Secondary sorting - Sorting Values in Hadoop's MR

- Sometimes, we would like to sort the values coming into the Reducer of a Hadoop Map/Reduce (MR) Job.

- You can indirectly sort the values by using a combination of implementations. They are as follows.
  - Use a composite key.
  - Extend org.apache.hadoop.mapreduce.Partitioner.
  - Extend org.apache.hadoop.io.WritableComparator.

The Problem

- Imagine we have stock data that looks like the following.
- Each line represents the value of a stock at a particular time.
- Each value in a line is delimited by a comma.
- The first value is the stock symbol (i.e. GOOG), the second value is the timestamp (i.e. the number of milliseconds since January 1, 1970, 00:00:00 GMT), and the third value is the stock's price.

- The data below is a toy data set.
- As you can see, there are 3 stock symbols: a, b, and c.
- The timestamps are also simple: 1, 2, 3, 4.
- The values are fake as well: 1.0, 2.0, 3.0, and 4.0.

```
a, 1, 1.0
b, 1, 1.0
c, 1, 1.0
a, 2, 2.0
b, 2, 2.0
c, 2, 2.0
a, 3, 3.0
b, 3, 3.0
c, 3, 3.0
a, 4, 4.0
b, 4, 4.0
c, 4, 4.0
```

Secondary Sorting

- Let's say we want for each stock symbol (the reducer key input, or alternatively, the mapper key output), to order the values descendingly by timestamp when they come into the reducer.
- How do we sort the timestamp descendingly?
- This problem is known as secondary sorting.
- Hadoop's M/R platform sorts the keys, but not the values. (Note, Google's M/R platform explicitly supports secondary sorting).

1. Use a Composite Key

- A solution for secondary sorting involves doing multiple things.
- First, instead of simply emitting the stock symbol as the key from the mapper, we need to emit a composite key, a key that has multiple parts.
- The key will have the stock symbol and timestamp.
- If you remember, the process for a M/R Job is as follows.
  - (K1,V1) → Map → (K2,V2)
  - (K2,List[V2]) → Reduce → (K3,V3)

Composite Key Cont'd

- In the toy data above, K1 will be of type LongWritable, and V1 will be of type Text.
- Without secondary sorting, K2 will be of type Text and V2 will be of type DoubleWritable (we simply emit the stock symbol and price from the mapper to the reducer).
- So, K2=symbol, and V2=price, or (K2,V2) = (symbol,price).
- However, if we emit such an intermediary key-value pair, secondary sorting is not possible.
- We have to emit a composite key, K2={symbol,timestamp}.
- So the intermediary key-value pair is (K2,V2) = ({symbol,timestamp},price).
- Note that composite data structures, such as the composite key, is held within the curly braces.
- Our reducer simply outputs a K3 of type Text and V3 of type Text; (K3,V3) = (symbol, price).
- The complete M/R job with the new composite key is shown below.

- (LongWritable,Text) –> Map –> ({symbol,timestamp},price)
- ({symbol,timestamp},List[price]) –> Reduce –> (symbol,price)

- K2 is a composite key, but inside it, the symbol part/component is referred to as the "natural" key. It is the key which values will be grouped by.

Composite Key Comparator

- The composite key comparator is where the secondary sorting takes place.
- It compares composite key by symbol ascendingly and timestamp descendingly.
- Notice here we sort based on symbol and timestamp.
- All the components of the composite key is considered.

```
public class CompositeKeyComparator extends WritableComparator {
    protected CompositeKeyComparator() {
        super(StockKey.class, true);
    }
    @Override
    public int compare(WritableComparable w1, WritableComparable w2) {
        StockKey k1 = (StockKey)w1;
        StockKey k2 = (StockKey)w2;
        int result = k1.getSymbol().compareTo(k2.getSymbol());
        if(0 == result) {
            result = -1* k1.getTimestamp().compareTo(k2.getTimestamp());
        }
        return result;
    }
}
```

Natural Key Grouping Comparator

- The natural key group comparator “groups” values together according to the natural key.
- Without this component, each K2={symbol,timestamp} and its associated V2=price may go to different reducers.
- Notice here, we only consider the “natural” key.

```
public class NaturalKeyGroupingComparator extends WritableComparator {
    protected NaturalKeyGroupingComparator() {
        super(StockKey.class, true);
    }
    @SuppressWarnings("rawtypes")
    @Override
    public int compare(WritableComparable w1, WritableComparable w2) {
        StockKey k1 = (StockKey)w1;
        StockKey k2 = (StockKey)w2;

        return k1.getSymbol().compareTo(k2.getSymbol());
    }
}
```

Natural Key Partitioner

- The natural key partitioner uses the natural key to partition the data to the reducer(s).
- Again, note that here, we only consider the “natural” key.

```
public class NaturalKeyPartitioner extends Partitioner<StockKey, DoubleWritable>
{
    @Override
    public int getPartition(StockKey key, DoubleWritable val, int numPartitions) {
        int hash = key.getSymbol().hashCode();
        int partition = hash % numPartitions;
        return partition;
    }

}
```

The M/R Job

- Once we define the Mapper, Reducer, natural key grouping comparator, natural key partitioner, composite key comparator, and composite key, in Hadoop's new M/R API, we may configure the Job as follows.

```
public class SsJob extends Configured implements Tool {
    public static void main(String[] args) throws Exception {
        ToolRunner.run(new Configuration(), new SsJob(), args);
    }
    @Override
    public int run(String[] args) throws Exception {
        Configuration conf = getConf();
        Job job = new Job(conf, "secondary sort");

        job.setJarByClass(SsJob.class);
        job.setPartitionerClass(NaturalKeyPartitioner.class);
        job.setGroupingComparatorClass(NaturalKeyGroupingComparator.class);
        job.setSortComparatorClass(CompositeKeyComparator.class);

        job.setMapOutputKeyClass(StockKey.class);
        job.setMapOutputValueClass(DoubleWritable.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);

        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);

        job.setMapperClass(SsMapper.class);
        job.setReducerClass(SsReducer.class);

        job.waitForCompletion(true);

        return 0;
    }
}
```

Chaining MapReduce jobs

You've been doing data processing tasks which a single MapReduce job can accomplish. As you get more comfortable writing MapReduce programs and take on more ambitious data processing tasks, you'll find that many complex tasks need to be broken down into simpler subtasks, each accomplished by an individual MapReduce job. For example, from the citation data set you may be interested in finding the ten most-cited patents. A sequence of two MapReduce jobs can do this. The first one creates the "inverted" citation data set and counts the number of citations for each patent, and the second job finds the top ten in that "inverted" data.

Chaining MapReduce jobs in a sequence

Though you can execute the two jobs manually one after the other, it's more convenient to automate the execution sequence. You can chain MapReduce jobs to run sequentially, with the output of one MapReduce job being the input to the next. Chaining MapReduce jobs is analogous to Unix pipes.

```
mapreduce-1 | mapreduce-2 | mapreduce-3 | ...
```

Chaining MapReduce jobs sequentially is quite straightforward. Recall that a driver sets up a `JobConf` object with the configuration parameters for a MapReduce job and passes the `JobConf` object to `JobClient.runJob()` to start the job. As `JobClient.runJob()` blocks until the end of a job, chaining MapReduce jobs involves calling the driver of one MapReduce job after another. The driver at each job will have to create a new `JobConf` object and set its input path to be the output path of the previous job. You can delete the intermediate data generated at each step of the chain at the end.

Chaining MapReduce jobs with complex dependency

Sometimes the subtasks of a complex data processing task don't run sequentially, and their MapReduce jobs are therefore not chained in a linear fashion. For example, mapreduce1 may process one data set while mapreduce2 independently processes another data set. The third job, mapreduce3, performs an inner join of the first two jobs' output. (We'll discuss data joining in the next sections.) It's dependent on the other two and can execute only after both mapreduce1 and mapreduce2 are completed. But mapreduce1 and mapreduce2 aren't dependent on each other.

Hadoop has a mechanism to simplify the management of such (nonlinear) job dependencies via the `Job` and `JobControl` classes. A `Job` object is a representation of a MapReduce job. You instantiate a `Job` object by passing a `JobConf` object to its constructor. In addition to holding job configuration information, `Job` also holds dependency information, specified through the `addDependingJob()` method. For `Job` objects `x` and `y`,

```
x.addDependingJob(y)
```

means `x` will not start until `y` has finished. Whereas `Job` objects store the configuration and dependency information, `JobControl` objects do the managing and monitoring of the job execution. You can add jobs to a `JobControl` object via the `addJob()` method. After adding all the jobs and dependencies, call `JobControl`'s `run()` method to spawn a thread to submit and monitor jobs for execution. `JobControl` has methods like `allFinished()` and `getFailedJobs()` to track the execution of various jobs within the batch.

Chaining preprocessing and postprocessing steps

A lot of data processing tasks involve record-oriented preprocessing and postprocessing. For example, in processing documents for information retrieval, you may have one step to remove *stop words* (words like *a*, *the*, and *is* that occur frequently but aren't too meaningful), and another step for *stemming* (converting different forms of a word into the same form, such as *finishing* and *finished* into *finish*.) You can write a separate MapReduce job for each of these pre- and postprocessing steps and chain them together, using IdentityReducer (or no reducer at all) for these steps. This approach is inefficient as each step in the chain takes up I/O and storage to process the intermediate results. Another approach is for you to write your mapper such that it calls all the preprocessing steps beforehand and the reducer to call all the postprocessing steps afterward. This forces you to architect the pre- and postprocessing steps in a modular and composable manner. Hadoop introduced the `ChainMapper` and the `ChainReducer` classes in version 0.19.0 to simplify the composition of pre- and postprocessing.

You can think of chaining MapReduce jobs, as explained in section 5.1.1, symbolically using the pseudo-regular expression:

```
[MAP | REDUCE]+
```

where a reducer `REDUCE` comes after a mapper `MAP`, and this `[MAP | REDUCE]` sequence can repeat itself one or more times, one right after another. The analogous expression for a job using `ChainMapper` and `ChainReducer` would be

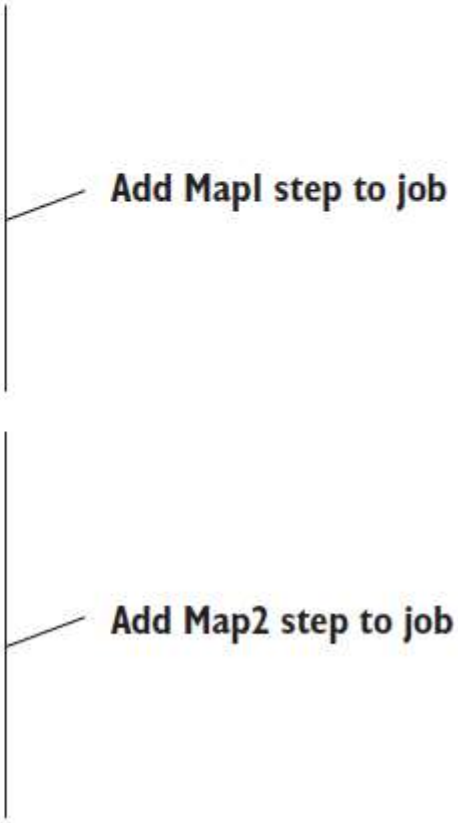
```
MAP+ | REDUCE | MAP*
```

The job runs multiple mappers in sequence to preprocess the data, and after running reduce it can optionally run multiple mappers in sequence to postprocess the data. The beauty of this mechanism is that you write the pre- and postprocessing steps as standard mappers. You can run each one of them individually if you want. (This is useful when you want to debug them individually.) You call the `addMapper()` method in `ChainMapper` and `ChainReducer` to compose the pre- and postprocessing steps, respectively. Running all the pre- and postprocessing steps in a single job leaves no intermediate file and there's a dramatic reduction in I/O.

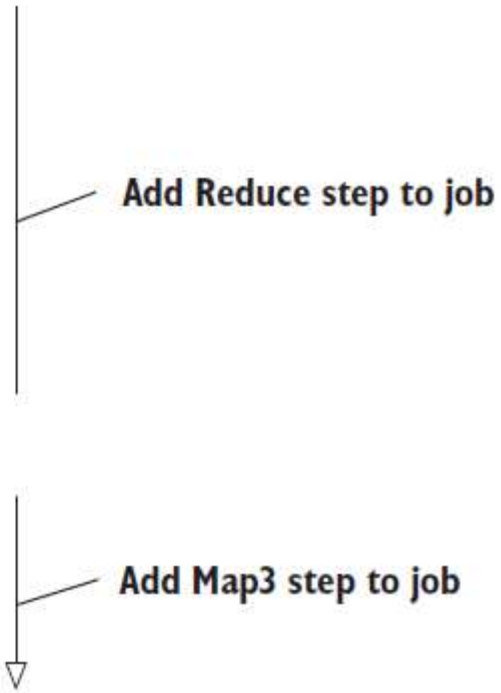
Consider the example where there are four mappers (`Map1`, `Map2`, `Map3`, and `Map4`) and one reducer (`Reduce`), and they're chained into a single MapReduce job in this sequence:

```
Map1 | Map2 | Reduce | Map3 | Map4
```

```
Configuration conf = getConf();
JobConf job = new JobConf(conf);

job.setJobName("ChainJob");
job.setInputFormat(TextInputFormat.class);
job.setOutputFormat(TextOutputFormat.class);

FileInputFormat.setInputPaths(job, in);
FileOutputFormat.setOutputPath(job, out);

JobConf map1Conf = new JobConf(false);
ChainMapper.addMapper(job,
                      Map1.class,
                      LongWritable.class,
                      Text.class,
                      Text.class,
                      Text.class,
                      true,
                      map1Conf);

JobConf map2Conf = new JobConf(false);
ChainMapper.addMapper(job,
                      Map2.class,
                      Text.class,
                      Text.class,
                      LongWritable.class,
                      Text.class,
                      true,
                      map2Conf);
```



```
JobConf reduceConf = new JobConf(false);
ChainReducer.setReducer(job,
                        Reduce.class,
                        LongWritable.class,
                        Text.class,
                        Text.class,
                        Text.class,
                        true,
                        reduceConf);

JobConf map3Conf = new JobConf(false);
ChainReducer.addMapper(job,
                       Map3.class,
                       Text.class,
                       Text.class,
                       LongWritable.class,
```



```
                        Text.class,
                        true,
                        map3Conf);

JobConf map4Conf = new JobConf(false);
ChainReducer.addMapper(job,
                        Map4.class,
                        LongWritable.class,
                        Text.class,
                        LongWritable.class,
                        Text.class,
                        true,
                        map4Conf);

JobClient.runJob(job);
```