Apache Spark

Session 10 - Spark Streaming

WELCOME - KNOWBIGDATA

- ❑ Expert Instructors
- ❑ CloudLabs
- ❑ Lifetime access to LMS
  - ❑ Presentations
  - ❑ Class Recording
  - ❑ Assignments + Quizzes
  - ❑ Project Work
- ❑ Real Life Project
- ❑ Course Completion Certificate
- ❑ 24x7 support
- ❑ KnowBigData - Alumni
  - ❑ Jobs
  - ❑ Stay Abreast (Updated Content, Complimentary Sessions)
  - ❑ Stay Connected

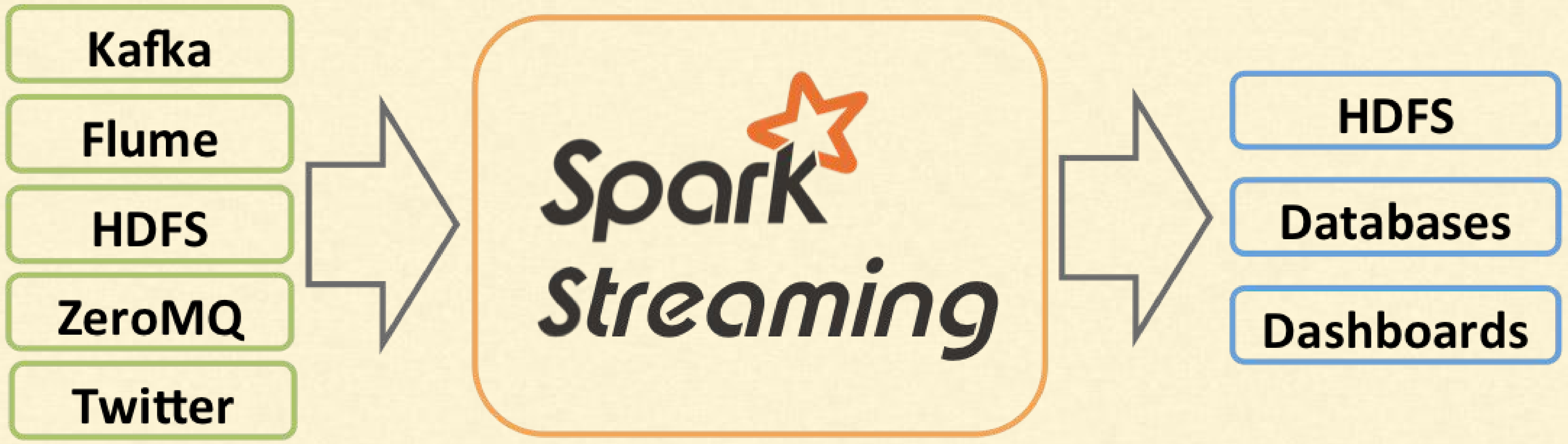
COURSE CONTENT

About Instructor?

2014	**KnowBigData**	Founded
2014 – 2012	**Amazon**	Built High Throughput Systems for Amazon.com site using in-house NoSql.
2012	**InMobi**	Built Recommender that churns 200 TB
2011 – 2006	**tBits Global**	Founded tBits Global Built an enterprise grade Document Management System
2006 – 2002	**D.E.Shaw**	Built the big data systems before the term was coined
2002	**IIT Roorkee**	Finished B.Tech.

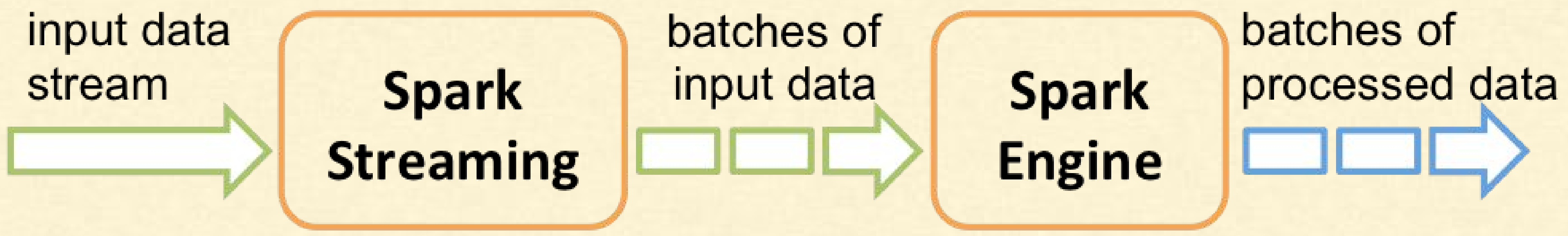
SPARK STREAMING

Extension of the core Spark API:
high-throughput, fault-tolerant



SPARK STREAMING

Provides a discretized stream or DStream - a continuous stream of data.

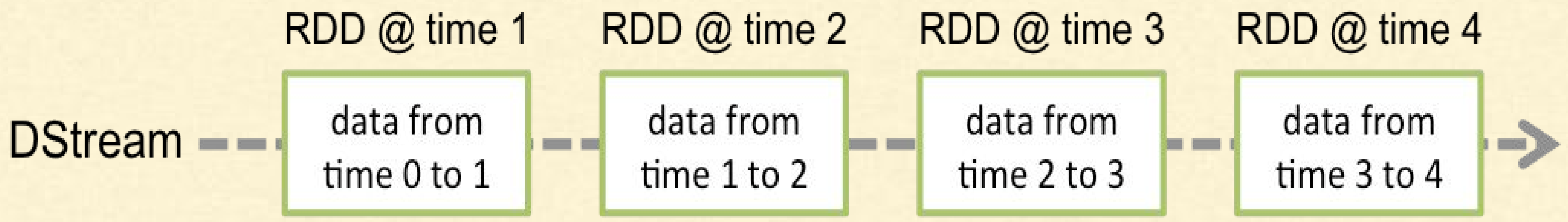


Workflow

- Spark Streaming receives live input data streams
- Divides the data into batches
- Spark Engine generates the final stream of results in batches.

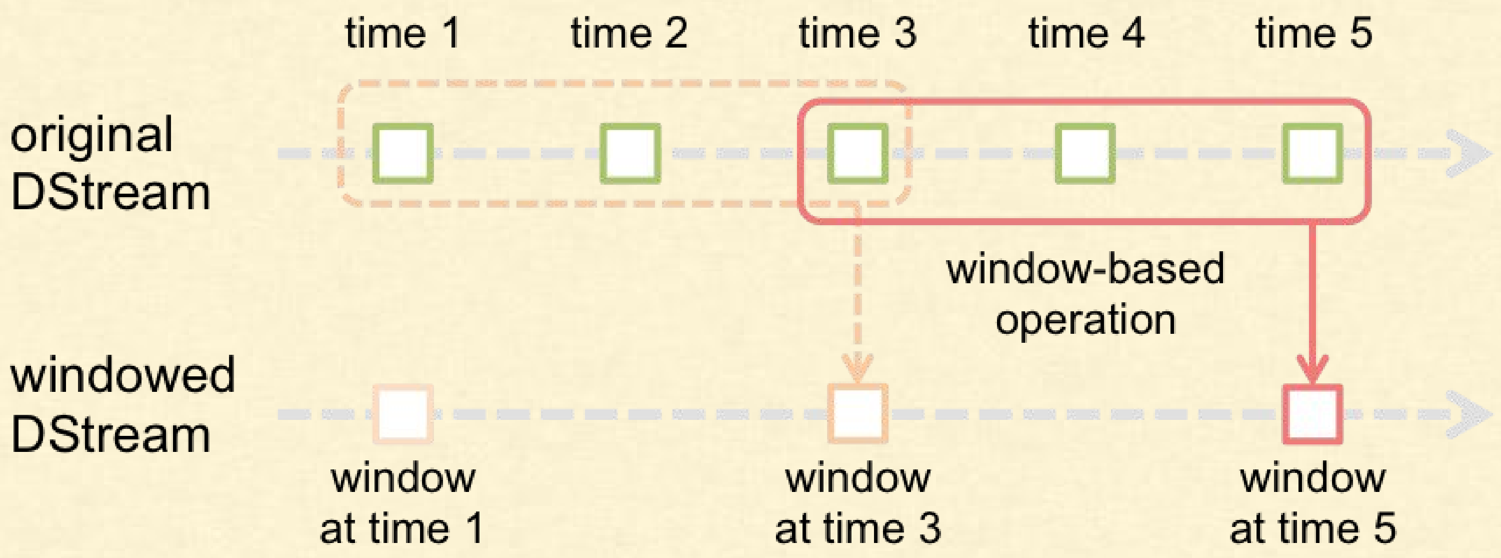
SPARK STREAMING - DSTREAM

Internally represented using RDD



Each RDD in a DStream contains data from a certain interval.

Window Operations



```
pairs.reduceByKeyAndWindow(reduceFunc, new Duration(30000), new Duration(10000));
// Reduce last 30 seconds of data, every 10 seconds
```

SPARK STREAMING - EXAMPLE

Problem: do the word count every second.

Step 1:

Create a connection to the service

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

# Create a local StreamingContext with two working thread and
# batch interval of 1 second
sc = SparkContext("local[2]", "NetworkWordCount")
ssc = StreamingContext(sc, 1)
# Create a DStream that will connect to hostname:port,
# like localhost:9999
lines = ssc.socketTextStream("localhost", 9999)
```

SPARK STREAMING - EXAMPLE

Problem: do the word count every second.

Step 2:

Split each line into words, convert to tuple and then count.

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

# Count each word in each batch
pairs = words.map(lambda word: (word, 1))

#Do the count
wordCounts = pairs.reduceByKey(lambda x, y: x + y)
```

SPARK STREAMING - EXAMPLE

Problem: do the word count every second.

Step 3:

Print the stream. It is a periodic event

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

SPARK STREAMING - EXAMPLE

Problem: do the word count every second.

Step 4:

Every Thing Setup: Lets Start

```
# Start the computation
ssc.start()

# Wait for the computation to terminate
ssc.awaitTermination()
```

SPARK STREAMING - EXAMPLE

Problem: do the word count every second.

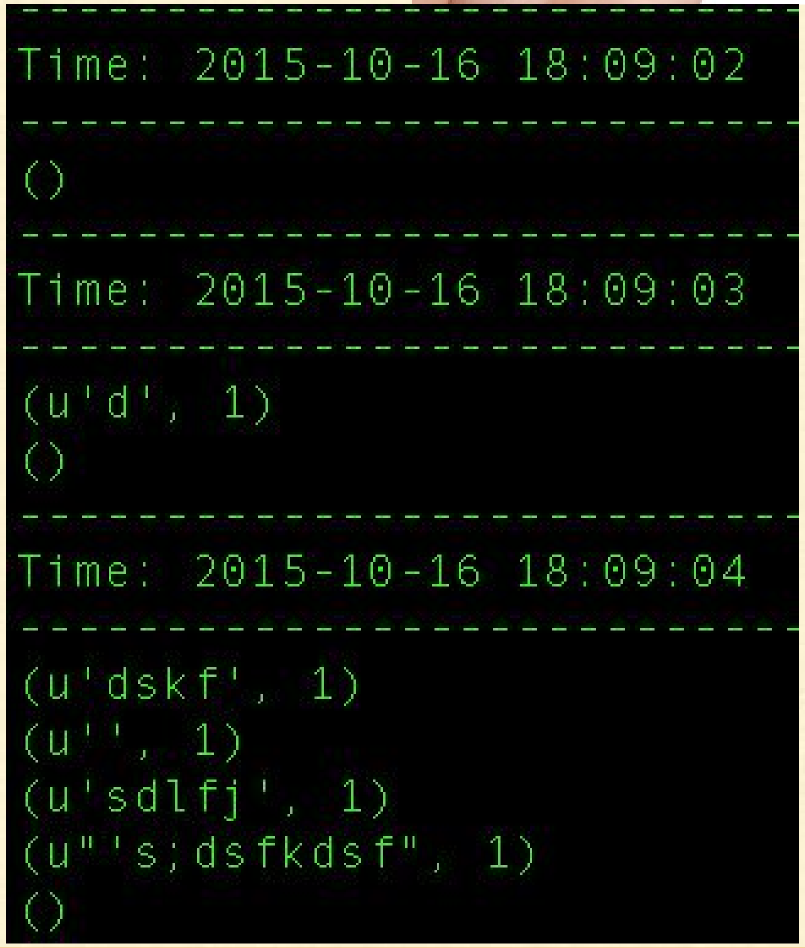
```
spark-submit spark_streaming_ex.py
2>/dev/null
```

(Also available in HDFS at /data/spark)

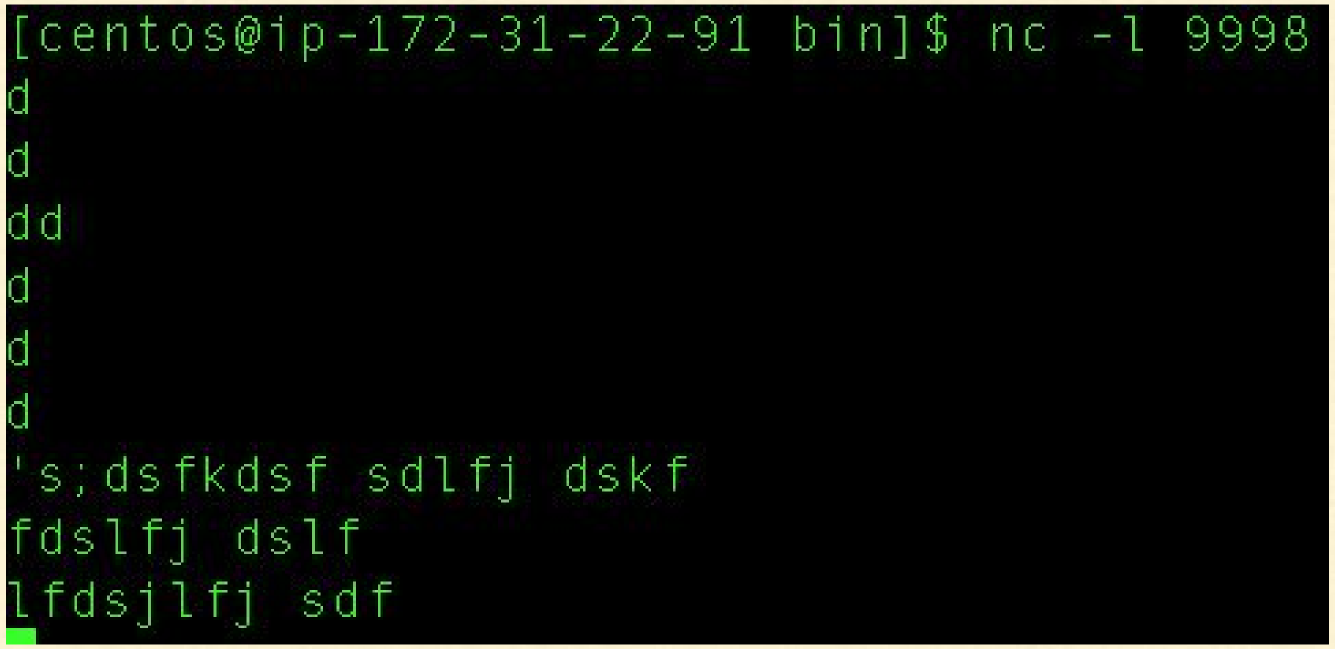
```
nc -l 9999
```

SPARK STREAMING - EXAMPLE

Problem: do the word count every second.



```
-------------------------------------------
Time: 2015-10-16 18:09:02
-------------------------------------------
()
-------------------------------------------
Time: 2015-10-16 18:09:03
-------------------------------------------
(u'd', 1)
()
-------------------------------------------
Time: 2015-10-16 18:09:04
-------------------------------------------
(u'dskf', 1)
(u'', 1)
(u'sdlfj', 1)
(u"'s;dsfkdsf", 1)
()
```



```
[centos@ip-172-31-22-91 bin]$ nc -l 9998
d
d
dd
d
d
d
's;dsfkdsf sdlfj dskf
fdslfj dslf
lfdsjlfj sdf
```

SPARK STREAMING - EXAMPLE

Problem: do the word count every second.

```
spark-submit spark_streaming_ex.py
2>/dev/null
```

```
yes|nc -l 9999
```

Basic Concepts - Maven Getting Started

Maven is a build tools that helps in managing/acquire dependencies
It is used for Java/Scala projects

To Get started with maven:

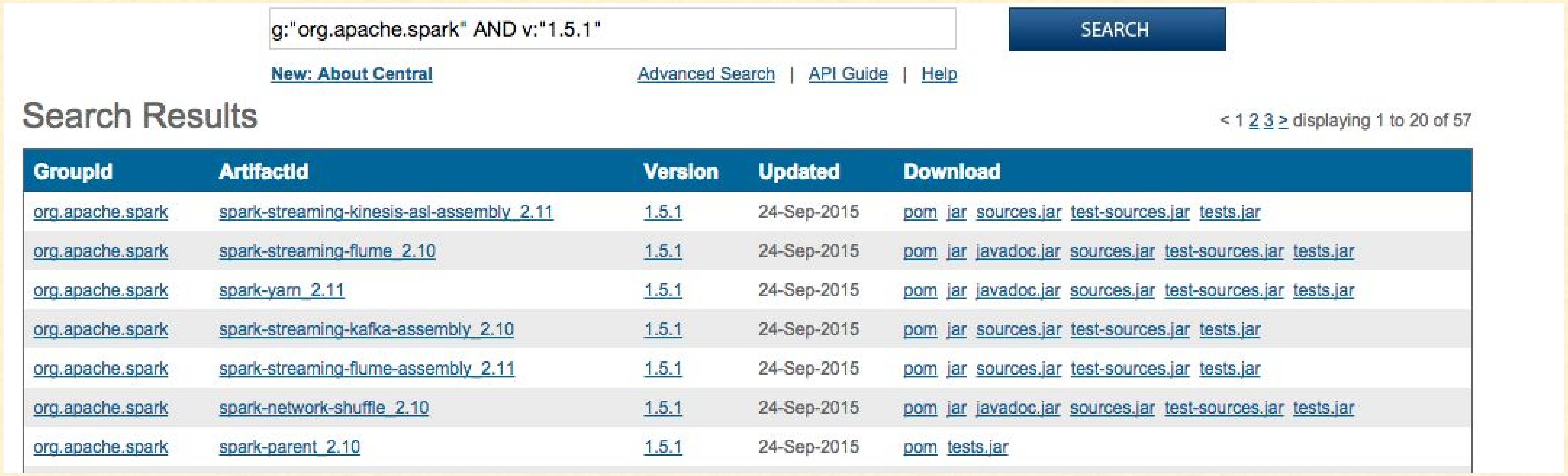
1. install maven
2. Create project:
   ```
   mvn archetype:generate -DgroupId=com.app -DartifactId=my-app -DarchetypeArtifactId=maven-archetype-quickstart  -DinteractiveMode=false
   ```
3. Edit pom.xml and add dependencies, e.g.:
   ```
   <dependency>
       <groupId>org.apache.spark</groupId>
       <artifactId>spark-streaming_2.10</artifactId>
       <version>1.5.2</version>
   </dependency>
   ```
4. To Generate Jars: *mvn package*

Basic Concepts - Linking

Source	Artifact
Kafka	spark-streaming-kafka_2.10
Flume	spark-streaming-flume_2.10
Kinesis	spark-streaming-kinesis-asl_2.10 [Amazon Software License]
Twitter	spark-streaming-twitter_2.10
ZeroMQ	spark-streaming-zeromq_2.10
MQTT	spark-streaming-mqtt_2.10

Basic Concepts - Linking

For python, it is better to download the jars binaries from maven repository directly



g:"org.apache.spark" AND v:"1.5.1"

SEARCH

New: About Central

Advanced Search | API Guide | Help

Search Results

< 1 2 3 > displaying 1 to 20 of 57

GroupId	ArtifactId	Version	Updated	Download
org.apache.spark	spark-streaming-kinesis-asl-assembly_2.11	1.5.1	24-Sep-2015	pom jar sources.jar test-sources.jar tests.jar
org.apache.spark	spark-streaming-flume_2.10	1.5.1	24-Sep-2015	pom jar javadoc.jar sources.jar test-sources.jar tests.jar
org.apache.spark	spark-yarn_2.11	1.5.1	24-Sep-2015	pom jar javadoc.jar sources.jar test-sources.jar tests.jar
org.apache.spark	spark-streaming-kafka-assembly_2.10	1.5.1	24-Sep-2015	pom jar sources.jar test-sources.jar tests.jar
org.apache.spark	spark-streaming-flume-assembly_2.11	1.5.1	24-Sep-2015	pom jar sources.jar test-sources.jar tests.jar
org.apache.spark	spark-network-shuffle_2.10	1.5.1	24-Sep-2015	pom jar javadoc.jar sources.jar test-sources.jar tests.jar
org.apache.spark	spark-parent_2.10	1.5.1	24-Sep-2015	pom tests.jar

Initializing StreamingContext

```
from pyspark import SparkContext

from pyspark.streaming import StreamingContext

sc = SparkContext(master, appName)

ssc = StreamingContext(sc, 1)
```

Notes:

- StreamingContext is created from sc
- appname is the name that you want to show in UI
- batch interval must be set based on latency requirements

Initializing StreamingContext

After a context is defined, you have to do the following.

1. Define the input sources by creating input DStreams.
2. Define the streaming computations by applying transformation and output operations to DStreams.
3. Start receiving data and processing it using streamingContext.start().
4. Wait for the processing to be stopped (manually or due to any error) using streamingContext.awaitTermination().
5. The processing can be manually stopped using streamingContext.stop().

Initializing StreamingContext

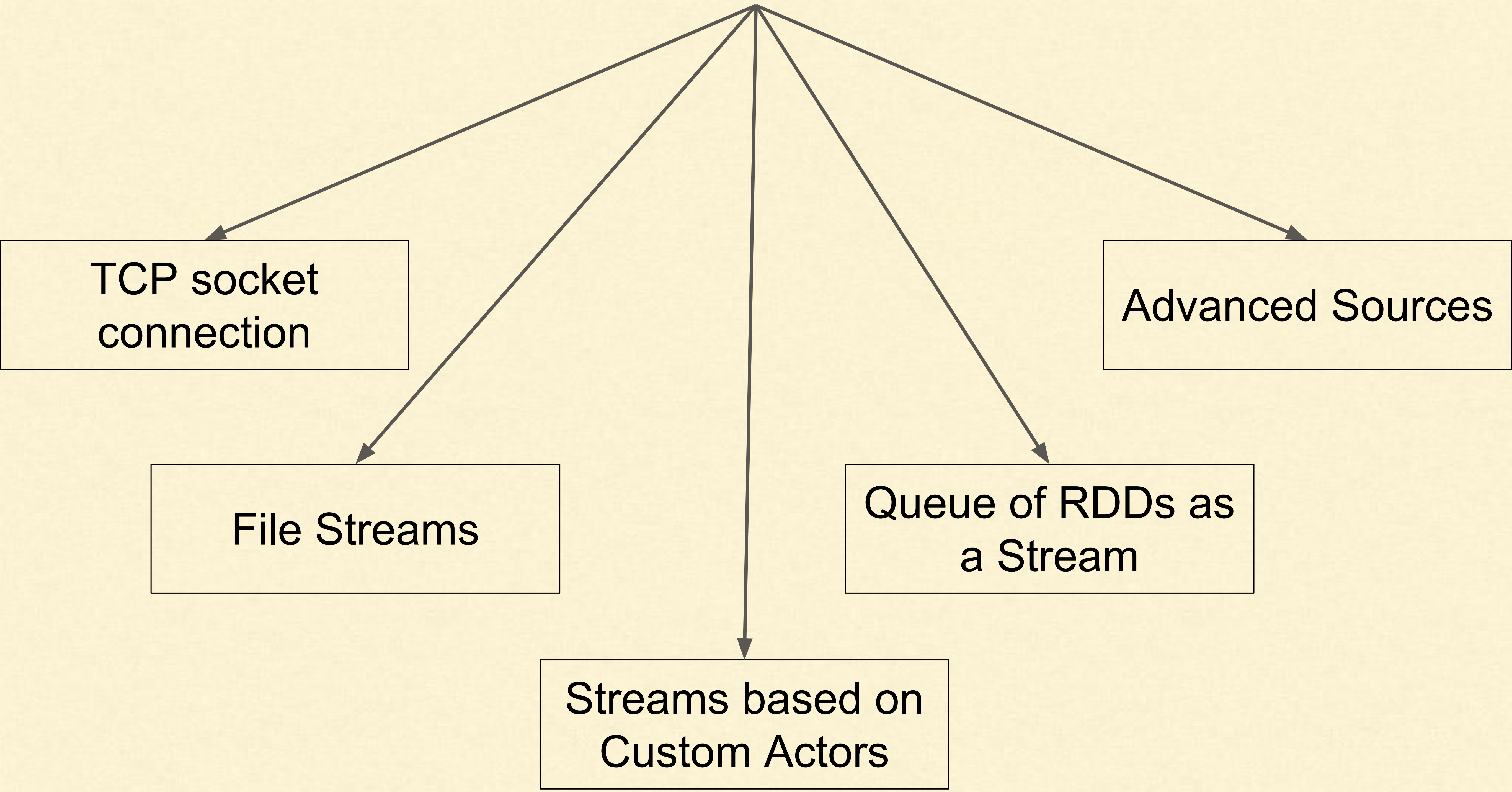
For running locally,

- Do not use "local" or "local[1]" as the master URL. Only one thread is used.
  - It leaves no thread for processing the received data.
- So, Always use "local[n]" as the master URL , where n > no. of receivers to run.

In cluster,

- Number of cores allocated must be > no. of receivers.
- Else system will receive data, but not be able to process it.

Sources


TCP socket connection
File Streams
Streams based on Custom Actors
Queue of RDDs as a Stream
Advanced Sources

Sources - File Streams

streamingContext.textFileStream(dataDirectory)

- For reading data from files on any file system compatible with the HDFS API (that is, HDFS, S3, NFS, etc.)
- Spark Streaming will monitor the directory dataDirectory
- Process any files created in that directory.
- Directory within directory not supported
- The files must have the same data format.
- Files are created by moving or renaming them into the data directory
- Once moved, the files must not be changed
- In continuously appended files, new data isn't read

Streams based on Custom Actors

- Data streams received through Akka actors by using streamingContext.actorStream (actorProps, actor-name).
- Implement Custom Actors: https://spark.apache.org/docs/latest/streaming-custom-receivers.html
- actorStream is not available in the Python API.

Basic Sources - Queue of RDDs as a Stream

- For testing a Spark Streaming application with test data
- create a DStream based on a queue of RDDs
- using streamingContext.queueStream(queueOfRDDs)
- Each RDD pushed into the queue will be treated as a batch of data in the DStream

Advanced Sources

1. Interface with external non-Spark libraries, some of them with complex deps (Kafka & Flume)
2. As of Spark 1.5.1, Kafka, Kinesis, Flume and MQTT are available in the Python API.

Library	**Spark 1.5.1 works with**	**Documentation**
Kafka	0.8.2.1	Guide
Flume	1.6.0	Guide
Kinesis	1.2.1	Guide
Twitter	Twitter4j 3.0.3	Guide examples (TwitterPopularTags and TwitterAlgebirdCMS).

Spark Streaming + Kafka Integration

Apache Kafka

- Publish-subscribe messaging
- A distributed, partitioned, replicated commit log service.

Spark Streaming + Kafka Integration

Prerequisites

- Zookeeper
- Kafka
- Spark
- All of above are installed by Ambari with HDP
- Kafka Library - you need to download from maven
    - also available in /data/spark

Spark Streaming + Kafka Integration

Problem: do the word count every second from kafka

Step 1:

Download the spark assembly from here. Include essentials

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

from __future__ import print_function
import sys
```

Spark Streaming + Kafka Integration

Problem: do the word count every second from kafka

Step 2:

Create the streaming objects

```
sc = SparkContext(appName="KafkaWordCount")
ssc = StreamingContext(sc, 1)

#Read name of zk from arguments
zkQuorum, topic = sys.argv[1:]

#Listen to the topic
kvs = KafkaUtils.createStream(ssc, zkQuorum, "spark-
streaming-consumer", {topic: 1})
```

Spark Streaming + Kafka Integration

Problem: do the word count every second from kafka

Step 3:

Create the RDDs by Transformations & Actions

```
#read lines from stream
lines = kvs.map(lambda x: x[1])

# Split lines into words, words to tuples, reduce
counts = lines.flatMap(lambda line: line.split(" ")) \
.map(lambda word: (word, 1)) \
.reduceByKey(lambda a, b: a+b)

#Do the print
counts.pprint()
```

Spark Streaming + Kafka Integration

Problem: do the word count every second from kafka

Step 4:

Start the process

```
ssc.start()
ssc.awaitTermination()
```

Spark Streaming + Kafka Integration

Problem: do the word count every second from kafka

Step 5:

Create the topic

```
#Login via ssh or Console
ssh centos@e.cloudxlab.com
# Add following into path
export PATH=$PATH:/usr/hdp/current/kafka-broker/bin

#Create the topic
kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --
partitions 1 --topic test

#Check if created
kafka-topics.sh  --list --zookeeper localhost:2181
```

Spark Streaming + Kafka Integration

Problem: do the word count every second from kafka

Step 6:

Create the producer

```
# find the ip address of any broker from zookeeper-client using command get /brokers/ids/0
kafka-console-producer.sh --broker-list ip-172-31-13-154.ec2.internal:6667 --topic test

#Test if producing by consuming in another terminal
kafka-console-consumer.sh --zookeeper localhost:2181 --topic test --from-beginning

#Produce a lot
yes|kafka-console-producer.sh --broker-list ip-172-31-13-154.ec2.internal:6667 --topic test
```

Spark Streaming + Kafka Integration

Problem: do the word count every second from kafka

Step 7:

Do the stream processing. Check the graphs at :4040/

```
(spark-submit --jars spark-streaming-kafka-assembly_2.10-1.6.0.jar
kafka_wordcount.py localhost:2181 test) 2>/dev/null
```

Spark Streaming + Kafka Integration

Transformation	Meaning
map(*func*)	Return 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*)	Return a new DStream by selecting only the 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*)	Return a new DStream that contains the union of the elements in the source DStream and *otherDStream*.
count()	Return a new DStream of single-element RDDs by counting the number of elements in each RDD of the source DStream.

Spark Streaming + Kafka Integration

Transformation	Meaning
reduce(*func*)	Return a new DStream of single-element RDDs by aggregating the elements in each RDD of the source DStream using a function *func* (which takes two arguments and returns one). The function should be associative so that it can be computed in parallel.
countByValue()	When called on a DStream of elements of type K, return a new DStream of (K, Long) pairs where the value of each key is its frequency in each RDD of the source DStream.
reduceByKey(*func*, [*numTasks*])	When called on a DStream of (K, V) pairs, return a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function. **Note:** By default, this uses Spark's default number of parallel tasks (2 for local mode, and in cluster mode the number is determined by the config propertyspark.default.parallelism) to do the grouping. You can pass an optional numTasks argument to set a different number of tasks.
join(*otherStream*, [*numTasks*])	When called on two DStreams of (K, V) and (K, W) pairs, return a new DStream of (K, (V, W)) pairs with all pairs of elements for each key.

Spark Streaming + Kafka Integration

Transformation	Meaning
cogroup(*otherStream*, [*numTasks*])	When called on a DStream of (K, V) and (K, W) pairs, return a new DStream of (K, Seq[V], Seq[W]) tuples.
transform(*func*)	Return a new DStream by applying a RDD-to-RDD function to every RDD of the source DStream. This can be used to do arbitrary RDD operations on the DStream.
updateStateByKey(*func*)	Return a new "state" DStream where the state for each key is updated by applying the given function on the previous state of the key and the new values for the key. This can be used to maintain arbitrary state data for each key.

UpdateStateByKey Operation

- The updateStateByKey operation allows you to maintain arbitrary state while continuously updating it with new information.
- To use this, you will have to do two steps.
    - Define the state - The state can be an arbitrary data type.
    - Define the state update function - Specify with a function how to update the state using the previous state and the new values from an input stream
- In every batch, Spark will apply the state update function for all existing keys, regardless of whether they have new data in a batch or not.
- If the update function returns None then the key-value pair will be eliminated

UpdateStateByKey Operation

Objective: Maintain a running count of each word seen in a text data stream.
The running count is the state and it is an integer

```
def updateFunction(newValues, runningCount):
    if runningCount is None:
       runningCount = 0
    # add the new values with the previous running count
    # to get the new count
    return sum(newValues, runningCount)

runningCounts = pairs.updateStateByKey(updateFunction)
```

Transform Operation

- Applies arbitrary RDD-to-RDD functions on DStream (like transformWith)
- Apply any RDD operation that is not exposed in the DStream API

Example Usecase:

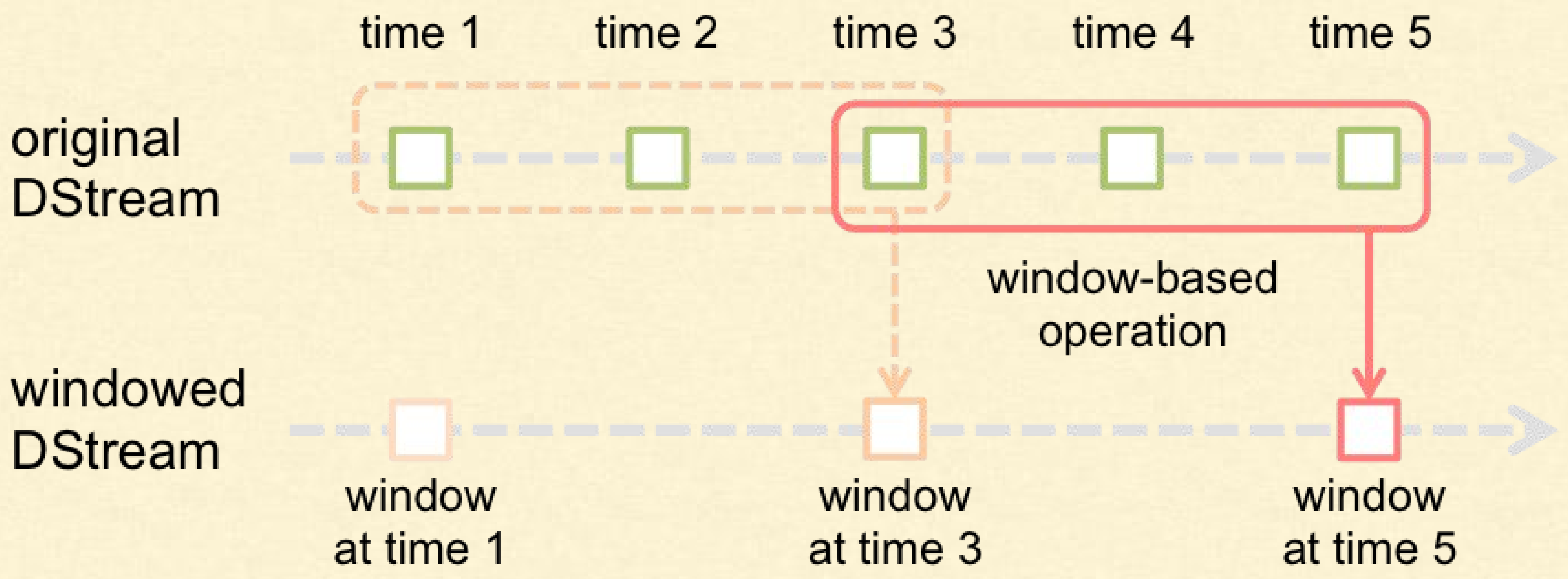
Do real-time data cleaning by joining the input data stream with precomputed spam information and filter based on it

```
# RDD containing spam information
spamInfoRDD = sc.pickleFile(...)

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

Window Operations

Apply transformations over a sliding window of data



Needs to specify two parameters:

1. window length - The duration of the window (3 in the figure).
2. sliding interval - The interval at which the window operation is performed (2 in the figure).
3. These two parameters must be multiples of the batch interval of the source DStream (1 in the figure).

Window Operations

```
# Reduce last 30 seconds of data, every 10 seconds
windowedWordCounts = pairs.reduceByKeyAndWindow(lambda x, y: x
+ y, lambda x, y: x - y, 30, 10)
```

Provides the operations such as:

- window(windowLength, slideInterval)
- countByWindow(windowLength, slideInterval)
- reduceByWindow(func, windowLength, slideInterval)
- reduceByKeyAndWindow(func, windowLength, slideInterval, [numTasks])
- reduceByKeyAndWindow(func, invFunc, windowLength, slideInterval, [numTasks])
- countByValueAndWindow(windowLength, slideInterval, [numTasks])

See More

Join Operations

Stream-stream joins

```
strm1 = ...
strm2 = ...
joinedStream = strm1.join(strm2)
```

Windowed Stream-stream joins

```
windowedStrm1 = strm1.window(20)
windowedStrm2 = strm2.window(60)
jndStrm = windowedStrm1.join(windowedStrm2)
```

Stream-dataset joins

- dataset = ... # some RDD
- windowedStream = stream.window(20)
- joinedStream = windowedStream.transform(lambda rdd: rdd.join(dataset))

Note: You can also do leftOuterJoin, rightOuterJoin, fullOuterJoin.

See More

Output Operations on DStreams

print()

Prints the first ten elements of every batch

pprint() in python

saveAsTextFiles(prefix, [suffix])

Save as text files.

File name at each batch interval is: "prefix-TIME_IN_MS[.suffix]".

Not available in the Python API.

saveAsObjectFiles(prefix, [suffix])

Saves as SequenceFiles of serialized Java objects. File name is same as

saveAsHadoopFiles(prefix, [suffix])

Saves as Hadoop files. File name is same as saveAsTextFiles()

Not available in the Python API.

Output Operations on DStreams

foreachRDD(func)

- Applies a function, func, to each RDD generated from the stream
- Should push the data in each RDD to an external system
- func is executed in the driver process

foreachRDD(func)

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

Note: This is wrong because it is forcing connection object to serialize and send. Instead use the below code.

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

Apache Spark

Thank you.

+1 419 665 3276 (US)
+91 803 959 1464 (IN)

reachus@knowbigdata.com

Subscribe to our Youtube channel for latest videos - https://www.youtube.com/channel/UCxugRFe5wETYA7nMH6VGyEA