A Write-friendly Store for SILT: Part III

Learn how to design a key-value store for fast writing.

We'll cover the following

GET requests

Since we are using a partial key, it is important to note how this will affect `GET` requests. We will only proceed with a lookup for a `GET` request if the tag matches the key in the request. By lookup, we mean lookup in storage. So, when the write-friendly store receives a `GET` request for key `Kg`:

- It computes the candidate buckets `h1(Kg)` and `h2(Kg)` and looks inside these buckets.
  - Inside `h1(Kg)`, it looks for the tag `h2(Kg)`. If the computed tag `h2(Kg)` matches the tag inside bucket `h1(Kg)`, then it looks up the entry in the storage log stored on the offset marked with `h2(Kg)`.
  - Inside `h2(Kg)`, it looks for the tag `h1(Kg)`. If the computed tag `h1(Kg)` matches the tag inside bucket `h2(Kg)`, then it looks up the entry in the storage log stored on the offset marked with `h1(Kg)`.
- Upon a successful match in either of the two cases above, it looks up the key-value pair (in the storage log) at the offset marked with the computed tag: `(Ks,`

`Vs)`. It then matches the key `Ks` from the key-value pair returned from storage with `Kg` to confirm that we have looked up the correct key.

- If the keys match (`Kg` equals `Ks`), we check the value `Vs` in the key-value pair returned from storage.
  - If `Vs` is not the special `DELETE` indicator (`Vs` does not equal to `DELETE`), the write-friendly store signals that the **`GET` request should terminate** and return that **`Vs` is the value stored against `Kg`**.
  - If `Vs` is the special `DELETE` indicator (`Vs` equals `DELETE`), the write-friendly store signals that the **`GET` request should terminate** and return that **`Kg` is not stored in the key-value store (the entire key-value store and not just the write-friendly store)**.
- If the keys do not match (`Kg` is not equal to `Ks`), the write-friendly store signals that **`Kg` is not present inside the write-friendly store**, and the **`GET` request should continue in the intermediary stores**.

- Upon an unsuccessful match in both candidate buckets, the write-friendly store signals that **`Kg` is not present inside the write-friendly store**, and the **`GET` request should continue in the intermediary stores**. *This is the case where our hash table acts as an in-memory filter. We have returned that `Kg` is not stored inside the write-friendly store without checking the storage log.*

Improving design for memory efficiency

We can further improve our design using the techniques mentioned below.

Memory bound

Although we have tried to keep per-key memory consumption low, this store will still take up more per-key memory than our other stores. Therefore, we will keep a memory cap beyond which this store will not grow. Once the store size becomes greater than or equal to this memory bound, we will initialize a new store and send the old one for conversion to an intermediary store. From this point, this new store will serve incoming requests.

One store

Since this store is relatively expensive in terms of per-key memory consumption, we will only have one instance of this store in our system at a time.

One consideration is that having more write-friendly stores could parallelize new requests. For example, consider having two write-friendly stores serving. If one of the stores is busy inserting a request, the other can serve requests. On average, requests are served faster than in a single write-friendly store.

We must consider that the hash table's memory is allocated as soon as the write-friendly store is initialized. The store's per-key memory consumption reduces as more and more keys are entered into the hash table.

With the two write-friendly stores, we are using twice the memory, and to make using the two stores worthwhile, we will have to ensure high occupancy of two stores rather than one store. These two stores will take longer to reach our desired occupancy. On average, our system will have more per-key memory consumption.

If we were going to give up per-key memory consumption in the first place, we could have kept a lower number of displacements before we initialized a new write-friendly store. This way average occupancy of the in-memory hash table would have been lower (higher per-key memory consumption), but the design would insert requests faster (on average).

While an argument for parallelization by having more than one store can be made, for now, it is important to understand that the trade-offs involved with having more than one write-friendly store are the same as with configuring a single write-friendly store.

What’s next?

When the condition to initialize a new write-friendly store is met, what happens to the old one? It is converted to another store that is easy to convert to our memory-efficient store. We’ll discuss this further in the next lesson.