Frequently asked questions from the previous class survey

- Difference between x and x_1? x tuple
- Does optimizer run before RDD is setup?
- Only transformations change the lineage graph?
- Are lineage graphs replicated as well?
- How can I make sure that the data is replicated?
- When would you use these different persistence levels?
- Can you change these persistence levels?
- Lots of similarities with SQL?

March 30, 2017

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

- pairs=lines.map(lambda x: (x.split(" "))[0], x))
  - Creates a pairRDD using the first word as the key
- Java does not have a built-in tuple type
  - scala.Tuple2 class
    - new Tuple2(elem1, elem2)
**Transformations on Pair RDDs**

**March 30, 2017**

---

**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
  - Value associated with each key is a tuple of the value from the source and an Option for the value from the other pair RDD
  - In python if a value is not present, None is used.
  - 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
  - Tuple has an option for the source rather than 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]]) }
Example of chaining operations:
Calculation of per-key average

<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>

**mapValues:**

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

**reduceByKey:**

```scala
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

```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

```scala
data = [('a', 3), ('b', 4), ('a', 1)]
s.parallize(data).reduceByKey(lambda x, y: x + y) #default
s.parallize(data).reduceByKey(lambda x, y: x + y, 10) #Custom
```

What if you want to tune parallelism outside of grouping and aggregation operations?

- There is `repartition()`
  - Shuffles data across the network to create a new set of partitions
  - Very expensive operation!

- There is the `coalesce()` operation
  - Allows avoiding data movement
    - But only if you are **decreasing** the number of partitions
  - Check `rdd.getNumPartitions()` and make sure you are coalescing to fewer partitions than current
**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 and Scala
- Recent support for 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)

**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”

```java
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()`

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

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