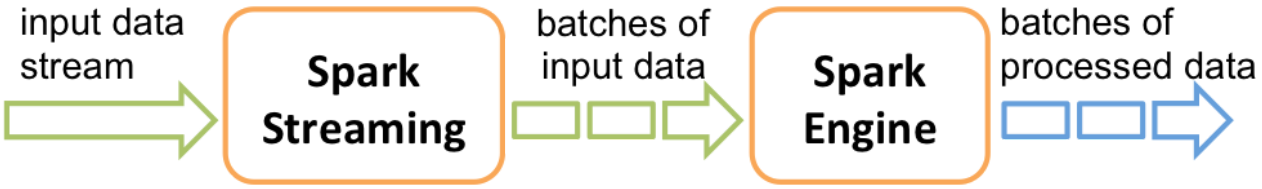
02 Spark Streaming -KirkYagami 👨‍💻🕵️

Spark Streaming

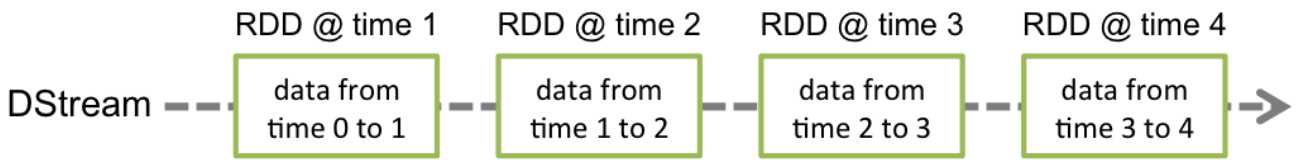
Apache Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. Data can be ingested from many sources like Kafka, Flume, Kinesis, or TCP sockets, and can be processed using complex algorithms expressed with high-level functions like map, reduce, join, and window. Finally, the processed data can be pushed out to filesystems, databases, or live dashboards.



1. Core Concepts of Spark Streaming

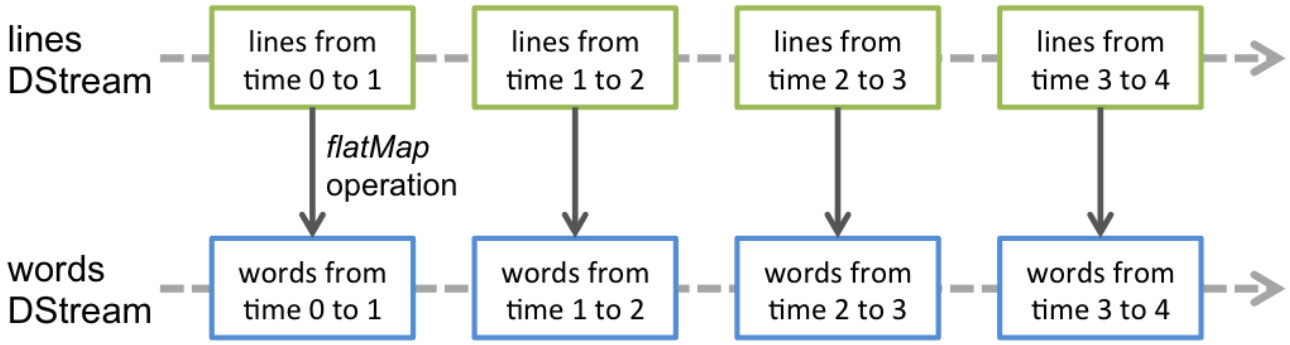
1.1 Discretized Streams (DStreams)

- **Definition**: The basic abstraction provided by Spark Streaming is a 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, or HDFS, or by transforming existing DStreams.
- **Internals**: Under the hood, DStream is represented as a sequence of RDDs (Resilient Distributed Datasets). Each RDD in a DStream contains data from a specific interval, allowing Spark to process the stream data in micro-batches.



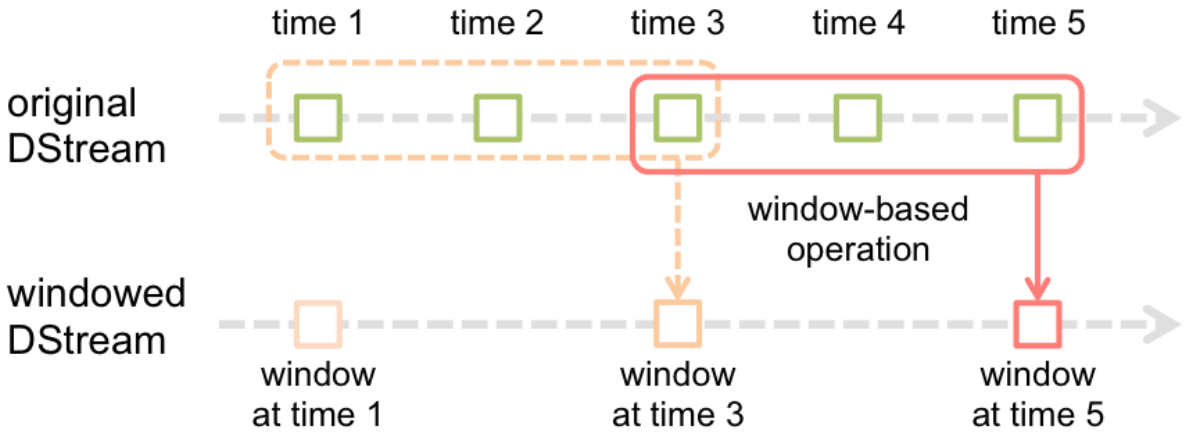
1.2 Transformations and Actions

- **Transformations**: Just like RDDs, DStreams support transformations. These transformations can be stateless (operating on the data within a single batch) or stateful (keeping some state across batches).
  - **Examples**: `map`, `flatMap`, `filter`, `reduceByKey`, `join`, `window`, `updateStateByKey`, etc.
- **Actions**: Actions in DStreams trigger the actual execution of the computation defined by transformations. Actions include operations like `print`, `saveAsTextFiles`, `saveAsObjectFiles`, `saveAsHadoopFiles`, etc.



1.3 Window Operations

- **Definition**: Window operations allow you to perform transformations over a sliding window of data. This means you can analyze data not just from the current batch but from a range of batches.
- **Operations**:
  - `window`: Applies a transformation to a sliding window.
  - `**reduceByWindow**`: Reduces over a sliding window.
  - `countByWindow`: Counts elements over a sliding window.
  - `reduceByKeyAndWindow`: Applies `reduceByKey` over a sliding window.
  - `countByValueAndWindow`: Counts values over a sliding window.



As shown in the figure, 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 (1 in the figure).

1.4 Checkpointing

- **Definition**: Checkpointing is the process of saving the state of a streaming application to a fault-tolerant storage like HDFS. It is essential for stateful transformations to recover from failures.
- **Types**:
  - **Metadata Checkpointing**: Saving the metadata information about the streaming computation (like batch offsets) to ensure that when a node crashes, the computation can restart from the point of failure.
  - **Data Checkpointing**: Saving the generated RDDs to reliable storage so that they can be used to recompute the state in case of a failure.

1.5 Fault Tolerance

- **Definition**: Spark Streaming provides fault tolerance through a combination of data replication, lineage, and checkpointing.
- **Mechanism**:
  - Data received from sources is replicated across nodes.
  - Lineage information (how an RDD was generated from other RDDs) allows re-computation of lost RDDs.
  - Checkpointing, as mentioned above, helps in recovering from failures.

2. Working with Spark Streaming

Below is a step-by-step guide to working with Spark Streaming:

2.1 Setup and Initialization

First, set up a Spark Streaming context. In modern versions, you'll use `SparkSession` along with `SparkContext`.

```
from pyspark.sql import SparkSession
from pyspark.streaming import StreamingContext
# Initialize Spark Session and Context

spark = SparkSession.builder \
    .appName("SparkStreamingExample") \
    .getOrCreate()

sc = spark.sparkContext
ssc = StreamingContext(sc, 1)  # Batch interval of 1 second
```

2.2 Creating DStreams

You can create DStreams from various sources. For example, a socket text stream:

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

2.3 Applying Transformations

Apply transformations to process the DStream. For instance, counting words in each batch:

```
words = lines.flatMap(lambda line: line.split(" "))
wordCounts = words.map(lambda word: (word, 1)).reduceByKey(lambda a, b: a + b)
```

2.4 Output Operations

Finally, perform an action to trigger the execution of the stream processing:

```
wordCounts.pprint()  # Print the first 10 elements of each RDD
```

2.5 Starting the Stream

Start the computation and wait for it to finish:

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

3. Implementation Example: Real-time Log Monitoring

This example demonstrates a real-time log monitoring solution using Spark Streaming. The objective is to monitor a stream of log data and detect error messages.

3.1 Problem Statement

You have a stream of logs coming from multiple servers. You want to monitor these logs and alert when error messages appear, indicating potential issues.

3.2 Solution Design

1. **Data Source**: A socket source will be used to simulate log messages being pushed to the system.
2. **Processing**: We will filter the stream for log lines containing "ERROR" and count them in a sliding window.
3. **Output**: The count of error messages will be printed to the console, but this could be extended to send alerts via email or SMS.

3.3 Implementation

```
from pyspark.sql import SparkSession
from pyspark.streaming import StreamingContext
# Initialize Spark Session and Streaming Context
spark = SparkSession.builder \
    .appName("LogMonitoring") \
    .getOrCreate()
sc = spark.sparkContext
ssc = StreamingContext(sc, 10)  # 10-second batch interval
# Checkpointing for fault tolerance
ssc.checkpoint("hdfs://checkpoint-directory")
# Simulate stream of log lines from a socket
lines = ssc.socketTextStream("localhost", 9999)
# Filter lines containing "ERROR"
errorLines = lines.filter(lambda line: "ERROR" in line)
# Count errors over a sliding window of 30 seconds, sliding every 10 seconds
errorCounts = errorLines.countByWindow(30, 10)
# Print the counts
errorCounts.pprint()
# Start the streaming computation
ssc.start()
ssc.awaitTermination()
```

3.4 Explanation

- **Checkpointing**: Used to save the state of the streaming context to allow recovery from failures.
- **Window Operations**: The `countByWindow` function is applied to calculate the number of errors over a sliding window of 30 seconds, sliding every 10 seconds.
- **Fault Tolerance**: By enabling checkpointing, the application can recover and resume processing in case of a failure.

3.5 Scalability and Production Deployment

- **Cluster Setup**: In production, Spark Streaming jobs would run on a distributed cluster, taking advantage of Spark's fault tolerance and scalability.
- **Integration with Message Brokers**: Instead of using a socket, real-world applications would typically use Kafka, Flume, or another message broker to ingest data.
- **Monitoring**: Tools like Spark UI, Ganglia, and custom monitoring scripts can be used to monitor the performance of the streaming job.

- **Alerting**: The counts can be fed into an alerting system, such as PagerDuty, to trigger notifications when error thresholds are exceeded.

4. Advanced Topics

4.1 Structured Streaming

Structured Streaming is the successor to Spark Streaming, introduced in Spark 2.0. It addresses some limitations of the original Spark Streaming, such as late data handling, and it integrates deeply with Spark SQL and DataFrames.

4.2 Handling Late Data

In real-time streams, data might arrive late. Spark Streaming provides mechanisms to handle such data, using watermarking in structured streaming.

4.3 Stateful Operations

Stateful operations allow you to keep track of state across batches. For example, tracking running counts of elements or maintaining session information for users.

4.4 Fault Tolerance and Exactly-Once Semantics

Structured Streaming supports exactly-once semantics, ensuring that each record is processed exactly once, even in the presence of failures.

Conclusion

Apache Spark Streaming is a powerful framework for real-time data processing, enabling developers to build scalable and fault-tolerant streaming applications with ease. By understanding the core concepts like DStreams, transformations, actions, windowing, and fault tolerance, one can implement a wide range of streaming applications, from real-time analytics to complex event processing.

This is just a starting point. The real power of Spark Streaming shines when combined with other tools like Kafka, Cassandra, and HDFS in a well-architected data pipeline.

Example which we covered in the class…

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

# Create a local StreamingContext with four working thread and batch interval of 3 second
sc = SparkContext("local[4]", "NetworkWordCount")
ssc = StreamingContext(sc, 1)
# Create a DStream that will connect to hostname:port, like localhost:9999
lines = ssc.socketTextStream("localhost", 9999)

# 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))
wordCounts = pairs.reduceByKey(lambda x, y: x + y)

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

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

Run the below command in PowerShell or WSL Ubuntu.

```
nc -lk 9999
```