# CS x55: Distributed Systems [Spark Streaming]

**Drinking from a fire hose**
A packet in isolation seems fine
Why then, do streams, strain systems design?

If processing lags the rate of arrival?
Imperil, you will, your process' survival

Shrideep Pallickara
Computer Science
Colorado State University

Computer Science Department
Colorado State University

1

## Frequently asked questions from the previous class survey

- Does Spark try to satisfy wide dependencies first?
- In narrow & wide transformations does the data shuffling happen only when action is called on the transformation?

Colorado State University Professor: Shrideep Pallickara Computer Science Department Spark Streaming L33.2

2

## Topics covered in this lecture

- Alleviating inefficiencies with shuffles
- Spark Streaming
  - Architecture and Abstractions
  - Execution
  - Stateful and stateless transformations
  - Windowed operations
  - Performance considerations
  - Example



## Two primary techniques to avoid performance problems associated with shuffles

- Shuffle Less
- Shuffle Better

## Shuffle Less

- Preserve partitioning across narrow transformations to avoid reshuffling data
- Use the same partitioner on a sequence of wide transformations. This can be particularly useful:
  - To avoid shuffles during joins and …
  - To reduce the number of shuffles required to compute a sequence of wide transformations



## Shuffle Better [1/2]

- Sometimes, computation cannot be completed without a shuffle
- However, not all wide transformations and not all shuffles are equally expensive or prone to failure

## Shuffle Better [2/2]

- By using wide transformations such as `reduceByKey` and `aggregateByKey` that can preform map-side reductions and that do not require loading all the records for one key into memory?
  - You can prevent memory errors on the executors and
  - Speed up wide transformations, particularly for aggregation operations
- Lastly, shuffling data in which **records are distributed evenly throughout the keys**, and which contain a **high number of distinct keys**?
  - Prevents out-of-memory errors on the executors and "straggler tasks"



7

# PARTITIONERS



8

## Partitioners

- The partitioner defines **how records will be distributed** and thus which records will be completed by each task
- Practically, a partitioner is actually an interface with two methods
  - `numPartitions` that defines the number of partitions in the RDD after partitioning
  - `getPartition` that defines a mapping from a key to the integer index of the partition where records with that key should be sent



## There are two implementations for the partitioner object provided by Spark

- HashPartitioner
  - Determines the index of the child partition based on the hash value of the key
- RangePartitioner
  - Assigns records whose keys are in the same range to a given partition
  - Required for **sorting** since it ensures that by sorting records within a given partition, the entire RDD will be sorted
- It is possible to define a custom partitioner

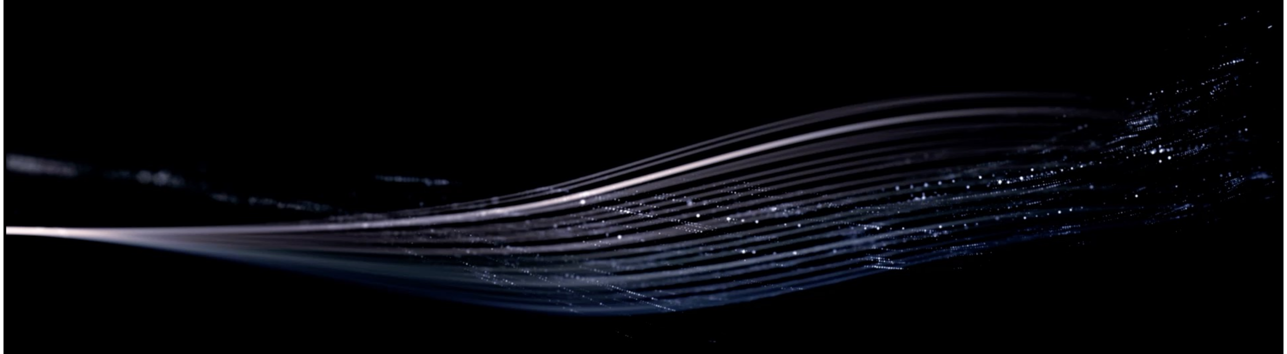
## Partitioners and transformations

- Unless a transformation is known to only change the value part of the key/value pair in Spark
  - The resulting RDD will **not have a known** partitioner
    - Even if the partitioning has not changed



## Using narrow transformations that preserve partitioning

- Some narrow transformations, such as `mapValues`, **preserve the partitioning** of an RDD if it exists
- Common transformations like `map` and `flatMap` can change the key
  - So even if your function does not change the key, the resulting RDD will not have a known partitioner.
  - Instead, if you don't want to modify the keys, call the `mapValues` function (defined only on pair RDDs)
    - It keeps the keys, and therefore the partitioner, exactly the same.
    - The `mapPartitions` function will also preserve the partition if the `preservesPartitioning` flag is set to true.



## SPARK STREAMING

13

## Related Work

Thilina Buddhika*, Sangmi Lee Pallickara, and Shrideep Pallickara. Pebbles: Leveraging Sketches for Processing Voluminous, High Velocity Data Streams. IEEE Transactions on Parallel and Distributed Systems. Vol 32 (8) pp 2005 - 2020. 2021.

Thilina Buddhika*, Ryan Stern*, Kira Lindburg*, Kathleen Ericson*, and Shrideep Pallickara. Online Scheduling and Interference Alleviation for Low-latency, High-throughput Processing of Data Streams. IEEE Transactions on Parallel and Distributed Systems. Vol. 28(12) pp 3553-3569. 2017.

Thilina Buddhika* and Shrideep Pallickara. Neptune: Real Time Stream Processing for Internet of Things and Sensing Environments. Proceedings of the 30th IEEE International Parallel & Distributed Processing Symposium. pp 1143-1152. Chicago, USA. 2016.



14

## Spark Streaming

- Act on data **as soon as it arrives**
  - Track statistics of page views in real time, detect anomalies, etc.
- Spark streaming
  - Spark's module for dealing with streaming data
  - Uses an API very similar to what we have seen with batch jobs (centered around RDDs)
- Available in Java, Scala, and Python





## Spark Streaming: Core concepts

- Provides an abstraction called **DStreams** (discretized streams)
- A DStream is a **sequence of data** arriving over time
- Internally, a DStream is represented as a **sequence of RDDs** arriving at each time step

## DStreams

- DStreams can be created from various input sources
  - Flume, Kafka, or HDFS
- Once built, DStreams offer two types of operations:
  - **Transformations**: Yields a new DStream
  - **Output operations**: Writes data to an external system
- Provides many of the same operations available on RDDs
  - PLUS new operations related to time (e.g., sliding windows)



## Simple Streaming Example [1/2]

- Start by creating a `StreamingContext`
  - Main entry point for streaming functionality
  - Specify batch interval, specifying **how often** to process new data
- We will use `socketTextStream()` to create a DStream based on text data received over a port
- Transform DStream with filter to get lines that contain "error"

## Simple Streaming Example [2/2]

```
JavaStreamingContext jssc =
      new JavaStreamingContext(conf, Durations.seconds(1));

JavaDStream<String> lines =
      jssc.socketTextStream("localhost", 7777);

JavaDStream<String> errorLines =
      lines.filter(new Function<String, Boolean> () {
          public Boolean call(String line) {
             return line.contains("error");
          }
      };
```



## Previous snippet only sets up the computation

- To start receiving the data?
  - Explicitly call `start()` on `StreamContext`
- SparkStreaming will start to schedule Spark jobs on the underlying SparkContext
  - Occurs in a **separate thread**
  - To keep application from terminating?
    - Also call `awaitTermination()`

```
jssc.start();
jssc.awaitTermination()
```

## ARCHITECTURE & ABSTRACTION

COMPUTER SCIENCE DEPARTMENT

COLORADO STATE UNIVERSITY

21

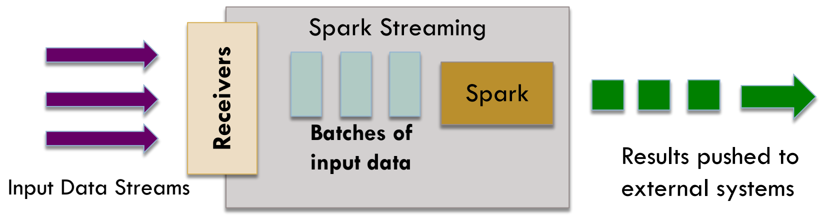
## Spark Streaming Architecture

- Spark Streaming uses a **micro-batch** architecture
  - Streaming computation is treated as continuous series of batch computations on small **batches** of data
- Receives data from various input sources and groups into small batches
- New batches are **created at regular intervals**
  - At the start of each time interval, a new **batch** is created
    - Any data arriving in that interval is added to the batch
    - Size of batch is controlled by the *batch interval*

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT SPARK STREAMING L33.22

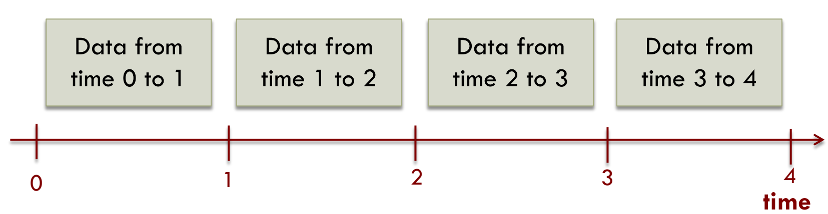
22

## High-level architecture of Spark Streaming

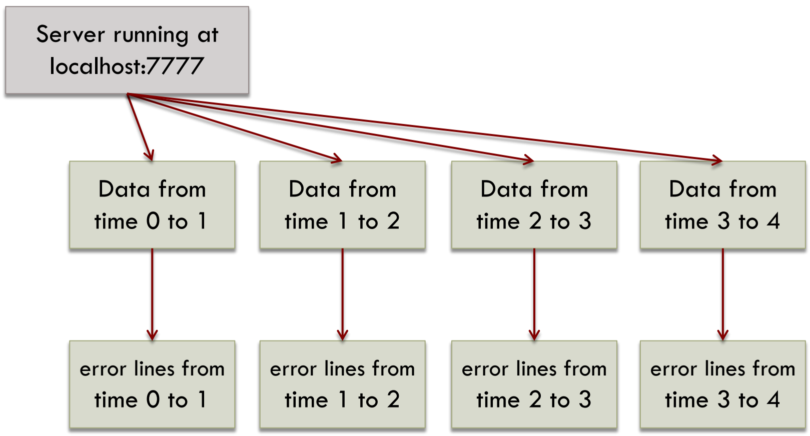





## DStream is a sequence of RDDs, where each RDD has one slice of data in stream



## DStreams and the transformations in our example






## DStreams support output operations, such as `print()`

- Output operations are similar to RDD actions in that they write data to an external system
- But in Spark Streaming they *run periodically* on each time step, producing **output in batches**

## Spark Streaming: Execution

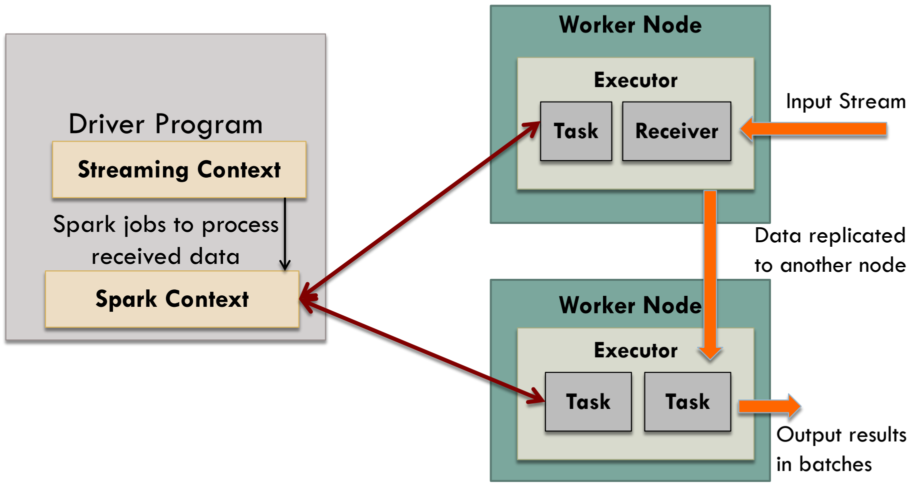
- For each input source, Spark Streaming launches receivers
  - Tasks running within the application's executors that collect data from source and save as RDDs
  - Receives input data and replicates it (by default) to another executor for fault tolerance
  - Data is **stored in memory of the executors** in the same way that RDDs are cached



## Spark Streaming: Execution

- StreamingContext in the driver program then periodically runs Spark jobs to:
  - Process this data and …
  - Combine it with RDDs from previous time steps

## Spark Streaming: Execution






29

## Spark Streaming: Fault Tolerance [1/2]

- Spark Streaming offers the **same fault-tolerance** properties for DStreams as Spark has for RDDs
  - As long as a copy of the input data is still available, it can recompute any state derived from it using the lineage of the RDDs
    - By rerunning the operations used to process it



30

## Spark Streaming: Fault Tolerance [2/2]

- By default, data is replicated across two nodes
  - Can tolerate single worker failures
- Using lineage graphs to recompute any derived state? Impractical
- Spark Streaming relies on **checkpointing**
  - Saves state *periodically*
  - Checkpoint every 5-10 batches of data
  - When recovering, only go back to the last checkpoint



## Spark Streaming: Transformations

- **Stateless** transformations
  - Each batch does not depend on data of its previous batches
- **Stateful** transformations
  - Use data or intermediate results from *previous batches* to compute results of the current batch

## STATELESS TRANSFORMATIONS



## Stateless transformations

- Stateless transformations are simple RDD transformations being applied on every batch — that is, every RDD in a DStream
- Many of the RDD transformations that we have looked at are also available on DStreams

## Examples of stateless transformations [1/6]

- `map()`
- Apply a function to each element in the DStream and return a DStream of the result
- `ds.map (x => x + 1)`



## Examples of stateless transformations [2/6]

- `flatMap()`
- Apply a function to each element in the DStream and return a DStream of the contents of the iterators returned
- `ds.flatMap (x => x.split(" ") )`

## Examples of stateless transformations [3/6]

- `filter()`
- Return a DStream consisting of only elements that pass the condition passed to filter
- `ds.filter (x => x != 1 )`



## Examples of stateless transformations [4/6]

- `repartition()`
- Change the number of partitions of the DStream
  - Distributes the received batches across the specified number of machines in the cluster before processing
    - The physical manifestation of the DStream is different in this case
- `ds.repartition(10)`

## Examples of stateless transformations [5/6]

- `reduceByKey()`
- Combine values with the same key in each batch
- `ds.reduceByKey(   (x,  y)  = > x + y)    )`



## Examples of stateless transformations [6/6]

- `groupByKey()`
- Group values with the same key in each batch
- `ds.groupByKey( )`

## A note about stateless operations

- Although it may seem that they are being applied over the whole stream …
  - Each DStream has multiple RDDs (batches)
  - Stateless transformation applies ***separately*** to each RDD
  - E.g., `reduceByKey()` will reduce data for each timestep, but *not across* timesteps

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT SPARK STREAMING L33.41

41

## STATEFUL TRANSFORMATIONS

COMPUTER SCIENCE DEPARTMENT COLORADO STATE UNIVERSITY

42

## Stateful transformations

- Operations on DStreams that track data across time
  - Data from previous batches used to generate results for a new batch
- Two types of windowed operations
  - Act over **sliding window** of time periods
  - `updateStateBykey()` track state across events for each key



## Stateful transformations and fault tolerance

- Requires checkpointing to be enabled in `StreamingContext` for fault tolerance

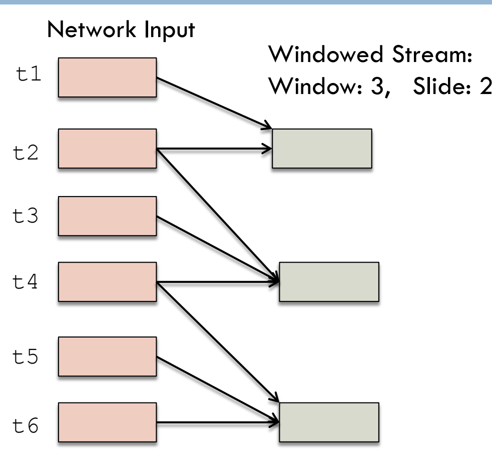
```
ssc.checkpoint("hdfs:// …");
```

## Windowed Transformations

- Compute results across a longer time period than the batch interval
- Two parameters: window and sliding durations
  - Both must be a *multiple* of the batch interval
- Window duration controls **how many** previous batches of data are considered
  - `window Duration/batchInterval`
  - If the batch interval is 10 seconds and the sliding window is 30 seconds … last 3 batches



## A windowed stream: Window duration (3) & slide duration (2)




Every 2 time steps, we compute a result over the previous 3 time steps

## Simplest window operation on a DStream

- `window()`
- Returns new DStream with data from the requested window
- Each RDD in the DStream resulting from `window()`, will contain data from multiple batches

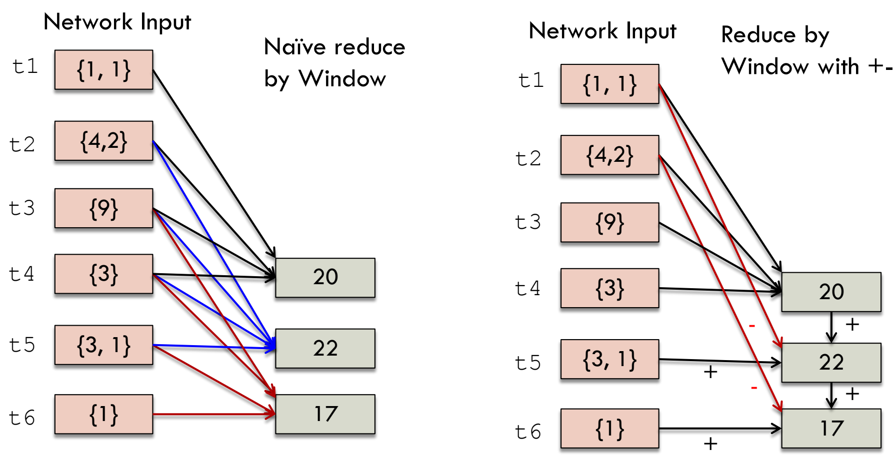




## Other operations on top of `window()`

- `reduceByWindow` and `reduceByKeyAndWindow`
- Includes a special form that allows reduction to be performed **incrementally**
  - Considering only the data coming into the window and the data that is going out
  - Special form requires an **inverse** of the reduce function
    - Such as – for +
  - More efficient for large windows if your function has an inverse

## Difference between naïve and incremental `reduceByWindow()`








## Maintaining state across batches

- `updateStateByKey()`
  - Provides access to a state variable for DStreams of key/value pairs
  - Given a DStream of (key, value) pairs
    - Construct a new DStream of (key, state) pairs by taking a function that specifies how to update the state for each key, given new events

## The contents of this slide-set are based on the following references


- *Learning Spark: Lightning-Fast Big Data Analysis. 1st Edition. Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. O'Reilly. 2015. ISBN-13: 978-1449358624.*
  [Chapter 10]
- Spark Streaming Programming Guide: http://spark.apache.org/docs/latest/streaming-programming-guide.html#memory-tuning
- Processing Twitter Streams using Spark: https://databricks-training.s3.amazonaws.com/realtime-processing-with-spark-streaming.html