SkimpyStash: RAM Space Skimpy Key-Value Store on Flash

Q&A Report

Summary

SkimpyStash, a RAM space skimpy key-value store on flash-based storage, is designed for high throughput, low latency server applications. The distinguishing feature of SkimpyStash is the design goal of extremely low RAM footprint at about 1 byte per key-value pair. SkimpyStash uses a hash table directory in RAM to index key-value pairs stored in a log-structured manner on flash. However, SkimpyStash moves most of the pointers that locate each key-value pair from RAM to flash itself. That is SkimpyStash stores the linked lists on flash itself with a pointer in each hash table bucket in RAM pointing to the beginning record of the chain on flash. Two further techniques are used to improve performance: (i) two-choice based load balancing to reduce wide variation in bucket sizes and (ii) compaction procedure to pack bucket chain records contiguously onto flash pages so as to reduce flash reads during lookup.

Question 1

Q: “Our base design uses less than 1 byte in RAM per key-value pair and our enhanced design takes slightly more than 1 byte per key-value pair.” In FAWN, even a pointer to a KV pair needs a 4-byte pointer. How can it be possible for SkimpyStash to achieve such a low memory cost for metadata?

A: SkimpyStash only stores a pointer which points to a bucket chain, so all the key-value pairs in the same bucket chain share the same pointers. Thus, on average, each key-value pair only uses part on RAM space used by the pointer. For example, a pointer needs 4 bytes RAM space, and there is a bucket chain has eight key-value pairs, then each key-value pair in the bucket chain only use 0.5(4/8) byte RAM space.

Question 2

Q: “SkimpyStash uses a hash table directory in RAM to index key-value pairs stored in a log-structure on flash.” Why are key-value pairs on the flash organized as a log?

A:
Flash has a major drawback that it does not allow in-place update. When users try to do in-place update, the controller of Flash would do the following steps:

1. Read entire block which usually includes 32-64 pages to memory.
2. Update data in memory.
3. Erase block.
4. Put the data in memory back to the same block.

Therefore, write operation is prolonged as compared to read operations. To solve this issue, SkimpyStash uses log-structured file system. Log-structured file system always appends new data to the tail of log, making write operation fast.

**Question 3**

Q: **“The average bucket size is the critical design parameter that serves as a powerful knob for making a continuum of tradeoffs between low RAM usage and low lookup latencies.”** Please explain this statement.

A: The reason that SkimpyStash has such low RAM usage is that all key-value pairs in the same bucket chain share the same pointer. Hence, the longer the bucket chain is, the less RAM space a key-value pair needs. On the other hand, as the number of key-value pairs increases in a single bucket chain, SkimpyStash needs more time to go through it to search specific key-value pair. That is, long bucket chain would cause high lookup latencies. Thus, this is the tradeoff users need to make.

**Question 4**

Q: **“The chain of records on flash pointed to by each slot comprises the bucket of records corresponding to this slot in the HT directory. This is illustrated in Figure 3.”** Please use the figure to describe SkimpyStash’s data structure. Also explain how lookup, insert, and delete operations are executed.

A: This figure is the figure 3 and I already explain the data structure in Summary section.
Here are the steps of lookup operation when users want to lookup key1:

1. SkimpyStash hash key1 to get the hashed key.
2. Find the bucket by hashed key, then find the bucket chain.
3. Go through the chain and find the the target key-value pair if there is any.
4. Return the just found key-value pair to user.

Here are the steps of insert operation when user want to insert this pair(key1, value1):

1. SkimpyStash hash key1 to get the hashed key.
2. Find the bucket by hashed key, then find the bucket chain.
3. Insert the new pair to the head of the chain.
4. Update the pointer in the bucket to point to the new pair.

Because deleting is kind of in-place update, the cost of deleting is too expensive if SkimpyStash deletes the key-value pair on the flash directly. So SkimpyStash uses insert operation to implement delete operation and uses Garbage Collection to recycle RAM space. For example, when user want to delete this pair (key1, value1). SkimpyStash would insert this pair (key1, Null).

Question 5

Q:
“Because we store the chain of key-value pairs in each bucket on flash, we incur multiple flash reads upon lookup of a key in the store.” Please explain how this issue can be alleviated. [Hint: please refer to Section “Compaction to Reduce Flash Reads during”]

A:
For answering this question, we need to know why SkimpyStash needs multireads for a lookup operation. SkimpyStash stores key-value pairs in bucket chain, but SkimpyStash put each the key-value pairs in the same bucket chain to different pages or blocks on flash. Namely, There may be many flash read operations just for a single lookup operation because the smallest unit of flash read operation is page.

To solve this issue, SkimpyStash uses compaction. By doing the lookup operation to the target bucket chain, SkimpyStash reinserts all key-value pairs in the bucket chain to the same page or consecutive pages. Next, SkimpyStash would update the pointer in the corresponding bucket of the hash table. Hence, a lookup may need less flash read operations by using compaction.
Question 6

Q:
“...two-choice based load balancing strategy is used to reduce variations in the number of keys assigned to each bucket”. Explain how this is achieved.

A:
SkimpyStash uses “Power of Two Choice” to control load balancing of the hash table on RAM. Each key would be hashed to two candidate HT directory buckets, using two hash functions h1 and h2, and actually inserted into the one that has currently fewer elements.

Question 7

Q:
"... when the last record in a bucket chain is encountered in the log during garbage collection, all valid records in that chain are compacted and relocated to the tail of the log.". Please explain how garbage is collected.

A:
When garbage collection started, it would scan from the head of the log. There are two situations after garbage collection encountered a key-value pair.

1. Garbage collection encountered a valid key-value pair. It re-installs this key-value pair to the tail of log. Next, garbage collection reclaims the RAM space occupied by the key-value pair.
2. Garbage collection encountered an invalid key-value pair. In this case, it would directly reclaim this space occupied by the invalid key-value pair without doing anything else.

In addition, there two types of invalid key-value pair:
- Invalid key-value pair caused by the delete operation.
- Invalid key-value pair caused by compaction.