Documentation: 'Compete' Questionnaire

Author: Preeti Singh, Computer Science Department Carnegie Mellon University

Objective: The program does an IP lookup of the batch of the users who sent us their clicks and replaces the IP s with some sort of a hash.

Input: A file having ip_address, user_id eg: 123.45.676 user_id1

Output: The program should output a file with IP s replaced with their hashes, also tab separated looking like this: hF567+/ajaj= user_id1

(****Approach A: Code attached, filename – iphash.py)**

NOTE: There could be various approaches to solve this problem.

I used the following approach:

****Approach A)** Connect the storage file (having ip_address, hash_code) to the SQLite database. Then do the lookup in the database for the input file. Check if there are repeated Ip_addresses in the input file.
Runtime: This approach will also give us O(mlogn + nlog n) since the SQLite uses B-tree as its underlying Data structure. Then for look up it will be n*log n since there are 'n' enteries in the input file.

*****Code Attached : iphash.py. Please run using input.txt and prev_ips.txt as the example test files.***

Assumptions made in the code: I am assuming that the Awesome_hash() function is written inside this file. For now I have written my own awesome_hash() function which you can replace with your own awesome_hash() function or you can remove my awesome_hash function and import your awesome_hash().

***Approach B)** Construct B-Tree on file (having ip_address, hash_code). Then do 'n' lookups on this B-tree(where 'n' is number of new ips in input file having ip_address, user_id). Then use Awesome_has() function if some ip_address is not found in the file stored (having ip_address, hash_code).
Reason behind using Btree is that while B-tree keeps data in a structured fashion and allows searches, sequential access, insertions, and deletions in logarithmic time, yet it optimizes for mitigating the high seek and read time from hard disk. (More details in optimization.)

Runtime: Runtime for this approach will be O(m*log n + n*log n). Since Traversing B-tree will be logn for one entry and m*log n for 'm' entries in the input file.

Pseudocode Attached (iphash_btree.py, iphash_btree_Lookup.py)

Approach C) External Sort on the stored file(having ip_address, hash). Then merge both files (input nd stored sorted file) and create new entries in the merged file if some ip_address is not found in the stored file and use Awesome_hash() function to convert the new ip_addresses into hash_code.
Runtime: Runtime for this approach will also be O(m*logn + n*logm). Since external sorting will take O(m*log n) and merging these two files will take linear time.

OPTIMISATION:

1) Since, the input files for this program are coming from 3rd parties and we
cannot be sure of the quality of their data. We need to take care of:
(i) invalid ip addresses: For example some term may not be between 0-255. We have to discard such entries.
(ii) duplicate ip addresses: If the same ip address occurs more than once, we only keep the last occurrence. This is because the file is record of history and the IP address could have been flushed and assigned to a new user, so the last entry represents the correct assignment.

2) We proposed not to use Binary search tree and instead B-Tree to exploit the system hardware. Reading from hard disk is slow and always entire page is read from the hard-disk. So it is very wise to use all the data read in the page rather than discarding it. Thus, we propose to select the order of B-Tree of order so that each node of B-Tree fits in a page. This leads to much better utilization of the data read and helps reduce the effective cost of a seek and read on the hard disk.

3) We can save the IP addresses (32 bits) as one single integer. This will increase the search speed since the comparisons become easier

4) Furthermore, while building the B-tree , instead of again and again inserting each entry into the B-tree we can do Bulk loading (specified in pseudo code, iphash_btree.py).