2 Questions of sorting

In this chapter we will accomplish a few goals. We will add more order to the MapReduce tasks, by sorting not only on keys but also on values. There are cases where sorting on values is useful, and by adding a little touch to your MapReduce code, you can achieve significant results. Maybe even more important is using this practical need to show you how to modify the MapReduce default behavior. This will allow you to start tweaking the framework and to get a glimpse into the inner working of it. Finally, we will look at Terasort, show what you can gain from this knowledge.

Sorting is an essential step of MapReduce. As part of processing, records are sorted by keys after the map step, they are partitioned on the local disk of each tasktracker node, according to the number of reducers, and then shipped to the reducers for merging and grouping. Input to reduce is always sorted by keys. You can and should rely on this guarantee when writing your code. For example, in the previous chapter we used the fact that all equal keys come to your reducer in one group, when we eliminated duplicates in the data.

However, the values that correspond to a particular key are in no particular order, since the merge step in the reducer does not take the values into account. In other words, in the code segment below, which was part of our Reducer in Chapter 1, the values themselves are not sorted, although the keys are.

```
int sum = 0;
for (IntWritable val : values) {
  sum += val.get();
}
context.write(key, new IntWritable(sum));
```

2.1 Secondary sort on values

In some use cases, sorting the values may be a desirable feature. For example, consider the employee-department data in the table below:

Table 2.1 Employee-Department

Employee ID	Name	Department
1	Max Depaoli	SALES
2	Neil Hoefer	OPERATIONS
3	Guy Orosz	IT
4	Tyrone Bradburn	IT
5	Malinda Twining	SALES
3	Guy Orosz	IT
1	Max Depaoli	SALES

It contains records from a company's database holding information for employees. There are three fields associated with each employee: Employee Id, Name and Department. Notice that the sample input contains some duplicate entries.

Suppose we are interested in finding the number of employees in each department. In SQL, assuming the data is stored in a table called employee, we can achieve this functionality by the following query:

```
SELECT DISTINCT COUNT(*)
FROM employee GROUP BY department
```

which filters duplicate rows by DISTINCT and reports the count per each department using the GROUP BY operator yielding the following result.

Table 2.2 GROUP BY result

Department	**Count(*)**
IT	2
OPERATIONS	1
SALES	2

One way of obtaining the same output using MapReduce is to run two jobs, one for filtering the duplicate entries, and another for grouping by the department and for counting the records. The first mapper emits pairs of the form <EmployeeId, Department> so a single call to reduce contains all entries associated with a single employee. The first reducer then simply emits only one pair of the form <Department, 1>, filtering the duplicates. The output of the first MapReduce job will be as follows:

```
IT 1
IT 1
OPERATIONS 1
SALES 1
SALES 1
```

The second MapReduce job is identical to the Word Count example from Chapter 1, which counts identical entries from each department and yields the desired output as above.

Although we obtained the correct results, we had to run two MapReduce jobs, which means twice as much job initialization time, in addition to writing the output of the first job to the disk and reading it back as input to the second job. However, we can perform the same computation in a single MapReduce job, if we use a better design, which sorts the values that are sent to a single call in the reduce function.

Consider what happens when the mappers output a <Department, EmployeeId> pairs. MapReduce then sorts and groups records by their keys, but the values that are associated with a particular key may be out of order. In our sample dataset, the input to the reducers may look as follows:

```
Key          Value
IT             3
IT             4
IT             3
OPERATIONS     2
SALES          1
SALES          5
SALES          1
```

To filter duplicates, the reducer can start reading records into a hash table where the key is employee id. Using the `exists` operator, duplicates can easily be skipped and the final count associated with a department is reported as the output. This approach obviously would only work as long as there is enough memory to store all distinct values for the corresponding department. This is not scalable and may result in heap errors with large data or limited physical memory.

The need for a hash table can be eliminated by sorting the values in each group. For example, let's assume we can guarantee that the output from the mapper looks like:

```
Key          Value
IT             3
IT             3
IT             4
OPERATIONS     2
SALES          1
SALES          1
SALES          5
```

For each group, the reducer can simply iterate over the values and compare every value with the previous one to decide whether it is a duplicate or not. Since the values arrive to the iterator in sorted order, there is no need to store all distinct values in memory. But, how do we do that?

2.1.1 Implementing Custom WritableComparable Objects

We have to get a bit more creative to implement this solution in Hadoop, since there is no default option to automatically sort values in each group. To start with, we need a custom key which is a tuple containing two fields: *Department* and *Employee Id*. We define a custom key as a subclass of the `org.apache.hadoop.io.WritableComparable` interface:

Listing 2.1 KeyTuple.java

```
private Text department;
private LongWritable id;
public KeyTuple() {
  set(new Text(), new LongWritable());
}
public KeyTuple(String department, long id) {
  set(new Text(department), new LongWritable(id));
}
public void set(Text department, LongWritable id) {
  this.department = department;
  this.id = id;
}
public void set(String department, long id) {
  this.department.set(department);
  this.id.set(id);
}
public Text getDepartment(){
  return department;
}
public LongWritable getId(){
  return id;
}
@Override
public void readFields(DataInput in) throws IOException {
  department.readFields(in);
  id.readFields(in);
}
@Override
public void write(DataOutput out) throws IOException {
  department.write(out);
  id.write(out);
}
@Override
public int hashCode() {
  return department.hashCode() * 1000003 + id.hashCode();
}
@Override
public boolean equals(Object o) {
  if(this == o) {
    return true;
  }
  if (!(o instanceof KeyTuple)) {
    return false;
  }
  KeyTuple other = (KeyTuple) o;
  return department.equals(other.department) &&
    id.equals(other.id);
}
//@Override
public int compareTo(KeyTuple other) {
  int cmp = department.compareTo(other.department);
  if (cmp != 0) {
    return cmp;
  }
  return id.compareTo(other.id);
}
```

The class KeyTuple.java has two private members: `department` and `id` which are instances of org.apache.hadoop.io.Text and org.apache.hadoop.io.LongWritable respectively. The write() and readFields() methods are overriden to have explicit control over serialization. The hashCode() method uses a prime multiplier to produce accurate hash values for KeyTuple objects. Note that careful attention must be paid when overriding hashCode() as it is used by the partitioner to decide which reducer receives a KeyTuple instance. For example, not using a prime multiplier may result in overloading one of the reducers with more intermediate output than the others, causing load balancing problems which in turn may yield to non-utilization of the reducers and thus longer job execution times. The equals() method first checks whether two KeyTuple object references are the same before checking for the equality of the actual class members. Finally, the compareTo() method compares two KeyTuple instances first by department, and next by id, which is used to define the sort order. With KeyTuple to represent a key having two data fields, the mappers can emit records of the form <<Department, EmployeeId>,EmployeeId >. Observe that we duplicate Employee Id information using it both as part of the key as well as the value. The implementation of the mapper class is given in this listing

Listing 2.2 MapClass.java

```
public class MapClass extends Mapper<LongWritable,Text,
  KeyTuple,LongWritable> {
    private KeyTuple K = new KeyTuple();
    private LongWritable V = new LongWritable();
    private String id, dept;
    @Override
    public void map(LongWritable key, Text value, Context
      context) throws
      IOException, InterruptedException {
      String record = value.toString();
      String tokens[] = record.split(",");
      id = tokens[0];
      dept = tokens[2];
      K.set(dept, Long.parseLong(id));
      V.set(Long.parseLong(id));
      context.write(K,V);
    }
}
```

Since the map output is sorted by the keys, which in this case done by the ordering defined in the compareTo() method, the intermediate output will be as shown.

```
Key              Value
IT,3               3
IT,3               3
IT,4               4
OPERATIONS,2       2
SALES,1            1
SALES,1            1
SALES,1            5
```

2.1.2 Overriding the Default Partitioner

At the shuffle step, the challenge is to ship records of the same department to the same reducer. The default partitioner uses the hashCode() method of the key class to produce a hash value for the key, and takes modulo R - where R is the number of reducers - to decide which reducer an intermediate record goes to. However, their default behavior is not sufficient for our purposes, since two KeyTuple objects having the same department but different id fields may end up at separate reducers. A custom partitioner that only takes into account the department field is shown in the listing below

Listing 2.3 CustomPartitioner.java

```
public class CustomPartitioner extends Partitioner<KeyTuple, LongWritable> {
  @Override
  public int getPartition(KeyTuple key, LongWritable value, int numReduceTasks) {
    return (key.getDepartment().hashCode() & Integer.MAX_VALUE) % numReduceTasks;
  }
}
```

2.1.3 Designing a Custom Grouping Comparator

Hadoop uses two comparator classes at different stages of a MapReduce job.

1. Sort Comparator: Used to control how keys are sorted before reduce.
2. Grouping Comparator: Used to control which keys are a single call to reduce.

Again by default, Hadoop uses the compareTo() method of the key class for both sorting and grouping the records. In our case, the objective is to group all records having the same department field together and pass them in a single call to the reduce method. The trick is similar to what we just did for the partitioner. The grouping comparator should only consider the department field when comparing two KeyTuple objects, and ignore the id part. Listing 1.1.3 shows a custom grouping comparator implemented for this purpose.

Listing 2.4 GroupComparator.java

```
public class GroupComparator extends WritableComparator{
  public GroupComparator() {
    super(KeyTuple.class, true);
  }
  @Override
  public int compare(WritableComparable K1,
    WritableComparable K2) {
    KeyTuple t1 = (KeyTuple) K1;
    KeyTuple t2 = (KeyTuple) K2;
    return t1.getDepartment().compareTo(t2.getDepartment());
  }
}
```

Observe how we simulate a secondary sort on values by using a cus- tom grouping comparator. Records having a common department arrive at the same reducer via CustomPartitioner, and they are already sorted by their id fields via KeyTuple.compareTo(), which is the default sort compara- tor. Hadoop then performs a linear scan accross the sorted records and uses GroupComparator to specify groups of records having the same department field.

The custom partitioner and grouping comparator implementations can be passed as parameters to an instance of org.apache.hadoop.mapreduce.Job via setPartitionerClass() and setGroupingComparatorClass() methods dur- ing job configuration.

Since the reduce method receives the records in sorted order by the value - in this case id - all it has to do is to skip duplicates by comparing each value with the preceding one. The reducer implementation is given in listing 1.1.3.

Listing 2.5 Reducer.java

```
public class ReduceClass extends Reducer<KeyTuple,LongWritable,Text,LongWritable> {
long preceding = -1; // represents the preceding id\bigcup
@Override
public void reduce(KeyTuple K, Iterable<LongWritable>
  values, Context context) throws
  IOException, InterruptedException {
    int sum = 0;
    for (LongWritable val : values) {
      if( !(val.get() == preceding) ) {
        sum ++;
        preceding = val.get();
      }
    }
    context.write(K.getDepartment(), new LongWritable(sum));
  }
}
```

2.1.4 Optimizing the Comparators

KeyTuple.compareTo() is used to compare two KeyTuple objects that are deserialized from binary streams before the actual method is called. Deserialization is a time consuming operation since it involves object creation. Hadoop uses an optimization technique to compare WritableComparable objects in raw binary format. As an example, consider a Text object which is represented by a UTF-8 byte array in the underlying implementation. The listing below shows part of the implementation of org.apache.hadoop.io.Text class.

Listing 2.6 Text.java

```
public class Text extends BinaryComparable
  implements WritableComparable<BinaryComparable> {
  ...
  private byte[] bytes;
  private int length;
  ...
  public void write(DataOutput out) throws IOException {
    WritableUtils.writeVInt(out, length);
    out.write(bytes, 0, length);
  }
  ...
  public void readFields(DataInput in) throws IOException {
    int newLength = WritableUtils.readVInt(in);
    setCapacity(newLength, false);
    in.readFully(bytes, 0, newLength);
    length = newLength;
  }
  public static class Comparator extends WritableComparator {
    public Comparator() {
      super(Text.class);
    }
  public int compare(byte[] b1, int s1, int l1,
    byte[] b2, int s2, int l2) {
    int n1 = WritableUtils.decodeVIntSize(b1[s1]);
    int n2 = WritableUtils.decodeVIntSize(b2[s2]);
    return compareBytes(b1, s1+n1, l1-n1, b2, s2+n2, l2-n2);
  }
}
  static {
    // register this comparator
    WritableComparator.define(Text.class, new Comparator());
  }
}
```

When an instance of Text is serialized, first a variable length integer representing the length of the string in UTF-8 bytes is written to the out- put stream. Following that the actual byte array which contains the string in UTF-8 bytes is serialized. Text.readFields() simply reverses this opera- tion by initially reading the length of the byte array, followed by the bytes themselves from the input stream. Text.Comparator() is an optimized com- parator that compares two Text instances in binary form, without the need for deserialization. The compare() method reads the lengths of byte arrays and compares them starting at the corresponding indices where the actual UTF-8 byte array representations of the strings start at.

The same idea of comparing two WritableComparable's in raw binary format can be applied to KeyTuple as it contains a Text and a LongWritable member. Listing below shows an optimized comparator for KeyTuple.

Listing 2.7 FastComparator.java

```
public static class FastComparator extends
  WritableComparator{
  public FastComparator() {
    super(KeyTuple.class, true);
  }
  @Override
  public int compare(byte[] b1, int s1, int l1, byte[]
    b2, int s2, int l2) {
    WritableComparator comparator;
    try{
      comparator = new Text.Comparator();
      int deptLen1 = WritableUtils.decodeVIntSize(b1[s1]) + readVInt(b1, s1);
      int deptLen2 = WritableUtils.decodeVIntSize(b2[s2]) + readVInt(b2, s2);
      int cmp = comparator.compare(b1, s1, deptLen1, b2, s2, deptLen2);
      if(cmp != 0) {
        return cmp;
      }
      comparator = new LongWritable.Comparator();
      return comparator.compare(b1, s1+deptLen1, l1-deptLen1, b2, s2+deptLen2, l2-de
      } catch(IOException e) {
        throw new IllegalArgumentException(e);
      }
    }
  }
  static {
    WritableComparator.define(KeyTuple.class, newFastComparator());
  }
}
```

KeyTuple.write() serializes an object by writing the department and id fields to the output stream respectively. FastComparator.compare() first compares the department fields in raw binary format by computing the number of bytes required to store the Text objects. If department is the same for both, the id fields are compared in raw binary format via an instance of LongWritable.Comparator. Notice how the record boundries are computed using the utility methods WritableUtils.decodeVIntSize() and WritableComparator.readVInt(). This is illustrated in figure below:

Figure 1.1 here

2.1.5 Sort benchmarks

Hadoop is designed for large scale data analysis, and it mainly targets I/O intensive applications although there are some exceptions to this general use case. An essential way to test and verify the correctness of a distributed framework is to benchmark it with useful metrics.

TeraByte Sort is an I/O benchmark that measures the amount of time to sort 1012 bytes of data. The sort input records are 100 bytes in length, with the first 10 bytes being a random key and the rest being payload. Random records are generated according to the official TeraSort generator, and the input may initially be divided into smaller files to allow node-level parallelism. The sorted output may be in the form of multiple files with the requirement that there is total ordering among them. The benchmarks are performed every year and the winners are awarded at ACM SIGMOD.

In 2009, Hadoop won the TeraByte Sort benchmark and the code to generate, sort and valide the benchmark data is publicly available in package org.apache.hadoop.examples.terasort. In this section, we will explain how Hadoop's distributed sorting algorithm works and discuss some of the key design decisions that resulted in satisfactory scalability characteristics. Package org.apache.hadoop.examples.terasort contians three programs:

- TeraGen: Generates the data and stores the input in HDFS.
- TeraSort: Sorts the data. This is the actual benchmark code.
- TeraValidate: Validates the correct ordering of the output.

You can run the sort benchmark on your Hadoop cluster with a relatively smaller data set to test your cluster after installation. To generate, sort and validate 10 GB of data, type in:

Listing 2.8 Type in

```
hadoop jar hadoop-0.2*-examples.jar teragen 100000000 input
hadoop jar hadoop-0.2*-examples.jar terasort input output
hadoop jar hadoop-0.2*-examples.jar teravalidate output validate
```

We will now take a deeper look at the implementation of these programs.

2.1.6 Generating Data

TeraGen generates 100 byte long records in the following format:

Table of generated data here

Throughout the code, a record is referred as a row. The keys are random characters from the set ' ' .. ' '. The rowid is the right justified row id as an integer starting from 0. The filler consists of 7 runs of 10 characters from 'A' to 'Z'.

TeraGen contains a RandomGenerator class which is used to generate random keys. RandomGenerator is capable of quickly jumping to a particular key in the key space. This is done by pre-computing the seeds at every 27x106 keys. The default constructor initializes the random generator at seed 0. The overloaded constructor can jump to a particular key by linearly iterating over the key, starting at an appropriate pre-computed seed.

Listing 2.9 Random Generator

```
static class RandomGenerator {
  private long seed = 0;
  private static final long mask32 = (1l<<32) - 1;
  /**
  * The number of iterations separating the precomputed seeds.
  */
  private static final int seedSkip = 128 * 1024 * 1024;
  /**
  * The precomputed seed values after every seedSkip iterations.
  * There should be enough values so that a 2**32 iterations are
  * covered.
  */
  private static final long[] seeds =
    new long[]{0L,
    4160749568L,
    4026531840L,
    ...
    134217728L,
  };
  /**
  * Start the random number generator on the given iteration.
  * @param initalIteration the iteration number to start on
  */
  RandomGenerator(long initalIteration) {
    int baseIndex = (int) ((initalIteration & mask32) / seedSkip);
    seed = seeds[baseIndex];
    for(int i=0; i < initalIteration % seedSkip; ++i) {
      next();
    }
  }
  RandomGenerator() {
    this(0);
  }
  long next() {
    seed = (seed * 3141592621l + 663896637) & mask32;
    return seed;
  }
}
```

2.1.7 Designing a custom InputFormat

TeraGen generates input data by creating map tasks each of which produce rows within a given range. For example, the first map task produces rows in the range [0;M-1], the second map task produces rows in the range [M; 2M- 1] and so on. This is accomplished by designing a custom InputFormat, InputSplit and a RecordReader.

InputFormat describes the input-specification for a Map-Reduce job. It validates the types of the input < key; value > pairs, divides the input files into logical units called Inputsplit's and provides a RecordReader class to read and process records inside each InputSplit.

InputSplit is essentially a structure that represents a particular range in the input. For example, when processing a file, an InputSplit contains a byte location in the file that represents the starting offset, and a length variable that represents the size of the split. Each file may be divided into one or more InputSplits depending on the minimum and maximum split size and the length of the file.

RecordReader takes an InputSplit, and transforms it into one or more records which are then sent to Map or Reduce tasks. TeraGen defines a custom InputFormat called RangeInputFormat. RangeIn- putFormat assigns ranges of longs to each mapper where a long represents the index that the mapper will start generating records from. In our example above, the first map task will start generating rows from 0, the second map task will start generating rows from M and so on. The split boundaries are computed by the getSplits() method, which divides the total number of rows by the number of splits (map tasks). RangeInputFormat also defines the getRecordReader() method which is automatically called by the framework to process the input splits.

Part of the implementation of RangeInputFormat is given in listing 1.2.2.

Listing 2.10 RangeInputFormat

```
/**
* An input format that assigns ranges of longs to each
mapper.
*/
static class RangeInputFormat
  implements InputFormat<LongWritable, NullWritable> {
  ...
  public RecordReader<LongWritable, NullWritable>
  getRecordReader(InputSplit split, JobConf job,
    Reporter reporter) throws IOException {
    return new RangeRecordReader((RangeInputSplit) split);
  }
  ...
  /**
  * Create the desired number of splits, dividing the number of rows
  * between the mappers.
  */
  public InputSplit[] getSplits(JobConf job, int numSplits) {
    long totalRows = getNumberOfRows(job);
    long rowsPerSplit = totalRows / numSplits;
    System.out.println("Generating " + totalRows + " using " + numSplits +
      " maps with step of " + rowsPerSplit);
    InputSplit[] splits = new InputSplit[numSplits];
    long currentRow = 0;
    for(int split=0; split < numSplits-1; ++split) {
      splits[split] = new RangeInputSplit(currentRow, rowsPerSplit);
      currentRow += rowsPerSplit;
    }
    splits[numSplits-1] = new RangeInputSplit(currentRow, totalRows - currentRow);
    return splits;
  }
}
```

RangeInputFormat defines the static class RangeInputSplit, which represents a section of the input records, identified by a starting offset and the number of rows inside that range. RangeRecordReader takes a RangeInput- Split as argument, reads the starting row and the number of rows inside that split and produces all numbers within that range.

Notice that RangeRecordReader does not generate the actual random numbers. It only produces the row id's, which are longs in that range. In our example above, the first map task will receive numbers in the range [0,M-1] and use the random number generator to generate all the rows within that range. The second map task will do the same for the rows within range [M, 2M-1]. Inside a map task, the framework calls RangeRecor- dReader.next() repeatedly until all the numbers in that range are produced by RangeRecordReader.

The implementation of RangeRecordReader is given in listing 1.2.2

Listing 2.11 RangeRecordReader

```
/**
* A record reader that will generate a range of numbers.
*/
static class RangeRecordReader implements RecordReader<LongWritable, NullWritable> {
  long startRow;
  long finishedRows;
  long totalRows;
  public RangeRecordReader(RangeInputSplit split) {
    startRow = split.firstRow;
    finishedRows = 0;
    totalRows = split.rowCount;
   }
 }
 public void close() throws IOException {
    // NOTHING
  }
  public LongWritable createKey() {
    return new LongWritable();
  }
  public NullWritable createValue() {
    return NullWritable.get();
  }
  public long getPos() throws IOException {
    return finishedRows;
  }
  public float getProgress() throws IOException {
    return finishedRows / (float) totalRows;
  }
  public boolean next(LongWritable key, NullWritable value) {
    if (finishedRows < totalRows) {
      key.set(startRow + finishedRows);
      finishedRows += 1;
      return true;
    } else {
      return false;
    }
  }
}
```

Mappers receive a sequence of numbers starting from a particular range where each number corresponds to a row id in the key space. For each row id, random keys and fillers are generated according to TeraSort specifications and emitted as output. The methods addKey(), addRowId() and addFiller() set the key, row id and the filler respectively. The output of the map tasks is in the form of < key; value > pairs that are 10 and 90 bytes long Text objects.

Listing 1.2.2 shows the code for the SortGenMapper class.

Listing 2.12 SortGenMapper

```
/**
  * The Mapper class that given a row number, will generate the appropriate
  * output line.
*/
public static class SortGenMapper extends MapReduceBase
  implements Mapper<LongWritable, NullWritable, Text, Text> {
    private Text key = new Text();
    private Text value = new Text();
    private RandomGenerator rand;
    private byte[] keyBytes = new byte[12];
    private byte[] spaces = "  ".getBytes();
    private byte[][] filler = new byte[26][];
    {
      for(int i=0; i < 26; ++i) {
      filler[i] = new byte[10];
      for(int j=0; j<10; ++j) {
      filler[i][j] = (byte) ('A' + i);
    }
  }
}
/**
* Add a random key to the text
* @param rowId
*/
private void addKey() {
  for(int i=0; i<3; i++) {
    long temp = rand.next() / 52;
    keyBytes[3 + 4*i] = (byte) (' ' + (temp % 95));
    temp /= 95;
    keyBytes[2 + 4*i] = (byte) (' ' + (temp % 95));
    temp /= 95;
    keyBytes[1 + 4*i] = (byte) (' ' + (temp % 95));
    temp /= 95;
    keyBytes[4*i] = (byte) (’ ’ + (temp % 95));
  }
  key.set(keyBytes, 0, 10);
}
/**
* Add the rowid to the row.
* @param rowId
*/
private void addRowId(long rowId) {
  byte[] rowid = Integer.toString((int) rowId).getBytes();
  int padSpace = 10 - rowid.length;
  if (padSpace > 0) {
    value.append(spaces, 0, 10 - rowid.length);
  }
  value.append(rowid, 0, Math.min(rowid.length, 10));
}
/**
* Add the required filler bytes. Each row consists of 7 blocks of
* 10 characters and 1 block of 8 characters.
* @param rowId the current row number
*/
private void addFiller(long rowId) {
  int base = (int) ((rowId * 8) % 26);
  for(int i=0; i<7; ++i) {
    value.append(filler[(base+i) % 26], 0, 10);
  }
  value.append(filler[(base+7) % 26], 0, 8);
}
public void map(LongWritable row, NullWritable ignored,
  OutputCollector<Text, Text> output, Reporter reporter) throws IOException {
    long rowId = row.get();
    if (rand == null) {
      // we use 3 random numbers per a row. See addKey()
      rand = new RandomGenerator(rowId*3);
    }
    addKey();
    value.clear();
    addRowId(rowId);
    addFiller(rowId);
    output.collect(key, value);
  }
}
```

You can read the output of the data generator by the cat command. Ob- serve the right justified row id next to the filler.

hadoop fs -cat input/part-00000 j head -n 5

Listing 2.13 Output

```
.tˆ#\|v$2\
0AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDDD
DDEEEEEEEEEEFFFFFFFFFFGGGGGGGGGGHHHHHHHH
75@˜?’WdUF
1IIIIIIIIIIJJJJJJJJJJKKKKKKKKKKLLLLLLLL
LLMMMMMMMMMMNNNNNNNNNNOOOOOOOOOOPPPPPPPP
w[o||:N>H,
2QQQQQQQQQQRRRRRRRRRRSSSSSSSSSSTTTTTTTT
TTUUUUUUUUUUVVVVVVVVVVWWWWWWWWWWXXXXXXXX
ˆEu)<n#kdP
3YYYYYYYYYYZZZZZZZZZZAAAAAAAAAABBBBBBBB
BBCCCCCCCCCCDDDDDDDDDDEEEEEEEEEEFFFFFFFF
+l-$$OE/ZH
4GGGGGGGGGGHHHHHHHHHHIIIIIIIIIIJJJJJJJJ
JJKKKKKKKKKKLLLLLLLLLLMMMMMMMMMMNNNNNNNN
```

The overall data generation process is visualized in figure 1.2.2.

Figure 1.2.2 here

2.1.8 Sorting

Input records are sorted by TeraSort, which uses a custom InputFormat to interpret 100 byte long rows as < key; value > pairs having 10 and 90 bytes respectively.

TeraInputFormat contains TeraRecordReader, which takes an org.apache. hadoop.mapred.FileSplit as input. FileSplit is a section of an input file that contains a starting byte offset and the length of the section. Input files containing unsorted data are automatically divided into one or more FileSplits and each split is then passed to an instance of TeraRecordReader. Observe that each record ends with a new line character to identify record boundaries easily. Hadoop already contains org.apache.hadoop.mapred.

LineRecordReader which reads files line by line, producing a key as the offset in the file and the line as the value. TeraRecordReader uses LineRecor- dReader to identify new lines in a given split and produces records having a 10-byte key and 90-byte values. This operation is performed inside the next() method, which uses LineRecordReader to read a single line and partitions it into a key and a value.

The source code for TeraRecordReader is given in listing 1.2.3.

Listing 2.14 TeraRecordReader

```
static class TeraRecordReader implements RecordReader<Text,Text> {
  private LineRecordReader in;
  private LongWritable junk = new LongWritable();
  private Text line = new Text();
  private static int KEY_LENGTH = 10;
  public TeraRecordReader(Configuration job, FileSplit split) throws IOException {
    in = new LineRecordReader(job, split);
  }
...
  public boolean next(Text key, Text value) throws IOException {
    if (in.next(junk, line)) {
      if (line.getLength() < KEY_LENGTH) {
        key.set(line);
        value.clear();
      } else {
        byte[] bytes = line.getBytes();
        key.set(bytes, 0, KEY_LENGTH);
        value.set(bytes, KEY_LENGTH, line.getLength() - KEY_LENGTH);
    }
      return true;
    } else {
      return false;
    }
  }
}
```

2.1.9 Total Order Partitioning

TeraSort benchmarks allow the final output to be in multiple files with the requirement that there is total ordering among them. Total ordering means if the output files are concatenated, the resulting output should also be sorted. Consider an example scenario where we want to sort numbers in the set f7,4,2,8,1,6,9,0,3,5g with two reducers. If we use Hadoop's default hash partitioner, we cannot guarantee that records in ranges [0-4] and [5-9] will go to the first and second reducer respectively. We may instead have two sorted output files as:

Listing 2.15 Two sorted output files

```
File 1 File 2
0   1
3   2
6   4
8   5
9   7
```

When these two files are concatenated, the output will be out of order. Thus, we have to implement a custom partitioner that sends all records less than five to the first reducer and the rest to the second one.

Hadoop's TeraSort implementation follows a similar approach by dividing the key space into N partitions where N is the number of reducers. A key requirement to achieve this functionality is to determine N - 1 partition boundaries. The total order partitioner then sends each row to the correct reducer ensuring there is total ordering among the keys. In particular, all rows with a key in the range [boundary i-1, boundary i] are sent to reducer i.

2.1.10 Sampling Input Data

Notice that in the example above, we know in advance that 5 is a perfect partition boundary because it divides the record set into two subsets with same cardinality. However, we cannot always come up with perfect parti- tion boundaries unless sorting the data - which turns this to a chicken-egg problem.

As a workaround, TeraSort samples the input by reading N small sections from the unsorted input files and sorts them. Then, it down samples this smaller sorted data set by dividing it into N equally sized partitions and selecting N - 1 keys in the partition boundaries as final samples. These keys are then used by TeraSort's TotalOrderPartitioner to determine the correct reducer that a row should be shipped to. This process is visualized in figure 1.2.5.

Figure 1.2.5 - Sort input files generated by TeraGen

TeraInputFormat contains the method writePartitionFile() which gets the input splits from the framework through a JobConf object and samples them. The maximum number of samples is 10. The default sample size is 100000, but it can be increased to improve the quality of the sample and thus balance the load among the reducers more effectively when the actual sorting takes place. The number of records per sample is computed by di- viding the total sample size by the number of samples. Records from each sample input split are read and their keys are added to a TextSampler ob- ject. Once all the sample records are collected, they are down sampled via TextSampler.createPartitions(), which returns N - 1 keys to be used as par- tition boundaries. The final output is written to an org.apache.hadoop.io. SequenceFile instance on HDFS. A SequenceFile is a at file consisting of binary < key, value > pairs.

Part of the code for TeraInputFormat is given in listing 1.2.5.

Listing 2.16 TeraInputFormat

```
public class TeraInputFormat extends FileInputFormat<Text,Text> {
  static final String PARTITION_FILENAME = "_partition.lst";
  static final String SAMPLE_SIZE = "terasort.partitions.sample";
  private static JobConf lastConf = null;
  private static InputSplit[] lastResult = null;
  ...
  /**
  * Use the input splits to take samples of the input and generate sample
  * keys. By default reads 100,000 keys from 10 locations in the input, sorts
  * them and picks N-1 keys to generate N equally sized partitions.
  * @param conf the job to sample
  * @param partFile where to write the output file to
  * @throws IOException if something goes wrong
  */
  public static void writePartitionFile(JobConf conf, Path partFile) throws
    IOException {
    TeraInputFormat inFormat = new TeraInputFormat();
    TextSampler sampler = new TextSampler();
    Text key = new Text();
    Text value = new Text();
    int partitions = conf.getNumReduceTasks();
    long sampleSize = conf.getLong(SAMPLE_SIZE, 100000);
    InputSplit[] splits = inFormat.getSplits(conf, conf. getNumMapTasks());
    int samples = Math.min(10, splits.length);
    long recordsPerSample = sampleSize / samples;
    int sampleStep = splits.length / samples;
    long records = 0;
    // take N samples from different parts of the input
    for(int i=0; i < samples; ++i) {
      RecordReader<Text,Text> reader =
        inFormat.getRecordReader(splits[sampleStep * i], conf, null);
      while (reader.next(key, value)) {
        sampler.addKey(key);
        records += 1;
        if ((i+1) * recordsPerSample <= records) {
          break;
        }
      }
    }
    FileSystem outFs = partFile.getFileSystem(conf);
    if (outFs.exists(partFile)) {
      outFs.delete(partFile, false);
    }
    SequenceFile.Writer writer =
      SequenceFile.createWriter(outFs, conf, partFile, Text. class, NullWritable.cla
    NullWritable nullValue = NullWritable.get();
    for(Text split : sampler.createPartitions(partitions)) {
      writer.append(split, nullValue);
    }
    writer.close();
  }
...
}
```

The down sampling process is done via TextSampler implemented as a static class inside TeraInputFormat. TextSampler implements the utility in- terface org.apache.hadoop.util.IndexedSortable which allows creating collections capable of being sorted by org.apache.hadoop.util.IndexedSorter algorithms - HeapSort and QuickSort. It also overrides swap() and compare() methods that are used by any class that implements IndexedSorter. The code for TextSampler is given in listing 1.2.5.

Listing 2.17 TextSampler

```
static class TextSampler implements IndexedSortable {
  private ArrayList<Text> records = new ArrayList<Text>();
    public int compare(int i, int j) {
    Text left = records.get(i);
    Text right = records.get(j);
    return left.compareTo(right);
  }
  public void swap(int i, int j) {
    Text left = records.get(i);
    Text right = records.get(j);
    records.set(j, left);
    records.set(i, right);
  }
  public void addKey(Text key) {
    records.add(new Text(key));
  }
  /**
  * Find the split points for a given sample. The sample keys are sorted
  * and down sampled to find even split points for the partitions. The
  * returned keys should be the start of their respective partitions.
  * @param numPartitions the desired number of partitions
  * @return an array of size numPartitions - 1 that holds the split points
  */
  Text[] createPartitions(int numPartitions) {
    int numRecords = records.size();
    System.out.println("Making " + numPartitions + " from " + numRecords + " records
    if (numPartitions > numRecords) {
      throw new IllegalArgumentException
        ("Requested more partitions than input keys (" + numPartitions + " > " + num
  }
  new QuickSort().sort(this, 0, records.size());
  float stepSize = numRecords / (float) numPartitions;
  System.out.println("Step size is " + stepSize);
  Text[] result = new Text[numPartitions-1];
  for(int i=1; i < numPartitions; ++i) {
    result[i-1] = records.get(Math.round(stepSize * i));
  }
  return result;
}
```

The createPartitions() method sorts a colletion of records, and down sam- ples them by selecting numPartitions records that are stepSize apart. It returns the result as the final sample to be used by the total order partitioner.

2.1.11 Distributed Cache

Any class that implements org.apache.hadoop.mapred,Partitioner con- tains the getPartition() method that decides which reducer a given record should be sent to. The partitioning operation is performed locally on-the-fly, right after a record is emitted inside the map function. Thus, each map task results in local partition files as intermediate output, which are then sorted and sent to their corresponding reducers.

TeraSort contains TotalOrderPartitioner, a custom partitioner class that decides where to send a 100 byte TeraSort record by comparing its 10 byte key with the partition boundaries sampled previously from the input data. Since the partitioner code is run locally during each map task, all the mappers need to get the partition boundaries and possibly store them in an associative data structure in local memory for fast access. This brings the need to distribute the partition boundaries to each mapper task before the sorting job begins.

Hadoop has a special distributed cache designed exactly for this purpose. Distributed Cache is a service that copies files and archives (compressed files in zip, tar, gzip, jar formats) to the local drives of the task nodes. Typically, the files are only copied to each task node once per job, so multiple tasks running on the same node do not have to download the files again - they just read them from the local cache. At the end of the job, the local files in the cache may be automatically deleted depending on the available cache space left in the local drive. Map or reduce tasks can access to a file in the cache through its name without knowing the full path. This is accomplished by creating a symbolic link from the task's current working directory. Archives added to the distributed cache are automatically uncompressed before the task begins. This is especially useful when you have a lot of small files to be added to the distributed cache. You can compress them as a single archive and throw it into the cache, rather than worry about adding files one by one.

When a TeraSort job is submitted to the cluster, the input files contain- ing unsorted data are sampled through TeraInputFormat.writePartitionFile() and partition boundaries are written to a file on HDFS. Before the actual sorting task begins, the file containing the boundaries is added to the dis- tributed cache with DistributedCache.addCacheFile() and a symbolic link is created for the task nodes with a call to DistributedCache.createSymlink(). This ensures that the partition boundaries are locally available to all map tasks. These steps are performed inside TeraSort.run(), which is given in listing 1.2.6.

Listing 2.18 TeraSort

```
public class TeraSort extends Configured implements Tool {
  private static final Log LOG = LogFactory.getLog(TeraSort. class);
  public int run(String[] args) throws Exception {
    LOG.info("starting");
    JobConf job = (JobConf) getConf();
    Path inputDir = new Path(args[0]);
    inputDir = inputDir.makeQualified(inputDir.getFileSystem(job));
    Path partitionFile = new Path(inputDir, TeraInputFormat.PARTITION_FILENAME);
    URI partitionUri = new URI(partitionFile.toString() + "#" + TeraInputFormat.PART
    TeraInputFormat.setInputPaths(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    job.setJobName("TeraSort");
    job.setJarByClass(TeraSort.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(Text.class);
    job.setInputFormat(TeraInputFormat.class);
    job.setOutputFormat(TeraOutputFormat.class);
    job.setPartitionerClass(TotalOrderPartitioner.class);
    TeraInputFormat.writePartitionFile(job, partitionFile);
    DistributedCache.addCacheFile(partitionUri, job);
    DistributedCache.createSymlink(job);
    job.setInt("dfs.replication", 1);
    TeraOutputFormat.setFinalSync(job, true);
    JobClient.runJob(job);
    LOG.info("done");
    return 0;
  }
  public static void main(String[] args) throws Exception {
    int res = ToolRunner.run(new JobConf(), new TeraSort(), args);
    System.exit(res);
  }
}
```

2.1.12 Efficient Partition Discovery

Deciding which reducer a TeraSort record should be sent to requires com- paring it against the keys in partition boundaries which are computed and sorted by TeraInputformat.createPartitions(). Suppose we would like to lo- cate the correct partition for the key [6 - 4 - *] and have the following keys in partition boundaries:

Partition figure here

For simplicity, we only pay attention to the first two bytes and ignore the rest. There are eight partitions indexed from zero, each corresponding to a reduce task and is represented by an empty row.

A naive search algorithm would sequentially compare the record starting with the first key in partition boundaries and move forward until the correct partition is located. For the example above, this would require four compar- isons until we reach boundary3 - [6 - 9 - *], which is greater than [6 - 4 - *] so it goes to partition 3.

This algorithm has linear complexity with the number of partitions. When sorting 1 Terabyte of data, it is executed once per record - 10 billion times in total! Thus, although discovering the partitions seems like a very small part of the process, a considerable speed up can be achieved by implementing an efficient partition discovery algorithm which makes less comparisons. Hadoop's TeraSort implementation builds a two level tree to speed up the partition discovery process. A tree built up from the example partition boundaries above is shown in figure 1.2.7.

Tree figure here

Two kinds of TrieNodes make up a trie; InnerTrieNode and LeafTrieNode. An InnerTrieNode can be in level 0 or level 1 and must have 256 children - one for each byte. The root is an InnerTrieNode in level 0 and its children can be InnerTrieNodes or LeafTrieNodes. In the example figure, child0 and child1 of the root are LeafTrieNodes and child2 is an InnerTreeNode which further branches down to level 2.

A LeafTrieNode can be in level 1 or level 2 and has no children. It contains an interval [l,u] - a lower and upper bound that correspond to a range of partition boundaries to be compared against a record during partition discovery.

The partition discovery is a recursive algorithm having two key steps:

- Locating a leaf node based on at most the first two bytes.
- Comparing the key with all the partition boundaries that are in the [l,u] range of the corresponding leaf node.

Suppose we would like to determine the partitions for records [1 - 7 - *], [2 - 5 - *] and [2 - 3 - *] using the trie in the example figure.

The first byte of [1 - 7 - *] is 1 so we start from the root and observe its child1 which is a LeafTrieNode with interval [0,0]. We compare it against boundary0 - [2 - 3 - *] and see that it is smaller. Thus, [1 - 7 - *] goes to partition 0.

The first byte of [2 - 5 - *] is 2 so we start from the root and observe its child2 which is an InnerTrieNode. We go one level down and observe its child5 since the second byte of [2 - 5 - *] is 5. This is a LeafTreeNode with interval [1,1]. We compare it against boundary1 - [2 - 6 - *] and see that it is smaller. Thus, [2 - 5 - *] goes to partition 1.

The first byte of [2 - 3 - *] is 2 so we start from the root and observe its child2 which is an InnerTreeNode. We go one level down and observe its child3 since the second byte of [2 - 3 - *] is 3. This is a LeafTrieNode with interval [0,1]. We start comparing it against boundary0 - [2 - 3 - *]. If it is smaller, it goes to partition 0 else it goes to partition 1.

Observe how the trie structure cuts down the number of comparisons needed to locate a partition. Based on the initial two bytes, we locate a leaf node in at most two steps. Then, we compare the record only against a small subset of the partition boundaries within interval [l,u].

Listing 1.2.7 shows the code for the abstract class TreeNode.

Listing 2.19 TreeNode

```
/**
* A generic trie node
*/
static abstract class TrieNode {
  private int level;
  TreeNode(int level) {
    this.level = level;
  }
  abstract int findPartition(Text key);
  abstract void print(PrintStream strm) throws IOException;
  int getLevel() {
    return level;
  }
}
```

InnerTrieNode extends TrieNode and contains an array of 256 children, one for each byte. The constructor takes an integer parameter as the level of the node. The findPartition() method checks the first or the second byte of the key depending on the current level and makes a recursive call to a child node. The print() method prints the node and its child nodes recursively. This is given in listing 1.2.7.

Listing 2.20 InnerTreeNode

```
/**
* An inner tree node that contains 256 children based on the next
* character.
*/
static class InnerTreeNode extends TreeNode {
  private TreeNode[] child = new TreeNode[256];
  InnerTreeNode(int level) {
    super(level);
  }
  int findPartition(Text key) {
    int level = getLevel();
    if (key.getLength() <= level) {
      return child[0].findPartition(key);
    }
    return child[key.getBytes()[level]].findPartition(key);
  }
  void setChild(int idx, TreeNode child) {
    this.child[idx] = child;
  }
  void print(PrintStream strm) throws IOException {
    for(int ch=0; ch < 255; ++ch) {
      for(int i = 0; i < 2*getLevel(); ++i) {
        strm.print(’ ’);
      }
      strm.print(ch);
      strm.println(" ->");
      if (child[ch] != null) {
        child[ch].print(strm);
      }
    }
  }
}
```

LeafTreeNode extends TreeNode and contains an array of partition bound- aries as splitPoints. The constructor takes a level, a list of partition bound- aries, a lower and an upper bound as input parameters. The findPartition() method compares a given key against the partition boundaries in the [l,u] range. The print() method sequentially prints the range of partitions that the leaf node corresponds to. The code is given in listing 1.2.7.

Listing 2.21 LeafTreeNode

```
/**
* A leaf tree node that does string compares to figure out where the given
* key belongs between lower..upper.
*/
static class LeafTrieNode extends TreeNode {
  int lower;
  int upper;
  Text[] splitPoints;
  LeafTrieNode(int level, Text[] splitPoints, int lower, int upper) {
    super(level);
    this.splitPoints = splitPoints;
    this.lower = lower;
    this.upper = upper;
  }
  int findPartition(Text key) {
    for(int i=lower; i<upper; ++i) {
      if (splitPoints[i].compareTo(key) >= 0) {
        return i;
      }
    }
    return upper;
  }
  void print(PrintStream strm) throws IOException {
    for(int i = 0; i < 2*getLevel(); ++i) {
      strm.print(’ ’);
    }
    strm.print(lower);
    strm.print(", ");
    strm.println(upper);
  }
}
```

The TotalOrderPartitioner class contains two methods for contructing a trie. The method readPartitions() reads a sorted list partition boundaries from a given sequence file and returns them. The buildTree() method builds up a two level trie recursively using the partition boundaries as input. The getPartition() method takes a 10-byte key, 90-byte value as input and calls the findPartition() method of the root node to determine the correct partition.

Part of the code for TotalOrderPartitioner is given in listing 1.2.7.

Listing 2.22 TotalOrderPartitioner

```
static class TotalOrderPartitioner implements Partitioner<Text,Text>{
  private TreeNode tree;
  private Text[] splitPoints;
  ...
  public int getPartition(Text key, Text value, int numPartitions) {
   return tree.findPartition(key);
  }
}
```

2.1.13 I/O Optimizations

At the map stage, the size of the memory buffer to use while sorting the map output is controlled by io.sort.mb. The maximum number of streams to merge at once is controlled by io.sort.factor. Increasing these two parame- ters may give better sort throughput if you have enough physical memory.

Map output files stay in the local disk of the map task nodes and reduc- ers start fetching them from multiple mappers as soon as they are available. The map outputs are initially stored in the reducer task's memory, and later spilled to the disk and merged into larger files before the actual reduce func- tion is executed. There are a few parameters that control when to sort and write the buffer contents to the disk: mapred.job.shuffle.merge.percent determines a memory percentage threshold for the buffer. When the buffer exceeds this threshold, map outputs in the buffer are merged and written to a file on the reduce task's local disk. The threshold number of map output files to start merging and spilling to disk is controlled by mapred.inmem.merge. threshold. Setting this parameter to 0 means there is no threshold and the merging and spilling behaviour is only controlled by mapred.job.shuffle. merge.percent. The amount of memory allocated for the in-memory file system used to merge map outputs at the reduce step is controlled by fs.inmemory.size.mb. The percentage of the heap size to keep the map out- puts during the reduce stage is controlled by mapred.job.reduce.input. buffer.percent. By default, this parameter is set to 0 ensuring all map outputs are written to the reduce task's local disk before the reduce process starts.

The dafault behavior of Hadoop is to write the buffer contents to the disk whenever it reaches a threshold because it aims to leave as much memory space as possible for the actual reduce process. If the reduce task does not have enough memory space, it may result in out of heap exceptions and terminate unsuccessfully. These parameters should be controlled depending on the reduce task's memory requirements. If your reduce task does not need too much memory, increasing the thresholds and keeping the map outputs aggresively in memory may give good speed ups because your reducers will not spend time on writing the buffer into the disk and reading it back again.

When Hadoop won the TeraSort benchmarks in 2009, io.sort.mb, io.sort. factor, fs.inmemory.size.mb the heap size and the buffer thresholds were set to large enough values to ensure that intermediate output is never spilled to the disk except for the end of the map stage.

2.1.14 Concluding Remarks

In this chapter, we covered two sorting problems. Simulating secondary sort on values may be needed when the reducers need to keep track of all values as- sociated with a particular key - for example to eliminate duplicates. We saw how the partitioner and the grouping comparator works in a Hadoop job and how we can use different comparators at different stages of the computation to customize the sorting and grouping behavior. We also implemented basic WritableComparable objects having multiple fields and how they are serial- ized by the framework. We developed a custom comparator that efficiently compares two WritableComparable objects by interpreting their binary rep- resentations rather than deserializing them.

In the second section, we discussed how Hadoop's TeraSort implementa- tion works. We covered how to write a custom InputFormat, RecordReader and InputSplit. We saw how the total order partitioner divides the map out- puts into a globally ordered set of data blocks while efficiently discovering the partitions by utilizing a two level tree. We finally talked about some of the I/O optimizations to tune a Hadoop job by customizing configuration parameters.