# COMP47470 : Big Data Programming
# Project 3
### -Sai Patil (19200089)
## Introduction:

This project is divided into two sections:

**1) Spark Streaming**

In this exercise you will Spark Streaming to manipulate a stream of data representing taxi trips in New York City. This data is part of a dataset released by the New York City Taxi & Limousine Commission that captures over one billion individual taxi trips, but we will only be using one day's worth of data.

The dataset is in the form of CSV les that contain data about two types of trips: green and yellow taxi rides. The lines for each type of ride have a different schema shown below. For both types, the rst column contains the type, "yellow" or "green".

The schema for **yellow taxi rides** is as follows:

type, VendorID, tpep pickup ,datetime, tpep drop off datetime, passenger count, Trip distance, pickup longitude, pickup latitude, RatecodeID, store and fwd flag, Drop Off longitude, drop off latitude, payment type, fare amount, extra, mta tax, tip amount, tolls amount, improvement surcharge, total amount

The schema for **green taxi rides** is as follows:

type, VendorID, lpep pickup datetime, Lpep dropoff datetime, Store and fwd flag, RateCodeID, Pickup longitude, Pickup latitude, Dropoff longitude, Dropoff latitude, Passenger count, Trip distance, Fare amount, Extra, MTA tax, Tip amount, Tolls amount, Ehail fee, improvement surcharge, Total amount, Payment type, Trip type

**2) Reflection**

This section summarizes one of the research papers uploaded on the brightspace. I have summarized the paper “ **Discretized Streams: An Efficient and Fault-Tolerant Model forStream Processing on Large Clusters”.**

**Section 1 : Spark Streaming**
**1.3 Tasks**
**1.**

```
val wc = stream.map(_.split(","))
       .map(tuple => ("all", 1))
       .reduceByKeyAndWindow(
         (x: Int, y: Int) => x + y, (x: Int, y: Int) => x - y,
         ↪  Minutes(60), Minutes(60))
       .persist()
```

**(a) Explain what this snippet of code does. What does the output we get represent?**
Let's take a look at the functions used in this snippet
**stream.map-** returns a stream consisting of the results of applying the given function to the elements of this stream.
**.reduceByKeyAndWindow-** When called on a DStream of (K, V) pairs, returns a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function *func* over batches in a sliding window.
**.persist() -** It is the optimization technique for interactive and iterative Spark computations. It helps to save intermediate results so we can reuse them in subsequent stages. These intermediate results as RDDs are thus kept in memory (default) or more solid storages like disk and/or replicated. Benefits- Time efficient, Cost efficient, Lessen the execution time.
**This code snippet is like a map reduce job. We match each split followed by a comma. Post that each line is counted and passed on to the reduce function. The reduceByKeyAndWindow () generated taxi counts over the last 60 minutes of data and this function is repeated every hour (60 minutes).**

**(b) Could something be taken out of this code without modifying the result? If so, what and why**
**Ans.** Yes,if we remove persist() the output will still be the same.
The persist() on a DStream will persist every RDD of that DStream in the memory. This is useful when the data in the DStream is computed multiple times. For window based operations reduceByKeyAndWindow, this is implicitly true which means DStreams generated by window based operations are automatically persisted in the memory. There is no need to explicitly call the persist().

Task 1 Ques 2) I have only mentioned a few of the results.

Goldman

```
21600000:(goldman,3)
25200000:(goldman,11)
28800000:(goldman,17)
32400000:(goldman,25)
36000000:(goldman,39)
39600000:(goldman,26)
43200000:(goldman,16)
46800000:(goldman,7)
50400000:(goldman,11)
54000000:(goldman,12)
57600000:(goldman,14)
61200000:(goldman,5)
64800000:(goldman,1)
68400000:(goldman,0)
72000000:(goldman,1)
```

Citigroup

```
3600000:(citigroup,5)
7200000:(citigroup,2)
10800000:(citigroup,1)
14400000:(citigroup,0)
18000000:(citigroup,1)
21600000:(citigroup,8)
25200000:(citigroup,46)
28800000:(citigroup,62)
32400000:(citigroup,56)
36000000:(citigroup,60)
39600000:(citigroup,18)
43200000:(citigroup,17)
46800000:(citigroup,24)
50400000:(citigroup,25)
54000000:(citigroup,35)
```

**Task 1 Ques 3:** Check events.scala file for reference

## Section 2: Reflection - **Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters** by Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, Ion Stoica from University of California, Berkeley

**Question/Challenge Addressed by the research paper:**

This paper Addresses the Challenges of Distributed Stream Processing over Large clusters. Current programming models for distributed stream processing are relatively low-level, leaving the user to worry about the consistency of state across the system and fault recovery.

Much of real-life "big data" is received in real-time and is most
valuable at its time of arrival. To handle the volumes of data and computation they involve, these applications need to be distributed over clusters. Most current distributed stream processing systems, are based on a record-at-a-time processing model, where nodes receive each
record, update the internal state, and send out new records in response.

Several Challenges are faced by the user due to these models. Such as:

- Fault Tolerance: Record-at-a-time systems provide recovery through replication or upstream-backup; both of which are unsuitable and slow for large clusters.
- Consistency: Different nodes may be processing data that arrived at different times which makes it harder for the user to determine the consistency of the system.
- Unification of batches: Two versions of each analytics task has to be written for streaming and batch systems API. Also, It is difficult to combine streaming data with historical data.

**Related Work:**

The seminal academic work on stream processing was in streaming databases such as Aurora, Borealis, Telegraph, and STREAM. These systems provide a SQL interface and achieve fault recovery through either replication or upstream backup.

The only other parallel recovery protocol developed by Hwang et al only tolerates one node failure, and cannot handle stragglers. Several recent research systems have looked at online processing in clusters. MapReduce Online is a streaming Hadoop runtime. iMR is an in-situ MapReduce engine for log processing.

CBP and Comet provide "bulk incremental processing" by running MapReduce jobs on new data every few minutes to update state in a distributed file system. Naiad runs computations incrementally. Percolator performs incremental computations using triggers. D-Stream's parallel recovery mechanism is conceptually similar to recovery techniques in MapReduce, GFS, and RAMCloud.

**Outline of the solution proposed:**

The key idea proposed in this paper is to treat streaming computations
as a series of deterministic batch computations on small time intervals. The input data received during each interval is stored reliably across the cluster to form an input dataset for that interval. Once the time interval
completes; this dataset is processed via deterministic parallel operations, such as  map, reduce and groupBy, to produce new datasets representing program outputs or intermediate states. These results are then stored in resilient distributed datasets (RDDs), an efficient storage abstraction
that avoids replication by using lineage for fault recovery.
D-Streams provide both stateless operators, such as map, which act independently on each time interval, and stateful operators, such as
aggregation over a sliding window, which operate on multiple
intervals and may produce intermediate RDDs as state.

**Fault recovery:**
**Results description:**

- **Scalability**: In this Paper, the scalability of the system was evaluated through two applications: Grep, which counts input records matching a pattern, and WordCount, which performs a sliding window word count over 10-second windows. For both applications, the maximum throughput achievable was measured on different-sized clusters with an end-to-end latency target of either 1 or 2 seconds. The system can process roughly 40 MB/second/node for Grep and 20 MB/s/node for WordCount at sub-second latency, as well as slightly more data if we allow 2s of latency. The system also scales nearly linearly to 50 nodes
- **Parallel fault recovery:** In this paper, Parallel fault recovery was evaluated using two applications, both of which received 10 MB/s of data per node on 10 nodes, and used 2-second batching intervals. The first application, MaxCount, performed a word count in each 2-second interval and computed the maximum count for each word over the past 30 seconds using a sliding window.

**My impression on the Idea:**
D-Streams has its application and benefits in stream processing over large clusters.
D-Streams improves efficiency over the traditional replication and upstream backup solutions in streaming databases.
This paper proposes an idea that is conceptually a small improvement on the existing models for stream processing and is similar to MapReduce, GFS, and RAMCloud to some extent.

## Conclusion

In this project I worked around the Spark Streaming and tried to perform the tasks given in section 1. In the Second I read a research paper “Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters” by Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, Ion Stoica from University of California, Berkeley and wrote a brief description on the same. This project has been the most challenging part of the module.