Fraudulent activity detection

Streaming Data project report

Teachers :
Ciritoglu Hegemen
Nitavskyi Oleksander
Viveret Jérôme
Group 4, students :
De Trogoff Charles
Millavet Maxime

The objective of this project is to create a Flink application to read events consisting in ad display and ad clicks and detect fraudulent behaviours. 3 fraud patterns are present and need to be identified. Info on the fraudulent event is then fed back as a stream.

We first proceeded to write a window of data for offline analysis, after which we detected 2 fraudulent patterns. We detected a third abnormal pattern however we could not validate whether it was truly a fraud or not. Finally, we send the fraudulent events to a Kafka topic for shared access.

I. Event streaming and output file generation

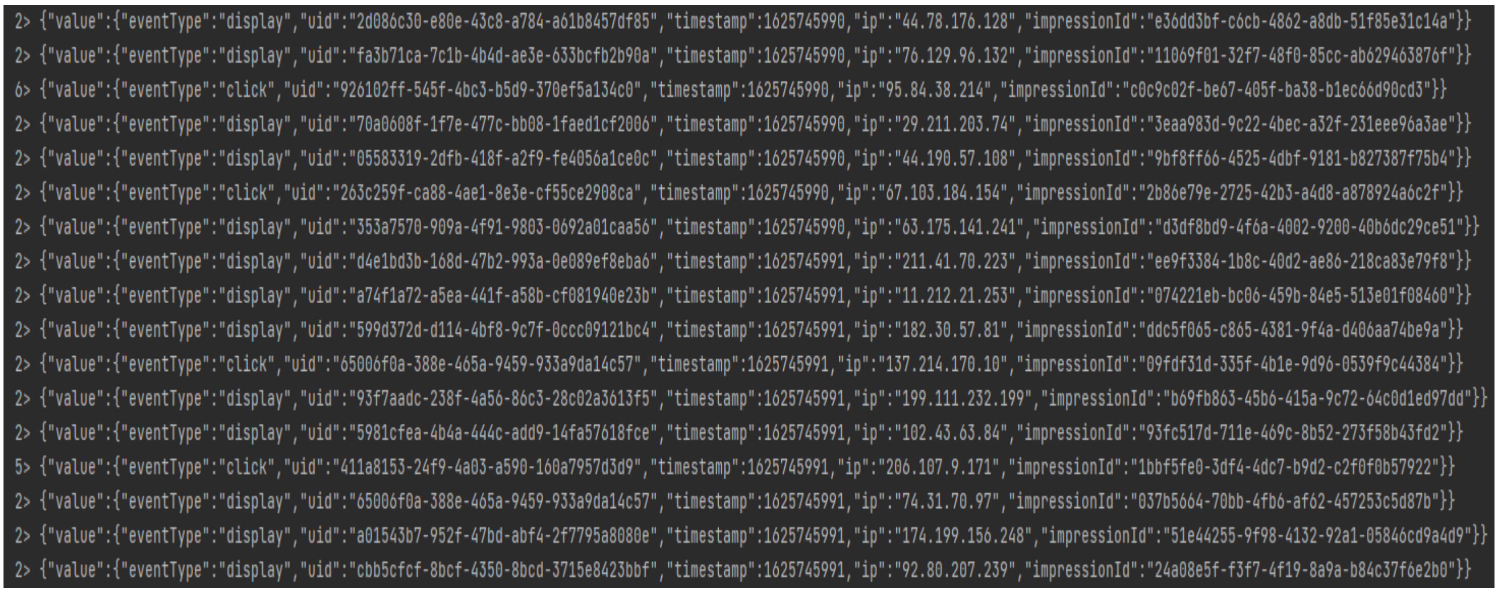
Event streaming :

In order to analyse and filter data we stream the display and clicks topics together. For easier processing we read the event in Json format in order to be able to extract the event info.

Later on we will select windows of data. To do so we had to add timestamps and watermarks to the events.

```
val topics: List[String] = new ArrayList[String]()
topics.add("clicks")
topics.add("displays")

// CONSUME STREAMS AND CONVERT TO JSON WITH WATERMARKS
val stream = env
 .addSource(new FlinkKafkaConsumer[ObjectNode](topics, new
JSONKeyValueDeserializationSchema(false), properties)

.assignTimestampsAndWatermarks(WatermarkStrategy.forBoundedOutOfOrdernes
s[ObjectNode](Duration.ofSeconds(20)))
 )
stream.print // stream the original data
```

The original data flow is as follows :



```
2> {"value":{"eventType":"display","uid":"2d086c30-e80e-43c8-a784-a61b8457df85","timestamp":1625745990,"ip":"44.78.176.128","impressionId":"e36dd3bf-c6cb-4862-a8db-51f85e31c14a"}}
2> {"value":{"eventType":"display","uid":"fa3b71ca-7c1b-4b4d-ae3e-633bcfb2b90a","timestamp":1625745990,"ip":"76.129.96.132","impressionId":"11069f01-32f7-48f0-85cc-ab629463876f"}}
6> {"value":{"eventType":"click","uid":"926102ff-545f-4bc3-b5d9-370ef5a134c0","timestamp":1625745990,"ip":"95.84.38.214","impressionId":"c0c9c02f-be67-405f-ba38-b1ec66d90cd3"}}
2> {"value":{"eventType":"display","uid":"70a0608f-1f7e-477c-bb08-1faed1cf2006","timestamp":1625745990,"ip":"29.211.203.74","impressionId":"3eaa983d-9c22-4bec-a32f-231eee96a3ae"}}
2> {"value":{"eventType":"display","uid":"05583319-2dfb-418f-a2f9-fe4056a1ce0c","timestamp":1625745990,"ip":"44.190.57.108","impressionId":"9bf8ff66-4525-4dbf-9181-b827387f75b4"}}
2> {"value":{"eventType":"click","uid":"263c259f-ca88-4ae1-8e3e-cf55ce2908ca","timestamp":1625745990,"ip":"67.103.184.154","impressionId":"2b86e79e-2725-42b3-a4d8-a878924a6c2f"}}
2> {"value":{"eventType":"display","uid":"353a7570-909a-4f91-9803-0692a01caa56","timestamp":1625745990,"ip":"63.175.141.241","impressionId":"d3df8bd9-4f6a-4002-9200-40b6dc29ce51"}}
2> {"value":{"eventType":"display","uid":"d4e1bd3b-168d-47b2-993a-0e089ef8eba6","timestamp":1625745991,"ip":"211.41.70.223","impressionId":"ee9f3384-1b8c-40d2-ae86-218ca83e79f8"}}
2> {"value":{"eventType":"display","uid":"a74f1a72-a5ea-441f-a58b-cf081940e23b","timestamp":1625745991,"ip":"11.212.21.253","impressionId":"074221eb-bc06-459b-84e5-513e01f08460"}}
2> {"value":{"eventType":"display","uid":"599d372d-d114-4bf8-9c7f-0ccc09121bc4","timestamp":1625745991,"ip":"182.30.57.81","impressionId":"ddc5f065-c865-4381-9f4a-d406aa74be9a"}}
2> {"value":{"eventType":"click","uid":"65006f0a-388e-465a-9459-933a9da14c57","timestamp":1625745991,"ip":"137.214.170.10","impressionId":"09fdf31d-335f-4b1e-9d96-0539f9c44384"}}
2> {"value":{"eventType":"display","uid":"93f7aadc-238f-4a56-86c3-28c02a3613f5","timestamp":1625745991,"ip":"199.111.232.199","impressionId":"b69fb863-45b6-415a-9c72-64c0d1ed97dd"}}
2> {"value":{"eventType":"display","uid":"5981cfea-4b4a-444c-add9-14fa57618fce","timestamp":1625745991,"ip":"102.43.63.84","impressionId":"93fc517d-711e-469c-8b52-273f58b43fd2"}}
5> {"value":{"eventType":"click","uid":"411a8153-24f9-4a03-a590-160a7957d3d9","timestamp":1625745991,"ip":"206.107.9.171","impressionId":"1bbf5fe0-3df4-4dc7-b9d2-c2f0f0b57922"}}
2> {"value":{"eventType":"display","uid":"65006f0a-388e-465a-9459-933a9da14c57","timestamp":1625745991,"ip":"74.31.70.97","impressionId":"037b5664-70bb-4fb6-af62-457253c5d87b"}}
2> {"value":{"eventType":"display","uid":"a01543b7-952f-47bd-abf4-2f7795a8080e","timestamp":1625745991,"ip":"174.199.156.248","impressionId":"51e44255-9f98-4132-92a1-05846cd9a4d9"}}
2> {"value":{"eventType":"display","uid":"cbb5cfcf-8bcf-4350-8bcd-3715e8423bbf","timestamp":1625745991,"ip":"92.80.207.239","impressionId":"24a08e5f-f3f7-4f19-8a9a-b84c37f6e2b0"}}
```

Stream converted to Json output

They are 5 infos contained in the event :

- eventType : either display or click
- uid : the user ID
- timestamp : the date at which the request was issued
- ip : the ip address from the request
- impressionId : an ID identifying the ad banner of the display or the click

Outputting the data to a file :

Sink

In order to analyze the stream, we want to output a long window of data to an output file for offline analysis. In order to do so we can create a sink that will output the data to a file :

```
CREATE SINK TO WRITE TO
val outputPath = "../eventSink"
val config = OutputFileConfig
 .builder()
 .withPartPrefix("5min_event")
 .withPartSuffix(".txt")
 .build()
val sink: StreamingFileSink[String] = StreamingFileSink
 .forRowFormat(new Path(outputPath), new
SimpleStringEncoder[String]("UTF-8"))
 .withRollingPolicy(
   DefaultRollingPolicy.builder()
     .withRolloverInterval(TimeUnit.MINUTES.toMillis(5)) // closes after
X minutes
     .withInactivityInterval(TimeUnit.MINUTES.toMillis(1)) // closes
after X minute of inactivity
     .withMaxPartSize(10 * 1024 * 1024) // Max size of 10 Mo
     .build())
 .withOutputFileConfig(config)
 .build()

stream
 .map(x => x.toString())
 .addSink(sink)
```

As we had trouble handling the sink data output, we finally opted for another solution and used a Python script to write the terminal output to a csv file.

Python script :

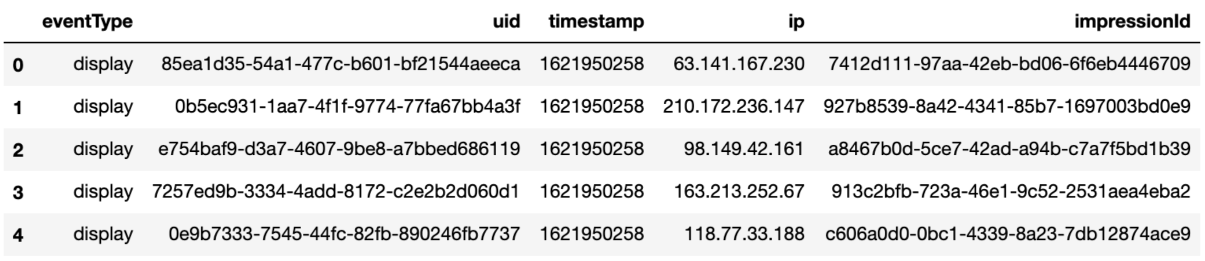
As the sink solution turned out to be rather unsatisfactory, we finally decided to use a python script in order to convert the terminal output originally in string format to a .csv. Thus, an offline analysis from a python notebook allowed us to identify the main fraudulent patterns. Below is the code to pass from a .txt to a .csv file:

```
#%% First part : static dataset creation

import os
from pathlib import Path, PurePath
import json
import csv
from csv import writer
from tqdm import tqdm


assert __name__ == '__main__', 'Wrong execution directory'

rootpath = Path(os.path.dirname(__file__))
textpath = rootpath / 'docker_compose_extract.txt'
csvpath = rootpath / 'static.csv'


# Create .csv file
with open(str(csvpath), 'w') as csvfile:
   filewriter = csv.writer(csvfile, delimiter=',',
                           quotechar='|',
quoting=csv.QUOTE_MINIMAL)
   filewriter.writerow(['eventType', 'uid', 'timestamp', 'ip',
'impressionId'])

def append_list_as_row(file_name, list_of_elem):
   # Open file in append mode
   with open(file_name, 'a+', newline='') as write_obj:
       # Create a writer object from csv module
       csv_writer = writer(write_obj)
       # Add contents of list as last row in the csv file
       csv_writer.writerow(list_of_elem)


# Iterate over raw .txt, parse and insert line by line into .csv
with open(str(textpath), 'r') as txtfile:
   count = 0
   for line in txtfile:
       print("Processing line{}".format(count))
       line_text = line.strip()

       # Select only the lines beginning with 'generator_1'
       if line_text[0:11] != 'generator_1':
           continue

       # Get dictionnary and feed csv with its elements
       str_dic = line_text[21:-1]
       dic = json.loads(str_dic)
       list_of_elem = [dic['eventType'], dic['uid'],
dic['timestamp'], dic['ip'], dic['impressionId']]
       append_list_as_row(str(csvpath), list_of_elem)

       count += 1
   print('Done')
```

II. Offline analysis

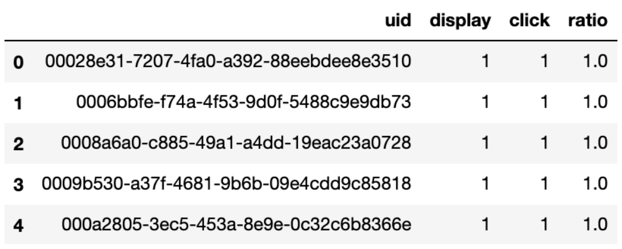
We did an offline analysis of the stream data with a Jupyter notebook in Python.

The data is stored in a dataframe which is the result of recording data from 2021-05-25 15:44:18 to 2021-05-25 17:34:41, i.e. almost two hours, which is more than sufficient for a confidence analysis.



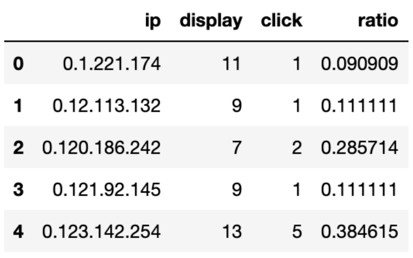
	eventType	uid	timestamp	ip	impressionId
0	display	85ea1d35-54a1-477c-b601-bf21544aeeca	1621950258	63.141.167.230	7412d111-97aa-42eb-bd06-6f6eb4446709
1	display	0b5ec931-1aa7-4f1f-9774-77fa67bb4a3f	1621950258	210.172.236.147	927b8539-8a42-4341-85b7-1697003bd0e9
2	display	e754baf9-d3a7-4607-9be8-a7bbed686119	1621950258	98.149.42.161	a8467b0d-5ce7-42ad-a94b-c7a7f5bd1b39
3	display	7257ed9b-3334-4add-8172-c2e2b2d060d1	1621950258	163.213.252.67	913c2bfb-723a-46e1-9c52-2531aea4eba2
4	display	0e9b7333-7545-44fc-82fb-890246fb7737	1621950258	118.77.33.188	c606a0d0-0bc1-4339-8a23-7db12874ace9

First, we looked at the CTR ratio per user, represented by the uid. We do the calculation: number of clicks divided by the number of displays. We notice that on average we get almost 100% CTR on the whole dataset, which is strange. Indeed, this would mean that all users systematically click on the ad as soon as they see it appear, which is totally unrealistic in reality. So there must be some fraud somewhere. We can visualize the table that shows this uids and the corresponding ratio in the figure below.



	uid	display	click	ratio
0	00028e31-7207-4fa0-a392-88eebdee8e3510	1	1	1.0
1	0006bbfe-f74a-4f53-9d0f-5488c9e9db73	1	1	1.0
2	0008a6a0-c885-49a1-a4dd-19eac23a0728	1	1	1.0
3	0009b530-a37f-4681-9b6b-09e4cdd9c85818	1	1	1.0
4	000a2805-3ec5-453a-8e9e-0c32c6b8366e	1	1	1.0

In a second step, we chose to perform exactly the same calculation but this time using IP addresses as reference. We obtain very different results as we can see in the table below. Indeed, the CTR is now around **30%** on average for each IP address. This is still higher than the **10%** value that would be expected in normal circumstances. This analysis suggests that there are necessarily many more users than IP addresses, and that perhaps this IP is fraudulent.

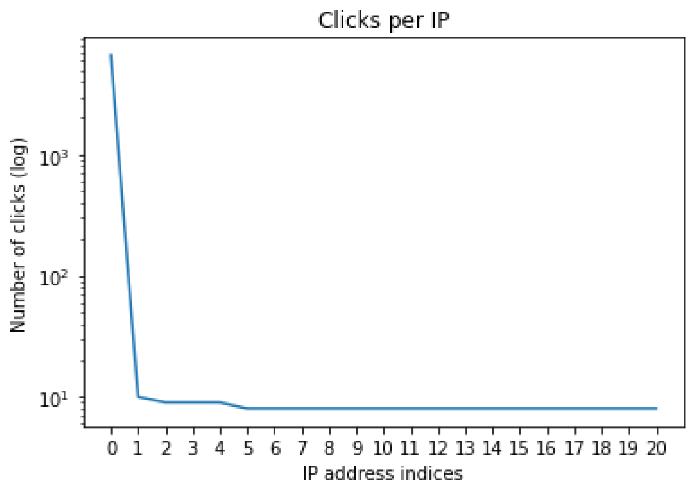


	ip	display	click	ratio
0	0.1.221.174	11	1	0.090909
1	0.12.113.132	9	1	0.111111
2	0.120.186.242	7	2	0.285714
3	0.121.92.145	9	1	0.111111
4	0.123.142.254	13	5	0.384615

If we do a little investigation on the distinct number of IP addresses, users, and advertisements, we get: **48795 UIDs, 10001 IPs, 94679 impressionIDs**, which confirms our suspicions.

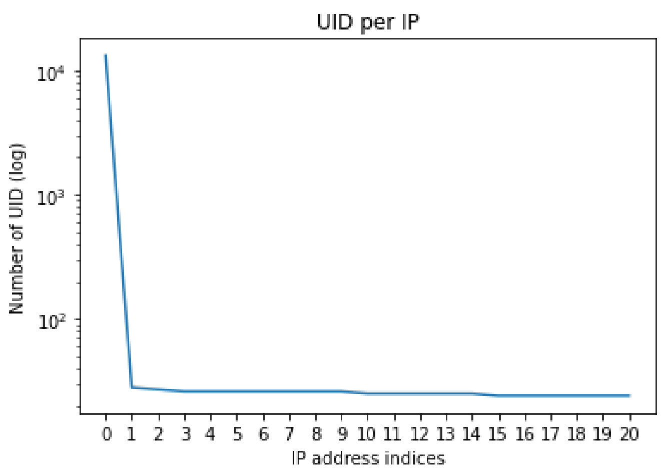
The three figures below show that there is clearly an outliner among the IP addresses. The average number of clicks for all IP addresses is 6.545511811023622. The IP with 6630 clicks is 238.186.83.58.

	ip	click_count
0	238.186.83.58	6630
1	206.218.212.93	10
2	16.7.14.204	9
3	7.45.3.132	9
4	178.192.51.223	9

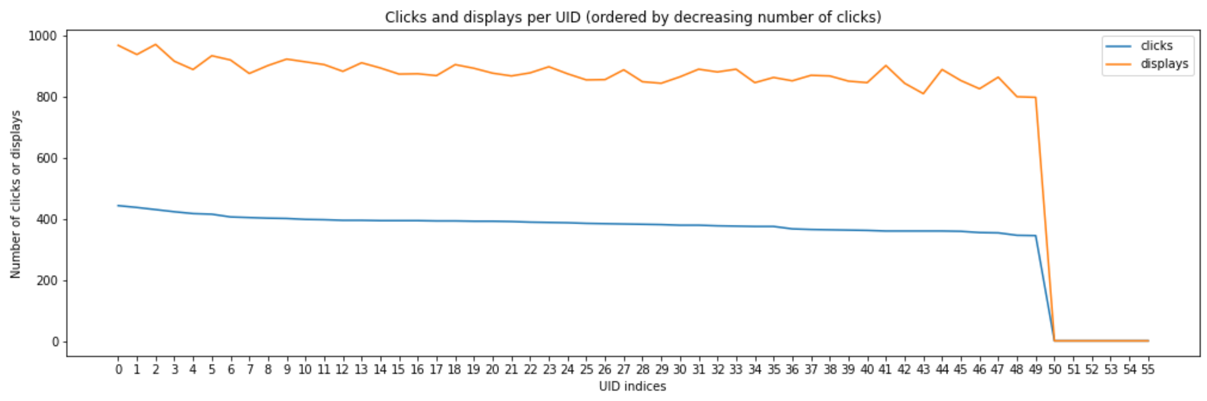


Now that we know that this IP address is fraudulent, we will confirm our analysis by looking at the number of users who used this IP address. The result is clear, in our sample of almost two hours, almost 14,000 users came from this same fraudulent IP address. **We can therefore assume that this IP address generated fake users who each clicked on ads.**

	ip	uid_count
0	238.186.83.58	13260
1	124.34.56.145	28
2	241.16.82.53	27
3	206.218.212.93	26
4	82.23.146.70	26



Finally, we wanted to look at the 50 users who had generated the most clicks. For these 50 users we also plotted the number of advertisements they faced.



III. Pattern filtering

Data extraction :

First, in order to filter the values, we wrote a function which gets the data from the event and maps it to a tuple value of the form ('uid', 'ip', 'impressionId', 'display_count', 'click_count', 'timestamp') :

```
////////////////////////////////////////////////////////////////////////
/////////////////////// EXTRACT DATA FROM STREAM

   def extract_data(x:ObjectNode) : (String, String, String, Int, Int,
Int) = {
     // maps to a tuple ('uid', 'ip', 'impressionId', 'display_count',
'click_count', 'timestamp')

     (
     x.get("value").get("uid").textValue,
     x.get("value").get("ip").textValue,
     x.get("value").get("impressionId").textValue,
     if (x.get("value").get("eventType").textValue=="display") 1 else 0,
     if (x.get("value").get("eventType").textValue=="click") 1 else 0,
     x.get("value").get("timestamp").intValue
     )
   }
   // FORMAT THE JSON STREAM TO THE DESIRED OUTPUT FOR ALL FILTERS
   val dataStream = stream
     //  maps to a tuple ('uid', 'IP', 'display_count', 'click_count')
       .map(x => extract_data(x))
```

We replace the 'eventType' of the data by two values 'display_count' and 'click_count' indicating what the event type is. It will be useful later when aggregating data.

Output sink :

Before filtering data, we prepare a sink to which the data will be sent to and then streamed alone :

```
// INSTANTIATE SINK TO SEND FRAUDULENT DATA TO WITH KafkaProducer
val fraudFilter = new FlinkKafkaProducer[String](
 "fraudulentEvent",                   // target topic
 new SimpleStringSchema(),     // serialization schema
 properties                   // producer config
 //      FlinkKafkaProducer.Semantic.NONE // fault-tolerance  // Does
not work
)

// ... (perform stream filtering)

// STREAM BACK FRAUDULENT DATA AND PRINT
val streamFraudulent = env
 .addSource(new FlinkKafkaConsumer[String]("fraudulentEvent", new
SimpleStringSchema(), properties))
streamFraudulent.print()
```

Filtering streams :

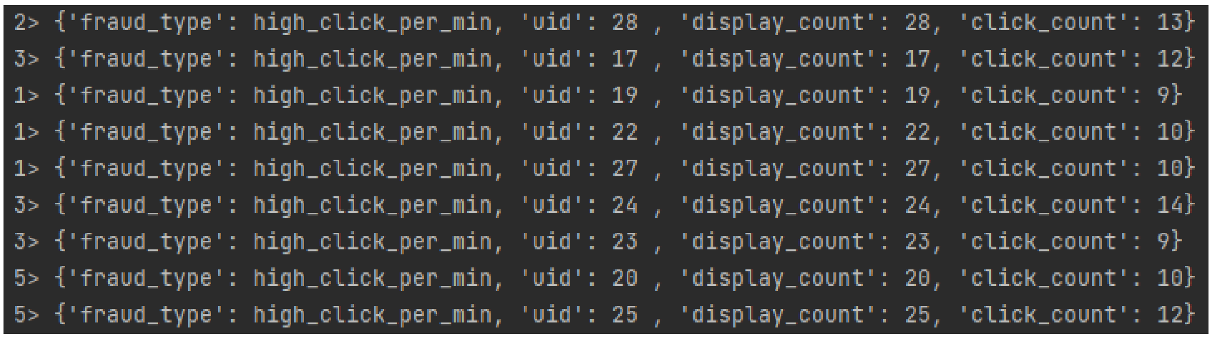
We implemented here the three filters to set apart the suspect events. In order to make computations on the data, we had to use a window of time. Kafka allows the use of a sliding window, which consists of several windows of fixed size started at regular intervals.

Pattern 1 : High click per minute rate

The first type of data that we filtered are the users who click too much. To do so we aggregated the data on the user ID and counted the number of displays and the number of clicks for each user. Afterward we had to choose a click per minute rate to filter the data based on our offline analysis.

```
val windowLength = 4
val windowDelay = 2
// FRAUD PATTERN 1 : HIGH CLICK PER MINUTE RATE, FILTER ABOVE 20 CLICKS
IN 10 MINUTES
// FORM WINDOW KEYED ON UID
val clickThroughRate = dataStream
     // keep ('uid', 'display_count', 'click_count')
     .map(x => (x._1, x._4, x._5))
     .keyBy(x => x._1) // key on uid
     .window(SlidingEventTimeWindows.of(Time.minutes(windowLength),
Time.minutes(windowDelay)))
     .reduce( (a, b) => (a._1, a._2+b._2, a._3+b._3) )
     .filter(x => x._3 > 2*windowLength)
     .map(x => f"{'fraud_type': high_click_per_min, 'uid': ${x._2} ,
'display_count': ${x._2}, 'click_count': ${x._3}}")
//      .map(x => format(fraud_type = "high_CTR", uid = x._1,
display_count = x._2, click_count = x._3))
     .addSink(fraudFilter)
```

The output of this filter is as follows :



```
2> {'fraud_type': high_click_per_min, 'uid': 28 , 'display_count': 28, 'click_count': 13}
3> {'fraud_type': high_click_per_min, 'uid': 17 , 'display_count': 17, 'click_count': 12}
1> {'fraud_type': high_click_per_min, 'uid': 19 , 'display_count': 19, 'click_count': 9}
1> {'fraud_type': high_click_per_min, 'uid': 22 , 'display_count': 22, 'click_count': 10}
1> {'fraud_type': high_click_per_min, 'uid': 27 , 'display_count': 27, 'click_count': 10}
3> {'fraud_type': high_click_per_min, 'uid': 24 , 'display_count': 24, 'click_count': 14}
3> {'fraud_type': high_click_per_min, 'uid': 23 , 'display_count': 23, 'click_count': 9}
5> {'fraud_type': high_click_per_min, 'uid': 20 , 'display_count': 20, 'click_count': 10}
5> {'fraud_type': high_click_per_min, 'uid': 25 , 'display_count': 25, 'click_count': 12}
```

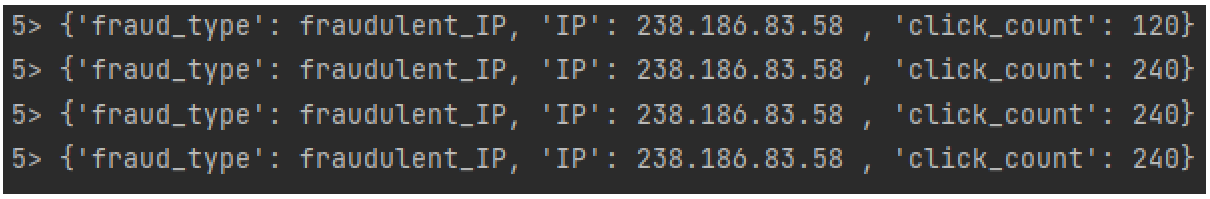
Filter 1 output : high click per minute rate for a user

Pattern 2 : Fraudulent IP

In our analysis we saw the IP ‘238.186.83.58’ was associated to an abnormally high number of events. This IP triggered a display and a click event with a lot of different user IDs, in the same impression ID and same timestamp. To filter fraudulent IPs we chose to aggregate the stream on the ‘ip’ value and set a filter for IPs with a click per minute rate above a threshold we chose based on the data. Another would have been to aggregate on the timestamp and impression ID values and filter data with both a display and a click.

```
// FRAUD PATTERN 2 : IP ADDRESS WITH ABNORMALLY HIGH NUMBER OF CLICK
ASSOCIATED, ABOVE 5 PER MINUTE
// FORM WINDOW KEYED ON IP
val fraudulentIP = dataStream
 // keep ('IP', 'click_count')
 .map(x => (x._2, x._5))
 .keyBy(x => x._1) // key on IP
 .window(SlidingEventTimeWindows.of(Time.minutes(windowLength),
Time.minutes(windowDelay)))
 .reduce( (a, b) => (a._1, a._2+b._2) )
 .filter(x => x._2 > 5*windowLength)
 .map(x => f"{'fraud_type': fraudulent_IP, 'IP': ${x._1} ,
'click_count': ${x._2}}")
 .addSink(fraudFilter)
```

This filter allowed us to isolate the fraudulent IP :

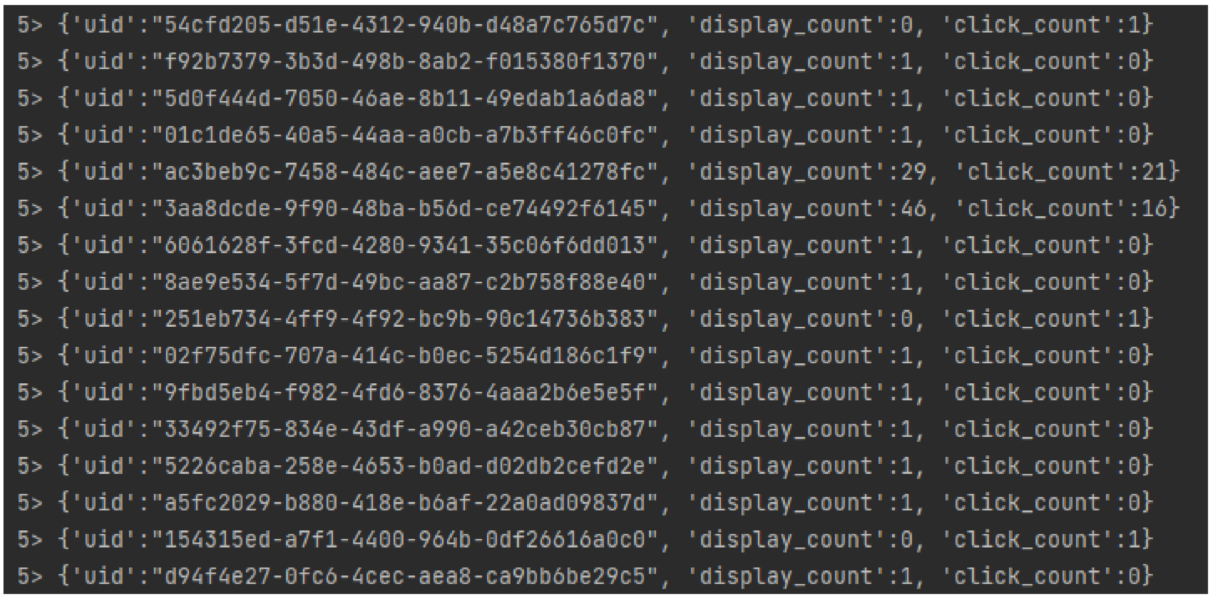


```
5> {'fraud_type': fraudulent_IP, 'IP': 238.186.83.58 , 'click_count': 120}
5> {'fraud_type': fraudulent_IP, 'IP': 238.186.83.58 , 'click_count': 240}
5> {'fraud_type': fraudulent_IP, 'IP': 238.186.83.58 , 'click_count': 240}
5> {'fraud_type': fraudulent_IP, 'IP': 238.186.83.58 , 'click_count': 240}
```

Filter 2 output : high click per minute rate for an IP

Pattern 3 : Clicks without display ?

When exploring our data, we saw that some user ID had a click event but without a display event.



```
5> {'uid':"54cfd205-d51e-4312-940b-d48a7c765d7c", 'display_count':0, 'click_count':1}
5> {'uid':"f92b7379-3b3d-498b-8ab2-f015380f1370", 'display_count':1, 'click_count':0}
5> {'uid':"5d0f444d-7050-46ae-8b11-49edab1a6da8", 'display_count':1, 'click_count':0}
5> {'uid':"01c1de65-40a5-44aa-a0cb-a7b3ff46c0fc", 'display_count':1, 'click_count':0}
5> {'uid':"ac3beb9c-7458-484c-aee7-a5e8c41278fc", 'display_count':29, 'click_count':21}
5> {'uid':"3aa8dcde-9f90-48ba-b56d-ce74492f6145", 'display_count':46, 'click_count':16}
5> {'uid':"6061628f-3fcd-4280-9341-35c06f6dd013", 'display_count':1, 'click_count':0}
5> {'uid':"8ae9e534-5f7d-49bc-aa87-c2b758f88e40", 'display_count':1, 'click_count':0}
5> {'uid':"251eb734-4ff9-4f92-bc9b-90c14736b383", 'display_count':0, 'click_count':1}
5> {'uid':"02f75dfc-707a-414c-b0ec-5254d186c1f9", 'display_count':1, 'click_count':0}
5> {'uid':"9fbd5eb4-f982-4fd6-8376-4aaa2b6e5e5f", 'display_count':1, 'click_count':0}
5> {'uid':"33492f75-834e-43df-a990-a42ceb30cb87", 'display_count':1, 'click_count':0}
5> {'uid':"5226caba-258e-4653-b0ad-d02db2cefd2e", 'display_count':1, 'click_count':0}
5> {'uid':"a5fc2029-b880-418e-b6af-22a0ad09837d", 'display_count':1, 'click_count':0}
5> {'uid':"154315ed-a7f1-4400-964b-0df26616a0c0", 'display_count':0, 'click_count':1}
5> {'uid':"d94f4e27-0fc6-4cec-aea8-ca9bb6be29c5", 'display_count':1, 'click_count':0}
```

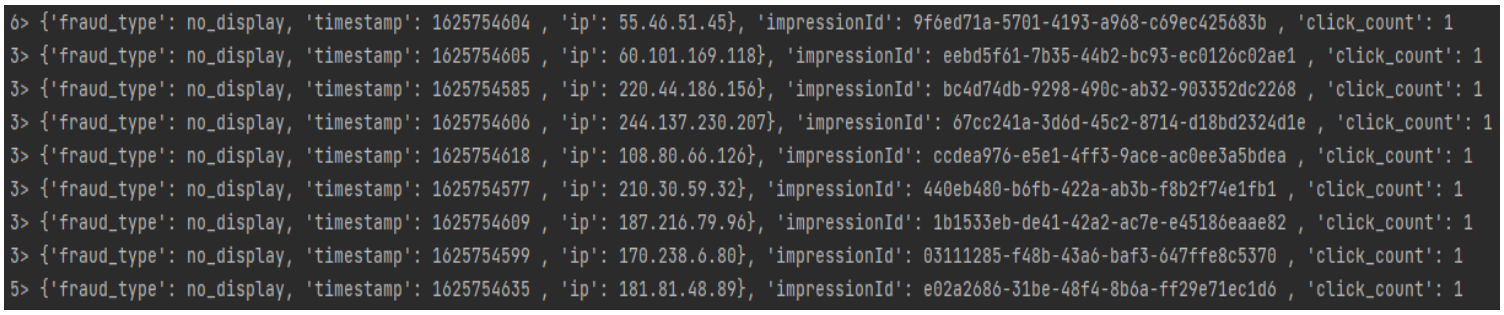
User ID with one click but no display

At first we thought that this would be a fraudulent event, however we concluded that this behaviour should not be made possible, and that a click should be guaranteed to come after a display by the system. We inferred this phenomenon to be due to the display event being out of the studied window, as this phenomenon impacts a lot of different users, impressions and IPs. However, if this phenomenon is not due to the windowing, it is likely to be a fraud. Conclusion on this matter would be easier with a field expertise.

As these events are likely to add noise, we set them apart :

```
// FRAUD PATTERN 3 : FILTER EVENTS WITHOUT DISPLAY
// FORM WINDOW KEYED ON UID
val overused_banner = dataStream
 // keep ('uid', 'display_count', 'click_count', 'timestamp', 'ip',
'impressionId')
 .map(x => (x._1, x._4, x._5, x._6, x._2, x._3))
 .keyBy(x => (x._1)) // key on uid
 .window(SlidingEventTimeWindows.of(Time.minutes(windowLength),
Time.minutes(windowDelay)))
 .reduce( (a, b) => (a._1, a._2+b._2, a._3+b._3, a._4, a._5, a._6) )
 .filter(x => x._2 == 0)
 .map(x => f"{'fraud_type': no_display, 'timestamp': ${x._4} , 'ip':
${x._5}}, 'impressionId': ${x._6} , 'click_count': ${x._3}")
 .addSink(fraudFilter)
```

As seen previously, the events filtered are very different and cannot really be linked together. Here we aggregated the data on the uid but we tried to aggregate on the timestamp, impression ID and IP with similar results.



```
6> {'fraud_type': no_display, 'timestamp': 1625754604 , 'ip': 55.46.51.45}, 'impressionId': 9f6ed71a-5701-4193-a968-c69ec425683b , 'click_count': 1
3> {'fraud_type': no_display, 'timestamp': 1625754605 , 'ip': 60.101.169.118}, 'impressionId': eebd5f61-7b35-44b2-bc93-ec0126c02ae1 , 'click_count': 1
3> {'fraud_type': no_display, 'timestamp': 1625754585 , 'ip': 220.44.186.156}, 'impressionId': bc4d74db-9298-490c-ab32-903352dc2268 , 'click_count': 1
3> {'fraud_type': no_display, 'timestamp': 1625754606 , 'ip': 244.137.230.207}, 'impressionId': 67cc241a-3d6d-45c2-8714-d18bd2324d1e , 'click_count': 1
3> {'fraud_type': no_display, 'timestamp': 1625754618 , 'ip': 108.80.66.126}, 'impressionId': ccdea976-e5e1-4ff3-9ace-ac0ee3a5bdea , 'click_count': 1
3> {'fraud_type': no_display, 'timestamp': 1625754577 , 'ip': 210.30.59.32}, 'impressionId': 440eb480-b6fb-422a-ab3b-f8b2f74e1fb1 , 'click_count': 1
3> {'fraud_type': no_display, 'timestamp': 1625754609 , 'ip': 187.216.79.96}, 'impressionId': 1b1533eb-de41-42a2-ac7e-e45186eaae82 , 'click_count': 1
3> {'fraud_type': no_display, 'timestamp': 1625754599 , 'ip': 170.238.6.80}, 'impressionId': 03111285-f48b-43a6-baf3-647ffe8c5370 , 'click_count': 1
5> {'fraud_type': no_display, 'timestamp': 1625754635 , 'ip': 181.81.48.89}, 'impressionId': e02a2686-31be-48f4-8b6a-ff29e71ec1d6 , 'click_count': 1
```

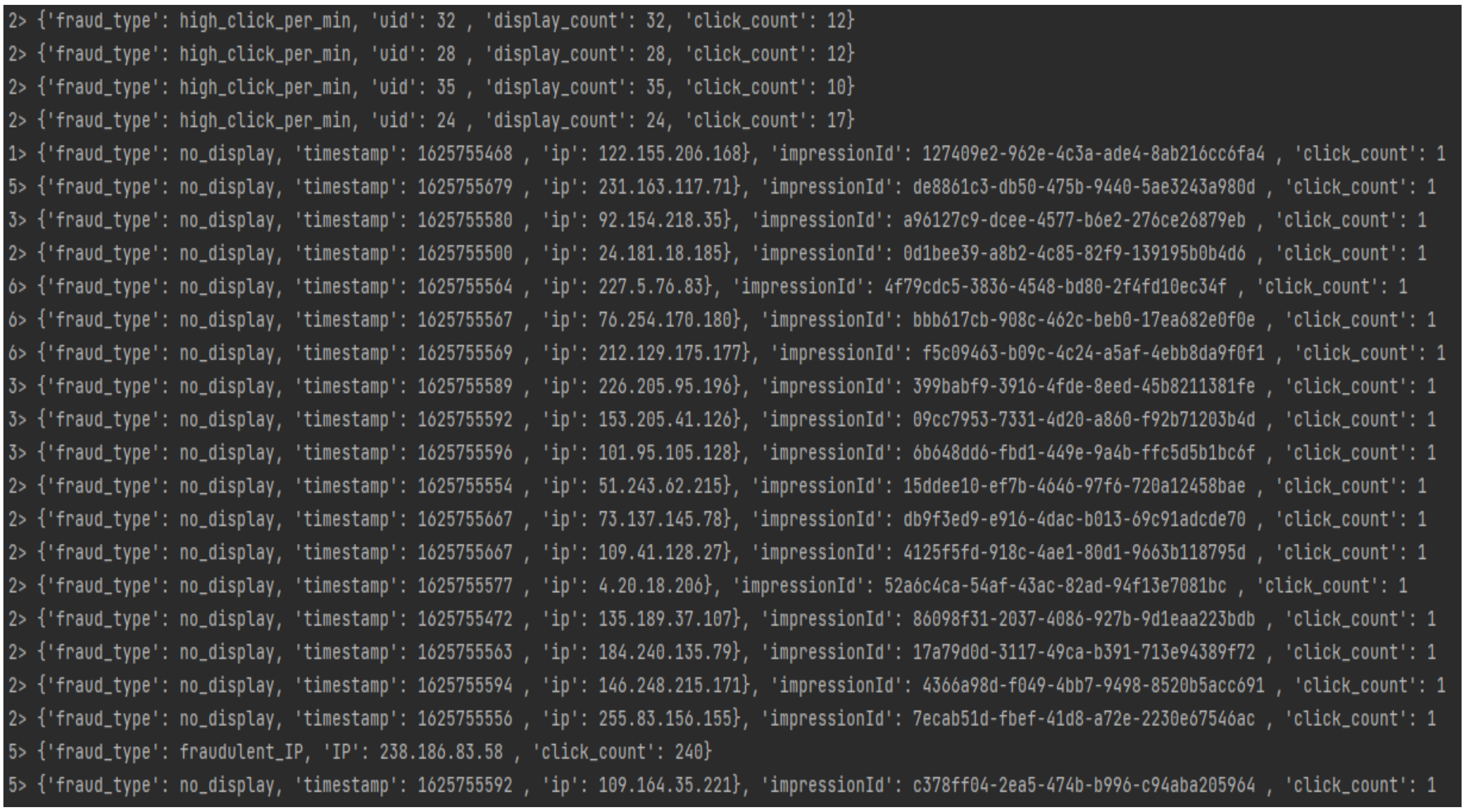
User ID with one click but no display, no similarity between events

Other analysis

Apart from the previous analysis we also looked if among the events with the same impression ID they were some abnormal data, as well for events with the same timestamp. Apart from the pattern previously studied, we didn't find any other abnormal data.

Assembling patterns

After sending the filtered data to the sink, the data can be streamed from a new topic named 'fraudulentEvent'. In order to do so we had to slightly change the docker-compose file to create this topic.



```
2> {'fraud_type': high_click_per_min, 'uid': 32 , 'display_count': 32, 'click_count': 12}
2> {'fraud_type': high_click_per_min, 'uid': 28 , 'display_count': 28, 'click_count': 12}
2> {'fraud_type': high_click_per_min, 'uid': 35 , 'display_count': 35, 'click_count': 10}
2> {'fraud_type': high_click_per_min, 'uid': 24 , 'display_count': 24, 'click_count': 17}
1> {'fraud_type': no_display, 'timestamp': 1625755468 , 'ip': 122.155.206.168}, 'impressionId': 127409e2-962e-4c3a-ade4-8ab216cc6fa4 , 'click_count': 1
5> {'fraud_type': no_display, 'timestamp': 1625755679 , 'ip': 231.163.117.71}, 'impressionId': de8861c3-db50-475b-9440-5ae3243a980d , 'click_count': 1
3> {'fraud_type': no_display, 'timestamp': 1625755580 , 'ip': 92.154.218.35}, 'impressionId': a96127c9-dcee-4577-b6e2-276ce26879eb , 'click_count': 1
2> {'fraud_type': no_display, 'timestamp': 1625755500 , 'ip': 24.181.18.185}, 'impressionId': 0d1bee39-a8b2-4c85-82f9-139195b0b4d6 , 'click_count': 1
6> {'fraud_type': no_display, 'timestamp': 1625755564 , 'ip': 227.5.76.83}, 'impressionId': 4f79cdc5-3836-4548-bd80-2f4fd10ec34f , 'click_count': 1
6> {'fraud_type': no_display, 'timestamp': 1625755567 , 'ip': 76.254.170.180}, 'impressionId': bbb617cb-908c-462c-beb0-17ea682e0f0e , 'click_count': 1
6> {'fraud_type': no_display, 'timestamp': 1625755569 , 'ip': 212.129.175.177}, 'impressionId': f5c09463-b09c-4c24-a5af-4ebb8da9f0f1 , 'click_count': 1
3> {'fraud_type': no_display, 'timestamp': 1625755589 , 'ip': 226.205.95.196}, 'impressionId': 399babf9-3916-4fde-8eed-45b8211381fe , 'click_count': 1
3> {'fraud_type': no_display, 'timestamp': 1625755592 , 'ip': 153.205.41.126}, 'impressionId': 09cc7953-7331-4d20-a860-f92b71203b4d , 'click_count': 1
3> {'fraud_type': no_display, 'timestamp': 1625755596 , 'ip': 101.95.105.128}, 'impressionId': 6b648dd6-fbd1-449e-9a4b-ffc5d5b1bc6f , 'click_count': 1
2> {'fraud_type': no_display, 'timestamp': 1625755554 , 'ip': 51.243.62.215}, 'impressionId': 15ddee10-ef7b-4646-97f6-720a12458bae , 'click_count': 1
2> {'fraud_type': no_display, 'timestamp': 1625755667 , 'ip': 73.137.145.78}, 'impressionId': db9f3ed9-e916-4dac-b013-69c91adcde70 , 'click_count': 1
2> {'fraud_type': no_display, 'timestamp': 1625755667 , 'ip': 109.41.128.27}, 'impressionId': 4125f5fd-918c-4ae1-80d1-9663b118795d , 'click_count': 1
2> {'fraud_type': no_display, 'timestamp': 1625755577 , 'ip': 4.20.18.206}, 'impressionId': 52a6c4ca-54af-43ac-82ad-94f13e7081bc , 'click_count': 1
2> {'fraud_type': no_display, 'timestamp': 1625755472 , 'ip': 135.189.37.107}, 'impressionId': 86098f31-2037-4086-927b-9d1eaa223bdb , 'click_count': 1
2> {'fraud_type': no_display, 'timestamp': 1625755563 , 'ip': 184.240.135.79}, 'impressionId': 17a79d0d-3117-49ca-b391-713e94389f72 , 'click_count': 1
2> {'fraud_type': no_display, 'timestamp': 1625755594 , 'ip': 146.248.215.171}, 'impressionId': 4366a98d-f049-4bb7-9498-8520b5acc691 , 'click_count': 1
2> {'fraud_type': no_display, 'timestamp': 1625755556 , 'ip': 255.83.156.155}, 'impressionId': 7ecab51d-fbef-41d8-a72e-2230e67546ac , 'click_count': 1
5> {'fraud_type': fraudulent_IP, 'IP': 238.186.83.58 , 'click_count': 240}
5> {'fraud_type': no_display, 'timestamp': 1625755592 , 'ip': 109.164.35.221}, 'impressionId': c378ff04-2ea5-474b-b996-c94aba205964 , 'click_count': 1
```

Assembled stream of the fraudulent event

Conclusion

In this project we used Flink to monitor the display and click events on ads. This allowed us to analyze and filter out fraudulent data. After setting aside the fraudulent data, we resend them to a new topic, allowing access to this stream to other users !