Topics covered in this lecture

- Spark Streaming
  - Architecture and Abstractions
  - Execution
  - Stateful and stateless transformations
  - Windowed operations
  - Performance considerations
  - Example

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

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:
- Data from time 0 to 1
- Data from time 1 to 2
- Data from time 2 to 3
- Data from time 3 to 4

DStreams support output operations, such as the `print()` used in our example.
- 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: 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

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

- **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

- **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

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

Examples of stateless transformations

- **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

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

Examples of stateless transformations

- **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
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()` tracks state across events for each key

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 Durations/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, event) 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

Performance considerations
- Batch size
  - **500 milliseconds** is considered a good minimum size
  - Start with a large batch size (~10 seconds) and work down to a smaller batch size
    - If processing times remain consistent, explore decreasing the batch size
    - If the processing times increase? You have reached the limit
- Window size
  - Has a great impact on performance
  - Consider increasing this for expensive operations

Garbage collections and memory usage
- Cache RDDs in serialized form
  - Using Kryo for serialization reduces this even more
    - Reduces space for in-memory representations
  - By default, Spark uses an in-memory cache
    - Can also evict RDDs older than a certain time-period
    - `spark.cleaner.ttl`
    - This preemptive eviction of RDDs also reduces the garbage collection pressure
Levels of parallelism in data receiving [1/4]

- Each input DStream creates a single receiver that receives a single stream of data
- Receiving multiple data streams possible by creating multiple input DStreams
  - Each DStream must be configured to receive different partitions of the data stream from the source(s)
- For a Kafka DStream receiving data on two topics?
  - Split into two DStreams each receiving one topic
  - Two receivers would run and receive data in parallel

April 4, 2017
CS455: Introduction to Distributed Systems [Spring 2017]
Dept. Of Computer Science, Colorado State University

Levels of parallelism in data receiving [2/4]

- Another approach is to tune the receiver’s block interval
  - Determined by spark.streaming.blockInterval
- For most receivers, received data is coalesced into blocks of data before storing in memory
- The number of blocks in each batch determines the number of tasks used to process the received data in a map-like transformation
- Number of tasks per batch?
  - Batch interval/block interval

April 4, 2017
CS455: Introduction to Distributed Systems [Spring 2017]
Dept. Of Computer Science, Colorado State University

Levels of parallelism in data receiving [3/4]

- Number of tasks per batch?
  - Batch interval/block interval
- Block interval of 200 ms will create 10 tasks per 2 second batches
- If the number of tasks is too low?
  - All available cores might not be available to use all the data
- To increase number of tasks for a given batch interval?
  - Reduce the block interval

April 4, 2017
CS455: Introduction to Distributed Systems [Spring 2017]
Dept. Of Computer Science, Colorado State University

Levels of parallelism in data receiving [4/4]

- What if you did not want to receive data with multiple input streams?
  - Explicitly repartition the input data stream
- Repartitioning is done using the inputStream.repartition(<number of partitions>)
  - Distributes the received batches of data across the specified number of machines in the cluster before further processing

April 4, 2017
CS455: Introduction to Distributed Systems [Spring 2017]
Dept. Of Computer Science, Colorado State University

Data serialization [1/2]

- Data received through receivers is stored with StorageLevel.MEMORY_AND_DISK_SER_2
- Data that does not fit in memory spills over to disk
- Input data and persisted RDDs generated by DStream transformations are automatically cleared
  - If you are using a window operation of 10 minutes, then Spark Streaming will keep around the last 10 minutes of data, and actively throw away older data
- Data can be retained for a longer duration by setting streamingContext.remember

April 4, 2017
CS455: Introduction to Distributed Systems [Spring 2017]
Dept. Of Computer Science, Colorado State University

Data serialization [2/2]

- RDDs generated by streaming computations may be persisted in memory
  - Persisted RDDs generated by streaming computations are persisted with StorageLevel.MEMORY_ONLY_SER
- If you are using batch intervals of a few seconds and no window operations?
  - You can try disabling serialization in persisted data
    - Reduce CPU overheads due to serialization, without excessive GC overheads.

April 4, 2017
CS455: Introduction to Distributed Systems [Spring 2017]
Dept. Of Computer Science, Colorado State University
Processing Twitter Streams using Spark

April 4, 2017

Spark-streaming example [1/5]

- Step-by-step approach to finding the top 10 hashtags from a stream of tweets using counts (Every second there is an output over data from the last 300 seconds)
- Step-1: Create a SparkStream context and Twitter credential setup

```java
SparkConf sparkConf = new SparkConf().setAppName("Spark-streaming-twitter-trends");
```

//Twitter authentication details - [Not included here]

//JavaStreamingContext
JavaStreamingContext jssc = new JavaStreamingContext(sparkConf, new Duration(1000));

//Discretized stream of tweets
JavaDStream<Status> twitterStream = (JavaDStream<Status>) TwitterUtils.createStream(jssc);
```

Spark-streaming example [2/5]

- Step-2: Map Input DStream of Status to String

```java
//Discretized stream of Strings
JavaDStream<String> statuses = twitterStream.map(  
    new Function<Status, String>() {  
        public String call(Status status) {  
            return status.getText();  
        }  
    });

statuses.print();

//trigger the execution of code
jssc.start();
jssc.awaitTermination();
```

Spark-streaming example [3/5]

- Step-3: Stream of hashtags from stream of tweets

```java  
//Tokenize words from status
JavaDStream<String> wordsFromStatuses = statuses.flatMap(  
    new FlatMapFunction<String, String>() {  
        public Iterable<String> call(String input) {  
            return Arrays.asList(input.split(" "));  
        }  
    });

//Extract hashtags
JavaDStream<String> hashTags = wordsFromStatuses.filter(  
    new Function<String, Boolean>() {  
        public Boolean call(String word) {  
            return word.startsWith("#");  
        }  
    });
```

Spark-streaming example [4/5]

- Step-4: Count the hashtag over 5 min window

```java
//Mapping to tuple of (hashtag,1) in order to count
JavaPairDStream<String, Integer> hashtagtuples = hashTags.mapToPair(  
    new PairFunction<String, String, Integer>() {  
        public Tuple2<String, Integer> call(String input) {  
            return new Tuple2<String, Integer>(input, 1);  
        }  
    });

//Aggregating over window of 5 min and slide of 1s
JavaPairDStream<Integer, String> counts = hashtagtuples.reduceByKeyAndWindow(  
    new Function2<Integer, Integer, Integer>() {  
        public Integer call(Integer int1, Integer int2) {  
            return int1 + int2;  
        }  
    },  
    new Function2<Integer, Integer, Integer>() {  
        public Integer call(Integer int1, Integer int2) {  
            return int1 - int2;  
        }  
    },  
    new Duration(60 * 5 * 1000), new Duration(1 * 1000));
```

Spark-streaming example [5/5]

- Step-5: Find top 10 hashtags according to counts

```java
JavaPairDStream<Integer, String> swapCounts = counts.mapToPair(  
    new PairFunction<Tuple2<String, Integer>, Integer, String>() {  
        public Tuple2<Integer, String> call(Tuple2<String, Integer> input) {  
            return input.swap();  
        }  
    });

JavaPairDStream<Integer, String> sortedCount = swapCounts.transformToPair(  
    new Function<JavaPairRDD<Integer, String>, JavaPairRDD<Integer, String>>(){  
        public JavaPairRDD<Integer, String> call(JavaPairRDD<Integer, String> input)  
            throws Exception {  
            return input.sortByKey(false);  
        }  
    });

sortedCount.foreach(new Function<JavaPairRDD<Integer, String>, Void> () {  
    public Void call(JavaPairRDD<Integer, String> rdd) {  
        String out = "Trending hashtags:
";
        for (Tuple2<Integer, String> t: rdd.take(10)) {  
            out = out + t.toString() + "  
        }
        System.out.println(out);
    }});
```

CS455: Introduction to Distributed Systems [Spring 2017]
Dept. Of Computer Science, Colorado State University
The contents of this slide-set are based on the following references:

