# Spark Streaming

## Need of Streaming

- Credit Card Fraud Use Case
- Need of powerful analytics of real time streaming data like twitter, stock market, geographical data etc.

## Streaming Data

- A data stream is an unbounded sequence of data arriving continuously.

- Streaming divides continuously flowing input data into discrete units for further processing.

- Stream processing is low latency processing and analyzing of streaming data.

## Spark Streaming

- **Spark Streaming** was added to Apache Spark in 2013, an extension of the core Spark API that provides scalable, high-throughput and fault-tolerant stream processing of live data streams.

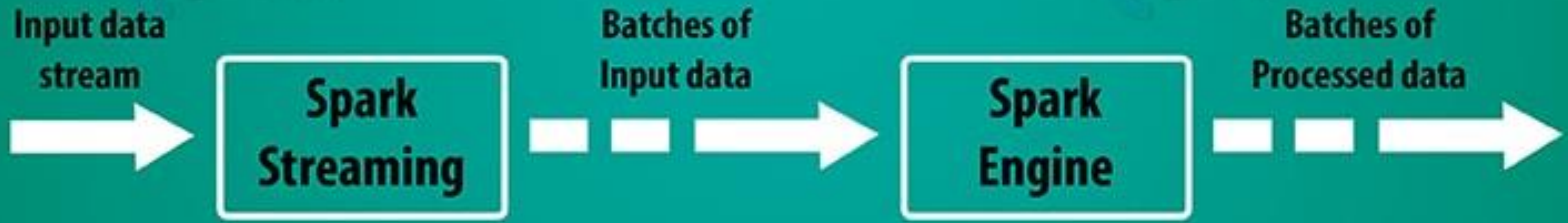
- Data ingestion can be done from many sources like Kafka, Apache Flume, Amazon Kinesis or TCP sockets and processing can be done using complex algorithms that are expressed with high-level functions like map, reduce, join and window.

- Finally, processed data can be pushed out to filesystems, databases and live dashboards.

## Internal working

- Live input data streams is received and divided into batches by Spark streaming, these batches are then processed by the Spark engine to generate the final stream of results in batches.
- Its key abstraction is Apache Spark Discretized Stream or, in short, a Spark DStream, which represents a stream of data divided into small batches.
- DStreams are built on Spark RDDs, Spark’s core data abstraction.
- This allows Streaming in Spark to seamlessly integrate with any other Apache Spark components like Spark MLlib and Spark SQL.

# Spark Streaming



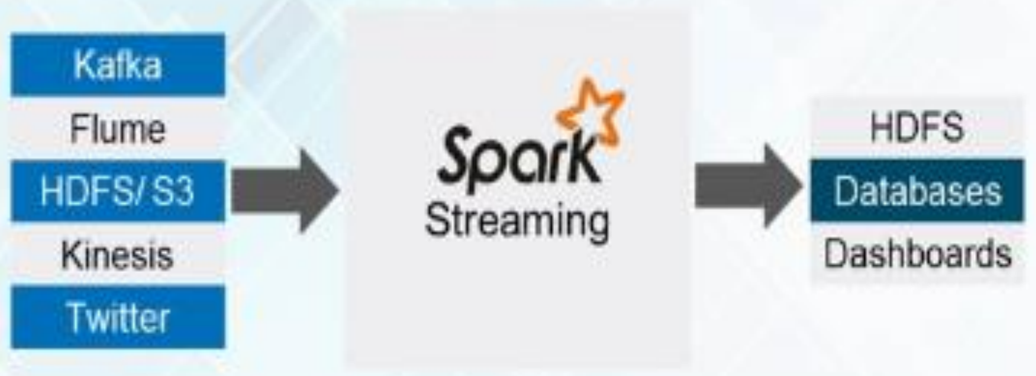


**Figure:** *Data from a variety of sources to various storage systems*

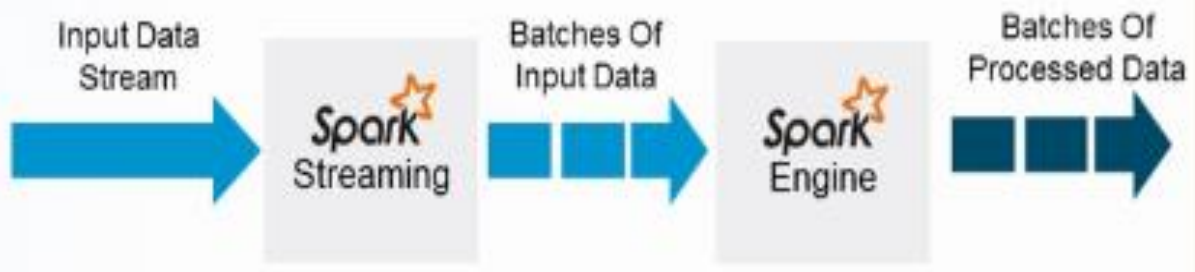


**Figure:** *Incoming streams of data divided into batches*

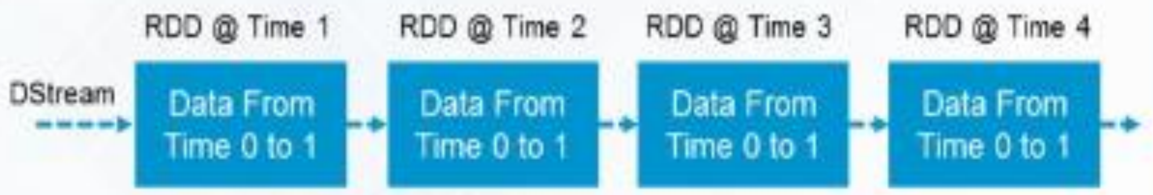


**Figure:** *Input data stream divided into discrete chunks of data*

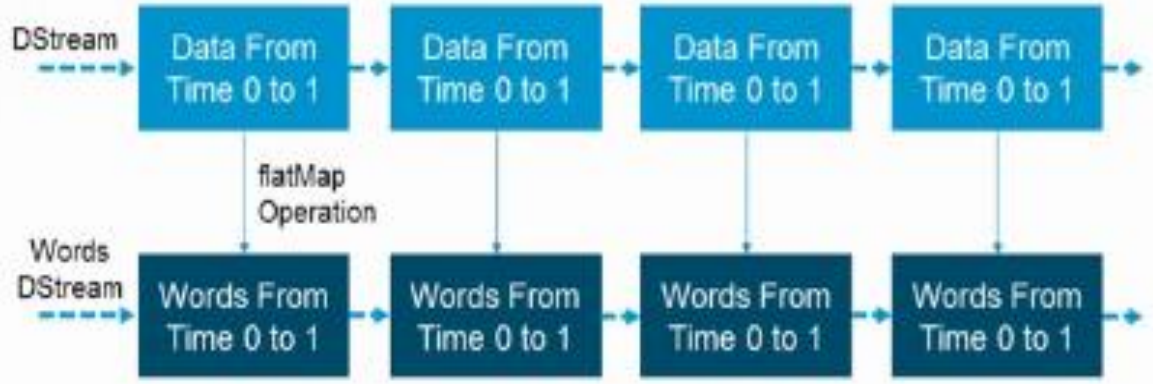


**Figure:** *Extracting words from an InputStream*

## Spark Streaming

- Instead of processing the streaming data one record at a time, Spark Streaming discretizes the data into tiny, sub-second micro-batches.
- In other words, Spark Streaming receivers accept data in parallel and buffer it in the memory of Spark's workers nodes.
- Then the latency-optimized Spark engine runs short tasks to process the batches and output the results to other systems.
- Unlike the traditional continuous operator model, where the computation is statically allocated to a node, Spark tasks are assigned to the workers dynamically on the basis of data locality and available resources.
- This enables better load balancing and faster fault recovery.

## Goals of Spark Streaming

## Dynamic load balancing

- Dividing the data into small micro-batches allows for fine-grained allocation of computations to resources.
- Let us consider a simple workload where partitioning of input data stream needs to be done by a key and processed. In the traditional record-at-a-time approach, if one of the partitions is more computationally intensive than others, the node to which that partition is assigned will become a bottleneck and slow down the pipeline.
- The job’s tasks will be naturally load balanced across the workers where some workers will process a few longer tasks while others will process more of the shorter tasks in Spark Streaming.

## Fast failure and straggler recovery

- Traditional systems have to restart the failed operator on another node to recompute the lost information in case of node failure.
- Only one node is handling the recomputation due to which the pipeline cannot proceed until the new node has caught up after the replay.
- In Spark, the computation discretizes into small tasks that can run anywhere without affecting correctness.
- So failed tasks we can distribute evenly on all the other nodes in the cluster to perform the recomputations and recover from the failure faster than the traditional approach.

## Unification of batch, streaming and interactive analytics

- A DStream in Spark is just a series of RDDs in Spark that allows batch and streaming workloads to interoperate seamlessly.

- Arbitrary Apache Spark functions can be applied to each batch of streaming data.

- Since the batches of streaming data are stored in the Spark’s worker memory, it can be interactively queried on demand.

## Advanced analytics like machine learning and interactive SQL

- Spark interoperability extends to rich libraries like MLlib (machine learning), SQL, DataFrames, and GraphX.

- RDDs generated by DStreams can convert to DataFrames and query with SQL.

- Machine learning models generated offline with MLlib can apply to streaming data.

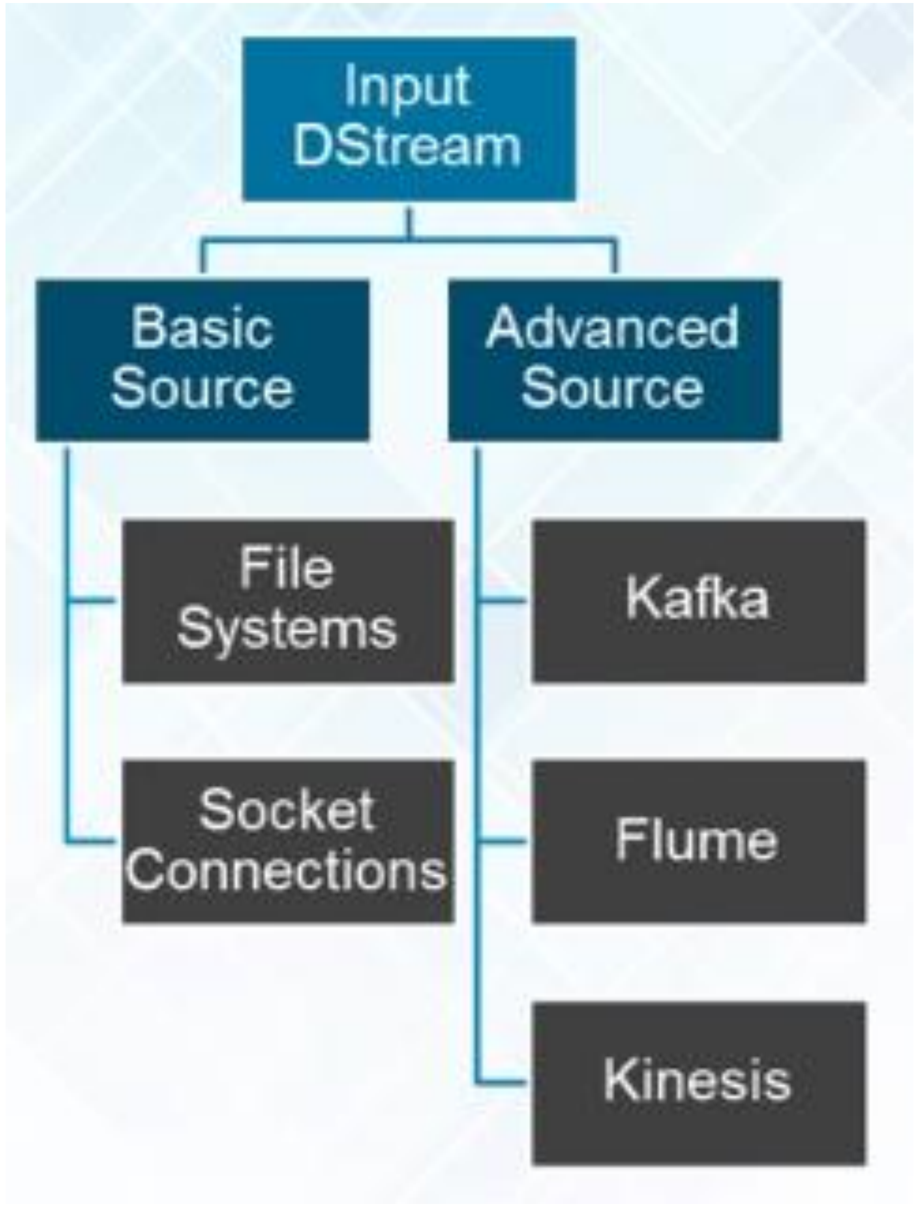
## Performance

- Spark Streaming’s ability to batch data and leverage the Spark engine leads to almost higher throughput to other streaming systems.

- Spark Streaming can achieve latencies as low as a few hundred milliseconds.

## Spark Streaming Sources

- Every input DStream (except file stream) associate with a Receiver object which receives the data from a source and stores it in Spark's memory for processing. There are two categories of built-in streaming sources:
- **Basic sources –** These are the sources directly available in the StreamingContext API. Examples: file systems, and socket connections.
- **Advanced sources –** Sources like Kafka, Flume, Kinesis, etc. are available through extra utility classes. These require linking against extra dependencies.
- There are two types of receivers base on their reliability:
- **Reliable Receiver –** A reliable receiver is the one that correctly sends an acknowledgment to a source when the data receives and stores in Spark with replication.
- **Unreliable Receiver –** An unreliable receiver does not send an acknowledgment to a source. This we can use for sources when one does not want or need to go into the complexity of acknowledgment.


Input DStream
Basic Source
Advanced Source
File Systems
Socket Connections
Kafka
Flume
Kinesis

## Streaming Context

- Consumes a stream of data in Spark
- Registers an InputDStream to produce a Receiver object.
- It is the main entry point for Spark functionality.
- Spark provides a number of default implementations of sources like Twitter,Akka Actor and ZeroMQ that are accessible from the streaming context.

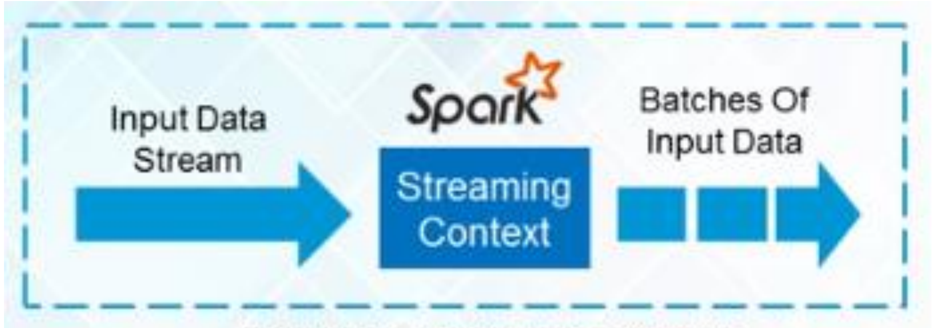


**Figure:** *Spark Streaming Context*

**Figure:** *Default Implementation Sources*

## DStream

- ***DStream is a continuous stream of RDD (Spark abstraction)***.
- Every RDD in DStream contains data from the certain interval.
- Any operation on a DStream applies to all the underlying RDDs.
- Spark Streaming offers fault-tolerance properties for DStreams as that for RDDs.
- As long as a copy of the input data is available, it can recompute any state from it using the lineage of the RDDs.
- By default, Spark replicates data on two nodes. As a result, Spark Streaming can bear single worker failures.

## Apache Spark DStream Operations

- Spark DStream also support two types of Operations:
  - **Transformation:** There are two types of transformation in DStream:
    - Stateless Transformations
    - Stateful Transformations
  - **Output Operation :**Once we get the data after transformation, on that data output operation are performed in Spark Streaming. After the debugging of a program, using output operation we can only save our output. Some of the output operations are print(), save() etc..

# Stateless Transformations

- The processing of each batch has no dependency on the data of previous batches.

- *Stateless transformations* are simple RDD transformations. It applies on every batch meaning every RDD in a DStream.

- It includes common **RDD transformations** like *map(), filter(), reduceByKey()* etc. Although these functions seem like applying to the whole stream, ***each DStream is a collection of many RDDs (batches)***. As a result, each stateless transformation applies to each RDD.

## Stateless Transformations

- Stateless transformations are capable of combining data from many DStreams within each time step. For example, key/value DStreams have the same join-related transformations as RDDs— *cogroup(), join(), leftOuterJoin()* etc.

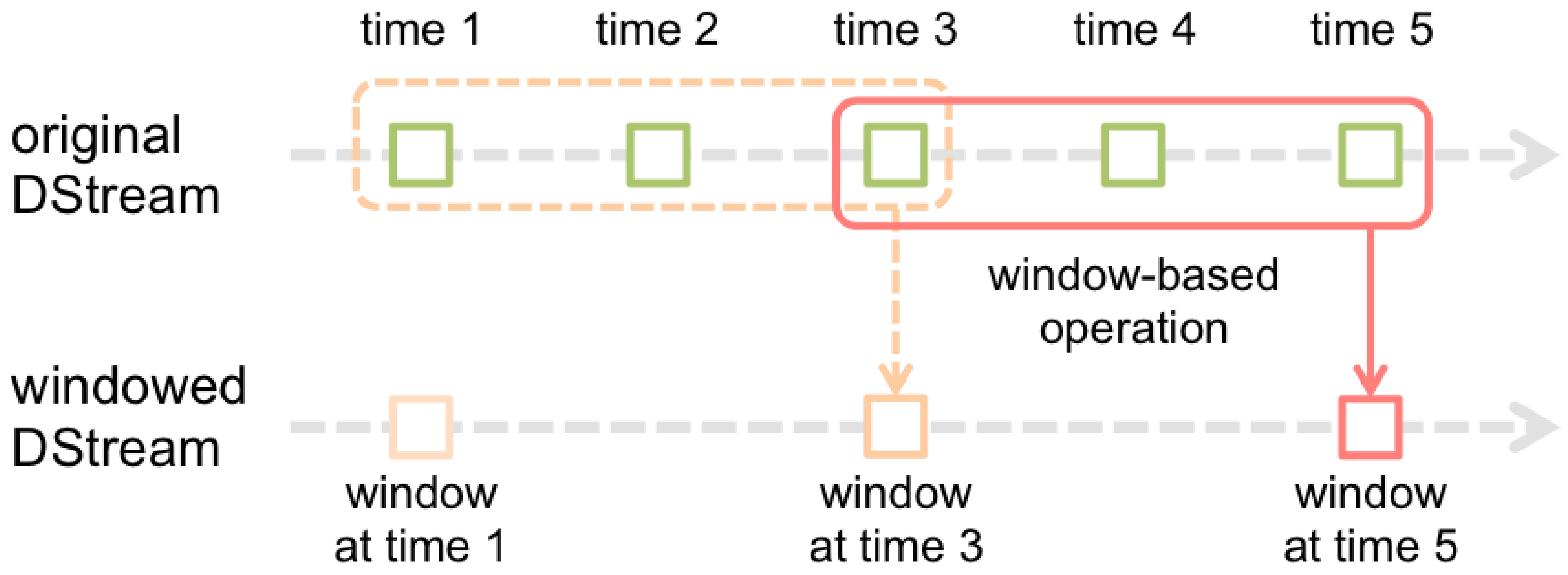
- We can use these operations on DStreams to perform underlying RDD operations on each batch.

- If *stateless transformations* are insufficient, DStreams comes with an advanced operator called transform(). **transform()** allow operating on the RDDs inside them. The *transform()* allows any arbitrary RDD-to-RDD function to act on the DStream. This function gets called on each batch of data in the stream to produce a new stream.

# Stateful Transformations

- It uses data or intermediate results from previous batches and computes the result of the current batch.
- *Stateful transformations* are operations on DStreams that track data across time.
- Thus it makes use of some data from previous batches to generate the results for a new batch.
- The two main types are **windowed operations**, which act over a sliding window of time periods, and **updateStateByKey()**, which is used to track state across events for each key (e.g., to build up an object representing each user session).

# Window Operations



# Window Operations

- Every time the window slides over a source DStream, the source RDDs that fall within the window are combined and operated upon to produce the RDDs of the windowed DStream. In previous figure, the operation is applied over the last 3 time units of data, and slides by 2 time units. This shows that any window operation needs to specify two parameters:
  - window length - The duration of the window (3 in the figure).
  - sliding interval - The interval at which the window operation is performed (2 in the figure).

- These two parameters must be multiples of the batch interval of the source DStream (1 in the figure).

- E.g. Assume we want to do word count over the last 30 seconds of data, every 10 seconds. To do this, we have to apply the reduceByKey operation on the pairs DStream of (word, 1) pairs over the last 30 seconds of data. This is done using the operation reduceByKeyAndWindow.

## Input DStreams and Receivers

- Input DStream is a DStream representing the stream of input data from streaming source.
- Receiver object is associated with every input DStream object. It receives the data from a source and stores it in Spark's memory for processing.
- Streaming sources create many inputs DStream to receive multiple streams of data in parallel.
- It creates multiple receivers that receive many data stream.
- Spark worker/executor is a long-running task. Thus, occupies one of the cores which associate to Spark Streaming application. So, it is necessary that, Spark Streaming application has enough cores to process received data.

## Spark Streaming Demostration

# Python Code for WordCount

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext

//sc = SparkContext("local[2]", "NetworkWordCount")
ssc = StreamingContext(sc, 3)
lines = ssc.socketTextStream("hadoop-master", 9999)
words = lines.flatMap(lambda line: line.split(" "))
pairs = words.map(lambda word: (word, 1))
wordCounts = pairs.reduceByKey(lambda x, y: x + y)
wordCounts.pprint()
ssc.start()
ssc.awaitTermination()  # Wait for the computation to terminate
```

## Steps for execution

- Launch python shell using following command:

  ‘pyspark –master local[2]’ //Lanches spark in local mode with 2 worker threads
- Start netcat server using following command to get streaming data:

  ‘nc –lk 9999’
- Copy previous lines in shell and press enter.

[Note:3rd line is commented as executing in pyspark shell. Sparkcontext is already created.]

- Enter data in netcat server and observe output in shell.
- Command to stop StreamingContext : ssc.stop()

# References

- https://data-flair.training/blogs/apache-spark-streaming-tutorial/
- https://spark.apache.org/docs/latest/streaming-programming-guide.html
- https://www.youtube.com/watch?v=uD_q4Rm4i2Q

Thank You!