Q&A Report

Fan Ni (ga1829)

Q1: “Figure 1: The memory overhead and lookup performance of SILT and the recent key-value stores. For both axes, smaller is better.” Explain the positions of FAWN-DS, SkimpyStash, BufferHash, and SILT on the graph.

In key-value (KV) store design, in-memory metadata overhead and performance usually conflicts with each other, that is, if the metadata in memory is reduced, the performance will be compromised as the metadata as well as data has to be retrieved from slower storage device such as Flash and hard disk. In existing KV store designs, different tradeoffs are carried out on these two aspects for different motivations.

![Figure 1](image)

In Fawn-DS and BufferHash, the read performance is maximized by storing more metadata in memory, about 12-byte and 4-byte per key respectively. While in SkimpyStash, the read performance is compromised (about 5 reads per lookup) because little metadata is stored in memory (less than 1-byte per key).

In Silt, the authors claims that they achieve good read performance and meanwhile minimizing memory overhead.

Q2: Two design goals of SILT are low read amplification and low write amplification. Use any KV store we have studied as an example to show how these amplifications are produced.

Read and write amplification means that more read or write operations are performed that are not benefit for the upper level applications. For example, when a read request is issued by an application, it expects that the request is satisfied by only one read to get the real data it need. While this is not always the case, because before being able to get the real data, the metadata must be retrieved first, which may incur additional read operations.

I will give two examples about read and write amplification in KV store designs.
The first example is SkimpyStash, where the linked lists are stored on flash with a pointer in each hash slot in DRAM pointing to the head of the list. When a read request is served, it may need to go through the list on flash which may cover more than one pages. So before the target KV pair is found, more than one flash read may be performed, incurring RA. Also, in order to reclaim some space, the garbage collector will incur WA when moving active items to a new place.

Another example is LevelDB, where a multi-level design is used. To serve a read request, more than one level may be involved and if the bloom filter returns false positive, read operations may be performed on level where data does not exist, and RA is introduced. And the compaction process incurs WA.

Q3: Describe SILT’s structure using Figure 2 (Architecture of SILT). Compared with LevelDB, SILT has only three levels. What's concern with a multi-level KV store when it has too few levels?

The design and implementation of SILT includes three basic key-value stores: LogStore, HashStore and SortedStore. The LogStore serves PUT and DELETE by writing sequentially to flash. For the in-memory metadata storing, cuckoo hashing are used to map keys to their location in the flash log. Once a LogStore fills up, SILT freezes the LogStore and converts it into a more memory-efficient data structure, where the in-memory hash table is removed and the KV items are stored in hash order on flash. When a number of hash tables are accumulated in HashStore, they are merged with the SortedStore to form a new StoredStore. In SortedStore, KV entries are sorted by key on flash and indexed by a new entropy-coded trie data structure. The HashStore and SortedStore are only accessed to serve lookups.

*If too few levels are used in a multi-level KV store, the following two cases may cause problems.*

The first case is that the size of each level may be large, which makes it impossible to reside in memory even for the very first level, so either we need more memory for good performance or we keep the memory size changed and the performance is compromised. Also, the cost for compaction can be higher if the file size increases.

The other case is that the size of each level may increases sharply with level, for example, level n is 100 times larger than level n-1, much more data have to be loaded into memory and written back to storage when compaction completes.

Q4: Use Figure 3 (Design of LogStore: an in-memory cuckoo hash table (index and filter) to describe how a PUT request and a GET request is served in a LogStore. In particular, explain how the tag is used in a LogStore.

In LogStore, cuckoo hasing is used to index the metadata in memory. The basic idea of cuckoo hashing is to use more than one hash functions instead of only one. This provides two possible locations in the hash table for each key.
1. Let’s say PUT (K1, V1), I assume K1 can be stored at slot 1 and 5, and another key is stored at slot 1 or 3. The steps to serve a PUT request,
   - Write (K1, V1) at the end of the log
   - Look into slot 1(5), if empty, write 5(1) as tag and offset pointing to the flash location of (K1, V1) pair; if they are occupied, one must check whether it is an old item for K1 or other K2;
     - if it is K1, update offset to pointing to the flash location of (K1, V1) pair;
     - If it is K2, evict K2 out to slot h2(K2) (recorded as tag), where h1(K2) as tag and the offset are recorded; then insert h2(K1) as tag and offset pointing to the flash location of (K1, V1) pair; if slot h2(K2) is occupied by another key, the key should be evicted before K2 is inserted, the process continues until no collision or maximum number of displacements reaches.

2. Let’s say GET (K1, V1),
   - tag in bucket 1 (h1(KI)) is 5 or tag stored in bucket 5 (h2(KI)) is 1?
     - If matches, use offset in the slot to retrieve the (Key, Value) pair from flash
     - if Key=K1, return Value
   - Otherwise, not found in LogStore

Q5: Use Figure 4 to explain how a LogStore is converted into a HashStore?
In LogStore, KV items are sorted natively in *insert order* because of log. Once a LogStore fills up, begin to convert it into HashStore. During conversion, the old (immutable) LogStore serves lookups and a new LogStore receives inserts. In HashStore, the KV items are stored in *hash order* based on the hash map data structure. For example, as shown in the figure, the item order is converted into (K2, K4, K1, K3) from (K1, K2, K3, K4). The assumption here is that KV items are of equal size, and then the offset of each item can be calculated as `sizeof(kv item) * bucket_index`. The hash table in memory is removed and a hash filter is used to check whether a key exists or not, it works like a boom filter and can also return false positive. When the conversion process completes, the old LogStore is deleted.

Q6: “Once a LogStore fills up (e.g., the insertion algorithm terminates without finding any vacant slot after a maximum number of is placements in the hash table), SILT freezes the LogStore and converts it into a more memory-efficient data structure.” Compared to LogStore, what’s the advantage of HashStore? Why doesn’t SILT create HashStore at the beginning (without first creating LogStore)?

With HashStore, the in-memory hash table can be removed as KV items are sorted in hash order and the location of entries can be calculated with hash functions. However, if HashStore is directly used to serve insert, a lot of extra flash write will be introduced because of hash collision and the performance will be degraded. It is not a big problem in LogStore, because the collision is resolved in memory.

Q7: “When fixed-length key-value entries are sorted by key on flash, a trie for the shortest unique prefixes of the keys serves as an index for these sorted data.” While a SortedStore is fully sorted, could you comment on the cost of merging a HashStore with a SortedStore? Compare this cost to the major compaction cost for LevelDB?

It will be very expensive to merge a HashStore and a Sorted Store because the SortedStore can be very large and the items in HashStore may scatter at any place and the SortedStore needs to be rewritten.
In levelDB, tables at level L and L+1 are sorted and only limited tables are involved in the compaction (1:10), so the cost can be controlled.

Q8: “Figure 5 shows an example of using a trie to index sorted data.” Please use Figures 5 and 6 to explain how the index of a SortedStore is produced.

![Figure 5: Example of a trie built for indexing sorted data.](image)

In Figure 5, Key prefixes with no shading—shortest unique prefixes which are used for indexing and shaded part—ignored for indexing since suffix part would not change key location. For instance, to lookup a key 10010, follows down to the leaf node that represents 100. As there are three preceding leaf node, index of key is 3.

For example, the key to lookup is 10010, and the trie=3213111;

1. Key[0] =1, goes to right subtrie by skipping the left subtrie. But how?
2. Read trie[0]=3, meaning there are 3 leaf nodes on the left subtrie
3. go to trie[1]=2, meaning there are 2 leaf nodes on the left subtrie, so there is another inner layer
4. go to trie[2]=1, meaning a leaf node is attached to it
5. So the left subtrie has finished traversing
6. Go to trie[3], we arrive the right subtrie

7. Key[1]=0, go to left subtrie(3111), meaning there are 3 leaf nodes on the left subtrie

8. Key[2]=0, go to the left subtrie, as there is only one left leaf node (subtrie[1]=1), if it exists, it must be there.

9. Because it is the first item on the right subtrie, the index=leaf nodes on the left subtrie, which is 3.