AN ATTEMPT TO IMPLEMENT LSM TREE

Overview:

We are implementing a LSM tree so we need in essence **2 types of storages**.

One on **memory** and the second **on disk**.

The memory will correspond to a **map** due to its fast retrieving nature as required of a traditional memory, and the disk will correspond to a data structure which is contiguous in nature. A **vector(dynamic array)** would be the perfect fit for it.

Memory(Map) : It is a <key,value> container wherein we store all the incoming <key,value> pairs until a certain amount of allocated memory. Let's denote this reserved memory as L MB. It is a red black tree.

Note : The extra small overheads of space i.e for example default allocation during runtime etc are neglected here and we assume that we have L MB of free space in addition to those overheads.

Disk(2D vector) : It is essentially a contiguous array containing a bunch of other arrays in it. These “other” arrays denote the readable and writable chunks of space on the HDD for our purpose. This by default is allotted a max size of L total such chunks. Each chunk can hold a large amount of data.

So now we inspect it further,

Let's denote our disk as : vector<vector<data>> disk(2*L + 1)
Then disk[i] denotes the ith node/chunk. And disk[i][j] denotes the data stored at disk reference j on node i.

Now let's have a look at how the things will flow from the very top before jumping into further detail,

LSM tree will support 3 kinds of operations :
1) Insert <key,value>
2) Find by key
3) Delete key

We have an **input query stream(order 10^5)** from the user in the form of the above 3 operations and we need to implement an LSM tree to **store/update/answer** these queries online.

Structure **dat** is our data in the form of key value time tuple.

Function name	**Function return type**	**Function parameters**	**Function use**
merge	vector<struct dat>	vector<dat> d1, vector<dat> d2	
compress	void	vector<vector<dat>> &segments, int L, map<string,map<int, int>> &h, int sz	Function compresses all L segments to 1 segment and puts it into node - 1. And put the pending sz/2 segments on node - 2.
update	void	vector<vector<dat>> &segments, int L, map<string,map<int,in t>> &h, int sz, int ava	
search_m	struct dat	(map<string,map<int,i nt>> &h, string k	Function searches for key=k in memory, and gets time values from it and returns the latest of them. If not found, return -1.
search_d	struct dat	vector<vector<dat>> &segments, string k, int ava	Function searches for key=k in disk, and gets time values from it and returns the latest of them. If not found, return -1.
delete_m	void	map<string,map<int,in t>> &h, string k	Delete every instance of key = k from memory
delete_d	void	vector<vector<dat>> &segments, string k, int ava	Delete every instance of key = k from disk.

<u>Algorithm and Pseudo code:</u>

Now we explain the functions in detail :

Insertion (compress+merge+update) :

When the operation is of the 1st type i.e Insert :

1)Read the current <k,v> pair.

2)If we have enough size to put it in memory(map) then we insert it.

3)Else our memory is full. So we free the first L/2 amount of memory on the latest available disk chunk

4)Now we are inside disk and are checking if we actually have any free space or have used up all our amount of space on disk.

5)If we have used all amount of space then it is time to apply the process of **compaction/compression** and then

5-1) Call function **compress(disk,L,memory)**

5-2) We are inside compress and now the first step is to merge all the disk chunks into one i.e the very first node.

5-3)Call function **merge** if we want to merge disk node i to current disk node 1 as so : **merge(disk[1],disk[i])**. It is a O(Total disk size) process. After merging we will have the contents of the entire disk in node 1 with the other nodes being free.

5-4)Now we are done with compaction and we proceed to free half of our memory into the next node of the disk i.e node 2.

5-5)After we insert the data in our disk node 2. We erase the first half of data from our memory as it now resides on our disk.

5-6)Exit function compress and put the current standing <k,v> pair in memory. Also update the next freenode in disk to 3.

6)Else we don't need compaction and can directly update the available disk node with L/2 amount of memory. In that case we call the function : **update(disk,L,memory,freenode)**.

6-1)We are inside our **update function**.

6-2)Migrate the first L/2 amount of data from our memory to **disk[freenode]**.

6-3)After migration is done, delete the migrated amount of data from memory and exit.

7)Update **freenode:=freenode+1** and put the current standing <k,v> pair in memory.

Search:

1) We are given a key and want to return its value if it exists on either of our disk or memory.

2) We need to bifurcate this into 2 methods : **search_m** and **search_d for searching in memory and disk respectively.**

3) **Search_m(memory,key)**. We run a search inside our memory(using the count function of map) and return NULL if not found. As its a RB tree this operation takes O(log(L)) average time.

4) **Search_d(disk,key)**, Notice that the order of migration from memory to disk needs to be and is lexicographic by nature. This is because we can search for a key faster even in disk like memory. So here we can apply binary search on key k in the disk from the latest last used disk node to the oldest node i.e.in the order freenode-1 to 1. So we did this operation in O(L*log(MAX(disk[i].size()))) best case time.

5) Get the values returned from both the above functions and if both are NULL then print NOT FOUND else print the value.

Delete from disk:

When the operation is of the 3rd type i.e delete :
The key is deleted both from memory and disk.

delete_d deletes every instance of key = k from disk.

1) Run a loop for all disk nodes(i from available length-1 to 1)
2)Assign p1 to lower bound function for disk nodes that returns the position of the oldest key instance.
3)Assign p2 to upper bound function for disk nodes that returns the position of the latest key instance.
4)Now delete/erase all instances of that key from p1 to p2(oldest to newest). (erase function)

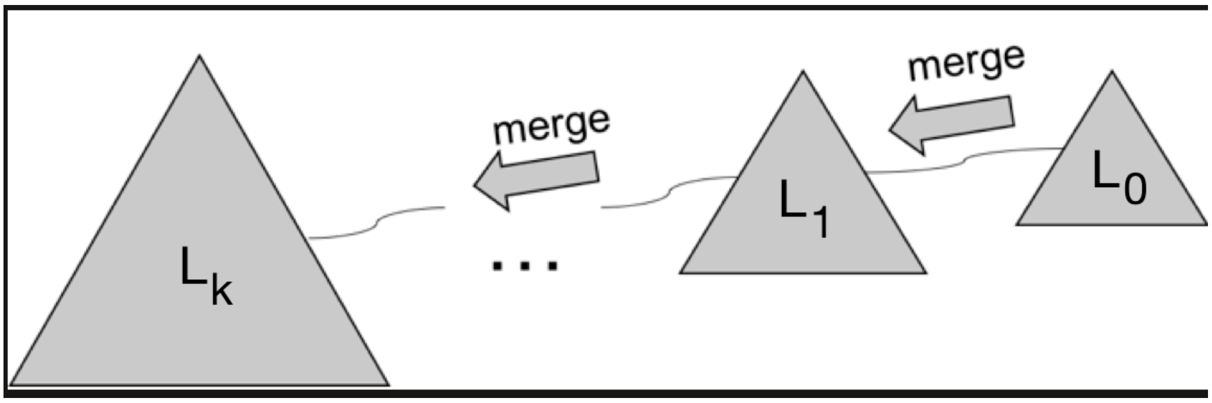
Delete from memory:

When the operation is of the 3rd type i.e delete :
The key is deleted both from memory and disk.

delete_m deletes every instance of key = k from memory.

1)If string key has no count in the map (if(!h.count(k))), return; (end of function)
2)else, erase every instance of the key in it. (h.erase(k))

On the desired exponential nature of memory and how our LSM tree adheres to it.



On the internal scale it looks like we are compacting the size of our disk which we actually are by compressing all the L chunks into 1. But on the outside it looks as we are expanding our memory linerarly which is a drawback of the LSM tree since it is intended to be write heavy.
So,

We have L chunks.

Lets say each chunk can hold K*L space for data for some constant K.

Now assume our memory is filled and its time to compress stuff.

We put K*L^2 data in the first node itself assuming no deletes.

Then we wait for the disk to fill up again.

And we put K*L^2 amount of data again after compressing for the second time to get 2*K*L^2 data

So we see that the total memory increases linearly as depicted in the diagram after each merge+compaction process.

So in order to satisfy the above requirements atleast one of the disk chunks should have sufficient/large amount of memory available.

Hence proved.