Spark Streaming

DStreams & Structured Streaming

SPARK STREAMING

Agenda

In this module, we are going to look at the following topics:

- ✓ Stream Processing
- ✓ Spark Stream Processing APIs
- ✓ Dstreams API
- ✓ Streaming Context
- ✓ Streaming Sources
- ✓ DStream Transformations

Stream Processing

- Stream processing is the act of continuously incorporating new data to compute a result.
- In stream processing, the input data is unbounded and has no predetermined beginning or end. It simply forms a series of events that arrive at the stream processing system
- Examples:
  - credit card transactions
  - clicks on a website
  - sensor readings from IoT devices etc.

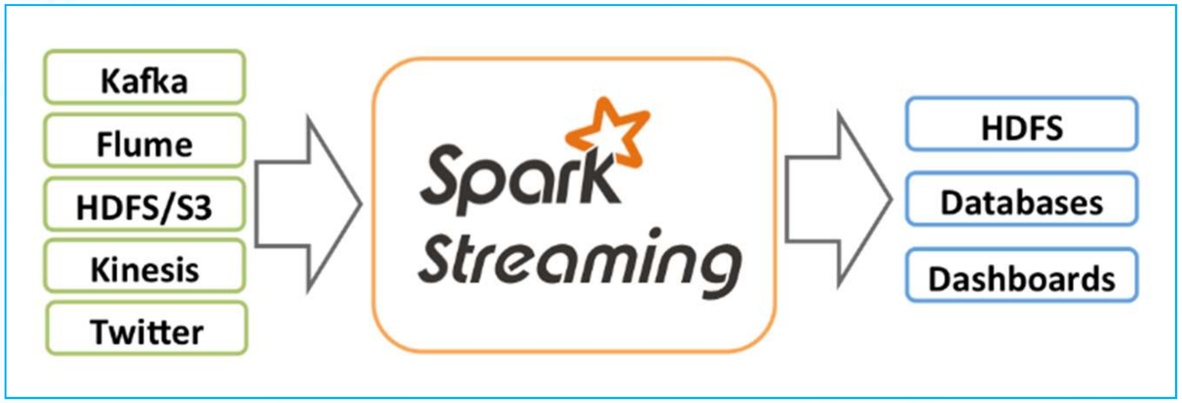
Stream Processing Use Cases

- Notifications and alerting
  - Ex: Driving an alert to an employee at a fulfillment center
- Real-time reporting
  - Ex: Real-time Dashboards about a systems usage patterns
- Incremental ETL
  - Ex: Streaming processing of batch jobs
- Update data to serve in real time
  - Ex: Web analytics products such as Google Analytics
- Real-time decision making
  - Ex: Tracking fraudulent credit card transactions

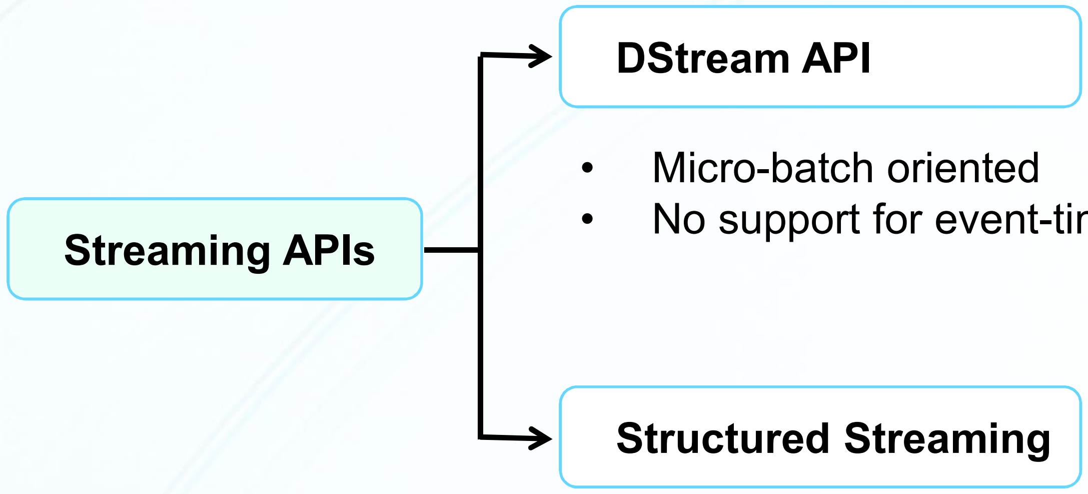
Spark Streaming

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

- Data can be ingested from different streaming sources and can be processed using complex algorithms expressed with high-level functions like `map`, `reduce`, `join` and `window`. Finally, processed data can be pushed out to file systems, databases, and live dashboards.



Spark’s Stream APIs



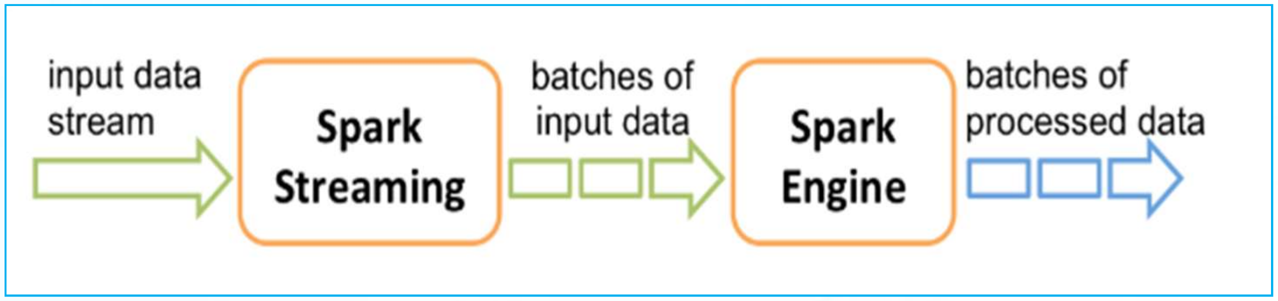
- Micro-batch oriented
- No support for event-time

- Higher level optimizations
- Supports event-time processing
- Support for continuous processing

DStreams API

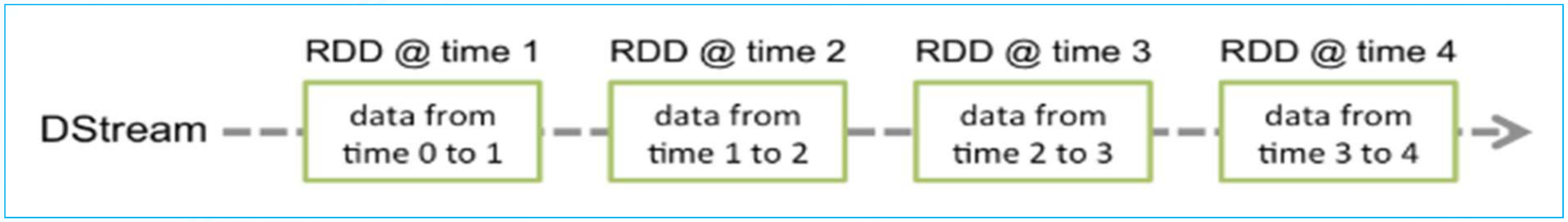
Discretized Stream Processing

- Spark Streaming provides a high-level abstraction called ***discretized stream*** or ***DStream***, which represents a continuous stream of data, represented as a **sequence of RDDs**
- DStreams can be created either from input data stream sources such as Kafka, Flume or by applying high-level operations on other DStreams.
- 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.



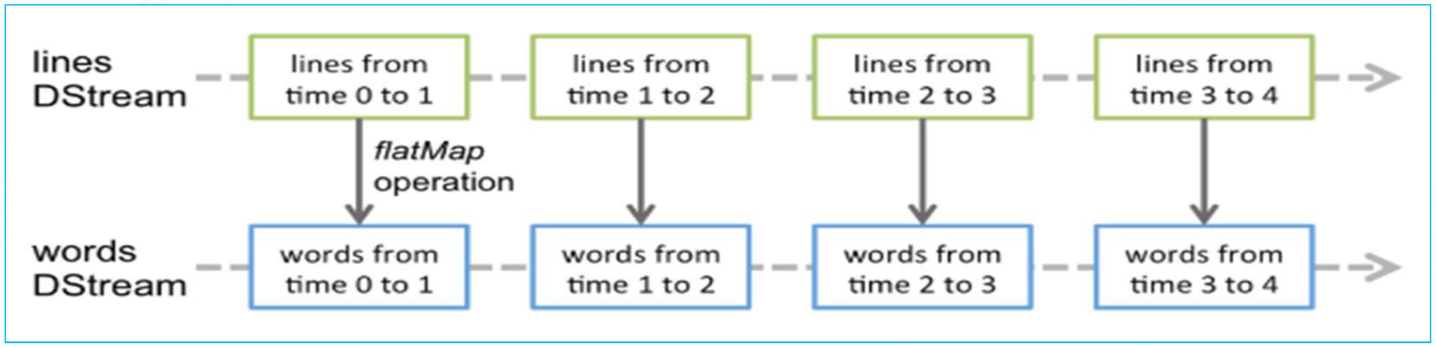
DStreams

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



DStreams

- 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 ‘Word Count’ Example

```
val conf = new SparkConf().setMaster("local[2]“).setAppName(“SWC”)

// Create a Streaming Context object from SparkConf
val ssc = new StreamingContext(conf, Seconds(1))

// Create a Socket Dstream that reads streaming data from localhost:9999
val lines = ssc.socketTextStream(localhost, 9999)

// split lines into words, count them and print them
val words = lines.flatMap(_.split(" "))
val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
wordCounts.print()

// start processing the stream and keep running until terminated.
ssc.start()
ssc.awaitTermination()
```

Here 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 interval of 1 second.

Dependencies for Spark Streaming

- All the dependencies that are required to be included in your Spark Streaming 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>
```

Dependencies for Spark Streaming

- 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.

```
val conf = new SparkConf().setAppName(appName).setMaster(master)
val ssc = new StreamingContext(conf, Seconds(1))

val sc = ... // existing SparkContext
val ssc = new StreamingContext(sc, Seconds(1))
```

StreamingContext

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

- Define the input sources by creating input DStreams.
- Define the streaming computations to DStreams.
- Start receiving data and processing it using `streamingContext.start()`
- Wait for the processing to be stopped using `streamingContext.awaitTermination()`
- The processing can be manually stopped using `streamingContext.stop()`

StreamingContext

- 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 a 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 & 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.

```
val ssc = new StreamingContext(sparkConf, Seconds(2))
val lines = ssc.textFileStream(args(0))

// generic definition
ssc.fileStream[KeyClass, ValueClass, InputFormatClass](dataDirectory)
val fs = ssc.fileStream[IntWritable, Text, TextInputFormat]("/path/file")
```

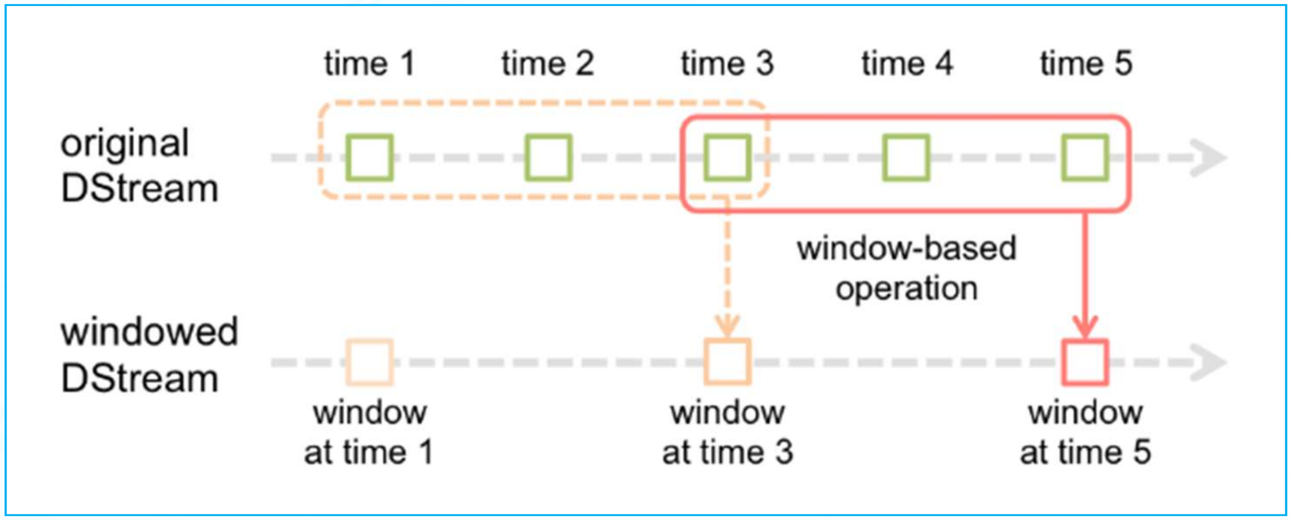
DStream Transformations

- 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*)

Windowed Transformations

- Spark Streaming also provides *windowed computations*, which allow you to apply transformations over a sliding window of data.



Windowed Transformations

- 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.

- 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

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.

THANK YOU