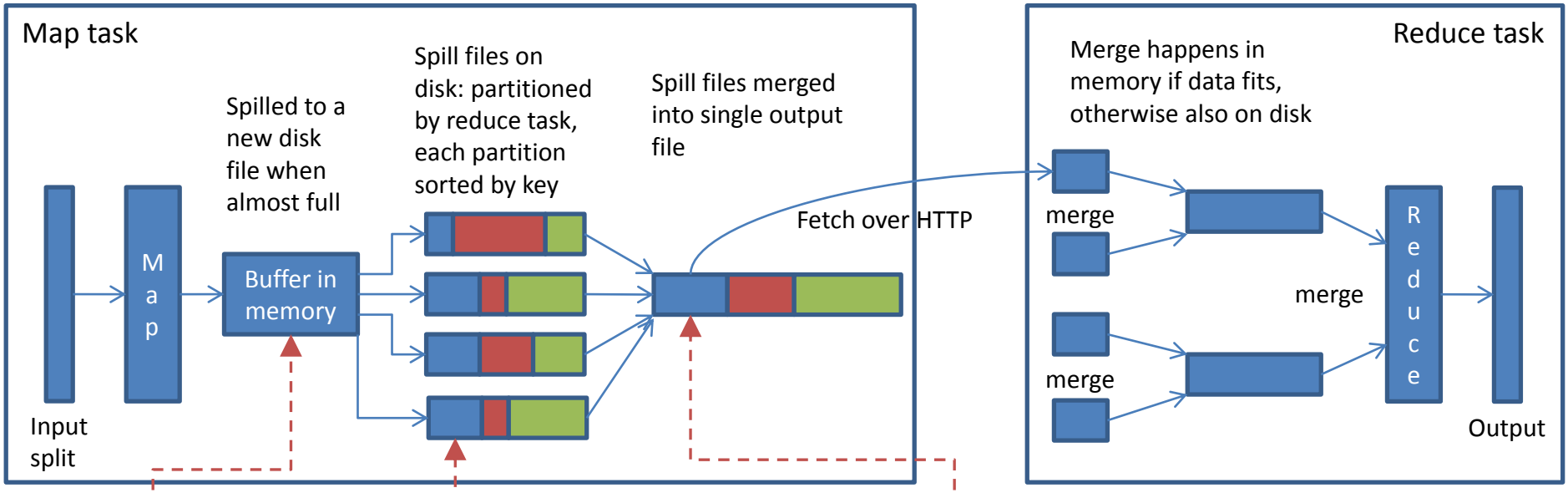
1. Data Aggregation in the Mapper

- We discussed that the Map function can only work on a single input record, hence cannot be used to combine information across different records.
- However, as we will discuss in this section, there are ways to combine across different records at the level of a Map *task*.
  - Recall that typically there are many Map function calls per Map task.

1.1 Combiner

- We introduced Combiners in Module 2. They are provided as an optional optimization for MapReduce programs, enabling Reducer-style functionality on the Mappers in order to decrease the cost of the shuffle-and-sort and the reduce phase.
  - Recall the Word Count example where a Map task (i.e., a Mapper) processes two different lines L1 and L2 of a document. L1 contains 3 occurrences of the word “Northeastern”, while L2 contains another 5 occurrences. A Combiner can replace the 8 instances of (Northeastern, 1) by a single record (Northeastern, 8).
- While Combiners are an elegant way of adding data reduction functionality to a Map task, they cannot be controlled by the user. In particular, the MapReduce system decides when and on which Map output records the Combiner is executed. It might not be executed at all. Or it might be executed only on some subset of the output records, e.g., only the records currently in memory. And it might be executed multiple times, possibly reading records output by an earlier Combiner execution together with “fresh”, recently emitted records. For an illustration, follow this LINK.

Combiner Execution Options



The Combiner could be executed on some or all records in any of these locations, at any time, as often as desired.

Illustration based on White's book

1.2 Local Aggregation

- Can we do something like a Combiner, but have more control over the process?
- Consider again Word Count.

```
map(offset B, line L )
  for each word w in L do
    emit(w, 1)
```

```
reduce(word w, [c1, c2,...])
  total = 0
  for each c in input list do
    total += c
  emit(w, total)
```

1.2.1 Tally Counts Per Line

```
map(offset B, line L )
  h = new hashMap                  // stores count for each word in line L
  for each word w in L do
    h[w]++
  for each word w in h do
    emit(w, h[w])
```

- Our first attempt combines counts inside a single Map function call. To do so, we need a data structure in Map that tracks the word occurrences in the input line. (See code above.) The Reduce function does not change.
- While simple, this modification will typically not be very effective, because it combines the counts only within a single line of text.
- To increase combining opportunities, we would like to aggregate counts across the entire Map input split, not just a single line.

1.2.2 Tally Counts Per Map Task

- To aggregate counts across the entire Map task input, we have to work at the task level, i.e., above the level of individual Map calls. More precisely, we need a data structure H that is a private member of the Mapper class. This data structure can be updated by each Map call in the same task.
  - Notice that H is local to a single task, i.e., it does not introduce any task synchronization issues.
- Data structure H needs to be initialized before the first Map function call in the task. This is done inside the Map task's setup() function, which is guaranteed to be executed when the task starts, before any of the Map calls.
  - This function was called configure() in the old API.
- Each Map function call then updates the counts for the words it finds in its input line. However, Map does not emit any output any more!
- To emit the final tally for the entire task, the counts in H have to be emitted after the *last* Map call in the task has completed. This is achieved by "emptying out" the contents of H in the cleanup() function of the Mapper class. Cleanup() is guaranteed to be executed after the last Map call.
  - This function was called close() in the old API.
- The Reducer remains unchanged.

```
Class Mapper {
  hashMap h

  setup() {
    H = new hashMap
  }

  map(offset B, line L ) {
    for each word w in L do
      H[w]++
  }

  cleanup() {
    for each word w in H do
      emit(w, H[w])
  }
}
```

1.3 Summary of the Design Pattern for Local Aggregation

- The tally-per-task version of Word Count is an example for the in-mapper combining design pattern. The main idea of this pattern is to preserve state at the task level, across different Map calls in the same task. The same pattern can also be applied to Reduce tasks.
- Advantages over using Combiners:
  - The Combiner does not guarantee if, when, or how often it will be executed.
  - A Combiner combines data *after* it was generated. In-mapper combining avoids generating large amounts of intermediate data by immediately aggregating it as it is produced by a Map call. This often results in significant reduction of local CPU and disk I/O cost on the Mappers.
- Drawbacks compared to Combiners:
  - In-mapper combining code needs to be integrated with the Mapper code, increasing code complexity and hence the probability for introducing errors.
  - It needs more memory for managing state, e.g., to hold the hashMap H in memory. If the data structure used for in-mapper combining exceeds the amount of available memory, the programmer has to write non-trivial memory-management code to page it to disk.

2. Distributed Sorting

- Sorting is one of the most commonly used tasks for data processing.
- Let's see how we can sort Big Data using MapReduce.
- Somewhat surprisingly, it will turn out that we can let MapReduce do the sorting for us. The sort program uses trivial "identity" Mappers and Reducers.

2.1 First (Unsuccessful) Attempt

- Classic parallel sort idea: Each worker locally sorts a data chunk. Then the sorted chunks (usually called “runs”) are merged, often in multiple passes.
- Can we use Map for the pre-sorting and Reduce for merging?
  - If a single Map call receives multiple input records, we can sort these records in the Map function. To sort the entire file split assigned to the Map task, we either need a record-reader that passes all records in the file split to a single Map function call. Or we use local aggregation to (i) collect all records in a set data structure and then (ii) sort this set in the cleanup() function.
  - The Mapper then emits the presorted file chunk as (dummy, sortedFileChunk), so that a single Reduce call can perform the merge operation.
- Unfortunately this approach does not scale in the Reduce phase.
- We could use multiple dummy keys $dummy_1$,…, $dummy_k$ to distribute the merge work over up to k Reducers. The drawback of this approach is that we end up with k locally sorted runs, but not a globally sorted result. Hence at least one additional MapReduce program needs to be executed to merge these k runs.
- Is there a better way to get multiple Reducers involved and still perform the sorting in a single MapReduce program?

2.2 Distributed Sorting Through Range-Partitioning

- The merge step of the previous algorithm can be avoided if we partition the data the right way.
- Consider a simple scenario with 2 machines to sort the set {5, 2, 4, 6, 1, 3}. If all small numbers, i.e., set {2, 1, 3}, are assigned to machine 0 and all large numbers, i.e., set {5, 4, 6} to machine 1, then sorting is easy.
  - Machine 0 sorts locally and writes [1, 2, 3] to output0
  - Machine 1 sorts locally and writes [4, 5, 6] to output 1
  - The totally sorted output [1, 2, 3, 4, 5, 6] is obtained by simply "concatenating" output0 with output1. (This is a constant-time operation and requires no merging.)
- The challenge for this approach is to separate the given set into "small" and "large" numbers. This is also referred to as range partitioning. Range partitioning ensures that if records i and k, $i \leq k$, are assigned to a partition, then all records j between them, i.e., $i \leq j \leq k$, will be assigned to that same partition.
- In addition, we also would like the different partitions to be of about the same size, so that the local sort work is evenly distributed.

- [Question box: Where have you seen this idea before?]

2.2.1 Range-Partitioning in MapReduce: "Explicit" Approach

- First run a MapReduce job to find the approximate median of the given data set. Exact median computation in the worst case is as expensive as sorting itself. Fortunately, the median can be estimated reliably through sampling. In Hadoop there are two basic approaches for doing this.
  - Option 1: We can use Hadoop's InputSampler class and select the median from this sample.
    - The documentation for this class is a bit sparse (as of April 2014), but it seems that it samples records uniformly at random from a small number of file splits from the input file. Only few file splits are accessed, because InputSampler runs on the client, i.e., not in parallel. Hence to work well, the selected splits should ideally contain a random subset of the input data. (LINK1: What could go wrong if the selected splits do not contain a random subset of the input file?)
  - Option 2: We write our own uniform random sampling program in MapReduce to sample a "sufficiently large" number of input records. Then we find the median of the sample and use it as an approximation of the true median.
    - This program is very simple and will be discussed in a future lecture.
- Once we know the (approximate) median, we can use the program shown below to sort the input data. Intuitively, all smaller records are processed by the Reduce call for key 0, all larger ones by the one for key 1.
- There is a little more to this program:
  - We also need a Partitioner that assigns the smaller keys to lower task numbers to ensure ordering across tasks. (Otherwise it is not clear which task output files contain the smaller records and which contain the larger ones.)
  - The approach generalizes to larger partition numbers by using more quantiles, not just the median. (In that case, it is best to assign each key to a different task. Otherwise we also need to apply range partitioning when assigning keys to tasks. LINK2: Why?)
- Notice that the Reduce function below assumes that the data partition will fit into memory. If that is not the case, smaller partitions need to be created (by using more quantiles) or a disk-based sort implementation is needed.
- We will discuss next how to completely eliminate the need for any explicit sort calls in Map or Reduce by leveraging MapReduce's built-in sort functionality.

```
map(…, record r )
 if (r <= median)
  emit(0, r)
 else
  emit(1, r)
```

```
reduce(key, [r1, r2,…])
  load the value list into memory and sort it
  for each record r in the sorted list
    emit(r)
```

2.2.2 Range-Partitioning in MapReduce: "Minimalist" Approach

- As before, this approach also relies on (approximate) quantiles to partition the input data.
- It exploits the fact that the MapReduce environment guarantees the following property: For each Reduce task, the assigned set of intermediate keys is processed in key order.
- We can leverage this guarantee for sorting, taking advantage of the fact that MapReduce is already heavily optimized for dealing with big data.
- The program below shows how this is done. Map emits the record itself as the key. Reduce simply outputs the key as many times as it received it. The "magic" of this program lies in the Partitioner. Notice that getPartition() assigns the small keys (i.e., the small input records) to Reduce task 0 and the large ones to Reduce task 1. Since Reduce function calls in a task are processed in key order, Reduce task 0 will output the small records correctly sorted. Similarly, Reduce task 1 will output the sorted large records.
  - Obviously, for correct sorting we have to make sure that the appropriate key comparator is defined. Since Hadoop keys have to implement WritableComparable, they have to have such a compareTo function anyway.
- Notice that both Map and Reduce do not do anything but just emitting the records they receive. In Hadoop we can use the predefined IdentityMapper and IdentityReducer classes to do just that.
- This idea generalizes to arbitrary range partitions, e.g., quantiles. Instead of an explicit if-then-else or case statement in getPartition, Hadoop already offers the TotalOrderPartitioner class to assign key ranges to Reduce tasks.
- LINK: Look at the MapReduce sort code from a recent Hadoop distribution here.

```
map( record r, … )
  emit(r, NULL)
```

```
reduce(key, [NULL, NULL,…])
  for each record r in the input list
    emit( key )
```

```
getPartition( key )
  if (key <= median) return 0
  else return 1
```

```
package org.apache.hadoop.examples;
import java.io.IOException; import java.net.URI; import java.util.*;
import org.apache.hadoop.conf.Configuration; import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.filecache.DistributedCache; import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.BytesWritable; import org.apache.hadoop.io.Writable; import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.mapred.lib.IdentityMapper; import org.apache.hadoop.mapred.lib.IdentityReducer;
import org.apache.hadoop.mapred.lib.InputSampler; import org.apache.hadoop.mapred.lib.TotalOrderPartitioner;
import org.apache.hadoop.util.Tool; import org.apache.hadoop.util.ToolRunner;
/**
 * This is the trivial map/reduce program that does absolutely nothing
 * other than use the framework to fragment and sort the input values.
 *
 * To run: bin/hadoop jar build/hadoop-examples.jar sort
 *          [-m <i>maps</i>] [-r <i>reduces</i>]
 *          [-inFormat <i>input format class</i>]
 *          [-outFormat <i>output format class</i>]
 *          [-outKey <i>output key class</i>]
 *          [-outValue <i>output value class</i>]
 *          [-totalOrder <i>pcnt</i> <i>num samples</i> <i>max splits</i>]
 *          <i>in-dir</i> <i>out-dir</i>
 */
public class Sort<K,V> extends Configured implements Tool {
  private RunningJob jobResult = null;

  static int printUsage() {
    System.out.println("sort [-m <maps>] [-r <reduces>] " +
                       "[-inFormat <input format class>] " +
                       "[-outFormat <output format class>] " +
                       "[-outKey <output key class>] " +
                       "[-outValue <output value class>] " +
                       "[-totalOrder <pcnt> <num samples> <max splits>] " +
                       "<input> <output>");
    ToolRunner.printGenericCommandUsage(System.out);
    return -1;
  }
```

Sort Code in Hadoop 1.0.3 Distribution; part 1: boilerplate code

Sort Code in Hadoop 1.0.3 Distribution; part 2: Map and Reduce definition

```
/**
 * The main driver for sort program.
 * Invoke this method to submit the map/reduce job.
 * @throws IOException When there is communication problems with the
 *                     job tracker.
 */
public int run(String[] args) throws Exception {

 JobConf jobConf = new JobConf(getConf(), Sort.class);
 jobConf.setJobName("sorter");

 jobConf.setMapperClass(IdentityMapper.class);
 jobConf.setReducerClass(IdentityReducer.class);

 JobClient client = new JobClient(jobConf);
 ClusterStatus cluster = client.getClusterStatus();
 int num_reduces = (int) (cluster.getMaxReduceTasks() * 0.9);
 String sort_reduces = jobConf.get("test.sort.reduces_per_host");
 if (sort_reduces != null) {
   num_reduces = cluster.getTaskTrackers() * Integer.parseInt(sort_reduces);
 }
 Class<? extends InputFormat> inputFormatClass = SequenceFileInputFormat.class;
 Class<? extends OutputFormat> outputFormatClass = SequenceFileOutputFormat.class;
 Class<? extends WritableComparable> outputKeyClass = BytesWritable.class;
 Class<? extends Writable> outputValueClass = BytesWritable.class;
 List<String> otherArgs = new ArrayList<String>();
 InputSampler.Sampler<K,V> sampler = null;
```

```
for(int i=0; i < args.length; ++i) {
  try {
    if ("-m".equals(args[i])) {
      jobConf.setNumMapTasks(Integer.parseInt(args[++i]));
    } else if ("-r".equals(args[i])) {
      num_reduces = Integer.parseInt(args[++i]);
    } else if ("-inFormat".equals(args[i])) {
      inputFormatClass =
        Class.forName(args[++i]).asSubclass(InputFormat.class);
    } else if ("-outFormat".equals(args[i])) {
      outputFormatClass =
        Class.forName(args[++i]).asSubclass(OutputFormat.class);
    } else if ("-outKey".equals(args[i])) {
      outputKeyClass =
        Class.forName(args[++i]).asSubclass(WritableComparable.class);
    } else if ("-outValue".equals(args[i])) {
      outputValueClass =
        Class.forName(args[++i]).asSubclass(Writable.class);
    } else if ("-totalOrder".equals(args[i])) {
      double pcnt = Double.parseDouble(args[++i]);
      int numSamples = Integer.parseInt(args[++i]);
      int maxSplits = Integer.parseInt(args[++i]);
      if (0 >= maxSplits) maxSplits = Integer.MAX_VALUE;
      sampler =
        new InputSampler.RandomSampler<K,V>(pcnt, numSamples, maxSplits);
    } else {
      otherArgs.add(args[i]);
    }
  } catch (NumberFormatException except) {
    System.out.println("ERROR: Integer expected instead of " + args[i]);
    return printUsage();
  } catch (ArrayIndexOutOfBoundsException except) {
    System.out.println("ERROR: Required parameter missing from " + args[i-1]);
    return printUsage(); // exits
  }
}
```

Sort Code in Hadoop 1.0.3 Distribution; part 3: more boilerplate code

Sort Code in Hadoop 1.0.3 Distribution; part 4: job settings, sampling to get pivot elements, run job command

```
  // Set user-supplied (possibly default) job configs
  jobConf.setNumReduceTasks(num_reduces);
  jobConf.setInputFormat(inputFormatClass);
  jobConf.setOutputFormat(outputFormatClass);
  jobConf.setOutputKeyClass(outputKeyClass);
  jobConf.setOutputValueClass(outputValueClass);

  // Make sure there are exactly 2 parameters left.
  if (otherArgs.size() != 2) {
    System.out.println("ERROR: Wrong number of parameters: " + otherArgs.size() + " instead of 2.");
    return printUsage();
  }
  FileInputFormat.setInputPaths(jobConf, otherArgs.get(0));
  FileOutputFormat.setOutputPath(jobConf, new Path(otherArgs.get(1)));

  if (sampler != null) {
    System.out.println("Sampling input to effect total-order sort...");
    jobConf.setPartitionerClass(TotalOrderPartitioner.class);
    Path inputDir = FileInputFormat.getInputPaths(jobConf)[0];
    inputDir = inputDir.makeQualified(inputDir.getFileSystem(jobConf));
    Path partitionFile = new Path(inputDir, "_sortPartitioning");
    TotalOrderPartitioner.setPartitionFile(jobConf, partitionFile);
    InputSampler.<K,V>writePartitionFile(jobConf, sampler);
    URI partitionUri = new URI(partitionFile.toString() + "#" + "_sortPartitioning");
    DistributedCache.addCacheFile(partitionUri, jobConf);
    DistributedCache.createSymlink(jobConf);
  }

  System.out.println("Running on " + cluster.getTaskTrackers() + " nodes to sort from " + FileInputFormat.getInputPaths(jobConf)[0] +
      " into " + FileOutputFormat.getOutputPath(jobConf) + " with " + num_reduces + " reduces.");
  Date startTime = new Date(); System.out.println("Job started: " + startTime);
  jobResult = JobClient.runJob(jobConf);
  Date end_time = new Date(); System.out.println("Job ended: " + end_time);
  System.out.println("The job took " + (end_time.getTime() - startTime.getTime()) /1000 + " seconds.");
  return 0;
}
```

```
  public static void main(String[] args) throws Exception {
    int res = ToolRunner.run(new Configuration(), new Sort(), args);
    System.exit(res);
  }

  /**
   * Get the last job that was run using this instance.
   * @return the results of the last job that was run
   */
  public RunningJob getResult() {
    return jobResult;
  }
}
```

Sort Code in Hadoop 1.0.3 Distribution;
part 5: main function

3. Secondary Sorting

- After having seen how MapReduce can be used to sort big data, we discuss a related design pattern that also leverages MapReduce's built-in sorting functionality.
- This pattern enables the programmer to ensure that each Reduce function call receives a value list that is sorted by some field (aka attribute) of each value.

3.1 Example Application

- Consider a big file of weather observations. For simplicity, assume each record is a tuple (date, temperature). (In reality there will be many more fields).
- Our goal is to find the maximum temperature for each year.

3.2 First Attempt: Year as Key

- The code below shows our first attempt. By using the year as the key, all temperature values for a year end up in the same Reduce call, making it easy to find the maximum by scanning through the list.
- Can we avoid having to scan through the entire list by making sure the largest temperature comes first?

```
map( date, temperature )
  emit( date.year, temperature)
```

```
reduce( year, [t1, t2,…] )
  max = t1
  for each other t in the input list
    if (t > max) then t = max
  emit( year, max )
```

3.3 Second Attempt: Year and Temperature as Key

- To leverage MapReduce's built-in sort functionality, we have to include all fields we want to use for sorting in the key. This implies that the temperature has to be in the key.
- The code below shows the corresponding program fragment. Unfortunately, each Reduce call can only see a single temperature value for each year. Hence it cannot find the maximum.
- Can we still make this work?

```
map( date, temperature )
  emit( (date.year, temperature), NULL )
```

```
reduce( (year, temperature), [NULL, NULL,…] )
  // ???
```

3.4 Second Attempt Continued

- We can use in-reducer combining, similar to in-mapper combining, to keep track of "state" across the different Reduce calls.
- For this to work, we need the right Partitioner that ensures that all temperatures for a given year are assigned to the same Reduce task. Any Partitioner that only considers the year for partition assignment will have this property.
- Since all Reduce calls in the same task are executed in key order, we know the following:
  - All records for some year Y will be processed before any records for any later year Y'.
  - By defining an appropriate key comparator that sorts by year first and then in decreasing order of temperature next, it is guaranteed that the first record the Reducer finds for year Y will be the one with the maximum temperature.
- Unfortunately, so far we have not won anything. We replaced the simple sequential scan of the value list (see first attempt) by a more complicated version that does the same through a series of Reduce function calls.
- Ideally we would like to combine the best of the first and the second attempt: have a single Reduce call per year (like in the first attempt), but ensure that the temperatures in the value list are in decreasing order (like in the second attempt).

Second Attempt: Algorithm

```
map( date, temperature )
  emit( (date.year, temperature), NULL )

getPartition( (year, temperature) )
  return myPartition( year )

keyComparator( (year, temperature) )
  // Sorts in increasing order of year first.
  // If the year is equal, sorts in decreasing
  // order of temperature next.
```

```
Class Reducer {
  currentYear

  setup() {
    currentYear = NULL
  }

  reduce( (year, temperature), [NULL,…] ) {
    if (year <> currentYear)
      // Another year found. Output its max temperature which
      // is guaranteed to be in the first record for this year.
      emit( year, temperature )
      currentYear = year

    // Else, another temperature for the current year was
    // found. It cannot exceed the first one found, hence we
    // ignore the record.
  }

  cleanup() {
    // Nothing to be done here
  }
}
```

3.5 Third Attempt: Best of Both Worlds

- We use the same Mapper, Partitioner, and key comparator as for the second attempt, but will use the Reducer from the first attempt.
- To make this work, we need a special grouping comparator. It is used to determine which key-value pairs are passed to the same Reduce function call.

Pseudo-Code for 3.5

```
map( date, temperature )
  emit( (date.year, temperature), NULL )

getPartition( (year, temperature) )
  return myPartition( year )

keyComparator( (year, temperature) )
  // Sorts in increasing order of year first.
  // If the year is equal, sorts in decreasing
  // order of temperature next.

groupingComparator( (year, temperature) )
  // Sorts in increasing order of year.
  // Does not consider temperature for sorting.
  // Hence two keys with the same year value
  // are considered identical, no matter the
  // temperature value.

reduce( year, [t1, t2,…] )
  emit( year, t1 )
```

3.6 How Does this Actually Work?

Assume Reduce task 0 received the records below from the Mappers. Notice that these records are sorted based on the key comparator (increasing order of year, decreasing order of temperature).

Key: Year	Key: Temperature	Value
2012	90	NULL
2012	80	NULL
2012	70	NULL
2014	85	NULL
2014	75	NULL
2014	65	NULL

3.6 Continued

Without the special grouping comparator, there is one Reduce call per distinct (year, temperature) pair.

Reduce call for key (2012, 90)

Reduce call for key (2014, 85)

Key: Year	Key: Temperature	Value
2012	90	NULL
2012	80	NULL
2012	80	NULL
2014	85	NULL
2014	85	NULL
2014	75	NULL

Reduce call for key (2012, 80)

Reduce call for key (2014, 75)

3.6 Continued

With the special grouping comparator, keys such as (2012, 90) and (2012, 80) are considered identical. Hence they are processed in the same Reduce function call. In general, there is only a single Reduce function call per year.

Reduce call for key (2012, *)

Key: Year	Key: Temperature	Value
2012	90	NULL
2012	80	NULL
2012	80	NULL
2014	85	NULL
2014	85	NULL
2014	75	NULL

Reduce call for key (2014, *)

3.7 Hadoop Code for Secondary Sort

```
public class MaxTemperatureUsingSecondarySort
 extends Configured implements Tool {

 static class MaxTemperatureMapper
  extends Mapper<LongWritable, Text, IntPair, NullWritable> {

  private NcdcRecordParser parser = new NcdcRecordParser();

  @Override
  protected void map(LongWritable key, Text value,
     Context context) throws IOException, InterruptedException {

   parser.parse(value);
   if (parser.isValidTemperature()) {
    context.write(new IntPair(parser.getYearInt(), parser.getAirTemperature()),
       NullWritable.get());
   }
  }
}
```

Notice that the value is NULL, because the year and temperature are in the key.

```
static class MaxTemperatureReducer
  extends Reducer<IntPair, NullWritable, IntPair, NullWritable> {

  @Override
  protected void reduce(IntPair key, Iterable<NullWritable> values,
      Context context) throws IOException, InterruptedException {

    context.write(key, NullWritable.get());
  }
}
```

The reducer only emits the first key, which due to secondary sorting, is the (year, temperature) pair with the maximum temperature for that year.

```
public static class FirstPartitioner extends Partitioner<IntPair, NullWritable> {

  public int getPartition(IntPair key, NullWritable value, int numPartitions) {

    // multiply by 127 to perform some mixing
    return Math.abs(key.getFirst() * 127) % numPartitions;
  }
}
```

The Partitioner makes sure all records for the same year end up in the same partition.

```
// Controls how keys are sorted before they are passed to the reducer
public static class KeyComparator extends WritableComparator {
  protected KeyComparator() {
    super(IntPair.class, true);
  }
  public int compare(WritableComparable w1, WritableComparable w2) {
    IntPair ip1 = (IntPair) w1; IntPair ip2 = (IntPair) w2;
    int cmp = IntPair.compare(ip1.getFirst(), ip2.getFirst());
    if (cmp != 0) {
      return cmp;
    }
    return -IntPair.compare(ip1.getSecond(), ip2.getSecond()); //reverse
  } }
```

This is the "regular" key comparator, using both year and temperature.

```
// Controls which keys are grouped into a single call of the reduce function
public static class GroupComparator extends WritableComparator {
  protected GroupComparator() {
    super(IntPair.class, true);
  }
  public int compare(WritableComparable w1, WritableComparable w2) {
    IntPair ip1 = (IntPair) w1; IntPair ip2 = (IntPair) w2;

    return IntPair.compare(ip1.getFirst(), ip2.getFirst());
  } }
```

This is the grouping comparator, using only year.

```
  @Override
  public int run(String[] args) throws Exception {
    Job job = JobBuilder.parseInputAndOutput(this, getConf(), args);
    if (job == null) {
      return -1;
    }

    job.setMapperClass(MaxTemperatureMapper.class);
    job.setPartitionerClass(FirstPartitioner.class);
    job.setSortComparatorClass(KeyComparator.class);
    job.setGroupingComparatorClass(GroupComparator.class);
    job.setReducerClass(MaxTemperatureReducer.class);
    job.setOutputKeyClass(IntPair.class);
    job.setOutputValueClass(NullWritable.class);

    return job.waitForCompletion(true) ? 0 : 1;
  }

  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new MaxTemperatureUsingSecondarySort(), args);
    System.exit(exitCode);
  }
}
```

Custom comparators and Partitioner have to be set explicitly.

3.8 Value-to-Key Conversion Design Pattern

- In the example, we wanted to use year to partition the data, but (year, temperature) for sorting.
- The general value-to-key conversion design pattern for this problem is defined as follows:
  - To partition by attribute X and then sort each X-group by another attribute Y, make (X, Y) the key.
  - Define a key comparator to order by composite key (X, Y).
  - Define Partitioner and grouping comparator for key (X, Y) to consider only X for partitioning and grouping.
- For the weather data example, sorting by temperature is actually not necessary since the maximum for a year can be found in linear time (see first attempt), instead of sorting's superlinear compelxity ($n \cdot \log n$).
- However, if the goal is to emit the decreasing list of temperatures for each year, then secondary sort is typically more efficient than user code that does its own sorting in the reduce function.

3.9 Summary

- Local aggregation:
  - Opportunities to perform local aggregation vary.
  - Combiners can significantly reduce cost by decreasing the amount of data sent from Mappers to Reducers. Unfortunately the programmer cannot control if and when a Combiner will be executed by Hadoop.
  - In-mapper combining can be used instead of a Combiner. It gives the programmer explicit control, but requires changes to the Mappers.
  - Both Combiners and in-mapper combining are only applicable for certain types of aggregate functions. They are only effective if many Map output records in the same Map task have the same key.
- Exploiting MapReduce's built in sorting functionality:
  - We can sort big data easily with a simple identity Mapper and Reducer. This requires the use of an appropriate Partitioner that performs range-partitioning.
  - Secondary sort requires an appropriate Partitioner as well, but also separate key and grouping comparators.