Towards Automatic Optimization of MapReduce Programs
(Position Paper)

Shivnath Babu
Duke University
Roadmap

• Call to action to improve automatic optimization techniques in MapReduce frameworks
• Challenges & promising directions
Lifecycle of a MapReduce Job

```java
public class WordCount {
    public static class Map extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
            String line = value.toString();
            StringTokenizer tokenizer = new StringTokenizer(line);
            while (tokenizer.hasMoreTokens()) {
                word.set(tokenizer.nextToken());
                output.collect(word, one);
            }
        }
    }

    public static class Reduce extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {

        public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
            int sum = 0;
            while (values.hasNext()) { sum += values.next().get(); }
            output.collect(key, new IntWritable(sum));
        }
    }

    public static void main(String[] args) throws Exception {
        JobConf conf = new JobConf(WordCount.class);
        conf.setJobName("wordcount");
        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(IntWritable.class);
        conf.setMapperClass(Map.class);
        conf.setCombinerClass(Reduce.class);
        conf.setReducerClass(Reduce.class);
        conf.setInputFormat(TextInputFormat.class);
        conf.setOutputFormat(TextOutputFormat.class);
        TextInputFormat.setInputPaths(conf, new Path(args[0]));
        TextOutputFormat.setOutputPath(conf, new Path(args[1]));
        JobClient.runJob(conf);
    }
}
```

Map function

Reduce function

Run this program as a MapReduce job
Lifecycle of a MapReduce Job

public class WordCount {
    public static class Map extends MapReduceBase implements 
        Mapper<LongWritable, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();
        public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
            String line = value.toString();
            StringTokenizer tokenizer = new StringTokenizer(line);
            while (tokenizer.hasMoreTokens()) {
                word.set(tokenizer.nextToken());
                output.collect(word, one);
            }
        }
    }

    public static class Reduce extends MapReduceBase implements 
        Reducer<Text, IntWritable, Text, IntWritable> {
        public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
            int sum = 0;
            while (values.hasNext()) { sum += values.next().get(); }
            output.collect(key, new IntWritable(sum));
        }
    }

    public static void main(String[] args) throws Exception {
        JobConf conf = new JobConf(WordCount.class);
        conf.setJobName("wordcount");
        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(IntWritable.class);
        conf.setMapperClass(Map.class);
        conf.setCombinerClass(Reduce.class);
        conf.setReducerClass(Reduce.class);
        conf.setInputFormat(TextInputFormat.class);
        conf.setOutputFormat(TextOutputFormat.class);
        TextInputFormat.setInputPaths(conf, new Path(args[0]));
        TextOutputFormat.setOutputPath(conf, new Path(args[1]));
        JobClient.runJob(conf);
    }
}
How are the number of splits, number of map and reduce tasks, memory allocation to tasks, etc., determined?
Job Configuration Parameters

- 190+ parameters in Hadoop
- Set manually or defaults are used
- Are defaults or rules-of-thumb good enough?
Experiments

On EC2 and local clusters

Running time (seconds)

Running time (minutes)

75GB TeraSort in Hadoop

50GB TeraSort in Hadoop

mapred.reduce.tasks

io.sort.factor

Running Time

mapred.reduce.tasks

Running Time (seconds)

io.sort.factor

Running Time (minutes)

mapred.reduce.tasks

io.sort.factor

Running Time (seconds)

mapred.reduce.tasks

Running Time (minutes)

mapred.reduce.tasks

io.sort.factor

io.sort.factor

Running Time (minutes)

mapred.reduce.tasks

io.sort.factor

Running Time (seconds)

mapred.reduce.tasks
**Illustrative Result: 50GB Terasort**

17-node cluster, 64+32 concurrent map+reduce slots

<table>
<thead>
<tr>
<th>mapred.reduce. tasks</th>
<th>io.sort. factor</th>
<th>io.sort.record. percent</th>
<th>Running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>0.15</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>500</td>
<td>0.15</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>10</td>
<td>0.15</td>
<td></td>
</tr>
<tr>
<td>300</td>
<td>10</td>
<td>0.15</td>
<td></td>
</tr>
<tr>
<td>300</td>
<td>500</td>
<td>0.15</td>
<td></td>
</tr>
</tbody>
</table>

Based on popular rule-of-thumb

- Performance at default and rule-of-thumb settings can be poor
- Cross-parameter interactions are significant
Current approaches:
- Predominantly manual
- Post-mortem analysis

Is this where we want to be?
Can DB Query Optimization Technology Help?

But:

- MapReduce jobs are not declarative
- No schema about the data
- Impact of concurrent jobs & scheduling?
- Space of parameters is huge

Can we:

- Borrow/adapt ideas from the wide spectrum of query optimizers that have been developed over the years
  - Or innovate!
- Exploit design & usage properties of MapReduce frameworks
Spectrum of Query Optimizers

Conventional Optimizers

Cost models + statistics about data
Rule-based

AT’s Conjecture: Rule-based Optimizers (RBOs) will trump Cost-based Optimizers (CBOs) in MapReduce frameworks

Insight: Predictability(RBO) >> Predictability(CBO)
AT’s Conjecture: Rule-based Optimizers (RBOs) will trump Cost-based Optimizers (CBOs) in MapReduce frameworks

Insight: Predictability(RBO) >> Predictability(CBO)
Exploit usage & design properties of MapReduce frameworks:

- High ratio of repeated jobs to new jobs
- Schema can be learned (e.g., Pig scripts)
- Common sort-partition-merge skeleton
- Mechanisms for adaptation stemming from design for robustness (speculative execution, storing intermediate results)
- Fine-grained and pluggable scheduler
Summary

• Call to action to improve automatic optimization techniques in MapReduce frameworks
  – Automated generation of optimized Hadoop configuration parameter settings, HiveQL/Pig/JAQL query plans, etc.
  – Rich history to learn from
  – MapReduce execution creates unique opportunities/challenges