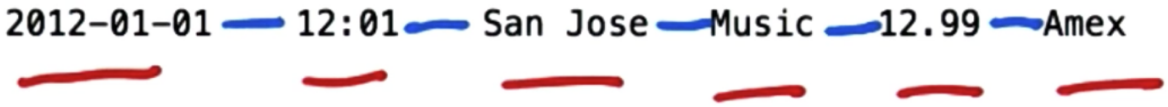


Lesson 3 Notes

Introduction

In this lesson we're going to take a look at the actual code we ran in the last lesson. Then you'll write your own MapReduce code.

If you recall, the code found the total sales per store. Typically, the data would have come from database tables and we'd use Sqoop to import it into HDFS.

Input Data

So first let's take a closer look at the input data format. Remember that each Mapper processes a portion of the input data, and each one will be given a line at a time. The lines look like this:

```
2012-01-01    12:01    San Jose    Music    12.99    Amex
```

The Mapper needs to take that line and extract the information it needs. Often when we're dealing with text it's pretty free-form so we'd use something like a regular expression. But in this case, it's nice and regular: it's tab delimited. So we can just split the line based on tabs and extract the values for all fields.

Quiz: How To Find Total Sales?

Question - You've been asked to calculate the total sales for each store. What should you use as the intermediate key and value?

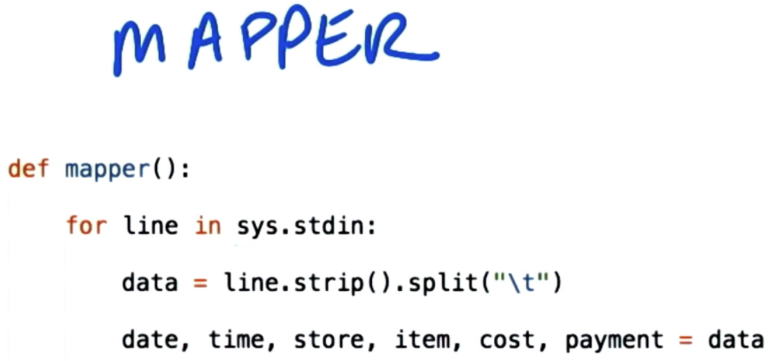
Key	Value
[ ] time	store name
[ ] cost	store name
[ ] store name	cost
[ ]store name	item description

Answer:

The store name and the amount is the correct answer.

Defensive Mapper Code

Here's our Mapper code. Let's look at it line by line.We're going to loop around standard input, which will give us a line at a time. Of course, the line will have a newline character at the end of it, so let's strip out that, plus any other white space around the line, and since our line is tab delimited, we can split it at the same time. That gives us an array, which we'll call data.
re



Programming Quiz: Finish The Mapper

As you use Hadoop more and more, you'll discover that the more data you have, the more likely you are to encounter weirdness in that data. Lines will be malformed, there will be strange log messages in the data… you're going to come across every strange edge case you can imagine, and plenty that you can't. So here, you should make sure that no matter what kind of malformed line the file has, the mapper can continue working. You wouldn't want your 2TB processing job to die part way through.

So, we'd like you to add some defensive programming to make sure things don't break if you get a strange line in the middle of your data.

Answer:

In this case, we're just checking that the line actually has six fields. If it doesn't, we'll just ignore that line. But another good thing to do would be to check that the cost is actually a valid number.

Assuming that the line is OK, we'll simply write our intermediate data out in the form of the key, then a tab, then the value. And then we loop back and read the next line from our input file.

Quiz: What Happens Between Mapper And Reducer?

Once the Mapper is done, the Hadoop framework passes the intermediate data to the Reducers. What's the process called that happens between Mappers and Reducers?

[ ] Bubble sort
[ ] Shuffle and sort
[ ] Find and Shuffle
[ ] Quick sort

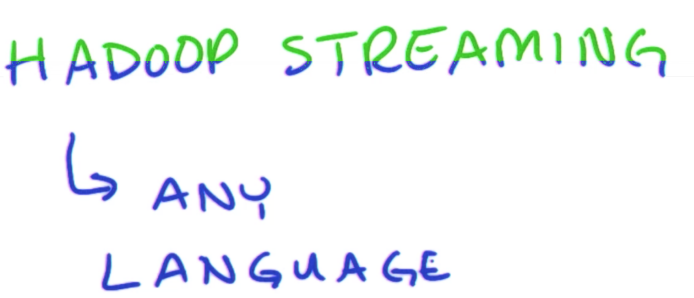
Answer:

The process is called the Shuffle and Sort. It ensures that the values for any particular key are collected together, and sends the keys and their lists of values to the Reducer.

Reducer

In our case, we only have a single Reducer, because that's the Hadoop default, so it will get all the keys. If we had specified more than one Reducer, each would receive some of the keys, along with all the values from all the Mappers for those keys.

```
Miami   12.34
Miami   99.07
Miami   3.14
NYC     99.77
NYC     88.99
```



We're using Hadoop Streaming here, because we're writing our code in Python. Hadoop Streaming allows you to write your Mappers and Reducers in pretty much any language, rather than forcing you to use Java.

But the way the Reducer gets the data is a little tricky to deal with. It's going to get the data coming in something like this.

Quiz: What Variables Do We Need To Keep Track Of?

As you can see, the data comes in as a stream of lines, each containing a store name and cost. The store names are sorted, which we're guaranteed because of the shuffle and sort, so we know that all the lines for, say, Miami will appear one after the other. So, what variables do we need to keep track of to calculate the sales per store, based on how the data is appearing?

[ ] previous sale
[ ] current sale
[ ] total sales per store
[ ] previous store
[ ] current store name
[ ] all store names

Answer:

So when we code the Reducer, we're going to need to keep track of the keys. When the key changes, we know we've received all the data from the previous key, so we can then write out the final result for that previous key -- and in our case, we'll have been adding up all the values for that key.

Reducer Code

So here's the Reducer code. Let's step through it.

We'll start by setting a couple of variables up. salesTotal is what we'll use to keep the running total. We initialize that to zero. And since we haven't read any data in yet, we haven't had any keys, so oldKey is initialized to None.

```
salesTotal = 0
oldKey = None
```

Then we start reading from standard input. Each line will contain a key, a tab, and a value. In our case, a store name, a tab, and one of the sales from that store.

So, we strip off the newline character the end of the line and split the line based on the tab. That should give us exactly two items, which we'll store in the 'data' array.

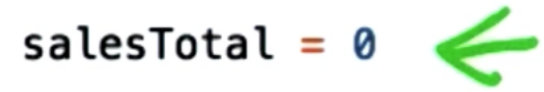
```
for line in sys.stdin:
    data = line.strip().split("\t")

    if len(data) != 2:
        continue
```

If we don't have two items, something strange has happened, so we'll skip that line of input -- although that should in fact never be the case, since we know our Mappers are writing the data out in this format.

Now we'll pull the two elements of the array out into named variables for clarity. thisKey will hold the store name, thisValue the sale amount.

Now here's the tricky part. We want to know if the key has changed since the last one we read. So we check to see if oldKey is even set -- because if it's not then this will be the first line we've read -- and, if it is, we see if it's different to the key we just read in.

If that's true, then the key has just changed -- in the example we just looked at, we'd have read all the Miami lines and now we've just received a New York key. So we need to write out the data for the previous key. We do that by writing that key, a tab, and the running total we've been keeping. That data will be written to a file in HDFS by the Hadoop framework.

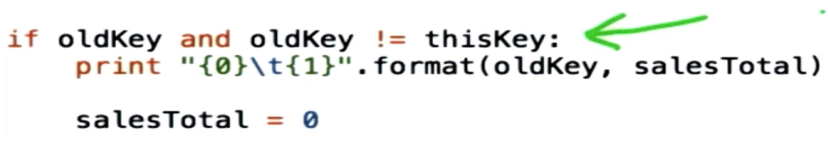



```
def reducer():

    salesTotal = 0
    oldKey = None

    for line in sys.stdin:
        data = line.strip().split("\t")

        if len(data) != 2:
            continue

        thisKey, thisSale = data

        if oldKey and oldKey != thisKey:
            print "{0}\t{1}".format(oldKey, salesTotal)

            salesTotal = 0

        oldKey = thisKey
        salesTotal += float(thisSale)
```

Once we've done that, we set the salesTotal back to zero since we're now dealing with a new store.

OK, now we can actually process the data we've just read. We set oldKey up with the contents of the key, and then add the value to our running total.

And then we loop back and do the whole thing again.



Eventually, we'll run out of data to process, which takes us out of the loop.

Question: Are We Done?

Do you think the Reducer is finished at this point?

[ ] Yes, it's finished
[ ] No, another process needs to be run on the output
[ ] No, the last key has not yet been output

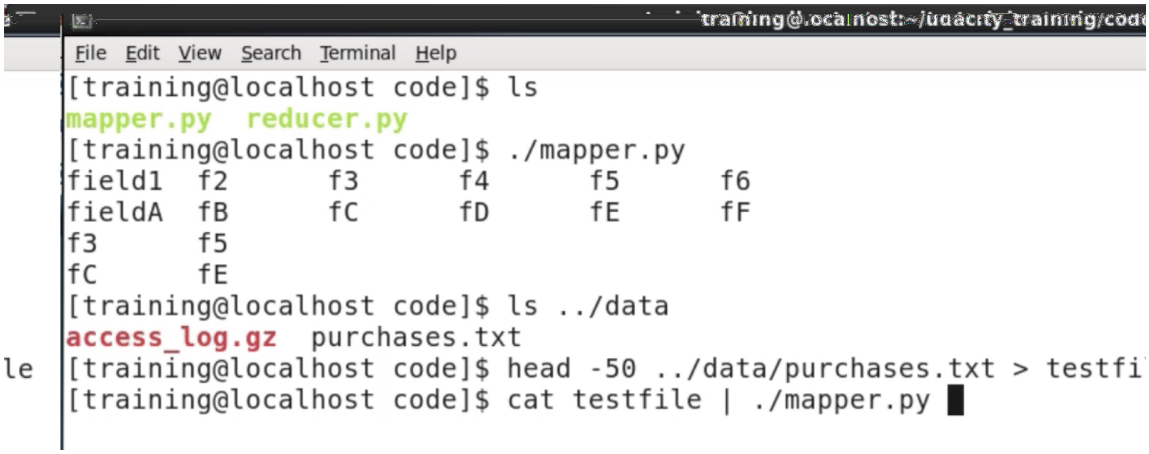
Answer:

Be careful! When we exit the loop, we haven't yet output the data for the last key we've been tracking. That's what the last two lines are for; we write out the key and value for the last store we've processed. If we didn't have those lines, we wouldn't write out data for that last store.

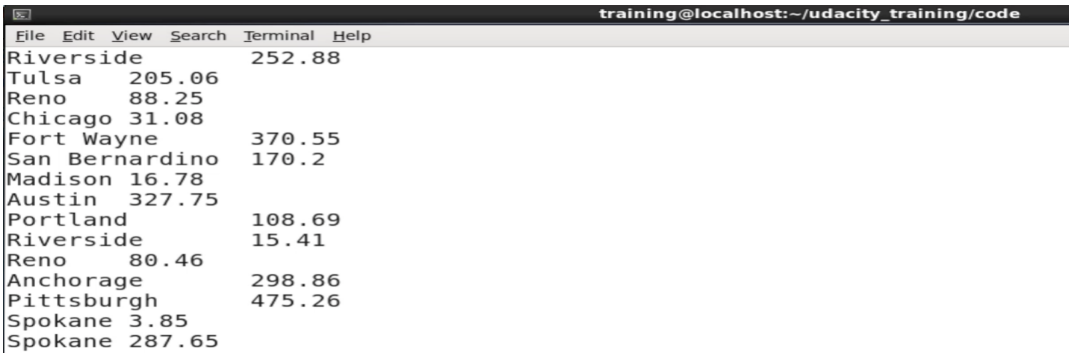
Now let's have Ian talk a little about testing the code, and then actually run it on our cluster.

Putting It All Together

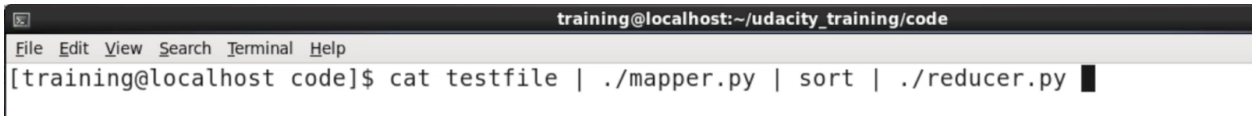
So that's the code. One of the nice things about using Hadoop streaming is that it's easy to test our code outside of Hadoop. Let's see how to do that. Our Mapper takes data in from standard input, and writes its results to standard output, so we can just run it from the command line and type data in to test it. Or, even better, we can build just a small sample data file and pipe that to the Mapper. Let's do that. Here we have a very small file -- just 10 or so lines. So to test the Mapper, we can just do this: cat testfile | ./mapper.py



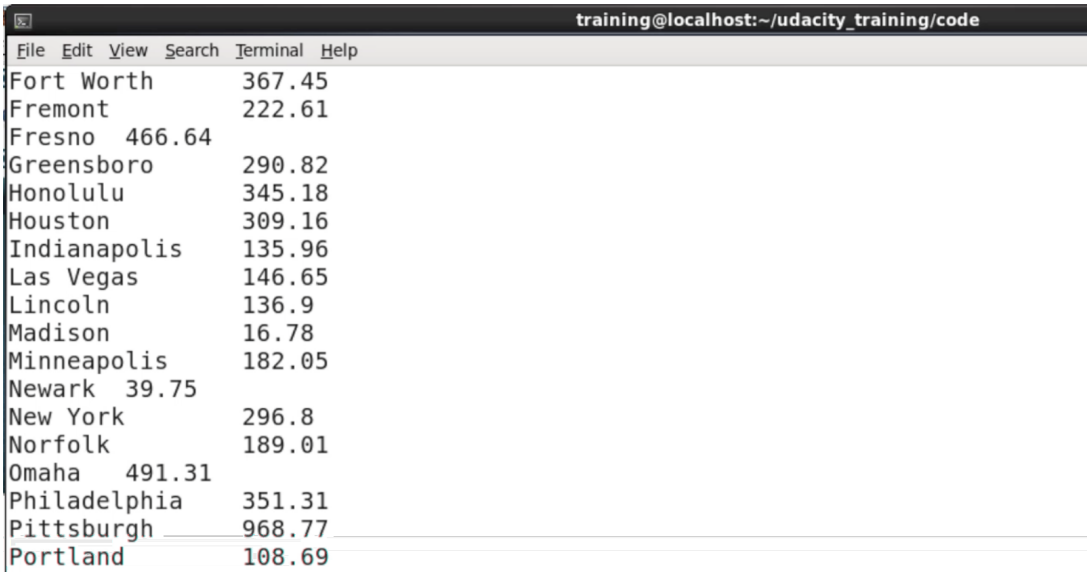
And there's our result -- store names and sales. Excellent! If we had problems, we could go back and edit the Mapper until it worked, and it's really nice and quick to do this without needing to run it via Hadoop every time.



We can do a similar thing with the Reducer. It's expecting a set of lines which look like storename tab value so, again, we can create a sample file which looks like that and pass it in. But even nicer, we can test the entire pipeline. Remember that the Mapper's output is sorted by the Hadoop framework and then passed to the Reducer. So we can simulate the entire thing on the command line like this, using the Unix sort utility in between the Mapper and Reducer:



...and there's our output, exactly as we'd expect. So now that we've tested on the command line, we can now test it on the cluster.Best practice when you're developing MapReduce jobs is to first test with a small data set before you run your code on your entire, huge set of data, but we're pretty confident here so let's just run the thing on our whole purchases.txt file.



We'll use our alias 'hs' to cut down on our typing. Here's our Mapper... and our Reducer… and the -files arguments we also have to specify. Our input directory is myinput, and we'll tell Hadoop to write the results to output2. Remember, the output directory must not already exist or the job will fail.

Off it goes. On this pseudo-distributed cluster we can only run two Mappers simultaneously, and it turns out that we need four to process the entire data set because of its size. So two will run, then when they've finished the next two will start. Once they're done the Reducers will then begin. It turns out, you can watch this happening via a Web-based user interface that Hadoop gives us. You point your Web browser at the JobTracker, which on our machine is just 'localhost', on port 50030. Here you can see that there's one running job, and when we click on it we can see the Mappers and Reducers running.

It gives us a ton of other interesting information; one key thing is that if a Mapper or Reducer fails, you can actually drill down and view the logs from that particular piece of code.

OK, it's done. So let's take a look at the output… and there it is. Our sales, totalled by store, just as we'd expected.