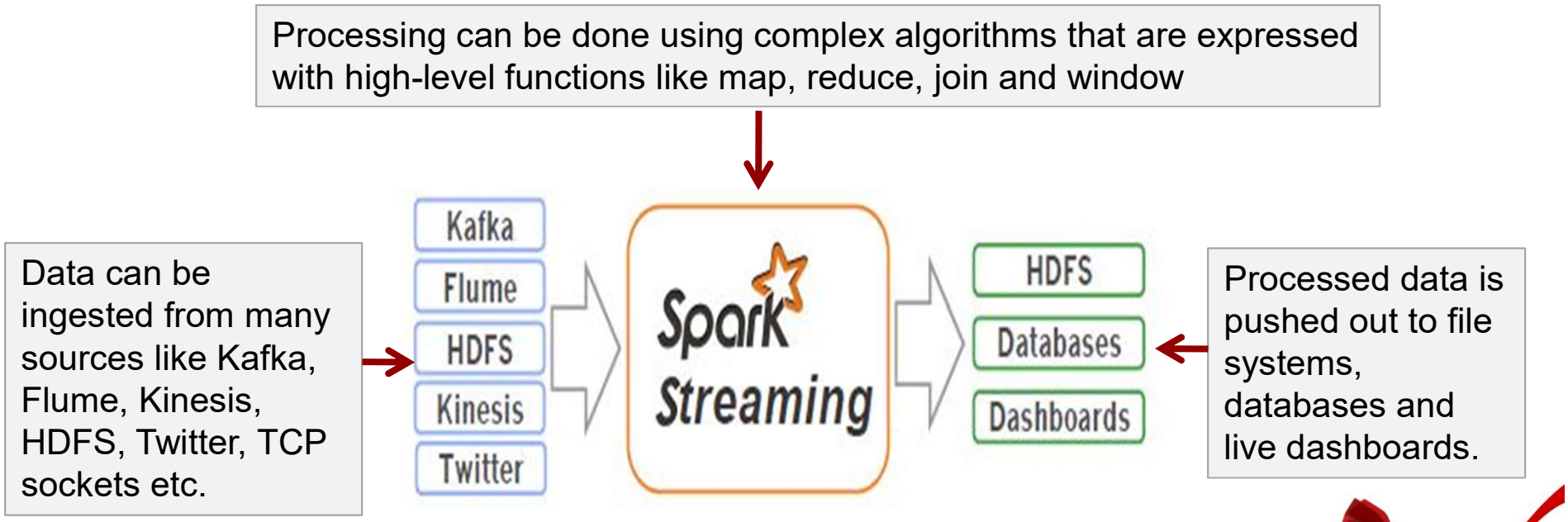

Spark
Streaming

Spark Streaming

What is Spark Streaming?

- Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams.
- Added in 2013, an extension of the core Apache Spark API



What is Spark Streaming?

- Extends spark for doing large scale stream processing
- Scales to 100s of nodes and achieves second scale latencies
- Efficient and fault-tolerant stateful stream processing
- Integrates with Spark's batch and interactive processing
- Provides simple batch-like API for implementing complex algorithms
- Receive data streams from input sources, process them in a cluster, push out to databases/ dashboards

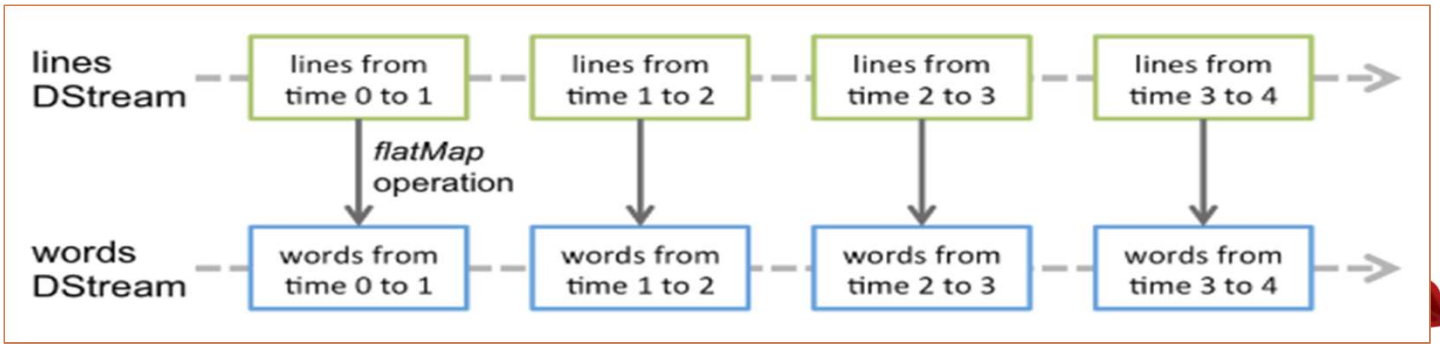
Discretized Stream Processing

- Spark Streaming provides a high-level abstraction called *discretized stream* or *DStream*, which represents a continuous stream of data.
- DStreams can be created either from input data streams from sources such as Kafka, Flume, and Kinesis, or by applying high-level operations on other DStreams.
- Internally, a DStream is represented as a sequence of RDDs
- Spark Streaming receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches.

DStreams

- A DStream is represented by a continuous series of RDDs. Each RDD in a DStream contains data from a certain interval.

- Any operation applied on a DStream translates to operations on the underlying RDDs. For example, we can convert a stream of lines to words, apply **`flatMap`** operation on each RDD in the lines DStream to generate the RDDs of the words DStream.



Streaming WordCount Example

```
# Create a local StreamingContext with two threads and batch interval of 1 sec.
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.print()

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

In the above example we are creating a stateless DStream that listens to the streaming data generated from a socket program that runs on localhost @ port 9999 with a batch intervel of 1 second. We can use NetCat utility to generate data on port 9999

Dependencies for Spark-Streaming

All the dependencies that are required to be included in your SparkStreaming projects can be obtained from Maven Central. Shown below is primary dependency that need to be added.

```
<dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-streaming_2.11</artifactId>
        <version>2.3.1</version>
</dependency>
```

To use streaming data ingestion tools like Kafka, Flume, and Kinesis that are not present in the Spark Streaming core API, you will have to add the corresponding artifact to the dependencies.

Source	Artifact
Kafka	spark-streaming-kafka-0-10_2.11
Flume	spark-streaming-flume_2.11
Amazon Kinesis	spark-streaming-kinesis-asl_2.11

StreamingContext

`StreamingContext` is the main entry point of all Spark Streaming functionality that can be created from a **`SparkConf`** object or from an existing **`SparkContext`** object.

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

sc = SparkContext(master, appName)
ssc = StreamingContext(sc, 1)
```

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 to DStreams.
3. Start receiving data and processing it using **`streamingContext.start()`**
4. Wait for the processing to be stopped using **`streamingContext.awaitTermination()`**
5. The processing can be manually stopped using **`streamingContext.stop()`**

StreamingContext contd..

- 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 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 re-used to create multiple StreamingContexts, as long as the previous StreamingContext is stopped (without stopping the SparkContext) before the next StreamingContext is created.

Streaming Sources

The stream of input data received from streaming sources is represented by **Input DStreams**. Every Input Dstream is associated with a Receiver object which receives the data from a source and stores it in Spark's memory for processing.

Spark Streaming provides two categories of built-in streaming sources:

Basic Sources:

- Directly available in the StreamingContext API.
- Examples: file systems, and socket connections.

Advanced Sources:

- Available through extra utility classes requiring additional dependencies.
- Examples: Kafka, Flume, Kinesis, etc.

File Streams

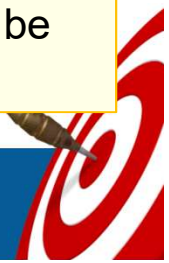
A FileStream can read data from any HDFS API compatible File Systems such as HDFS, S3, NFS etc.

```
sc = SparkContext(master, appName)
ssc = StreamingContext(sc, 1)
lines = ssc.textFileStream(args(0))
```

Use Case:

FileStreams can be used to monitor files appearing in a specified directory path and process them. All files directly under such a path will be processed as they are discovered. A POSIX glob pattern can be supplied, such as "hdfs://namenode:8040/logs/2017/*". Here, the DStream will consist of all files in the directories matching the pattern.

- All files must be in the same data format.
- A file is considered part of a time period based on its modification time.
- Once processed, changes to a file within the current window will not cause the file to be reread. That is: *updates are ignored*.



DStream Transformations

Similar to that of RDDs, transformations allow the data from the input DStream to be modified. DStreams support many of the transformations available on normal Spark RDD's. Some of the common ones are as follows.

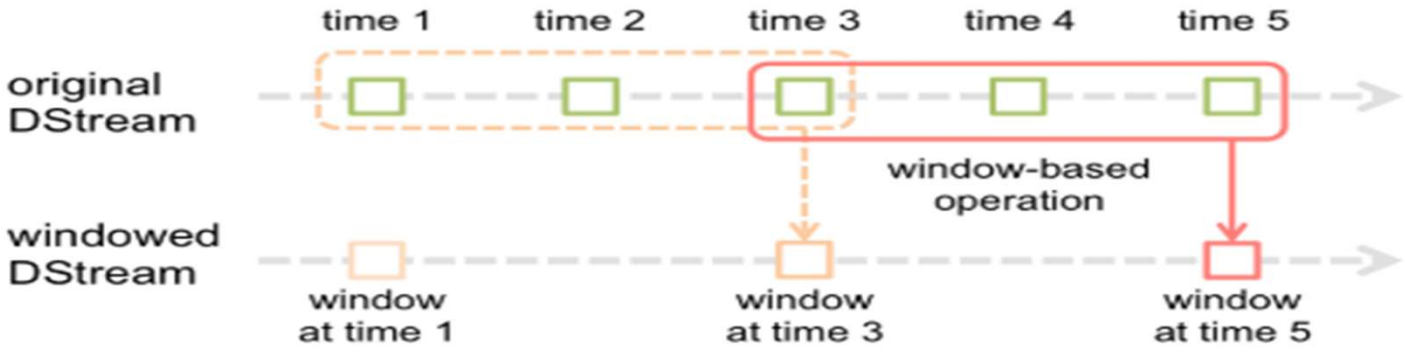
Common DStream Transformations		
map(*func*)	**flatMap**(*func*)	**filter**(*func*)
repartition(*numPart*)	**union**(*otherStream*)	**count**()
reduce(*func*)	**countByValue**()	**reduceByKey**(*func*, [*#Tasks*])
join(*otherStream*, [*#Task*])	**updateStateByKey**(*func*)	**transform**(*func*)

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, Define the state update function

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.

Windowed Transformations

Spark Streaming also provides *windowed computations*, which allow you to apply transformations over a sliding window of data. The following figure illustrates this sliding window.



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 this specific case, 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

Windowed Transformations contd..

Transformation	Meaning
window(*windowLength, slideInterval*)	Return a new DStream which is computed based on windowed batches of the source DStream
countByWindow(*windowLength, slideInterval*)	Return a sliding window count of elements in the stream.
reduceByWindow(*func, windowLength, slideInterval*)	Return a new single-element stream, created by aggregating elements in the stream over a sliding interval using *func*. The function should be associative and commutative so that it can be computed correctly in parallel.
reduceByKeyAndWindow(*func, windowLength, slideInterval,* [*numTasks*])	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.
countByValueAndWindow(*windowLength,slideInterval,* [*numTasks*])	When called on a DStream of (K, V) pairs, returns a new DStream of (K, Long) pairs where the value of each key is its frequency within a sliding window. Like in reduceByKeyAndWindow, the number of reduce tasks is configurable through an optional argument.

Checkpointing

- Needs to be setup in Spark Streaming for fault tolerance.
- Saves data periodically on reliable storage such as HDFS, S3 etc.
- There are two types of data that are checkpointed.
  - **Metadata Checkpointing**: Stores only metadata of streaming computation. It stores: *Configuration* , *DStream operations*, *Incomplete batches*
  - **Data checkpointing** - Saving of the generated RDDs to reliable storage. This is necessary in some *stateful* transformations that combine data across multiple batches. In such transformations, the generated RDDs depend on RDDs of previous batches, which causes the length of the dependency chain to keep increasing with time. To avoid such unbounded increases in recovery time (proportional to dependency chain), intermediate RDDs of stateful transformations are periodically *checkpointed* to reliable storage .
- Metadata check-pointing is primarily needed for recovery from driver failures; Data check-pointing is necessary even for basic functioning if stateful transformations are used.

Stateful Streaming

Stateful Streaming allows the RDD to maintain the state of entire streaming operation by checking the previous state of the RDD in order to update the new state of the RDD.

Spark provides 2 API's to perform stateful streaming, which is **updateStateByKey** and **mapWithState.**

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

countStream = text.flatMap(lambda line: line.split(" "))\
                  .map(lambda word: (word, 1))\
                  .reduceByKey(lambda a, b: a + b)

totalCounts = countStream.updateStateByKey(updateTotalCount)
totalCounts.pprint()

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

THANK YOU