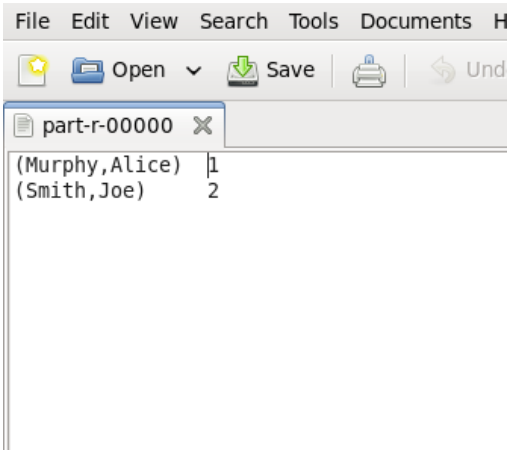
Problem 1: Custom WritableComparable (30%)

(a) Implement a custom WritableComparable type holding two strings.
See java code

(b) Implement a MAPREDUCE program counting the number of occurrences of each (full) name (i.e., first and last name) in a list of names provided as given in the test input above.
The output of test input is shown as below.



(c) Test your code using local job runner on the following input:~/training_materials/developer/data/nameyeartestdata. Provide the last 7 lines of the results file in your written submission.

```
(Smith,John)    3
(Turing,Alan)   1
(Wamsley,Jayme) 1
(Webre,Josh)    1
(Weston,Clark)  1
(Woodburn,Louis)        1
(Woodburn,Providencia)  1
```

Problem 2: Secondary Sort (30%)

(a) 1) Assume your MAPREDUCE job finishes successfully. What is a disadvantage of this approach?

The values are not sorted. The order in which the values appear is not even stable from one run to the next, because they come from different map tasks, which may finish at different times from run to run.

It cannot be executed parallelly on several compute nodes.

2) Now, let's consider the case where your MAPREDUCE job breaks. Why does your job fail?

If the values are not sorted in particular order, we have to iterate through all the values of a particular key to find the maximum or minimum, which requires a lot of memory space.

(b) 1) Because key-value pairs are assigned to several reducers by partitioner and get sorted independently.

2) The youngest person with last name Newton is Newton Candace.

```
Newton,1987)    Newton Candace 1987-Sep-05
```

(c) Modify your MAPREDUCE program incorporate the following changes. The output should be globally sorted. Modify the NameYearPartitioner to achieve this. See java codes.

(d) You want to output the oldest person instead of the youngest person?

We can run the program with out using custom sort comparator. Delete or comment the following codes:

```
job.setSortComparatorClass(NameYearComparator.class);
```

In that case, the key-value pairs will be sorted in default order, which is name ascending and year ascending.

(e) Why do we need to implement the group comparator in the secondary sort example discussed in Lab 4?

Because it determines which keys and values are passed in a single call to the Reducer. Pairs with the same name will be grouped into the same reduce call, regardless of the year.

Problem 3: Creating an Inverted Index (40%)

(a) Which InputFormat can be used to read this data directly as key value pairs? How can you retrieve the file name in a Mapper or Reducer?
We can use LongWritable as InputFormat to read data. We can retrieve file name from Context object by using getPath() and getName.
Respective codes:

```
Path filePath=((FileSplit)context.getInputSplit()).getPath();
//String filePathString=filePath.toString();
String fileName=filePath.getName();
```

(b) Implement an indexer that produces an index of all the words in the text.
See java codes.

(c) What is the Mapper output for the following input lines from Hamlet?
Mapper output: (have, hamlet@282); (heaven, hamlet@282); (and, hamlet@282); (earth, hamlet@282); (there, hamlet@133); (are, hamlet@133); (more, hamlet@133); (things, hamlet@133); (in, hamlet@133); (heaven, hamlet@133); (and, hamlet@133); (earth, hamlet@133);