Design a streaming system using PySpark to perform different data aggregation analyses using SQL queries for any given data Report

BY Pratishtha Sharma (IDS2022004) and Gomtesh Jain (IDS2022006)

Table Of Contents

About the data

Paysim synthetic dataset of mobile money transactions. Each step represents an hour of simulation. This dataset is scaled down 1/4 of the original dataset which is presented in the paper "PaySim: A financial mobile money simulator for fraud detection".

We have downloaded the dataset from kaggle

1. Step - Maps a unit of time in the real world. In this case 1 step is 1 hour of time.
2. Type - CASH-IN, CASH-OUT, DEBIT, PAYMENT and TRANSFER
3. Amount - amount of the transaction in local currency
4. NameOrig - customer who started the transaction
5. oldbalanceOrg- initial balance before the transaction
6. newbalanceOrig - customer's balance after the transaction.
7. nameDest - recipient ID of the transaction.
8. oldbalanceDest - initial recipient balance before the transaction.
9. newbalanceDest - recipient's balance after the transaction.
10. IsFraud - identifies a fraudulent transaction (1) and non fraudulent (0)
11. IsFlaggedFraud- flags illegal attempts to transfer more than 200.000 in a single transaction.

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

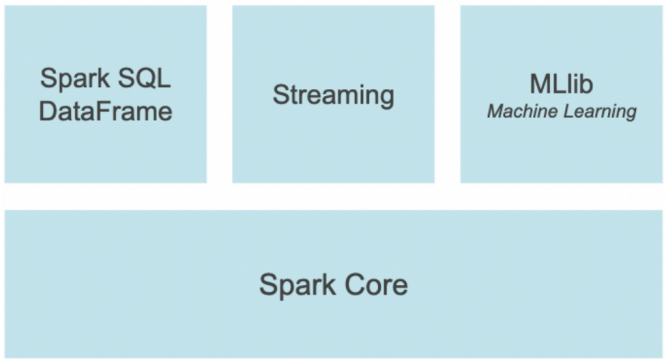
Shape of Data - (6362620, 11)

there is **no missing** and **garbage value**, there is no need for data cleaning

Technologies used are

PySpark -

PySpark is an interface for Apache Spark in Python. It not only allows you to write Spark applications using Python APIs, but also provides the PySpark shell for interactively analyzing your data in a distributed environment. PySpark supports most of Spark's features such as Spark SQL, DataFrame, Streaming, MLlib (Machine Learning) and Spark Core.



Spark SQL and DataFrame

Spark SQL is a Spark module for structured data processing. It provides a programming abstraction called DataFrame and can also act as distributed SQL query engine.

pandas API on Spark

pandas API on Spark allows you to scale your pandas workload out. With this package, you can:

Be immediately productive with Spark, with no learning curve, if you are already familiar with pandas.

Have a single codebase that works both with pandas (tests, smaller datasets) and with Spark (distributed datasets).

Switch to pandas API and PySpark API contexts easily without any overhead.

Streaming

Running on top of Spark, the streaming feature in Apache Spark enables powerful interactive and analytical applications across both streaming and historical data, while inheriting Spark's ease of use and fault tolerance characteristics.

MLlib

Built on top of Spark, MLlib is a scalable machine learning library that provides a uniform set of high-level APIs that help users create and tune practical machine learning pipelines.

Spark Core

Spark Core is the underlying general execution engine for the Spark platform that all other functionality is built on top of. It provides an RDD (Resilient Distributed Dataset) and in-memory computing capabilities.

Python-

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and

dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

Pandas-

Pandas is a Python library used for working with data sets.It has functions for analyzing, cleaning, exploring, and manipulating data.The name "Pandas" has a reference to both "Panel Data", and "Python Data Analysis" and was created by Wes McKinney in 2008.

Pandas allows us to analyze big data and make conclusions based on statistical theories.Pandas can clean messy data sets, and make them readable and relevant.Relevant data is very important in data science.

SQL-

SQL is a database computer language designed for the retrieval and management of data in a relational database. SQL stands for Structured Query Language.

SQL is Structured Query Language, which is a computer language for storing, manipulating and retrieving data stored in a relational database.

SQL is the standard language for Relational Database System. All the Relational Database Management Systems (RDMS) like MySQL, MS Access, Oracle, Sybase, Informix, Postgres and SQL Server use SQL as their standard database language.

Also, they are using different dialects, such as −

- MS SQL Server using T-SQL,
- Oracle using PL/SQL,
- MS Access version of SQL is called JET SQL (native format) etc

Aggregate Functions

SQL aggregate functions collate information about what is in a database. For instance, you can use SUM to find the total of all the values in a column. Aggregate functions save you time when you need to find out information that involves aggregating records.,

Here is a list of the aggregate functions in SQL you can use:

- COUNT
- SUM
- AVG
- MIN
- MAX

SQL COUNT

The SQL COUNT function returns the total number of rows returned by a query. Using a WHERE statement, the COUNT function returns the number of rows that meet your condition.

```
SELECT COUNT(name) FROM employees WHERE branch = "Stamford";
```

Our query returns the number of employees working at the Stamford branch:

count
1

(1 row)

SQL MIN and MAX

The SQL MIN function returns the smallest value within a column. An SQL MAX statement returns the largest value in a column. Both of these statements are SQL aggregate functions.

For example, say you want to get the lowest number of employee of the month awards held by a single person. We could retrieve this data using this query:

```
SELECT MIN(employee_month_awards) FROM employees;
```

Our query returns:

min
1

```
SELECT MAX(employee_month_awards) FROM employees;
```

The output for our query is as follows:

max
6

SQL AVG

The SQL AVG function returns the average value of a particular column.

Let's say we want to get the average number of employee of the month awards held by each employee. We would use the following query to accomplish this goal:

```
SELECT AVG(employee_month_awards) FROM employees;
```

avg
4

SQL SUM

The SQL SUM function finds the total sum of a particular column.

```
SELECT SUM(employee_month_awards) FROM employees;
```

Our query returns the following:

sum
20

All About Streaming ..

Also known as event stream processing, streaming data is the continuous flow of data generated by various sources. By using stream processing technology, data streams can be processed, stored, analyzed, and acted upon as it's generated in real-time.

The term "streaming" is used to describe continuous, never-ending data streams with no beginning or end, that provide a constant feed of data that can be utilized/acted upon without needing to be downloaded first.

Similarly, data streams are generated by all types of sources, in various formats and volumes. From applications, networking devices, and server log files, to website activity, banking transactions, and location data, they can all be aggregated to seamlessly gather real-time information and analytics from a single source of truth.

Overview of Stream Data Processing

Today's data is generated by an infinite amount of sources - IoT sensors, servers, security logs, applications, or internal/external systems. It’s almost impossible to regulate structure, data integrity, or control the volume or velocity of the data generated.

While traditional solutions are built to ingest, process, and structure data before it can be acted upon, streaming data architecture adds the ability to consume, persist to storage, enrich, and analyze data in motion.

As such, applications working with data streams will always require two main functions: storage and processing. Storage must be able to record large streams of data in a way that is sequential and consistent. Processing must be able to interact with storage, consume, analyze and run computation on the data.

This also brings up additional challenges and considerations when working with legacy databases or systems. Many platforms and tools are now available to help companies build streaming data applications.

Examples

Some real-life examples of streaming data include use cases in every industry, including real-time stock trades, up-to-the-minute retail inventory management, social media feeds, multiplayer game interactions, and ride-sharing apps.

For example, when a passenger calls Lyft, real-time streams of data join together to create a seamless user experience. Through this data, the application pieces together real-time location tracking, traffic stats, pricing, and real-time traffic data to simultaneously match the rider with the best possible driver, calculate pricing, and estimate time to destination based on both real-time and historical data.

In this sense, streaming data is the first step for any data-driven organization, fueling big data ingestion, integration, and real-time analytics.

Batch Processing vs Real-Time Streams

Batch data processing methods require data to be downloaded as batches before it can be processed, stored, or analyzed, whereas streaming data flows in continuously, allowing that data to be processed simultaneously, in real-time the second it's generated.

Today, data arrives naturally as never ending streams of events. This data comes in all volumes, formats, from various locations and cloud, on-premises, or hybrid cloud.

With the complexity of today's modern requirements, legacy data processing methods have become obsolete for most use cases, as it can only process data as groups of transactions collected over time. Modern organizations need to act on up-to-the-millisecond data, before the data becomes stale. This continuous data offers numerous advantages that are transforming the way businesses run.

Benefits of Streaming Data

Data collection is only one piece of the puzzle. Today's enterprise businesses simply cannot wait for data to be processed in batch form. Instead, everything from fraud detection and stock market platforms, to ride share apps and e-commerce websites rely on real-time event streams.

Paired with streaming data, applications evolve to not only integrate data, but process, filter, analyze, and react to event as they happen in real-time. This opens a new plethora of use cases such as real-time fraud detection, Netflix recommendations, or a seamless shopping experience across multiple devices that updates as you shop.

In short, any industry that deals with large volumes of real-time data can benefit from continuous, real-time event stream processing platforms.

Use Cases

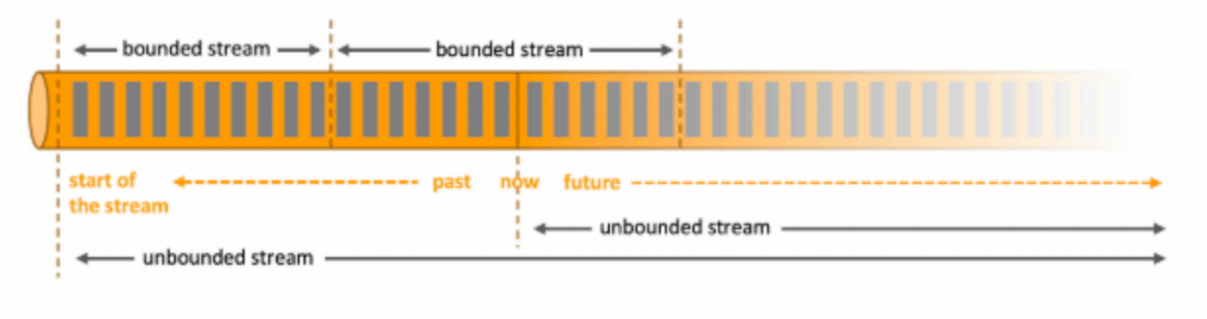
Stream processing systems like Apache Kafka and Confluent bring real-time data and analytics to life. While there are use cases for event streaming in every industry, this ability to integrate, analyze, troubleshoot, and/or predict data in real-time, at massive scale, opens up new use cases. Not only can organizations use past data or batch data in storage, but gain valuable insights on data in motion.

Typical uses cases include:

- Location data
- Fraud detection
- Real-time stock trades
- Marketing, sales, and business analytics
- Customer/user activity
- Monitoring and reporting on internal IT systems
- Log Monitoring: Troubleshooting systems, servers, devices, and more
- SIEM (Security Information and Event Management): analyzing logs and real-time event data for monitoring, metrics, and threat detection
- Retail/warehouse inventory: inventory management across all channels and locations, and providing a seamless user experience across all devices

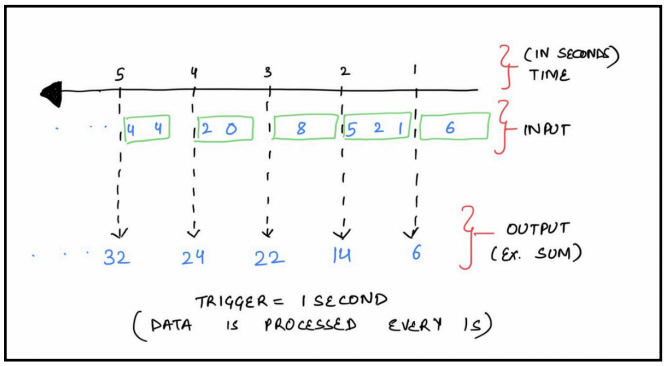
- Ride share matching: Combining location, user, and pricing data for predictive analytics - matching riders with the best drivers in term of proximity, destination, pricing, and wait times
- Machine learning and A.I.: By combining past and present data for one central nervous system, this brings new possibilities for predictive analytics

Types of Data Streaming

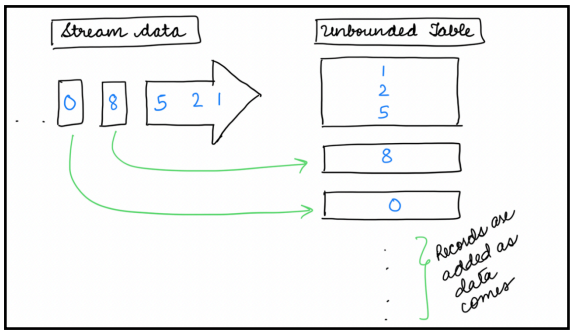


Bounded vs Unbounded Streams

The data can be processed to get better insights for decision making. The data can be processed either as a bounded stream or an unbounded stream.



1. Bounded Stream



The bounded stream will have a defined start and an end. While processing a bounded stream, we can ingest the entire data-set before starting any computation. Hence, we could perform operations

such as sorting and summarize data. The processing of bounded stream is referred to as Batch processing.

2. Unbounded Stream

The unbounded stream will have a start but no end. Hence, the data needs to be continuously processed when it is generated. Data is processed based on the event time (Event time is the time that each individual event occurred on its producing device). The paradigm of processing unbounded stream is referred to as Stream processing.

Structured streaming

Stream data processing

Processing of data as and when it comes to make near real time decisions.

For example, Fraud detection as and when it happens, detection of an erroneous server by analyzing the error rate, etc.

How does stream processing differ from batch processing?

Batch data processing is processing of data accumulated over a period of time.

These are the normal flows/jobs that you have running say at daily, weekly, or twice a day frequency. No matter when the data comes, it will always be processed at fixed defined intervals.

Another difference I want to highlight here is that in terms of reliability, batch processing is always more reliable than aggregates generated via stream processing, especially if the stream processing is not configured properly, but stream processes allow for quicker interpretation of data as compared to batch processing.

Another general difference is stream processing generally takes less memory as compared to batch processing since, at a time, you are processing fewer data.

Spark Structured Streaming

Spark structured streaming allows for near-time computations of streaming data over Spark SQL engine to generate aggregates or output as per the defined logic.

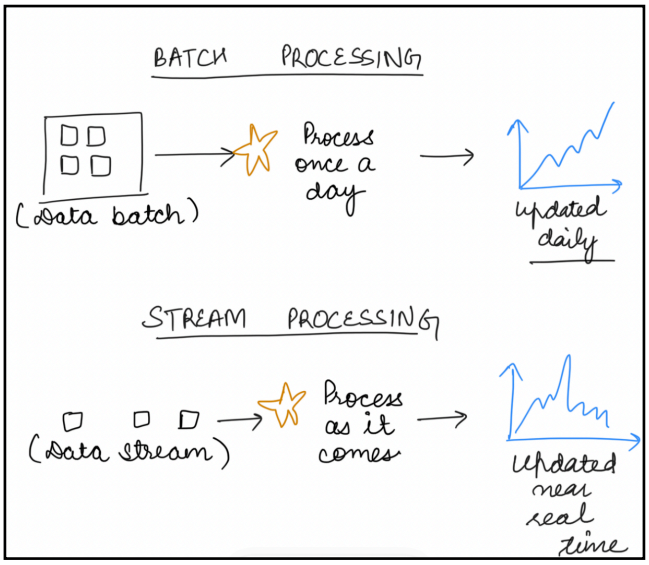
This streaming data can be read from a file, a socket, or sources such as Kafka.

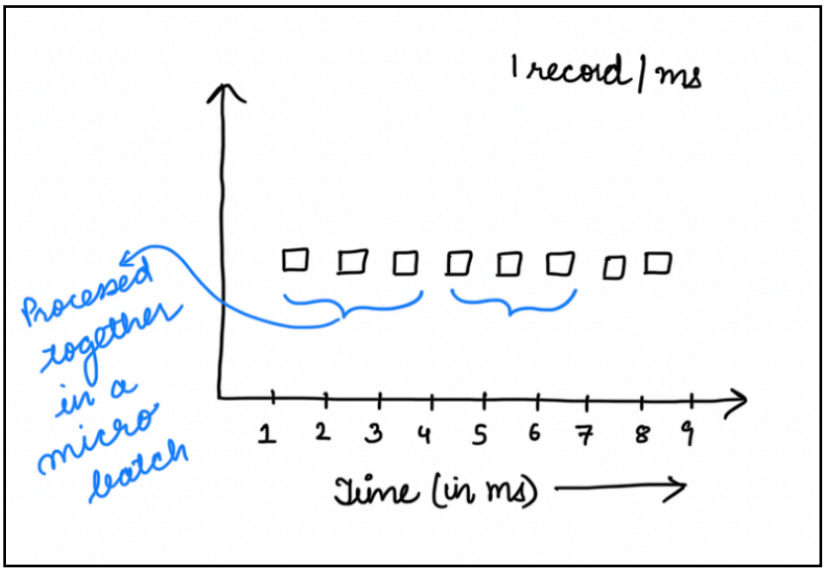
And the super cool thing about this is that the core logic of the implementation for processing is very closely related to how you would process that data in batch mode. Basically, you can define data frames and work with them how you normally do while writing a batch job, but the processing of data differs.

One thing important to note here is structured streaming does not process the data in real-time but instead in near-real-time. In fact, rarely will you find a pipeline or a system, that processes data in “real” real-time without any delays, but well that’s a separate discussion.

Micro batches: Structured Streaming

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



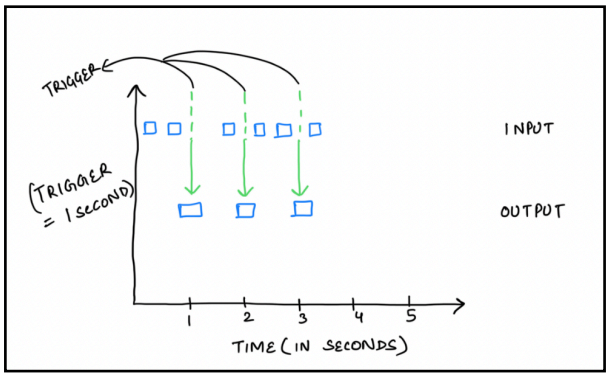


Triggers

Now how does Spark knows when to generate these micro-batches and append them to the unbounded table?

This mechanism is called triggering. As explained, not every record is processed as it comes, at a certain interval, called the “trigger” interval, a micro-batch of rows gets appended to the table and

gets processed. This interval is configurable, and has different modes, by default mode is start the next process as previous finishes.



Streaming Windows

fixed/tumbling:

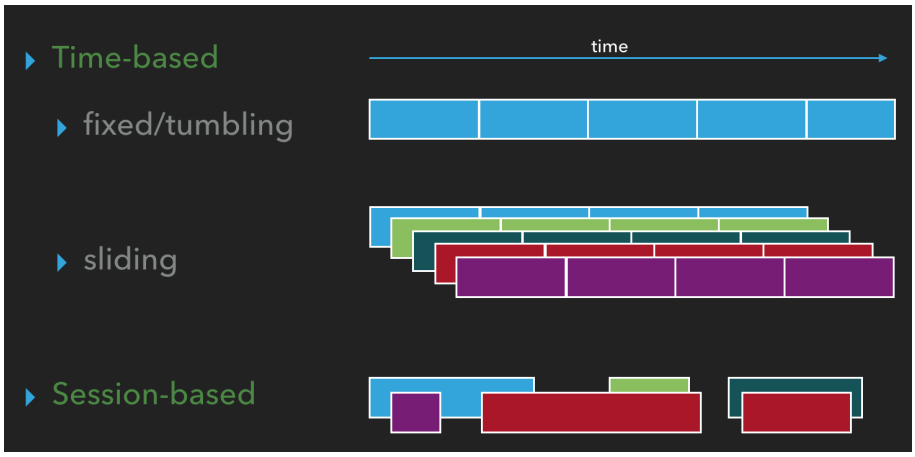
time is partitioned into same-length, non-overlapping chunks. Each event belongs to exactly one window

sliding:

windows have fixed length, but are separated by a time interval (step) which can be smaller than the window length. Typically the window interval is a multiplicity of the step. Each event belongs to a number of windows ([window interval]/[window step]).

session:

windows have various sizes and are defined basing on data, which should carry some session identifiers



About the Project

Initially we had taken a Bank data of fraudulent transaction detection through credit cards which has around 63 Lakhs rows and 11 column and size is around 500 Mb. and divided them according to time stamps into various of .csv files which were around 800 in number using the GROUP BY function of SQL on Step Column. i.e. Maps a unit of time in the real world. In this case 1 step is 1 hour of time.

Group by function help us to identify steps and its count which means number of rows or number of transitions on that particular step.

We initially had 11 columns out of which 2 are found to be redundant which were removed during the data cleaning process to avoid crowding.

In these particular files, records of data which only carried one transaction in a time stamp were further removed to further remove redundancy.

After this step we had around 63 lakh rows and 9 columns.

After that our main objective that was to design a streaming system using pyspark and perform aggregation analytics of SQL.

In this structured streaming, we have used a function called as “maxFilesPerTrigger” and set it value to 1 but its value is variable and since we were getting best results keeping the value equal to 1 .

We have used various kinds of streaming windows on self-made data set ans check the difference in results.

CODE

1. We Install the needful packages

```
!pip install pyspark
```

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

```
import pandas as pd
```

2. We import needful lib

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

```
22/11/14 12:55:19 WARN Utils: Your hostname, Gomteshs-MacBook-Pro.local resolves to a loopback address: 127.0.0.1;
using 172.27.12.20 instead (on interface en0)
22/11/14 12:55:19 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
```

```
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
```

```
22/11/14 12:55:19 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-ja
va classes where applicable
```

3. Read the CSV file

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

4. Drop unwanted columns and print the data

```
df.columns
```

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

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

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

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

5. Print group by on Step with count of transactions in that each step

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

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

```
[Stage 12:>
```

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

6. In this particular step , we have to break the csv into multiple csv which is broken on basis of different time

   a. We can therefore save the output of that job by filtering on each step and saving it to a separate file

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

7. Print about csv names and 1 csv file

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

```
part.show()
```

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

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

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

8. Explain streaming function

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

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

    c. maxFilesPerTrigger allows you to control how quickly Spark will read all of the files in the folder.In this example we're limiting the flow of the stream to one file per trigger.

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

9. Code of Count

    a. Let's set up a transformation.

    b. apply aggregate function on amount

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

    c. Now that we have our transformation,we need to specify an output sink for the results.

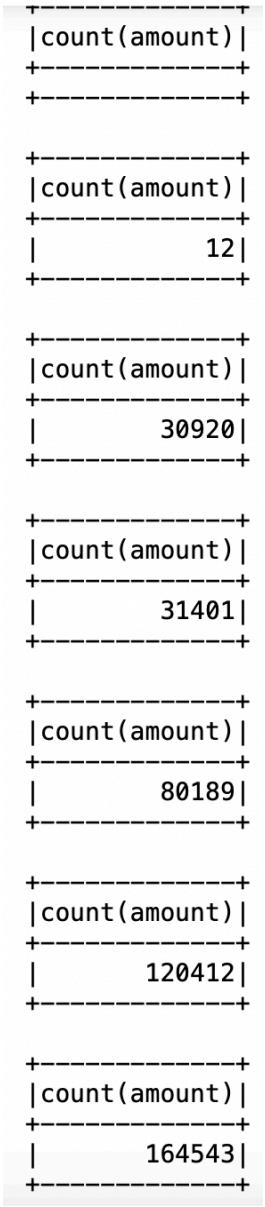
    d. For this example,we're going to write to a memory sink which keeps the results in memory. We also need to define how Spark will output that data. In this example,we'll use the complete output mode(rewriting all of the keys along with their counts after every trigger).

    e. In this example we won't include activityQuery.await Termination()because it is required only to prevent the driver process from terminating when the stream is active. So in order to be able to run this locally in a notebook we won't include it.

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

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

f. Output of the code



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

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

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

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

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

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

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

10. Code of AVG

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

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

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

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

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

11. Code of max

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

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

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

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

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

12. Code of min

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

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

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

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

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

13. Code of sum

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

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

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

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

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

14. Stoping the streaming

Check if stream is active

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

```
activityQuery.status
```

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

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

```
activityQuery.stop()
```

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

15. Window

    a. Created a data with column name event id and time received in which event_id is int and timeReceived is in DateTime.

    b. We have created this to check behaviour of different windows

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

16. Tumbling window

```
from pyspark.sql.functions import *

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

tumblingWindows.show(truncate = False)
```

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

17. Sliding window

```
from pyspark.sql.functions import *

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

slidingWindows.show(truncate = False)
```

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

18. Session window

```
from pyspark.sql.functions import *

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

sessionWindows.show(truncate = False)
```

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

Conclusion

So in this project we have converted Static data into streaming data using the streaming function and then applying SQL through various aggregate functions to perform aggregation analytics.

In this project we have converted time date column into a streaming event using different types of windows such as fixed, sliding , session.

Future Scope

Apache Hadoop has been the foundation for big data applications for a long time now, and is considered the basic data platform for all big-data-related offerings. However, in-memory database and computation is gaining popularity because of faster performance and quick results. Apache Spark is a new framework which utilizes in-memory capabilities to deliver fast processing (almost 100 times faster than Hadoop). So , using Spark we can performing Data Analysis and later on for future prospects we can use machine learning algorithms to make this data streaming process more fool proof when it comes to Fraudulent Transaction Detection of Credit cards which is the basis of data we have used in data analysis.

REFERENCES

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