Lab Lecture #4

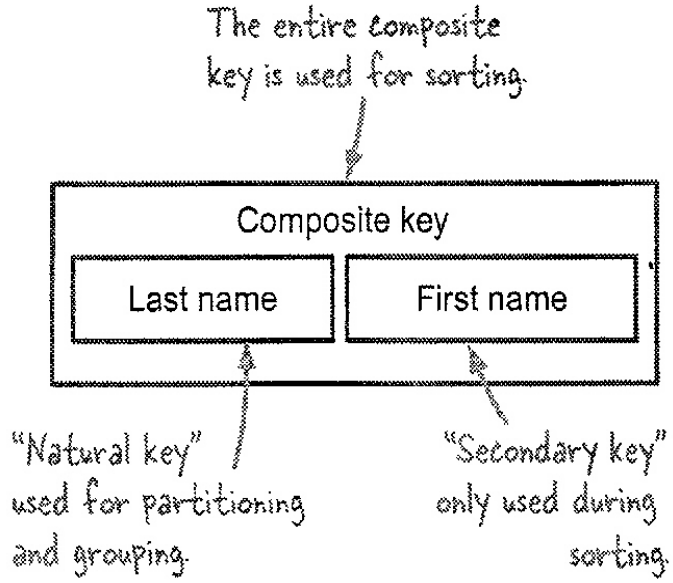
Introduction

Secondary sort is useful when we want some of the values for a unique map key to arrive at a reducer in a fixed order. Essentially, we want the reducer's values iterator to be sorted. The value of secondary sorting is fundamental for some MapReduce computations, such as the friends-of-friends alghorithm (we will see it in the next lecture). Hadoop's MapReduce platform sorts the keys, but not the values. (note: Google's MapReduce platform explicitly supports secondary sorting).

Problem statement

Suppose we have a file with a bunch of line separated people's name, and we want to sort them. We want our reducer to receive names partitioned by last name, and sorted by first name. This is actually somewhat difficult to do, since we want to partition by key, but sort the reducer's values iterator.
The trick is to write custom partitioner, sort comparator and grouping comparator classes, which are required for secondary sort to work, but also to have the mapper output a composite key, which will contain two parts:

1. the *natural key*, which is the key to use for joining purposes
2. the *secondary key*, which is the key to use to order all of the values sent to the reducer for the natural key



Composite key

The composite key contains both the first and the last name. It extends WritableComparable, which is recommended for Writable classes that are emitted as keys from map functions:

```
public class Person implements WritableComparable<Person> {

  private String firstName;
  private String lastName;

  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getLastName() {
    return lastName;
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }

  public void set(String lastName, String firstName) {
    this.lastName = lastName;
    this.firstName = firstName;
  }

  @Override
  public void readFields(DataInput in) throws IOException {
    this.lastName = in.readUTF();
    this.firstName = in.readUTF();
  }

  @Override
  public void write(DataOutput out) throws IOException {
    out.writeUTF(lastName);
    out.writeUTF(firstName);
  }

  @Override
  public int compareTo(Person other) {
    int cmp = this.lastName.compareTo(other.lastName);
    if (cmp != 0) {
      return cmp;
    }
    return this.firstName.compareTo(other.firstName);
  }
```

Partitioner

The partitioner has a single function that determines which partition (reducer) your map output should go to. The default MapReduce partitioner (HashPartitioner) calls the hashCode method of the output key and performs a modulo with the number of reducers to determine which reducer should receive the output. The default partitioner use the entire key, which won't work for our composite key, because it will likely send keys with the same natural key to different reducers. Instead, you need to write your own Partitioner, which partition on the natural key only.

The following code show the Partitioner we need to implement. The getPartition method is passed the key, value and the number of partitions:

```
public interface Partitioner<K2, V2> extends JobConfigurable {
  int getPartition(K2 key, V2 value, int numPartitions);
}
```

Our partitioner will calculate a hash based on the last name in the Person class, and perform a modulo of that with the number of partitions (which is the number of reducer):

```
public class PersonNamePartitioner extends Partitioner<Person, Text> {

  @Override
  public int getPartition(Person key, Text value, int numPartitions) {
    return Math.abs(key.getLastName().hashCode() * 127) %
        numPartitions;
  }
}
```

Sorting

Both the map and the reduce participating in sorting. The map-side sorting is an optimization to help the recuder sorting more efficiently. We want MapReduce to use the entire key for sorting purposes, which will order key according to both the last name and the first name.

In the following code we will see the implementation of the WritableComparator, which compares user based on their full name:

```
public class PersonComparator extends WritableComparator {
  protected PersonComparator() {
    super(Person.class, true);
  }

  @Override
  public int compare(WritableComparable w1, WritableComparable w2) {

    Person p1 = (Person) w1;
    Person p2 = (Person) w2;

    int cmp = p1.getLastName().compareTo(p2.getLastName());
    if (cmp != 0) {
      return cmp;
    }

    return p1.getFirstName().compareTo(p2.getFirstName());
  }
}
```

Grouping

Grouping occurs when the reduce phase is streaming map output records from local disk. Grouping is the process by which we can specify how records are combined to form one logical sequence of records for a reducer invocations.

When we're at grouping stage, all of the records are already in secondary-sort order, and the grouping comparator needs to bundle togheter records with the same last name:

```
public class PersonNameComparator extends WritableComparator {

  protected PersonNameComparator() {
    super(Person.class, true);
  }

  @Override
  public int compare(WritableComparable o1, WritableComparable o2) {

    Person p1 = (Person) o1;
    Person p2 = (Person) o2;

    return p1.getLastName().compareTo(p2.getLastName());
  }
}
```

Map and Reduce tasks

To complete the description of the solution, we need to write the map and the recuce code. The mapper creates the composite key and emits that in conjunction with the first name as the output value. The reducer produces output identical to the input:

```
public static class Map extends Mapper<Text, Text, Person, Text> {

	private Person outputKey = new Person();

	@Override
	protected void map(Text lastName, Text firstName, Context context)
			throws IOException, InterruptedException {
		outputKey.set(lastName.toString(), firstName.toString());
		context.write(outputKey, firstName);
	}
}

public static class Reduce extends Reducer<Person, Text, Text, Text> {

	Text lastName = new Text();

	@Override
	public void reduce(Person key, Iterable<Text> values,
		Context context)
			throws IOException, InterruptedException {
		lastName.set(key.getLastName());
		for (Text firstName : values) {
			context.write(lastName, firstName);
		}
	}
}
```

MapReduce job definition

The final step involve telling the MapReduce to use the partitioner, sort comparator and group comparator classes above definded, as well as the Map/Reducer output key/value type and the InputFormatClass:

```
public static void runSortJob(String input, String output)
            throws Exception {
      Configuration conf = new Configuration();

      Job job = new Job(conf);
      job.setJarByClass(SortMapReduce.class);

      job.setMapperClass(Map.class);
      job.setReducerClass(Reduce.class);

      job.setInputFormatClass(KeyValueTextInputFormat.class);

      job.setMapOutputKeyClass(Person.class);
      job.setMapOutputValueClass(Text.class);

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

      job.setPartitionerClass(PersonNamePartitioner.class);
      job.setSortComparatorClass(PersonComparator.class);
      job.setGroupingComparatorClass(PersonNameComparator.class);

      Path outputPath = new Path(output);

      FileInputFormat.setInputPaths(job, input);
      FileOutputFormat.setOutputPath(job, outputPath);

      outputPath.getFileSystem(conf).delete(outputPath, true);

      job.waitForCompletion(true);
}

public static void main(String... args) throws Exception {
      runSortJob(args[0], args[1]);
}
```