Spark Program

# CHAPTER 5:
# SPARK STREAMING

# Chapter Objectives

In this chapter, we will:

- Learn the special processing needs for the high-velocity data under Spark’s streaming architecture
- Explore various streaming data sources

# Introduction to Spark Streaming

- Spark Streaming is an extension of the core API
  - Provides high-throughput stream processing of live data
  - Built on Spark's fault-tolerant and highly scalable architecture
- Support for a wide variety of data sources
  - File systems, TCP Sockets, Kafka, Flume, Twitter, Kinesis, and ZeroMQ
  - Implement custom *Receivers* to integrate arbitrary data sources
- Streams can be processed using complex algorithms
  - Designed and implemented using:
    - Spark transformations and actions
    - Sliding window operations
    - The other extension APIs for SQL, Machine Learning, R, and GraphX
- Processed data can be pushed out to:
  - File systems, databases, live dashboards, etc.

# Discretized Streams and RDDs

- Live input streams are divided into batches of data items
  - Known as *discretized streams*
  - Spark's high-level abstraction is called a DStream
- The batching interval is specified when the DStream is created
  - A new RDD is generated every interval for the batch of collected data
    - Spark operations are then applied to the RDD
  - Windowed DStream instances will contain multiple RDDs
    - A configurable number of historical RDDs from earlier batches
    - Spark operations are applied to their aggregate
- Input streams are obtained from Receivers
  - Built-in or custom classes for generating DStreams from data sources
  - Received data is reliably stored in Spark's memory for processing
- Spark provides two categories of supported streaming sources
  - Basic sources include file and socket streams
  - Advanced sources include Kafka, Flume, Kinesis, Twitter, etc.

# Obtaining a DStream

- Applications must create a `StreamingContext`
  - Can use an existing `SparkContext` or `SparkConf` instance
    - At least two threads must be specified for the Worker/Executor and Receiver
  - Specify the batch interval

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
sc = SparkContext("local[2]", "textStream")
sc.setLogLevel('Error')
ssc = StreamingContext(sc, 5)
lines = ssc.socketTextStream('localhost', 9999)
words = lines.flatMap(lambda line: line.split(' '))
pairs = words.map(lambda w: (w, 1))
wordCount = pairs.reduceByKey(lambda x, y : x + y)
wordCount.pprint()
ssc.start()
ssc.awaitTerminationOrTimeout(10000)
ssc.stop()
```

# Setting the Batch Rate

- Applications should be able to process data as fast as it is being received
  - The batch rate is specified when the `StreamingContext` is instantiated
- The batch processing time should be less than the specified batch interval
  - Start with a longer than expected batch interval
    - Monitor the processing time and then reduce the batch interval
- Processing times can be monitored
  - Many useful streaming statistics are reported in the Web UI
  - The driver log files contain a “Total delay” entry
- Momentary delays may be acceptable
  - As long as the delay reduces back to an appropriate value
  - Caused by temporary increases in the data rate
- Sustained delays will accumulate over time
  - The application will not be able to keep up and may become unstable
- The level of processing parallelism can also be increased

# DStream Transformations

- The following transformations are available on DStream instances: `map()`, `flatMap()`, `filter()`, `reduce()`, `reduceByKey()`, and `count()`
- `countByKey()`, `cogroup()`, and `join()`
  - These operate in exactly the same way as the standard RDD equivalents
- `transform()`
  - Applies any RDD to RDD operation not exposed by the DStream API
- `union()`
  - Combines multiple DStream instances of the same type

# DStream Output Operations

- Output operations trigger the execution of any pending transformations
  - They behave like RDD actions
- Currently, DStream instances provide three save operations
  - Filenames are generated using the time with a prefix and optional suffix
    - Provided as arguments
      - `prefix-TIME_IN_MS.suffix`
- `saveAsTextFiles()`
- `saveAsObjectFiles()`
  - Save the DStream as serialized Java SequenceFile
  - Values only, keys are serialized as NullWriteable
- `saveAsHadoopFiles()`
  - Requires a Hadoop OutputFormat
  - The DStream must contain (key, value) pairs
- `foreachRDD(func)`
  - Generic output operator that calls the `func` for each DStream

# File Streams

- `fileStream()` monitors a file system directory
  - Called on a `StreamingContext` instance
  - Delegates to an underlying Hadoop InputFormat

- Files must be created in, or copied/moved to, the monitored directory
  - If required, existing files can be processed at start-up
  - File names starting with "." are ignored
    - A generic filename filter can be specified

- The `textFileStream()` method delegates to a TextInputFormat

- For full details, view the Spark Scala API documentation

```
ssc = StreamingContext(sc, 5)
lines = ssc.textFileStream('stream')
words = lines.flatMap(lambda line: line.split(' '))
pairs = words.map(lambda w: (w, 1))
wordCount = pairs.reduceByKey(lambda x, y : x + y).transform(lambda x :
x.sortBy(lambda x : -x[1]))
wordCount.pprint()
ssc.start()
ssc.awaitTerminationOrTimeout(10000)
ssc.stop()
```

# DataFrames

- DStreams can be turned into DataFrames and then temporary views to make it easier to process them
- You need do a little trick to make the `SparkContext` object the same as the `StreamingContext`
  - The following lazy evaluated singleton instance will help do that

```
def getSparkSessionInstance(sparkConf):
    if ("sparkSessionSingletonInstance" not in globals()):
        globals()["sparkSessionSingletonInstance"] = \
            SparkSession.builder \
            .config(conf=sparkConf) \
            .getOrCreate()
    return globals()["sparkSessionSingletonInstance"]
```

# DataFrames (continued)

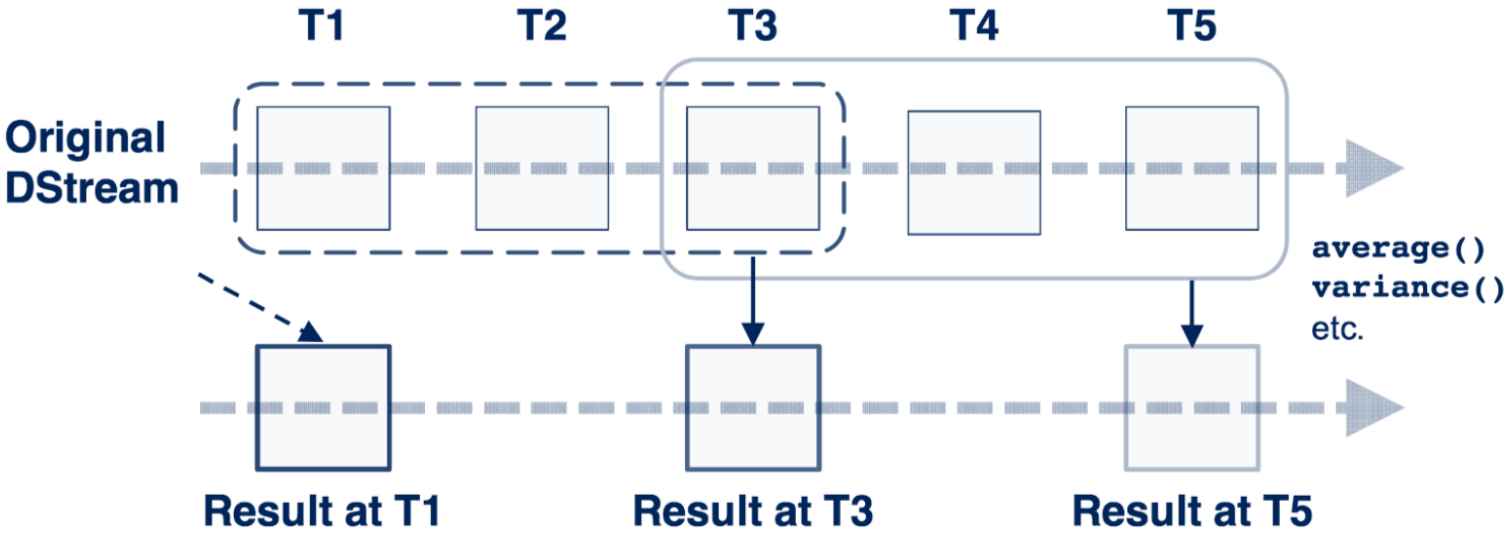
- To convert the DStream, you need to call the `foreachRDD` and pass it a function to call
- You need do a little trick to make the `SparkContext` object the same as the `StreamingContext`
  - The following lazy evaluated singleton instance will help do that

```
def process(time, rdd):
  try:
    spark = getSparkSessionInstance(rdd.context.getConf())
    rdd1 = rdd.map(lambda x : x[1].split(',')) \
              .map(lambda x : (int(x[0]), float(x[1])))
    df = spark.createDataFrame(rdd1, schema='id:int, amount:float')
    df.createOrReplaceTempView('newdata')
    join = spark.sql('select n.id, c.name, n.amount from newdata as n
           join codes as c on n.id = c.id')
    join.show()
  except:
    print(rdd.collect())

stream.foreachRDD(process)
```

# Window Operations

- Spark streaming also provides an API for window computations
- As the window slides over a source DStream:
  - Operations are applied to the aggregate of RDDs that fall within the window
- Window operators must specify two parameters
  - Window length: the duration of the window
  - Sliding interval: the number of intervals to advance the window



# Window Transformations

- Spark provides the following windowed transformations
  - Each requires a *window length* and *slide interval*
- `window()`
  - Returns DStream based on the *window length* and *slide interval*
- `countByWindow()`
  - Counts the number of elements in the window
- `countByValueAndWindow()`
  - Expects a DStream of (key, value) pairs
  - Returns a new DStream of (key, long) pairs in the window
- `reduceByWindow()` and `reduceByKeyAndWindow()`
  - Applies a reducing function to the values or (key, values) in the window
- The operation of `reduceByKeyAndWindow()` can be optimized
  - As a window slides, the reduced value can be calculated incrementally
    - An *inverse-reduce* function can be specified to remove old values
    - New values are then amalgamated by the reduce function

# Advanced Streaming Sources

- Spark supports a number of advanced streaming sources
  - Not part of the core Spark API and require additional libraries
- Apache Kafka
  - A distributed publish-subscribe messaging system written in Scala
    - Designed to be fast, scalable, and robust
  - Originally developed by LinkedIn and became open source in 2011
- Apache Flume
  - Highly available distributed service for collecting and aggregating data
    - Designed to handle very large quantities of data
    - Originally developed by Cloudera
- Amazon Kinesis
  - Commercial EC2 service for collecting and processing stream-based data
    - Highly scalable and designed for used by real-time applications
    - Provide a Kinesis Client Library (KCL) under the Amazon Software License
- Twitter

# Chapter Summary

In this chapter, we have:

- Learned the special processing needs for the high-velocity data under Spark's streaming architecture
- Explored various streaming data sources