LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data(II)

Compare with LevelDB, the LSM-trie use a prefix tree structure to organize the data into the algorithm. The goal of LSM-trie is to reduce the write amplification down to 5 (for a 5 level LSM-trie) from 55 (for a 5 level LevelDB). The LSM-trie use SHA-1 hash function to hash the key to maintain the prefix tree structure by the prefix of the hashed key and store the kv item in HTable based on the prefix and the bucket based on the suffix. To maintain the balance of the tree, it uses its infix value as the hash function to balance the movement of each kv item. By all the three-value work together, the LSM-trie can utilize a balanced prefix tree and provide a good performance. Also, because the prefix tree structure, it can provide more efficient way to check the bloom filter, which called cluster bloom filter. Because each bucket with the same position at each sublevel within a main level have the same prefix, it can physical store the bloom filters sequentially as a cluster bloom filter. So that one-time cluster bloom filter check can check the whole one level and this way we only need to retrieve the cluster bloom filter rather than keep all bloom filter in memory.

Compare with LevelDB, the main difference is LSM-trie does not have the index in the storage which save a lot of space. However, the drawback is the LSM-trie does not support range search.

(1) “The indices and Bloom filters in a KV store can grow very large.” Use an example to show that these metadata in LevelDB may have to be out of core.

The metadata here we mean are the indices and bloom filters.

For example, we have a hard disk with 10TB. If each KV item require 50B, we could store 50 billion items. For each kv item we need 10 bit-per-key bloom filter, that’s 10 bits * 50 billion = 250GB for the space required of bloom filter and for a 4KB blocks, we can store 80 of 50B KV items within one block, the index of the block is 160 bits and only one index is needed for one block. That’s mean the average index size for items is 160 bits/80 = 2 bits-per-item, which require about 25 GB. Total of space metadata required for 10TB data is 250GB + 25GB = 275GB.

The out of core means we need to keep the metadata on the disk rather than on the memory. Because, the widely used memory currently we have is about 256 GB large, which can barely hold about this amount metadata, but near future when the hard disk is getting larger, the memory cannot hold that much. Which we mean that these metadata in LevelDB may have to be not on the memory.

(2) “Therefore, the Bloom filter must be beefed up by using more bits.” Use an example to show why the Bloom filters have to be longer?
The use of bloom filter can help people to save the operation of disk read. Before the disk read, we check the bloom filter to make sure the kv item we want is here. They reason why we need to have a long bloom filter is that, because we have the algorithm to separate the architecture into multiple levels (sub-levels), for example, the LSM-trie algorithm have 5 main level with 112 sub-levels. If we use the 10 bits-per-key bloom filter, it has a possibility of 100% to get a false positive. The false positive makes people to read the disk although the item is not there, which makes a very expensive waste.

The longer bloom filter bit can help to reduce the false positive rate which increase the accurate of bloom filter. For example, if we use 16 bits-per-key bloom filter instead of 10 bits-per-key bloom filter, we can reduce the false positive rate to 5%, which means it can almost not happen. So that, the worst disk reads operation can be 2.05.

<table>
<thead>
<tr>
<th>bitskey</th>
<th>50 Levels</th>
<th>100 Levels</th>
<th>150 Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>40.95%</td>
<td>81.90%</td>
<td>122.85%</td>
</tr>
<tr>
<td>12</td>
<td>15.70%</td>
<td>31.40%</td>
<td>47.10%</td>
</tr>
<tr>
<td>14</td>
<td>6.00%</td>
<td>12.00%</td>
<td>18.00%</td>
</tr>
<tr>
<td>16</td>
<td>2.30%</td>
<td>4.59%</td>
<td>6.89%</td>
</tr>
<tr>
<td>18</td>
<td>0.88%</td>
<td>1.76%</td>
<td>2.64%</td>
</tr>
</tbody>
</table>

Table 1: Bloom filter false-positive rate.

In another hand, it leads to another concern that the bloom filter gets longer and require most space for it. We have another technique to solve this problem by using cluster bloom filter that we can retrieve very small amount of bloom filters to get the information we want. So that, although the total space is larger, we can use less information to achieve the same goal and the bloom filter gets longer would not make any issues.

(3) What’s the difference between SSTable in LevelDB and HTable in LSM-trie?

The main difference between SSTable with HTable is the HTable does not have the indices.

The LSM-trie use SHA-1 hash function to hash the key at the beginning. The algorithm dumps the kv item to the lower level according to the hashed key (hashkey). Whenever we want to do a lookup, we can use the hashkey as the location to find the corresponding position the key would be in at each level, then we use the cluster bloom filters to check each level by one-time lookup. Because each hashkey in the same cluster have the same prefix, it’s guaranteed the technique working well in real deployment.

(4) “However, a challenging issue is whether the buckets can be load balanced in terms of aggregate size of KV items hashed into them” Why may the buckets in an HTable be load unbalanced? How to correct the problem?

As people know, although the buckets in the system have the same chance to receive packet. The real time after beginning is that the count of items in different bucket has the normal distribution shape. That’s why the buckets in the HTable are load unbalanced.
The LSM-trie uses a very simple sorting algorithm to correct this problem which has two steps. First, sort the buckets according to the load of the KV pairs. Second, move from the most overloaded to the most underloaded, take off the balanced overloaded bucket and repeat this step.

This algorithm leads to three concerns.

First is how to let the user know the items in the bucket have been moved to another bucket. For each bucket, the algorithm sets a 4KB limit. The algorithm marks the infix of SHA-1 hashkey at the 4KB limits at watermark which it called HashMark. Everything beyond this HashMark means have been moved and below this HashMark means in this bucket. The algorithm records the HashMark (4 Byte) with source bucket ID (2 Byte) and destination bucket ID (2 Byte) together as the rehashing information with the HTable file. Every time we check the bucket by the infix of the hashkey and know whether it gets moved or not.

Second is how to reduce the chance that one item keeping move multiple times. This situation happened when one overloaded bucket cannot move all overloaded items to another underloaded bucket. The algorithm uses the hash function in infix to sort the order of the items in each bucket. It rotates the infix hash function by bucket ID to make sure that each bucket has a different hash function. By this method, the high position item in one bucket would not be kept in the same position in other bucket. So that it can overcome this concern.

The third concern is to deal with the item cannot be moved to another bucket. This situation happened when a large overloaded item is in a bucket and other underloaded bucket also does not have enough space for the large item. The algorithm would create a special bucket for this kind of items with fully indexed with the HTable file. Each time user retrieves information from this HTable, the special file would be retrieved at the same time. This is the reason why the LSM-trie is forcing on manage a large set of small data and eliminate almost all the indices. The large data the system has would increase the number of special buckets which reduce the performance of the LSM-trie.