaysHeader","true")`  
likes in lab2, you will be charged for the data transfer. without this configuration you can't access the data.
- `spark.conf.set("spark.sql.shuffle.partitions", 5)` : set the number of partitions for the shuffle phase. A partition is in Spark the name of a bloc of data. By default Spark use 200 partitions to shuffle data. But in this lab, our mini-batch will be small, and to many partitions will lead to performance issues.

```
spark.conf.set("spark.sql.shuffle.partitions", 5)
```

The shuffle dispatches data according to their key between a *map* and a *reduce* phase. For instance, if you are counting how many records have each *g* group, the *map* phase involve counting each group member in each Spark partition : {g1:5, g2:10, g4:1, g5:3} for one partition, {g1:1, g2:2, g3:23, g5:12} for another. The *shuffle* phase dispatch those first results and group them by key in the same partition, one partition gets {g1:5, g1:1, g2:10, g2:2}, the other gets : {g4:1, g5:3, g3:23, g5:12} Then each *reduce* can be done efficiently.

#### Import all required libraries.

```
[16]: from time import sleep
from pyspark.sql.functions import from_json, window, col, expr, size, explode,
    ↪count, countDistinct, approx_count_distinct, min, max, avg, first, last,
    ↪sum, sumDistinct
from pyspark.sql.types import StructType, StructField, StringType, IntegerType,
    ↪ArrayType, TimestampType, BooleanType, LongType, DoubleType
```

## 1.4 2 Stream processing

Stream processing is the act to process data in real-time. When a new record is available, it is processed. There is no real beginning nor end to the process, and there is no “result”. The result is updated in real time, hence multiple versions of the results exist. For instance, you want to count how many tweet about cat are posted in twitter every hour. Until the end of an hour, you do not

have you final result. And even at this moment, your result can change. Maybe some technical problems created some latency and you will get some tweets later. And you will need to update your previous count.

Some commons use cases of stream processing are :

- **Notifications and alerting** : real-time bank fraud detection ; electric grid monitoring with smart meters ; medical monitoring with smart meters, etc.
- **Real time reporting**: traffic in a website updated every minute; impact of a publicity campaign ; stock option portfolio, etc.
- **Incremental ELT (extract transform load)**: new unstructured data are always available and they need to be processed (cleaned, filtered, put in a structured format) before their integration in the company IT system.
- **Online machine learning** : new data are always available and used by a ML algorithm to improve its performance dynamically.

Unfortunately, stream processing has some issues. First because there is no end to the process, you cannot keep all the data in memory. Second, process a chain of event can be complex. How do you raise an alert when you receive the value 5, 6 and 3 consecutively ? Don't forget you are in a distributed environment, and there is latency. Hence, the received order can be different from the emitted order.

## 1.5 3 Spark and stream processing

Stream processing was gradually incorporated in Spark. In 2012 Spark Streaming and its DStreams API was added to Spark (it was before an external project). This made it possible use high-level operator like **map** and **reduce** to process stream of data. Because of its implementation, this API has some limitations, and its syntax was different from the DataFrame one. Thus, in 2016 a new API was added, the Structured Streaming API. This API is directly build built on DataFrame, unlike DStreams. **This has an advantage, you can process your stream like static data.** Of course there are some limitations, but the core syntaxes is the same. You will chain transformations, because each transformation takes a DataFrame as input and produces a DataFrame as output. The big change is there is no action at the end, but an **output sink**.

data stream

*Figure 1 : data stream representation (source [structured streaming programming guide](#))*

Spark offer two ways to process stream, one **record at a time**, or processing **micro batching** (processing a small amount of line at once).

- **one record at a time** every time a new record is available it's processed. This has a big advantage, it achieves **very low latency** . But there is a drawback, the system can not handle too much data at the same time (low throughput). It's the default mode. Because in this lab, you will process files with record, even if you will process one file at a time, you will process mini batch of records
- as for **micro batching** it process new records every  $t$  seconds. Hence records are not process really in "real-time", but periodically, **the latency will be higher, and so the throughput**. Unless you really need low latency, make it you first choice option.

To get the best ratio latency/throughput, a good practice is to decrease the micro-batch size until the mini-batch throughput is the same as the input throughput. Then

increase the size to have some margin

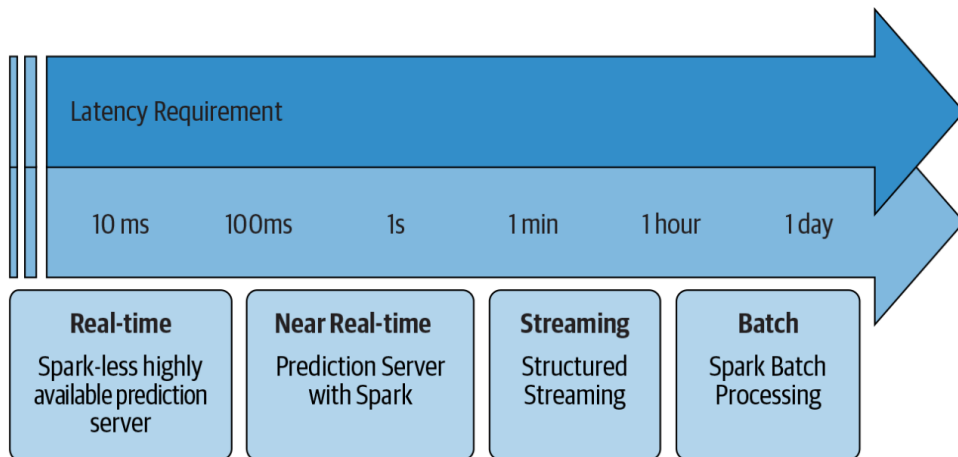


Figure 2 : which Spark solution suit best giving latency requirement (source : [Learning Spark, O'Reilly](#))

To understand why processing one record at a time has lower latency and throughput than batch processing, imagine a restaurant. Every time a client order something the chef cooks its order independently of the other current orders. So if two clients order pizza, the chief makes two small doughs, and cook them individually. If clients there is only a few clients, the chief can finish each order before a new client comes. The latency is the lowest possible when the chief is idle when a client come. Now imagine a restaurant where the chief process the orders by batch. He waits some minutes to gather all the orders then he mutualizes the cooking. If there are 5 pizza orders, he only does one big dough, divides it in five, add the toppings then cook all five at once. The latency is higher because the chief waits before cooking, but so the throughput because he can cook multiple things at once.

## 1.6 4 The basics of Spark's Structured Streaming

### 1.6.1 4.1 The different sources for stream processing in Spark

In lab 2 you discovered Spark DataFrame, in this lab you will learn about [Structured Streaming](#). It's a stream processing framework built on the Spark SQL engine, and it uses the existing structured APIs in Spark. So once you define a way to read a stream, you will get a DataFrame. Like in lab2 ! **So except state otherwise, all transformations presented in lab2 are still relevant in this lab.**

Spark Streaming supports several input source for reading in a streaming fashion :

- Apache Kafka an open-source distributed event streaming platform (not show in this lab)
- Files on distributed file system like HDFS or S3 (Spark will continuously read new files in a directory)
- A network socket : an end-point in a communication across a network (sort of very simple webservice). It's not recommend for *production* application, because a socket connection doesn't provide any mechanism to check the consistency of data.

Defining an input source is like loading a DataFrame but, you have to replace `spark.read` by

`spark.readStream`. For instance, if I want to open a stream to a folder located in S3 you have to read every new files put in it, just write

```
my_first_stream = spark\
.readStream\
.schema(schema_tweet)\
.json("s3://my-awesome-bucket/my-awesome-folder")
```

The major difference with lab2, it is Spark cannot infer the schema of the stream. You have to pass it to Spark. There is two ways :

- A reliable way : you define the schema by yourself and gave it to Spark
- A quick way : you load one file of the folder in a DataFrame, extract the schema and use it. It works, but the schema can be incomplete. It's a better solution to create the schema by hand and use it.

For Apache Kafka, or socket , it's a slightly more complex, *(not used today, it's jute for you personal knowledge)* :

```
my_first_stream = spark\
.readStream\
.format("kafka")
.option("kafka.bootstrap.servers", "host1:port1, host2:port2 etc")
.option("subscribePattern", "topic name")
.load()
```

**Why is a folder a relevant source in stream processing ?** Previously, in lab 1, you loaded all the files in a folder stored in MinIO with Spark. Powered by Kubernetes, MinIO delivers scalable, secure, S3 compatible object storage to every public cloud. And it worked pretty well. But this folder was static, in other words, Its content didn't change. But in some cases, new data are constantly written into a folder. For instance, in this lab you will process a stream of tweets. A python script is running in a VS Code service reading tweets from the Twitter's web service and writing them in a S3 buckets. Every 2 seconds or so, a new file is added to the bucket with 1000 tweets. If you use DataFrame like in lab 1, your process cannot proceed those new files. You should relaunch your process every time. But with Structured Streaming Spark will dynamically load new files.

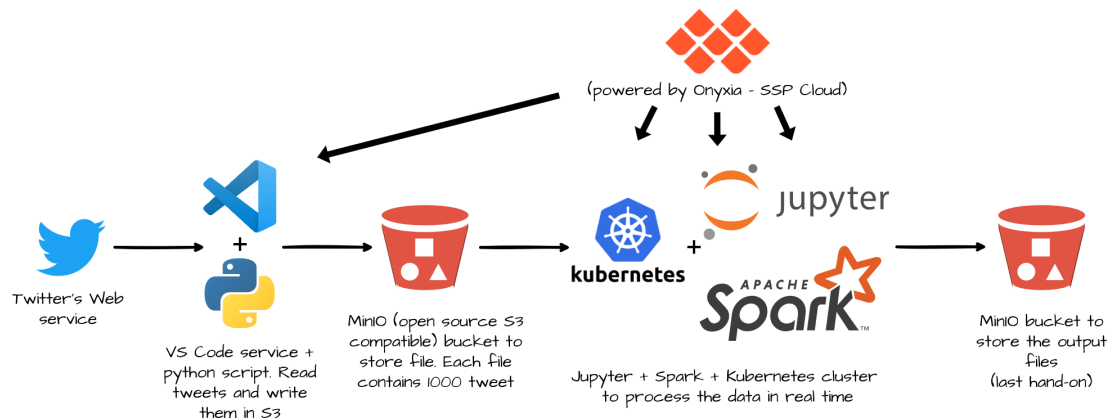


Figure 3 : Complete lab architecture to stream process tweets

The remaining question is, why don't we connect Spark to the twitter webservice directly ? And the answer is : we can't. Spark cannot be connected to a webservice directly. You need a middle-man between Spark and a webservice. There are multiple solutions, but an easy and reliable one is to write tweet to MinIO, the open implementation of s3 (because we use Onyxia, if you use Microsoft Azure, Google Cloud Platform or OVH cloud replace S3 by their storage service).

### 1.6.2 4.2 Hand-on 1 : Open a stream

Like in lab 1, you will use tweets in this lab. The tweets are stored in jsonl file (*json line* every line of the file is a complete json). Here is an example. The schema changed a little, because this time tweets aren't pre-processed.

```
{
  "data": {
    "public_metrics": {
      "retweet_count": 0,
      "reply_count": 0,
      "like_count": 0,
      "quote_count": 0
    },
    "text": "Day 93. Tweeting every day until Colby cheez its come back #bringcolbyback @colbycheez",
    "possibly_sensitive": false,
    "created_at": "2021-05-03T07:55:46.000Z",
    "id": "1389126523853148162",
    "entities": {
      "annotations": [
        {
          "start": 33,
          "end": 43,
          "probability": 0.5895,
          "type": "Person",
          "normalized_text": "Colby cheez"
        }
      ],
      "mentions": [
        {
          "start": 75,
          "end": 83,
          "username": "cheezit"
        }
      ],
      "hashtags": [
        {
          "start": 59,
          "end": 74,
          "tag": "bringcolbyback"
        }
      ]
    }
  }
}
```

```

        }
    ]
},
"lang": "en",
"source": "Twitter for iPhone",
"author_id": "606856313"
},
"includes": {
    "users": [
        {
            "created_at": "2012-06-13T03:36:00.000Z",
            "username": "DivinedHavoc",
            "verified": false,
            "name": "Justin",
            "id": "606856313"
        }
    ]
}
}
}

```

- Define a variable with this schema (you will find a file *schema pyspark tweet* on moodle with the schema to copy / paste)

```
from pyspark.sql.types import StructType, StructField, StringType, IntegerType, ArrayType,
```

```

StructType([
    StructField("data", StructType([
        StructField("author_id", StringType(), True),
        StructField("text", StringType(), True),
        StructField("source", StringType(), True),
        StructField("lang", StringType(), True),
        StructField("created_at", TimestampType(), True),
        StructField("entities", StructType([
            StructField("annotations", ArrayType(StructType([
                StructField("end", LongType(), True),
                StructField("normalized_text", StringType(), True),
                StructField("probability", DoubleType(), True),
                StructField("start", LongType(), True),
                StructField("type", StringType(), True)
            ]), True), True),
            StructField("cashtags", ArrayType(StructType([
                StructField("end", LongType(), True),
                StructField("start", LongType(), True),
                StructField("tag", StringType(), True)
            ]), True), True),
            StructField("hashtags", ArrayType(StructType([
                StructField("end", LongType(), True),
                StructField("start", LongType(), True),
                StructField("tag", StringType(), True)
            ]), True), True),

```



```

    ]),True),True),
    StructField("mentions", ArrayType(StructType([
        StructField("end", LongType(), True),
        StructField("start", LongType(), True),
        StructField("username", StringType(), True)
    ]),True),True),
    StructField("urls", ArrayType(StructType([
        StructField("description", StringType(), True),
        StructField("display_url", StringType(), True),
        StructField("end", LongType(), True),
        StructField("expanded_url", StringType(), True),
        StructField("images", ArrayType(StructType([
            StructField("height", LongType(), True),
            StructField("url", StringType(), True),
            StructField("width", LongType(), True)
        ]),True),True),
        StructField("start", LongType(), True),
        StructField("status", LongType(), True),
        StructField("title", StringType(), True),
        StructField("unwound_url", StringType(), True),
        StructField("url", StringType(), True),
    ]),True),True),
    ]),True),
    StructField("public_metrics", StructType([
        StructField("like_count", LongType(), True),
        StructField("reply_count", LongType(), True),
        StructField("retweet_count", LongType(), True),
        StructField("quote_count", LongType(), True),
    ]),True),
    ]),True),
    StructField("includes", StructType([
        StructField("users", ArrayType(StructType([
            StructField("created_at", TimestampType(), True),
            StructField("id", StringType(), True),
            StructField("name", StringType(), True),
            StructField("username", StringType(), True),
            StructField("verified", BooleanType(), True)
        ]),True),True)
    ]),True)
])

```

```

[8]: from pyspark.sql.types import StructType, StructField, StringType, IntegerType, \
    ↪ ArrayType, TimestampType, BooleanType, LongType, DoubleType

schema_tweet=StructType([
    StructField("data", StructType([
        StructField("author_id", StringType(), True),

```

```

StructField("text",StringType(),True),
StructField("source",StringType(),True),
StructField("lang",StringType(),True),
StructField("created_at",TimestampType(),True),
StructField("entities",StructType([
    StructField("annotations", ArrayType(StructType([
        StructField("end", LongType(), True),
        StructField("normalized_text", StringType(), True),
        StructField("probability", DoubleType(), True),
        StructField("start", LongType(), True),
        StructField("type", StringType(), True)
    ]),True),True),
    StructField("cashtags", ArrayType(StructType([
        StructField("end", LongType(), True),
        StructField("start", LongType(), True),
        StructField("tag", StringType(), True)
    ]),True),True),
    StructField("hashtags", ArrayType(StructType([
        StructField("end", LongType(), True),
        StructField("start", LongType(), True),
        StructField("tag", StringType(), True)
    ]),True),True),
    StructField("mentions", ArrayType(StructType([
        StructField("end", LongType(), True),
        StructField("start", LongType(), True),
        StructField("username", StringType(), True)
    ]),True),True),
    StructField("urls", ArrayType(StructType([
        StructField("description", StringType(), True),
        StructField("display_url", StringType(), True),
        StructField("end", LongType(), True),
        StructField("expanded_url", StringType(), True),
        StructField("images", ArrayType(StructType([
            StructField("height", LongType(), True),
            StructField("url", StringType(), True),
            StructField("width", LongType(), True)
        ]),True),True),
        StructField("start", LongType(), True),
        StructField("status", LongType(), True),
        StructField("title", StringType(), True),
        StructField("unwound_url", StringType(), True),
        StructField("url", StringType(), True),
    ]),True),True),
    StructField("public_metrics", StructType([
        StructField("like_count", LongType(), True),
        StructField("reply_count", LongType(), True),

```

```

        StructField("retweet_count", LongType(), True),
        StructField("quote_count", LongType(), True),
    ]), True)
    ]), True),
    StructField("includes", StructType([
        StructField("users", ArrayType(StructType([
            StructField("created_at", TimestampType(), True),
            StructField("id", StringType(), True),
            StructField("name", StringType(), True),
            StructField("username", StringType(), True),
            StructField("verified", BooleanType(), True)
        ]), True), True)
    ]), True)
])

```

- Create a stream to this s3 bucket :
  - s3a://ludo2ne/diffusion/ensai/stream\_tweet/stream for SSPCloud
  - s3://ensai-labs-2023-2024-files/lab3/destination/ for AWS
  - Name it `tweet_stream`
  - Use the option `option("maxFilePerTrigger", "1")` to process each new files one by one

```

[9]: tweet_stream = spark\
    .readStream\
    .schema(schema_tweet)\
    .json("s3a://ludo2ne/diffusion/ensai/stream_tweet/stream")

```

Nothing happen ? It's normal ! Do not forget, Spark use lazy evaluation. It doesn't use data if you don't define an action. For now Spark only know how to get the stream, that's all.

- In a cell just execute `tweet_stream`.
  - It should print the type of `tweet_stream` and the associated schema. You can see you created a DataFrame like in lab2 !

```

[12]: tweet_stream

```

```

[12]: DataFrame[data: struct<author_id:string,text:string,source:string,lang:string,created_at:timestamp,entities:struct<annotations:array<struct<end:bigint,normalized_text:string,probability:double,start:bigint,type:string>>,cashtags:array<struct<end:bigint,start:bigint,tag:string>>,hashtags:array<struct<end:bigint,start:bigint,tag:string>>,mentions:array<struct<end:bigint,start:bigint,username:string>>,urls:array<struct<description:string,display_url:string,end:bigint,expanded_url:string,images:array<struct<height:bigint,url:string,width:bigint>>,start:bigint,status:bigint,title:string,unwound_url:string,url:string>>>,public_metrics:struct<like_count:bigint,reply_count:bigint,retweet_count:bigint,quote_count:bigint>>, includes: struct<users:array<struct<created_at:timestamp,id:string,name:string,username:string,verified:boolean>>>]

```

- Print the size of your DataFrame by using this piece of code :

```
stream_size_query= tweet_stream\
.writeStream\
.queryName("stream_size")\
.format("memory")\
.start()

for _ in range(10): # we use an _ because the variable isn't used. You can use i if you prefer
    sleep(3)
    spark.sql("""
        SELECT count(1) FROM stream_size
    """).show()
stream_size_query.stop() #needed to close the query
```

```
[17]: stream_size_query= tweet_stream\
.writeStream\
.queryName("stream_size")\
.format("memory")\
.start()

for _ in range(10): # we use an _ because the variable isn't used. You can use i if you prefer
    sleep(3)
    spark.sql("""
        SELECT count(1) FROM stream_size
    """).show()
stream_size_query.stop() #needed to close the query
```

```
+-----+
|count(1)|
+-----+
|      0|
+-----+
```

```
+-----+
|count(1)|
+-----+
|      0|
+-----+
```

```
+-----+
|count(1)|
```

```
+-----+
|      0|
+-----+
```

```
+-----+
|count(1)|
+-----+
|  68000|
+-----+
```

```
+-----+
|count(1)|
+-----+
|  68000|
+-----+
```

```
+-----+
|count(1)|
+-----+
|  79000|
+-----+
```

```
+-----+
|count(1)|
+-----+
|  80000|
+-----+
```

```
+-----+
|count(1)|
+-----+
|  82000|
+-----+
```

```
+-----+
|count(1)|
+-----+
|  83000|
+-----+
```

```
+-----+
|count(1)|
+-----+
|  85000|
+-----+
```

+-----+

### 1.6.3 4.3 How to output a stream ?

Remember, Spark has two types of methods to process DataFrame:

- Transformations which take a DataFrame as input and produce another DataFrame
- And actions, which effectively run computation and produce something, like a file, or an output in your notebook/console.

Stream processing looks the same as DataFrame processing. Hence, **you still have transformations**, the exact same one that can be applied on classic DataFrame (with some restriction, for example you cannot sample a stream with the `sample()` transformation). The action part is a little different. Because a stream runs continuously, you cannot just print the data or run a count at the end of the process. **In fact actions will not work on stream.** To tackle this issue, Spark proposes different **output sinks**. An output sink is a possible output for your stream. The different output sinks are (this part came from the official Spark [documentation](#)) :

- **File sink** - Stores the output to a file. The file can be stored locally (on the cluster), remotely (on S3). The file format can be json, csv etc

```
writeStream\
  .format('json')\
  .option("checkpointLocation", "output_folder/history") \
  .option("path", "output_folder")\
  .start()
```

- **Kafka sink** - Stores the output to one or more topics in Kafka.
- **Foreach sink** - Runs arbitrary computation on the records in the output. It does not produce an DataFrame. Each processed line is lost

```
writeStream
  .foreach(...)
  .start()
```

- **Console sink (for debugging)** - Prints the output to the console standard output (*stdout*) every time there is a trigger. This should be used for debugging purposes on low data volumes as the entire output is collected and stored in the driver's memory after every trigger. *Sadly console sink does not work with jupyter notebook.*

```
writeStream
  .format("console")
  .start()
```

- **Memory sink (for debugging)** - The output is stored in memory as an in-memory table. This should be used for debugging purposes on low data volumes as the entire output is collected and stored in the driver's memory. Hence, use it with caution. Because we are in a simple lab, you will use this solution. But keep in mind it's a very bad idea because data must fit in the RAM of the driver node. And in a big data context it's impossible. Because it's not a big data problem if one computer can tackle it.

```
writeStream
  .format("memory")
  .queryName("tableName") # to resquest the table with spark.sql()
  .start()
```

We just talked where we can output a stream, but there is another question, how ?

To understand why it's a issue, let's talk about two things that spark can do with streams : filter data and group by + aggregation

- **Filter** : your process is really simple. Every time you get a new data you just compute a score and drop records with a score less than a threshold. Then you write into a file every kept record. In a nutshell, you just append new data to a file. Spark does not have to read an already written row, it just add new data.
- **Group by + aggregation** : in this case you want to group by your data by key than compute a simple count. Then you want to write the result in a file. But now there is an issue, Spark needs to update some existing rows in your file every time it writes somethings. But is your file is stored in HDFS or S3, it's impossible to update in a none append way a file. In a nutshell, it's impossible to output in a file your operation.

To deal with this issue, Spark proposes 3 mode. **And you cannot use every mode with every output sink, with every transformation.** The 3 modes are ([more info here](#)) :

- **Append mode (default)** - This is the default mode, where only the new rows added to the Result Table since the last trigger will be outputted to the sink. This is supported for only those queries where rows added to the Result Table is never going to change. Hence, this mode guarantees that each row will be output only once (assuming fault-tolerant sink). For example, queries with only **select**, **where**, **map**, **flatMap**, **filter**, **join**, etc. will support Append mode.
- **Complete mode** - The whole Result Table will be outputted to the sink after every trigger. This is supported for aggregation queries.
- **Update mode** - (*Available since Spark 2.1.1*) Only the rows in the Result Table that were updated since the last trigger will be outputted to the sink. More information to be added in future releases.

Sink	Supported Output Modes
<b>File Sink</b>	Append
<b>Kafka Sink</b>	Append, Update, Complete
<b>Foreach Sink</b>	Append, Update, Complete
<b>ForeachBatch Sink</b>	Append, Update, Complete
<b>Console Sink</b>	Append, Update, Complete
<b>Memory Sink</b>	Append, Complete

#### 1.6.4 4.4 How to output a stream : summary

To sum up to output a stream you need

- DataFrame (because once load a stream is a DataFrame)
- A format for your output, like console to print in console, memory to keep the Result Table in memory, json to write it to a file etc

- A mode to specify how the Result Table will be updated.

For instance for the memory sink

```
memory_sink = df\
.writeStream\
.queryName("my_awesome_name")\
.format('memory')\
.outputMode("complete" or "append")\
.start() #needed to start the stream
```

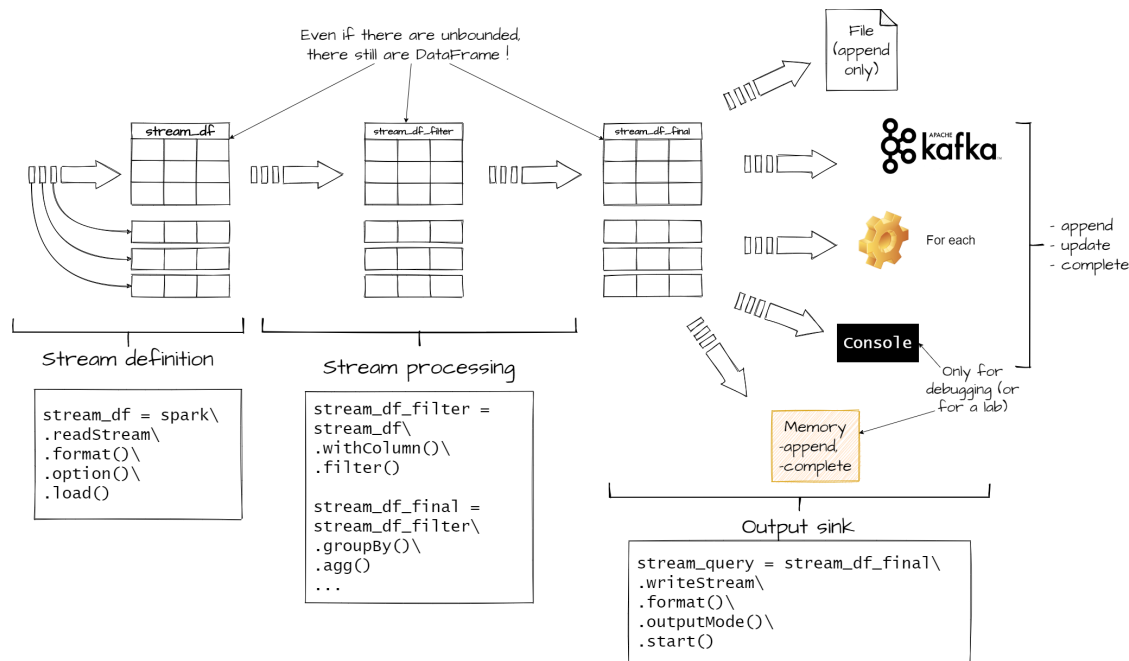


Figure 4 : The different phases of stream processing in Spark

## 1.6.5 4.5 Hand-on 2 : output a stream

### Lang count

- Compute a DataFrame that group and count data by the lang column.
  - Name your DataFrame lang\_count

```
[21]: lang_count = tweet_stream.groupBy("data.lang").count()
```

- Use this DataFrame to create a output stream with the following configuration :
  - Names the variable lang\_query
  - Memory sink
  - Complete mode (because we are doing an agregation)
  - Name you query lang\_count

```
[23]: lang_query = lang_count.writeStream.format('memory').queryName("lang_table").
      ↪outputMode("complete").start()
```

[Stage 106:=====>

(82 + 5) / 200]



□ Then past this code

```
for _ in range(10): # we use an _ because the variable isn't use. You can use i if you prefer
    sleep(3)
    spark.sql("""
        SELECT * FROM lang_count""").show()
    lang_query.stop() #needed to close the stream
```

```
[25]: for _ in range(10): # we use an _ because the variable isn't use. You can use i if you prefer
        ↪if you prefere
            sleep(3)
            spark.sql("""
                SELECT * FROM lang_table""").show()
            lang_query.stop() #needed to close the stream
```

```
+-----+-----+
|lang|count|
+-----+-----+
| art|  523|
|  en|68827|
|  vi|  258|
|  ne|   53|
|  ps|   24|
|  ro|  125|
|  sl|   51|
| und| 3958|
|  ka|    1|
|  ur| 3752|
|  lv|   71|
|  pl|  699|
|  pt| 7605|
|  tl| 2087|
| qht|  331|
|  in| 6126|
|  ko| 9033|
|  uk|  227|
| zxx| 6189|
|  cs|  156|
+-----+-----+
```

only showing top 20 rows

```
+-----+-----+
|lang|count|
+-----+-----+
| art|  525|
```

	en	69444
	vi	259
	ne	54
	ps	25
	ro	126
	sl	52
	und	3990
	ka	1
	ur	3787
	lv	72
	pl	703
	pt	7671
	tl	2103
	qht	332
	in	6170
	ko	9123
	uk	230
	zxx	6248
	cs	158

+-----+-----+

only showing top 20 rows

+-----+-----+

	lang	count
--	------	-------

+-----+-----+

	art	535
	en	70036
	vi	260
	ne	54
	ps	25
	ro	129
	sl	53
	und	4033
	ka	1
	ur	3829
	lv	74
	pl	706
	pt	7741
	tl	2118
	qht	337
	in	6220
	ko	9204
	uk	233
	zxx	6289
	cs	159

+-----+-----+

only showing top 20 rows

```
+-----+-----+
|lang|count|
+-----+-----+
| art|  537|
|  en|70700|
|  vi|  261|
|  ne|   55|
|  ps|   25|
|  ro|  129|
|  sl|   53|
| und| 4062|
|  ka|    1|
|  ur| 3868|
|  lv|   74|
|  pl|  717|
|  pt| 7821|
|  tl| 2127|
| qht|  339|
|  in| 6270|
|  ko| 9283|
|  uk|  235|
| zxx| 6347|
|  cs|  160|
+-----+-----+
```

only showing top 20 rows

```
+-----+-----+
|lang|count|
+-----+-----+
| art|  540|
|  en|71377|
|  vi|  264|
|  ne|   55|
|  ps|   25|
|  ro|  131|
|  sl|   53|
| und| 4093|
|  ka|    1|
|  ur| 3898|
|  lv|   74|
|  pl|  725|
|  pt| 7897|
```

	tl		2147	
	qht		341	
	in		6330	
	ko		9361	
	uk		236	
	zxx		6391	
	cs		162	
+	-----	+		+

only showing top 20 rows

+	-----	+		
	lang		count	
+	-----	+		
	art		540	
	en		71377	
	vi		264	
	ne		55	
	ps		25	
	ro		131	
	sl		53	
	und		4093	
	ka		1	
	ur		3898	
	lv		74	
	pl		725	
	pt		7897	
	tl		2147	
	qht		341	
	in		6330	
	ko		9361	
	uk		236	
	zxx		6391	
	cs		162	
+	-----	+		

only showing top 20 rows

+	-----	+		
	lang		count	
+	-----	+		
	art		544	
	en		72005	
	vi		266	
	ne		55	
	ps		26	

	ro	131
	sl	53
	und	4140
	ka	1
	ur	3934
	lv	74
	pl	728
	pt	7978
	tl	2165
	qht	345
	in	6392
	ko	9433
	uk	239
	zxx	6444
	cs	163
+-----+-----+		

only showing top 20 rows

+-----+-----+		
	lang	count
+-----+-----+		
	art	544
	en	72005
	vi	266
	ne	55
	ps	26
	ro	131
	sl	53
	und	4140
	ka	1
	ur	3934
	lv	74
	pl	728
	pt	7978
	tl	2165
	qht	345
	in	6392
	ko	9433
	uk	239
	zxx	6444
	cs	163
+-----+-----+		

only showing top 20 rows

```

+-----+-----+
|lang|count|
+-----+-----+
| art|  550|
|  en|72664|
|  vi|  268|
|  ne|   56|
|  ps|   26|
|  ro|  133|
|  sl|   53|
| und| 4174|
|  ka|    1|
|  ur| 3975|
|  lv|   74|
|  pl|  740|
|  pt| 8042|
|  tl| 2184|
| qht|  347|
|  in| 6457|
|  ko| 9500|
|  uk|  241|
| zxx| 6505|
|  cs|  167|
+-----+-----+

```

only showing top 20 rows

[Stage 148:=====> (129 + 5) / 200][Stage 149:> (0 + 0) / 1]

```

+-----+-----+
|lang|count|
+-----+-----+
| art|  557|
|  en|73284|
|  vi|  270|
|  ne|   56|
|  ps|   26|
|  ro|  133|
|  sl|   53|
| und| 4211|
|  ka|    1|
|  ur| 4008|
|  lv|   75|
|  pl|  746|
|  pt| 8121|
|  tl| 2208|
| qht|  354|
|  in| 6518|
|  ko| 9573|

```

```
| uk| 243|
| zxx| 6558|
| cs| 168|
```

```
+-----+-----+
```

only showing top 20 rows

```
10:31:58.877 [stream execution thread for lang_table [id =
a19f78a0-1b5b-4745-b697-d8b17ee71434, runId =
3a90d865-9025-4b85-89e6-477666f7eff6]] ERROR
org.apache.spark.sql.execution.datasources.v2.WriteToDataSourceV2Exec - Data
source write support MicroBatchWrite[epoch: 15, writer:
org.apache.spark.sql.execution.streaming.sources.MemoryStreamingWrite@624e6f9b]
is aborting.
10:31:58.887 [stream execution thread for lang_table [id =
a19f78a0-1b5b-4745-b697-d8b17ee71434, runId =
3a90d865-9025-4b85-89e6-477666f7eff6]] ERROR
org.apache.spark.sql.execution.datasources.v2.WriteToDataSourceV2Exec - Data
source write support MicroBatchWrite[epoch: 15, writer:
org.apache.spark.sql.execution.streaming.sources.MemoryStreamingWrite@624e6f9b]
aborted.
```

- After 30 seconds, 10 tables will appeared in your notebook. Each table represents the contain of `lang_count` at a certain time. The `.stop()` method close the stream.

In the rest of this tutorial, you will need two steps to print data :

1. Define a stream query with a memory sink
2. Request this stream with the `spark.sql()` function

Instead of a for loop, you can just write you `spark.sql()` statement in a cell and rerun it. In this case you will need a third cell with a `stop()` method to close your stream.

For instance:

- Cell 1

```
my_query = my_df\
    .writeStream\
    .format("memory")\
    .queryName("query_table")\
    .start()
```

- Cell 2

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

- Cell 3

```
my_query.stop()
```

Count tweets with and without hashtag

- Add a column `has_hashtag` to your DataFrame.
  - This column equals True if `data.entities.hashtags` is not null. Else it's false.
  - Use the `withColumn` transformation to add a column.
  - You can count the size of `data.entities.hashtags` to check if it's empty or not.

```
[34]: tweet_hashtags = tweet_stream.withColumn("has_hashtags", size(tweet_stream.data.
      ↳ entities.hashtags)>0)
```

- Group and count by the `has_hashtag` column

```
[35]: tweet_hashtags_count = tweet_hashtags.groupBy("has_hashtags").count()
```

- Print some results

```
[36]: tweet_hashtags_query = tweet_hashtags_count.writeStream.format('memory').
      ↳ queryName("hashtags").outputMode("complete").start()

for _ in range(10): # we use an _ because the variable isn't use. You can use i_
      ↳ if you prefere
    sleep(3)
    spark.sql("""
      SELECT * FROM hashtags""").show()
tweet_hashtags_query.stop() #needed to close the stream
```

```
[Stage 155:=====> (3 + 2) / 5]
```

```
+-----+-----+
|has_hashtags|count|
+-----+-----+
+-----+-----+
```

```
[Stage 156:=====> (148 + 5) / 200]
```

```
+-----+-----+
|has_hashtags|count|
+-----+-----+
+-----+-----+
```

```
+-----+-----+
|has_hashtags|count|
+-----+-----+
|      true|10544|
|      false|45456|
+-----+-----+
```



has_hashtags	count
true	11120
false	47880

has_hashtags	count
true	11542
false	49458

has_hashtags	count
true	11919
false	51081

has_hashtags	count
true	12293
false	52707

has_hashtags	count
true	12647
false	54353

has_hashtags	count
--------------	-------

```
|      true|13240|
|      false|56760|
+-----+-----+
```

```
[Stage 186:=====>(187 + 4) / 200][Stage 187:> (0 + 1) / 1]
```

```
+-----+-----+
|has_hashtags|count|
+-----+-----+
|      true|13421|
|      false|57579|
+-----+-----+
```

```
10:58:20.146 [stream execution thread for hashtags [id =
1078892e-b606-4fe0-927a-b7ad5ebceefc, runId =
dccbe0a9-e774-48d0-ae2a-0e9ee8ceccb2]] ERROR
org.apache.spark.sql.execution.datasources.v2.WriteToDataSourceV2Exec - Data
source write support MicroBatchWrite[epoch: 8, writer:
org.apache.spark.sql.execution.streaming.sources.MemoryStreamingWrite@431c6917]
is aborting.
10:58:20.147 [stream execution thread for hashtags [id =
1078892e-b606-4fe0-927a-b7ad5ebceefc, runId =
dccbe0a9-e774-48d0-ae2a-0e9ee8ceccb2]] ERROR
org.apache.spark.sql.execution.datasources.v2.WriteToDataSourceV2Exec - Data
source write support MicroBatchWrite[epoch: 8, writer:
org.apache.spark.sql.execution.streaming.sources.MemoryStreamingWrite@431c6917]
aborted.
```

## 1.6.6 4.6 Debugging tip

If at any moment of this lab you encounter an error like this one :

```
'Cannot start query with name has_hashtag as a query with that name is already active'
Traceback (most recent call last):
```

```
File "/usr/lib/spark/python/lib/pyspark.zip/pyspark/sql/streaming.py", line 1109, in start
    return self._sq(self._jwrite.start())
File "/usr/lib/spark/python/lib/py4j-0.10.7-src.zip/py4j/java_gateway.py", line 1257, in __c
    answer, self.gateway_client, self.target_id, self.name)
File "/usr/lib/spark/python/lib/pyspark.zip/pyspark/sql/utils.py", line 79, in deco
    raise IllegalArgumentException(s.split(':', 1)[1], stackTrace)
pyspark.sql.utils.IllegalArgumentException: 'Cannot start query with name has_hashtag as a query
```

Run in a cell the following code :

```
for stream in spark.streams.active:
    stream.stop()
```

spark.streams.active returns an array with all the active stream, and the code loops over all the

active stream and closes them.

## 1.7 5 Stream processing basics

### 1.7.1 5.1 Hand-on 3 : transformations on stream

- Filter all records with missing / null value then count how many records you keep
  - For this filter, you will use the `na.drop("any")` transformation. The `na.drop("any")` drop every line with a null value in at least one column. It's simpler than using a `filter()` transformation because you don't have to specify all the column. For more precise filter you can use `na.drop("any" or "all", subset=list of col)` (all will drop rows with only null value in all columns or in the specified list).
  - Use the SQL `COUNT(1)` function in the sql request to get the count
  - Because you don't perform aggregation the `outputMode()` must be `append`

You will notice no record are dropped.

```
[37]: # trans_stream = tweet_stream.dropna()

trans_stream = tweet_stream.na.drop("any")

trans_query = trans_stream.writeStream.format('memory').queryName("trans").
    ↪outputMode("append").start()

for _ in range(10): # we use an _ because the variable isn't use. You can use i_
    ↪if you prefere
        sleep(3)
        spark.sql("""
            SELECT COUNT(1) FROM trans""").show()
trans_query.stop() #needed to close the stream
```

```
+-----+
|count(1)|
+-----+
|      0|
+-----+
```

```
+-----+
|count(1)|
+-----+
|      0|
+-----+
```

```
+-----+
|count(1)|
+-----+
|      0|
+-----+
```

```
+-----+
|count(1)|
+-----+
|  132000|
+-----+
```

```
+-----+
|count(1)|
+-----+
|  134000|
+-----+
```

```
+-----+
|count(1)|
+-----+
|  135000|
+-----+
```

```
+-----+
|count(1)|
+-----+
|  137000|
+-----+
```

```
+-----+
|count(1)|
+-----+
|  139000|
+-----+
```

```
+-----+
|count(1)|
+-----+
|  141000|
+-----+
```

```
+-----+
|count(1)|
+-----+
|  143000|
+-----+
```

- Drop all records with unverified (`includes.users.verified == True`) user then group the remaining records by `hashtag`.
  - `includes.users` is an array with only one element. You will need to extract it.
  - `data.entities.hashtags` is an array too ! To group by tag (the hashtag content) you will need to explode it too.

```
[78]: trans_stream_bis = trans_stream\
      .withColumn("verified", explode("includes.users.verified"))\
      .filter("verified==True")\
      .withColumn("tags", explode("data.entities.hashtags"))\
      .groupBy("tags.tag").count()
```

- Find ukraine related tweet (or any other topic like cat, dog, spring, batman, dogecoin etc) :
  - Define a new column, name `ukraine_related`. This column is equal to `True` if `data.text` contains “ukraine”, else it’s equal to `False`.
  - Use the `withColumn()` transformation, and the `expr()` function to define the column. `expr()` takes as input an SQL expression. You do not need a full SQL statement (`SELECT ... FROM ... WHERE ...`) but just an SQL expression that return `True` or `False` if `data.text` contains “ukraine”. To help you :
    - \* `LOWER()` put in lower case a string
    - \* `input_string LIKE wanted_string` return `True` if `input_string` is equal to `wanted_string`
    - \* You can use `%` as wildcards [For more help](#)
  - Only keep `data.text`, `data.lang`, `data.public_metrics` and `data.created_at`

```
[57]: trans_stream_ter = trans_stream\
      .withColumn("ukraine_related", expr("LOWER(data.text) LIKE '%ukraine%'"))\
      .select("data.text", "data.lang", "data.public_metrics", "data.created_at")
```

## 1.7.2 5.2 Hand-on 4 : Aggregation and grouping on stream

- Count the number of different hashtag.

```
[79]: bis_query = trans_stream_bis.writeStream.format("memory").queryName("tags").
      ↪outputMode("complete").start()

for _ in range(6): # we use an _ because the variable isn't use. You can use i_
    ↪if you prefere
    sleep(3)
    spark.sql("""
        SELECT * FROM tags""").show()
bis_query.stop() #needed to close the stream
```

[Stage 462:>

(0 + 5) / 5]

```
+---+-----+
|tag|count|
+---+-----+
```

+---+-----+

[Stage 462:=====>

(4 + 1) / 5]

+---+-----+

|tag|count|

+---+-----+

+---+-----+

[Stage 463:=====>

(169 + 5) / 200]

+---+-----+

|tag|count|

+---+-----+

+---+-----+

+-----+-----+

|tag|count|

+-----+-----+

|cliqueart| 1|

|InterMilan| 3|

|MilliGörüşMeclise| 2|

| | 1|

| | 1|

|masturbating| 1|

|KongthupProduction| 1|

|Lavrov| 1|

|Bangalore| 2|

|EnDirecto| 1|

|türkiye| 1|

|MeninasDaGávea| 1|

|CSIX| 1|

|SAFEMOON| 5|

|youthsudeepians| 1|

|overcooked2| 1|

|ASTRO| 15|

| | 1|

|beckyangels| 2|

|KabirisGod| 1|

+-----+-----+

only showing top 20 rows

+-----+-----+

|tag|count|

+-----+	
	EnDirecto  1
	türkiye  1
	youthsudeepians  1
	CoffeeTime  1
	MilliGörüşMeclise  2
	masturbating  1
	oland  1
	Bangalore  2
	CSIX  1
	overcooked2  1
	ASTRO  15
	1
	beckyangels  2
	AutomotiveIndustry  1
	PAW  1
	cliqueart  1
	1
	MeninasDaGávea  1
	InterMilan  3
	1
+-----+	

only showing top 20 rows

[Stage 471:=====> (175 + 5) / 200][Stage 472:>

(0 + 0) / 1]

+-----+	
	tag count
+-----+	
	EnDirecto  1
	türkiye  1
	youthsudeepians  1
	CoffeeTime  1
	MilliGörüşMeclise  2
	masturbating  1
	oland  1
	Bangalore  2
	CSIX  1
	overcooked2  1
	ASTRO  15
	1
	beckyangels  2
	AutomotiveIndustry  1
	PAW  1
	cliqueart  1
	1
	MeninasDaGávea  1
	InterMilan  3

```
|          |    1|
+-----+-----+
only showing top 20 rows
```

```
12:32:31.121 [stream execution thread for tags [id =
28336244-1a9b-4f96-a8f7-c07eb5d4dbef, runId =
a78667ec-d8ac-4b40-a9eb-f7e9ad224fd0]] ERROR
org.apache.spark.sql.execution.datasources.v2.WriteToDataSourceV2Exec - Data
source write support MicroBatchWrite[epoch: 3, writer:
org.apache.spark.sql.execution.streaming.sources.MemoryStreamingWrite@6575935a]
is aborting.
12:32:31.121 [stream execution thread for tags [id =
28336244-1a9b-4f96-a8f7-c07eb5d4dbef, runId =
a78667ec-d8ac-4b40-a9eb-f7e9ad224fd0]] ERROR
org.apache.spark.sql.execution.datasources.v2.WriteToDataSourceV2Exec - Data
source write support MicroBatchWrite[epoch: 3, writer:
org.apache.spark.sql.execution.streaming.sources.MemoryStreamingWrite@6575935a]
aborted.
```

- Group by hashtag and compute the average, min and max of like\_count
  - Use the `groupBy()` and `agg()` transformations

```
[69]: like_stream = trans_stream\
      .withColumn("verified",explode("includes.users.verified"))\
      .filter("verified==True")\
      .withColumn("tags",explode("data.entities.hashtags"))\
      .groupBy("tags")\
      .agg(max("data.public_metrics.like_count").alias("max_like"),
           min("data.public_metrics.like_count").alias("min_like"),
           avg("data.public_metrics.like_count").alias("avg_like"))
```

```
[75]: like_query = like_stream.writeStream.format("memory").queryName("likes").
      ↪outputMode("complete").start()

for _ in range(6): # we use an _ because the variable isn't use. You can use i_
      ↪if you prefere
      sleep(3)
      spark.sql("""
        SELECT * FROM likes ORDER BY max_like DESC""").show()
like_query.stop() #needed to close the stream
```

[Stage 447:=====> (40 + 5) / 200]

```
+---+-----+-----+-----+
|tags|max_like|min_like|avg_like|
+---+-----+-----+-----+
+---+-----+-----+-----+
```



	tags	max_like	min_like	avg_like
	{230, 212, Dinast...	1	1	1.0
	{21, 0, RenewSupe...	1	1	1.0
	{244, 231, MarceT...	1	1	1.0
	{69, 66, ad}	1	1	1.0
	{27, 19, Bazooka}	1	1	1.0
	{38, 22, Superman...	1	1	1.0
	{257, 242, Juntos...	1	1	1.0
	{142, 132, Mammoo...	1	1	1.0
	{127, 117, ThePla...	0	0	0.0
	{27, 21, ASTRO}	0	0	0.0
	{26, 18, }	0	0.0	
	{26, 19, KARINA}	0	0	0.0
	{123, 120, }	0	0	0.0
	{69, 58, Healthcare}	0	0	0.0
	{291, 284, BSCGEM}	0	0	0.0
	{55, 39,  ... _	0	0	0.0
99} , 90, }		0	0.0	
	{89, 70, Olympien...	0	0	0.0
	{70, 63, SpaceX}	0	0	0.0
	{42, 36, Diaby}	0	0	0.0

only showing top 20 rows

	tags	max_like	min_like	avg_like
	{21, 0, RenewSupe...	1	1	1.0
	{27, 19, Bazooka}	1	1	1.0
	{244, 231, MarceT...	1	1	1.0
	{257, 242, Juntos...	1	1	1.0
	{230, 212, Dinast...	1	1	1.0
	{142, 132, Mammoo...	1	1	1.0
	{38, 22, Superman...	1	1	1.0
	{69, 66, ad}	1	1	1.0
	{55, 39,  ... _	0	0	0.0
	{27, 21, ASTRO}	0	0	0.0
	{179, 172, findom}	0	0	0.0
	{42, 36, Diaby}	0	0	0.0
	{52, 35, TOP100KP...	0	0	0.0
	{69, 58, Healthcare}	0	0	0.0

{159, 138, Nothin...	0	0	0.0
{26, 19, KARINA}	0	0	0.0
{49, 43, }	0	0	0.0
{89, 70, Olympien...	0	0	0.0
{32, 16,  ... _	0	0	0.0
{52, 47, Mark}	0	0	0.0

only showing top 20 rows

tags	max_like	min_like	avg_like
{21, 0, RenewSupe...	1	1	1.0
{276, 267, DAOVERSE}	1	1	1.0
{27, 19, Bazooka}	1	1	1.0
{244, 231, MarceT...	1	1	1.0
{230, 212, Dinast...	1	1	1.0
{38, 22, Superman...	1	1	1.0
{257, 242, Juntos...	1	1	1.0
{9, 0, _ }	1	1	1.0
{142, 132, Mammoo...	1	1	1.0
{69, 66, ad}	1	1	1.0
{213, 206, kernow}	0	0	0.0
{136, 120, 18Year...	0	0	0.0
{27, 21, ASTRO}	0	0	0.0
{106, 87, YSLBEAU...	0	0	0.0
{69, 58, Healthcare}	0	0	0.0
{79, 74, EEUU}	0	0	0.0
{89, 70, Olympien...	0	0	0.0
{245, 233, blauer...	0	0	0.0
{131, 125, NuNew}	0	0	0.0
{159, 138, Nothin...	0	0	0.0

only showing top 20 rows

tags	max_like	min_like	avg_like
{21, 0, RenewSupe...	1	1	1.0
{276, 267, DAOVERSE}	1	1	1.0
{27, 19, Bazooka}	1	1	1.0
{244, 231, MarceT...	1	1	1.0
{230, 212, Dinast...	1	1	1.0
{38, 22, Superman...	1	1	1.0

{257, 242, Juntos...	1	1	1.0
{9, 0, _ }	1	1	1.0
{142, 132, Mammoo...	1	1	1.0
{69, 66, ad}	1	1	1.0
{213, 206, kernow}	0	0	0.0
{159, 138, Nothin...	0	0	0.0
{27, 21, ASTRO}	0	0	0.0
{28, 14, AzucenaA...	0	0	0.0
{69, 58, Healthcare}	0	0	0.0
{32, 16,  ... _	0	0	0.0
{89, 70, Olympien...	0	0	0.0
{65, 56, shingeki}	0	0	0.0
{131, 125, NuNew}	0	0	0.0
{179, 172, findom}	0	0	0.0

only showing top 20 rows

[Stage 459:=====> (168 + 5) / 200][Stage 460:>

(0 + 0) / 5]

	tags	max_like	min_like	avg_like
{21, 0, RenewSupe...	1	1	1.0	
{276, 267, DAOVERSE}	1	1	1.0	
{27, 19, Bazooka}	1	1	1.0	
{244, 231, MarceT...	1	1	1.0	
{230, 212, Dinast...	1	1	1.0	
{38, 22, Superman...	1	1	1.0	
{257, 242, Juntos...	1	1	1.0	
{9, 0, _ }	1	1	1.0	
{142, 132, Mammoo...	1	1	1.0	
{69, 66, ad}	1	1	1.0	
{96, 86, fingering}	0	0	0.0	
{104, 88, youths...	0	0	0.0	
{279, 274, POJK}	0	0	0.0	
{99, 96, _ }	0	0	0.0	
{131, 122, BORN...	0	0	0.0	
{21, 17, _ }	0	0	0.0	
{223, 214, Cornwall}	0	0	0.0	
{29, 16, RT ...	0	0	0.0	
{272, 259, ShibeS...	0	0	0.0	
{55, 39,  ... _	0	0	0.0	

only showing top 20 rows

12:28:59.743 [stream execution thread for likes [id = 44ae25f3-397d-4888-a4a8-467788c9a079, runId = 674875dc-3cb3-407b-a025-e0523591f598]] ERROR

```
org.apache.spark.sql.execution.datasources.v2.WriteToDataSourceV2Exec - Data
source write support MicroBatchWrite[epoch: 4, writer:
org.apache.spark.sql.execution.streaming.sources.MemoryStreamingWrite@79c7e8d0]
is aborting.
12:28:59.744 [stream execution thread for likes [id =
44ae25f3-397d-4888-a4a8-467788c9a079, runId =
674875dc-3cb3-407b-a025-e0523591f598]] ERROR
org.apache.spark.sql.execution.datasources.v2.WriteToDataSourceV2Exec - Data
source write support MicroBatchWrite[epoch: 4, writer:
org.apache.spark.sql.execution.streaming.sources.MemoryStreamingWrite@79c7e8d0]
aborted.
```

□ Compute the average of `like_count`, `retweet_count` and `quote_count` :

- across all `hashtag` and `lang`
- for each `lang` across all `hashtag`
- for each `hashtag` across all `lang`
- for each `hashtag` and each `lang`

To do so, replace the `groupBy()` transformation by the `cube()` one. `cube()` group compute all possible cross between dimensions passed as parameter. You will get something like this

hashtag	lang	avg(like_count)	avg(retweet_count)	avg(quote_count)
cat	null	1	2	3
dog	null	4	5	6
...	...	...	...	...
bird	fr	7	8	9
null	en	10	11	12
null	null	13	14	15

A `null` value mean this dimension wasn't use for this row. For instance, the first row gives the averages when `hashtag==cat` independently of the `lang`. The before last row gives averages when `lang==en` independently of the `hashtag`. And the last row gives the averages for the full `DataFrame`.

```
[86]: count_stream = trans_stream\
      .withColumn("verified",explode("includes.users.verified"))\
      .filter("verified==True")\
      .withColumn("tags",explode("data.entities.hashtags"))\
      .cube("tags.tag","data.lang")\
      .agg(max("data.public_metrics.retweet_count").alias("avg_retweet"),
           avg("data.public_metrics.quote_count").alias("avg_quote"),
           avg("data.public_metrics.like_count").alias("avg_like"))

# count_stream = trans_stream\
#   .withColumn("verified",explode("includes.users.verified"))\
#   .filter("verified==True")\
```

```

# .withColumn("tags", explode("data.entities.hashtags"))\
# .cube("data.lang")\
# .agg(max("data.public_metrics.retweet_count").alias("avg_retweet"),
#      avg("data.public_metrics.quote_count").alias("avg_quote"),
#      avg("data.public_metrics.like_count").alias("avg_like"))

# count_stream = trans_stream\
# .withColumn("verified", explode("includes.users.verified"))\
# .filter("verified==True")\
# .withColumn("tags", explode("data.entities.hashtags"))\
# .cube("tags.tag")\
# .agg(max("data.public_metrics.retweet_count").alias("avg_retweet"),
#      avg("data.public_metrics.quote_count").alias("avg_quote"),
#      avg("data.public_metrics.like_count").alias("avg_like"))

# count_stream = trans_stream\
# .withColumn("verified", explode("includes.users.verified"))\
# .filter("verified==True")\
# .withColumn("tags", explode("data.entities.hashtags"))\
# .agg(max("data.public_metrics.retweet_count").alias("avg_retweet"),
#      avg("data.public_metrics.quote_count").alias("avg_quote"),
#      avg("data.public_metrics.like_count").alias("avg_like"))

count_query = count_stream.writeStream.format("memory").queryName("count").
    ↪outputMode("complete").start()

for _ in range(6): # we use an _ because the variable isn't use. You can use i_
    ↪if you prefere
        sleep(3)
        spark.sql("""
            SELECT * FROM count """).show()
count_query.stop() #needed to close the stream

```

```

+---+---+-----+-----+-----+
|tag|lang|avg_retweet|avg_quote|avg_like|
+---+---+-----+-----+-----+
+---+---+-----+-----+-----+

```

[Stage 549:=====>

(4 + 1) / 5]

```

+---+---+-----+-----+-----+
|tag|lang|avg_retweet|avg_quote|avg_like|
+---+---+-----+-----+-----+
+---+---+-----+-----+-----+

```

[Stage 550:=====>

(91 + 4) / 200]

```
+---+---+-----+-----+-----+
|tag|lang|avg_retweet|avg_quote|avg_like|
+---+---+-----+-----+-----+
+---+---+-----+-----+-----+
```

[Stage 550:=====>(199 + 1) / 200]

```
+---+---+-----+-----+-----+
|tag|lang|avg_retweet|avg_quote|avg_like|
+---+---+-----+-----+-----+
+---+---+-----+-----+-----+
```

```
+---+---+-----+-----+-----+
|tag|lang|avg_retweet|avg_quote|avg_like|
+---+---+-----+-----+-----+
+---+---+-----+-----+-----+
```

[Stage 551:=====>

(3 + 1) / 4]

```
+-----+---+-----+-----+-----+
|          tag|lang|avg_retweet|avg_quote|avg_like|
+-----+---+-----+-----+-----+
|          | ja|      1260|      0.0|      0.0|
|          GlazersOut|NULL|      0|      0.0|      0.0|
| Amministrative2023| it|      6|      0.0|      0.0|
|          24 | ja|      0|      0.0|      0.0|
|          natasitt|NULL|     44|      0.0|      0.0|
|PakistanUnderFascism|NULL|    4562|      0.0|      0.0|
|          onlyfan| zh|    3897|      0.0|      0.0|
|          JaiShriRam| en|     198|      0.0|      0.0|
|          BSC| qme|      0|      0.0|      0.0|
|          TOTN|NULL|     10|      0.0|      0.0|
|          CommunistHell|NULL|     42|      0.0|      0.0|
|          Camels|NULL|      0|      0.0|      0.0|
|          KicchaSudeep| kn|     33|      0.0|      0.0|
| TheWorstIsYetToCome| en|      0|      0.0|      0.0|
|          _ | hi|      46|      0.0|      0.0|
|          FPL|NULL|      9|      0.0|      0.0|
|          |NULL|     340|      0.0|      0.0|
|          |NULL|      1|      0.0|      0.0|
|          Texcoco|NULL|    1867|      0.0|      0.0|
|          COYBIG| en|      9|      0.0|      0.0|
+-----+---+-----+-----+-----+
```

only showing top 20 rows

13:50:30.328 [stream execution thread for count [id =

```

699d65de-72dc-4322-8834-fcf62c06cbd2, runId =
9745b62a-99c0-4af2-b3ca-59f99bd1bfbb]] ERROR
org.apache.spark.sql.execution.datasources.v2.WriteToDataSourceV2Exec - Data
source write support MicroBatchWrite[epoch: 1, writer:
org.apache.spark.sql.execution.streaming.sources.MemoryStreamingWrite@5eb28a44]
is aborting.
13:50:30.328 [stream execution thread for count [id =
699d65de-72dc-4322-8834-fcf62c06cbd2, runId =
9745b62a-99c0-4af2-b3ca-59f99bd1bfbb]] ERROR
org.apache.spark.sql.execution.datasources.v2.WriteToDataSourceV2Exec - Data
source write support MicroBatchWrite[epoch: 1, writer:
org.apache.spark.sql.execution.streaming.sources.MemoryStreamingWrite@5eb28a44]
aborted.

```

## 1.8 6 Event-time processing

Event-time processing consists in processing information with **respect to the time that it was created, not received**. It's a hot topic because sometime you will receive data in an order different from the creation order. For example, you are monitoring servers distributed across the globe. Your main datacentre is located in Paris. Something append in New York, and a few milliseconds after something append in Toulouse. Due to location, the event in Toulouse is likely to show up in your datacentre before the New York one. If you analyse data bases on the received time the order will be different than the event time. Computers and network are unreliable. Hence, when temporality is important, you must consider the creation time of the event and not it's received time.

Hopefully, Spark will handle all this complexity for you ! If you have a timestamp column with the event creation spark can update data accordingly to the event time.

For instance is you process some data with a time window, Spark will update the result based on the event-time not the received time. So previous windows can be updated in the future.

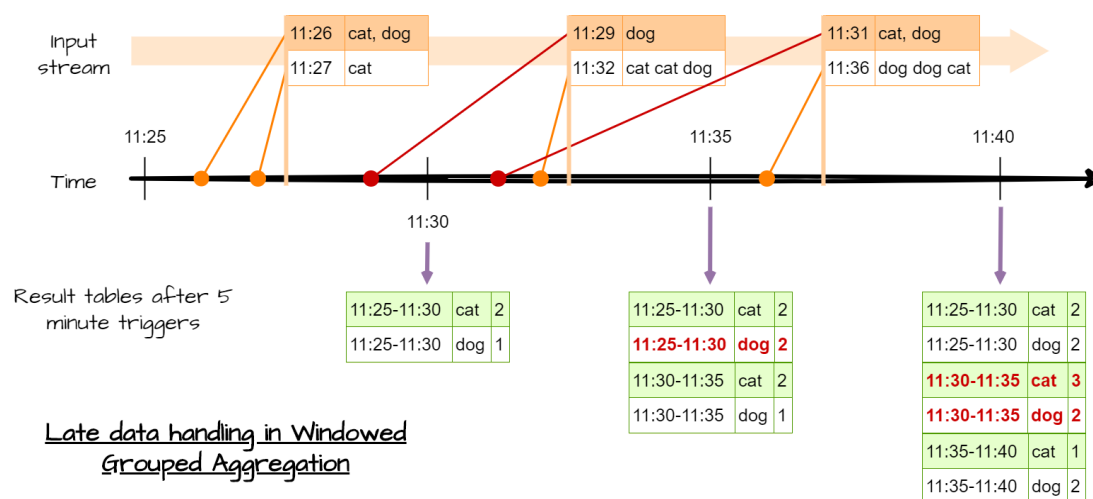


Figure 5 : Time-event processing, event grouped by time windows

To work with time windows, Spark offers two type of windows

- Normal windows. You only consider event in a given windows. All windows are disjoint, and a event is only in one window.
- Sliding windows. You have a fix window size (for example 1 hour) and a trigger time (for example 10 minute). Every 10 minute, you will process the data with an event time less than 1h.

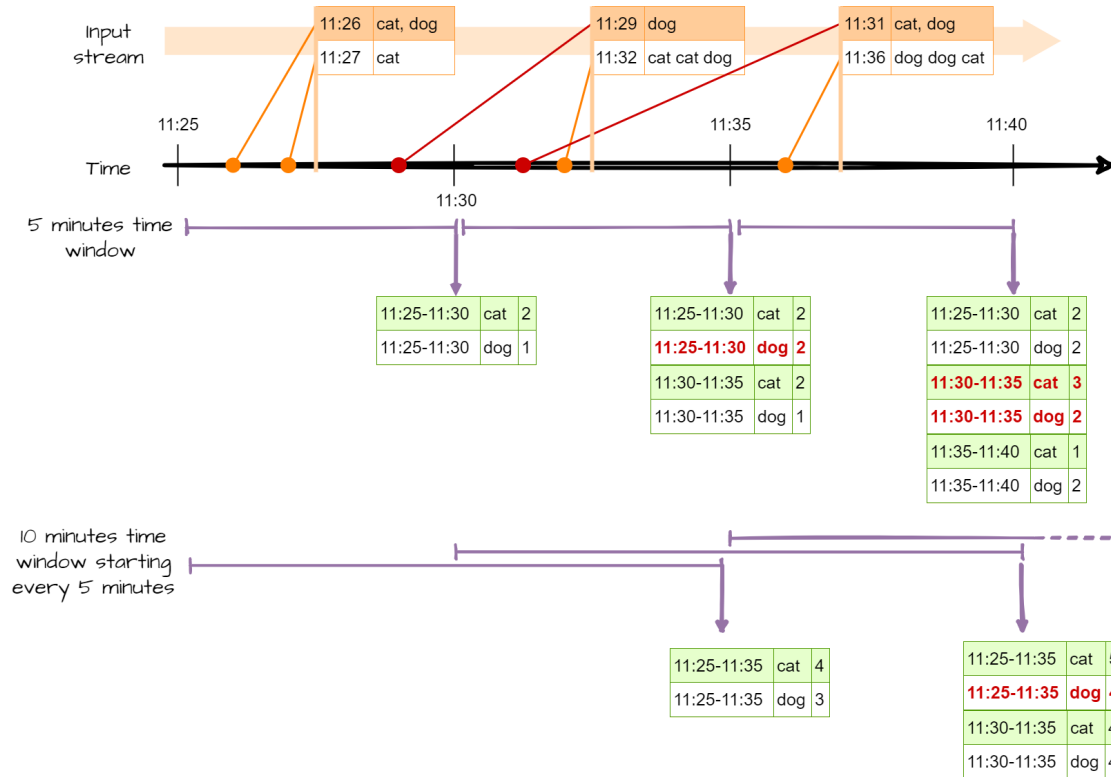


Figure 6 : Time-event processing, event grouped by sliding time windows

To create time windows, you need :

- to define a time window : `window(column_with_time_event : str or col, your_time_window : str, timer_for_sliding_window) : str`
- grouping row by event-time using your window : `df.groupBy(window(...))`

To produce the above processes :

*# Need some import*

```
from pyspark.sql.functions import window, col
```

*# word count + classic time window*

```
df_with_event_time.groupBy(
    window(df_with_event_time.event_time, "5 minutes"),
    df_with_event_time.word).count()
```

*# word count + sliding time window*

```
df_with_event_time.groupBy(
    window(df_with_event_time.event_time, "10 minutes", "5 minutes"),
    df_with_event_time.word).count()
```



### 1.8.1 6.1 Hand-on 5 : Event-time processing

- Count the number of event with a 10 seconds time window (use the `created_at` column)

```
[87]: sliding_windows_tweet_count = tweet_stream\
      .groupByKey(window(tweet_stream.data.created_at, "10 seconds"))\
      .count()

sliding_windows_tweet_count_query = sliding_windows_tweet_count\
      .writeStream\
      .queryName("sliding_windows_tweet_count")\
      .format('memory')\
      .outputMode("complete")\
      .start()

for _ in range(10):
    spark.sql("""
        SELECT *
        FROM sliding_windows_tweet_count """).show()
    sleep(3)

sliding_windows_tweet_count_query.stop()
```

```
+-----+-----+
|window|count|
+-----+-----+
+-----+-----+
```

[Stage 554:=====> (201 + 4) / 238]

```
+-----+-----+
|window|count|
+-----+-----+
+-----+-----+
```

[Stage 555:> (0 + 4) / 8]

```
+-----+-----+
|window|count|
+-----+-----+
+-----+-----+
```

[Stage 555:=====> (7 + 1) / 8]

```
+-----+-----+
|window|count|
+-----+-----+
+-----+-----+
```

[Stage 556:=====>

(112 + 5) / 200]

```
+-----+-----+
|window|count|
+-----+-----+
+-----+-----+
```

```
+-----+-----+
|                window|count|
+-----+-----+
|{2023-05-12 16:39...| 606|
|{2023-05-12 16:08...| 610|
|{2023-05-12 16:54...|   4|
|{2023-05-12 15:46...| 558|
|{2023-05-12 16:26...| 560|
|{2023-05-12 15:55...| 582|
|{2023-05-12 16:43...| 547|
|{2023-05-12 15:48...| 559|
|{2023-05-12 15:47...| 572|
|{2023-05-12 15:58...| 611|
|{2023-05-12 16:08...| 599|
|{2023-05-12 15:49...| 582|
|{2023-05-12 16:02...| 566|
|{2023-05-12 15:50...| 567|
|{2023-05-12 16:38...| 562|
|{2023-05-12 15:56...| 546|
|{2023-05-12 16:34...| 566|
|{2023-05-12 16:52...| 582|
|{2023-05-12 16:50...| 554|
|{2023-05-12 16:47...| 592|
```

```
+-----+-----+
only showing top 20 rows
```

```
+-----+-----+
|                window|count|
+-----+-----+
|{2023-05-12 16:39...| 606|
|{2023-05-12 16:08...| 610|
|{2023-05-12 16:54...|   4|
|{2023-05-12 15:46...| 558|
|{2023-05-12 16:26...| 560|
|{2023-05-12 15:55...| 582|
|{2023-05-12 16:43...| 547|
```

{2023-05-12 15:48...	559
{2023-05-12 15:47...	572
{2023-05-12 15:58...	611
{2023-05-12 16:08...	599
{2023-05-12 15:49...	582
{2023-05-12 16:02...	566
{2023-05-12 15:50...	567
{2023-05-12 16:38...	562
{2023-05-12 15:56...	546
{2023-05-12 16:34...	566
{2023-05-12 16:52...	582
{2023-05-12 16:50...	554
{2023-05-12 16:47...	592

+-----+-----+

only showing top 20 rows

	window count
--	--------------

+-----+-----+

{2023-05-12 16:39...	606
{2023-05-12 16:55...	572
{2023-05-12 16:08...	610
{2023-05-12 16:54...	542
{2023-05-12 15:46...	558
{2023-05-12 16:26...	560
{2023-05-12 15:55...	582
{2023-05-12 16:43...	547
{2023-05-12 15:48...	559
{2023-05-12 15:47...	572
{2023-05-12 15:58...	611
{2023-05-12 16:08...	599
{2023-05-12 15:49...	582
{2023-05-12 16:02...	566
{2023-05-12 15:50...	567
{2023-05-12 16:38...	562
{2023-05-12 16:34...	566
{2023-05-12 15:56...	546
{2023-05-12 16:52...	582
{2023-05-12 16:50...	554

+-----+-----+

only showing top 20 rows

	window count
--	--------------

```

+-----+-----+
|{2023-05-12 16:39...| 606|
|{2023-05-12 16:55...| 572|
|{2023-05-12 16:08...| 610|
|{2023-05-12 16:54...| 542|
|{2023-05-12 15:46...| 558|
|{2023-05-12 16:26...| 560|
|{2023-05-12 15:55...| 582|
|{2023-05-12 16:43...| 547|
|{2023-05-12 15:48...| 559|
|{2023-05-12 15:47...| 572|
|{2023-05-12 15:58...| 611|
|{2023-05-12 16:08...| 599|
|{2023-05-12 15:49...| 582|
|{2023-05-12 16:02...| 566|
|{2023-05-12 15:50...| 567|
|{2023-05-12 16:38...| 562|
|{2023-05-12 16:34...| 566|
|{2023-05-12 15:56...| 546|
|{2023-05-12 16:52...| 582|
|{2023-05-12 16:50...| 554|
+-----+-----+
only showing top 20 rows

```

```

+-----+-----+
|                window|count|
+-----+-----+
|{2023-05-12 16:39...| 606|
|{2023-05-12 16:55...| 572|
|{2023-05-12 16:08...| 610|
|{2023-05-12 16:54...| 542|
|{2023-05-12 15:46...| 558|
|{2023-05-12 16:26...| 560|
|{2023-05-12 15:55...| 582|
|{2023-05-12 16:43...| 547|
|{2023-05-12 15:48...| 559|
|{2023-05-12 15:47...| 572|
|{2023-05-12 15:58...| 611|
|{2023-05-12 16:08...| 599|
|{2023-05-12 15:49...| 582|
|{2023-05-12 16:02...| 566|
|{2023-05-12 15:50...| 567|
|{2023-05-12 16:38...| 562|
|{2023-05-12 16:34...| 566|
|{2023-05-12 15:56...| 546|
|{2023-05-12 16:52...| 582|

```

```
|{2023-05-12 16:50...| 554|
+-----+-----+-----+
only showing top 20 rows
```

```
13:54:15.727 [stream execution thread for sliding_windows_tweet_count [id =
9a8b7e5e-8c47-4a1f-9230-42098ab104c9, runId =
c62bfa97-fbce-4508-946b-e0fcd2a4fa9c]] ERROR
org.apache.spark.sql.execution.datasources.v2.WriteToDataSourceV2Exec - Data
source write support MicroBatchWrite[epoch: 4, writer:
org.apache.spark.sql.execution.streaming.sources.MemoryStreamingWrite@6ef2d464]
is aborting.
13:54:15.727 [stream execution thread for sliding_windows_tweet_count [id =
9a8b7e5e-8c47-4a1f-9230-42098ab104c9, runId =
c62bfa97-fbce-4508-946b-e0fcd2a4fa9c]] ERROR
org.apache.spark.sql.execution.datasources.v2.WriteToDataSourceV2Exec - Data
source write support MicroBatchWrite[epoch: 4, writer:
org.apache.spark.sql.execution.streaming.sources.MemoryStreamingWrite@6ef2d464]
aborted.
```

- ☐ Count the number of event by verified / unverified user with a 10 seconds time window (use the Creation\_Time column)

```
[89]: sliding_windows_tweet_count = tweet_stream\
      .withColumn("verified", explode("includes.users.verified"))\
      .groupBy(window(tweet_stream.data.created_at, "10 seconds"), "verified")\
      .count()

sliding_windows_tweet_count_query = sliding_windows_tweet_count\
      .writeStream\
      .queryName("sliding_windows_tweet_count")\
      .format('memory')\
      .outputMode("complete")\
      .start()

for _ in range(10):
    spark.sql("""
        SELECT *
        FROM sliding_windows_tweet_count
        ORDER BY window """).show(truncate=False)
    sleep(3)

sliding_windows_tweet_count_query.stop()
```

```
+-----+-----+-----+
|window|verified|count|
+-----+-----+-----+
```

+-----+-----+-----+

[Stage 574:=====>

(68 + 5) / 200]

+-----+-----+-----+

|window|verified|count|

+-----+-----+-----+

+-----+-----+-----+

window	verified count
{2023-05-12 15:45:00, 2023-05-12 15:45:10} false	634
{2023-05-12 15:45:00, 2023-05-12 15:45:10} true	108
{2023-05-12 15:45:10, 2023-05-12 15:45:20} true	203
{2023-05-12 15:45:10, 2023-05-12 15:45:20} false	1115
{2023-05-12 15:45:20, 2023-05-12 15:45:30} true	128
{2023-05-12 15:45:20, 2023-05-12 15:45:30} false	1047
{2023-05-12 15:45:30, 2023-05-12 15:45:40} true	132
{2023-05-12 15:45:30, 2023-05-12 15:45:40} false	990
{2023-05-12 15:45:40, 2023-05-12 15:45:50} false	996
{2023-05-12 15:45:40, 2023-05-12 15:45:50} true	156
{2023-05-12 15:45:50, 2023-05-12 15:46:00} false	1018
{2023-05-12 15:45:50, 2023-05-12 15:46:00} true	143
{2023-05-12 15:46:00, 2023-05-12 15:46:10} false	988
{2023-05-12 15:46:00, 2023-05-12 15:46:10} true	165
{2023-05-12 15:46:10, 2023-05-12 15:46:20} true	168
{2023-05-12 15:46:10, 2023-05-12 15:46:20} false	1016
{2023-05-12 15:46:20, 2023-05-12 15:46:30} false	1018
{2023-05-12 15:46:20, 2023-05-12 15:46:30} true	167
{2023-05-12 15:46:30, 2023-05-12 15:46:40} true	174
{2023-05-12 15:46:30, 2023-05-12 15:46:40} false	969

only showing top 20 rows

window	verified count
{2023-05-12 15:45:00, 2023-05-12 15:45:10} false	634
{2023-05-12 15:45:00, 2023-05-12 15:45:10} true	108
{2023-05-12 15:45:10, 2023-05-12 15:45:20} true	203
{2023-05-12 15:45:10, 2023-05-12 15:45:20} false	1115
{2023-05-12 15:45:20, 2023-05-12 15:45:30} true	128
{2023-05-12 15:45:20, 2023-05-12 15:45:30} false	1047

{2023-05-12 15:45:30, 2023-05-12 15:45:40} true	132	
{2023-05-12 15:45:30, 2023-05-12 15:45:40} false	990	
{2023-05-12 15:45:40, 2023-05-12 15:45:50} false	996	
{2023-05-12 15:45:40, 2023-05-12 15:45:50} true	156	
{2023-05-12 15:45:50, 2023-05-12 15:46:00} false	1018	
{2023-05-12 15:45:50, 2023-05-12 15:46:00} true	143	
{2023-05-12 15:46:00, 2023-05-12 15:46:10} false	988	
{2023-05-12 15:46:00, 2023-05-12 15:46:10} true	165	
{2023-05-12 15:46:10, 2023-05-12 15:46:20} true	168	
{2023-05-12 15:46:10, 2023-05-12 15:46:20} false	1016	
{2023-05-12 15:46:20, 2023-05-12 15:46:30} false	1018	
{2023-05-12 15:46:20, 2023-05-12 15:46:30} true	167	
{2023-05-12 15:46:30, 2023-05-12 15:46:40} true	174	
{2023-05-12 15:46:30, 2023-05-12 15:46:40} false	969	

+-----+-----+-----+

only showing top 20 rows

window	verified count
{2023-05-12 15:45:00, 2023-05-12 15:45:10} false	634
{2023-05-12 15:45:00, 2023-05-12 15:45:10} true	108
{2023-05-12 15:45:10, 2023-05-12 15:45:20} false	1115
{2023-05-12 15:45:10, 2023-05-12 15:45:20} true	203
{2023-05-12 15:45:20, 2023-05-12 15:45:30} true	128
{2023-05-12 15:45:20, 2023-05-12 15:45:30} false	1047
{2023-05-12 15:45:30, 2023-05-12 15:45:40} true	132
{2023-05-12 15:45:30, 2023-05-12 15:45:40} false	990
{2023-05-12 15:45:40, 2023-05-12 15:45:50} true	156
{2023-05-12 15:45:40, 2023-05-12 15:45:50} false	996
{2023-05-12 15:45:50, 2023-05-12 15:46:00} false	1018
{2023-05-12 15:45:50, 2023-05-12 15:46:00} true	143
{2023-05-12 15:46:00, 2023-05-12 15:46:10} false	988
{2023-05-12 15:46:00, 2023-05-12 15:46:10} true	165
{2023-05-12 15:46:10, 2023-05-12 15:46:20} true	168
{2023-05-12 15:46:10, 2023-05-12 15:46:20} false	1016
{2023-05-12 15:46:20, 2023-05-12 15:46:30} false	1018
{2023-05-12 15:46:20, 2023-05-12 15:46:30} true	167
{2023-05-12 15:46:30, 2023-05-12 15:46:40} true	174
{2023-05-12 15:46:30, 2023-05-12 15:46:40} false	969

+-----+-----+-----+

only showing top 20 rows

+-----+-----+-----+

window	verified count
{2023-05-12 15:45:00, 2023-05-12 15:45:10} false	634
{2023-05-12 15:45:00, 2023-05-12 15:45:10} true	108
{2023-05-12 15:45:10, 2023-05-12 15:45:20} false	1115
{2023-05-12 15:45:10, 2023-05-12 15:45:20} true	203
{2023-05-12 15:45:20, 2023-05-12 15:45:30} true	128
{2023-05-12 15:45:20, 2023-05-12 15:45:30} false	1047
{2023-05-12 15:45:30, 2023-05-12 15:45:40} true	132
{2023-05-12 15:45:30, 2023-05-12 15:45:40} false	990
{2023-05-12 15:45:40, 2023-05-12 15:45:50} true	156
{2023-05-12 15:45:40, 2023-05-12 15:45:50} false	996
{2023-05-12 15:45:50, 2023-05-12 15:46:00} false	1018
{2023-05-12 15:45:50, 2023-05-12 15:46:00} true	143
{2023-05-12 15:46:00, 2023-05-12 15:46:10} false	988
{2023-05-12 15:46:00, 2023-05-12 15:46:10} true	165
{2023-05-12 15:46:10, 2023-05-12 15:46:20} true	168
{2023-05-12 15:46:10, 2023-05-12 15:46:20} false	1016
{2023-05-12 15:46:20, 2023-05-12 15:46:30} false	1018
{2023-05-12 15:46:20, 2023-05-12 15:46:30} true	167
{2023-05-12 15:46:30, 2023-05-12 15:46:40} true	174
{2023-05-12 15:46:30, 2023-05-12 15:46:40} false	969

only showing top 20 rows

window	verified count
{2023-05-12 15:45:00, 2023-05-12 15:45:10} false	634
{2023-05-12 15:45:00, 2023-05-12 15:45:10} true	108
{2023-05-12 15:45:10, 2023-05-12 15:45:20} false	1115
{2023-05-12 15:45:10, 2023-05-12 15:45:20} true	203
{2023-05-12 15:45:20, 2023-05-12 15:45:30} true	128
{2023-05-12 15:45:20, 2023-05-12 15:45:30} false	1047
{2023-05-12 15:45:30, 2023-05-12 15:45:40} true	132
{2023-05-12 15:45:30, 2023-05-12 15:45:40} false	990
{2023-05-12 15:45:40, 2023-05-12 15:45:50} true	156
{2023-05-12 15:45:40, 2023-05-12 15:45:50} false	996
{2023-05-12 15:45:50, 2023-05-12 15:46:00} false	1018
{2023-05-12 15:45:50, 2023-05-12 15:46:00} true	143
{2023-05-12 15:46:00, 2023-05-12 15:46:10} false	988
{2023-05-12 15:46:00, 2023-05-12 15:46:10} true	165
{2023-05-12 15:46:10, 2023-05-12 15:46:20} true	168
{2023-05-12 15:46:10, 2023-05-12 15:46:20} false	1016
{2023-05-12 15:46:20, 2023-05-12 15:46:30} false	1018
{2023-05-12 15:46:20, 2023-05-12 15:46:30} true	167



{2023-05-12 15:46:30, 2023-05-12 15:46:40} true	174	
{2023-05-12 15:46:30, 2023-05-12 15:46:40} false	969	

+-----+-----+-----+

only showing top 20 rows

window	verified	count
--------	----------	-------

+-----+-----+-----+

{2023-05-12 15:45:00, 2023-05-12 15:45:10} false	634	
{2023-05-12 15:45:00, 2023-05-12 15:45:10} true	108	
{2023-05-12 15:45:10, 2023-05-12 15:45:20} false	1115	
{2023-05-12 15:45:10, 2023-05-12 15:45:20} true	203	
{2023-05-12 15:45:20, 2023-05-12 15:45:30} true	128	
{2023-05-12 15:45:20, 2023-05-12 15:45:30} false	1047	
{2023-05-12 15:45:30, 2023-05-12 15:45:40} true	132	
{2023-05-12 15:45:30, 2023-05-12 15:45:40} false	990	
{2023-05-12 15:45:40, 2023-05-12 15:45:50} true	156	
{2023-05-12 15:45:40, 2023-05-12 15:45:50} false	996	
{2023-05-12 15:45:50, 2023-05-12 15:46:00} false	1018	
{2023-05-12 15:45:50, 2023-05-12 15:46:00} true	143	
{2023-05-12 15:46:00, 2023-05-12 15:46:10} false	988	
{2023-05-12 15:46:00, 2023-05-12 15:46:10} true	165	
{2023-05-12 15:46:10, 2023-05-12 15:46:20} true	168	
{2023-05-12 15:46:10, 2023-05-12 15:46:20} false	1016	
{2023-05-12 15:46:20, 2023-05-12 15:46:30} false	1018	
{2023-05-12 15:46:20, 2023-05-12 15:46:30} true	167	
{2023-05-12 15:46:30, 2023-05-12 15:46:40} true	174	
{2023-05-12 15:46:30, 2023-05-12 15:46:40} false	969	

+-----+-----+-----+

only showing top 20 rows

window	verified	count
--------	----------	-------

+-----+-----+-----+

{2023-05-12 15:45:00, 2023-05-12 15:45:10} false	634	
{2023-05-12 15:45:00, 2023-05-12 15:45:10} true	108	
{2023-05-12 15:45:10, 2023-05-12 15:45:20} true	203	
{2023-05-12 15:45:10, 2023-05-12 15:45:20} false	1115	
{2023-05-12 15:45:20, 2023-05-12 15:45:30} true	128	
{2023-05-12 15:45:20, 2023-05-12 15:45:30} false	1047	
{2023-05-12 15:45:30, 2023-05-12 15:45:40} true	132	
{2023-05-12 15:45:30, 2023-05-12 15:45:40} false	990	
{2023-05-12 15:45:40, 2023-05-12 15:45:50} true	156	
{2023-05-12 15:45:40, 2023-05-12 15:45:50} false	996	

{2023-05-12 15:45:50, 2023-05-12 15:46:00} false	1018
{2023-05-12 15:45:50, 2023-05-12 15:46:00} true	143
{2023-05-12 15:46:00, 2023-05-12 15:46:10} false	988
{2023-05-12 15:46:00, 2023-05-12 15:46:10} true	165
{2023-05-12 15:46:10, 2023-05-12 15:46:20} true	168
{2023-05-12 15:46:10, 2023-05-12 15:46:20} false	1016
{2023-05-12 15:46:20, 2023-05-12 15:46:30} false	1018
{2023-05-12 15:46:20, 2023-05-12 15:46:30} true	167
{2023-05-12 15:46:30, 2023-05-12 15:46:40} true	174
{2023-05-12 15:46:30, 2023-05-12 15:46:40} false	969

+-----+-----+-----+

only showing top 20 rows

window	verified count
{2023-05-12 15:45:00, 2023-05-12 15:45:10} false	634
{2023-05-12 15:45:00, 2023-05-12 15:45:10} true	108
{2023-05-12 15:45:10, 2023-05-12 15:45:20} false	1115
{2023-05-12 15:45:10, 2023-05-12 15:45:20} true	203
{2023-05-12 15:45:20, 2023-05-12 15:45:30} false	1047
{2023-05-12 15:45:20, 2023-05-12 15:45:30} true	128
{2023-05-12 15:45:30, 2023-05-12 15:45:40} false	990
{2023-05-12 15:45:30, 2023-05-12 15:45:40} true	132
{2023-05-12 15:45:40, 2023-05-12 15:45:50} false	996
{2023-05-12 15:45:40, 2023-05-12 15:45:50} true	156
{2023-05-12 15:45:50, 2023-05-12 15:46:00} false	1018
{2023-05-12 15:45:50, 2023-05-12 15:46:00} true	143
{2023-05-12 15:46:00, 2023-05-12 15:46:10} false	988
{2023-05-12 15:46:00, 2023-05-12 15:46:10} true	165
{2023-05-12 15:46:10, 2023-05-12 15:46:20} false	1016
{2023-05-12 15:46:10, 2023-05-12 15:46:20} true	168
{2023-05-12 15:46:20, 2023-05-12 15:46:30} false	1018
{2023-05-12 15:46:20, 2023-05-12 15:46:30} true	167
{2023-05-12 15:46:30, 2023-05-12 15:46:40} true	174
{2023-05-12 15:46:30, 2023-05-12 15:46:40} false	969

+-----+-----+-----+

only showing top 20 rows

[Stage 598:=====> (106 + 6) / 200]

13:56:07.449 [stream execution thread for sliding\_windows\_tweet\_count [id = 66a51207-893e-4967-9200-e49f8240b92a, runId = 540db304-e037-49b9-aa8b-a08502f6238d]] ERROR  
org.apache.spark.sql.execution.datasources.v2.WriteToDataSourceV2Exec - Data source write support MicroBatchWrite[epoch: 8, writer:

```
org.apache.spark.sql.execution.streaming.sources.MemoryStreamingWrite@3eed8415]
is aborting.
13:56:07.450 [stream execution thread for sliding_windows_tweet_count [id =
66a51207-893e-4967-9200-e49f8240b92a, runId =
540db304-e037-49b9-aa8b-a08502f6238d]] ERROR
org.apache.spark.sql.execution.datasources.v2.WriteToDataSourceV2Exec - Data
source write support MicroBatchWrite[epoch: 8, writer:
org.apache.spark.sql.execution.streaming.sources.MemoryStreamingWrite@3eed8415]
aborted.
```

□ Count the number of event with a 10 seconds time window sliding every 5 seconds

```
[91]: sliding_windows_tweet_count = tweet_stream\
      .groupBy(window(tweet_stream.data.created_at, "10 seconds", "5 seconds" ))\
      .count()

sliding_windows_tweet_count_query = sliding_windows_tweet_count\
      .writeStream\
      .queryName("sliding_windows_tweet_count")\
      .format('memory')\
      .outputMode("complete")\
      .start()

for _ in range(10):
    spark.sql("""
        SELECT *
        FROM sliding_windows_tweet_count
        ORDER BY window """).show(truncate=False)
    sleep(3)

sliding_windows_tweet_count_query.stop()
```

```
+-----+-----+
|window|count|
+-----+-----+
+-----+-----+
```

[Stage 612:=====> (3 + 1) / 4]

```
+-----+-----+
|window|count|
+-----+-----+
+-----+-----+
```

[Stage 613:=====> (105 + 4) / 200]

```
+-----+-----+
|window|count|
```

```
+-----+-----+
+-----+-----+
```

window	count
{2023-05-12 10:24:45, 2023-05-12 10:24:55}	2
{2023-05-12 10:24:50, 2023-05-12 10:25:00}	2
{2023-05-12 15:40:00, 2023-05-12 15:40:10}	1
{2023-05-12 15:40:05, 2023-05-12 15:40:15}	1
{2023-05-12 15:44:55, 2023-05-12 15:45:05}	71
{2023-05-12 15:45:00, 2023-05-12 15:45:10}	373
{2023-05-12 15:45:05, 2023-05-12 15:45:15}	617
{2023-05-12 15:45:10, 2023-05-12 15:45:20}	621
{2023-05-12 15:45:15, 2023-05-12 15:45:25}	574
{2023-05-12 15:45:20, 2023-05-12 15:45:30}	548
{2023-05-12 15:45:25, 2023-05-12 15:45:35}	569
{2023-05-12 15:45:30, 2023-05-12 15:45:40}	562
{2023-05-12 15:45:35, 2023-05-12 15:45:45}	526
{2023-05-12 15:45:40, 2023-05-12 15:45:50}	575
{2023-05-12 15:45:45, 2023-05-12 15:45:55}	586
{2023-05-12 15:45:50, 2023-05-12 15:46:00}	568
{2023-05-12 15:45:55, 2023-05-12 15:46:05}	547
{2023-05-12 15:46:00, 2023-05-12 15:46:10}	565
{2023-05-12 15:46:05, 2023-05-12 15:46:15}	611
{2023-05-12 15:46:10, 2023-05-12 15:46:20}	552

only showing top 20 rows

window	count
{2023-05-12 10:24:45, 2023-05-12 10:24:55}	2
{2023-05-12 10:24:50, 2023-05-12 10:25:00}	2
{2023-05-12 15:40:00, 2023-05-12 15:40:10}	1
{2023-05-12 15:40:05, 2023-05-12 15:40:15}	1
{2023-05-12 15:44:55, 2023-05-12 15:45:05}	71
{2023-05-12 15:45:00, 2023-05-12 15:45:10}	373
{2023-05-12 15:45:05, 2023-05-12 15:45:15}	617
{2023-05-12 15:45:10, 2023-05-12 15:45:20}	621
{2023-05-12 15:45:15, 2023-05-12 15:45:25}	574
{2023-05-12 15:45:20, 2023-05-12 15:45:30}	548
{2023-05-12 15:45:25, 2023-05-12 15:45:35}	569
{2023-05-12 15:45:30, 2023-05-12 15:45:40}	562

{2023-05-12 15:45:35, 2023-05-12 15:45:45} 526	
{2023-05-12 15:45:40, 2023-05-12 15:45:50} 575	
{2023-05-12 15:45:45, 2023-05-12 15:45:55} 586	
{2023-05-12 15:45:50, 2023-05-12 15:46:00} 568	
{2023-05-12 15:45:55, 2023-05-12 15:46:05} 547	
{2023-05-12 15:46:00, 2023-05-12 15:46:10} 565	
{2023-05-12 15:46:05, 2023-05-12 15:46:15} 611	
{2023-05-12 15:46:10, 2023-05-12 15:46:20} 552	
+-----+	+-----+

only showing top 20 rows

+-----+	+-----+
window	count
+-----+	+-----+
{2023-05-12 10:24:45, 2023-05-12 10:24:55} 2	
{2023-05-12 10:24:50, 2023-05-12 10:25:00} 2	
{2023-05-12 15:40:00, 2023-05-12 15:40:10} 1	
{2023-05-12 15:40:05, 2023-05-12 15:40:15} 1	
{2023-05-12 15:44:55, 2023-05-12 15:45:05} 71	
{2023-05-12 15:45:00, 2023-05-12 15:45:10} 373	
{2023-05-12 15:45:05, 2023-05-12 15:45:15} 617	
{2023-05-12 15:45:10, 2023-05-12 15:45:20} 621	
{2023-05-12 15:45:15, 2023-05-12 15:45:25} 574	
{2023-05-12 15:45:20, 2023-05-12 15:45:30} 548	
{2023-05-12 15:45:25, 2023-05-12 15:45:35} 569	
{2023-05-12 15:45:30, 2023-05-12 15:45:40} 562	
{2023-05-12 15:45:35, 2023-05-12 15:45:45} 526	
{2023-05-12 15:45:40, 2023-05-12 15:45:50} 575	
{2023-05-12 15:45:45, 2023-05-12 15:45:55} 586	
{2023-05-12 15:45:50, 2023-05-12 15:46:00} 568	
{2023-05-12 15:45:55, 2023-05-12 15:46:05} 547	
{2023-05-12 15:46:00, 2023-05-12 15:46:10} 565	
{2023-05-12 15:46:05, 2023-05-12 15:46:15} 611	
{2023-05-12 15:46:10, 2023-05-12 15:46:20} 552	
+-----+	+-----+

only showing top 20 rows

+-----+	+-----+
window	count
+-----+	+-----+
{2023-05-12 10:24:45, 2023-05-12 10:24:55} 2	
{2023-05-12 10:24:50, 2023-05-12 10:25:00} 2	
{2023-05-12 15:40:00, 2023-05-12 15:40:10} 1	
{2023-05-12 15:40:05, 2023-05-12 15:40:15} 1	

{2023-05-12 15:44:55, 2023-05-12 15:45:05} 71	
{2023-05-12 15:45:00, 2023-05-12 15:45:10} 373	
{2023-05-12 15:45:05, 2023-05-12 15:45:15} 617	
{2023-05-12 15:45:10, 2023-05-12 15:45:20} 621	
{2023-05-12 15:45:15, 2023-05-12 15:45:25} 574	
{2023-05-12 15:45:20, 2023-05-12 15:45:30} 548	
{2023-05-12 15:45:25, 2023-05-12 15:45:35} 569	
{2023-05-12 15:45:30, 2023-05-12 15:45:40} 562	
{2023-05-12 15:45:35, 2023-05-12 15:45:45} 526	
{2023-05-12 15:45:40, 2023-05-12 15:45:50} 575	
{2023-05-12 15:45:45, 2023-05-12 15:45:55} 586	
{2023-05-12 15:45:50, 2023-05-12 15:46:00} 568	
{2023-05-12 15:45:55, 2023-05-12 15:46:05} 547	
{2023-05-12 15:46:00, 2023-05-12 15:46:10} 565	
{2023-05-12 15:46:05, 2023-05-12 15:46:15} 611	
{2023-05-12 15:46:10, 2023-05-12 15:46:20} 552	
+-----+-----+	

only showing top 20 rows

window	count
{2023-05-12 10:24:45, 2023-05-12 10:24:55} 2	
{2023-05-12 10:24:50, 2023-05-12 10:25:00} 2	
{2023-05-12 15:40:00, 2023-05-12 15:40:10} 1	
{2023-05-12 15:40:05, 2023-05-12 15:40:15} 1	
{2023-05-12 15:44:55, 2023-05-12 15:45:05} 71	
{2023-05-12 15:45:00, 2023-05-12 15:45:10} 373	
{2023-05-12 15:45:05, 2023-05-12 15:45:15} 617	
{2023-05-12 15:45:10, 2023-05-12 15:45:20} 621	
{2023-05-12 15:45:15, 2023-05-12 15:45:25} 574	
{2023-05-12 15:45:20, 2023-05-12 15:45:30} 548	
{2023-05-12 15:45:25, 2023-05-12 15:45:35} 569	
{2023-05-12 15:45:30, 2023-05-12 15:45:40} 562	
{2023-05-12 15:45:35, 2023-05-12 15:45:45} 526	
{2023-05-12 15:45:40, 2023-05-12 15:45:50} 575	
{2023-05-12 15:45:45, 2023-05-12 15:45:55} 586	
{2023-05-12 15:45:50, 2023-05-12 15:46:00} 568	
{2023-05-12 15:45:55, 2023-05-12 15:46:05} 547	
{2023-05-12 15:46:00, 2023-05-12 15:46:10} 565	
{2023-05-12 15:46:05, 2023-05-12 15:46:15} 611	
{2023-05-12 15:46:10, 2023-05-12 15:46:20} 552	
+-----+-----+	

only showing top 20 rows

```

+-----+-----+
|window                                     |count|
+-----+-----+
|{2023-05-12 10:24:45, 2023-05-12 10:24:55}|2    |
|{2023-05-12 10:24:50, 2023-05-12 10:25:00}|2    |
|{2023-05-12 15:40:00, 2023-05-12 15:40:10}|1    |
|{2023-05-12 15:40:05, 2023-05-12 15:40:15}|1    |
|{2023-05-12 15:44:55, 2023-05-12 15:45:05}|71   |
|{2023-05-12 15:45:00, 2023-05-12 15:45:10}|373  |
|{2023-05-12 15:45:05, 2023-05-12 15:45:15}|617  |
|{2023-05-12 15:45:10, 2023-05-12 15:45:20}|621  |
|{2023-05-12 15:45:15, 2023-05-12 15:45:25}|574  |
|{2023-05-12 15:45:20, 2023-05-12 15:45:30}|548  |
|{2023-05-12 15:45:25, 2023-05-12 15:45:35}|569  |
|{2023-05-12 15:45:30, 2023-05-12 15:45:40}|562  |
|{2023-05-12 15:45:35, 2023-05-12 15:45:45}|526  |
|{2023-05-12 15:45:40, 2023-05-12 15:45:50}|575  |
|{2023-05-12 15:45:45, 2023-05-12 15:45:55}|586  |
|{2023-05-12 15:45:50, 2023-05-12 15:46:00}|568  |
|{2023-05-12 15:45:55, 2023-05-12 15:46:05}|547  |
|{2023-05-12 15:46:00, 2023-05-12 15:46:10}|565  |
|{2023-05-12 15:46:05, 2023-05-12 15:46:15}|611  |
|{2023-05-12 15:46:10, 2023-05-12 15:46:20}|552  |
+-----+-----+
only showing top 20 rows

```

```

+-----+-----+
|window                                     |count|
+-----+-----+
|{2023-05-12 10:24:45, 2023-05-12 10:24:55}|2    |
|{2023-05-12 10:24:50, 2023-05-12 10:25:00}|2    |
|{2023-05-12 15:40:00, 2023-05-12 15:40:10}|1    |
|{2023-05-12 15:40:05, 2023-05-12 15:40:15}|1    |
|{2023-05-12 15:44:55, 2023-05-12 15:45:05}|71   |
|{2023-05-12 15:45:00, 2023-05-12 15:45:10}|373  |
|{2023-05-12 15:45:05, 2023-05-12 15:45:15}|617  |
|{2023-05-12 15:45:10, 2023-05-12 15:45:20}|621  |
|{2023-05-12 15:45:15, 2023-05-12 15:45:25}|574  |
|{2023-05-12 15:45:20, 2023-05-12 15:45:30}|548  |
|{2023-05-12 15:45:25, 2023-05-12 15:45:35}|569  |
|{2023-05-12 15:45:30, 2023-05-12 15:45:40}|562  |
|{2023-05-12 15:45:35, 2023-05-12 15:45:45}|526  |
|{2023-05-12 15:45:40, 2023-05-12 15:45:50}|575  |
|{2023-05-12 15:45:45, 2023-05-12 15:45:55}|586  |

```

```
|{2023-05-12 15:45:50, 2023-05-12 15:46:00}|568 |
|{2023-05-12 15:45:55, 2023-05-12 15:46:05}|547 |
|{2023-05-12 15:46:00, 2023-05-12 15:46:10}|565 |
|{2023-05-12 15:46:05, 2023-05-12 15:46:15}|611 |
|{2023-05-12 15:46:10, 2023-05-12 15:46:20}|552 |
+-----+-----+
```

only showing top 20 rows

```
[Stage 628:=====> (65 + 4) / 200]
```

```
13:57:54.670 [stream execution thread for sliding_windows_tweet_count [id =
2871fb59-d563-4e79-929b-8f27ad117bfc, runId =
17a1cdd5-8229-4082-b00e-3a1bd42dd2b5]] ERROR
org.apache.spark.sql.execution.datasources.v2.WriteToDataSourceV2Exec - Data
source write support MicroBatchWrite[epoch: 4, writer:
org.apache.spark.sql.execution.streaming.sources.MemoryStreamingWrite@4c2e60ab]
is aborting.
13:57:54.671 [stream execution thread for sliding_windows_tweet_count [id =
2871fb59-d563-4e79-929b-8f27ad117bfc, runId =
17a1cdd5-8229-4082-b00e-3a1bd42dd2b5]] ERROR
org.apache.spark.sql.execution.datasources.v2.WriteToDataSourceV2Exec - Data
source write support MicroBatchWrite[epoch: 4, writer:
org.apache.spark.sql.execution.streaming.sources.MemoryStreamingWrite@4c2e60ab]
aborted.
```

## 1.8.2 6.2 Handling late data with watermarks

Processing accordingly to time-event is great, but currently there is one flaw. We never specified how late we expect to see data. This means, Spark will keep some data in memory forever. Because streams never end, Spark will keep in memory every time windows, to be able to update some previous results. But in some cases, you know that after some time, you don't expect new data, or very late data aren't relevant any more. In other words, after a certain amount of time you want to freeze old results.

Once again, Spark can handle such process, with watermarks.



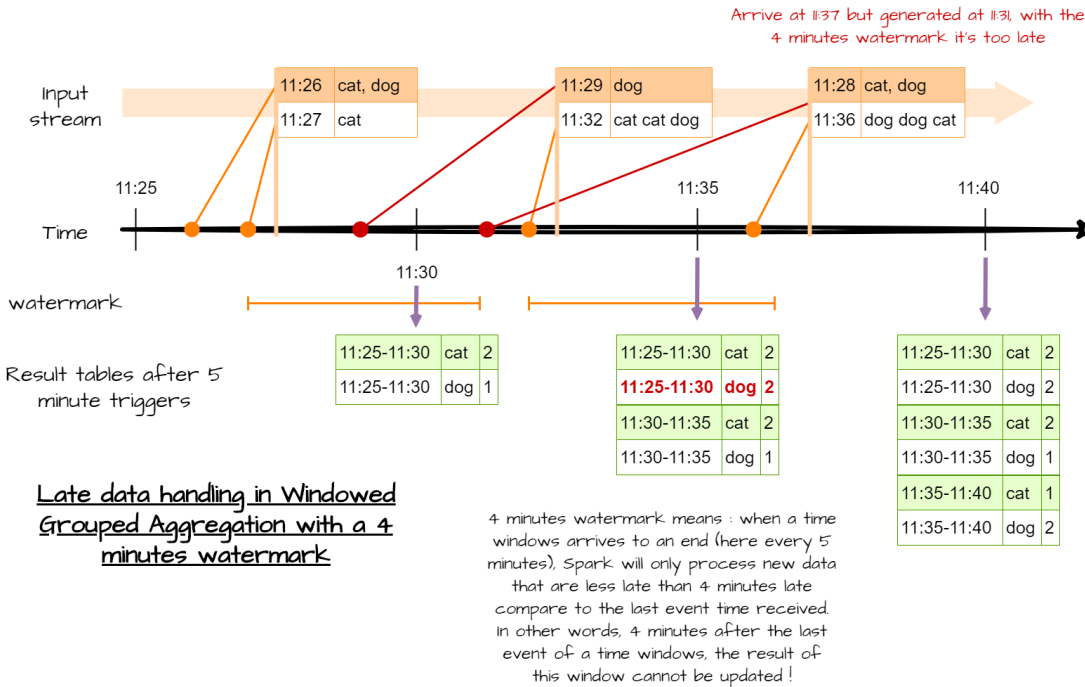


Figure 7 : Time-event processing with watermark

To do so, you have to define column as watermark and a the max delay. You have to use the `withWatermark(column, max_delay)` method.

*# Need some import*

```
from pyspark.sql.functions import window, col
```

*# word count + classic time window*

```
df_with_event_time.withWatermark(df_with_event_time.event_time, "4 minutes")\
.groupBy(
    window(df_with_event_time.event_time, "5 minutes"),
    df_with_event_time.word).count()
```

*# word count + sliding time window*

```
df_with_event_time.withWatermark(df_with_event_time.event_time, "4 minutes")\
.groupBy(
    window(df_with_event_time.event_time, "10 minutes", "5 minutes"),
    df_with_event_time.word).count()
```

Be careful, the watermark field cannot be a nested field ([link](#))

## Hand-on 6 : Handling late data with watermarks

- Count the number of event with a 10 seconds time window (use the `created_at` column) with a 5 seconds watermark

```
[92]: sliding_windows_tweet_count_with_watermark = tweet_stream\
.withColumn("timestamp", col("data.created_at"))\
```

```

.withWatermark("timestamp", "5 seconds")\
.groupBy(window("timestamp", "10 seconds"))\
.count()

sliding_windows_tweet_count_with_watermark_query =
↳sliding_windows_tweet_count_with_watermark\
.writeStream\
.queryName("sliding_windows_tweet_count_with_watermark")\
.format('memory')\
.outputMode("complete")\
.start()

for _ in range(10):
    spark.sql("""
        SELECT *
        FROM sliding_windows_tweet_count_with_watermark
        ORDER BY window """).show(truncate=False)
    sleep(5)

sliding_windows_tweet_count_with_watermark_query.stop()

```

```

+-----+-----+
|window|count|
+-----+-----+
+-----+-----+

```

[Stage 630:>

(0 + 5) / 6]

```

+-----+-----+
|window|count|
+-----+-----+
+-----+-----+

```

[Stage 631:=====>

(162 + 5) / 200]

```

+-----+-----+
|window|count|
+-----+-----+
+-----+-----+

```

```

+-----+-----+
|window|count|
+-----+-----+
|{2023-05-12 10:24:50, 2023-05-12 10:25:00}|2|
|{2023-05-12 15:40:00, 2023-05-12 15:40:10}|1|

```

{2023-05-12 15:45:00, 2023-05-12 15:45:10} 373	
{2023-05-12 15:45:10, 2023-05-12 15:45:20} 621	
{2023-05-12 15:45:20, 2023-05-12 15:45:30} 548	
{2023-05-12 15:45:30, 2023-05-12 15:45:40} 562	
{2023-05-12 15:45:40, 2023-05-12 15:45:50} 575	
{2023-05-12 15:45:50, 2023-05-12 15:46:00} 568	
{2023-05-12 15:46:00, 2023-05-12 15:46:10} 565	
{2023-05-12 15:46:10, 2023-05-12 15:46:20} 552	
{2023-05-12 15:46:20, 2023-05-12 15:46:30} 588	
{2023-05-12 15:46:30, 2023-05-12 15:46:40} 539	
{2023-05-12 15:46:40, 2023-05-12 15:46:50} 588	
{2023-05-12 15:46:50, 2023-05-12 15:47:00} 558	
{2023-05-12 15:47:00, 2023-05-12 15:47:10} 563	
{2023-05-12 15:47:10, 2023-05-12 15:47:20} 524	
{2023-05-12 15:47:20, 2023-05-12 15:47:30} 572	
{2023-05-12 15:47:30, 2023-05-12 15:47:40} 580	
{2023-05-12 15:47:40, 2023-05-12 15:47:50} 609	
{2023-05-12 15:47:50, 2023-05-12 15:48:00} 560	

+-----+

only showing top 20 rows

window	count
{2023-05-12 10:24:50, 2023-05-12 10:25:00} 2	
{2023-05-12 15:40:00, 2023-05-12 15:40:10} 1	
{2023-05-12 15:45:00, 2023-05-12 15:45:10} 373	
{2023-05-12 15:45:10, 2023-05-12 15:45:20} 621	
{2023-05-12 15:45:20, 2023-05-12 15:45:30} 548	
{2023-05-12 15:45:30, 2023-05-12 15:45:40} 562	
{2023-05-12 15:45:40, 2023-05-12 15:45:50} 575	
{2023-05-12 15:45:50, 2023-05-12 15:46:00} 568	
{2023-05-12 15:46:00, 2023-05-12 15:46:10} 565	
{2023-05-12 15:46:10, 2023-05-12 15:46:20} 552	
{2023-05-12 15:46:20, 2023-05-12 15:46:30} 588	
{2023-05-12 15:46:30, 2023-05-12 15:46:40} 539	
{2023-05-12 15:46:40, 2023-05-12 15:46:50} 588	
{2023-05-12 15:46:50, 2023-05-12 15:47:00} 558	
{2023-05-12 15:47:00, 2023-05-12 15:47:10} 563	
{2023-05-12 15:47:10, 2023-05-12 15:47:20} 524	
{2023-05-12 15:47:20, 2023-05-12 15:47:30} 572	
{2023-05-12 15:47:30, 2023-05-12 15:47:40} 580	
{2023-05-12 15:47:40, 2023-05-12 15:47:50} 609	
{2023-05-12 15:47:50, 2023-05-12 15:48:00} 560	

+-----+

only showing top 20 rows

window	count
{2023-05-12 10:24:50, 2023-05-12 10:25:00}	2
{2023-05-12 15:40:00, 2023-05-12 15:40:10}	1
{2023-05-12 15:45:00, 2023-05-12 15:45:10}	373
{2023-05-12 15:45:10, 2023-05-12 15:45:20}	621
{2023-05-12 15:45:20, 2023-05-12 15:45:30}	548
{2023-05-12 15:45:30, 2023-05-12 15:45:40}	562
{2023-05-12 15:45:40, 2023-05-12 15:45:50}	575
{2023-05-12 15:45:50, 2023-05-12 15:46:00}	568
{2023-05-12 15:46:00, 2023-05-12 15:46:10}	565
{2023-05-12 15:46:10, 2023-05-12 15:46:20}	552
{2023-05-12 15:46:20, 2023-05-12 15:46:30}	588
{2023-05-12 15:46:30, 2023-05-12 15:46:40}	539
{2023-05-12 15:46:40, 2023-05-12 15:46:50}	588
{2023-05-12 15:46:50, 2023-05-12 15:47:00}	558
{2023-05-12 15:47:00, 2023-05-12 15:47:10}	563
{2023-05-12 15:47:10, 2023-05-12 15:47:20}	524
{2023-05-12 15:47:20, 2023-05-12 15:47:30}	572
{2023-05-12 15:47:30, 2023-05-12 15:47:40}	580
{2023-05-12 15:47:40, 2023-05-12 15:47:50}	609
{2023-05-12 15:47:50, 2023-05-12 15:48:00}	560

only showing top 20 rows

window	count
{2023-05-12 10:24:50, 2023-05-12 10:25:00}	2
{2023-05-12 15:40:00, 2023-05-12 15:40:10}	1
{2023-05-12 15:45:00, 2023-05-12 15:45:10}	373
{2023-05-12 15:45:10, 2023-05-12 15:45:20}	621
{2023-05-12 15:45:20, 2023-05-12 15:45:30}	548
{2023-05-12 15:45:30, 2023-05-12 15:45:40}	562
{2023-05-12 15:45:40, 2023-05-12 15:45:50}	575
{2023-05-12 15:45:50, 2023-05-12 15:46:00}	568
{2023-05-12 15:46:00, 2023-05-12 15:46:10}	565
{2023-05-12 15:46:10, 2023-05-12 15:46:20}	552
{2023-05-12 15:46:20, 2023-05-12 15:46:30}	588
{2023-05-12 15:46:30, 2023-05-12 15:46:40}	539
{2023-05-12 15:46:40, 2023-05-12 15:46:50}	588
{2023-05-12 15:46:50, 2023-05-12 15:47:00}	558

{2023-05-12 15:47:00, 2023-05-12 15:47:10} 563	
{2023-05-12 15:47:10, 2023-05-12 15:47:20} 524	
{2023-05-12 15:47:20, 2023-05-12 15:47:30} 572	
{2023-05-12 15:47:30, 2023-05-12 15:47:40} 580	
{2023-05-12 15:47:40, 2023-05-12 15:47:50} 609	
{2023-05-12 15:47:50, 2023-05-12 15:48:00} 560	

only showing top 20 rows

window	count
{2023-05-12 10:24:50, 2023-05-12 10:25:00} 2	
{2023-05-12 15:40:00, 2023-05-12 15:40:10} 1	
{2023-05-12 15:45:00, 2023-05-12 15:45:10} 373	
{2023-05-12 15:45:10, 2023-05-12 15:45:20} 621	
{2023-05-12 15:45:20, 2023-05-12 15:45:30} 548	
{2023-05-12 15:45:30, 2023-05-12 15:45:40} 562	
{2023-05-12 15:45:40, 2023-05-12 15:45:50} 575	
{2023-05-12 15:45:50, 2023-05-12 15:46:00} 568	
{2023-05-12 15:46:00, 2023-05-12 15:46:10} 565	
{2023-05-12 15:46:10, 2023-05-12 15:46:20} 552	
{2023-05-12 15:46:20, 2023-05-12 15:46:30} 588	
{2023-05-12 15:46:30, 2023-05-12 15:46:40} 539	
{2023-05-12 15:46:40, 2023-05-12 15:46:50} 588	
{2023-05-12 15:46:50, 2023-05-12 15:47:00} 558	
{2023-05-12 15:47:00, 2023-05-12 15:47:10} 563	
{2023-05-12 15:47:10, 2023-05-12 15:47:20} 524	
{2023-05-12 15:47:20, 2023-05-12 15:47:30} 572	
{2023-05-12 15:47:30, 2023-05-12 15:47:40} 580	
{2023-05-12 15:47:40, 2023-05-12 15:47:50} 609	
{2023-05-12 15:47:50, 2023-05-12 15:48:00} 560	

only showing top 20 rows

window	count
{2023-05-12 10:24:50, 2023-05-12 10:25:00} 2	
{2023-05-12 15:40:00, 2023-05-12 15:40:10} 1	
{2023-05-12 15:45:00, 2023-05-12 15:45:10} 373	
{2023-05-12 15:45:10, 2023-05-12 15:45:20} 621	
{2023-05-12 15:45:20, 2023-05-12 15:45:30} 548	
{2023-05-12 15:45:30, 2023-05-12 15:45:40} 562	

```

|{2023-05-12 15:45:40, 2023-05-12 15:45:50}|575 |
|{2023-05-12 15:45:50, 2023-05-12 15:46:00}|568 |
|{2023-05-12 15:46:00, 2023-05-12 15:46:10}|565 |
|{2023-05-12 15:46:10, 2023-05-12 15:46:20}|552 |
|{2023-05-12 15:46:20, 2023-05-12 15:46:30}|588 |
|{2023-05-12 15:46:30, 2023-05-12 15:46:40}|539 |
|{2023-05-12 15:46:40, 2023-05-12 15:46:50}|588 |
|{2023-05-12 15:46:50, 2023-05-12 15:47:00}|558 |
|{2023-05-12 15:47:00, 2023-05-12 15:47:10}|563 |
|{2023-05-12 15:47:10, 2023-05-12 15:47:20}|524 |
|{2023-05-12 15:47:20, 2023-05-12 15:47:30}|572 |
|{2023-05-12 15:47:30, 2023-05-12 15:47:40}|580 |
|{2023-05-12 15:47:40, 2023-05-12 15:47:50}|609 |
|{2023-05-12 15:47:50, 2023-05-12 15:48:00}|560 |
+-----+

```

only showing top 20 rows

```

+-----+
|window                                     |count|
+-----+
|{2023-05-12 10:24:50, 2023-05-12 10:25:00}|2   |
|{2023-05-12 15:40:00, 2023-05-12 15:40:10}|1   |
|{2023-05-12 15:45:00, 2023-05-12 15:45:10}|373 |
|{2023-05-12 15:45:10, 2023-05-12 15:45:20}|621 |
|{2023-05-12 15:45:20, 2023-05-12 15:45:30}|548 |
|{2023-05-12 15:45:30, 2023-05-12 15:45:40}|562 |
|{2023-05-12 15:45:40, 2023-05-12 15:45:50}|575 |
|{2023-05-12 15:45:50, 2023-05-12 15:46:00}|568 |
|{2023-05-12 15:46:00, 2023-05-12 15:46:10}|565 |
|{2023-05-12 15:46:10, 2023-05-12 15:46:20}|552 |
|{2023-05-12 15:46:20, 2023-05-12 15:46:30}|588 |
|{2023-05-12 15:46:30, 2023-05-12 15:46:40}|539 |
|{2023-05-12 15:46:40, 2023-05-12 15:46:50}|588 |
|{2023-05-12 15:46:50, 2023-05-12 15:47:00}|558 |
|{2023-05-12 15:47:00, 2023-05-12 15:47:10}|563 |
|{2023-05-12 15:47:10, 2023-05-12 15:47:20}|524 |
|{2023-05-12 15:47:20, 2023-05-12 15:47:30}|572 |
|{2023-05-12 15:47:30, 2023-05-12 15:47:40}|580 |
|{2023-05-12 15:47:40, 2023-05-12 15:47:50}|609 |
|{2023-05-12 15:47:50, 2023-05-12 15:48:00}|560 |
+-----+

```

only showing top 20 rows

[Stage 654:=====>

(129 + 4) / 200]

14:01:33.185 [stream execution thread for

```

sliding_windows_tweet_count_with_watermark [id =
51b9c6c4-9464-4759-a2c4-560e238b69f1, runId =
4bf2a54d-3bf0-4f8e-a5ab-b9958eb9d2a4]] ERROR
org.apache.spark.sql.execution.datasources.v2.WriteToDataSourceV2Exec - Data
source write support MicroBatchWrite[epoch: 8, writer:
org.apache.spark.sql.execution.streaming.sources.MemoryStreamingWrite@371604c7]
is aborting.
14:01:33.185 [stream execution thread for
sliding_windows_tweet_count_with_watermark [id =
51b9c6c4-9464-4759-a2c4-560e238b69f1, runId =
4bf2a54d-3bf0-4f8e-a5ab-b9958eb9d2a4]] ERROR
org.apache.spark.sql.execution.datasources.v2.WriteToDataSourceV2Exec - Data
source write support MicroBatchWrite[epoch: 8, writer:
org.apache.spark.sql.execution.streaming.sources.MemoryStreamingWrite@371604c7]
aborted.

```

- ☐ Count the number of event by hashtag with a 30 seconds time window with a 1 minute watermark

```

[93]: sliding_windows_tweet_count = tweet_stream\
    .withColumn("timestamp", col("data.created_at"))\
    .withWatermark("timestamp", "1 minute")\
    .withColumn("tag", explode("data.entities.hashtags.tag"))\
    .groupBy(window("timestamp", "30 seconds"), "tag")\
    .count()

sliding_windows_tweet_count_query = sliding_windows_tweet_count\
    .writeStream\
    .queryName("sliding_windows_tweet_count")\
    .format('memory')\
    .outputMode("complete")\
    .start()

for _ in range(10):
    spark.sql("""
        SELECT *
        FROM sliding_windows_tweet_count
        ORDER BY window """).show(truncate=False)
    sleep(3)

sliding_windows_tweet_count_query.stop()

```

```

+-----+---+-----+
|window|tag|count|
+-----+---+-----+
+-----+---+-----+

```

```

+-----+---+-----+
|window|tag|count|
+-----+---+-----+
+-----+---+-----+

```

[Stage 656:=====> (2 + 5) / 7]

```

+-----+---+-----+
|window|tag|count|
+-----+---+-----+
+-----+---+-----+

```

[Stage 657:=====> (20 + 4) / 200]

```

+-----+---+-----+
|window|tag|count|
+-----+---+-----+
+-----+---+-----+

```

[Stage 657:=====> (80 + 2) / 200]

```

+-----+---+-----+
|window|tag|count|
+-----+---+-----+
+-----+---+-----+

```

[Stage 657:=====> (135 + 2) / 200]

```

+-----+---+-----+
|window|tag|count|
+-----+---+-----+
+-----+---+-----+

```

window	tag	count
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	albumnumber10	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}		4
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	adipose	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	AllStars8	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	airdrop	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	spotify	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	HANYUJIN	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	PayPay	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	cashappfriday	1



{2023-05-12 15:45:00, 2023-05-12 15:45:30}	2023	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}		2	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1		
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	ListosParaAyudar	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	FursuitFriday	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	XiJinping	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	homeinspo	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	LewdRP	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1		
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1		

+-----+-----+-----+

only showing top 20 rows

window	tag	count
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	adipose	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	PayPay	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	airdrop	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	buzz	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	albumnumber10	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	RIKUxYinWar	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	MewMyUniverse	2
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	4	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	LMx92NewCollection	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	WarriorNun	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	homeinspo	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	2	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	Sabsta	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	HiHiJets	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	

+-----+-----+-----+

only showing top 20 rows

window	tag	count
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	adipose	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	PayPay	1

{2023-05-12 15:45:00, 2023-05-12 15:45:30} airdrop	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} buzz	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} albumnumber10	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} RIKUxYinWar	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} 1		
{2023-05-12 15:45:00, 2023-05-12 15:45:30} MewMyUniverse	2	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	4	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} LMx92NewCollection	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} WarriorNun	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} homeinspo	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	2	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} Sabsta	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} HiHiJets	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	

+-----+-----+-----+

only showing top 20 rows

window	tag	count
{2023-05-12 15:45:00, 2023-05-12 15:45:30} Afyonkarahisar		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} YEJUN		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} PLAVE		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30} 1		
{2023-05-12 15:45:00, 2023-05-12 15:45:30} BadBuddy_fanart		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30} HotaruAI		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} china		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} Jimin		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30} rti		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30} AuthorsofTwitter		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	

+-----+-----+-----+

only showing top 20 rows

- Count the number of events post by verified user with a 10 seconds time window sliding every 5 seconds with 25 seconds watermark.

```
[95]: sliding_windows_tweet_count2 = tweet_stream\
      .withColumn("verified", explode("includes.users.verified"))\
      .withColumn("timestamp", col("data.created_at"))\
      .withWatermark("timestamp", "25 seconds")\
      .filter("verified")\
      .groupBy(window("timestamp", "10 seconds", "5 seconds"), "includes.users.\
      ↪username")\
      .count()

sliding_windows_tweet_count_query2 = sliding_windows_tweet_count\
  .writeStream\
  .queryName("sliding_windows_tweet_count2")\
  .format('json')\
  .format('memory')\
  .outputMode("complete")\
  .start()

for _ in range(10):
    spark.sql("""
        SELECT *
          FROM sliding_windows_tweet_count2
        ORDER BY window """).show(truncate=False)
    sleep(5)

sliding_windows_tweet_count_query2.stop()
```

```
+-----+---+-----+
|window|tag|count|
+-----+---+-----+
+-----+---+-----+
```

[Stage 665:=====>

(152 + 4) / 200]

```
+-----+---+-----+
|window|tag|count|
+-----+---+-----+
+-----+---+-----+
```

```
+-----+-----+-----+
|window|tag|count|
+-----+-----+-----+
```

{2023-05-12 15:45:00, 2023-05-12 15:45:30} krisyantoyenoni	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} NewRelease	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	2	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} WarriorNun	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} AKB48	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} TearsOfTheKingdom	1	
{2023-05-12 15:45:00, 2023-05-12 15:45: _  {30	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} MewMyUniverse	2	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} MarkTuanAsiatour2023	2	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} GilgitBaltistan	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} MIDZY	1	

+-----+-----+-----+

only showing top 20 rows

window	tag	count
{2023-05-12 15:45:00, 2023-05-12 15:45:30} PLAVE		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30} TearsOfTheKingdom		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} yinyin_anw		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30} LMx92NewCollection		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30} PapaFrancesco		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30} MewMyUniverse		2
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	2	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} PayPay		1
{2023-05-12 15:45:00, 2023-05-12 15:45: _  {30	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} technologies		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30} WANGYIBO		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	2	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}  3	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} Sabsta		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30} rti		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30} MIDZY		1

```
+-----+
only showing top 20 rows
```

window	tag	count
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	Nimritians	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	bullbearAI	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	abidjan	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	ReformedesRetraites	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	OHMYGIRL	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	BunlaraOyVerme	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}		2
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	FlashbackFriday	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	AKB48	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	KCON	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	HiHiJets	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	25 prsk 3	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	WireAndCable	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	technologies	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	M	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	ternakbabi	1

```
+-----+
only showing top 20 rows
```

window	tag	count
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	bambam	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	LMx92NewCollection	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	NewRelease	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	swaprum	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	AuthorsofTwitter	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	PayPay	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	CONDIVISIONE	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	ternakbabi	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	AKB48	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}		2
{2023-05-12 15:45:00, 2023-05-12 15:45:30}		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	25 prsk 3	1

{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} Sabsta	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} technologies	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} luigi	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} abidjan	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}  3	1	

only showing top 20 rows

window	tag	count
{2023-05-12 15:45:00, 2023-05-12 15:45:30} krisyantoyenoni	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} abidjan	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} ebayseller	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} yinyin_anw	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} TearsOfTheKingdom	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	2	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} FlashbackFriday	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} Afyonkarahisar	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} MewMyUniverse	2	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} HiHiJets	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} 25 prsk 3	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} LMx92NewCollection	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} WireAndCable	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} technologies	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} PayPay	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}  3	1	

only showing top 20 rows

window	tag	count
{2023-05-12 15:45:00, 2023-05-12 15:45:30} abidjan	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} yinyin_anw	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} MASSIMAAA	1	

{2023-05-12 15:45:00, 2023-05-12 15:45:30} PLAVE	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} LaSirenita	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} HANYUJIN	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} YaminoArt	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	2	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} SixTONES	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} XiJinping	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} Mainnet	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} CONDIVISIONE	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} INcto	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	2	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	

+-----+-----+-----+

only showing top 20 rows

window	tag	count
{2023-05-12 15:45:00, 2023-05-12 15:45:30} PLAVE		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} abidjan		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30} ebayseller		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} TearsOfTheKingdom		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30} CONDIVISIONE		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30} Face		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30} yinyin_anw		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30} MewMyUniverse		2
{2023-05-12 15:45:00, 2023-05-12 15:45:30} HiHiJets		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30} WarriorNun		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30} HANYUJIN		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30} LMx92NewCollection		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} PayPay		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	1	
{2023-05-12 15:45:00, 2023-05-12 15:45:30} ISLANDTV		1

+-----+-----+-----+

only showing top 20 rows

window	tag	count
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	PLAVE	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	YSLBEAUTYTHXBAMBAM	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	3	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	TearsOfTheKingdom	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	25 prsk 3	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	flexfriday	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	MewMyUniverse	2
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	INI_DROP_That	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	WarriorNun	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}		1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	LMx92NewCollection	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	airdrop	1
{2023-05-12 15:45:00, 2023-05-12 15:45:30}	bullbearAI	1

only showing top 20 rows

[Stage 689:=====> (131 + 5) / 200]

```
14:04:32.208 [stream execution thread for sliding_windows_tweet_count2 [id =
e34a37e8-d373-47f2-bfa0-5f8270afabf0, runId =
5e3b8bec-583f-4ddf-b4db-f19ccb81a7fc]] ERROR
org.apache.spark.sql.execution.datasources.v2.WriteToDataSourceV2Exec - Data
source write support MicroBatchWrite[epoch: 8, writer:
org.apache.spark.sql.execution.streaming.sources.MemoryStreamingWrite@6c7ec970]
is aborting.
14:04:32.208 [stream execution thread for sliding_windows_tweet_count2 [id =
e34a37e8-d373-47f2-bfa0-5f8270afabf0, runId =
5e3b8bec-583f-4ddf-b4db-f19ccb81a7fc]] ERROR
org.apache.spark.sql.execution.datasources.v2.WriteToDataSourceV2Exec - Data
source write support MicroBatchWrite[epoch: 8, writer:
org.apache.spark.sql.execution.streaming.sources.MemoryStreamingWrite@6c7ec970]
aborted.
```

[Stage 689:=====> (140 + 4) / 200]



## 1.9 7 For more details

- Spark official documentation
- ZAHARIA, B. C. M. (2018). *Spark: the Definitive Guide.* , O'Reilly Media, Inc. <https://proquest.safaribooksonline.com/9781491912201>
- <https://databricks.com/blog/2018/03/13/introducing-stream-stream-joins-in-apache-spark-2-3.html>
- <https://databricks.com/blog/2016/07/28/structured-streaming-in-apache-spark.html>
- <https://databricks.com/blog/2015/07/30/diving-into-apache-spark-streamings-execution-model.html>

## 1.10 End of the Lab

- ☐ Export your notebook
  - Right click and Download (.ipynb)
  - File > Save and Export Notebook > HTML

### 1.10.1 SSPCloud

- ☐ Delete the Jupyter-pyspark service
  - SSPCloud > My services > Delete

### 1.10.2 AWS

- ☐ Terminate your cluster
  - On *EMR* service page, click on *Clusters*
  - Select the active cluster and click on **Terminate**

#### **Solution**

Solution