SILT: A Memory-Efficient, High-Performance Key-Value Store

NOTE: Your slides/presentation only need to cover background information necessary to answer the given questions (in a clear and well-organized manner). You are allowed to borrow contents from other resources, such as online slides, as long as you acknowledge them. The presentation should be mostly question-focused and proceed mostly in a Q&A format. Please include the questions in your slides. Don’t write detailed answers in the slides and read them to the class. Instead, use bullet points, graphs, or animations to explain your answers to the class.

In your Q&A report, use text to more thoroughly answer the questions. Include a short paragraph at the beginning of the report to summarize the paper.

Overview of paper:
SILT (small index large table) is a memory efficient, high performance key value store systems based on flash storage system. Author had focused on two main bottleneck memory overhead and read amplification. Builds simple store systems which have only logstore, hashstore and sortedstore. Used partial key cuckoo hashing to reduce computation. Hash filters used so low false positive.
SILT combines new algorithmic systems and techniques to balance memory, storage, and computation to build a memory efficient and high performance key value store systems.

(1) “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.

Answer:
As shown in below diagram, Skimpy stash is good in terms of memory overhead but fails in read amplification. The Buffer hash and FAWN-DS better in terms of memory overhead but not good read amplification. But SILT focused both on read amplification and also memory overhead.
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?

Answer:
Multi-Level KV systems impose main 2 challenges:
1. Required effective designs and implementation should be individual stores.
2. Efficient to query at multiple stores when performing lookups.

When it has few levels the search for key element is faster i.e. read is faster than compare to multi-level KV store.

But level-DB these challenges are not focused, and all level are imposed same designs. But SILT focus on these challenges where they implemented only 3 level and all these 3 levels have individual store have unique implementation hence query at multiple stores is faster compared to Level-DB.

Architecture explanation:
SILT architecture uses 3 basic stores such as logstore, hashstore and sorted store.

Logstore: Handles PUT and DELETE individually. Writes are just appended at the end of the log file on flash in the order of insertion time to maintain persistent data on-flash.
Hashstore: Once logstore is full, then this is converted in hashstore. Hashstore uses partial key cuckoo hashing which provide 93% occupancy with low overhead and low computation. Hashstore is immutable and main table different hash table.

Sortedstore: Many Hashtable are merge using merge sort algorithm and stored in sorted form. Sorted store is highly compact index it contains only 0.4 bytes/key

---

(3) "SILT uses a memory-efficient, high-performance hash table based upon cuckoo hashing.". Explain what the cuckoo hashing is and why it is used.

Answer: The cuckoo hashing is hash function which resolves hash collision values using 2 different hash function and provides 93% occupancy with low computation.
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 for cuckoo hashing.

Answer: PUT and GET request are handled with partial key cuckoo hashing. Uses 2 hash function $h_1(x)$ and $h_2(x)$ to map 2 candidates in the hash table. When new key found, if one of the hashtable is empty then insert the new key in that empty slot. If it not find the empty slot in the hash table then it kicks out the current available to another location using different hash function.
If it unable to find the vacant slot then SILT freezes the logstore and initialize new without expensive rehashing. To make this compact this uses only tag as actual key and in sorted form to save memory.

(5) Use Figure 4 to explain how a LogStore is converted into a HashStore?
Answer: when log store is fills up, SILTs freeze the log store and converted it into memory efficient data structure. The hash store uses the hash function and place the tag to the particular slot. It doesn’t store offset hence memory is saved.
“Once a LogStore fills up (e.g., the insertion algorithm terminates without finding any vacant slot after a maximum number of displacements 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)?

Answer:
The main advantage of hash store:
1. Saves memory without saving indexing.
2. Avoid ordering process since hashing is used.

To be consistent data on-flash SILT doesn’t implement the hash store at the beginning. 

Because hash-store stores data on memory, but we should make sure all data should store on disk to prevent loss of data.
“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?

Answer: SILT is worse because single sorted store leads to worse write amplification. while many hash table merges to sorted store, if sorted store is big then write amplification cost is more.