CS 677: Big Data

Spark Streaming

Today's Schedule

- Discretized Streams
- Fault Tolerance

Today's Schedule

- **Discretized Streams**
- Fault Tolerance

Streams

- As we discussed before, pretty much any big data problem can be viewed as a streaming problem
- You'll rarely have all your data instantly in memory, so you will have to stream it from somewhere
  - Disk
  - Network
  - Kafka, storm, etc?
- Stream processing systems operate on this data while it is **in flight**

Spark Streaming

- Spark was originally **not** designed to be a stream processing system
    - Focused on batch jobs
- However, RDDs lend themselves fairly well to a particular type of streaming: **microbatches**
    - Don’t operate on each individual item streaming into the system
    - Instead, collect small **batches** over a window of time and process them instead

Creating a StreamingContext

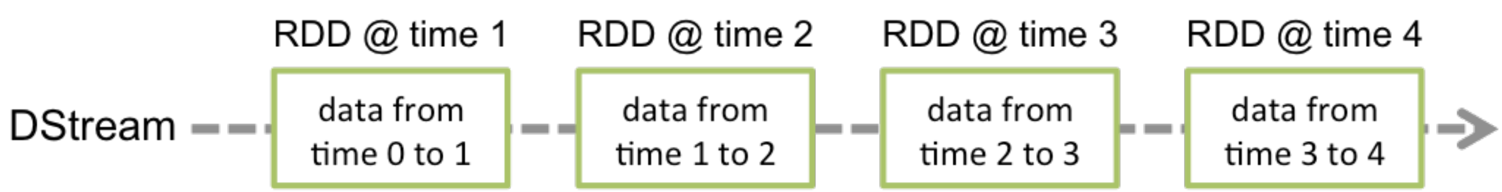
- ssc = StreamingContext(sc, N)
  - Where **sc** is your SparkContext
  - **N** is the batch interval (number of seconds between microbatches)
- Once your pipeline is set up, you can execute it with:
  - ssc.start()
- The computation will run forever, at least until you stop it with:
  - ssc.stop(stopSparkContext=False)
    - Without the parameter, your entire context is shut down and your driver will need to be restarted

Setting the Batch Interval

- You can specify very small batch intervals
- However, you should tune your batch interval based on how fast the stream can be processed
  - Small interval = more processing, but more “up to date”
  - Large interval = less processing, less frequent updates
- Use the web interface to check that the **batch processing time** is less than the batch interval

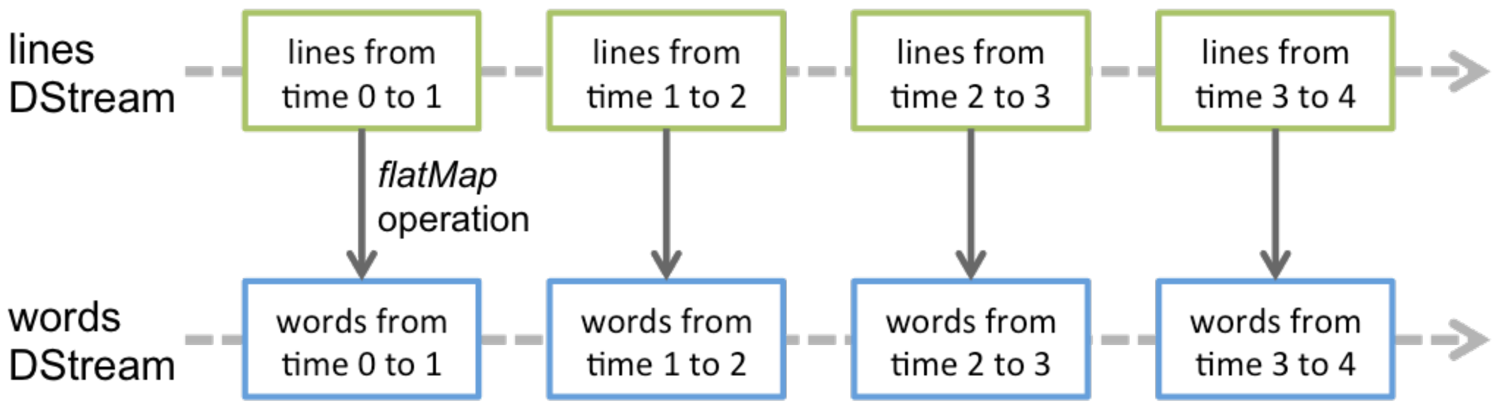
DStreams

- Microbatches are represented as **DStreams** (discretized streams)
- For each time step (specified by the user), Spark generates a new RDD that represents the microbatch



Source: *Spark Streaming Programming Guide*

WordCount with DStreams



Source: *Spark Streaming Programming Guide*

Transformations

- Okay, so a DStream is a collection of RDDs gathered over time as data streams into the system…
- …so that means they have roughly the same capabilities!
  - **Transformations** are largely the same
    - Even their laziness
  - We don't have terminal **actions** because the stream is assumed to be infinite
    - However, we DO have **output operations** like writing to a file, printing, etc.

Stateful Streaming

- In many cases, you'll want your streaming jobs to maintain **state** information
  - Watching trends over time, catching and handling anomalies, etc.
- There are two primary ways to do this:
  - updateStateByKey
  - foreachRDD

updateStateByKey

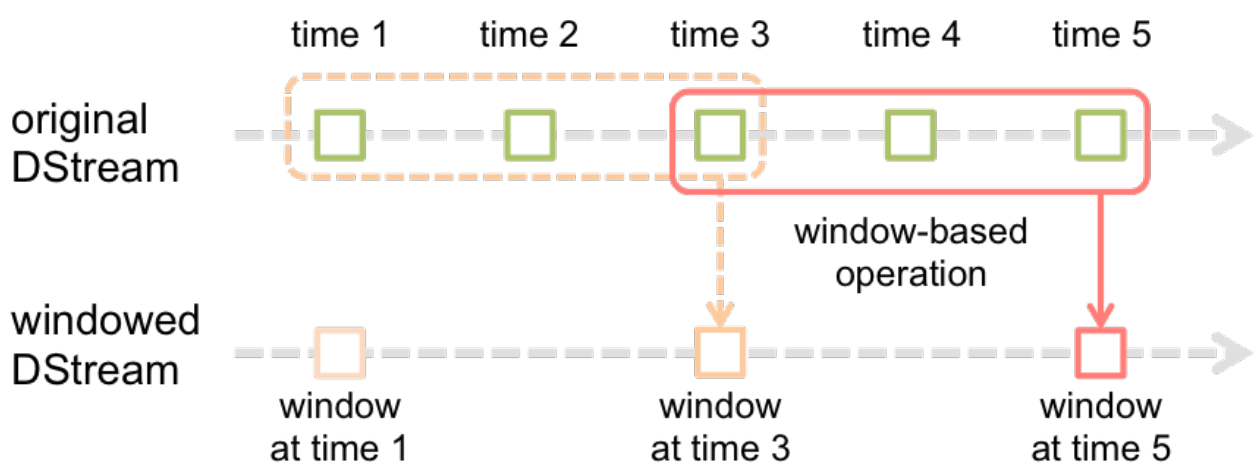
- Can be used to maintain state throughout the stream as a separate DStream
  - Sort of like a continuously-running reduce operation
- Takes a user function as a parameter
  - Current RDD state
  - Previous (potentially aggregated) RDD state

foreachRDD

- Kind of like a streaming version of **.collect()** but generally not as dangerous to use
    - (stream batches tend to be on the smaller side)
- Applies a user function to each RDD in a DStream **on the driver**
    - Good for doing lightweight updates, drawing visualizations, etc.

Windowed Computations

- You can also apply operations over sliding windows of data (spanning multiple RDDs) rather than just the individual RDDs you get every time unit



Source: *Spark Streaming Programming Guide*

Today's Schedule

- Discretized Streams
- **Fault Tolerance**

Fault Tolerance

- Handling failures tends to be more important in a streaming setup
  - After all, you can't just go back and read the data from the disk! It may be lost completely
  - Some stream sources, such as Kafka and HDFS do allow **replay** in the case of lost events
- If you are going to maintain state (e.g., with updateStateByKey) then you need to set up checkpointing

Checkpointing

- To set a checkpoint directory:
  - ssc.checkpoint(”hdfs://location/to/store")
- For our purposes, you may choose to skip checkpointing to HDFS
  - Not the end of the world if we lose data!

Event Processing Guarantees

- All streaming systems must choose event processing guarantees
- *At most once*: Records are processed either **once** or **not at all.**
- *At least once*: Records are processed **one or more times**. More reliable, but must deal with duplicates.
- *Exactly once*: Records are processed once with no data loss or duplicates.

Fault Tolerance: Input

- With files, HDFS, or Kafka, inputs are guaranteed to be processed exactly once
- With a general data stream, **reliable receivers** verify data has been received
    - In this case, records are processed **at least once**
- **Unreliable receivers** that do not verify receipt will result in loss of all buffered data if a failure occurs
    - In this case, records are processed **at most once**

Fault Tolerance: Output

- Output operations are processed **at least once**
- This includes writing to files or even applying a foreachRDD operation
- Extra processing needs to be done if duplicates cannot be present in the output data

What's Next?

- Much like RDDs, there is more to the story here
- **Structured Streaming** allows DataSet-like functionality over streams
- The tradeoff: latency and fault tolerance
- Structured Streaming:
    - Exactly-once delivery
    - High latency (could be hundreds of milliseconds)
- DStreams:
    - At-least-once delivery
    - Latencies in the low milliseconds