CARLSON SCHOOL OF MANAGEMENT
UNIVERSITY OF MINNESOTA

# Introduction to Spark Streaming

MSBA 6330 Prof Liu



## Learning Objectives

- Streaming applications and their typical architecture
- How to enable streaming in spark and connect to a streaming data source
- The concept of DStream and its operations



## Topics

- Overview of Streaming Processing
- Intro to Spark Streaming
- Creating StreamingContext and Input DStreams
- DStream Operations
- Spark SQL and DStream
- A Streaming Wordcount Example
- A Twitter Hashtag Monitor

Carlson School of Management

Introduction to Spark Streaming

# OVERVIEW OF STREAMING PROCESSING

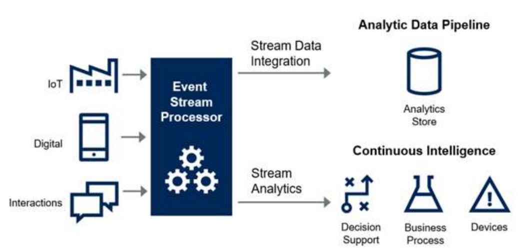
Carlson School of Management

## Streaming Applications

- **Stream processing**: real-time or near-real-time calculations on event data "in motion".
- Many big data applications need to process large data streams in real time, such as
  - Continuous ETL
  - Internet of Things (IoT)
  - Website monitoring
  - Fraud detection
  - Advertisement monetization
  - Social media analysis
  - Financial market trends
- According to Gartner, by 2022, more than half of major new business systems will incorporate real-time context data to improve decisions

Carlson School of Management

## Two Uses of Streaming Processing




Source: Gartner
IoT = Internet of Things
ID: 367575

Carlson School of Management

## Streaming versus Batch

- Advantages of Stream Processing
  - **Low latency**: respond quickly (minutes, seconds, or milliseconds)
  - **Efficiency**: streaming processing incrementalizes the computation.
    - May be more efficient than repeated batch jobs
- Challenges of Stream Processing
  - Diverse data formats (Json, Avro, Binary)
  - Data can be dirty, late, out-of-order
  - Complex workload: combine streaming with interactive queries, ML
  - Complex systems: more difficult to develop, understand, troubleshoot; diverse storage systems (Kafka, S3, Kinesis, RDBMS)

Carlson School of Management

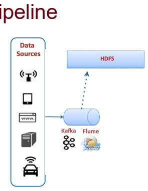
## Streaming Architecture (1) Source

- Sources of Data Streams
  - Transactional data (e.g., orders and claims)
  - Click streams
  - Server logs, API calls
  - Social media/news
  - Weather feeds
  - Market data
  - Internet of Things / sensor data
  - Data generated by smart devices, self-driving cars


Carlson School of Management

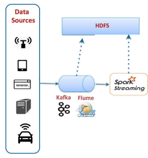
## Streaming Architecture (2) Streaming Pipeline

- Source data is delivered through a streaming pipeline system
- **Streaming pipeline**: served as a message queue and broker between senders and receivers.
  - Store, route, transform, aggregate, decompose, recompose, queue, etc.
- Leading streaming pipeline applications and protocols
  - Apache Kafka, Apache Flume, Amazon SQS, Amazon Kinesis, RabbitMQ
  - Messaging Protocols: MQTT, STOMP, AMQP



## Streaming Architecture (3) Stream Processing

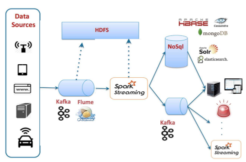
- The data is then processed by a stream processing system
  - transform
  - aggregate
  - decision making support
  - machine learning
  - …
- Popular streaming processing tools include
  - Apache Storm, Spark Streaming, Apache Flink, Kinesis Data Analytics, Kafka Streams






## Streaming Architecture (4) Output

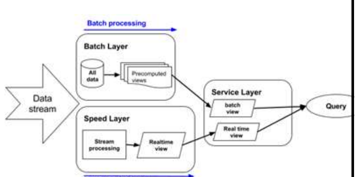
- Output to:
  - File systems, e.g. HDFS / S3
    - E.g., For further batch processing
  - NoSQL: e.g., Hbase, Amazon DynamoDB, MongoDB
    - High throughput; for data integration, query, search, visualization etc.
  - Search, e.g. ElasticSearch
  - SQL Databases (e.g. Hive)
    - For business intelligence, query and reporting etc.
  - Live dashboards
  - Other streaming pipelines: Kafka, Kinesis etc
    - For further processing






## Streaming & Lambda Architecture

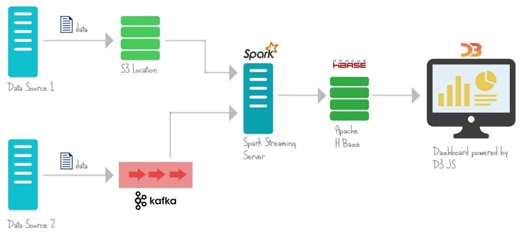
- Nathan Marz came up with the term **Lambda Architecture** ($\lambda$ -architecture) for a generic, scalable and fault-tolerant data processing architecture.
- All **data** entering the system is dispatched to both the batch layer and the speed layer for processing.
- The **batch layer** has two functions: (i) managing the master dataset (an immutable, append-only set of raw data), and (ii) to pre-compute the batch views.
- The **speed layer** deals with recent data only.
- Any incoming **query** can be answered by merging results from batch views and real-time views.




Many real-world big data architectures implement Lambda architecture design.

Carlson School of Management

## Sample Architecture 1 – Real-time Monitoring

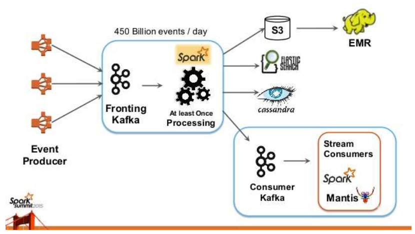



Source: https://www.sigmoid.com/integrating-spark-kafka-hbase-to-power-a-real-time-dashboard/

Carlson School of Management

## Sample Architecture 2 – Netflix Streaming

- Use for A/B testing, movie recommendation etc.




Lambda architecture used

https://sparkhub.databricks.com/video/spark-and-spark-streaming-at-netflix/

Carlson School of Management

Introduction to Spark Streaming

# INTRO TO SPARK STREAMING

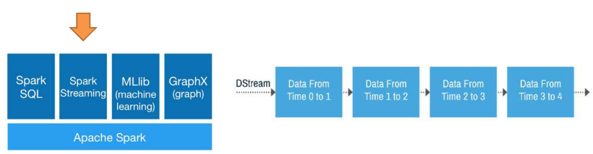
## Two Approaches of Distributed Stream Processing

- Continuous processing
  - Read/process records one by one as soon as they arrive.
  - Pro: lowest latency possible
  - Con: low throughput, due to significant overhead per-record.
  - **Examples**: Apache Storm, Kafka Streams, Apache Flink, Kinesis Streams
- Micro-batching
  - System wait to accumulate small batches of records (e.g. 1 second worth), then process it like a batch job
  - Pro: high throughput due to batch-based optimization, no per record overhead.
  - Con: latency is relatively high (0.5~1 seconds)
  - **Examples**: Apache Spark Streaming, Apache Flink



## What Is Spark Streaming?

- An extension of core Spark
- Provides **real-time** data processing
- Segments an incoming stream of data into micro-batches






## Evolution of of Spark Streaming

Message delivery guarantees in streaming systems:
**At most once**: data may be lost but no duplicates
**At least once**: no lost but may be duplicated
**Exactly once**: Can be neither lost or duplicated

- The Spark Streaming (DStream) APIs
  - Introduced in 2012, based on sequence of **RDDs**.
  - Only supports micro-batching
  - Popular stream processing tool used by many organizations
- The Structured Streaming APIs
  - Introduced in 2016, based on sequence of **DataFrames**
  - Meant to be easier-to-use, higher-performance evolution of DStream API
  - Supports both micro-batching (exactly once) and continuous processing (at least once)
  - Stable release in Spark 2.2 (July 11, 2017)
  - Continuous processing introduced in Spark 2.3 (February 28, 2018)

You can find spark streaming API documentation under pyspark.sql.streaming

Introduction to Spark Streaming

## CREATING STREAMINGCONTEXT AND INPUT DSTREAMS



### Create a Spark StreamingContext

- Entry point to all streaming functionality
- Can create new `StreamingContext` from existing `SparkContext` or `SparkConf`

Scala

```scala
import org.apache.spark._
import org.apache.spark.streaming._
val conf = new SparkConf().setAppName(appName).setMaster(master)
val ssc = new StreamingContext(conf, Seconds(1))
```

Python

```python
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
sc = SparkContext("local[2]", "networkWordCount")
ssc = StreamingContext(sc, 1)  # 1 second intervals
```

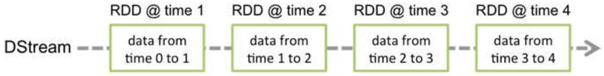
  - '`networkWorldCount`' is the name of your application
  - "`local[2]`" is the master (2 threads).
  - 1 second is the batch interval

One thread (core) will be used to run the receiver and at least one more is necessary for processing the received data. For a cluster, the number of allocated threads (cores) must be more than the number of receivers, otherwise the system can not process the data.



### Spark DStream

- Spark Streaming divides the data stream into micro batches called **discretized stream**, or DStream.
  - A DStream is a sequence of mini-batches, each of which is represented as a Spark RDD.



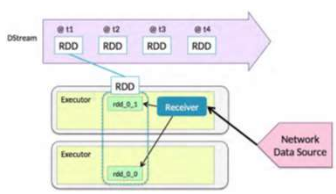
## Input DStream

- Input DStreams are the stream of input data from streaming sources
  - **Basic sources**: directly available through `StreamingContext`
    - file systems: continuously read new files in a folder
    - Network socket: continuously receive content from network socket
  - **Advanced sources**: extra utility classes (depending on some external jars)
    - Kafka, Flume, Kinesis, Twitter, etc.



## Input DStream and Receiver

- Most input DStreams are based on **receivers (**one exception is File Stream)
- Network data is received on an executor node
- Receiver distributes data (RDDs) to the cluster as partitions






## Create Input DStreams

- Basic sources
  - `textFileStream(dataDirectory)`
  - `socketTextStream(hostname, port)`

```
lines = ssc.socketTextStream("localhost", 9999)
```

Using input from port 9999 on the local host as an input DStream

- Advanced sources
  - Kafka , Kinesis, Flume (supported in pyspark)

```
from pyspark.streaming.kafka import KafkaUtils
kafkaStream = KafkaUtils.createStream(ssc, "zkquorum", "group_id", {"topic":1})
```

zk_quorum: zookeeper quorum
{topic_id:# of partitions}

http://spark.apache.org/docs/latest/api/python/pyspark.streaming.html#pyspark-streaming-kafka-module
http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.streaming.package

## Multiple Input DStreams

- It is possible to create multiple input DStreams
  - multiple receivers will be created to receive data from multiple sources simultaneously
  - but a Spark Streaming application must be assigned more cores than receivers to receive and process data at the same time.

Carlson School of Management

Introduction to Spark Streaming

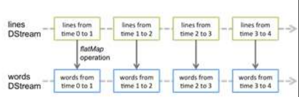
# DSTREAM OPERATIONS

Carlson School of Management

## Spark DStream Operations

- DStream operations are applied to every RDD in the stream
  - Executed once per duration
- Two types of DStream operations
  - **Transformations**
    - Create a new DStream from an existing one
  - **Output operations**
    - Write data (for example, to a file system, database, or console)
    - Similar to RDD *actions*

DStream Transformation




```
# Split each line into words
words = lines.flatMap(lambda line: line.split(" "))
```

Carlson School of Management

## DStream Transformations

- Many RDD transformations are also available on DStream
  - map, flatMap, filter, count, reduce, countByValue, reduceByKey, …

```
# Split each line into words
words = lines.flatMap(lambda line: line.split(" "))
```

Note that each input is an element in RDD

- To use arbitrary RDD transformations
  - transform(function): Return a new DStream by applying a RDD-to-RDD function to every RDD of the source DStream.
  - This enables very powerful possibilities. For example, one can do real-time data cleaning by joining the input data stream with precomputed spam information (maybe generated with Spark as well) and then filtering based on it.

```
# join data stream with spam information to do data cleaning
cleanedDStream = wordCounts.transform(lambda rdd: rdd.join(spamInfoRDD).filter(...))
```

Note that each input is an RDD

Carlson School of Management

## DStream Output Operations

- Processed results are pushed out in batches
  - Output operations allow DStream's data to be pushed out to external systems like a database or a file systems.
  - They trigger the actual execution of all the DStream transformations (similar to actions for RDDs)


Carlson School of Management

## DStream Output Operations

- Console Output
  - print() in scala; pprint() in python: print first 10 elements (default) in every batch

```
# Print the first 5 elements of each RDD generated in this DStream to the console
wordCounts.pprint(5)
```

- File Output
  - saveAsTextFiles(prefix, [suffix]): Save each RDD in this DStream as at text file. The file name at each batch interval is generated based on prefix and suffix: "prefix-TIME_IN_MS[.suffix]".

## DStream Output Operations (cont.)

Click links to see examples

- Arbitrary output
  - `foreachRDD(func)`: apply a function to each RDD in the Dstream
    - this can be used to send data to a NoSQL database (Hbase, DynamoDB, etc), RDBMS (hive, mysql etc, may not be ideal for high speed data), live dashboard, or a stream (Kafka, Kinesis etc).
    - It is not a good idea to create a connection to external source for each record in the RDD because of the time and resource overheads associated with each connection. Instead, you can send all records in one partition at once using `rdd.foreachPartition()`

```
def sendPartition(iter):
    connection = createNewConnection()
    for record in iter:
        connection.send(record)
    connection.close()
# send data to some external destinations
dstream.foreachRDD(lambda rdd: rdd.foreachPartition(sendPartition))
# save as Hive tmp table
dstream.foreachRDD(lambda rdd: rdd.toDF().registerTempTable('tmpTable'))
# then you can use SparkSQL to load data into respective hive table
```



Introduction to Spark Streaming

# SPARK SQL AND DSTREAM



## SQL Operations on DStreams

- Allow Spark Streaming to perform SQL style operations on streaming data
  - E.g. filtering, sorting, aggregating, filtering, joining, etc
- Allow Spark Streaming to access the SQL compatible data output methods (via `foreachRDD`), e.g.
  - `registerTempTable, createOrReplaceTempView`
  - `saveAsTable`

## DataFrame and SQL Operations

- What is the pump vendor and maintenance information for sensors with low pressure alerts?

```
# bring in static data
sc.textFile("vendor.csv").map(parsePump).toDF().registerTempTable("pump")
sc.textFile("maint.csv").map(parseMaint).toDF().registerTempTable("maint")

# join DStream data with static data using Spark SQL
def showAlert(rdd):
  sqlContext = SQLContext.getOrCreate(rdd.sparkContext)
  rdd.filter(lambda sensor:sensor.psi < 5.0).toDF().registerTempTable("alert")
  alertpumpmaint = sqlContext.sql( \
    "select ... from alert a join pump p on ... join m on ...")
  alertpumpmaint.show()

# show alert for each microbatch.
sensorDStream.foreachRDD(showAlert)
```



Introduction to Spark Streaming

# A STREAMING WORDCOUNT EXAMPLE



## Example – Streaming Wordcount (Demo using Jupyter)

- Input Stream: `socketTextStream`

```
# Open a terminal, run:
nc -lk 9999
```

  - Type words into the terminal to simulate text streaming over the network (port 9999)
- Spark streaming listens to port 9999, the does a word for each RDD in the DStream

DStream processing won't start until ssc.start()

```
from pyspark.streaming import StreamingContext
ssc = StreamingContext(sc, 1)  #create new streaming context
lines = ssc.socketTextStream("localhost", 9999) # input stream
lines.flatMap(lambda l: l.split(" ")) \
.map(lambda w: (w, 1)) \
.reduceByKey(lambda x, y: x + y) \
.pprint()  #DStream transformation and output operations
ssc.start()  #start the stream listener
```

To terminate stream processing within an interactive shell:

```
ssc.stop(False)  # stop streaming context without terminating sc.
```

## Streaming Application Patterns

Spark Streaming is intended as **standalone application** (not an interactive application)

1. Create a Spark Context `sc`
2. Create a StreamingContext object `ssc`.
3. Define the input source by creating an input DStream
4. Define the streaming computations by applying transformations and output operations to DStreams
5. Start receiving data and processing it using `ssc.start()`
6. Wait for the processing to be stopped using `ssc.awaitTermination()`
   – manually stop streaming using `ssc.stop()`

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
sc = SparkContext("local[2]", "NetworkWordCount")
ssc = StreamingContext(sc, 1)
lines = ssc.socketTextStream("localhost", 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()             # Start the computation
ssc.awaitTermination()  # Wait for the computation to
terminate
```



## Note about DStream Processing

- Once started, the DStream processor will continue indefinitely in its own process (unless fatal error occurs or stopped by `stop` from a different process)
- Once a context has been started, no new streaming computations can be set up or added to it.
- Once a context has been stopped, it cannot be restarted
- Only one `StreamContext` can be active in any JVM
- `stop()` also stops `SparkContext`. Use `stop(false)` if you want to keep `SparkContext` live after stream `listener` is stopped.

For an experimental DStream app that runs for a prespecified period of time, you may use sleep (import time)

```
import time
ssc.start()
time.sleep(100)
ssc.stop(false)
```

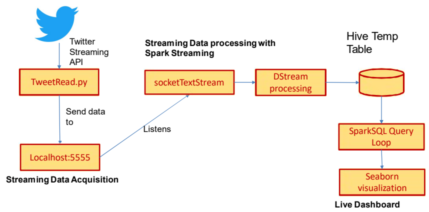


Introduction to Spark Streaming

## A TWITTER HASHTAG MONITOR

## Case Study: Twitter Hashtag Monitor

- System Architecture




Carlson School of Management

## Twitter Data Acquisition

To start (from terminal):
`python TweetRead.py`
To end (from Jupyter):
`!pkill -f 'python TweetRead.py'`

- Use tweepy package `pip install tweepy`
- `Tweepy stream's on_data`: parse tweet and publish to socket

```
msg = json.loads( data )
print( msg['text'].encode('utf-8') )
self.client_socket.send( msg['text'].encode('utf-8') )
```

- Set up socket

```
s = socket.socket()         # Create a socket object
s.bind(("127.0.0.1", 5555)) # Bind to the port
s.listen(5)                 # Now wait for client connection.
c, addr = s.accept()        # Establish connection with client.
print( "Received request from: " + str( addr ) )
```

- Set up tweepy streaming application and run it

```
#set up twitter API authentication
auth = OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_token, access_secret)
#Configure Stream Listener
twitter_stream = Stream(auth, TweetsListener(c))
twitter_stream.filter(track=['trump']) #Start Listening
```

Carlson School of Management

## Process Twitter Data using Spark Streaming

- Set up a socket stream listener
- Set up a sliding window of 30 seconds with 10 second refreshing interval
- Extract hash tags from incoming tweets, count tag frequency, and save results as a Hive Temp Table

```
ssc = StreamingContext(sc, 10 ) # 10 second batch interval
socket_stream = ssc.socketTextStream("127.0.0.1", 5555)
lines = socket_stream.window(30,10) # using a sliding window

def processRdd(rdd):
    (rdd.flatMap( lambda text: text.split( " " ) ) #Splits to a list
      .filter( lambda word: word.lower().startswith("#") ) # Checks for hashtag calls
      .map( lambda word: ( word.lower(), 1 ) ) # Lower cases the word
      .reduceByKey( lambda a, b: a + b ) # Reduces
      .map( lambda rec: Row(tag=rec[0], count=rec[1])) # Stores in a Tweet Object
      .toDF().sort(desc("count")) # Sorts Them in a DF
      .limit(10).registerTempTable("tweets")) # Registers to a table.

lines.foreachRDD(processRdd) # adding processing logic
ssc.start()  # start the streaming application
```

To manually end:
ssc.stop()

## Hashtag count live dashboard

- Using SparkSQL to obtain data, then convert to local Pandas DataFrame
- Use seaborn's plt to update output every 3 seconds for 60 cycles.

```
import time
from IPython import display
import matplotlib.pyplot as plt
import seaborn as sns
# Only works for Jupyter Notebooks!
%matplotlib inline

count = 0
while count < 60: # 60 sleep cycles

    time.sleep( 3 )   # refresh every 3 seconds
    top_10_tweets = sqlContext.sql( 'Select tag, count from tweets')
    top_10_df = top_10_tweets.toPandas() # to pandas dataframe
    display.clear_output(wait=True) #clear output when new output is available
    sns.plt.figure( figsize = ( 10, 8 ) )
    sns.barplot( x="count", y="tag", data=top_10_df)
    sns.plt.show()
    count = count + 1
```



## Summary: Spark Streaming Key Concepts

- Role of stream processing in continuous intelligence and ETL
- Continuous processing vs Micro-batching
- Lambda architecture
- DStream Input Sources:
  - File based: HDFS
  - Network based: TCP sockets
  - Advanced sources: Twitter, Kafka, Flume, etc
- Transformations
  - Standard RDD operations
  - Flexible operation: transform
  - Windowed operations: countByValueAndWindow,…
- Output operations: trigger computation
  - print/pprint – print first 10 elements
  - saveAsTextFiles – save to text files
  - forEachRDD – do anything with each batch of RDDs