ELEN-E6689: Large-Scale Stream Processing
Homework #1

Nicholas Christman nc2677@columbia.edu

February 20, 2019

Introduction

In this report a data-parallel processing pipelines are proposed and the results are discussed. Each implemented pipeline was designed to process web server logs containing information regarding HTTP-based accesses to web pages hosted by the server. The information of interest found in these web server logs include the Internet Protocol (IP) address of the client accessing the page, the time of the particular access, and the number of bytes sent by the server. For this assignment, the Apache Beam unified model was used to define a data-parallel processing pipeline for each of the four parts discussed in subsequent sections. Furthermore, given it's popularity in the research and development industry, the Beam Python SDK was chosen and, as such, the Python 2.7 programming language was used to implement each part of this assignment.

Data Description

Although the code can be easily extended to process other log files, the *epa-http.txt*[1] web server log provided to the class was used as the test dataset; therefore, the code as-submitted may not be compatible with other web server logs. For example, the date-time field used to create an event timestamp is quite specific for the given dataset, as shown below. Moreover, it is assumed that the log file is stored in the standard UTF-8 character encoding (Beam supports only UTF-8 and ASCII for text IO). Here is an example log entry from the *epa-http.txt*[1] web server log:

```
141.243.1.172 [29:23:53:25] "GET /Software.html HTTP/1.0" 200 1497
```

Although error checking has been implemented where appropriate, the architecture chosen to parse the log entries is somewhat naive. The architecture assumes each log entry is formatted consistently and uses whitespace as the delimiter for splitting the entry into elements. From a high level, the entry can be split into five groups: IP address, date-time, HTTP request, HTTP status code, and size of the message content (in bytes). For each of the parts discussed only the IP address, content size, and for some parts the date-time will be parsed. Parsing the IP address and content size is trivial (`element.split(" ")[0]` and `element.split(" ")[-1]`, respectively), but the date-time is more complicated if we want the code to process web server logs with different date-time formats. As an attempt to extend this code beyond the test data, the Python module `dateutil.parser` is utilized to (try to) automatically parse the date-time string before assuming the format specified above.

Discussion: Part 1

For **Part 1**, the problem statement is to write an application that reads in the logs produced by a web server and to compute the total number of bytes served to that IP address for each unique IP. The workflow illustrated in Figure 1 provides the architecture of the processing pipeline. The GetIpSize Ptransform executes a ParDo function for parsing the log entry and returning the IP address and content size as a tuple, with the IP address as the key and the content size served to that IP address (during that specific access) as the value. The parsing function is explained in the **Data Description** Section above. The GroupIps Ptransform executes the pre-built Beam `CombinePerKey(sum)` function for aggregating all `(ip,size)` tuples by the IP address and then summing the sizes – the output of this Ptransorm is a tuple with IP address as the key and total size served to that IP as the value. Finally, the FormatOutput Ptransform simply formats the Pcollection into a readable output and sends it to the pre-built Beam WriteToText IO Ptransform.

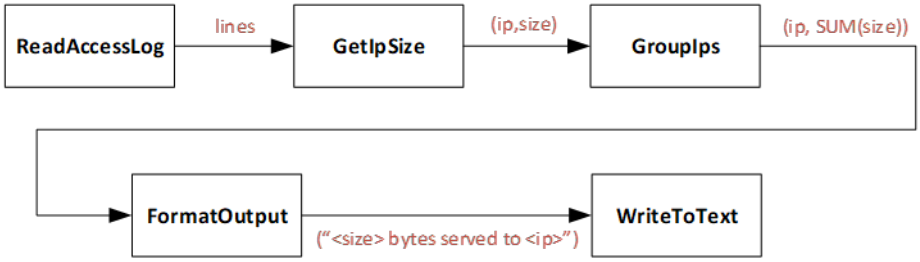

Figure 1: Workflow graph for Part 1

Discussion: Part 2

In **Part 2** the objective is to extend the application from **Part 1** and return the top-K IPs that were served the most number of bytes. Where `K` is a user input passed at the command line, instructing the program how many IPs should be returned. The default value of `K` is zero (0), which will return the same results as **Part 1** (i.e., no sorting/filtering). As shown in Figure 2, the processing pipeline is similar to **Part 1** adding CombineAsList and SortTopK Ptransforms to gather all Pcollections and sort/filter the elements, respectively. Gathering all Pcollections and sorting/filtering is an expensive operation; however, it was the best approach given the type of dataset. The output of this processing pipeline is a text file similar to **Part 1** except only the top-K IPs are returned and they are sorted in descending order.

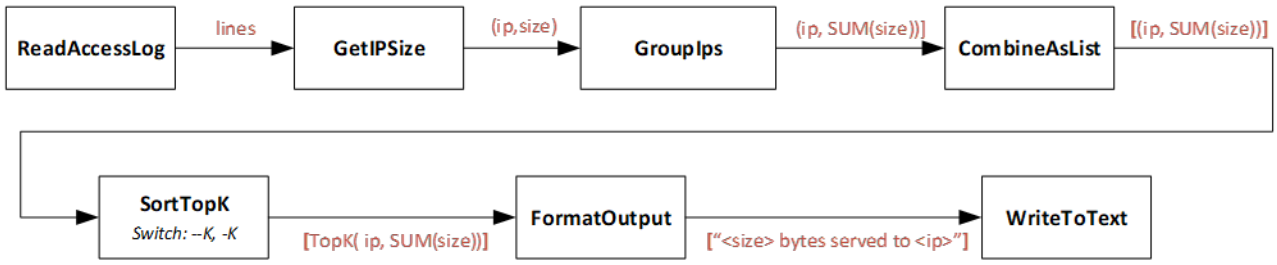

Figure 2: Workflow graph for Part 2

Part 3

Part 3 extends the pipeline developed **Part 2** and computes the total number of bytes served per time window of 1 hour (with tumbling windows) for each unique IP. It does not appear to be required, but the sorting/filtering functionality from **Part 2** is carried over and sorts/filters the IPs per window. From Figure 3, it appears that the pipeline has become significantly more complex; however, the output Pcollections are illustrating the windowing effect. The pipeline is quite similar to **Part 2**, adding a Timestamp and Window Ptransform to assign each tuple the timestamp parsed from the log entry and to define the fixed window, respectively. Operations of the downstream Ptransforms are identical to **Part 2**, except they now process Pcollections per window rather than on a single Pcollection. The output of this processing pipeline is a text file similar to **Part 2** except the results are grouped by window (1 hour period) and sorted by size in descending order iff the top-K is declared. Note: In the Beam model, a 1 hour *tumbling window* is implemented as a 1 hour *fixed window*.

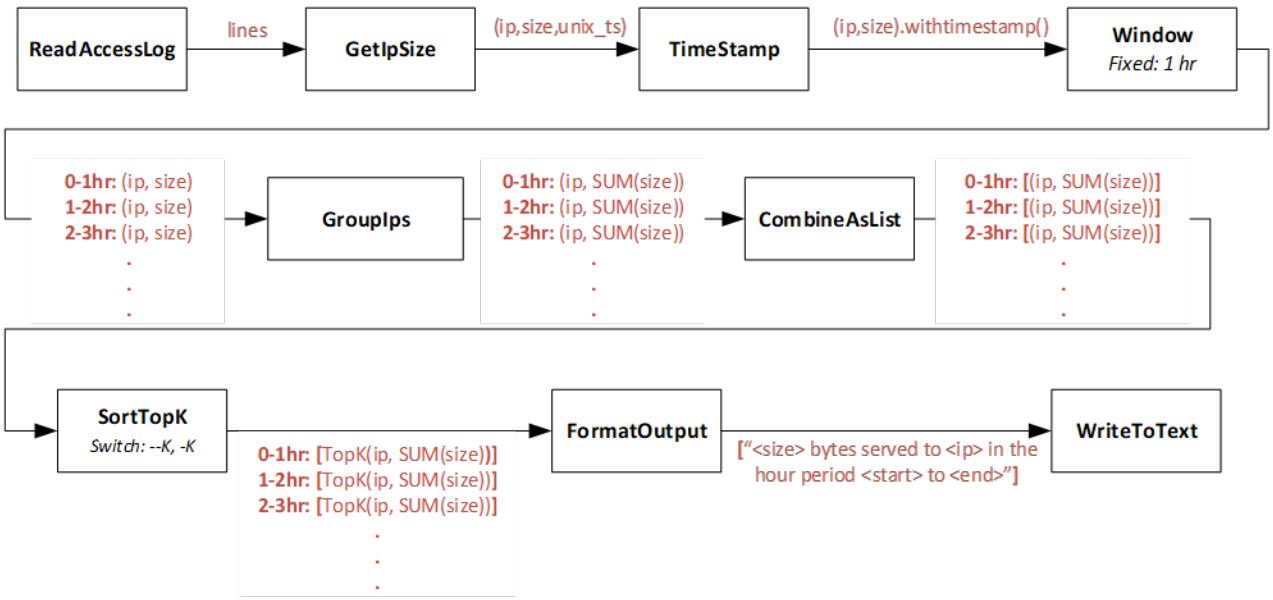

Figure 3: Workflow graph for Part 3

Part 4

Part 4 is mostly independent of the previous parts and although not required, it was decided to extend the pipeline developed in **Part 2**; therefore, the same functionality described in **Part 2** applies to **Part 4** (refer to Figure 4). This pipeline is to compute the same statistics for a specified subnet from the web server log; that is, to aggregate across IPs based on a specified prefix. A Ptransform, PrefixMask, is deployed downstream of the GetIpSize Ptransform and masks the `(ip,size,unix_ts)` IP address based on the `prefix` value. The `prefix` is a user input passed at the command line, providing a filter mask in the form of a standard, well understood sub-net mask. For example, `--prefix = "255.255.255.0"` would instruct the program to aggregate IPs based on the first three bytes of the IP address (i.e.,a.b.c.*) and `--prefix = "255.0.0.0"` instructs the program to aggregate IPs based on only the first byte of the IP address (i.e., a.*.*.*). In addition, all web-links are aggregated based on the first two words and are not affected by the `prefix`. The default value of the `prefix` is `255.255.255.255`, which should produce similar results to **Part 2**. Note: in **Part 2**, the web-links are not masked.

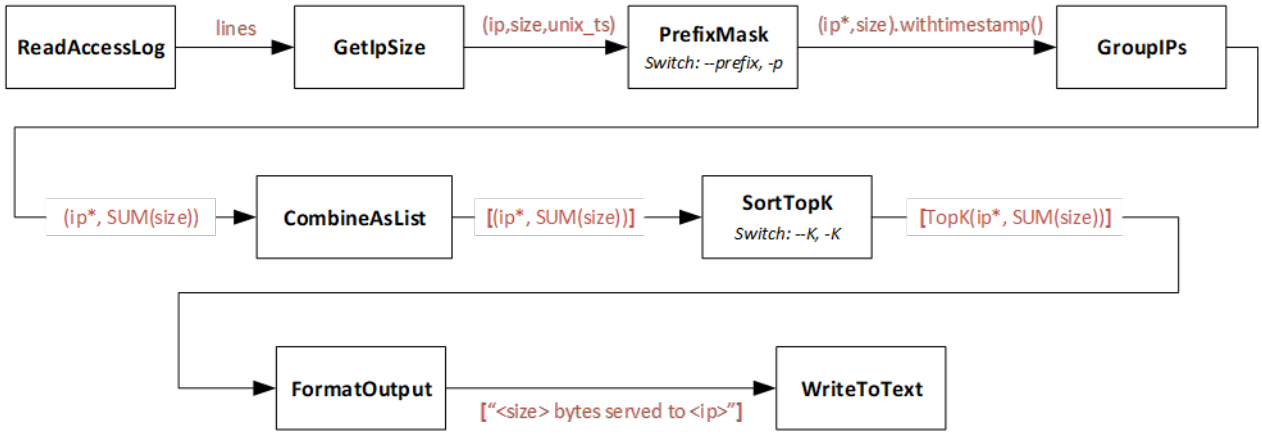

Figure 4: Workflow graph for Part 4

Source Code

Refer to the included Python 2.7 files, where each part is implemented in a separate Python script (`hw1_part1.py`, `hw1_part2.py`, etc.).

References

[1] http://ita.ee.lbl.gov/
[2] https://beam.apache.org/documentation/programming-guide/
[3] https://beam.apache.org/documentation/sdks/python/
[4] https://beam.apache.org/get-started/wordcount-example/

Note: some references are included in the Python scripts as in-line comments, where external help was required.