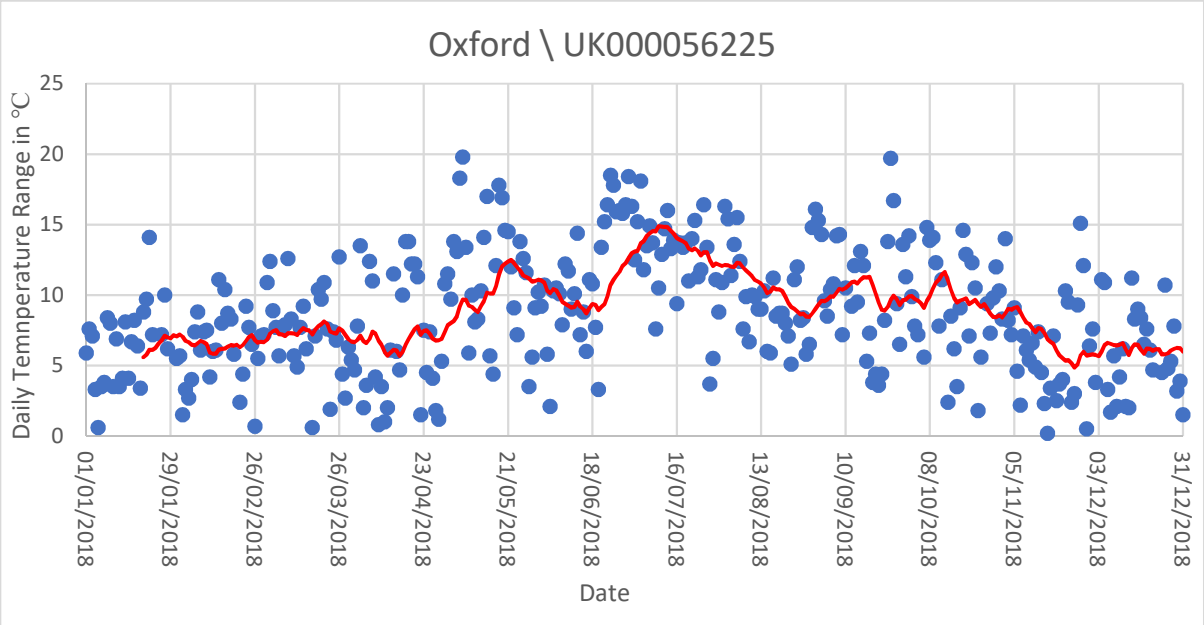
Cloud Computing Report

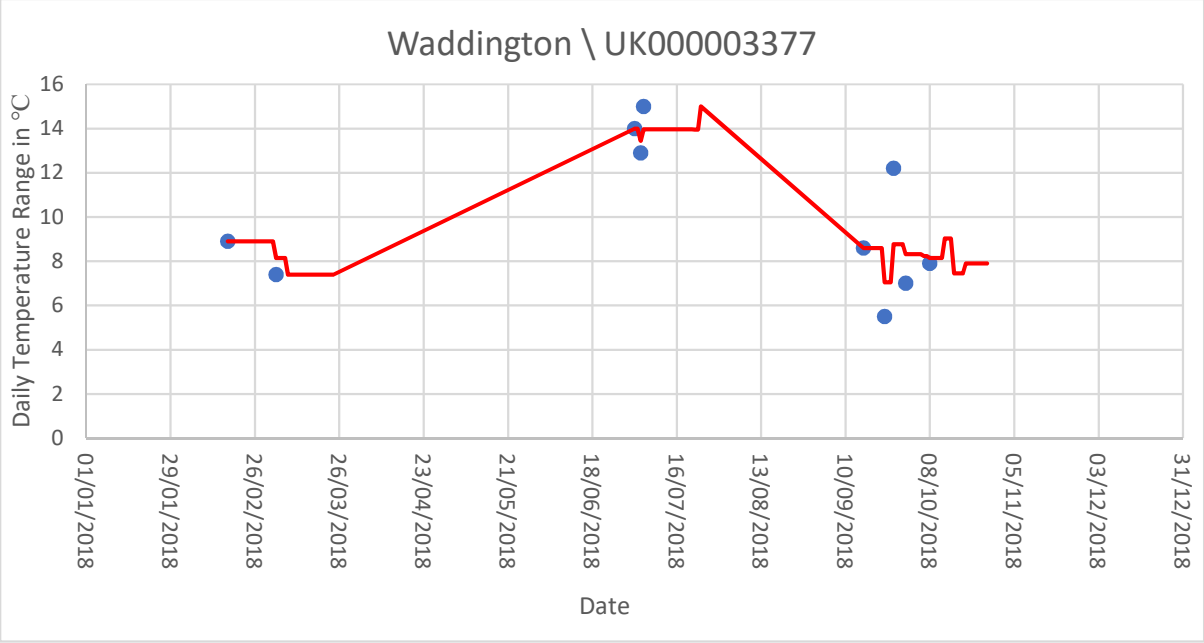
Section 1 - Google Storage paths to input and output directories

Input: gs://614743bucket1/2018.csv
Output: gs://614743bucket1/output-final/

NB: Results from location UK000056225 (Oxford) are in folder a. Results from location UK000003377 (Waddington) are in folder b.

Section 2 - Plots of your result





Section 3 - Challenges I've faced during the development of my MapReduce algorithm and how I have overcome these challenges.

Challenge 1 - Two locations

The specification requires us to process the data from two separate locations (Oxford and Waddington) and produce a one-column CSV file for each location. There a few challenges this creates.

Producing two files

Using the default apache-hadoop program on Google Cloud, the process will create as many files as there are reducers, which is usually determined by the size of your VM cluster. However, the way the data is split between these files is, at least by default, random, and not particularly systematic.

We would technically be able to process the outputted data into two files manually, or with the help of a script, if we know which location the data was for. This could be done with something as simple as a single character indicating either 'a' or 'b'. This would mean there would be some amount of processing done after Hadoop has run, which is not ideal.

Data mixing up between mapper/reducer

Using the default Hadoop-streaming program, the output from the mapper is sorted using the key value, and then fed into the reducer. If there were data from two locations, the key would need to be made in such a way so that the two locations' data are distinct and can be handled separately.

Solution

Whilst it isn't the ideal solution, in order to overcome these challenges, I decided to run Hadoop twice. Each run would target a different location, which means that there is no danger of the data becoming mixed up, and there is no need for extra processing for separating the two locations' data, either within the Hadoop process or afterwards. I also specified that I wanted one reducer to run, which produces only one file at the end of the process. Therefore, after running Hadoop twice, I am left with two files, one for each location.

Challenge 2 - Missing Data

When approaching this coursework, and testing my mapper and reducer on both locations, I discovered that for the second location, Waddington, there was a substantial amount of data missing. We are required to find the difference between the minimum and maximum temperatures for each day.

With regards to Waddington, within the dataset, there were 183 values for minimum temperatures, and only 22 values for maximum temperatures. In order to have a complete set of data, we want 365 values, one for each day. Furthermore, as we need to find the temperature difference, we need to have both maximum and minimum for each day, so that the difference can be calculated. With the available data, there are only 10 days where there are both the needed values.

There could be other files on the NCEI website that have the missing data. However, I decided that a search like that would take considerable time. It would take time to identify the data, and then find a way to incorporate that with the data you already have. This is assuming the data exists, and the search could have been fruitless. Instead, I decided to only accept and process the data for which I had all the necessary data.

Challenge 3 - Faulty Data

Within the input dataset, 2018.csv, there is a column that signifies a quality flag. Usually this is empty, but if it is filled, then that measurement has failed a quality check somewhere. I noticed this column when reading through the dataset's readme file. It wasn't mentioned explicitly within the specification, however I decided to filter out any failed measurements.

There were no failed measurements in Waddington, however in Oxford, there were 8 fails among the maximum temperature readings and 9 fails among the minimum temperature readings. Four of these fails were on the same day, which leads to 13 days where there was at least one failed measurement. Upon noticing a failure flag, my mapper adds 'fail' to the key. My reducer then won't process it.

Challenge 4 - Output one column

The specification requires us to use Hadoop to generate a one-column CSV file. This means that, ideally, the values we need to display (the temperature differences) are outputted, without any accompanying data, such as location or date. With a complete dataset, this would be simple. Hadoop automatically sorts between the mapper and reducer, so with a complete dataset, the reducer would output a sorted list of values.

However, with missing data, it becomes more of an issue. We need to use the output to plot a graph of the temperature differences for each day, which means we need to maintain some knowledge about the date of each piece of data so that it can be plotted accurately.

We can sneakily keep the date in the output, with it still technically being a one-column CSV file. For example, we could have '2018010111.3'. This is technically one value, and as the date is a constant length, we can extract the date as well. Additionally, we could separate the values with a symbol other than a comma, for example '20180101x11.3', and it would still be a one column file. That said, this isn't ideal, as there is still a need for significant post-hadoop processing, as you must extract the data and the date from the output in order to plot the results.

I decided to approach this challenge a different way – by only outputting the temperature difference data. I managed to maintain knowledge of the data by outputting a blank line for any date that does not have a complete set of data. I do this by iterating through the given range of dates (every day in 2018) and then checking through the entire input for entries that match that given day. At the end of this, I count how many entries there were, and if there was less than two, I output a blank line.

The output that this generates is convenient when plotting in a graph, as there is no processing required. When plotting using Excel, I only needed to copy and paste my data and then put a list of dates as the other axis. There was no splitting or processing needed.

Challenge 5 - Value in 2018.csv is tenths of degrees C, need degrees C

Within the provided input file, 2018.csv, the temperature minimum and maximum values are in tenths of degrees Celsius. Therefore, a value of '133' translates to 13.3℃. The specification would like us to plot the temperature difference in degrees Celsius, using the output file from our MapReduce program. I made my reducer divide the output by 10.

There was a slight issue with python variable types, however this was easily sorted. Initially I was dividing using two integers (eg. 133 / 10) and wanting a float, which python 2 doesn't support. I solved this by importing the functionally from python3 using 'from __future__ import division'. This issue could have been solved much simpler by dividing using a float (eg. 133 / 10.0), which python2 wouldn't have struggled with.

Challenge 6 - Testing without using GCP to maintain costs

Using the Google Cloud platform for all testing would not only have been potentially expensive, but also would have taken a lot of time. Instead, I used a couple of easier and quicker methods.

I used VirtualBox to create a basic Ubuntu VM on which I could test my mapper and reducer using apache-hadoop. This would save me time as I could directly run Hadoop from the VM rather than using gcloud's resources and waiting for the VMs to start up. I could also edit my files quicker, using the clipboard. Once I was happy with the mapper and reducer, I would test that they worked with Hadoop using this method.

I also used the Windows command line to simulate the Hadoop. As this was the easiest and quickest method, this is the primary method I used to test my mapper and reducer. On Windows I did this using 'type 2018.csv | python3 mapper.py | python3 reducer.py'. Using the pipe character ('|'), we can feed the output of one command into the input of the next, and so can simulate the Hadoop process. There are some missing elements, such as a sorting method between the mapper and the reducer.

Discuss how your MapReduce program's execution time can be improved

More VMs in the cluster

Using the Google Cloud platform, we can improve the execution time by creating a larger VM cluster. This will lead to more reducer tasks running parallel together. This may create multiple output files but there are console commands that concatenate these together.

Reducer efficiency

My reducer's efficiency can be improved. Currently, for every day of 2018, it checks the entire output from the mapper and finds the entries related to that day. Checking the entire mapper output is unnecessary as entries I have already checked do not need to be checked again.

As Hadoop sorts the mapper's output before feeding it to the reducer, we can assume that the output is sorted by date already, which means that as soon as we reach an entry with a date that is later than the current date we're looking at, we can stop looking as well.

Cut out failed values in the mapper

Currently, any entries that have a failed flag are still passed through to the reducer, and discarded there, however this should ideally happen in the mapper.

Two Hadoop Processes

My final solution consisted of running my MapReduce program twice, once for each location. I could improve the execution time if I was able to identify a method to accomplish the same in just one run of the program, without it adding significant additional process time.