AYUSH SINGH

CS 6240: Assignment 2

Combiners and Secondary Sort

Map-Reduce Algorithm

No Combiner

```
def mapper(Object key, Text value, Context context):
        parsedValues = value.split(",")
        id            = parsedValues[0]
        element       = parsedValues[2]
        temp          = parsedValues[3]
        if(element == "TMAX" || element == "TMIN"):
                context.write(id, element + "," + temp)

def reduce(Text key, Text[] values, Context context):
        for val in values:
                parsedValues = val.split(",")
                temp          = toInt(parsedValues[1])
                if parsedValues[0] == "TMAX":
                        tmaxSum  += temp
                        tmaxCount++
                else:
                        tminSum  += temp
                        tminCount++
        output = key + ", " +
                  (double)tminSum/tminCount + ", " +
                  (double)tmaxSum/tmaxCount
        context.write(output, null)
```

Combiner

```
def mapper(Object key, Text value, Context context):
        parsedValues = value.split(",")
        id           = parsedValues[0]
        element      = parsedValues[2]
        temp         = parsedValues[3]
        if element == "TMAX":
                context.write(id, "0,0," + temp + ",1")
        elif element == "TMIN":
                context.write(id, temp + ",1,0,0,")

def combine(Text key, Text[] values, Context context):
        tmaxSum, tmaxCount, tminSum, tminCount = 0
        for val in values:
                parsedValues = val.split(",")
                temp         = toInt(parsedValues[1])
                if parsedValues[0] == "TMAX":
                        tmaxSum  += temp
                        tmaxCount++
                else:
                        tminSum  += temp
                        tminCount++
        output = key + "," + tminSum + "," + tminCount
                     + "," + tmaxSum + "," + tmaxCount
        context.write(key, output)

def reduce(Text key, Text[] values, Context context):
        tmaxSum, tmaxCount, tminSum, tminCount = 0
        for val in values:
                parsedValues = val.split(",")
                tminSum   += toInt(parsedValues[0])
                tminCount += toInt(parsedValues[1])
                tmaxSum   += toInt(parsedValues[2])
                tmaxCount += toInt(parsedValues[3])
        output = key + ", " +
                 (double)tminSum/tminCount + ", " +
                 (double)tmaxSum/tmaxCount
        context.write(output, null)
```

In Mapper Combiner

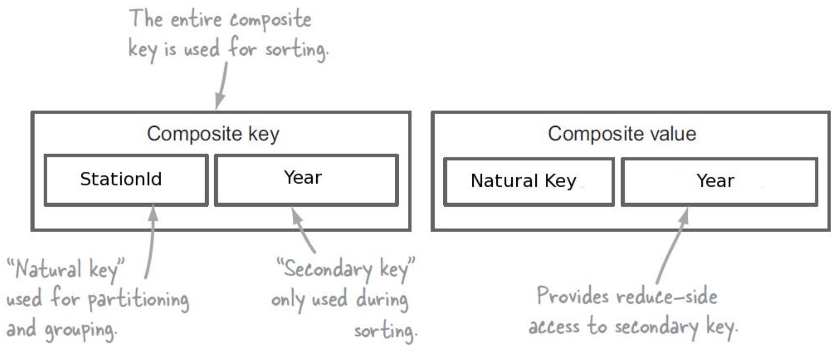
```
# Scope of this associative array(globalMeanMap) is limited to mapper and cleanup methods
globalMeanMap = {}
def mapper(Object key, Text value, Context context):
        parsedValues = value.split(",")
        id           = parsedValues[0]
        element      = parsedValues[2]
        temp         = parsedValues[3]
        if(globalMeanMap.containsKey(id)):
                t = globalMeanMap.get(id)
                if element == "TMAX":
                        if t[3] > 0: temp += t[2]
                        globalMeanMap.put(id, {t[0], t[1], temp, ++t[3]})
                if element == "TMIN":
                        if t[3] > 0: temp += t[2]
                        globalMeanMap.put(id, {temp, ++t[1], t[2], t[3]})
        else:
                if element == "TMAX":
                        globalMeanMap.put(id, {0, 0, temp, 1})
                if element == "TMIN":
                        globalMeanMap.put(id, {temp, 1, 0, 0})

def cleanup(Context context):
        for (key, value) in globalMeanMap:
                context.write(key,    value[0] + "," + value[1] + "," +
                                      value[2] + "," + value[3])

def reduce(Text key, Text[] values, Context context):
        tmaxSum, tmaxCount = 0
        tminSum, tminCount = 0
        for val in values:
                parsedValues = val.split(",")
                tminSum   += toInt(parsedValues[0])
                tminCount += toInt(parsedValues[1])
                tmaxSum   += toInt(parsedValues[2])
                tmaxCount += toInt(parsedValues[3])
        output = key + ", " + (double)tminSum/tminCount + ", " + (double)tmaxSum/tmaxCount
        context.write(output, null)
```

Secondary Sort

```
# CompositeKey{stationId, year} implements WritableComparable offering a constructor along
with write, readFields & compareTo with respective getter & setters.
```


The entire composite key is used for sorting.
Composite key
StationId
Year
Composite value
Natural Key
Year
"Natural key" used for partitioning and grouping.
"Secondary key" only used during sorting.
Provides reduce-side access to secondary key.

The user composite key and value

```
class CompositeKey:
        String stationId
        Integer year
        def Composite(String stationId, Integer year):
                this.stationId = stationId
                this.year      = year

        def write(DataOutput dataOutput):
           WritableUtils.writeString(dataOutput, stationId)
           dataOutput.writeInt(year)

        def readFields(DataInput dataInput):
           stationId = WritableUtils.readString(dataInput);
           year = dataInput.readInt();

        def compareTo(CompositeKey o) {
           result = stationId.compareTo(o.getStationId());
           if (result == 0):
               result = year.compareTo(o.getYear());
           return result

        def getStationId():
           return stationId
        def getYear():
           return year
```

```
def mapper(Object key, Text value, Context context):
        parsedValues = value.split(",")
        id           = parsedValues[0]
        year         = parsedValues[1].substring(0,4)
        element      = parsedValues[2]
        temp         = parsedValues[3]
        if(element == "TMAX" || element == "TMIN"):
                context.write(CompositeKey(id, year), element + "," + temp)

def compositeKeyComparator(CompositeKey k1, CompositeKey k2):
        return k1.compareTo(k2)
def naturalKeyGroupingComparator(CompositeKey k1, CompositeKey k2):
        return k1.getStationId().compareTo(k2.getStationId())
def naturalKeyPartitioner(CompositeKey k):
        return Math.abs(k.getStationId().hashCode() % numPartitions)

def reduce(CompositeKey key, Text[] values, Context context):
        tmaxSum, tmaxCount = 0
        tminSum, tminCount = 0
        keyYear     = key.getYear()
        output            = key.getStationId() + ", ["
        for val in values:
                parsedValues = val.split(",")
                temp         = toInt(parsedValues[1])
                if keyYear != key.getYear():
                        output += "(" + keyYear                      + ", " +
                                          (double)tminSum/tminCount + ", " +
                                          (double)tmaxSum/tmaxCount + "),"
                        tmaxSum = tmaxCount = tminSum = tminCount = 0
                        keyYear = key.getYear()
                if parsedValues[0] == "TMAX":
                        tmaxSum  += temp
                        tmaxCount++
                else:
                        tminSum  += temp
                        tminCount++
        output += "(" + keyYear + ", " + (double)tminSum/tminCount + ", " +
                                            (double)tmaxSum/tmaxCount + ")]"
        context.write(output, null)
```

INPUT

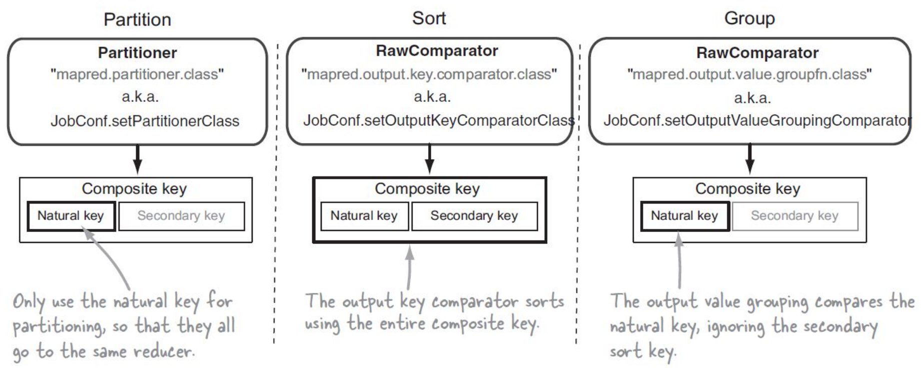
```
AGE00135039,18800403,TMAX,65,,,E,          FSG00285971,18800403,TMAX,65,,,E,
AGE00135039,18800403,TMIN,11,,,E,          FSG00285971,18830403,TMIN,32,,,E,
FSG00285971,18810403,TMIN,32,,,E,          FSG00285971,18800403,TMIN,11,,,E,
FSG00285971,18820403,TMAX,41,,,E,          FSG00285971,18810403,TMAX,41,,,E,
AGE00135039,18810403,TMAX,41,,,E,          FSG00285971,18820403,TMIN,32,,,E,
AGE00135039,18810403,TMIN,32,,,E,          FSG00285971,18830403,TMAX,41,,,E,
```

Custom Partitioner : To make sure each reduce task gets one key

Key Comparator : To make sure all values are sorted by ear also in CompositeKey

Grouping Comparator: To make sure all values for a particular Station Id grouped together

The partitioning happens in the Map Phase, data from various Map tasks are received by reducers where they are ***grouped*** *and then sent to the reduce method.*


Partition
Sort
Group
Partitioner
"mapred.partitioner.class"
a.k.a.
JobConf.setPartitionerClass
RawComparator
"mapred.output.key.comparator.class"
a.k.a.
JobConf.setOutputKeyComparatorClass
RawComparator
"mapred.output.value.groupfn.class"
a.k.a.
JobConf.setOutputValueGroupingComparator
Composite key
Natural key
Secondary key
Only use the natural key for partitioning, so that they all go to the same reducer.
The output key comparator sorts using the entire composite key.
The output value grouping compares the natural key, ignoring the secondary sort key.

Partitioning, sorting, and grouping settings and key utilization

Reduce 1	**Stack Variables**
(AGE00135039, 1880) -> (TMAX, 65)	mxSm: 65, mxCnt: 1, mnSm: 0, mnCnt: 0
(AGE00135039, 1880) -> (TMIN, 11)	mxSm: 65, mxCnt: 1, mnSm: 11, mnCnt: 1
(AGE00135039, 1881) -> (TMAX, 41)	mxSm: 41, mxCnt: 1, mnSm: 0, mnCnt: 0
(AGE00135039, 1881) -> (TMIN, 32)	mxSm: 41, mxCnt: 1, mnSm: 32, mnCnt: 1

Reduce 2	**Variables reset to ZERO on Year change**
(FSG00285971, 1880) -> (TMAX, 65)	mxSm: 65, mxCnt: 1, mnSm: 0, mnCnt: 0
(FSG00285971, 1880) -> (TMIN, 11)	mxSm: 65, mxCnt: 1, mnSm: 11, mnCnt: 1
(FSG00285971, 1880) -> (TMAX, 10)	mxSm: 75, mxCnt: 2, mnSm: 0, mnCnt: 0
(FSG00285971, 1880) -> (TMAX, 05)	mxSm: 80, mxCnt: 3, mnSm: 32, mnCnt: 1
(FSG00285971, 1881) -> (TMIN, 41)	mxSm: 00, mxCnt: 0, mnSm: 41, mnCnt: 1
(FSG00285971, 1881) -> (TMIN, 32)	mxSm: 00, mxCnt: 0, mnSm: 73, mnCnt: 2
(FSG00285971, 1882) -> (TMAX, 41)	mxSm: 41, mxCnt: 1, mnSm: 0, mnCnt: 0
(FSG00285971, 1882) -> (TMIN, 32)	mxSm: 41, mxCnt: 1, mnSm: 32, mnCnt: 1

Performance Comparison

Running Time Comparison Table 1

Name	Job 1 (ms)	Job 2 (ms)	Gain (%)
NoCombiner	271650	279850	Baseline
WithCombiner	267280	268110	1.91
InMapperCombiner	217640	217810	19.8

MapReduce Stats Comparison Table 2

Stats Fields	No Combiner	Combiner	In Mapper Combiner
Map Input Records	30868726	30868726	30868726
Map Output Records	8798241	8798241	223783
Map Output Bytes	182692275	182692275	6175140
Combine Input	0	8798241	0
Combine Output	0	223783	0
Reduce Input Groups	14135	14135	14135
Reduce Shuffle Bytes	50886128	5346458	4049305
Reduce Input Records	8798241	223783	223783
Reduce Output Records	14135	14135	14135
Spilled Records	17596482	447566	447566

Q1. Was the Combiner called at all in program Combiner? Was it called more than once per Map task ?

A1. Yes Combiner was called in my Combiner program as evident from table 2, the Combine input and output records for no Combiner are zero whereas those of Combiner are input: **8798241**and output: **223783** bytes which is substantial. The MapReduce

Framework decides how many times if at all to run a combiner so we cannot quantify the amount of time combiner was called more than once per map task.

Q2. What difference did the use of a Combiner make in Combiner compared to No Combiner ?
A2. Using combiner reduced the amount of intermediate data that gets transferred in the shuffle and sort phase by aggregating the values belonging to the same key. As we can see from table 2, Reduce Shuffle Bytes reduced by **10.5x** and Reduce input and spilled records by almost **40x**.

Q3. Was the local aggregation effective in InMapperComb compared to No Combiner ?
A3. Yes, it improved the running time by **~20%** which is alot although it comes it a tradeoff which bounds the mapper to maximum amount of memory that can be allocated to the shared associative array data structure. As evident from table 2, Map Output Bytes reduced by **30x**, Reduce Output Records by **12.5x** and reducing Map Output Records, Reduce Input and spilled records by **40x** which by any scale is on par with no combiner bu as I stated above this technique should be applied with due diligence since the global Hash Map used by mapper is limited by the amount of heap size.

Q4. How do the running times and accuracy of these MapReduce programs compare to the sequential implementation of the programs ?
A4. The reason behind Sequential version being so fast is the fact that there are almost no bootstrapping cost of starting up the cluster, splitting data, etc which account for high running times of MapReduce program run in pseudo distributed mode but as we saw in the MultiThreaded versions in HW1 they instantly gained at least 2x speedup which corroborates to the argument that when this program runs of file sizes of very large size, the real performance of MR framework would shine with replication, failure recovery, parallel execution.
I wrote a small python script to compute a accuracy and all of the MR version had 100%

Sequential Vs MapReduce Local Execution Table 3

Name	Running Time (ms)	Accuracy (%)
Sequential	39730	Baseline

NoCombiner	112650	100
WithCombiner	103220	100
InMapperCombiner	87380	100