CS61A Lab 14: MapReduce

Week 15, 2012

To get the files necessary for this lab (such as `mr.py`), issue the following Unix command:

```
# cp -r ~cs61a/lib/lab/lab14 .
```

For this lab, we'll be using MapReduce, a programming paradigm developed by Google, which allows a programmer to process large amounts of data in parallel on many computers. Hadoop is an opensource implementation of the MapReduce design.

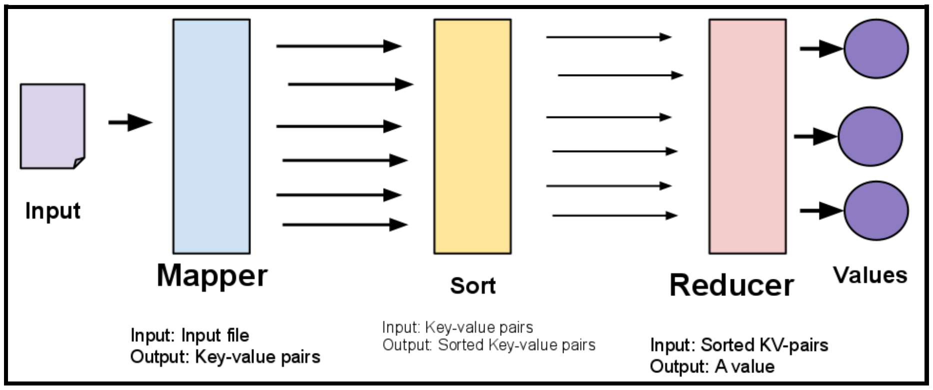
Any computation in mapreduce consists primarily of two components: the **mapper** and the **reducer**.

The **mapper** takes an input file, and outputs a series of key-value pairs, like:

```
hello 5
this 4
is 2
a 1
mapper-result 13
```

The **reducer** takes the (sorted) output from the mapper, and outputs a single value for each key. The mapper's output will be sorted according to the key.

The entire MapReduce pipeline can be summarized by the following diagram:



The MapReduce Pipeline

Your job will be to write the **mapper** and the **reducer**.

Example: Line-counting with Shakespeare and Unix

Let's see an example of using the MapReduce idea to count the number of lines in Shakespear's works.
On our servers, we happen to have all of Shakespeare's plays in text format. For instance, if you're so inclined, feel free to read a few phrases from 'Romeo and Juliet':

```
# emacs /home/ff/cs61a/lib/shakespeare/romeo_and_juliet.txt &
```

Or how about...'The Tempest'?

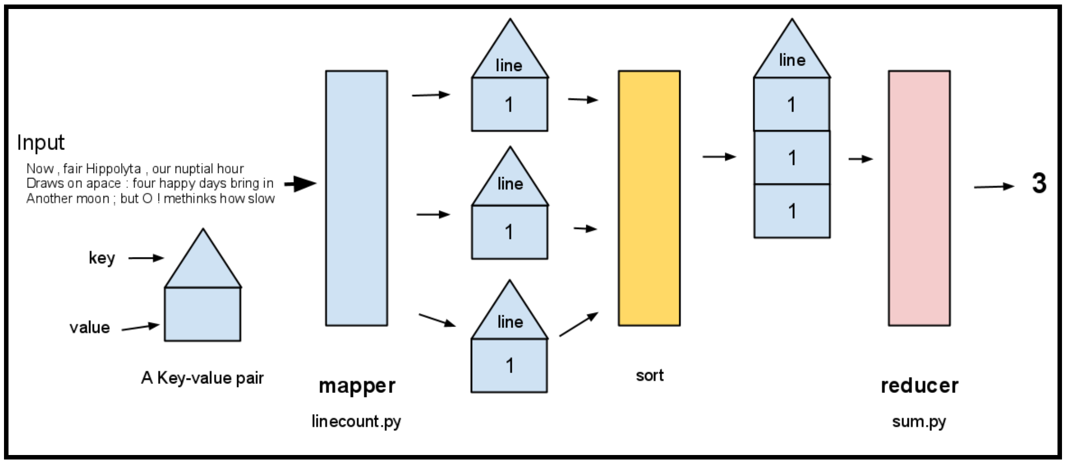
```
# emacs /home/ff/cs61a/lib/shakespeare/the_tempest.txt &
```

Anyways, we'd like to be able to count all the lines in all of his plays. Choose a Shakespeare play (say, `the_tempest.txt`) and copy it into your current directory by doing:

```
# cp ~cs61a/lib/shakespeare/the_tempest.txt .
```

In order to formulate this as a MapReduce problem, we need to define an appropriate `mapper` and `reducer` function.
One way to do this is to have the mapper create a key-value pair for every line in each play, whose key is always the word 'line', and whose value is always 1.

The reducer would then simply be a simple sum of all the values, as this picture illustrates:


Input
Now , fair Hippolyta , our nuptial hour
Draws on apace : four happy days bring in
Another moon ; but O ! methinks how slow
key
value
A Key-value pair
mapper
linecount.py
line
1
sort
reducer
sum.py
3

Line-counting in MapReduce

Let's implement each feature (mapper, reducer) separately, then see how each piece fits together.

The Mapper: line_count.py

Create a new file `line_count.py`, and fill it in with the following body:

line_count.py

```
#!/usr/bin/env python3

import sys
from ucb import main
from mr import emit

@main
def run():
    for line in sys.stdin:
        emit('line', 1)
```

`line_count.py` is the mapper, which takes input from `stdin` (i.e. 'standard in') and outputs one key-value pair for each line to standard out (i.e. to the terminal output). Let's try running `line_count.py` by feeding it `the_tempest.txt`. The question is, how do we give `the_tempest.txt` to `line_count.py` via `stdin`? We'll use the Unix pipe '`|`' feature (Note: the 'pipe' key isn't lowercase L, it's (typically) Shift+Backspace):

```
 # cat the_tempest.txt | ./line_count.py
```

Note: You might have to tell Unix to run `line_count.py` as an executable by issuing the following command:

```
 # chmod +x line_count.py
```

If you've completed `line_count.py` correctly, your terminal output should be full of key-value pairs, looking something like:

```
'line' 1
'line' 1
'line' 1
...
'line' 1
```

What pipe-ing does in Unix is take the output of one program (in this case, the `cat` program), and 'pipe' it into the input to another program (typically via standard in). This technique of pipe-ing programs together is ubiquitous in Unix-style programming, and is a sign of modular programming. The idea is: if you can write modular programs, then it will be easy to accomplish tasks by chaining together multiple programs. We'll do more with this idea in a moment.

The Reducer: sum.py

Create a new file `sum.py` in your current directory, and fill it in with the following body:

sum.py

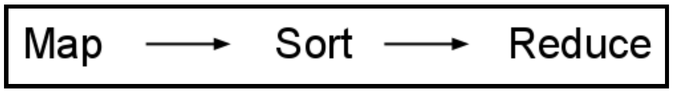
```
#!/usr/bin/env python3

import sys
from ucb import main
from mr import values_by_key, emit
```

```
@main
def run():
    for key, value_iterator in values_by_key(sys.stdin):
        emit(key, sum(value_iterator))
```

This is the reducer, which reads in sorted key-value pairs from `stdin`, and outputs a single value for each key. In this case, `sum.py` will return the `sum` of all the values for a given key. In other words, the reducer is *reducing* the values of a given key into a single value.
For the purposes of this simple line-counter, since the **mapper** only returns one type of key ('`line`'), the **reducer** will also only return one value - basically the total number of key-value pairs.

Putting it all together

Now that we have the **mapper** and the **reducer** defined, let's put it all together in the MapReduce framework:



The MapReduce Flow

To do this, issue the following Unix command:

```
 # cat the_tempest.txt | ./line_count.py | sort | ./sum.py
```

Note: You might have to tell Unix to run `sum.py` as an executable by issuing the following command:

```
 # chmod +x sum.py
```

Notice that we're using the Unix program `sort`, which is a 'built-in' Unix program. Take a moment and make sure you understand how the above Unix command is exactly the MapReduce flow. What's neat is that, in a very simple manner, we executed the MapReduce idea of using mappers and reducers to solve a problem. However, the main benefit of using the MapReduce idea is to take advantage of distributed computing - don't worry, we'll do that soon!

Exercises

Question 1

Use the MapReduce flow (i.e. Map -> Sort -> Reduce) to count the number of times the following (common) words occur:

```
the
he
she
it
thee
```

A question to ponder is: will you need to create a new mapper, a new reducer, or both?

MapReduce with Hadoop

We have provided a way to practice making calls to the MapReduce framework (using the Hadoop implementation).

cat

```
 #  python3 mr.py cat OUTPUT_DIR
```

This command prints out the contents of all files in one of the directories on the Hadoop FileSystem owned by you (given by `OUTPUT_DIR`).

ls

```
 # python3 mr.py ls
```

This command lists the contents of all output directories on the Hadoop FileSystem.

rm

```
 # python3 mr.py rm OUTPUT_DIR
```

This command will remove an output directory (and all files within it) on the Hadoop FileSystem. Use this with caution - remember, there's no 'undo'!

run

```
 # python3 mr.py run MAPPER REDUCER INPUT_DIR OUTPUT_DIR
```

This command will run a MapReduce job of your choosing, where:

`MAPPER` -- a `.py` file that contains the mapper function, i.e. `line_count.py`
`REDUCER` -- a `.py` file that contains the reducer function, i.e. `sum.py`
`INPUT_DIR` -- the input file, i.e. `../cs61a/shakespeare`
`OUTPUT_DIR` -- the name of the directory where you would like the results of the MapReduce job to be dumped into, i.e. `myjob1`

Example: Line-counting with Hadoop

In order to use the distributed-computing power, you'll need to SSH into the `icluster` servers (hadoop is installed only on these machines). To do this, issue the following terminal command:

```
# ssh icluster1.eecs.berkeley.edu -X
```

You will be prompted to log in.

Then, some environment variables need to be set - issue the following Unix command:

```
# eval `/home/ff/cs61a/projects/hadoop/bin/setup.sh -s`
```

Note: For some reason, in order to run some of the commonly-used Unix commands on `icluster1` (such has `mv`, `rm`, etc), you need to fully specify the filepath of the program, like:

```
# /bin/mv myfile.py newmyfile.py
```

Now, make sure that your `line_count.py`, `sum.py`, and `mr.py` are in the current directory, then issue the command:

```
# python3 mr.py run line_count.py sum.py ../cs61a/shakespeare mylinecount
```

Your terminal should then be flooded with the busy output of Hadoop doing it's thing. Once it's finished, you'll want to examine the Hadoop results! To do this, first call `mr.py`'s `ls` command to see the contents of your Hadoop directory:

```
 # python3 mr.py ls
```

You should see a directory listing for your `mylinecount` job. To view the results of this job, we'll use `mr.py`'s `cat` command:

```
 # python3 mr.py cat mylinecount
```

Exercises

Question 2:

Take your solution from **Question 1** and run it through the distributed MapReduce (i.e. by using `mr.py`) in order to discover the number of occurrences of the following words in the entirety of Shakespeare's works:

```
the
he
she
it
thee
```

Question 3a:

One common MapReduce application is a distributed word count. Given a large body of text, such as the works of Shakespeare, we want to find out which words are the most common.
Write a mapreduce program that returns each word in a body of text paired with the number of times it is used. For example, calling your solution with

`../cs61a/shakespeare` should output something like:

```
the 300
was 249
thee 132
...
```

Note: These aren't the actual numbers.

Question 3b:

Using the output of the program you wrote in part (a), return the most commonly used word.

Question 3c:

Now, create a file that contains all of the words used only once, one word per line, in sorted order. You'll probably want to use a combination of your mapreduce output from **3.a** and Unix pipe-ing.

Question 4a:

Write a MapReduce that, given a short list of phrases, lists the documents and lines containing each phrase.
Then, use your solution to find all instances of the following phrases originally attributed to Shakespeare:

```
pomp and circumstance                              foregone conclusion
full circle                                       the makings of
method in the madness                              neither rhyme nor reason
one fell swoop                                      seen better days
it smells to heaven                                a sorry sight
spotless reputation                               strange bedfellows
```

Hint: You'll want to use the provided `get_file()` function in your mapper. `get_file()` returns the name of the file it is currently mapping over - for `../cs61a/shakespeare`, the filenames will be play names.

Also, you might want to look at the included `set.py` reducer which reduces the values of each key into a `set` (i.e. removing duplicates).

Fin.