SLIMDB: A Space-Efficient Implementation of Logic Structured Merge Trees

Members: 201801162 Vatsal Gujarati

1. Introduction

- **Topic**
  Here, we will broadly discuss the topic of implementation of LSM trees using SlimDB and will see how it is better than other methods of implementation of LSM trees.

- **Motivation**
  The speed of read and write operations on solid state drive have been an issue of concern and there were many methods used to improve it. One of the best method was of key-value stores, it uses compact in-memory index and write optimized indexes to speed write operations and for reading it first searches for the key related to the data and searching of the single key in place of whole data takes less time.
  LSM (Log-Structured Merge) tree is one such write-optimized index which provides index to write intensive data. The implementation of LSM trees by the current methods are not optimized like they are poor at read operation. The main goal is to have an implementation method which has less read/write amplification rate and is storage-efficient and one such implementation of key-value store is SlimDB.

2. Statement of Purpose

- To make the implementation of LSM trees more space, read and write efficient by replacing the current filters and indexes with the new optimized ones which would be more efficient in many ways.

3. Description

- **LSM Tree**
  Logic structures merge trees are the data structures used for implementation of key-Value type databases. The main reason for their usage is that they support very high write rate compared to the traditionally used ones like the B-trees.[1]

In B-trees, for making even a small update it takes many random writes, which makes it highly inefficient for storage with limitations. On the other hand, LSM trees contain multiple sorted tables and each of them are created by sequential writes and are often implemented as Sorted String Table (SST) [2].
Nowadays, many databases require key-value store to be both read and write efficient. One of the research papers of yahoo has shown that the ratio of write to read operation has came down to 1 which used to be between 2 to 9. From this we can see that we require data structure which is both read and write efficient and one such is Logic structed merge trees. One the problem lies in the implementation of them and we have many ways of implementing them but need a way which is storage efficient.

- **Implementation of LSM using LevelDB**
  A LSM tree contains multiple sorted tables called SSTables (Sorted String tables) which appends data sequentially. Now, these SSTables are given a level and there are multiple levels. The level is assigned on the basics of time when the key is inserted. LevelDB uses in-memory buffer and multi-level structure. When a new entry is inserted than it is first inserted in the in-memory buffer and when the buffer becomes full then the entry and all other updates gets inserted into the disk and then it generates a SSTable at level 0. When the data limit of a SSTable is reached than the data is then merged with the SSTable of the next level and this process of moving of data from one level to another is called Compaction [3]. As it uses sequential writes so this process is more write efficient than other process which uses random writes.

- **Implementation of LSM using Stepped-Merge Algorithm**
  Whole process of implementation remains same, the only change lies in the levels and the process of compaction. In the above process of compaction, we merge a SSTable of level i with a SSTable of level i+1 but we can have some variance in that, if we break the original SSTable into r sub-SSTables and then store data in them and at the time of compaction they are first r-way merge sorted with themselves and then they are inserted into the next level. This algorithm is called Stepped-Merge Algorithm. By this method we can reduce write amplification. But by this method the read performance degrades as now it will have to look for the entry in various overlapping sub-levels of a single level.

- **Block Index and Bloom filter**
  The above problem of read latency can be solved by using bloom filter and block index. Bloom filter is a data structure, which can tell whether the key is present in the data block or not in a memory efficient way [4]. It is a probabilistic data structure as it is not always correct. Now, if the key is present

in the filter then it will now check the index block related to it and after this it will check in the index block and will eventually find the data block.

- **SlimDB: Design and Implementation**
  SlimDB unlike LevelDB also works for semi-sorted data. For this, it uses Three-Level block index in place of the traditionally used block index as it is most efficient for data which is semi-sorted. In place of bloom filters here Cuckoo filter for multiple levels is used as it has the functionality of returning the most recent sub-level which contains the required key. It is also a probabilistic data structure as it can also give wrong answer.

- **Sorted String Tables: Space Efficiency**
  In LevelDB, if we don't cache the block index then for reading any entry from the sorted string table, we require 2 blocks of data in which the first one is used for loading the data and the second one is used to read the data. In LevelDB, in each block of data as we store full key, so if we take an average of space required for storing the key comes out to be 8 bits. It takes a lot of space. To overcome this problem, SlimDB only supports the prefix or the suffix of the semi-sorted data and now due to this the keys need only be sorted by the suffix or the prefix not by both. To implement this, we will use ECT (Entropy-Coded Tries). This process makes SlimDB more efficient than the LevelDB. It reduces the space used by 4 times the space used by LevelDB.

- **ECT (Entropy – Coded Trie)**
  Main functionality of it is to assign a unique prefix for each unique data input as by this the data is compressed and by this, we can reduce the number of CPU cycles for the data. ECT is a radix tree where each internal node represents longest common prefix shared by all the nodes under this internal node and the external node or the leaf represents one key of the set. It assigns a rank to each input key of an array which is sorted.

- **Three-Level Index Design**
  As we know that SlimDB only works for the data which is semi-sorted, so we cannot use ECT alone as the block index as in SlimDB both the key fragments are hashed so we can use ECT individually on both of them. Three level indexes are generally made in 3 stages and each stage corresponds to an index. First, we will make a vanilla block index which contains the first key and the last key of all data blocks and now will try to compress it. First, we will store the prefix of all block keys and keep the unique entries and remove the duplicate ones and will form a prefix array. We will now use ECT on the prefix array to compress it and it will now form our first level index and this will be treated as input for the second stage or so-called second index.

In the second level, it will now take the rank from the prefix array and try to map it to the data entries in the SSTables which have the same prefix. By using these two levels we get of list of SSTable blocks which contains the keys prefix.
In the third level, first we will repeat the process done in the first level on the prefix but now will do the same process on suffix and on the SSTables blocks which we get from the second level. Now we will do a binary search on all the SSTable blocks which we get after the above process and will find our key which we are looking after. This full process defines the work of Three-Level index and we can see that it is more storage efficient than the traditionally used block index. Average cost of storage per SSTable block is no more than 10 bits

- **Cuckoo Filter for Multiple Levels**
  In-Memory filters are probabilistic data structure which can tell us whether a key is present in the data block or not. Till now they had been using bloom filters which have a false positive rate and it can lead to increase in read latency. For overcoming this issue, we will now use cuckoo filter at multiple levels, as it has the functionality of reducing the number of reads in the case of false positive answers. It also has properties like fingerprint-based filtering and is highly memory efficient. Cuckoo filter contains a hash table which can be seen as a linear array, in which each key has two candidate buckets and we need to implement different hash functions for both of them. When any new data is inserted than at that time it looks for an empty candidate bucket and if it is not found than it extends the size of the hash table and this process goes on. In the cuckoo filter it stores the full key but than it would make it space-inefficient. So, to reduce the space it stores hash fingerprint which is of constant size in place of the original inserted key. But this process comes with its own limitations as insert functions of the standard cuckoo filters are designed for the original key not its fingerprint. So, to overcome this we use the fingerprint to manipulate the entry data of both the candidate buckets. By this both the candidate buckets will also be less independent of each other and this leads to higher collision rate. Now, we can see that this filter is far more space efficient than the other traditionally used ones and it also has a higher collision rate than the bloom filter. It consists of two tables primary and secondary table. Primary table stores the fingerprint value and the level number of each data entry. This can result in conflicts as if two levels are assigned the same fingerprint than for that entries, we would need to traverse the full disk read and this would reduce the read latency. We can overcome this issue by assigning unique fingerprint for different levels. Now, we can store full key along with its respective fingerprint in the secondary level and by this read latency will also get reduced. Algorithms for the methods of lookup and insert are made by keeping in mind the consistency of the LSM tree.

- **SlimDB: Implementation**
  Its implementation is same as that of RocksDB. It just has some code changes in it. During prefix scan, as SlimDB uses stepped merge algorithm, it needs to maintain a SSTable iterator for each sub-level. Due to this it is a bit slower than LevelDB.

- **Conclusion**
  We can thus conclude that generally the LSM trees are read, write and storage inefficient and thus we present new technology like SlimDB which improves the insertion process and provide a better structure for lookup process. Due to improved methods of storage and insertion and update of data in it, it is more main-memory space efficient.

- **Future Works**
  They can have some changes in future where they can remove the SSTable iterator which they have used as it makes the scanning process a bit slower than LevelDB and after this, it can replace the current ones like the LevelDB or RocksDB and perform better than them. I would have liked to describe the evaluation phase but due to space constraint could not do it.

- **Reference**
  [1] https://dev.to/creativcoder/what-is-a-lsm-tree-3d75

  [2] https://stackoverflow.com/questions/2576012/what-is-an-sstable

  [3] https://practice.geeksforgeeks.org/problems/what-is-compaction-is-os#:~:text=Compaction%20is%20a%20process%20in,empty%20spaces%20together%20and%20processes.

  [4] https://llimllib.github.io/bloomfilter-tutorial/#:~:text=A%20Bloom%20filter%20is%20a,may%20be%20in%20the%20set.

  [5] https://en.wikipedia.org/wiki/Entropy_encoding

  * Ren, Kai, et al. "SlimDB: a space-efficient key-value storage engine for semi-sorted data." Proceedings of the VLDB Endowment 10.13 (2017): 2037-2048.