|  |  |
| --- | --- |
|  | I want to calculate means and standard deviation by columns in Hadoop.  I simple adopt single pass Naïve algorithm to MapReduce. I tested it on multivariate data sets 455000x90 and 650000x120 and got speedup lower, more lower, then count of processors. For standalone and pseudo-distributed mode with 2 active cores I got speedup 0,4 = 20seconds / 53seconds for 455000x90.  Why my programm is not effective ? Is it possible to improve it ?  Mapper: |

public class CalculateMeanAndSTDEVMapper extends

Mapper <LongWritable,

DoubleArrayWritable,

IntWritable,

DoubleArrayWritable> {

private int dataDimFrom;

private int dataDimTo;

private long samplesCount;

private int universeSize;

@Override

protected void setup(Context context) throws IOException {

Configuration conf = context.getConfiguration();

dataDimFrom = conf.getInt("dataDimFrom", 0);

dataDimTo = conf.getInt("dataDimTo", 0);

samplesCount = conf.getLong("samplesCount", 0);

universeSize = dataDimTo - dataDimFrom + 1;

}

@Override

public void map(

LongWritable key,

DoubleArrayWritable array,

Context context) throws IOException, InterruptedException {

DoubleWritable[] outArray = new DoubleWritable[universeSize\*2];

for (int c = 0; c < universeSize; c++) {

outArray[c] = new DoubleWritable(

array.get(c+dataDimFrom).get() / samplesCount);

}

for (int c = universeSize; c < universeSize\*2; c++) {

double val = array.get(c-universeSize+dataDimFrom).get();

outArray[c] = new DoubleWritable((val\*val) / samplesCount);

}

context.write(new IntWritable(1), new DoubleArrayWritable(outArray));

}

}

Combiner:

public class CalculateMeanAndSTDEVCombiner extends

Reducer <IntWritable,

DoubleArrayWritable,

IntWritable,

DoubleArrayWritable> {

private int dataDimFrom;

private int dataDimTo;

private int universeSize;

@Override

protected void setup(Context context) throws IOException {

Configuration conf = context.getConfiguration();

dataDimFrom = conf.getInt("dataDimFrom", 0);

dataDimTo = conf.getInt("dataDimTo", 0);

universeSize = dataDimTo - dataDimFrom + 1;

}

@Override

public void reduce(

IntWritable column,

Iterable<DoubleArrayWritable> partialSums,

Context context) throws IOException, InterruptedException {

DoubleWritable[] outArray = new DoubleWritable[universeSize\*2];

boolean isFirst = true;

for (DoubleArrayWritable partialSum : partialSums) {

for (int i = 0; i < universeSize\*2; i++) {

if (!isFirst) {

outArray[i].set(outArray[i].get()

+ partialSum.get(i).get());

} else {

outArray[i]

= new DoubleWritable(partialSum.get(i).get());

}

}

isFirst = false;

}

context.write(column, new DoubleArrayWritable(outArray));

}

}

Reducer:

public class CalculateMeanAndSTDEVReducer extends

Reducer <IntWritable,

DoubleArrayWritable,

IntWritable,

DoubleArrayWritable> {

private int dataDimFrom;

private int dataDimTo;

private int universeSize;

@Override

protected void setup(Context context) throws IOException {

Configuration conf = context.getConfiguration();

dataDimFrom = conf.getInt("dataDimFrom", 0);

dataDimTo = conf.getInt("dataDimTo", 0);

universeSize = dataDimTo - dataDimFrom + 1;

}

@Override

public void reduce(

IntWritable column,

Iterable<DoubleArrayWritable> partialSums,

Context context) throws IOException, InterruptedException {

DoubleWritable[] outArray = new DoubleWritable[universeSize\*2];

boolean isFirst = true;

for (DoubleArrayWritable partialSum : partialSums) {

for (int i = 0; i < universeSize; i++) {

if (!isFirst) {

outArray[i].set(outArray[i].get() + partialSum.get(i).get());

} else {

outArray[i] = new DoubleWritable(partialSum.get(i).get());

}

}

isFirst = false;

}

for (int i = universeSize; i < universeSize \* 2; i++) {

double mean = outArray[i-universeSize].get();

outArray[i].set(Math.sqrt(outArray[i].get() - mean\*mean));

}

context.write(column, new DoubleArrayWritable(outArray));

}

}

Where DoubleArrayWritable is simple class which extends ArrayWritable:

public class DoubleArrayWritable extends ArrayWritable {

public DoubleArrayWritable() {

super(DoubleWritable.class);

}

public DoubleArrayWritable(DoubleWritable[] values) {

super(DoubleWritable.class, values);

}

public DoubleWritable get(int idx) {

return (DoubleWritable) get()[idx];

}

}