Implementing Storage System Using Segmented LSM Tree

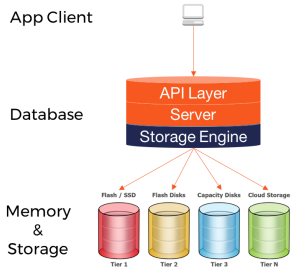
106120025
Brintha M
neof7@gmail.com

106120135
Udipta Pathak
udiptapathak13@gmail.com

Group No. 38

1 Introduction

1.1 Problem Statement



Storage engine is the core of a storage system. There are three main storage engine currently: hash storage engine like Redis and Memcached, LSM tree storage engine like LevelIDB and HBase, and B+ tree storage engine,used in traditional relational databases. B+ trees suffers from seek and rotational latency of the r/w head of hard disk, due to frequent disk i/o calls. The objective is to:

1. To create a storage engine (lsm tree based) that solves the problem of seek latency in hard disk due to frequent io calls

2. To optimise the engine by using hash table (segmented lsm)

1.2 Necessity of Optimizaton

among non relational databases, key value databases is an important class and key value store is the core of the key value database. As the need for more efficient data storage

and retrieval techniques increases, so does the value of key-value stores.

Data is increasing steadily in quantity, and key-value stores provide the speed needed to manipulate this data at high proficiency. The major advantages of key-value stores are scalability, speed, and flexibility. The performance of the storage engine directly determines the performance of R/W operations.

Traditional file systems utilizing a spinning disk, spends only 5-10% of disk's raw bandwidth whereas LFS permits about 65-75% in writing a new data.

As a widely used key storage engine, the LSM tree is inspired from log-structured files. its usage is mainly to solve the problem of disk I/O in B+ trees introduction of ram index table of hash storage structure will improve index performance of the lsm , thus improving the read operation speed of the system.

This hybrid storage engine is called segmented lsm. given limited ram space, optimisation of key length and use of efficient data structure for indexing is important. search for such a data structure and hashing function thus plays an key role in performance of the engine.

1.3 Existing Solution and Their Problems

1.3.1 B+ tree

It is a multiway search tree, with values stored in the leaf nodes and keys in the other nodes guiding way down to the leaves.It always maintains a unique copy of each key and thus space efficient. The read and write operation has a series of I/O calls to locate the address of the value corresponding to a given key. This is solved by a lsm tree.

1.3.2 LSM

The log structured file for write queries helps to reduce time required for write operation to o(1) (although the write amplification is $O(g \log_k(n))$, where k is the number of levels, n is the total number of nodes, g is the growth factor). However, the read operation is now $O((log_k(n))^2)$. Thus, sLSM was born.

1.3.3 Segmented LSM

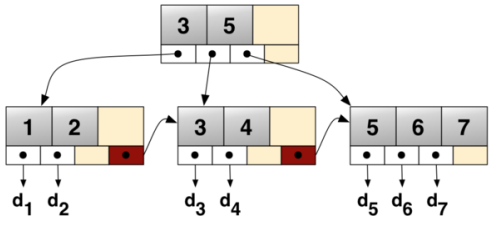
Segmented LSM maintains hash index tables in RAM which provides the physical address of the key value pair in the memory. This reduces the search complexity to $O(\log n)$ and thus making read faster. It also introduces a logical addressing scheme to reduce the number of rewrites in case of compaction or deletion. However, maintaining all keys in RAM and updating them in case of change in physical address increases resource utilisation.

1.4 Our Solution: Hybrid sLSM

Here, we use RAM index table just for the level 0 of the storage. The levels following it are arranged as a kind of an m way tree. Each node has a series of sTables. Each sTable has a maximum size of m, which depends on number of levels and the scale of the database. The background thread maintainsthe sTable with size $m - k \times p$, where k is the number of elements in the write buffer and p is decided based on the dataset and the tree in such a way that each node has at most $m-1$ nodes and a minimum of $\frac{(m-1)}{2}$ nodes (except the leaf nodes), where m is the order of the tree. When a sTable is filled, the thread reduces its size by compacting with its child or it can make a new sTable in the same node, depending on the size of neighbouring sTables and the number of elements in the node. The height of the tree is maintained in a way that the longest and shortest path from root to leaf differs by no more than 3.

2 B+ Trees

2.1 Description



B+ tree is a data structure with insertion, updation, deletion and searching in logarithmic time complexity. It is improved over B tree which keeps key and value in each of it's node while B+ tree keeps their value only at the last level of the tree, while all the intermediary nodes only have keys, and their value pointing to null.

A B+ tree can keep k elements in each node, making it a k-ary tree, and if one has the knowlegde of the amount of data one is planning to keep in a B+ tree, one can choose a value for k so as to optimise the $k \log_k n$ value.

If our B+ tree is k-ary, all children and internal nodes must have atleast $\lceil \frac{k}{2} \rceil$ records and children nodes respectively. Only the root nodes must have atleast 2 children, but if it is the only node in the tree, it will hold records, and can hold any number of records. All the number of children and record are bounded by k from above.

2.2 Application

B+ tree is used to store huge amount of data that cannot be stored in the main memory.

It provides faster insertion, deletion, updation and search compared to other basic trees.

It is used in multilivel indexing, and database indexing.

It is used as the underlying data structure for many database management system.

2.3 Operation

2.3.1 Insertion

1. Tranverse the tree to reach the appropriate leaf node to insert the new element. This takes $O(k \log_k n)$ time.

2. In case of no overflow, just insert the new element in that node and one is done.

3. In case of overflow, one need to create another node and split the entries from between distributing them between the nodes equally. One also needs to update the parent intermediary node too in this case.

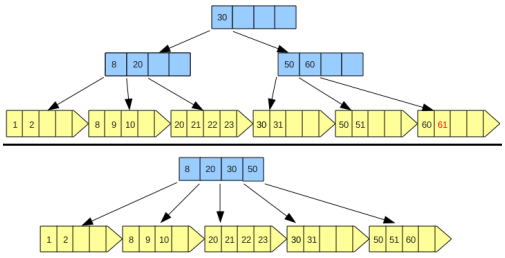
4. Overflow can be caused in the parent intermediary nodes too, in which case we need to split them, which has to be updated in their parent, possibly causing a chained spliting along the path. This can take $O(\log n)$ again.

2.3.2 Updation

1. Search through the tree to reach the entry to be updated. Searching will take $O(k \log_k n)$ time.

2. Once the element has been reached, one can just change the value as required.

2.3.3 Deletion



1. Search the entry to be deleted, this will take $O(k \log_k n)$ of time.

2. If the elements in the node is satisfy more than the minimum number of elements criteria, just delete it.

3. If the elements just satisfy the minimum number of criteria, delete it, and take an element from immediate sibling, or merge entirely with it if needed. If it is an k-ary B+ tree, this will take $O(k)$ time. So overall complexity is $O(k \log_k n)$

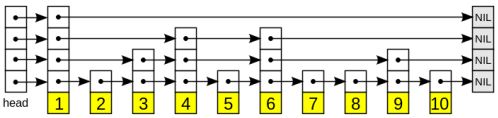
2.3.4 Searching

1. We will search for the key of the element in a given node, this takes $O(k)$ time, and tranverse down the tree in the required range till we find ourself in the desired leaf.

2. On reaching the required leaf, we can find the key matching our target if it is present. This takes $O(k \log_k n)$ time.

3 Skiplist

3.1 Description



Skiplist is a data structure that can be seen as probabilistic self balancing tree. It is better than common self balancing trees as it probabilistically decides pointer change instead of the hefty tree rotation employed by traditional self balancing trees.

Skiplist are much easier and shorter to write than the traditional self balancing trees and yet has same time complexity of insertion, search and deletion as them.

It's structure looks much like linked list, difference being in the fact that each node has random number of pointers pointing forward to the next node based on which level the pointer belongs to. An n level node has n pointers in it.

3.2 Application

It can be used as a more efficient and easy to code replacement for the traditional self balancing trees.

Hence it is used as data storing structure that stored that data that needs to be frequently accessed, modified or deleted.

3.3 Operation

3.3.1 Insertion

1. Transverse to a node that is just behind the smallest key greater than the key we intend to insert at any given height. This takes $O(\log n)$ time.

2. While transversing, before decrease our pointer height, if our level is less or equal the level of node to be inserted, we will change our pointer to point our new node, and our new node pointing to the node that was pointed by the former node. Changing pointer takes $O(1)$ time.

3.3.2 Updation

1. Tranverse to node with the same key as the key whose value needs updation. This takes $O(\log n)$ time.

2. Change the value as required. Changing value takes $O(1)$ time.

3.3.3 Deletion

1. Tranverse to the node just being the node that needs deletion. This takes $O(\log n)$ time.

2. While transversing, if our next node is the node that needs to be deleted, change the next pointer to point the node being pointed by the "to be deleted" node.

3. At the zero height, remember the address of the node to be deleted, change the pointer, and then delete the node.

3.3.4 Search

1. If our current node or the next node being pointed has the same key as targetted, our search is successful.

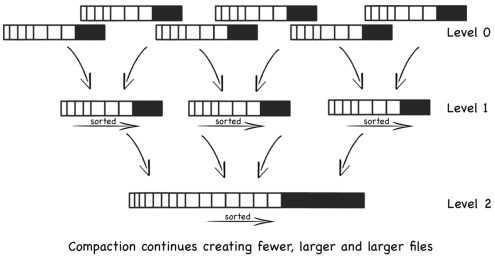
2. Keep moving forward till we reach a node that is just bigger than our targetted key.

3. On reaching such a node, decrease our pointer height if possible and repeat step 2.

4. If pointer height cannot be further decreased and we did not had a successful match, the search was a failure.

4 LSM Trees

4.1 Description



LSM tree, short for log structured merged tree is a multilevel data structure. The meaning of it being log structured is that most of the operations are done at the end of the structure.

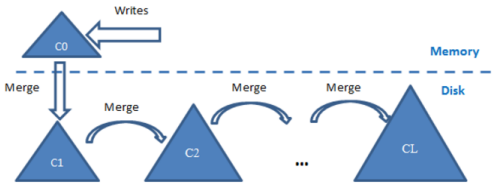
It has at minimum two levels, only one level is kept at the main memory and we will call it the mem block. And atleast one level is kept at the secondary storage and we will call it sBlock.

Having atleast two level, one at main memory and atleast one at secondary storage help us reduce the bandwidth of queries. Making a query directly to the secondary storage would cause an I/O interrupt decreasing time efficiency. So we decide to cache all the recent entries in our memblock to some extent.

The writing operation in LSM is super quick but the read operation can be hefty, and that is what we aim to optimize in our project by using segmented indexing.

Operations like insertion, updation and deletion in LSM are as easy as appending new entry at first non empty cell of an array, which is our memblock, and hence has a strict time complexity of $O(1)$.

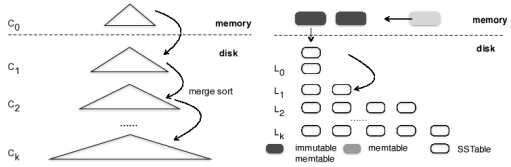
Searching would require us to linear scan memblock, and binary search in sTable, and hence is hefty.



We can improve the search operati0n by merging two sTables into a single sTable so we can have more benefits from using binary search on it. This operation is handled by a thread running in the background and this process is called **compacting** or **merging**.

As we keep merging the sTable, they

grow in size, and we keep them in next level, and as they keep merging, their levels are increased accordingly. One has to make sure that they don't grow too big that merging consumes more clock cycles than actual queries do.



Another way to improve the search operation is to use bloom filters which is a probabilistic data structure to tell if an element is present in our sTable in $O(1)$.

4.2 Application

Log structured merge tree is used as one of the data structure in storage engine of a database management system.

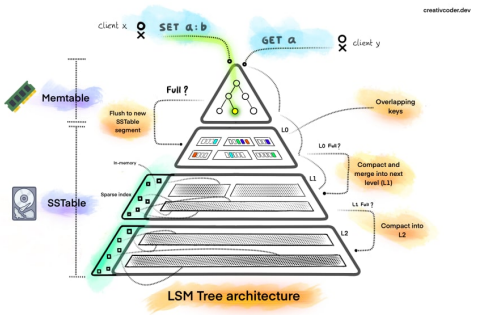
One popular DBMS that uses LSM tree is LevelDB.

LSM is used in DBMS by the intent of minimizing the I/O calls to be made to the secondary storage device for queries.

4.3 Limitation

The thread responsible for the merging operation runs in parallel to the thread that responds to the queries and hence the merging thread can eat the cycles required for query decreasing the peformance quality of the DBMS in rensponse of all the queries.

4.4 Operaton



4.4.1 Insertion

1. If the memblock (cache) in the main memory is not full, just append the new entry in it. Appending is just setting the first empty cell of the mem block (which is an array) to the new entry, and so it take $O(1)$ time to do so.

2. If the memblock is full, flush it's content to an another buffer where it will be process and sent to some sBlock residing in secondary storage. Once flushed, all the cells of memblock are empty once again to be filled and so will take $O(1)$ time for insertion again.

4.4.2 Updation

1. If the memblock is not full, just append a new entry with the same key but the new value. This takes $O(1)$ to do so.

2. If the memblock is full, flush it to the buffer and use the emptied memblock to insert the new entry with updated value.

4.4.3 Deletion

1. If the memblock is not full, just append a new entery with same key and the value pointed to null. This will be the representation we will use for the deletion query. This will take $O(1)$ time.

2. If the memblock is full, flush it's content to the buffer and the insert the new entry with value pointing to the null to the first empty cell. This will still take $O(1)$ time.

4.4.4 Searching

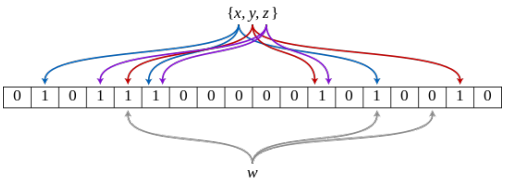
1. We will first try to search the element in the memblock and this will be a linear search from bottom to top, so if the size of the memblock is k, it will take $O(k)$ of search time.

2. In case the search fails in the memblock, we will check them in the sBlock. Since there are bloom filters to tell us probabilistically if there is an element present or not in a given sTable, and if there are n entries, it will take us $\omega(n/k)$ time to find the right sTable.

3. Once an sTable is selected, we will do binary search in it to find the target element which will take $O(\log k)$ of time. Hence the overall complexity will be $O(k + n/k + \log k)$.

5 Bloom Filter

5.1 Description



Bloom filters is an improvement over hash table (which itself can be thought of an improvement over set data structure implemented via self balancing tree) which intends to check set membership of an element.

It is a probabilistic data structure, which means that it's results are not always accurate. It returns a boolean to lookup queries. If it returns $false$, the element passed an argument does not exist in the set for sure. But if it returns $true$, there might be a chance of the element not being present in the set.

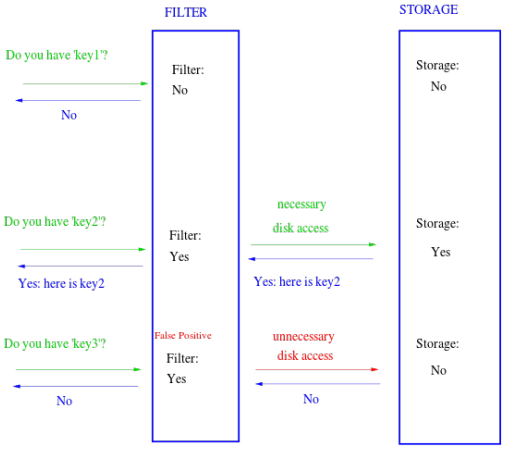
So bloom filter might sometimes give false positive to lookup queries, but chance of it happening is very low.

A bloom filter has a sequence of bits, and has two operations in it. One is to insert an element in the set, and the other is look up for that element in the set.

On inserting an element, we set few of the bits in sequence based on the positions determines by many hash functions that are employed by the bloom filter.

If we know the size of the set, we can choose the number of bits and the number of hash functions in our bloom filter in a way to minimize the probabilty of false positive to our desire.

5.2 Application

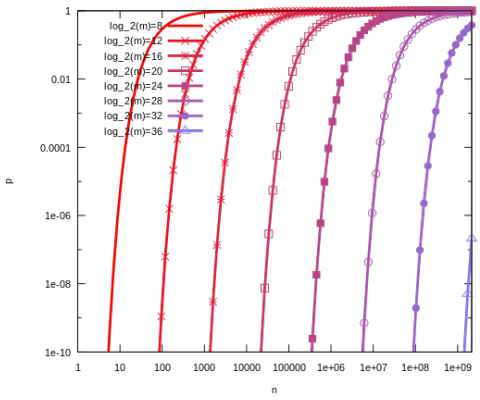


Bloom filters are preferred over hash table in DBMS despite false positive as bloom filters are very space efficient. A hash table can use a lot of space to store few elements but bloom filters can be used to test membership of a lot of elements using few bits.

Bloom filters are preferred over set implemented via self balancing trees despite false positiveness as we can test membership of an element in $O(1)$ time while it take $O(\log n)$ time in a self balanced tree.

5.3 Limitation

1. It might give us false positive sometimes.
2. The chances of getting false positive increases with the size of the set.



The chance of getting a false positive is

$$p = \left[1 - \left(1 - \left(\frac{1}{m}\right)^{nk}\right)\right]^{k}$$

where,
p = probability of getting a false positive
n = number of elements in our set
m = number of bits in our sequence
k = number of hash functions employed

Below is the plot of probability of false positive against number of element in set for different filter size.

5.4 Operation

5.4.1 Insertion

1. Take the element to be inserted into the set and hash it using k hash functions if k is the number of hash function employed by the bloom filter.
2. If our i^{th} hash function return us j (which will be bounded by the size limit

of our bit sequence), set the j^{th} bit of our bit sequence.

5.4.2 Lookup

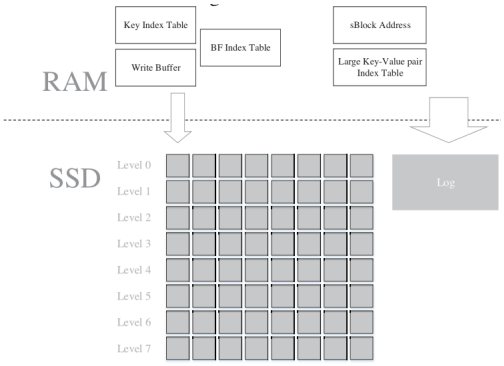
1. Take the element to be inserted into the set and hash it using k hash functions if k is the number of hash function employed by the bloom filter.

2. If our i^{th} hash function return us j (which will be bounded by the size limit of our bit sequence), check if the j^{th} bit of our bit sequence is set. If it is not set, return $false$.

3. Check all the positions generated by our hash function, if all the bits checked are set, return $true$.

6 sLSM Tree

6.1 Description

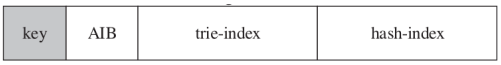


Segmented LSM tree keeps 5 tables in RAM. These are Write Buffer, sBlock address list, Key Index Table, BF Index Table and large- sized Key-Value index table.

Write Buffer is used to store Key-Value pairs which are latest written into system as a RAM buffer table. When the size of Write Buffer reaches a certain threshold, the data will be compacted into SSD.

sBlock address list is used to store the address of sBlock in the hard disk system.

BF Index Table (Bloom Filter index table) is an assistant index table, which is built in RAM to speed up queries in this paper.



Segmented index optimization is the core proposed optimization. The core index structure in the Key Index Table is Segmented-index which is divided into three parts -

1. Array Index Bit: It is used to quickly locates the data block storing the Key-Value pairs in the hard disk and reduces the frequent update of the index table caused by the compaction operation.

2. Trie Index Structure: It is used to index the location of the sTable of the Key-Value pair in the sBlock.

3. Hash Index Structure: It is used to index the relative position of Key-Value pairs in the SSD Page. The MurmurHash hash algorithm is adopted by sLSM-Tree.

6.2 Application

Segmented LSM can replace LSM because it is just improvised version of LSM with faster read operation.

Hence it can be applied to wherever LSM is currently employed, i.e. datbase management systems.

7 Performance Analysis

7.1 B+ Tree

7.1.1 Space

Let our B+ tree be m-ary. Assuming there are n nodes, level one contains m elements, and level two about $(m+1) \times m$ elements and the next about $(m+1)^2 \times m$ nodes. Thus, total number of elements is $O(m^k)$, where k is the number of levels. thus k is $log_m(n)$.

7.1.2 Search

There are $\log_m(n)$ levels and each level may have upto $O(m)$ steps. thus $O(m \log_m(n))$

7.1.3 Insertion

Traversing to the leaf takes $O(m \log_m(n))$ time and in case the node is full, splitting may take another $O(\log_m(n))$ time. Thus complexity is $O(m \log_m(n))$

7.1.4 Deletion

Traversing to the leaf takes $O(m \log_m(n))$ time and in case the node has less than minimum nodes or has children, combining or splitting may take another $O(\log_m(n))$ time. thus complexity is $O(m \log_m(n))$.

7.1.5 Space Amplifiation

Only the keys are duplicated, there is only onstance of the value corresponding to the key.

Say the ratio of size of key to key-value combined is $\frac{1}{r}$. there are $O(m^k)$ nodes excluding leaves. they occupy $\frac{1}{r}$ times the space, and thus, space amplification is $O(m^k/r)$.

7.1.6 Write Amplification

In worst case, the entire page storing a node might have to be re written, in that case the write amplification is o(b), where b is the size a node.

7.1.7 Read Amplification:

there are $\log_m(n)$ levels and each level may have upto $O(m)$ steps. thus there may be upto $O(m \log_m(n))$ disk queries.

7.2 Skiplist

Probability p of a particular node is present in next level is considered to be $\frac{1}{2}$. Therefore, given n keys, there may be a maximum of logn levels. There exists about 2^k keys in level k.

$\chi(k)$ is the random indicator variable of nodes present in level k but not in higher levels.

$\chi(k) = 2^k - 2^{k-1} = 2^{k-1}$

7.2.1 Space

Each node in level k, has pointer to k nodes.

$$\therefore \sum_{k=0}^{n-1}(\chi(k) \times k) = P(n \times 2^n) = O(n \times 2^n)$$

$\log N = n$

Thus, $O(n \log N)$ is the space complexity

7.2.2 Search

Considering uniform distribution of keys from lower to higher level, average search takes $3 \times k$ time for a node in level k.

$$\therefore \sum_{k=0}^{n-1}(\chi(k) \times k)/2^n = O(n)$$

Thus search is $O(\log N)$

7.2.3 Insert

Similar to search, the position of the key in the skiplist is searched for in $O(\log N)$ time. considering the key to be on level k, the time for creating new pointers in k levels is $O(k)$

$\therefore O(\log N + k) = O(\log N)$.

7.2.4 Delete

Searching for the keys takes $O(\log N)$ time, and deleting the pointers in k levels below it takes $O(k)$. thus, complexity is $O(\log N)$.

7.3 LSM Tree

Assume the growth factor to be g.

$\therefore$level$(g) = 4\times$level$(g-1)$

Lets take the total number of key value pairs in the entire databse is N. This is roughly the same as the the number of pairs in the last level.

Thus number of levels in the database is $\log_k n$.

time complexity: (plot graphs, the complexity is same as write na dread amplification)

7.3.1 Space Amplification

Each key may contain duplicates, those that are not yet merged and compacted. Thus space amplication is $O(n)$.

7.3.2 Write Amplification

Each piece of data is remerged into the same level about $\frac{g}{2}$ times(a level is g times bigger than its next level. the previous level is merged into the current level g times, meaning the first set of data is remerged g times and second $g-1$ times and so on.

$$\therefore \sum_{j=1}^{i})i = \frac{g(g-1}{2}.$$

There are $log_k(n)$ levels and thus, the complexity is $O(n = g\ \log_k(n))$.

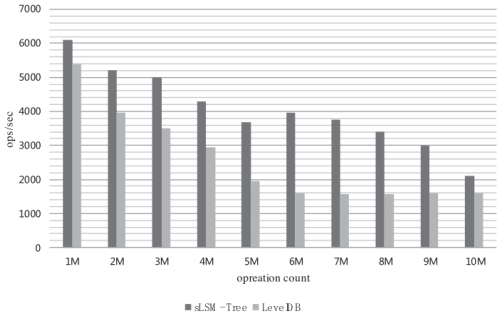
7.4 Read Amplication

One has to read the database level by level

$$\therefore \sum_{\log_k(n)}^{j=0} \log \frac{n}{k^j} = O((\log_k(n))^2).$$

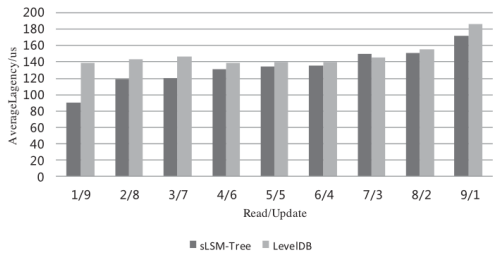
7.5 Segmented LSM Tree

7.5.1 Write



Since a log structured file is maintained for write queries the time complexity of write is $O(1)$.

7.5.2 Read



The key is looked up in the buffertable, which gives the pointer to the skiplist containing key index pair. this takes $O(\log n)$ time(search time in a skiplist), where n is the length of each skiplist. the logical address is converted to physical address using sblock address table and trie index and hash index as offset, which is $O(1)$. thus, read takes $O(\log n)$ time.

7.5.3 Space Amplification

The index tables in RAM occupies additional space over the $O(n)$ space occupied by duplicates. but since the size of a key in much smaller than size of its value, say key is r times smaller than its value, the index tables occupy $O(\frac{n}{r})$ space. thus space complexity is $O(n)$.

7.5.4 Read Amplification

Since the physical address of the key value pair is pointed to by the RAM index tables, there is only one disk I/O call. Thus, $O(1)$.

7.5.5 Write Amplification

this is similar to lsm where there are multiple instances of the keys which are later merged and compacted. Thus write is $O(g \log_k(n))$