Shiqi Wang

CS 32

Project 4

Report

1. None.

2. data structures and algorithm chose

Class Name: DiskMultiMap

* bool createNew

create an open hash table on disk

a head structure at front which contains a usigned int to record numBucket and a Binary::Offset prev\_delete to record the most recent deleted node’s position.

prev\_delete->most recent deleted position. m\_next->the deleted position before...-> -1

Add an array of bucket to keep track of the real offset of each bucket

Initialize each bucket to the offset of itself

* bool insert

open the head node on disk to calculate the bucket position for the key

push evey newly inserted node at the end of the list in the bucket, record the last inserted node’s position in the bucket.

bucket->most recent inserted position. m\_next->previous inserted... ->bucket offset

To determine the real position of the node on disk, read prev\_delete:

if is -1, means no available slot before, then write at the end of disk;

else, write at prev\_delete;

update prev\_delete to retrieve a new open slot (or -1);

* Iterator search

Use the given key to find the bucket its in, push all the match key’s position into a list;

Use that list with a this pointer to create a valid iterator and return;

If the key is invalid / not founded or cannot open the, return an invalid iterator;

* int erase

open the head node on disk to calculate the bucket position for the key

get position of first node, use m\_next to retrieve the next node until back to the bucket pos

if found a node to delete, increase numErase and remove it from the list, update delete list (write the pos in prev\_delete into m\_next, then update prev\_delete to the pos of deleted node)

Class Name: DiskMultiMap::Iterator

* bool isValid() const

if the m\_map ptr is nullptr or the list is empty, return false;

else return true;

* Iterator& operator++

Use a STL iterator to point to the beginning of the list

Erase one item and return; (when the iterator hits the end, the list will be empty, thus invalid return false)

Class Name: IntelWeb

* bool createNew

create two DiskMultiMap, one map key to value, one map value to key;

return true only if successfully create both of them;

* bool ingest

for each line, insert in both two maps;

* usigned int crawl

using breadth first search for the badEntity’s name:

create a unordered\_set bad to access the badEntity’s name quickly without repetition

create a set interaction to record bad interactions without repetition

first push all the indicators into a queue;

while the queue is not empty:

take out the front string and delete it in the queue;

search it in two maps;

push the values that is less than the minPrevalence into both queue and bad;

when the queue is empty, bad and interaction contains the values that we need for badEntitiesFound and interactions, thus push into vectors;

since set keep things in order, we don’t need to sort interactions again;

sort badEntitiesFound;

return badEntitiesFound.size();

* bool purge

first search KeytoValue map (key at “from”)

if found a node that key matches, erase it in both two maps;

then search ValuetoKey map(key at “to”)

if found a node that key matches, erase it in both two maps;

* int getPrevalence

search the key in both map using iterator and return the frequency that it appears;

3. Whether or not each method satisfies our big-O requirements, and if not, what

you did instead and what the big-O is for your version.

Every method satisfies the big-O requirement except crawling’s searching part;

The step insert InteractionTuple into the set cause O(logT);

The step getPrevalence needs search in the disk which cause O(K) where K is the num of nodes in that bucket;

Thus the crawling searching’s big O is O(T(logT+K));