### Topics covered in this lecture
- Pair RDDs
- Dependencies and Transformations
- Partitioners
- Spark Streaming

### Transformations on Pair RDDs

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

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

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)
```

Example of chaining operations: Calculation of per-key average

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

Wide and Narrow Transformations
Transformations and Dependencies

- Two categories of dependencies
  - Narrow
    - Each partition of the parent RDD is used by at most one partition of the child RDD
    - The narrow versus wide distinction has significant implications for the way Spark evaluates a transformation and, consequently, for its performance
  - Wide
    - Multiple child RDD partitions may depend on a single parent RDD partition

Narrow Transformations

- Narrow transformations are those in which each partition in the child RDD has simple, finite dependencies on partitions in the parent RDD
- Can be determined at design time, irrespective of the values of the records in the parent partitions
- Partitions in narrow transformations can either depend on:
  - One parent (such as in the map operator), or
  - A unique subset of the parent partitions that is known at design time (coalesce)
- Narrow transformations can be executed on an arbitrary subset of the data without any information about the other partitions.

Narrow Transformations: Dependencies between partitions

Wide Transformations

- Transformations with wide dependencies cannot be executed on arbitrary rows
- Require the data to be partitioned in a particular way, e.g., according to the value of their key
- In sort, for example, records have to be partitioned so that keys in the same range are on the same partition
- Transformations with wide dependencies include sort, reduceByKey, groupByKey, join, and anything that calls the repartition function

Wide Transformations: Dependencies between partitions

TUNING THE LEVEL OF PARALLELISM

Wide dependencies cannot be known fully before the data is evaluated
The dependency graph for any operations that cause a shuffle (such as groupByKey, reduceByKey, sort, and sortByKey) follows this pattern
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

```python
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
```

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

PAIR RDDS: WHAT TO WATCH FOR

- Despite their utility, key/value operations can lead to a number of performance issues
  - Most expensive operations in Spark fit into the key/value pair paradigm
  - Because most wide transformations are key/value transformations,
    - And most require some fine tuning and care to be performant
- In particular, operations on key/value pairs can cause
  1. Out-of-memory errors in the driver
  2. Out-of-memory errors on the executor nodes
  3. Shuffling failures
  4. "Straggler tasks" or partitions, which are especially slow to compute
- The last three performance issues are all most often caused by shuffles associated with the wide transformations
Memory errors in the driver, is usually caused by actions

- Several key/value actions (including countByKey, countByValue, lookUp, and collectAsMap) return data to the driver
- In most instances they return unbounded data since the number of keys and the number of values are unknown
- In addition to number of records, the size of each record is an important factor in causing memory errors

Preventing out-of-memory errors with aggregation operations [1/2]

- combineByKey and all of the aggregation operators built on top of it (reduceByKey, foldLeft, foldRight, aggregateByKey) may lead to memory errors if they cause the accumulator to become too large for one key
- What about groupByKey?
  - It is actually implemented using combineByKey where the accumulator is an iterator with all the data.

Preventing out-of-memory errors with aggregation operations [2/2]

- Use functions that implement map-side combinations
  - Meaning that records with the same key are combined before they are shuffled
  - This can greatly reduce the shuffled read
- The following four functions are implemented to use map-side combinations
  - reduceByKey
  - treeAggregate
  - aggregateByKey
  - foldByKey

Two primary techniques to avoid performance problems associated with shuffles

- Shuffle Less
- Shuffle Better

Shuffle Less

- Preserve partitioning across narrow transformations to avoid reshuffling data
- Use the same partitioner on a sequence of wide transformations. This can be particularly useful:
  - To avoid shuffles during joins and …
  - To reduce the number of shuffles required to compute a sequence of wide transformations

Shuffle Better [1/2]

- Sometimes, computation cannot be completed without a shuffle
- However, not all wide transformations and not all shuffles are equally expensive or prone to failure
Shuffle Better

- By using wide transformations such as `reduceByKey` and `aggregateByKey` that can perform map-side reductions and that do not require loading all the records for one key into memory?
  - You can prevent memory errors on the executors and speed up wide transformations, particularly for aggregation operations.
- Lastly, shuffling data in which records are distributed evenly throughout the keys, and which contain a high number of distinct keys?
  - Prevents out-of-memory errors on the executors and "straggler tasks."

Partitioners

- The partitioner defines how records will be distributed and thus which records will be completed by each task.
- Practically, a partitioner is actually an interface with two methods:
  - `numPartitions` that defines the number of partitions in the RDD after partitioning.
  - `getPartition` that defines a mapping from a key to the integer index of the partition where records with that key should be sent.

There are two implementations for the partitioner object provided by Spark:

- `HashPartitioner`:
  - Determines the index of the child partition based on the hash value of the key.
- `RangePartitioner`:
  - Assigns records whose keys are in the same range to a given partition.
  - Required for sorting since it ensures that by sorting records within a given partition, the entire RDD will be sorted.
  - It is possible to define a custom partitioner.

Partitioners and transformations

- Unless a transformation is known to only change the value part of the key/value pair in Spark:
  - The resulting RDD will not have a known partitioner.
  - Even if the partitioning has not changed.

Using narrow transformations that preserve partitioning

- Some narrow transformations, such as `mapValues`, preserve the partitioning of an RDD if it exists.
- Common transformations like `map` and `flatMap` can change the key:
  - So even if your function does not change the key, the resulting RDD will not have a known partitioner.
  - Instead, if you don't want to modify the keys, call the `mapValues` function (defined only on pair RDDs):
    - It keeps the keys, and therefore the partitioner, exactly the same.
    - The `mapPartitions` function will also preserve the partition if it preserves the partitioning flag is set to true.
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"

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

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