Evaluating and Extending the Design of SILT

Evaluate how our design meets our goals and learn about extensions that can be made to our design.

We'll cover the following

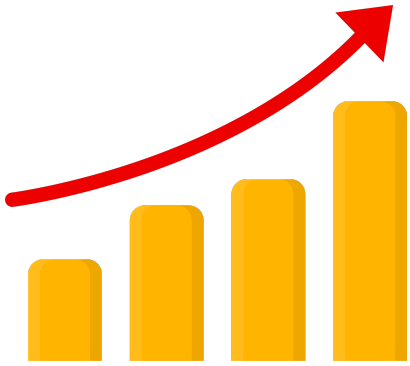
Evaluation

We designed our key-value store to meet many goals. Let's assess how well our store meets those goals. We will offer possible extensions to our design as well.

Reduced per-key memory consumption

Our three stores in decreasing per-key memory consumption are the write-friendly store, the intermediary store, and the memory-efficient store. In our design, the proportion of key-value entries stored increases in this order. We have ensured that our most memory-efficient store has most of our entries.

We cannot avoid storing keys in the write-friendly and intermediary stores since it is required to make our write requests efficient.



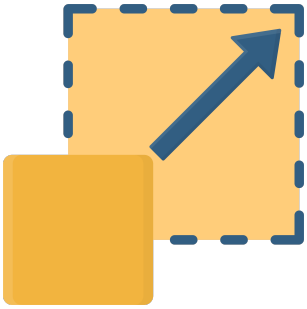
Memory efficiency allows for scalability

Low read amplification

Just because our entries are stored in multiple stores it does not mean that our design requires storage lookups for each store.

For the write-friendly and intermediary stores, their in-memory hash table serves as a filter. If the key does not exist in these stores, the request is forwarded to the next store without looking it up in storage. So all read requests that make it to the memory-efficient store have been filtered by the hash filters from the write-friendly store and the intermediary stores.

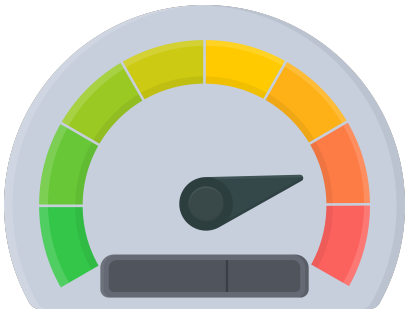
This design offers a near `1` read amplification (the value of `1` for the read amplification is the best possible value). The only cases where read amplification exceeds `1` are false positives, when the partial key hash matches the computed hash for `GET` requests in the write-friendly and intermediary stores, and the extracted key from storage does not match the lookup key. This can be kept low by selecting good criteria for extracting a partial key out of the complete key for partial-key cuckoo hashing.



We cannot afford multiple storage reads for one lookup if we want to scale to billions of keys

Fast lookup

The first two stores provide constant time lookup, while the memory-efficient store provides a lookup in $O(log\ n)$ time. We can make improvements to the memory-efficient store to provide lookups in constant time.



Scaling for billions of keys requires fast lookups

Write optimization

Our design appends new entries to the write-friendly store's in-storage log, a process that requires constant time since the write-friendly log writes sequentially to storage. Conversion to an intermediary can be kept optimally fast by configuring the size of the write-friendly store. It will be good to keep the store size the same as the flash page size while writing the flash-based storage.

Conversion from intermediary stores to one memory-efficient store is also fast since all stores in the merge are sorted.

To make a highly scalable design highly available, we need to have predictable write performance and controllable amplification

Controllable write-amplification

Our design writes one key multiple times, resulting in higher write amplification. However, we can control our write amplification by increasing the number of our design's instances on a node. We can also reduce amplification by reducing the number of times a bulk merge of intermediary stores and the memory-efficient store takes place, for example, by increasing the number of intermediary stores before doing a bulk merge.

While we want to keep write amplification as low as possible, that might compromise our goal of reduced per-key memory consumption. Hence, we aim for controllable write amplification by configuring our design so that it may exhibit a write amplification acceptable to us.

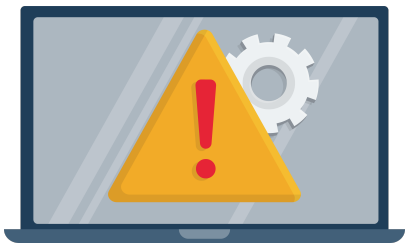
Here is how we can understand write amplification in our design. Every key-value entry is written in storage once when entered into the write-friendly store. It is written again when the write-friendly store is converted to an intermediary store. Now, the only time we will write a key to storage again is when there is a bulk merge of intermediary stores with the memory-efficient store. The frequency of this depends on the number of entries we decide to accumulate (in intermediary stores) before merging. Keeping a higher number increases the average memory used per key but reduces write amplification, and keeping it lower decreases the average memory used per key and increases write amplification. This is because

the per-key memory consumption of the intermediary store is higher than the memory-efficient store.

> **Note:** It's important to note why we have kept write amplification controllable. This is because it impacts average per-key memory consumption. Another important consideration is that this write amplification does not affect our write speed. Write amplification is a little high because we need to do some housekeeping—that happens offline—to keep memory consumption low.

Fault tolerance

Currently, our design provides crash tolerance at some level: our write-friendly store has a log file in storage. If the memory fails, we can go through the log file to recreate the in-memory hash table. We can also create in-memory indexes for all intermediary stores and our memory-efficient store.



Fault tolerance is important for providing a respectable service

Some empirical statistics

To get an idea of our design in practical terms, we will share the results of a workload test conducted for a similar design. Below is a measure of the construction performance of our design.

Construction Performance

Type	Speed (K keys/s)

Write-friendly store	204.6
Intermediary store	67.96
Memory-efficient store	26.76

We can see that the write-friendly store is the fastest to construct. This is because its construction requires no reading and rearranging. The construction speed of the intermediary store is a third of that—since it requires reading from and reordering in storage. And the memory-efficient store is the slowest to construct because it involves sorting entries by keys of the accumulated intermediary stores and then merging it with the old memory-efficient store.

Now let's look at the performance for `GET` requests.

GET Request Performance

Type	Memory-efficient	Intermediary	Write-friendly
GET (hit)	46.57	44.93	46.79
GET (miss)	46.61	7264	7086

> All numbers in the above table have the unit K ops/s.

The table above shows the minimum performance for `GET` requests when the key in the `GET` request exists in the store (first row; `GET(hit)`) and where it does not (second row; `GET(miss)`). Performance is fairly consistent in all stores for cases where the key exists. Notice how the memory-efficient store has the same performance for both cases. This is because its in-memory index is not a filter, and every `GET` request will result in a storage read—the worst case we discussed earlier. However, this is not the case for the other two stores because of their in-memory

filters, which help us avoid storage seeks for non-existent keys—these filters are also responsible for keeping read amplification low.

Extensions

Next, we'll look at use of extension to improve our design.

Variable-length key-values

Currently, our design works effectively with fixed-length entries. The offset in our write-friendly store is essentially the number of times we need to jump a fixed length to reach our desired entry. For the intermediary store, the position in the in-memory hash table provides us with this number.

However, some cases may require variable length-key functionality, such as storing comments or messages on microblogging websites. One solution is to have multiple instances of our design dedicated to different key sizes. This would require a management program to route requests to the relevant store.

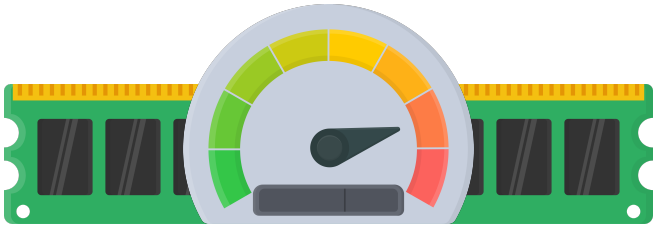
In our opinion, it is hard to change the node-level design to accommodate variable key lengths. We could have different nodes for different key lengths, or something in between like dedicate nodes for key length ranges.

High memory pressure tolerance

High memory pressure can occur if our hardware is short on memory. Under such conditions, our design might have degraded performance with higher read amplification.

To avoid crashes, we will implement the following contingencies in our design:

1. We may drop in-memory indexes and filters for our intermediary and memory-efficient stores. Our design stores the data for both these stores sorted in storage. We can trade one extra storage seek and two extra storage seeks for our memory-efficient and intermediary stores, respectively, to avoid a crash.
2. We may use binary search on our memory-efficient store and drop the trie.



Contingencies for high memory pressure are important in systems where memory is intensely used

Conclusion

In this chapter, we designed a key-value store that (a) we can scale for billions of keys because it presents low read amplification, and low memory consumption for keys (b) is highly available because of fast writes due to sequential writing.

System design wisdom in SILT design

- When we have multiple system aspects to optimize, we might need to build multiple mini-systems (each optimized for a specific aspect) and then combine them to meet our needs collectively. For our key-value store, we want low read and write amplification, fast writes, and reads, and efficient memory use to pack more keys in a node. We used three kinds of mini-systems and provided a higher layer to serve client requests (`GET`, `PUT`, `DELETE`) per our needs.
- SILT's design shows us that some design problems need intense attention to the details. Typically when a system is built initially it's simple, but as needs evolve, so does the design. Though after a certain point, further optimizations come at the cost of added complexity.