PYTHIAN BIG DATA CODING CHALLENGE

October 18, 2018

ANDREW T. SCHNEIDER

ANDREW T. SCHNEIDER CONSULTING, LTD | 26 IRONWOOD COURT EAST AMERST NY

Table of Contents

Requirements

The following summarizs the requirements;

Background

Show us what you can do! Demonstrate your technical expertise by completing the Big Data coding technical challenge. This is your first opportunity to demonstrate your knowledge, skills, and abilities to the hiring team by walking us through your solution to each question and showing us the steps that were involved, including showcasing the code you used, for the following scenario:

Problem

- "A Linux directory structure contains 100G worth of files. The depth and number of sub-directories and files is not known. Soft-links and hard-links can also be expected.
- Write, in the language of your choice, a program that traverses the whole structureas fast as possible and reports duplicate files.
- Duplicates are files with same content.
- Be prepared to discuss the strategy that you've taken and its trade-offs."

Note

- Please note, we expect an implementation of this problem using Python/Java or any other general purpose programming languages.
- Using existing tools like fdupe is not a proper solution to this challenge.

Deliverable

Please provide a quick documentation that briefly summarizes the following:

- Design approach [to as well state the ways how performance optimizations were considered and approached]
- Expected Inputs/Outputs
- Assumptions and known limitations with the delivered code
- Summary on how to run the test client and its pre-requisites

1. Design Approach

Analysis

- Abstract Problem:
  - In order to solve this problem we need to which files have the same content of one or more other files. Abstractly this could be represented as a N by N matrix with the file names the lables for the row and columns and the value in the matrix being a binary flag. More concretely we would like an output that consists of just the edges which are flagged as true. That could be expressed as: { set number, filename1}. The set number indicates which set the files belong to.
- Simplistic Approach:
  - The simplistic approach would be to create a hash in memory where the key of the hash is the file name and the value of the hash is a structure that contains a variable which has the contents of the file. The algorithm could consist of two nested loops that make all the comparisions, create a list of all the files the file is identical to. That list would be another hash stored within the the structure pointed to by the first hash.
  - The simplistic approach will not work because there is probably too much data to load into memory. Even if there was, the problem as stated is of N^2 complexity comparing every cell to every other cell. What is needed is a way to reduce memory usage and to reduce computational complexity.
- Using File Size:
  - Each file has a size in bytes which the file system directory maintains. Only files which contain the same number of bytes are potential duplicates. This can reduce the number of comparisons needed.
  - If two files have the same number of bytes, they still need to be compared byte by byte to see if they are identical. The simplistic approach is to write a method that compares the two files. This introduces the overhead of reading the file into memory. Worse yet each file needs to be read more than once.
- Using Digests:
  - We can reduce the number of times a file is read by creating a “digest” of the file. A hash function can reduce the file to a key of a standard size keys. We can update our sets of candidate files (which have the same size) by adding a this “digest” to our list of file properties.
  - Using hashing introduces the need to read every file of to create the hash. But helps to cull the number of files to actually compare.
- Comparing Files:
  - Two files which have the same “digest” may be identical because the keys are not unique. When two files are found with the same key, then the they actuall need to be read to determine if they are identical.
  - This comparison need not examine every byte in the file: it merely needs to find the first byte that are not identical.

- In summary, by using the file size, and computing “digests” we can reduce the complexity of the problem.

Algorithm

The following algorithm implements the concepts on comparison from the previous section:

- Step 1:
  - Read the file system directory,
  - create an array of a structure which contains the file name and the size in bytes.
  - Sort the array by the size in bytes, and then filename.
- Step 2:
  - Process the list from step1.
  - For each group of files which are are the same size, assign a unique group number to them. The group number is an integer that starts with 1 and is incremented each time a new file size group is detected.
- Step 3:
  - Process the list from step2.
  - Open all files which are part of a set of 2 or more entries, and create a “digest”.
- Step 4:
  - Process the list from step3.
  - Sort the list by “digest”.
  - Assign a the a “digest group” a small for all the files with the same digest.
- Step 5:
  - Process the results from step4.
  - Open each file in the group and load the first n bytes into a variable associated with each file.
  - Compare the buffers to see if they are identical.
    - If a an entry is not identical, remove it from the list.
    - If identical entries remain, read the next n bytes and compare again.
- Step 6: Process the results from step5:
  - create a file that contains the following colums: { group id, digest id, size in bytes, filename}

Peformance Enhancements

After proving the functionality the algorithm will be optimized.
Reading the files and comparing them are the most time consuming activities.
Reading files can be implemented in worker threads
This version of the code does is not so optimized

2. Interface

bin/run1

- Argument 1
    - Absolute Directory Path to scan
- StdOut
    - <group id>,<digest id>,<filename>,<size in bytes>
- StdError
    - Messages generated by the program

3. Assumptions

1. Can not use existing utilities.
2. List of files can always fit in available memory
3. Same character encoding for all files
4. While computing a hash requires reading a file, at least it only needs to be read once.
5. Matches based on hash key require one more read for each file.

4. Installation

```
install pip
install python

mkdir foo
unzip foo.zip
bin/run1
```

Sources Consulted

1. https://en.wikipedia.org/wiki/SHA-1
2. https://www.bogotobogo.com/python/python_traversing_directory_tree_recursively_os_walk.php