Hadoop Streaming Intro

The way you ordinarily run a map-reduce is to write a java program with at least three parts.

1. A Main method which configures the job, and lauches it
   * set # reducers
   * set mapper and reducer classes
   * set partitioner
   * set other hadoop configurations
2. A Mapper Class
   * takes K,V inputs, writes K,V outputs
3. A Reducer Class
   * takes K, Iterator[V] inputs, and writes K,V outputs

Hadoop Streaming is actually just a java library that implements these things, but instead of actually doing anything, it pipes data to scripts. By doing so, it provides an API for other languages:

* read from STDIN
* write to STDOUT

Streaming has some (configurable) conventions that allow it to understand the data returned. Most importantly, it assumes that Keys and Values are separated by a \t. This is important for the rest of the map reduce pipeline to work properly (partitioning and sorting). To understand why [**check out my intro to Hadoop**](https://blog.matthewrathbone.com/2013/04/17/what-is-hadoop.html), where I discuss the pipeline in detail.

**def** map(line):

fields = line.split(",")

**print**(fields.isArtificial, 1)

**def** reduce(isArtificial, totals):

**print**(isArtificial, sum(totals))

reduce('TRUE', Iterator(1, 1, 1, 1))

reduce('FALSE', Iterator(1, 1, 1))

To follow along, check out [**my git repository**](https://github.com/rathboma/hadoop-framework-examples) (on the virtual machine):

cd ~/workspace

git clone https://github.com/rathboma/hadoop-framework-examples.git

cd hadoop-framework-examples

### MAPPER

**import** **sys**

**for** line **in** sys.stdin:

line = line.strip()

unpacked = line.split(",")

stadium, capacity, expanded, location, surface, turf, team, opened, weather, roof, elevation = line.split(",")

results = [turf, "1"]

**print**("**\t**".join(results))

### REDUCER

**import** **sys**

*# Example input (ordered by key)*

*# FALSE 1*

*# FALSE 1*

*# TRUE 1*

*# TRUE 1*

*# UNKNOWN 1*

*# UNKNOWN 1*

*# keys come grouped together*

*# so we need to keep track of state a little bit*

*# thus when the key changes (turf), we need to reset*

*# our counter, and write out the count we've accumulated*

last\_turf = None

turf\_count = 0

**for** line **in** sys.stdin:

line = line.strip()

turf, count = line.split("**\t**")

count = int(count)

*# if this is the first iteration*

**if** **not** last\_turf:

last\_turf = turf

*# if they're the same, log it*

**if** turf == last\_turf:

turf\_count += count

**else**:

*# state change (previous line was k=x, this line is k=y)*

result = [last\_turf, turf\_count]

**print**("**\t**".join(str(v) **for** v **in** result))

last\_turf = turf

turf\_count = 1

*# this is to catch the final counts after all records have been received.*

**print**("**\t**".join(str(v) **for** v **in** [last\_turf, turf\_count]))

I have two datasets:

1. Users (id, email, language, location)
2. Transactions (transaction-id, product-id, user-id, purchase-amount, item-description)

Given these datasets, I want to find the number of unique locations in which each product has been sold. To do that, I need to join the two datasets together.

Previously I have implemented this solution [**in java**](https://blog.matthewrathbone.com/2013/02/09/real-world-hadoop-implementing-a-left-outer-join-in-hadoop-map-reduce.html), [**with hive**](https://blog.matthewrathbone.com/2013/02/20/real-world-hadoop---implementing-a-left-outer-join-in-hive.html) and [**with pig**](https://blog.matthewrathbone.com/2013/04/07/real-world-hadoop---implementing-a-left-outer-join-in-pig.html). The java solution was ~500 lines of code, hive and pig were like ~20 lines tops.

My [**beginners guide to python MapReduce**](https://blog.matthewrathbone.com/2013/11/17/python-map-reduce-on-hadoop-a-beginners-tutorial.html) does not solve this problem, but provides a more gentle introduction to running MapReduce with Python. Start there if you’re just getting started with these concepts.

The Python Solution

This solution assumes some preliminary understanding of hadoop-streaming and python, and uses concepts introduced in my [**earlier article**](https://blog.matthewrathbone.com/2013/11/17/python-map-reduce-on-hadoop-a-beginners-tutorial.html).

Demonstration Data

As in previous articles ([**java MR**](https://blog.matthewrathbone.com/2013/02/09/real-world-hadoop-implementing-a-left-outer-join-in-hadoop-map-reduce.html), [**hive**](https://blog.matthewrathbone.com/2013/02/20/real-world-hadoop---implementing-a-left-outer-join-in-hive.html) and [**pig**](https://blog.matthewrathbone.com/2013/04/07/real-world-hadoop---implementing-a-left-outer-join-in-pig.html)) we use two datasets called users and transactions.

**>** cat users

1 matthew@test.com EN US

2 matthew@test2.com EN GB

3 matthew@test3.com FR FR

and

**>** cat transactions

1 1 1 300 a jumper

2 1 2 300 a jumper

3 1 2 300 a jumper

4 2 3 100 a rubber chicken

5 1 3 300 a jumper

One big difference with Python MapReduce is that we treat them **as a single dataset** when we are writing our Mapper. I will show you how just below.

To start, let’s upload these files to HDFS.

hdfs dfs -mkdir users

hdfs dfs -mkdir transactions

hdfs dfs -put ./users.txt users

hdfs dfs -put ./transactions.txt transactions

Code

This job logically has two parts, so I will divide the code in the same way. Firstly we solve the problem of joining the two datasets to associate a location to each purchase, and secondly we use this joined dataset to evaluate how many unique locations each product has been sold in.

The code for the both parts of the solution and data used in this post can be found in my [**GitHub repository**](https://github.com/rathboma/hadoop-framework-examples/).

**PART 1: JOINING**

Mapper:

**import** **sys**

**for** line **in** sys.stdin:

*# Setting some defaults*

user\_id = ""

product\_id = "-"

location = "-"

line = line.strip()

splits = line.split("**\t**")

**if** len(splits) == 5: *# Transactions have more columns than users*

user\_id = splits[2]

product\_id = splits[1]

**else**:

user\_id = splits[0]

location = splits[3]

**print** '**%**s**\t%**s**\t%**s' % (user\_id,product\_id,location)

Reducer:

*#!/usr/bin/env python*

**import** **sys**

**import** **string**

last\_user\_id = None

cur\_location = "-"

**for** line **in** sys.stdin:

line = line.strip()

user\_id,product\_id,location = line.split("**\t**")

**if** **not** last\_user\_id **or** last\_user\_id != user\_id:

last\_user\_id = user\_id

cur\_location = location

**elif** user\_id == last\_user\_id:

location = cur\_location

**print** '**%**s**\t%**s' % (product\_id,location)

The Mapper reads both datasets and distinguishes them by the number of fields in each row. Transaction records have 5 fields, users have only 4.

The mapper does two things:

* For transactions - Extract the user\_id and product\_id
* For users - Extract the user\_id and the location

The mapper outputs three fields: user\_id, product\_id, location.

The output will look something like this:

*# From transaction data:*

2, 1, -

*# From user data:*

2, -, US

By using a feature of the streaming api we can tell Hadoop to treat BOTH of the first two fields as a combined key. This allows us to guarantee the order in which the reducer will recieve data:

1. User record with location (now we can remember the location)
2. Each user purchase in turn, ordered by product id.

We do this by specifying an option on the command line: -Dstream.num.map.output.key.fields=2. If we want to test this without Hadoop we can just use sort.

cat **\***.txt | ./joinMapperTU.py | sort

The output will look like this (I added notes):

1 - US *# user record*

1 1 - *# transaction record*

2 - GB *# user record*

2 1 - *# transaction record*

2 1 - *# transaction record*

3 - FR *# user record*

3 1 - *# transaction record*

3 2 - *# transaction record*

For each new user the Reducer will first remember that user’s location:

**if** **not** last\_user\_id **or** last\_user\_id != user\_id: *# if this is a new user*

last\_user\_id = user\_id

cur\_location = location

and then add this location to the transactions:

**elif** user\_id == last\_user\_id:

location = cur\_location

**print** '**%**s**\t%**s' % (product\_id,location)

So the reducer will take an input that looks like this (user\_id, product\_id, location):

3 - FR

3 1 -

3 2 -

Extract the location, and associate with each product id to produce this:

1 FR

2 FR

We can run the whole join pipeline easily without using Hadoop:

cat **\***.txt | ./joinMapperTU.py | sort | ./joinReducerTU.py | sort

And get a list of product/location pairs for stage 2. This shows the location of the purchaser (user) for each transaction, where the key is the product ID. Products are repeated the number of times that it appeared in a transaction.

1 FR

1 GB

1 GB

1 US

2 FR

Stage 2: Counting Distinct Locations for each Product

Mapper:

In fact we can just use cat here if we like.

*#!/usr/bin/env python*

**import** **sys**

**import** **string**

**for** line **in** sys.stdin:

line = line.strip()

product\_id,location = line.split("**\t**")

**print** '**%**s**\t%**s' % (product\_id,location)

Reducer:

*#!/usr/bin/env python*

**import** **sys**

**import** **string**

last\_product\_id = None

cur\_location = ""

count\_locations=0

**for** line **in** sys.stdin:

line = line.strip()

product\_id,location = line.split("**\t**")

*# if this is the first iteration*

**if** **not** last\_product\_id:

last\_product\_id = product\_id

cur\_location = location

count\_locations = 1

**if** product\_id == last\_product\_id:

**if** location != cur\_location:

count\_locations = count\_locations + 1

cur\_location = location

*# else we're transitioning from the last product to a new one*

**else**:

**print** '**%**s**\t%**s' % (last\_product\_id,count\_locations)

last\_product\_id = product\_id

cur\_location = location

count\_locations = 1

*# finally print out the last product / location combo. This is a gotcha! Easy to forget this line :-)*

**print** '**%**s**\t%**s' % (product\_id,count\_locations)

Our mapper just echos it’s input and the bulk of work happens in the reducer. In the Reducer phase we again exploit the fact that entries are ordered by key. Notice that unlike regular MapReduce this reducer’s API does not distinguish between keys and receives all of them in a big long list, so our reducer has to do it’s own bookeeping. See my [**beginners article**](https://blog.matthewrathbone.com/2013/11/17/python-map-reduce-on-hadoop-a-beginners-tutorial.html) for more of an explaination.

So we go through the list and count the number of locations we see for each product\_id, whilst making sure we transition between products properly.

Running the code

Again, this is easy to test without Hadoop:

cat **\***.txt | ./joinMapperTU.py | sort | ./joinReducerTU.py | sort | ./joinMapperTU1.py | sort | ./joinReducerTU1.py

Or using hadoop-streaming in two steps:

bin/hadoop jar ./contrib/streaming/hadoop-0.20.2-streaming.jar -Dmapred.reduce.tasks=1 -Dstream.num.map.output.key.fields=2 -input transactions -input users -output transactions\_and\_users\_output -file /path/to/joinMapperTU.py -file /path/to/joinReducerTU.py -mapper joinMapperTU.py -reducer joinReducerTU.py

bin/hadoop jar ./contrib/streaming/hadoop-0.20.2-streaming.jar -Dmapred.reduce.tasks=1 -Dstream.num.map.output.key.fields=2 -input transactions\_and\_users\_output -output transactions\_and\_users\_output\_final -file /path/to/joinMapperTU1.py -file /path/to/joinReducerTU1.py -mapper joinMapperTU1.py -reducer joinReducerTU1.py

The result in both cases is correct:

1 3

2 1