Frequently asked questions from the previous class survey

- Lazy Evaluation: Does it increase the pre-processing time?
- What does it entail to load lazily?
- When we perform a transformation on an old RDD to produce a new RDD is the old RDD loaded in memory?
- If we don’t persist the data, and it changes dynamically is it possible to get different data?
- Can you use different data sources in one Spark session?
- Who decides LRU policy for RDD evictions?
- The foreach() action produces a new RDD or mutates one?
- Set operations: Done using the API method calls
- Transformations where data needs to be exchanged between tasks?
- If data is stored in chunks (as bytes[]), two parts of a line can be on different machines?
- Combinations of Apache systems for batch (Hadoop) and real-time systems (Storm)

Topics covered in this lecture

- Pair RDDs
- Spark Streaming

Pair RDDs

- RDDs that contain key/value pairs
- Expose partitions that allow you to act on each key in parallel or regroup data across the network

Creating Pair RDDs

```scala
pairs = lines.map(lambda x: (x.splitted(), 0, x))
```

- Creates a pair RDD using the first word as the key
- Java does not have a built-in tuple type

```scala
scala.Tuple2 class
new Tuple2(elem1, elem2)
```
Transformations on Pair RDDs [1/5]

- Pair RDD = \{(1,2), (3,4), (3,6)\}
- `reduceByKey(func)`
  - Combine values with the same key
  - Invocation: `rdd.reduceByKey((x, y) => x + y)`
  - Result: \{(1, 2), (3, 10)\}

Transformations on Pair RDDs [2/5]

- Pair RDD = \{(1,2), (3,4), (3,6)\}
- `groupByKey(func)`
  - Group values with the same key
  - Invocation: `rdd.groupByKey()`
  - Result: \{(1, [2]), (3, [4, 6])\}

Transformations on Pair RDDs [3/5]

- Pair RDD = \{(1,2), (3,4), (3,6)\}
- `mapValues(func)`
  - Apply function to each value of a pair RDD without changing the key
  - Invocation: `rdd.mapValues(x=> x+1)`
  - Result: \{(1, 3), (3, 5), (3, 7)\}

Transformations on Pair RDDs [4/5]

- Pair RDD = \{(1,2), (3,4), (3,6)\}
- `values()`
  - Return an RDD of just the values
  - Invocation: `rdd.values()`
  - Result: \{2, 4, 6\}

Transformations on Pair RDDs [5/5]

- Pair RDD = \{(1,2), (3,4), (3,6)\}
- `sortByKey()`
  - Return an RDD sorted by the key
  - Invocation: `rdd.sortByKey()`
  - Result: \{(1, 2), (3, 4), (3, 6)\}
Transformations on two Pair RDDs [1/5]

- **rdd** = {(1,2), (3,4), (3,6)}  
  **other** = {(3,9)}

- **subtractByKey()**
  - Remove elements with a key present in the other RDD
  - Invocation: rdd.subtractByKey(other)
  - Result: { (1,2) }

Transformations on two Pair RDDs [2/5]

- **rdd** = {(1,2), (3,4), (3,6)}  
  **other** = {(3,9)}

- **join()**
  - Perform an inner join between two RDDs. Only keys that are present in both pair RDDs are output
  - Invocation: rdd.join(other)
  - Result: { (3, (4,9)) , (3, (6,9)) }

Transformations on two Pair RDDs [3/5]

- **rdd** = {(1,2), (3,4), (3,6)}  
  **other** = {(3,9)}

- **leftOuterJoin()**
  - Perform a join between two RDDs where the key must be present in the first RDD.
  - Tuple has an option for the source rather than other RDD. Tuple has an option for the source rather than other RDD
  - Invocation: rdd.leftOuterJoin(other)
  - Result: { (1, (2,None)) , (3, (4, 9)) , (3, (6, 9)) }

Transformations on two Pair RDDs [4/5]

- **rdd** = {(1,2), (3,4), (3,6)}  
  **other** = {(3,9)}

- **rightOuterJoin()**
  - Perform a join between two RDDs where the key must be present in the other RDD.
  - Invocation: rdd.rightOuterJoin(other)
  - Result: { (3, (4,9) ) ,  (3, (6,9)) }

Transformations on two Pair RDDs [5/5]

- **rdd** = {(1,2), (3,4), (3,6)}  
  **other** = {(3,9)}

- **cogroup()**
  - Group data from both RDDs using the same key
  - Invocation: rdd.cogroup(other)
  - Result: { (1, [(2,[])]) , (3, [(4, 6), [9]]) }

---

**Transformations on two Pair RDDs**

October 22, 2015  
Instructor: SHRIDEEP PALLICKARA  
Dept. Of Computer Science, Colorado State University
Example of chaining operations

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>panda</td>
<td>0</td>
</tr>
<tr>
<td>pink</td>
<td>3</td>
</tr>
<tr>
<td>pirate</td>
<td>3</td>
</tr>
<tr>
<td>panda</td>
<td>1</td>
</tr>
<tr>
<td>pink</td>
<td>4</td>
</tr>
</tbody>
</table>

```
rdd.mapValues(x => (x, 1)).reduceByKey((x, y) => (x._1 + y._1, x._2 + y._2))
```

A word count example

- We are using flatMap() to produce a pair RDD of words and the number 1
- Using reduceByKey() to count occurrences

```scala
rdd = sc.textFile("s3://...")
words = rdd.flatMap(lambda x: x.split(" "))
result = words.map(lambda x: (x,1)).reduceByKey(lambda x, y: (x+y))
```

Tuning the level of parallelism

- Every RDD has a fixed number of partitions
- Determine the degree of parallelism when executing operations
- During aggregations or grouping operations, you can ask Spark to use a specific number of partitions
- This will override defaults that Spark uses

Example: Tuning the level of parallelism

```
data = ["a", 3, "b", 4, "a", 1]
sc.parallelize(data).reduceByKey(lambda x, y: x+y) #default
sc.parallelize(data).reduceByKey(lambda x, y: x+y, 10) #Custom
```
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 only in Java and Scala

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)

Example

- 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

Example

```java
JavaStreamingContext jsc = new JavaStreamingContext(conf, Durations.seconds(1));
JavaDStream<String> lines = jsc.socketTextStream("localhost", 7777);
JavaDStream<String> errorLines = lines.filter(new Function<String, Boolean>() {
    public Boolean call(String line) {
        return line.contains("error");
    }
});
```
Spark Streaming Architecture

- Spark Streaming uses a **micro-batch** architecture
  - Streaming computation is treated as a 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

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

- 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();`

High-level architecture of Spark Streaming

- **Input Data Streams**
- **Batches of input data**
- **Spark Streaming**
- **Spark**
- Results pushed to external systems

DStreams and the transformations in our example

- Server running at localhost:7777
- Data from time 0 to 1
- Data from time 1 to 2
- Data from time 2 to 3
- Data from time 3 to 4
- error lines from time 0 to 1
- error lines from time 1 to 2
- error lines from time 2 to 3
- error lines from time 3 to 4
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.

Fault Tolerance

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

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() track state across events for each key.
  - Requires checkpointing to be enabled in StreamingContext.

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.
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
  - If the batch interval is 10 seconds and the sliding window is 30 seconds ... last 3 batches

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

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.cleanner.ttl`
  - This preemptive eviction of RDDs also reduces the garbage collection pressure

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