Quiz on SILT

Test your understanding of concepts related to the design of the SILT system via a quiz.

Question 5

How does changing the number of accumulated intermediary stores affect write amplification and per-key memory consumption?

Hide Answer

Having a higher number of intermediary stores reduces write amplification. This is because having more intermediary stores allows us to merge less frequently. For all entries, a third write only happens when the merge to the memory-efficient store takes place—this only happens for requests other than `DELETE` entries.

Having a higher number of intermediary stores increases per-key memory consumption since the intermediary store has a higher per-key memory consumption. Having more intermediary stores increases the proportion of entries stored by intermediary stores and raises the average per-key memory consumption.

Question 1

Why do we keep a `DELETE` request in the intermediate log?

Hide Answer

We keep the `DELETE` request in the intermediate log because the last-level store can't purge deleted keys from the store (and a subsequent `GET` request can provide a stale value).

Question 2

There can be so many stores involved in serving a `GET` request. Will this add latency in getting key values?

Hide Answer

The first two stores—write-friendly store and intermediary store(s)—have in-memory filters that only allow storage reads for keys that exist in them.

- If the keys do not exist in the store, the filter will suggest this in constant time so that the request may continue in the next store.
- If the key does exist in this store, then we have their offset and will skip a fixed length of key-value entries to get our entry—an action that completes in constant time.

If we look inside the memory-efficient store, it returns a result in logarithmic time in the average case.

Asymptotically, such `GET` requests will consume logarithmic time because there will be a constant number of stores involved, and each store except the memory-efficient store takes constant time to serve `GET` requests. Latency is minimum because of the in-memory indexes in our design.

Question 3

Our write-friendly store processes `PUT` and `DELETE` requests quickly since it writes sequentially in storage. It can even serve to `GET` requests in constant time. It is a high-performance key-value store that uses both memory and storage. It even uses memory efficiently by ensuring high occupancy of its in-memory hash table and using partial keys.

Why is it difficult to scale a design that uses our write-friendly store in a single-store approach?

Hide Answer

While the write-friendly store's in-memory hash table is optimized to use as less memory as possible, it still uses a high amount of memory per key. The hash table contains the partial key hash and offset in the storage log for a key. Scaling for billions of keys would require a large hash table. The relatively high per-key memory consumption would cause high overall memory use.

Also, scaling would mean a longer storage log containing key-value entries. We would require more bits to cover higher values for the offset in the offset column. This increases per-key memory consumption while scaling.

Furthermore, we must dedicate memory to the hash table before populating it. If we chose a larger size for the hash table (for scaling), it would take longer to populate, and average per-key memory consumption would increase. This is because, on average, buckets are now more likely to be empty than occupied when compared to a smaller hash table.

Our design approach exploits the fact that we do not need to store our key-value entries as the write-friendly store does for a long time. We only do this for the newest entries since it allows for fast sequential writing. Later, we move entries to stores that use memory sparingly.

Another thing to consider is that the write-friendly store does not remove old entries for keys in storage. When a new request comes in for an existing key in the store, it only updates its offset in memory. This will need to be considered when using a single-store design with a write-friendly store.

Question 4

Express the worst case for a `GET` request that results in the highest number of storage reads for a given number of intermediary stores, say I.

Hide Answer

The worse case is when, for a `GET` request, a key results in a false positive in the write-friendly store, false positives in all the intermediary stores, and a successful match or false positive in the memory-efficient store.

Here is how this happens:

1. The `GET` request first searches in the write-friendly store. A false positive occurs when the computed hash of the partial key matches the tag in the bucket, and the returned key from storage does not match the key from the `GET` request. We have incurred one storage seek.
2. Now the `GET` request searches in the intermediary stores from the latest to the oldest. False positive occurs in the intermediary store as in the write-friendly store. False positives in all the intermediary stores have made us incur I storage seeks.
3. Now in the memory-efficient store, our compact representation of a prefix tree will always provide us with a leaf node with an index to look for our key. We will incur a storage seek regardless of whether it was a false positive.

So in total, total storage seeks are $2 + I$.

Question 5

How does changing the number of accumulated intermediary stores affect write amplification and per-key memory consumption?

Hide Answer

Having a higher number of intermediary stores reduces write amplification. This is because having more intermediary stores allows us to merge less frequently. For all entries, a third write only happens when the merge to the memory-efficient store takes place—this only happens for requests other than `DELETE` entries.

Having a higher number of intermediary stores increases per-key memory consumption since the intermediary store has a higher per-key memory consumption. Having more intermediary stores increases the proportion of entries stored by intermediary stores and raises the average per-key memory consumption.