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

SOSP’11

Presented by Fan Ni
March, 2016
✓ **SILT** is *Small Index Large Tables* which is a memory efficient high performance key value store system based on flash storage.

✓ The design and implementation of *three* basic key-value stores:
  - **LogStore (writable):** writes **PUT** and **DELETE** sequentially to flash to achieve high write throughput
    - Cuckoo hash table in memory
  - **HashStore (readonly):** Once a LogStore fills, SILT freezes the LogStore and converts it into a more memory-efficient data structure
    - Hash filter in memory
  - **SortedStore (readonly):** stores (key, value) entries sorted by key on flash, indexed by a new entropy-coded trie data structure
    - Entropy-coded representation
    - Metadata is minimized
In-memory metadata overhead decreases
Tradeoff between in-memory metadata overhead and performance
Q&A

• 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.
Figure 1

*Tradeoff between metadata overhead and flash read (performance)*

- Fawn-DS and BufferHash: consume more memory for better performance
- SkimpyStash: minimize metadata overhead with more flash read

*SILT tries to minimize metadata overhead and RA (Read Amplification)*
• **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.
In SkimpyStash
- Storing the linked lists on flash with a pointer in each hash bucket in DRAM pointing to the head of the list
- When read,
  - go through the list on flash
  - May cover more than one pages
  - Before the target KV pair is found, more than one flash read may be performed, incurring RA
- GC will incur WA when moving items to new place

Also, in levelDB
- If read more than one level to service GET request, RA introduced
- The compaction 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?
Three basic KV stores with different tradeoffs between metadata overhead and performance

- **LogStore**: Serves PUT and DELETE by writing sequentially to flash.
  - Using cuckoo hash maps keys to their location in the flash log

- **HashStore**: Once a LogStore fills up, SILT freezes the LogStore and converts it into a more memory-efficient data structure.

- **SortedStore**: Serves read only, very low memory footprint.
  - Stores KV entries sorted by key on flash, indexed by a new entropy-coded trie data structure
  - Low RA(exactly 1) by directly pointing to the correct location on flash.
  - A configurable number of HashStores merge to a SortedStore
Too few levels in a multi-level KV store, what will happen?

✓ **Size of each level may be large, impossible to reside in memory even for the very first level:**
   - Need more memory for good performance
   - the cost for compaction can be higher if file size increases

✓ **Or the size increase sharply with level, for example, level n is 100 times larger than level n-1, then**
   - More reads are serviced at lower level, incurring RA
   - the cost for compaction can be very high (higher WAF)
   - 101 files are read into memory
**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.

Cuckoo hashing:
- Use more than one hash functions
- Each key can be store in several locations

Try Cuckoo hashing here
• **GET(K1)**
  - *tag* in bucket 1 (h1(K1)) is 5 or *tag* stored in bucket 5 (h2(K1)) is 1?
    - If matches, use *offset* in the bucket to retrieve the (Key, Value) pair from flash
    - if Key=K1, return *Value*
    - Otherwise, not found in LogStore

<table>
<thead>
<tr>
<th>Key</th>
<th>h1(Key)</th>
<th>h2(Key)</th>
</tr>
</thead>
<tbody>
<tr>
<td>K1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>K2</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Tag(h1(key))</th>
<th>Tag(h2(key))</th>
</tr>
</thead>
<tbody>
<tr>
<td>K1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>K2</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

• **PUT(K1, V1)**
  - Write (K1, V1) at the end of the log
  - Look into bucket 1(5), if *empty*, write 5(1) as *tag* and *offset* pointing to the flash location of (K1, V1) pair; if *occupied*, must check 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 bucket 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 bucket 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.
• **Q5: Use Figure 4 to explain how a LogStore is converted into a HashStore?**
In LogStore, KV items are sorted natively in \textit{insert order} because of log.

Once A LogStore fills up, begin to convert it into HashStore.

During conversion, the old (immutable) LogStore serve lookups and a new LogStore receives inserts.

Store the KV items in \textit{hash order} based on the hash map data structure:

- (K1, K2, K3, K4)\rightarrow(K2, K4, K1, K3)
- KV items are of equal size
- Offset: sizeof(kv item)\times bucket_index
- Hash table in memory is removed
- Use a hash filter to check whether a key exists or not
  - Like boom filter
  - Also may return False Positive

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, 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.
Key prefixes with no shading-shortest unique prefixes which are used for indexing

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

Representation for tree $T$ having left and right sub trees $L$ and $T$ is given by:

- $\text{Repr}(T) = |L| \cdot \text{Repr}(L) \cdot \text{Repr}(R)$
- Where $|L|$: no of leaf nodes in left sub tree
For example, the key to lookup is 10010, and the trie=3213111;

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

• Key[1]=0, go to left subtrie(3111), meaning there are 3 leaf nodes on the left subtrie
• 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.
• Because it is the first item on the right subtrie, so the index=leaf nodes on the left subtrie, which is 3