Optimized Indexes based on Log Structured Merge Trees

Team 05

RAGHAV GUPTA and SUCHET AGGARWAL

1 INTRODUCTION

Most modern databases deal with enormous amounts of data, with high frequency of data insertions and look ups. 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. The trade off one needs to make is then dependant on the particular use case, whether one wants efficient performance for search or for insertions depending upon the frequency of retrievals and updates. In this project we aim at comparing the serial implementations of these indexing methods with one another and then identify the the scope of parallelizing log structured merge trees to compare the speedups one can achieve over the best serial indexing method.

2 LITERATURE REVIEW

There are several key-value storage data structures like B-Trees, Log Structured Merge trees (LSM Trees), and Fractal Trees. We surveyed these data structures and their insertion and deletion algorithm, and GPU implementation of LSM Tree.

2.1 B Trees

B-trees was invented by Rudolf Bayer and Edward M McCreight [4] in 1972. B-tree is a tree data structure that keeps data sorted and allows searches, sequential access, insertions and deletions in logarithmic time. The B-tree is a generalization of binary search tree where a node could have more than two children.

Each B-tree node contains variable number of keys and can have variable number of child nodes, within some predefined range. When data is inserted or removed from a node, it's number of child nodes may be split or joined. Because a range of child nodes is permitted, B-tree does not need re-balancing as frequently as self balancing binary search tree.

2.2 Fractal Trees

Fractal trees (FT) are based on research on streaming B-trees [5]. FT indexes appear in Tokutek's database products. FT is very similar to B-trees except it has extra buffers. When a data record is inserted into the tree, instead of traversing the entire tree the way a B-tree would, we simply insert the data record into the buffer at the root of the tree. Eventually the root buffer will fill up with new data records. At that point the FT index copies the inserted records down a level of the tree. Eventually the newly inserted records will reach the leaves, at which point they are simply stored in a leaf node as a B tree would store them.

Having a buffer data and traversing elements down together ensures that leaf blocks remain contiguous and does not end up scattered over the disk. This allows FT to run much faster than B-trees for high entropy inserts and lookup.

2.3 Log Structured Merge Trees

2.3.1 ***Serial Implementations:*** Log structure merge trees provide an efficient method for frequent inserts by introducing a deferred and batched index insert mainly due to better locality of reference given I/O is done in

Authors' address: Raghav Gupta, raghav18076@iiitd.ac.in; Suchet Aggarwal, suchet18105@iiitd.ac.in.

	B-Tree	**Sorted Array**	**FT**	**LSM**
Insert	$O(log_B N)$	$O(N)$	$O(log_B N/B^\epsilon)$	$O(log_B N/B)$
Delete	$O(log_B N)$	$O(N)$	$O(log_B N/B^\epsilon)$	$O(log_B N/B)$
Search	$O(log_B N)$	$O(logN)$	$O(log_B N)$	$O(log^2 N/B)$
Count/Range	$O(log_B N + L/B)$	$O(logN + L/B)$	$O(log_B N + L/B)$	$O(log^2 N/B + L/B)$

Table 1. Summary of the theoretical complexities (Serial Implementations) for B-Tree, Sorted Array, FT and LSM. B is the cache-line size, N is the total number of items and L is the number of items returned (or counted) in a range (or count) query in terms of block transfers [3]

contiguous blocks. The main idea behind LSM trees is that disk I/O is costly if done randomly, but sequential block memory transfers are efficient. O'Neil et al [6] first presented the idea of LSM trees as a data structure composed of two or more tree like data structures. The authors propose that to mitigate disk I/O overheads the data is first inserted in a tree $[C_0]$ resident on the main memory, which eliminates I/O costs, and once this tree reaches a threshold capacity, its contents are then merged/transferred using the operation they terms as the "rolling merge" to the tree $[C_1]$ (which has a structure similar to a B-Tree) resident on the disk as multi page disk block, this being contiguous has much better efficiency than random disk writes.
Other operations like deletes and updates are also efficient but unlike the traditional B-Trees, search operations become costly as both the trees (C_0 and C_1) are needed to be hierarchically searched. The authors provide a detailed cost analysis of LSM trees and B-Trees stating that the LSM trees are more efficient in insertions, updates and deletes but searching is slower. They extend the two component structure to a multi component (having more than 2 levels of trees). Having multiple trees helps avoid frequent "rolling merge" operations as the threshold size for C_0 depends on the average number of entries in C_0 inserted in a single page leaf node of C_1. And when C_1 is bigger in size, more than one page node is brought in memory for merging and thus making the threshold smaller (making merge operations more frequent)

2.3.2 ***Parallel Implementations:*** O'Neil et. al [6] identify the conflicts one needs to eliminate to provide concurrency in the basic implementation of LSM Trees. These include conflicts of insert and search operations when existing processes are merging consecutive trees (C_i, C_{i+1}) and conflicts within the merge operations between consecutive pairs (C_{i-1}, C_i) and (C_i, C_{i+1}). Konstantin Kuznetsov [1] provides parallel implementation of a write optimised index on LSM trees on GPU wherein to improve search costs, the individual trees are split up into sets of subarrays with sizes in increasing powers of 2 (i.e. $N_i = 2^i$ where i is the index of the subarray and N_i is the number of elements the i^{th} subarray can hold). Searching can then be parallised by performing binary seach over all sub arrays in the tree in parallel, achieving an acceleration of more than 5 times over the typical implementation. Peng Xu [2] present GPU-LSM, a GPU dictionary data structure that combines LSM with Cache Oblivious lookahead arrays (COLA). Essentially using sorted arrays in place of the general B-Tree structure at every level, the authors state this is done because B-Trees are not cache oblivious and hence not suitable for GPU's as they have relatively small cache size. They also provide implementation details for insert/delete, lookup, count and range operations on the GPU.

3 MILESTONES

The identified milestones for the project are mentioned in Table 2

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

Table 2. Milestones for the project

4 APPROACH

We implement Search optimised index based on log structured merge trees. For optimising the index for search the cost for insertion is increased however, the search time reduces. For doing so one has to maintain data in sorted order in every level (if using a multilevel LSM Tree).
The LSM Tree is composed of $LEVEL$ levels where the first level is the $zero^{th}$ level and it is where insertions are made. The rest of the levels are updated once the levels prior to them fill up/empty. Each level in the LSM Tree can contain twice as many elements as the previous level.
For efficient parallelisation of LSM-Trees on GPU, we create a LSM-Forest, i.e. A collection of identical LSM-Trees, mainly to leverage the advantage of multiple workers. We maintain the runs/levels of the LSM trees in the form of sorted arrays, since arrays are cache oblivious and thus are better suited for GPU's. This can greatly improve searching and counting operations on the GPU. For optimising Insertions and deletions we propose the following algorithms:

The LSM-Trees has four elementary operations which are discussed below:

4.1 Insertion:

For the standard CPU implementation, insertion always takes place at the lowest level (C_0), If the C_0 overflows, we use the rolling merge operation to move data up the levels while maintaining the sorted order of the data. For parallelising the insertions, we have the following things for consideration:

(1) The insert at level 0, has to be such that the level maintains the sorted order
(2) A merge operation from level C_i to C_{i+1}, must occur before any merge can occur beyond the $i + 1^{th}$ level
(3) The merge operation can be replaced with parallel copy from one level to the next and then sorting the next level

4.1.1 GPU Sorting. We have used bitonic sort for sorting array on GPU. Bitonic sort is a comparison based sorting algorithm that can be run parallely. It focuses on converting a random sequence of numbers into a bitonic sequence, one that monotonically increases, then decreases. Rotations of a bitonic sequence are also bitonic.

The append method, copies one element per thread from C_i to C_{i+1}, and the sort method is as follows:
For sorting the arrays using the GPU, we use Bitonic Sort, which can be efficiently parallelised, The algorithm for

Algorithm 1: Insert in LSM Trees GPU

```
INPUT: Key K to be inserted, LSM Forest T = {$T_0$, ... $T_{Trees-1}$};
bucket = Randomly pick a index for the tree this key is inserted into;
$T^{'}$ = T[bucket] = {$C_0$, ... ,$C_{k-1}$};
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 is as defined below:

Algorithm 2: Sorting an Array on GPU

```
INPUT: An array A = {$A_0, A_1, ..., A_{N-1}$};
for $k \in \{2, 4, 8 \ldots, N-1\}$ do
    for $j \in \{k/2, k/4, k/8 \ldots, 1\}$ do
        Each Element is Compared between j and k using Algorithm 3 (Bitonic Kernel(A, j, k, N));
    end
end
Result: Array A with it's element sorted
```

Algorithm 3: Bitonic Kernel

```
INPUT: An array A = {$A_0, A_1, ..., A_{N-1}$}, index j, index k, size N;
index i ⟵ global index of thread in array;
index l = BITWISE XOR (i, j);
val A = BITWISE AND (i, k);
if $i < l$ then
    if (A == 0 AND A[i] > A[l]) OR (A != 0 A[i] < A[l]) then
        SWAP(A[i], A[l]);
    else
    end
else
end
```

Analysis and Limitations: Parallel run time for bitonic sort is $O(log^2(n))$. The outer two loops run for $O(logn)$ operations each. And the CUDA kernel takes constant time. For insertion in the LSM tree, the insert takes place in C_0, thus taking atmost 127 steps (inserting into sorted array) in the current implementation. The overhead of the merge operation is increased over the standard LSM Tree (which do not maintain sorted runs) and over the serial implementation as well, since here we first append the lower level in the next level, and then sort, instead of merging the two, this is more work but however the overall runtime is reduced as the array is appended in constant time, and the sorting takes $O(log^2(n))$. Thus the current implementation is fast but is not cost efficient.

4.2 Deletion:

Deletion suits an optimal purpose for GPU, as one can parallelly search in all trees and all levels within a tree for the presence of the element. If found the element is removed (all elements in front are moved behind to maintain sorted order). Here one One block is assigned to one tree within the forest, and then each thread within a forest works on one level. Since each level is sorted, one can do binary search

Algorithm 4: Deletion in LSM Trees GPU

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

Analysis and Limitations: The above algorithm searches in all the trees and in all levels within a tree, and deletes some occurrence of the given key. For doing so the current implementation searches using one thread per element and if a match is found that index is returned and the corresponding element is removed. This can be speeded up using some better searching technique. This would be implemented when searching for LSM Trees on GPU is implemented. In the worst case all trees are searched in parallel and all levels within each tree is searched in parallel and when a match is found that element is removed.

4.3 Search:

To be implemented in the next evaluation

4.4 Count:

To be implemented in the next evaluation

5 RESULTS

We aim to evaluate our GPU implementation with CPU implementations of LSM-Trees, Btrees and Fractal Trees. The current evaluation is presented for insertions and deletions. The current GPU implementation is naive and is not fully optimised i.e. does not minimise divergence, maximise occupancy etc. The optimised version of LSM trees will be presented in the final evaluation.

5.1 Timing Results

	Insertion	Deletion	Search	Count
CPU (B-Trees)	0.000337428	0.00167183	0.000454622	0.000693331
CPU (LSM Trees)	0.00049103	0.0058738	0.00579687	0.00591298
GPU	0.00030427	0.00123993	-	-

Table 3. Timing Results (All figures reported in milliseconds)

Table 3 gives the timing results for the three approaches. All times measured are in milliseconds and represent the average insertion/deletion time for one insertion/deletion. This is averaged over insertions/deletions of 100K keys.

	Insertion	Deletion	Search	Count
GPU vs CPU (B-Trees)	1.11x	1.30x	-	-
GPU vs CPU (LSM Trees)	1.61x	4.73x	-	-

Table 4. Speed ups

As seen from Table 4, the we observe comparatively high speedups for deletions, given we can effectively parallelise the operations involved in deletions as mentioned in Section 4.2. For serial implementation, we observe that B-Trees perform much better than LSM Trees, but the parallel version performs slightly better. This speedup is observed on the naive implementation that has not been fully optimised.

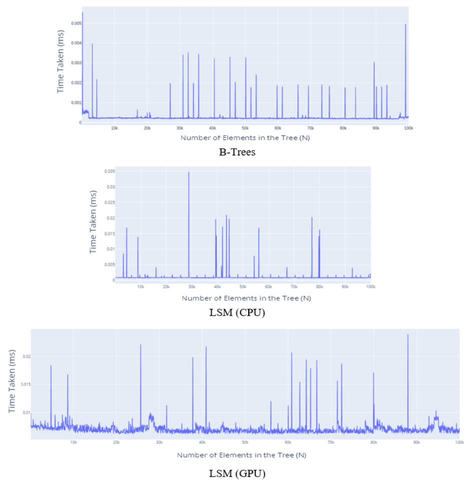


Fig. 1. Average Insertion Time (for the three approaches) for one insertion versus no of keys in the Tree/Forest

As seen from Figure 1, the insertion time variation over the number of elements in the index, does not vary heavily but has occasional spikes, this trend is observed across all the three implementations. This can be attributed to the splitting of nodes (in case of B-Trees) and the rolling merge operation (in case of LSM-Trees). For deletion (figure 2), 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.

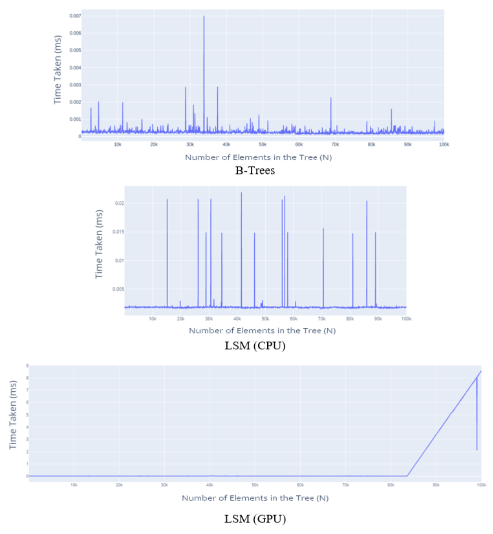


Fig. 2. Average Deletion Time (for the three approaches) for one deletion versus no of keys in the Tree/Forest

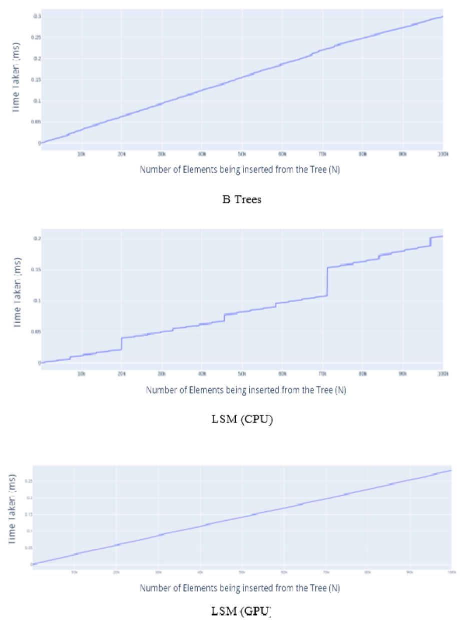
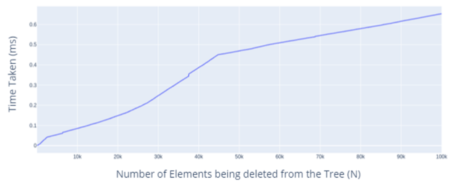
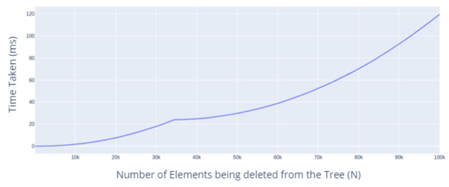


Fig. 3. Total Insertion Time (for the three approaches) versus Input Size (no of keys)

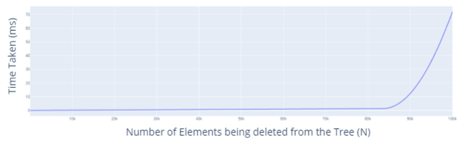
Figure 3, 4 denote the variation of total time of insertion and deletion all keys in the forest/tree with the number of keys inserted/deleted. For all three approaches for insertion we observe a linear increase in the time taken as the size (N) increase. For deletion, for the serial implementation the increase is somewhat linear with slight



B Trees



LSM (CPU)



LSM (GPU)

Fig. 4. Total Deletion Time (for the three approaches) versus Input Size (no of keys)

deviations in the middle, a similar trend is observed for GPU implementation as well, however the time increase non linearly for larger number of keys being deleted. 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.

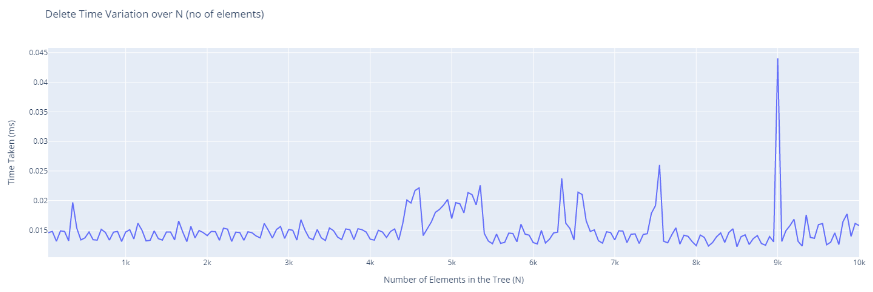


Fig. 5. Average Deletion Time for one deletion versus no of keys in the Forest (Maximum Keys = 10k)

This claim can further be backed up by figure 5, where only 10,000 keys were inserted keeping the number of trees in the forest and the levels within each tree constant. No such spike was observed in this case.

Further for investigating the performance benefits from GPU, we plot the cumulative insertion and deletion times for GPU, varying the number of trees in the LSM Forest (in Figures 6, 7). 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, so there is not much dependence on the number of trees. But for deletions we

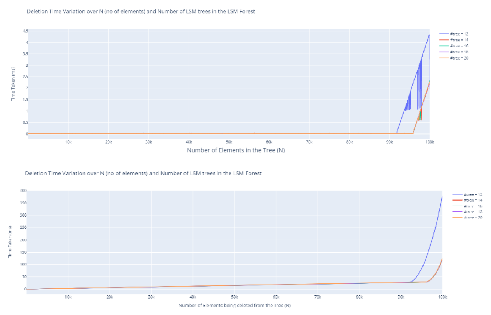


Fig. 6. Average Deletion Time (above) for one deletion v/s number of keys in the forest and Total Deletion Time (Below) versus no of keys deleted for different number of trees in the forest

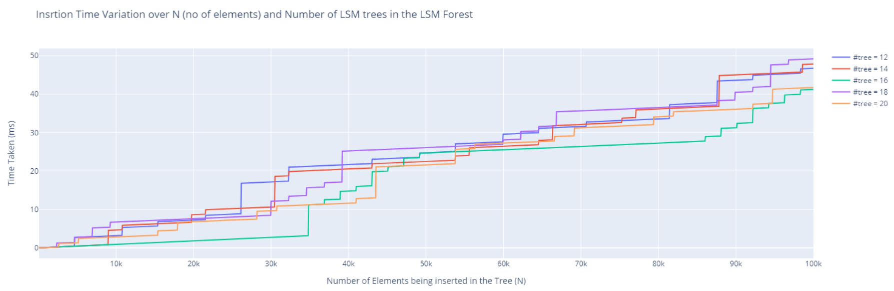


Fig. 7. Total Insertion Time v/s No of keys inserted for different number of trees in the forest

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. This further strengthens the hypothesis of increase in deletion time due to global synchronisation as stated above. From Figure 6, we get that the setting with number of trees equal to 16 starts increasing when the number of keys in the forest are higher than other settings, and this increase is the slowest. This could be attributed to the fact that since one block is assigned to one tree, and the threads within a block to individual levels, the total number of threads become a multiple of 32, which may reduce thread divergence to some extend. Further launching only 12 blocks for deletion, might not have very high occupancy either.

REFERENCES

[1] [n.d.]. Write Optimised Index based on Log Structured Merge Trees on GPU. https://on-demand.gputechconf.com/gtc/2017/posters/images/1920x1607/GTC_2017_Algorithms_AL_14_P7251_WEB.png. Accessed: 2021-03-29.

[2] S. Ashkiani, S. Li, M. Farach-Colton, N. Amenta, and J. D. Owens. 2018. GPU LSM: A Dynamic Dictionary Data Structure for the GPU. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 430–440. https://doi.org/10.1109/IPDPS.2018.00053

[3] Muhammad A. Awad, Saman Ashkiani, Rob Johnson, Martín Farach-Colton, and John D. Owens. 2019. Engineering a High-Performance GPU B-Tree. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 145–157. https://doi.org/10.1145/3293883.3295706

[4] R. Bayer and E. McCreight. 1970. Organization and Maintenance of Large Ordered Indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control* (Houston, Texas) *(SIGFIDET '70)*. Association for Computing Machinery,

New York, NY, USA, 107–141. https://doi.org/10.1145/1734663.1734671

[5] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. 2007. Cache-Oblivious Streaming B-Trees. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures* (San Diego, California, USA) *(SPAA '07)*. Association for Computing Machinery, New York, NY, USA, 81–92. https://doi.org/10.1145/1248377.1248393

[6] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.* 33, 4 (June 1996), 351–385. https://doi.org/10.1007/s002360050048