Optimized Indexes based on Log Structured Merge Trees

Team 05
RAGHAV GUPTA (2018076) and SUCHET AGGARWAL (2018105)

Introduction

- Modern databases deal with enormous amounts of data, with high frequency of data insertions and lookups.
- Handling such enormous data with traditional methods is not time efficient and thus databases are organised in indexes for fast data retrieval.
- Most modern day database systems implement these in the form of B-Trees, Fractal trees or Log structured merge trees.
- In this project we create a Search Optimised Index based on Log structured merge trees and compare its performance with serial implementations of LSM trees and B Trees

Milestones

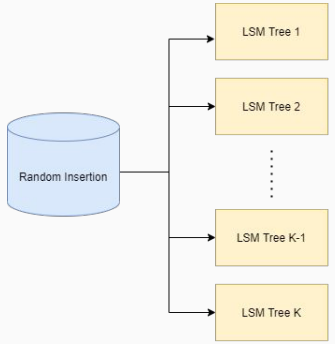
S. No.	Milestone
	Mid evaluation
1	Design Simulator for operations on data structures being compared ✔
2	Design GPU algorithm for insertion/deletion in LSM ✔
3	Implement B-Trees on CPU ✔
4	Implement LSM on CPU ✔
5	Implement insertion/deletion for LSM on GPU ✔
	Final evaluation
1	Implement Fractal-Trees on CPU
2	Design GPU algorithm for lookup/count in LSM
3	Implement lookup/count for LSM on GPU
4	Fine tune and Analyse GPU implementation using profiling/memory management.

Indexing Algorithms

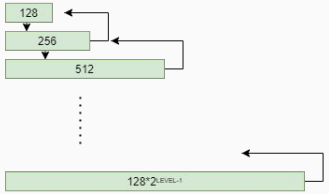
- B-Tree - generalization of binary search tree - node could have more than two children.
  - When a data is inserted or removed from a node then nodes are split or joined.
  - Since, B-Tree node contain variable amount of data they do not need rebalancing as frequently as binary search tree
- Log Structured Merge Trees provide an efficient indexing scheme for scenarios where the frequency of operations is high.
  - This is done using deferred batch insertions, but the cost of search goes a bit up.
  - The main idea is to maintain multiple levels within the tree and merge consecutive levels once the lower levels begin filling up.

How to make LSM Tree Parallel?

1. For parallelising LSM trees, we create a collection of LSM trees or a LSM Forest.
2. This way we can insert in any of the trees randomly
3. But Searching and Counting can be done in all the trees within the forest, making use of all the computing hardware available.



LSM Forest



LSM Tree

Approach - Insertion

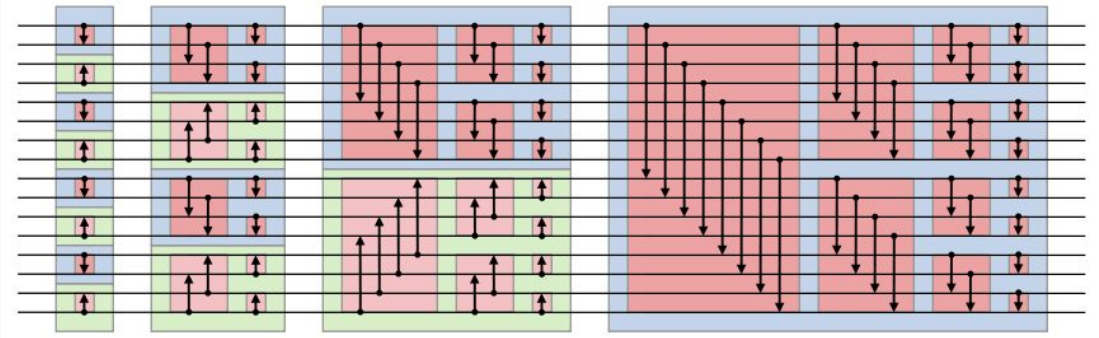
1. Insertions in LSM trees always occur at the lowest level C_0, If this level overflows, the elements are merged in the next level.
2. Thus at any given level i, if C_i is overflowing, the contents of C_i and C_{i+1} are merged and moved to C_{i+1}
3. Since we implement a search optimised index, data in all the levels is sorted, and the merge operation maintains this sorted order

Algorithm 1: Insert in LSM Trees GPU

INPUT: Key K to be inserted, LSM Forest T = $\{T_0, \ldots T_{Trees-1}\}$;
bucket = Randomly pick a index for the tree this key is inserted into;
T' = T[bucket] = $\{C_0, \ldots, C_{k-1}\}$;
Insert in C_0 in sorted order;
if *If* C_0 *overflows* **then**
  while C_i *overflows* **do**
    Append C_i to C_{i+1};
    Sort(C_{i+1});
  end
  Append C_0 to C_1;
  Sort(C_1);
else
end
Result: LSM Forest T

Sorting - Bitonic Sort

- Bitonic Sort is the fastest sorting network
- A sorting network is a special kind of sorting algorithm, where the sequence of comparisons is data independent. This makes sorting networks suitable for parallel implementation.
- Parallel run time = $O(\log^2 n)$
- Total number of comparisons = $O(n \log^2 n)$



Approach - Deletion

1. For deleting an element all the Tree in the Forest are searched in parallel. Thus one tree is searched by 1 block in the grid.
2. For searching and deleting the key within a tree, all levels can be parallely searched again, this is done per level by one thread.
3. In the current version, the searching is done over the entire array, one element per thread by a dynamic kernel launch within the delete kernel. This however can be replaced by a more efficient method (can be binary searched).

Algorithm 4: Deletion in LSM Trees GPU

INPUT: Key K to be deleted, LSM Forest T = $\{T_0, \ldots T_{Trees-1}\}$;
bucket = BLOCK ID;
level = THREAD ID;
T' = T[bucket] = $\{C_0, \ldots, C_{k-1}\}$;
if *Search K in* C_{level} **then**
  Delete K from C_{level};
else
end
Result: Updated LSM Forest T

Results

The current implementation is simple and is not fully optimised

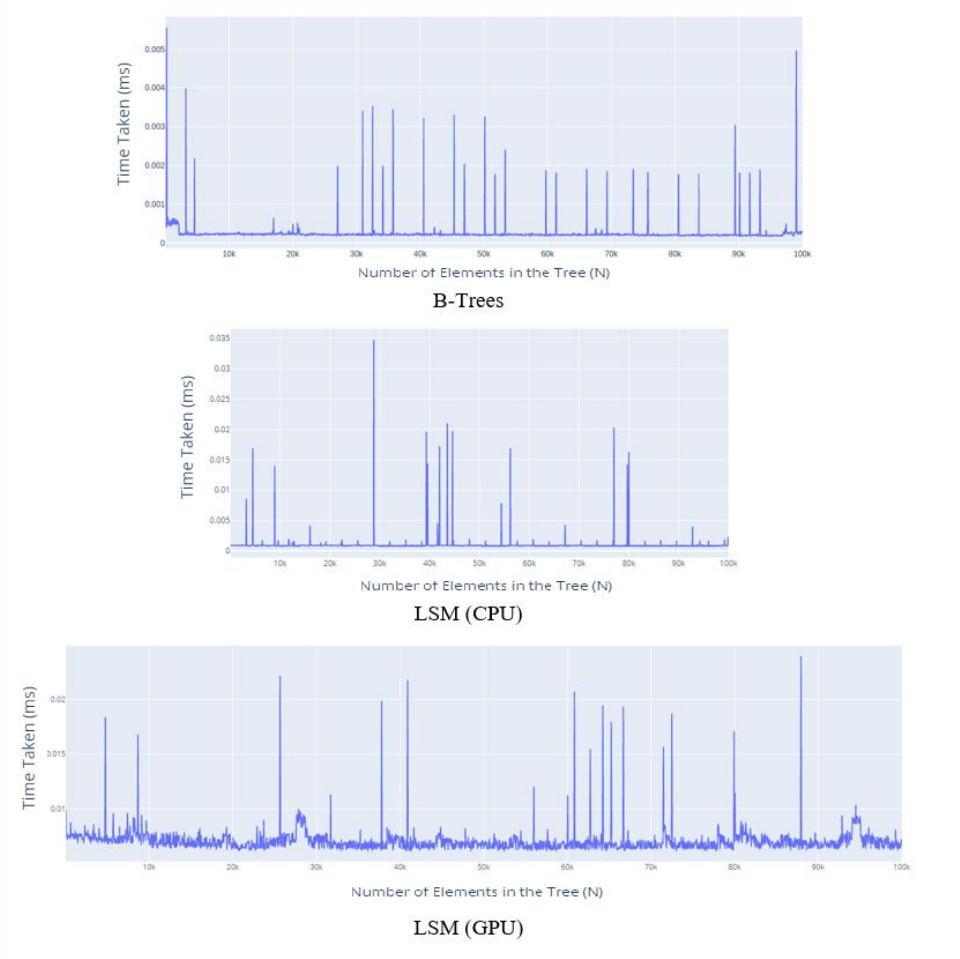
Insertion

Timing Results

- CPU (B-Trees): 0.000337428 ms
- CPU (LSM Trees): 0.00049103 ms
- GPU: 0.00030427 ms

Speedup

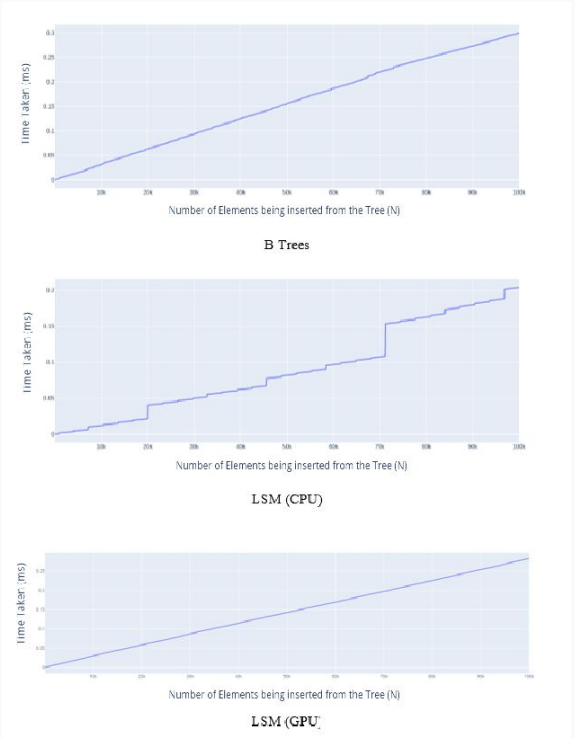
- GPU vs CPU (B-Trees): 1.11x
- GPU vs CPU (LSM Trees): 1.61x



Avg time for one insertion

Total Insertion Time versus Input Size

1. The insertion time variation over the number of elements in the index, does not vary heavily but has occasional spikes
2. This trend is observed across all the three implementations.
3. This can be attributed to the splitting of nodes (in case of B-Trees) and the rolling merge operation (in case of LSM-Trees)
4. For total insertion time, For all three approaches for insertion we observe a linear increase in the time taken as the size (N) increase



Total Time for inserting all keys

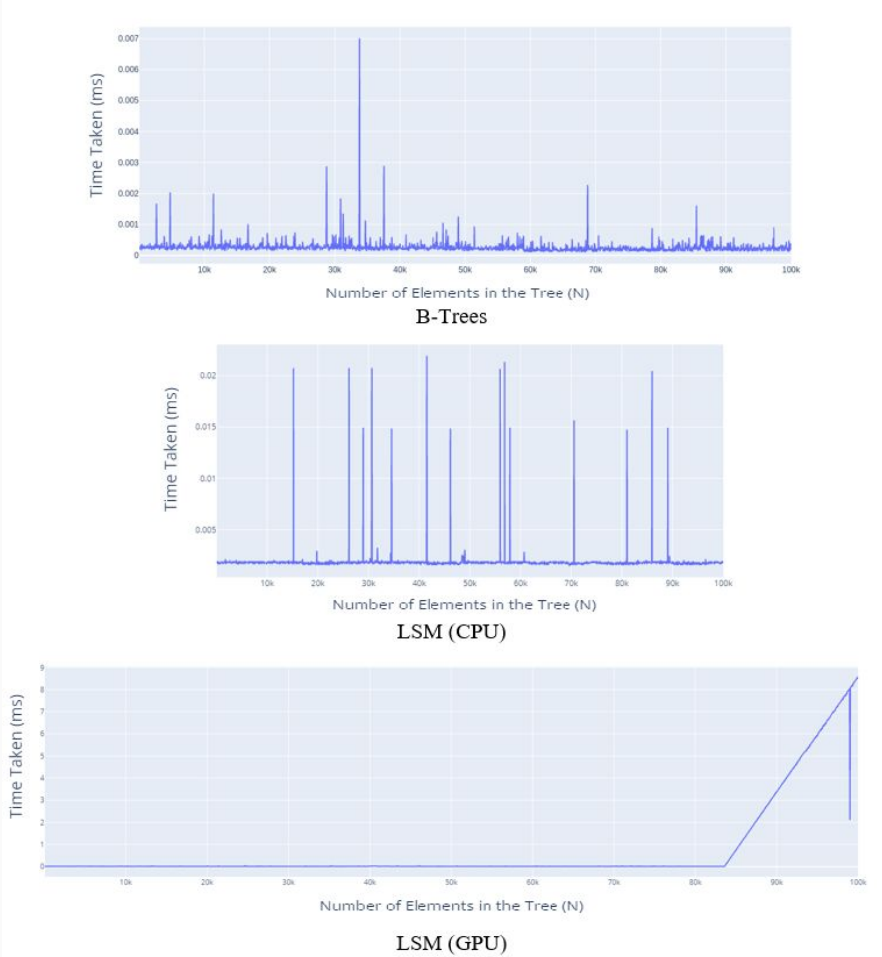
Deletion

Timing Results

- CPU (B-Trees): 0.00167183 ms
- CPU (LSM Trees): 0.0058738 ms
- GPU: 0.00123993 ms

Speedup

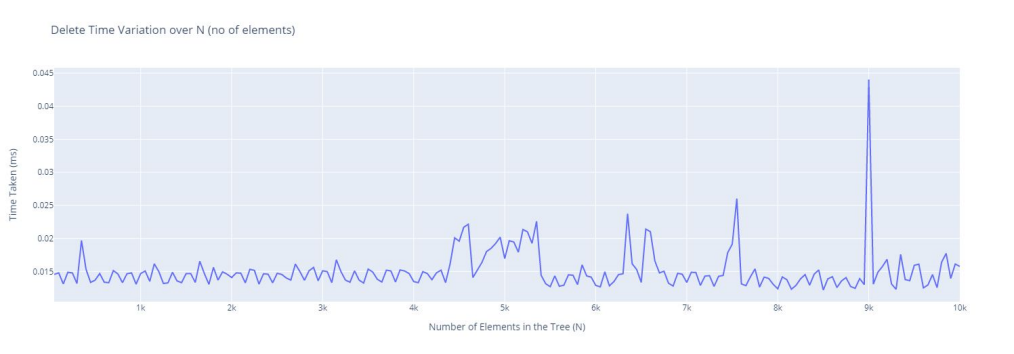
- GPU vs CPU (B-Trees): 1.30x
- GPU vs CPU (LSM Trees): 4.73x



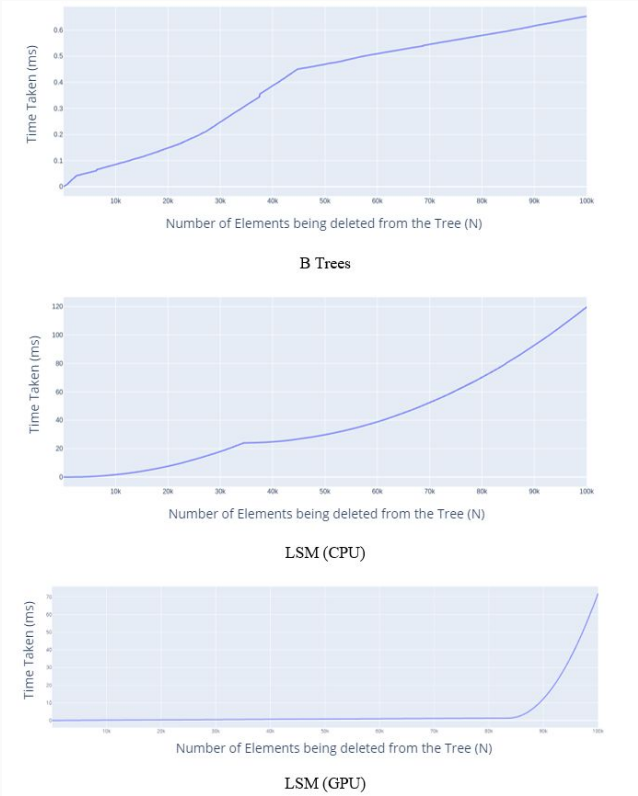
Avg time for one deletion

Total Deletion Time versus Input Size

For deletion the GPU implementation has a very different graph as compared to the CPU versions of B-Trees and LSM-Trees, particularly for high values of N (number of entries in the index) the time increases



Avg time for one deletion
(max size = 10k keys)



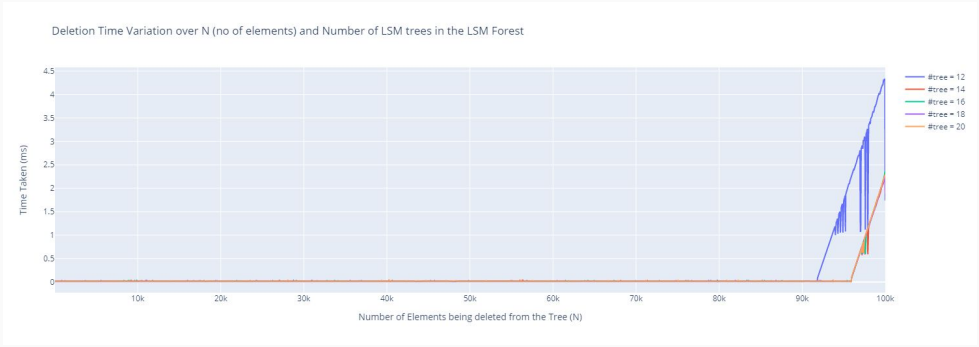
Total Time for deleting all keys

Total Deletion Time versus Input Size

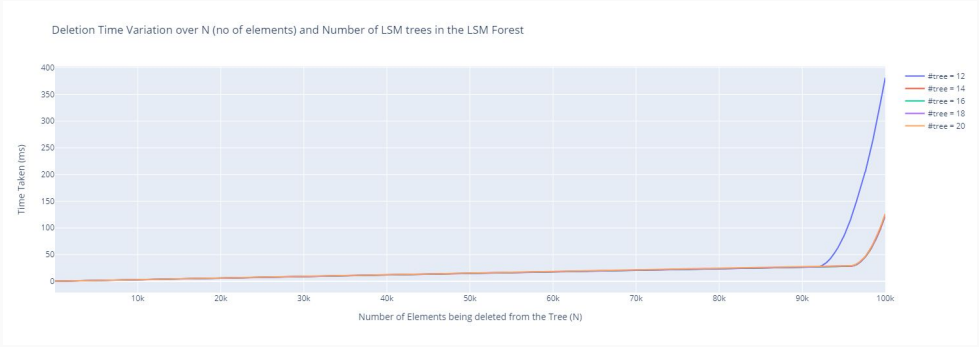
1. A similar trend is also observed in total time vs number of keys deleted.
2. This could be because majority of the levels in all the trees are filled, searching over all these in parallel takes longer as we need to synchronise across all blocks and threads than when some levels are empty and are not searched over
3. The above can further be strengthened by the graphs for average deletion time when only 10k keys are inserted.
4. And by Varying the number of Trees in the LSM Forest as Given on the next slide

Variation of deletion time with no of trees in Forest

- For Insertions almost all the settings have similar behaviour, which is expected since the insertion operation randomly picks one of the trees in the forest and then inserts in it
- But for deletions we observe that when the number of tree are 12, we get a distinctive jump in both the average deletion time for one deletion, and total deletion time for deleting all the key, beyond a certain number of keys.
- This number was lower for a lower number of trees in the forest (= 12) and higher for the higher values



Avg time for one deletion



Total Time for deleting all keys

Total Time for inserting all keys

Code

The code for the project is available at:

https://github.com/CSE-560-GPU-Computing-2021/project_-team_5

Thanks!