Using Hadoop and Java

- None

1 Map/Reduce Introduction

Phases:

- 1. Map
- 2. Local Sort - data sent from map node to a reduce node is sorted
- 3. Local Group (optional) - groups by key
- 4. Combine (optional) - combines values with the same key
- 5. Partition - decides how output of mapper is to be partitioned to the reducers
- 6. Global Sort (merge) - a merge sort is applied at reducer node so all data with same key is grouped together
- 7. Global Group - for each key, create groups
- 8. Reduce - merges everyone with same key to produce a single pair output for each key

Map/Reduce stores all intermediate data on disk.

2 Combiner Functions and Custom Classes

Problems where to use combiner:

- finds all words of size $\geq$ n characters and their **frequency**
- Summing up the values of a key using a SumCountPair class

3 Custom Partitioners, Sorters, and Grouping Comparators

- Custom Partition - define function `getPartition`; on the natural key
- Custom Sort - override `compareTo` method with sort comparator class; if primitive type key then create the custom class; (> 0 ($1 > 2$), 0 ($1 = 2$), < 0 ($1 < 2$))
- Custom Grouping Comparator (recs with same key passed as input to single call to the reduce method) - write a `compare` method
- Natural key is what final result is grouped by (subset of composite key)
- Composite key is output key of mapper (define sorting order on this)

4 Secondary Sort Example

- None

5 Finding Top N Example

- Single call to the reducer when having a top N problem (read problem); all have same key as NULL

6 Outer Joins and Multiple Jobs

- None

7 Intro to Scala

- var vs val (change vs constant)
- fold - takes initial value: `foldLeft(0)((total, x) => total + x)`
- reduce - no initial value: `reduceLeft((acc, x) => acc + x)`
- Tuple starts at 1
- _ in a map function is the current element
- _(num) is the num-th element of the tuple (need the _ to access)
- Iterating over maps with case: `test.map({case (k, v) => println(k + " " + v)})`
- match and case: `val result = x match {case 0 => "zero"; case 1 => "one"}`

8 RDDs in Spark

- flatMap - RDD of lists to RDD of elements; example input: `List(List(1, 2), List(3, 4))` output: `List(1, 2, 3, 4)`; `flatMap(_.split(" "))` - splits each line into words
- saveAsTextFile - saves RDD to a file
- union, intersection, subtract, cartesian; cartesian is combinations of two RDDs
- collect, count, countByValue, take, top, takeOrdered, reduce, fold, foreach, aggregate
- aggregate - `aggregate((0,0)(seqOp, combOp))` - seqOp is applied to each element in the partition and combOp is applied to the results of seqOp; ex. `rdd.aggregate((0,0))((acc, value) => (acc._1 + value, acc._2 + 1), (acc1, acc2) => (acc1._1 + acc2._1, acc1._2 + acc2._2))`

9 Working with Key/Value Pairs

- reduceByKey((x, y) => x + y) - adds up all values with the same key; combine values with the same key
- groupByKey(); returns a list of values for each key
- **mapValues - applies a function to the values of each key**
- sortByKey(); sorts by key (true for ascending, false for descending)
- keyBy - converts RDD of elements to RDD of key/value pairs; ex `rdd.keyBy(_.length)` - adds length as key to each element
- sortBy() - sort by a function; ex. `sortBy(_._(1))`; for descending, use `sortBy(_._(1), false)`
- join - joins two RDDs on the key; ex. `rdd1.join(rdd2)`; output is (key, (value1, value2))
- rightOuterJoin - rdd.join(rdd2) but with all keys in rdd2; if key is not present in rdd, value is None
- leftOuterJoin - rdd.join(rdd2) but with all keys in rdd; if key is not present in rdd2, value is None
- countByKey - returns a map of key to count