PySpark

Spark Streaming

Agenda

What is **Batch Processing?**

What is Batch Processing?

In batch processing, the data is collected and processed, and results are produced in batches

So far, we have seen the use cases wherein we have huge data, and we need insights on that data

Batch processing is very efficient in processing high volume of data, and we can leverage Spark to achieve parallelism in batch processing

Batch processing jobs can be scheduled to start up on their own as resources permit

What is **Stream Processing?**

What is Stream Processing?

Many Big Data applications receive data in real time, for example, weather forecast data, share trading data, data coming from IoT devices, etc.

Applications must act on data in real time rather than looking at it later

For these applications, the data is valuable at its time of arrival

Processing such data in real time or near real time is known as Stream processing

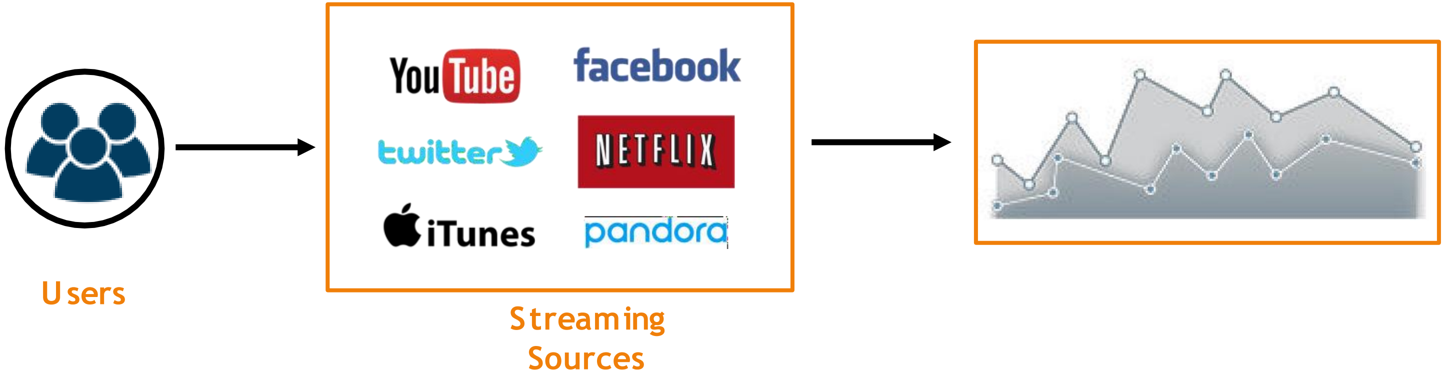
Batch vs Stream Processing

Today's business decisions are data and analytics driven. Hence, organizations cannot wait for the entire dataset to come in place and then start the analysis of that data to take the decisions. They rather rely on real-time data analytics to form the decisions

Batch processing is where the processing happens on blocks of data that have already been stored over a period, whereas stream processing refers to processing data as and when it arrives

More on Streaming

- Data streaming is a technique for transferring data so that it can be processed as a steady and continuous stream
- A data stream is an unbounded sequence of data arriving continuously
- Streaming technologies are becoming increasingly important with the growth of the Internet



More on Streaming

- Broadly, stream processing falls into the following two categories:

CEP (Complex Event Processing)

- It is a Stateful event processing system used in place where multiple event occurrence might co-relate so some activity

ESP (Event Stream Processing)

- Falls into stateless event processing used in jobs like continuous ETL transformation, trending topics in social media, etc

- Stream processing involves continuous input and outcome of data. The emphasis is more on the velocity of the data
- Data must be processed within a small time frame or near real time

More on Streaming

- Stream processing gives decision makers the ability to adjust contingencies based on events and trends that develop in real time

- In a distributed stream processing, an incoming data is acted upon immediately and the information/action is pushed onto the user to take some decisions

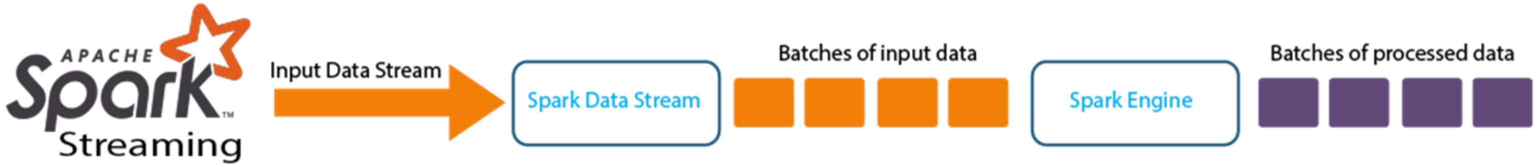
- Basically, it has inverted the access pattern we have been using so far

- Usually, a user investigates the data, processes it by running few queries, and takes some decisions

Introduction to
Spark Streaming

Spark Streaming Overview

- Spark streaming is used for processing real-time streaming of data
- It is a useful addition to the core Spark API
- Spark Streaming enables high-throughput and fault-tolerant stream processing of data streams
- The fundamental unit of stream is DStream which is basically a series of RDDs to process the real-time data



Companies Using **Streaming Data**

Companies Using Streaming Data

And many more…

Features of Spark Streaming

Features of Spark Streaming

Scaling:
Scales to hundreds of nodes

Speed:
Achieves low latency

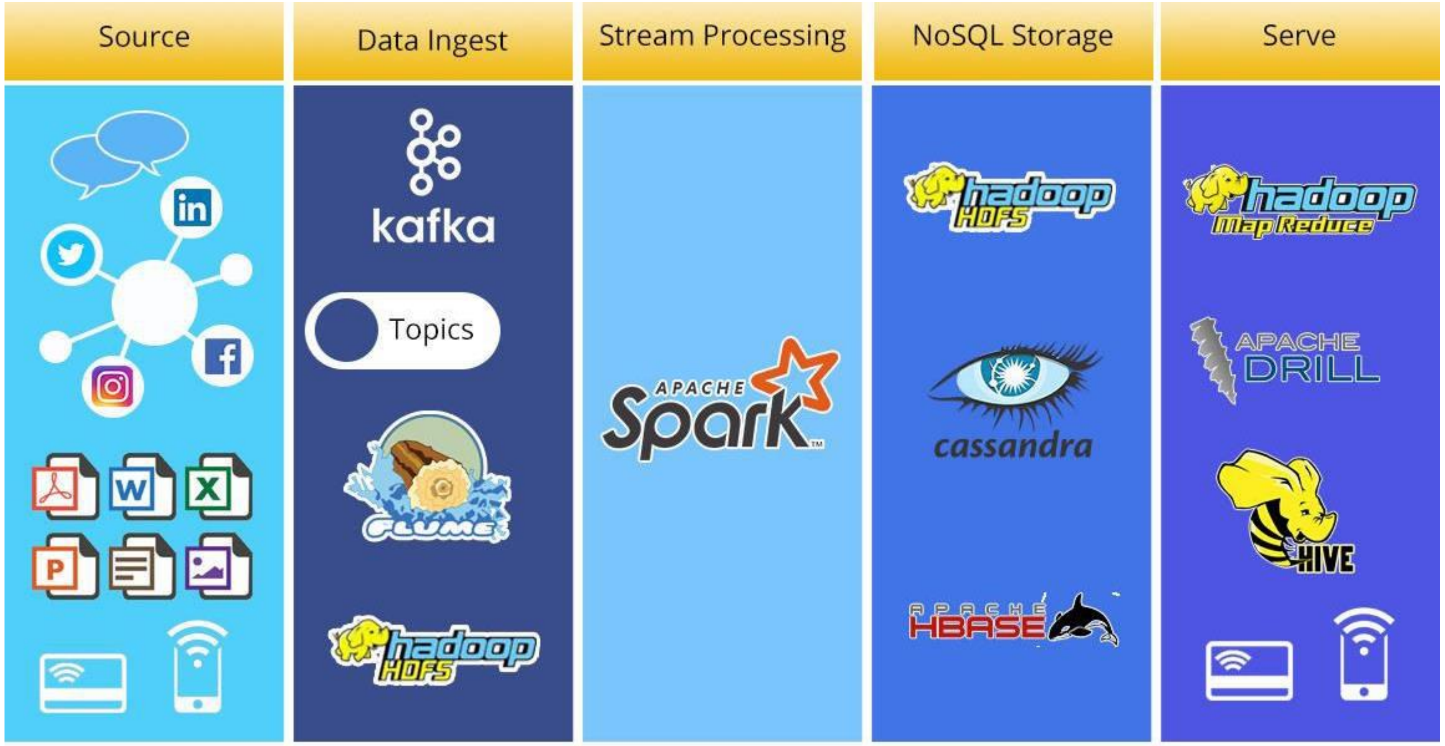
Fault Tolerance:
Efficiently recovers from failures

Integration:
Integrates with batch and real-time processing

Business Analysis:
Used to track behaviors of customers

Spark Streaming Workflow

Stream Processing Workflow



Initializing StreamingContext

Streaming Context

- It is the main entry point for Spark streaming functionality
- Spark provides several default implementations of sources such as Twitter, Akka actor, and ZeroMQ that are accessible from the context
- Whatever we do in Spark streaming has to start from creating an instance of StreamingContext

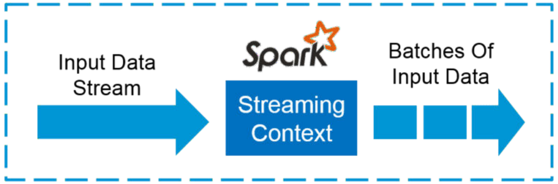


Figure: *Spark Streaming Context*

Figure: *Default Implementation Sources*

- Consumes a stream of data in Spark
- Registers an InputDStream to produce a Receiver object
- Provides methods used to create DStreams from various input sources

Streaming Context: Initialization

- StreamingContext can be created in three ways:
  - By providing a **Spark master URL** and an appName
  - From an **org.apache.spark.SparkConf** configuration
  - From an existing **org.apache.spark.SparkContext**
- The associated SparkContext can be accessed using context.sparkContext. After creating and transforming DStreams, the streaming computation can be started and stopped using context.start() and context.stop(), respectively

Example:

```
from pyspark, import SparkContext
from pyspark.streaming import StreamingContext
sc = SparkContext("local[*]", "PySpark")
ssc = StreamingContext(sc, 1)
```

Streaming Context: Creation Steps

- A **DStreamGraph** is created
- A **JobScheduler** is created
- A **StreamingJobProgressListener** is created
- A Streaming tab in web UI is created (when **spark.ui.enabled** is enabled)
- A **StreamingSource** is instantiated
- At this point, StreamingContext enters the **Initialized** state

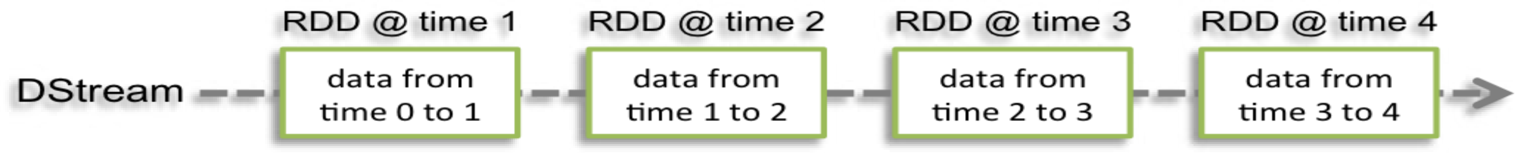
Streaming Context: Points to Remember

- Once a context has been started, no new streaming computations can be set up or added to it
- It cannot be restarted once stopped
- Only one StreamingContext can be active in a JVM at the same time
- stop() on StreamingContext also stops the SparkContext. To stop only the StreamingContext, set the optional parameter of stop() called stopSparkContext to false
- A SparkContext can be reused to create multiple StreamingContexts, if the previous StreamingContext is stopped (without stopping the SparkContext) before the next StreamingContext is created

Discretized Streams **(DStreams)**

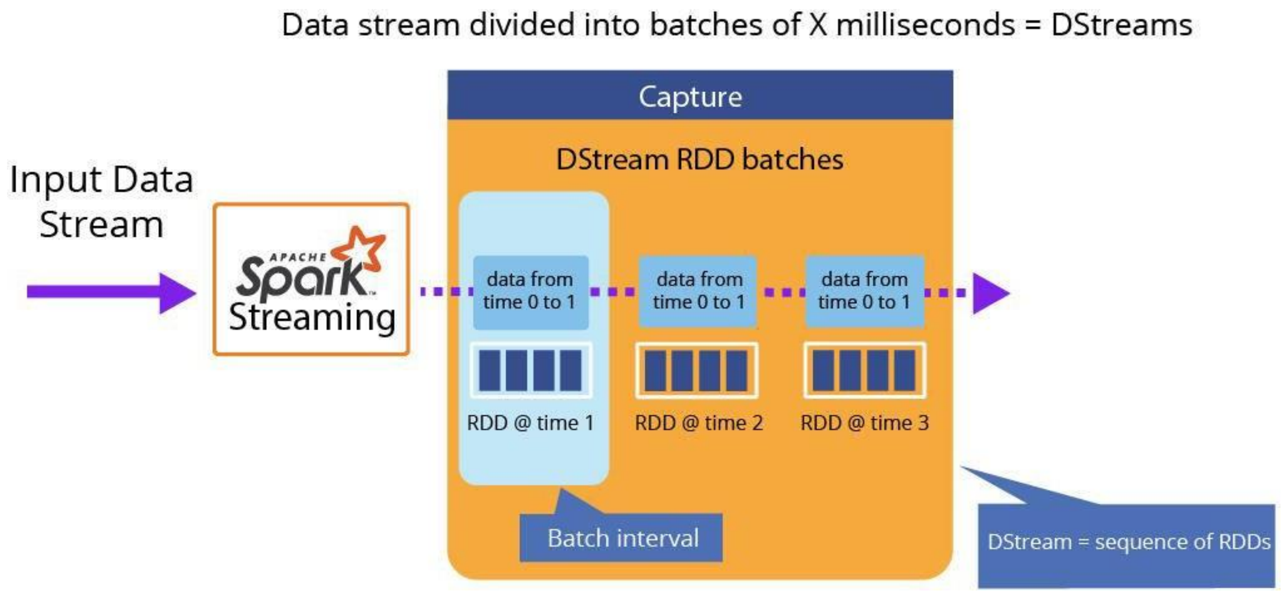
DStreams

- The way we have RDDs for Spark Core, SchemaRDD and Dataframes for Spark SQL, similarly DStream is the basic abstraction provided by Spark Streaming
- DStream represents a continuous stream of data, either the input data stream received from source or the processed data stream generated by transforming the input stream
- Internally, a DStream is represented by a continuous series of RDDs, which is Spark's abstraction of an immutable object

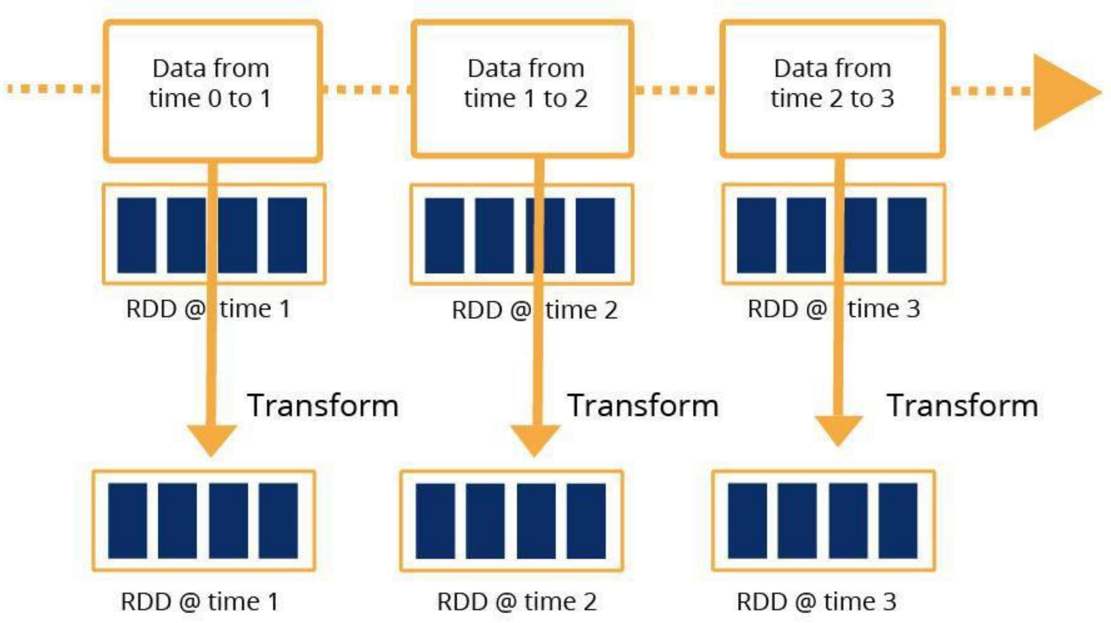


- Any operation applied on a DStream translates to operations on the underlying RDDs

Processing Spark DStreams



Processing Spark DStreams



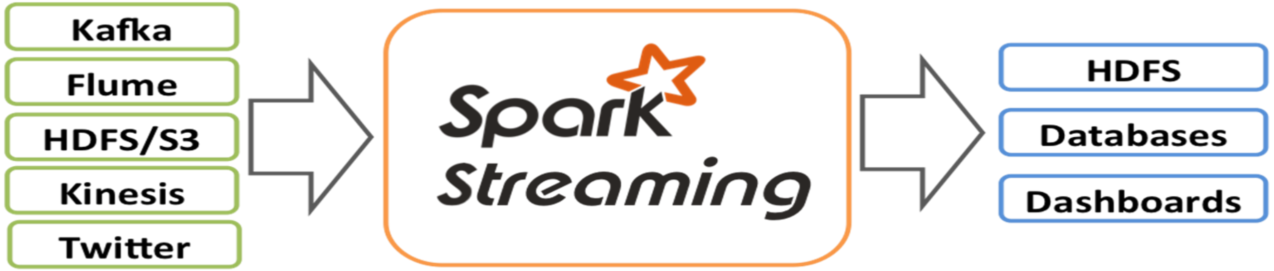
Input **DStreams** and **Receivers**

DStreams: Input DStreams

- Input DStreams represent the stream of input data received from streaming sources
- Spark Streaming provides two categories of built-in streaming sources:
  - **Basic sources:** 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 as discussed in the linking section
- If you want to receive multiple streams of data in parallel in your streaming application, then you can create multiple input DStreams

Streaming Source and Destination

- **DataSources:**
  - File-based source: HDFS
  - Network bases: TCP Sockets
  - Twitter, Kafka, Flume, and ZeroMQ



DStreams: Receiver

- Every input DStream is associated with a Receiver object that receives the data from a source and stores it in Spark’s memory for processing

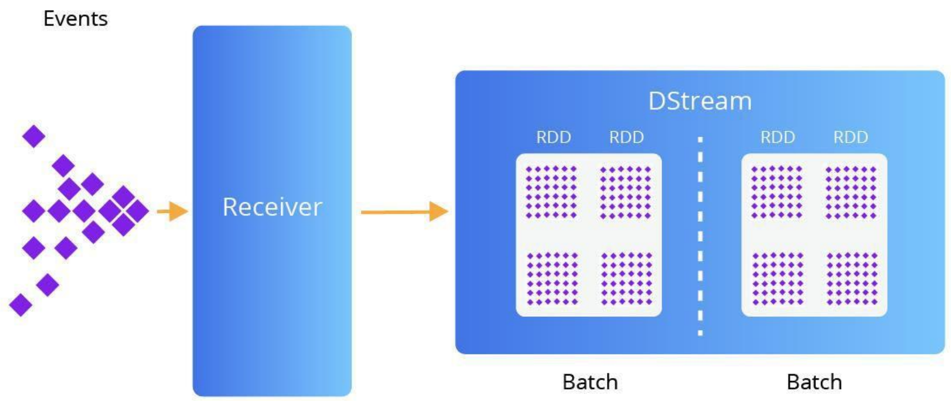


Figure: The receiver sends data onto the input DStream where each batch contains RDDs

Transformations on DStreams

Transformations on DStreams

- Similar to that of RDDs, transformations allow the data to be modified from the input DStream
- DStreams support many of the transformations available on normal Spark RDDs

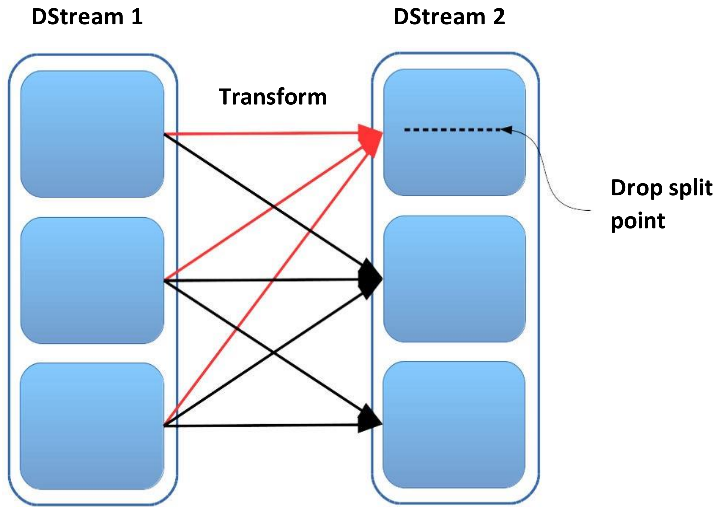


Figure: DStream Transformations

Transformations on DStreams

Transformation	Meaning
map(func)	Returns a new DStream by passing each element of the source DStream through a function func
flatMap(func)	Similar to map, but each input item can be mapped to 0 or more output items
filter(func)	Returns a new DStream by selecting only records of the source DStream on which func returns true
repartition(numPartitions)	Changes the level of parallelism in this DStream by creating more or fewer partitions
union(otherStream)	Returns a new DStream that contains the union of elements in the source DStream and otherDStream
count()	Returns a new DStream of single-element RDDs by counting the number of elements in each RDD of the source DStream
reduce(func)	Returns a new DStream of single-element RDDs by aggregating elements in each RDD of the source DStream using a function func (which takes 2 arguments and returns 1). The function should be associative and commutative so that it can be computed in parallel

Transformations on DStreams

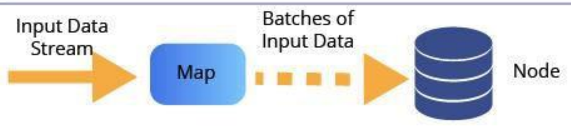


Figure: Input DStream being converted through map(func)

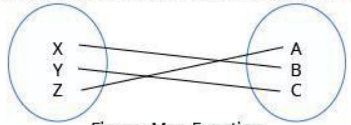


Figure: Map Function

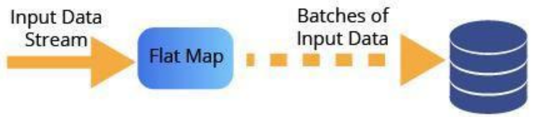
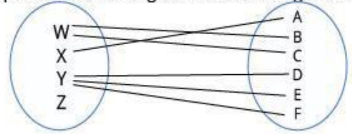


Figure: Input DStream being converted through flatMap(func)



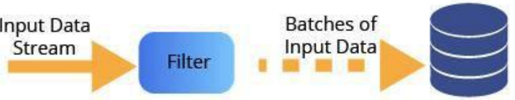


Figure: Input DStream being converted through filter(func)

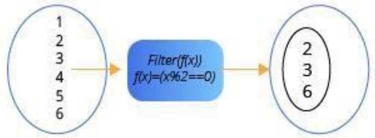


Figure: flatMap Function for Even Numbers

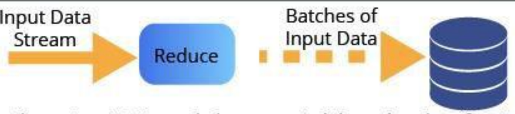


Figure: Input DStream being converted through reduce(func)

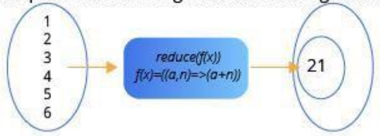


Figure: flatMap Function to get Cumulative Sum

Output Operations on **DStreams**

Output Operations on DStream

- Output operations allow DStream's data to be pushed out to external systems like databases or file systems
- Output operations trigger the actual execution of all the DStream transformations

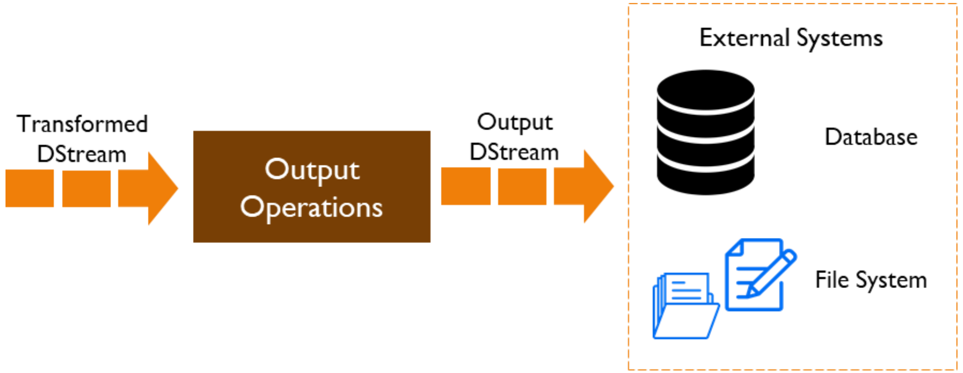


Figure: Output Operations on DStreams

Design Patterns for Using ForeachRDD

- dstream.foreachRDD is a powerful primitive that lets the data to be sent to external systems. But you must know the method of using it correctly and efficiently, as many common mistakes tend to occur
- Output operations **execute** DStreams **lazily**

```
def sendRecord(rdd):
    connection = createNewConnection()  # executed at the driver
    rdd.foreach(lambda record: connection.send(record))
    connection.close()

dstream.foreachRDD(sendRecord)
```

Design Patterns for Using ForeachRDD

- DStreams allow developers to cache/persist the stream's data in memory. This is useful in cases where the data in the DStream is computed multiple times
- This can be done using the persist() method on a Dstream
- For window-based operations like reduceByWindow and reduceByKeyAndWindow and state-based operations like updateStateByKey, this is implicitly true
- For input streams that receive data over the network (such as Kafka, Flume, Sockets, etc.), the default persistence level is set to replicate the data to two nodes for fault tolerance

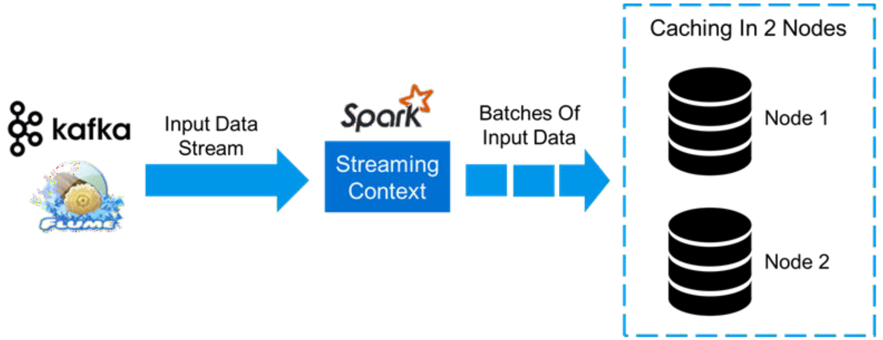


Figure: Caching Into 2 Nodes

Checkpointing

- Checkpoints are similar to checkpoints in gaming. They make it run 24/7, and make it resilient to failures unrelated to the application logic
- Spark Streaming needs to checkpoint enough information to a fault-tolerant storage system such that it can recover from failures
- There are two types of data that are checkpointed

It is saving of the information defining the streaming computation

Metadata Checkpoints

Data Checkpoints

It is saving of the generated RDDs to reliable storage

Checkpointing

- Checkpointing must be enabled for applications with any of the following requirements:
  - **Usage of stateful transformations:** If either updateStateByKey or reduceByKeyAndWindow (with inverse function) is used in the application, then the checkpoint directory must be provided to allow for periodic RDD checkpointing
  - **Recovering from failures of the driver running the application**: Metadata checkpoints are used to recover with progress information

Accumulators and Broadcast Variables

- Accumulators and Broadcast variables cannot be recovered from checkpoint in Spark Streaming
- If you enable checkpointing and use Accumulators or Broadcast variables as well, you'll have to create lazily instantiated singleton instances for Accumulators and Broadcast variables so that they can be re-instantiated after the driver restarts on failure

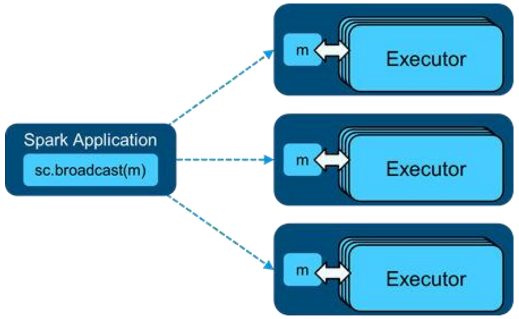


Figure: Broadcasting A Value To Executors

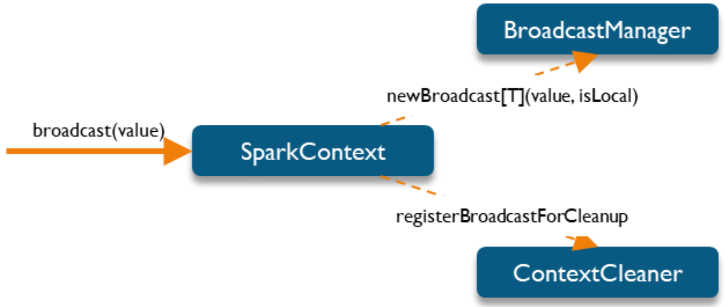
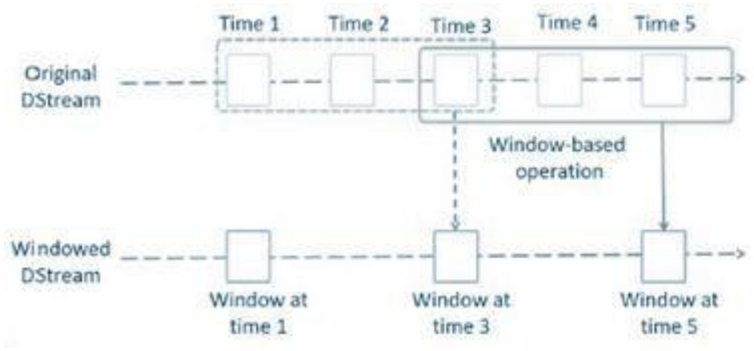


Figure: SparkContext and Broadcasting

Windowed Operators

Window Operators

Spark streaming supports window operations that allow you to implement transformations over a sliding window of data

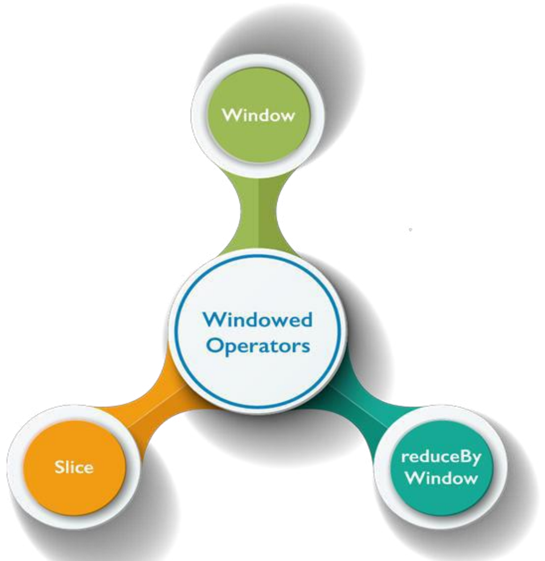


Example:

```
windowedWordCounts = pairs.reduceByKeyAndWindow(lambda x, y: x + y,
lambda x, y: x - y, 30, 10)
```

Window Operators

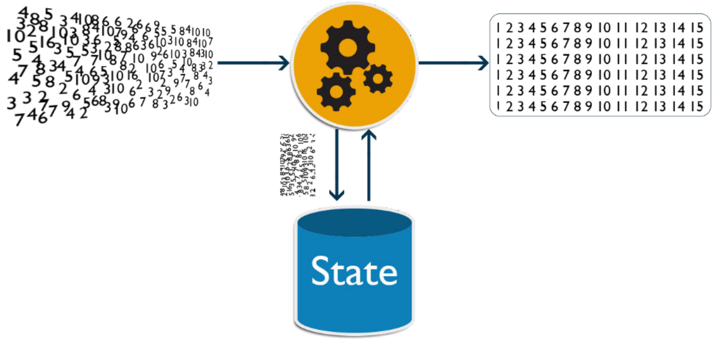
- As depicted, each time a window slides over a DStream source, the RDDs source falling within that window is being united
- They are then being operated upon for producing the windowed DStream RDDs
- In this case, slides are applied by 2-time units of data and the operation is applied to last 3-time units
- It implies that all window operations require at least two parameters to be specified, which are window length and sliding interval
- There are mainly three types of operators:



Stateful Operators

Stateful Operators

- Stateful operators (like mapWithState or updateStateByKey) are part of a set of additional operators available on DStreams of key–value pairs, i.e., instances of DStream[(K, V)]
- They allow you to build stateful stream processing pipelines and are also called cumulative calculations
- These operations are dependent on the earlier data batches. Therefore, they accumulate metadata over time continuously. For clearing this, Spark streaming saves intermediate data to HDFS, and hence supports periodic checkpointing
- Few Stateful Operators are:
    - mapWithState
    - StateSpec
    - updateStateByKey



Twitter Sentiment Analysis

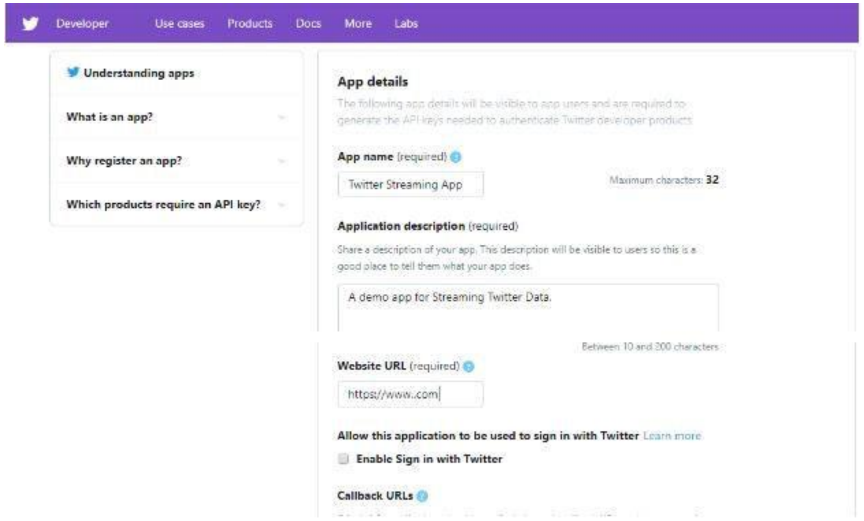
Twitter Sentiment Analysis

- Apache Spark is mostly used for real-time data streaming capabilities
- It can stream data with lightning-fast speed
- Companies use sentiment analysis to analyze the response of their customers
- Twitter Sentiment Analysis can be used to understand the sentiments behind a specific hashtag
- E.g. If #football is trending, then how many tweets have positive, negative, and neutral sentiments
- We will implement the following steps to perform Real-time Twitter Sentiment Analysis

Twitter Sentiment Analysis

Step1: Creating your own credentials for Twitter APIs:

In order to get tweets from Twitter, you need to register on https://developer.twitter.com/apps by clicking on “Create new app” and then fill the below form by clicking on “Create your Twitter app.”



App details

The following app details will be visible to app users and are required to generate the API keys needed to authenticate Twitter developer products

App name (required)

Twitter Streaming App

Maximum characters: **32**

Application description (required)

Share a description of your app. This description will be visible to users so this is a good place to tell them what your app does.

A demo app for Streaming Twitter Data.

Between 10 and 200 characters

Website URL (required)

https://www..com

Allow this application to be used to sign in with Twitter Learn more

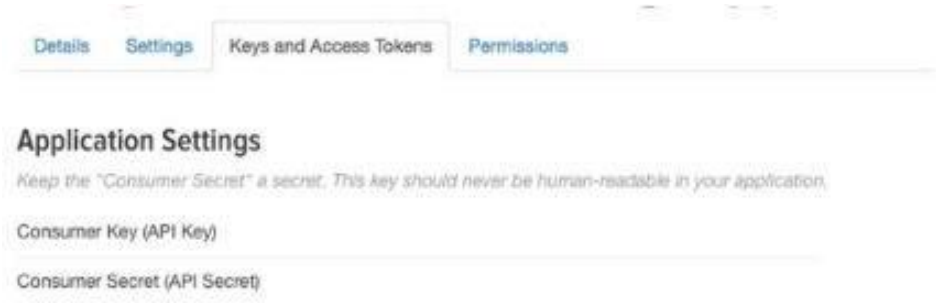
☐ **Enable Sign in with Twitter**

Callback URLs

Twitter Sentiment Analysis

Step2: Generate Access tokens

To get twitter access keys and tokens, go to your newly created app and open the “Keys and Access Tokens” tab. Then click on“Generate my access token.”



Application Settings

Keep the "Consumer Secret" a secret. This key should never be human-readable in your application.

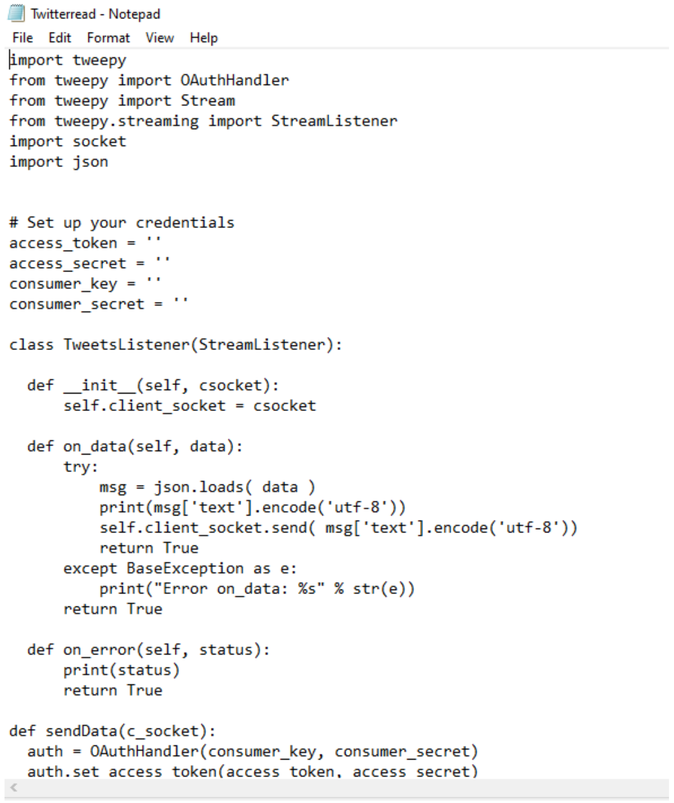
Consumer Key (API Key)

Consumer Secret (API Secret)

Twitter Sentiment Analysis

Step3: Building the Twitter HTTP Client

- In this step, I’ll show you how to build a simple client that will get the tweets from Twitter API using Python and pass them to the Spark Streaming instance
- We need to create a python file for fetching data using Twitter APIs
- You can create a python file easily in notepad
- You can find this python file in the course material

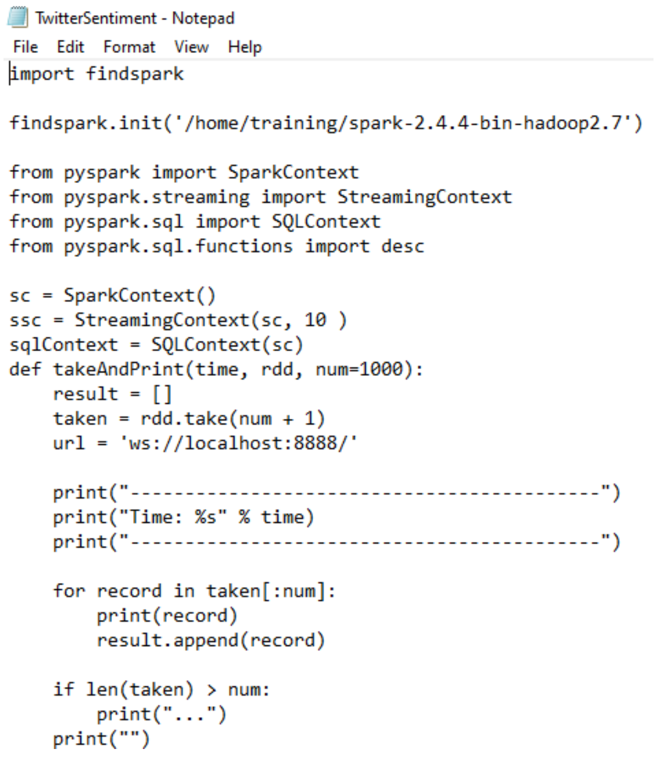


```
import tweepy
from tweepy import OAuthHandler
from tweepy import Stream
from tweepy.streaming import StreamListener
import socket
import json


# Set up your credentials
access_token = ''
access_secret = ''
consumer_key = ''
consumer_secret = ''

class TweetsListener(StreamListener):

  def __init__(self, csocket):
      self.client_socket = csocket

  def on_data(self, data):
      try:
          msg = json.loads( data )
          print(msg['text'].encode('utf-8'))
          self.client_socket.send( msg['text'].encode('utf-8'))
          return True
      except BaseException as e:
          print("Error on_data: %s" % str(e))
      return True

  def on_error(self, status):
      print(status)
      return True

def sendData(c_socket):
  auth = OAuthHandler(consumer_key, consumer_secret)
  auth.set_access_token(access_token, access_secret)
```

Twitter Sentiment Analysis

Step 4: Setting Up Our Apache Spark Streaming Application

- Let’s build up our Spark streaming app that will do real-time processing for the incoming tweets, extract the hashtags from them, and classify the tweets as positive, negative, and neutral
- We can create this python file in a text editor
- We can find this file in the course material



```
import findspark

findspark.init('/home/training/spark-2.4.4-bin-hadoop2.7')

from pyspark import SparkContext
from pyspark.streaming import StreamingContext
from pyspark.sql import SQLContext
from pyspark.sql.functions import desc

sc = SparkContext()
ssc = StreamingContext(sc, 10 )
sqlContext = SQLContext(sc)
def takeAndPrint(time, rdd, num=1000):
    result = []
    taken = rdd.take(num + 1)
    url = 'ws://localhost:8888/'

    print("-------------------------------------------")
    print("Time: %s" % time)
    print("-------------------------------------------")

    for record in taken[:num]:
        print(record)
        result.append(record)

    if len(taken) > num:
        print("...")
    print("")
```

Word Count Program Using Kafka

Word Count Program Using Kafka

- The current-day industry is emerging with lots of real-time data that needs to be processed in real time
- So, there are various frameworks to handle real-time data streaming efficiently
- Apache Kafka is an open-source, distributed, publish-subscribe messaging system that manages and maintains the real-time stream of data from different applications, websites, etc.
- We can easily integrate Kafka with Spark using Spark Streaming APIs
- In this demo, we will combine the capabilities of Kafka with Spark
- This helps organizations to handle and process real-time data at the same time
- In this demo, we will count the frequency of words in the arriving data
- A document is provided in course material about this demo

Streaming Using Netcat Server

Streaming Using Netcat Server

- Netcat (often abbreviated as “nc”) is a computer networking utility for reading from and writing to network connections using TCP or UDP
- It is designed to be a dependable back-end utility that can be used directly or easily driven by other programs and scripts
- At the same time, it is a feature-rich network debugging and investigation tool, since it can produce almost any kind of connection its user needs and has several built-in capabilities
- We can fetch real-time data from a Netcat server using Spark Streaming
- In this demo, We will count the number of words of receiving data streams
- A document is provided in this course explaining all the steps

Spark and Flume Integration

Spark and Flume Integration

- Big Data, as we know, is a collection of large datasets that cannot be processed using traditional computing techniques
- Generally, most of the data that is to be analyzed will be produced by various data sources like applications servers, social networking sites, cloud servers
- This data will be in the form of log files and events
- Apache Flume is a tool/data ingestion mechanism for collecting aggregating and transporting large amounts of streaming data such as log data, events, etc from various web servers to a centralized data store
- We can use Flume with Spark for enhanced data processing. If we combine the capabilities of Spark and Flume, they can process data with a more powerful mechanism
- A document is provided in this course explaining all the steps

Quiz!

Question #1

Which of the following is a basic abstraction of Spark Streaming?

A Shared variable

B RDD

C Dstream

D All of the above

Answer #1

Which of the following is a basic abstraction of Spark Streaming?

- **A** Shared variable
- **B** RDD
- [✓ **C** **Dstream**]
- **D** All of the above

Question #2

Which is a distributed stream processing framework on top of Spark?

A	Spark Streaming
B	MLlib
C	Spark SQL
D	Spark R

Answer #2

Which is a distributed stream processing framework on top of Spark?

A	**Spark Streaming** ✓
B	MLlib
C	Spark SQL
D	Spark R

+91-9030485102

rganesh0203@gmail.com

https://topmate.io/rganesh_0203