Université Constantine 2
جامعة قسنطينة 2

Big Data Analytics and Processing

– Cours 6 –

Chapitre 3 : Streaming Processing (Spark Streaming)

Dr. MENNOUR R.
Faculté des nouvelles technologies
rostom.mennour@univ-constantine2.dz

Etudiants concernés

Faculté/Institut	Département	Niveau	Spécialité
Nouvelles technologies	IFA	Master 2	Science de données et Intelligence Artificielle

Section 1 :
Streaming Processing Overview

Streaming processing overview

Streaming processing

Stream processing is a computational paradigm that continuously ingests, processes, and analyzes data in real-time as it flows from various sources. Unlike traditional batch processing, which processes data in fixed intervals, stream processing handles data as it arrives, enabling low-latency insights and immediate actions. This approach is particularly useful for applications requiring timely responses, such as fraud detection, real-time analytics, and IoT monitoring.

Streaming processing overview

Applications

Stream processing is widely used across industries to enable real-time decision-making and improve operational efficiency. Below are some key applications:

1. Financial Services

    Fraud Detection: Analyzing transaction streams in real time to identify suspicious activities.

    Algorithmic Trading: Making split-second trading decisions based on market data streams.

2. E-commerce

    Personalized Recommendations: Suggesting products to users based on their real-time browsing or purchase behavior.

    Dynamic Pricing: Adjusting prices instantly based on demand and supply trends.

3. Internet of Things (IoT)

    Smart Cities: Monitoring traffic, energy usage, or environmental conditions using sensor data.

    Industrial IoT: Analyzing machine performance data for predictive maintenance.

4. Healthcare

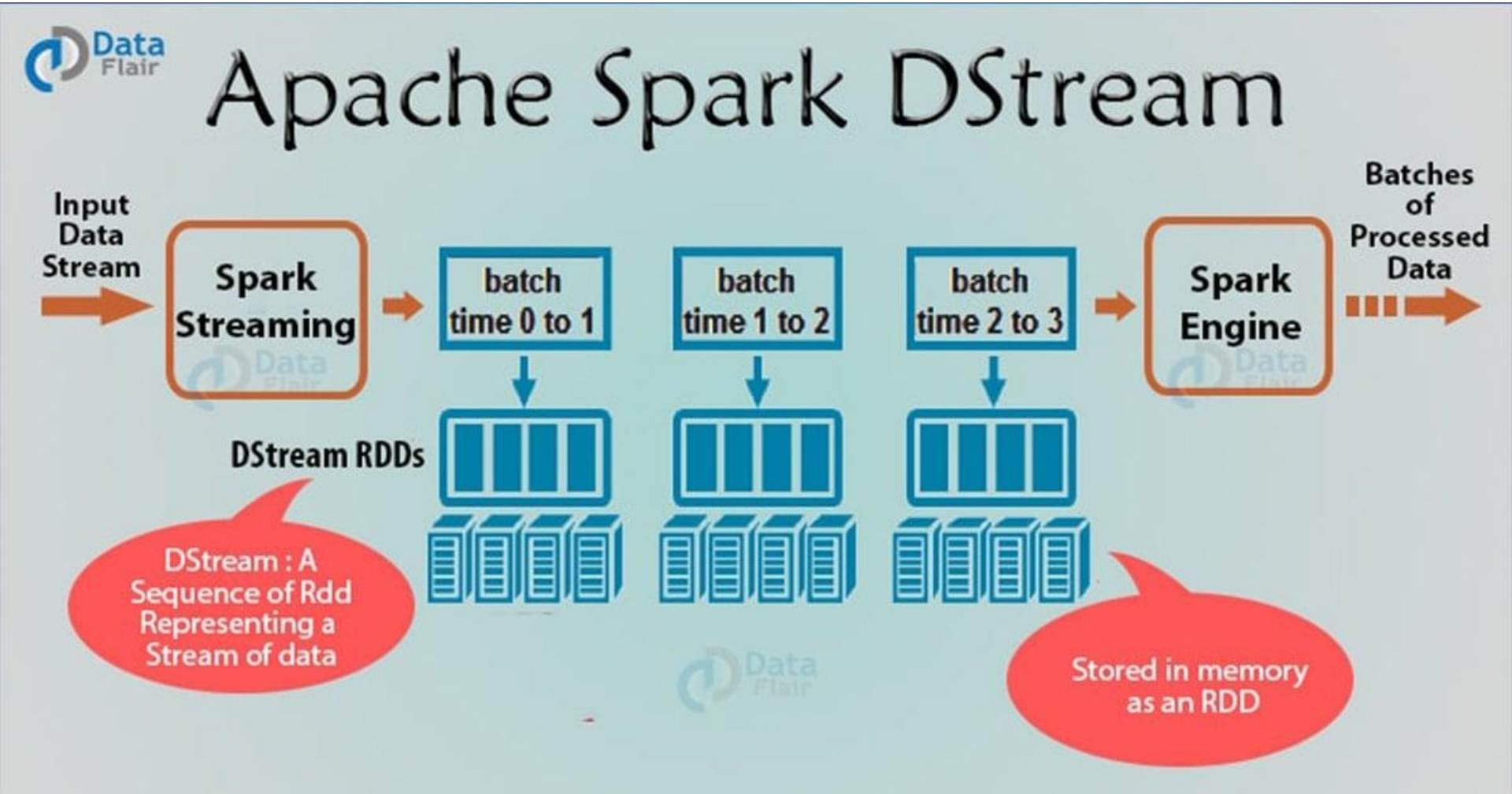
    Patient Monitoring: Real-time analysis of vital signs to detect emergencies.

    Telemedicine: Streaming patient data for remote diagnosis and treatment.

Stream processing can be categorized into two main types based on how data is handled:

1. **Stateless Stream Processing**: Each event is processed independently without relying on previous events.
   - Example: Filtering events based on a condition or transforming individual records.
   - Use Case: Real-time filtering of social media feeds.
2. **Stateful Stream Processing**: Maintains state information across multiple events to perform more complex operations.
   - Example: Aggregations (e.g., counting occurrences), joins, or windowed computations over time.
   - Use Case: Monitoring stock prices over a sliding time window to detect trends

Apache Spark Streaming is a powerful framework for real-time stream processing, designed to handle high-throughput and low-latency workloads. It extends the core Spark API to process live data streams, making it suitable for applications requiring real-time analytics, event detection, and continuous data processing.


Data Flair
Apache Spark DStream
Input Data Stream
Spark Streaming
batch time 0 to 1
batch time 1 to 2
batch time 2 to 3
Spark Engine
Batches of Processed Data
DStream RDDs
DStream : A Sequence of Rdd Representing a Stream of data
Stored in memory as an RDD

1. Unified Batch and Streaming Model: Spark Streaming operates on a unified engine that supports both batch and streaming workloads. This enables developers to use a single framework for diverse data processing needs.
2. Micro-Batch Processing: Spark Streaming divides incoming data streams into small, time-based batches (e.g., every second or minute). These micro-batches are processed using Spark's distributed computing engine.
3. DStream Abstraction: The core abstraction in Spark Streaming is the *Discretized Stream (DStream)*, which represents a continuous stream of data divided into RDDs (Resilient Distributed Datasets) for batch processing.
4. Fault Tolerance: Data is stored in a fault-tolerant manner, ensuring recovery from node failures without data loss. This is achieved through Spark's lineage-based computation model.

1. Integration with Ecosystem Tools: Spark Streaming seamlessly integrates with popular tools like Apache Kafka, Flume, Amazon Kinesis, and HDFS for data ingestion and output to databases or dashboards.
2. Scalability: It scales horizontally by distributing workload across multiple nodes in a cluster, making it capable of handling large-scale streaming applications.
3. Advanced Processing Capabilities: Developers can combine streaming data with static datasets, perform interactive queries using Spark SQL, or apply machine learning models using MLlib.

Operations on DStreams allow developers to transform and process streaming data in real-time. There are two main categories of DStream operations: transformations and output operations.

1- Transformations create new DStreams by applying computations to the data in the input DStream. These operations can be stateless (independent of previous batches) or stateful (dependent on previous batches).

2- Output operations allow saving or displaying the results of transformations. These are the final steps in a Spark Streaming application.

These transformations operate independently on each batch of data.

- map(func): Applies a function to each element in the DStream and returns a new DStream.
- flatMap(func): Similar to map, but allows returning multiple elements for each input element.
- filter(func): Filters elements based on a condition.
- reduceByKey(func): Aggregates values for each key using a specified function.
- countByValue(): Counts the occurrences of each unique value in the DStream.
- transform(func): Allows applying arbitrary RDD-to-RDD functions to each batch of data.

Stateful transformations in Spark Streaming allow the processing of data across multiple batches by maintaining and updating state information over time. These operations are particularly useful for applications that require tracking or aggregating data over a period, such as session tracking, running counts, or detecting trends.

Key Stateful Transformations

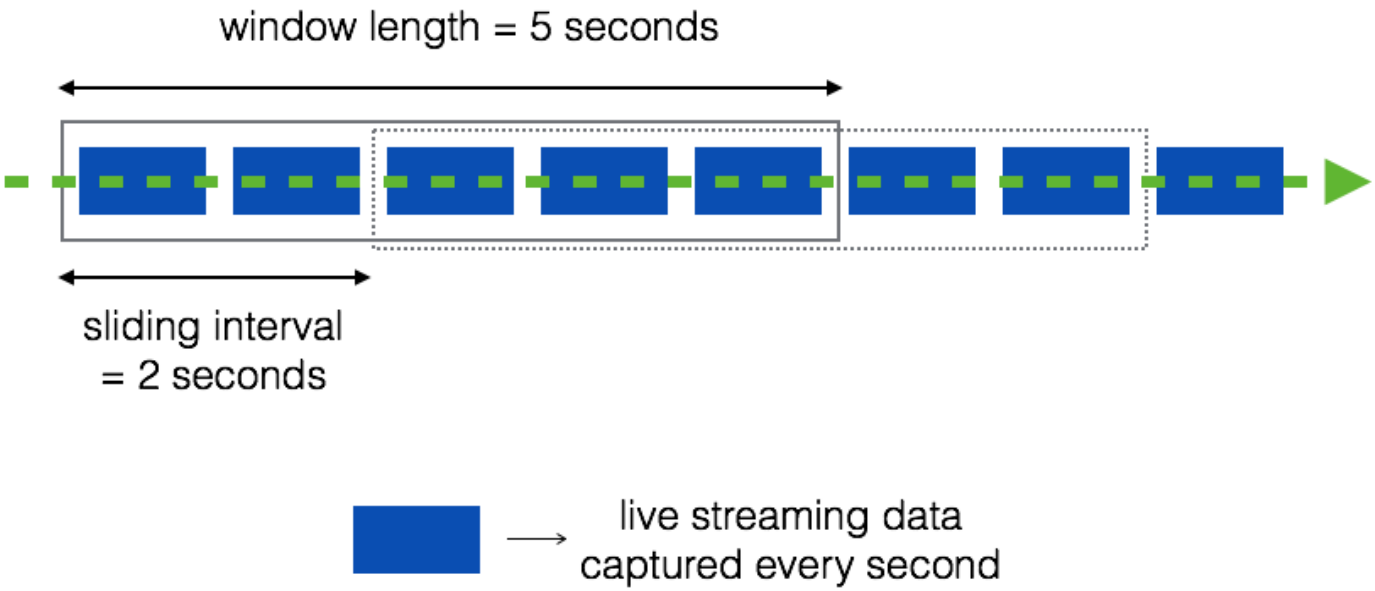
- **updateStateByKey**
- **Windowed Operations**

The updateStateByKey transformation enables maintaining a running state for each key by updating it with new data from each batch. This is achieved by providing a user-defined function that specifies how to update the state using the previous state and new values.

- **How It Works**
  - The state is stored as a key-value pair.
  - For each key, the function takes:
    - A sequence of new values from the current batch.
    - The previous state (if any).
  - It returns the updated state for that key.

```
# Create a DStream from a socket source
lines = ssc.socketTextStream("localhost", 9999)
 # Split lines into words and map them to (word, 1)
words = lines.flatMap(lambda line: line.split(" "))
word_counts = words.map(lambda word: (word, 1))
# Define the update function
def update_function(new_values, running_count):
        return sum(new_values) + (running_count or 0)
# Apply updateStateByKey to maintain running word counts
running_word_counts = word_counts.updateStateByKey(update_function)
# Print the results
running_word_counts.pprint()
# Start the streaming context
ssc.start()
ssc.awaitTermination()
```

Windowed operations compute results over a sliding window of time rather than just processing the latest batch. This allows analyzing trends and patterns over specific time intervals.



- **window(windowDuration, slideDuration): Creates a new DStream by collecting data over a specified window duration.**
- **countByWindow(windowDuration, slideDuration): Counts all elements in the stream within the window.**
- **reduceByWindow(func, windowDuration, slideDuration): Reduces all elements in the stream within the window using a specified function.**

```
# Count words over a sliding window of 10 seconds with updates every 5 seconds
windowed_word_counts = words.map(lambda word: (word, 1)) \
        .reduceByKeyAndWindow(
                lambda x, y: x + y,
                lambda x, y: x - y,
                windowDuration=10,
                slideDuration=5 )
 windowed_word_counts.pprint()
```