Request Flows in SILT

Learn how PUT, DELETE, and GET requests are processed in the SILT key-value store.

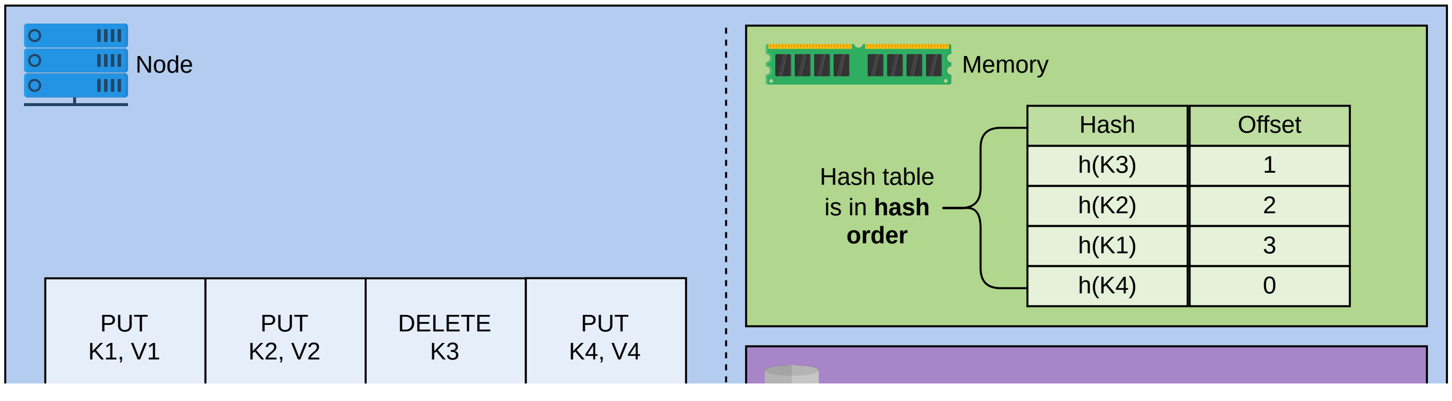
In this lesson, we will look at how our consolidated store serves the requests. Every request will have a key associated with it. For example, `PUT(key, value)`, `DELETE(key)`, and `GET(key)`.

`PUT` and `DELETE` requests

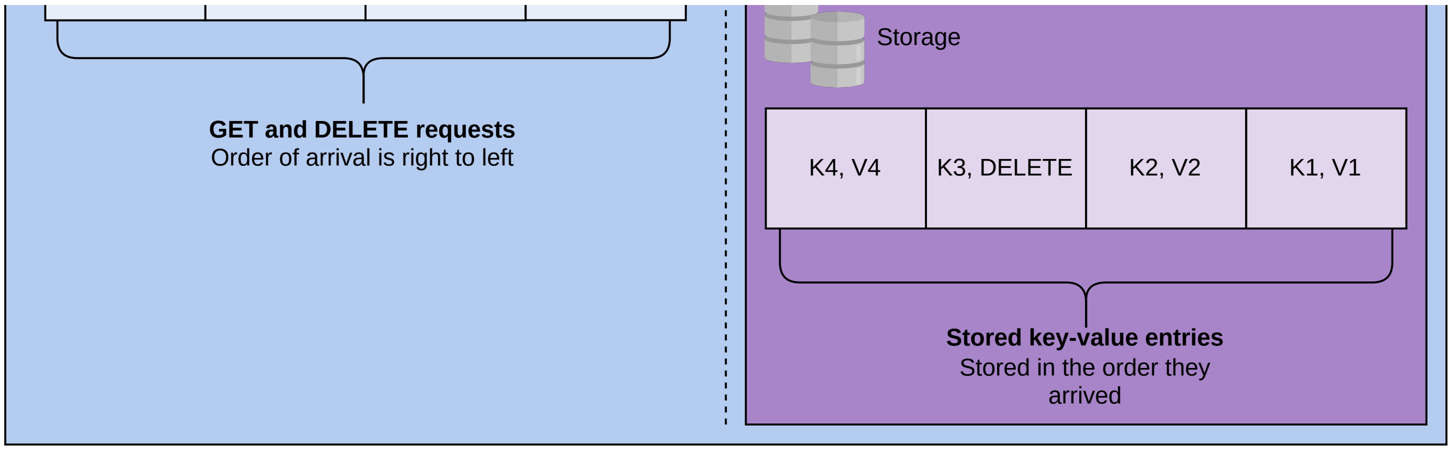
Our write-friendly store receives all new `PUT` and `DELETE` requests. Every request triggers two tasks—one in memory and the other in storage.

In storage, the write-friendly store has sequentially maintained a log to which it appends our request. The write-friendly store saves the offset—its position in the log—for all incoming requests.

In memory, the write-friendly store starts looking for a candidate bucket using partial-key cuckoo hashing on the key for the request, and as soon as it finds a candidate bucket, it stores the offset in that bucket.



Hash	Offset
h(K3)	1
h(K2)	2
h(K1)	3
h(K4)	0

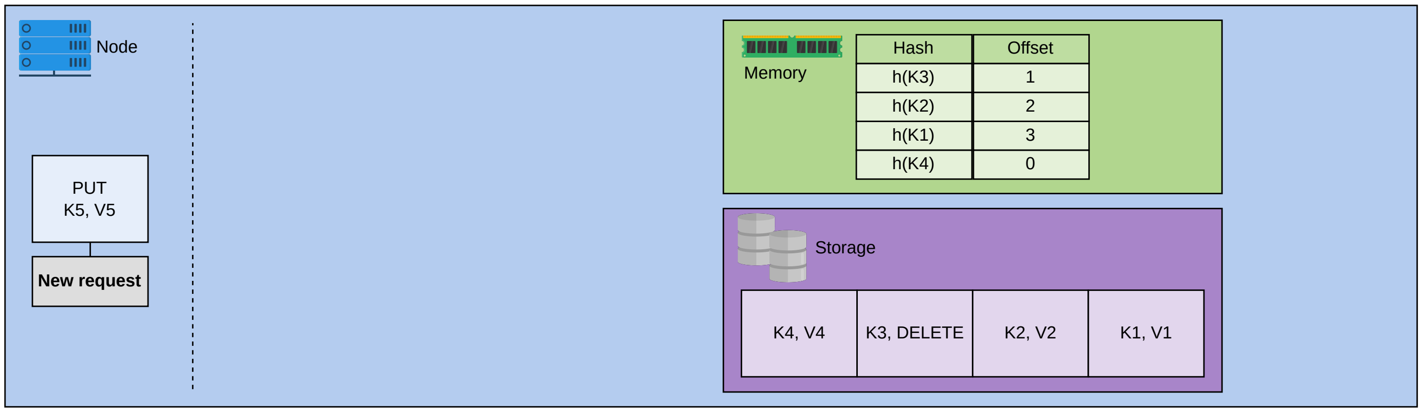


The write-friendly store initially receives PUT and DELETE requests. Note that both PUT and DELETE requests are saved in the same format. The difference is that DELETE request entries have a special DELETE indicator instead of a value.

Both `PUT` and `DELETE` requests remain stored in this state until a new request:

1. Exhausts the number of allowed displacements for partial-key cuckoo hashing to find a candidate bucket.
2. Cannot be placed in the current instance of the write-friendly store as it will result in the write-friendly store exceeding its memory bound.

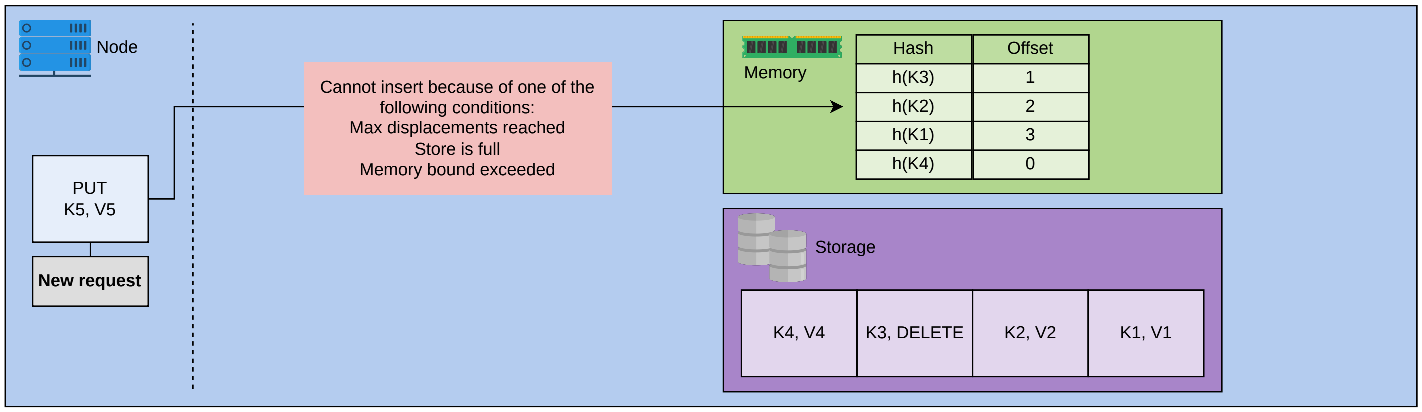
When any one of the conditions above is satisfied, it initializes a new write-friendly store that caters to new `PUT` and `DELETE` requests.



Hash	Offset
h(K3)	1
h(K2)	2
h(K1)	3
h(K4)	0

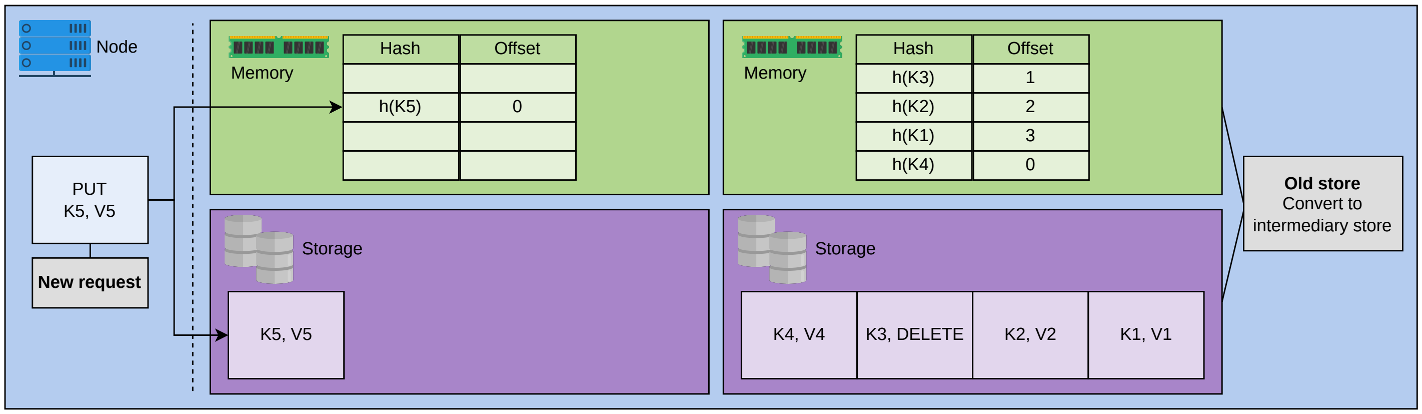
We get a new PUT request for K5

1 of 3



We cannot insert K5 into the write-friendly store because of one of the two conditions (max displacements or store full)

2 of 3

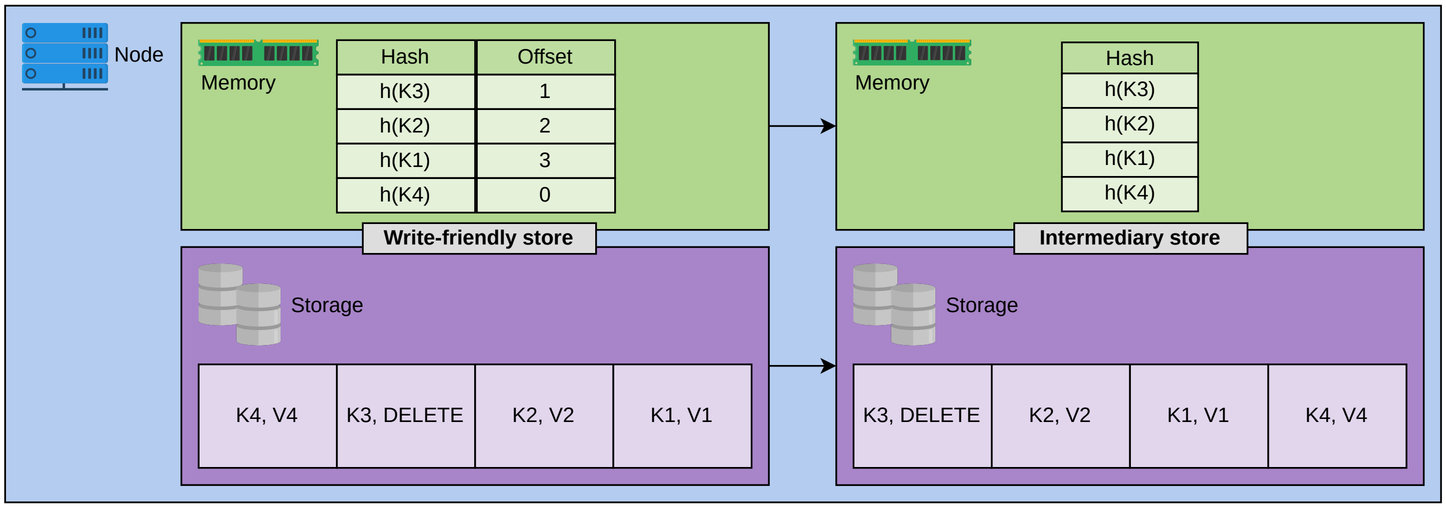


We initialize a new write-friendly store and send the old one for conversion to an intermediary store

3 of 3

Our design then converts the old write-friendly store into an intermediary store. The conversion happens in memory and storage (but does not add latency to the client's new requests as they go to the new store while the older one is being transformed into an intermediate one in parallel).

The conversion in storage involves reordering the log that our write-friendly store was maintaining. The in-memory hash table of the write-friendly store contains values ordered by hash values. Our design traverses the in-memory hash table: for each entry, it finds the corresponding key-value pair in the storage log and places it in the intermediary store's storage. By the end of the conversion process, values have the same order as the write-friendly log's in-memory hash table.



This diagram shows the changes that occur during the conversion of a write-friendly store to a memory-efficient store

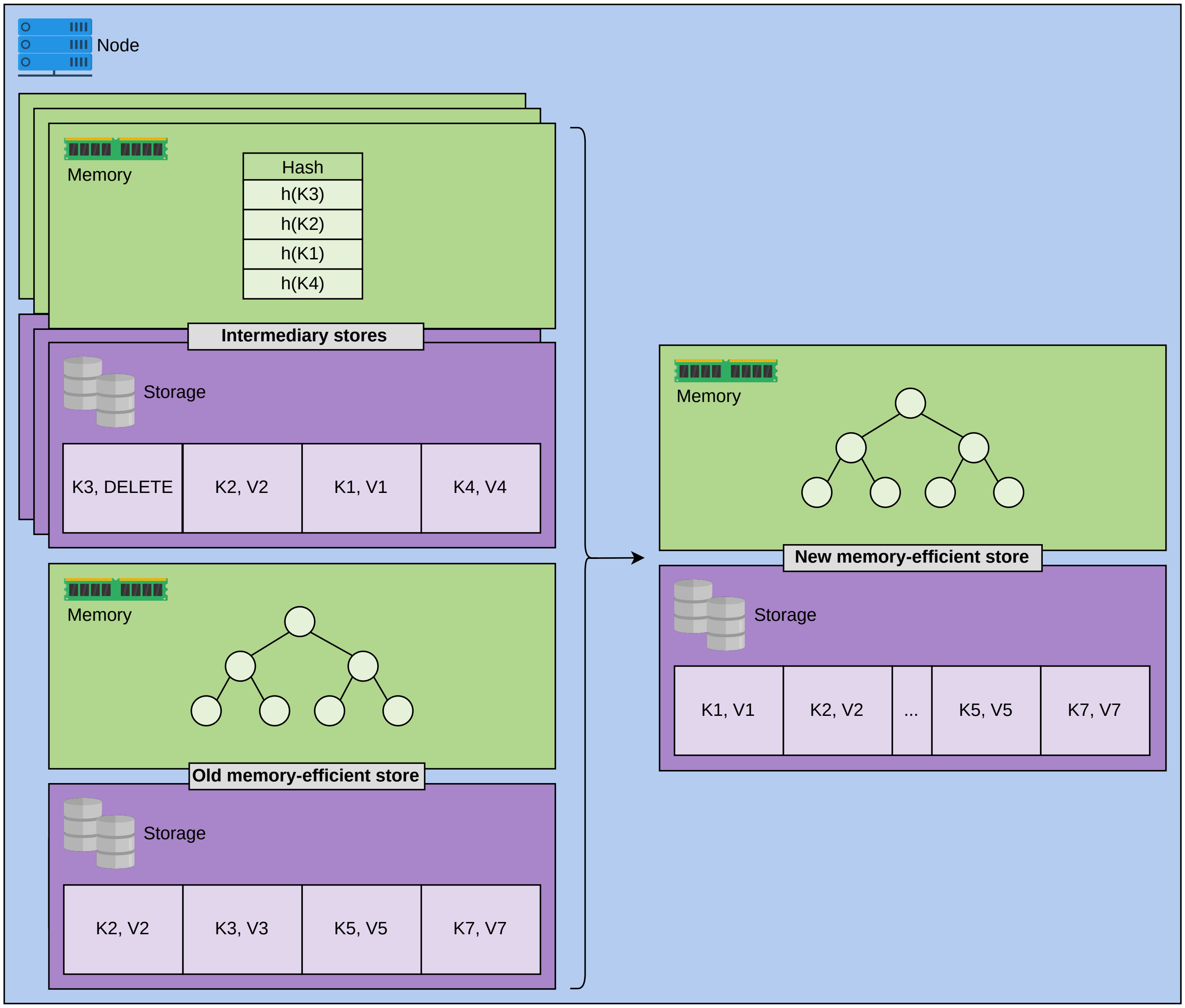
In memory, the conversion is simple. Our design only drops the offset of the in-memory hash table of the write-friendly store, as it is no longer required.

Now, our request is in an intermediary store instance. Our design accumulates a configurable number of intermediary stores and then merges all intermediary stores with the existing memory-efficient instance. The merging process creates a new memory-efficient store with values from the previous memory-efficient store and all existing intermediary store instances.

From this point onwards, `PUT` and `DELETE` requests behave differently. During the merging process, our design processes `DELETE` requests by not adding the pertinent key-value entry in the new memory-efficient store.

> **Note:** A `DELETE` request is not completed as soon as the write-friendly store receives it. It is completed during the merging of the memory-efficient store and intermediary stores, and the memory from the key in the `DELETE` request is released.

Our design also handles `PUT` requests during the merging process. If the key for a `PUT` request already exists in the old memory-efficient store, the new memory-efficient store will have the value from the `PUT` request (in the intermediary store). If there is no key for a `PUT` request, our design will add it to the new memory-efficient store.



Intermediary and old memory-efficient stores are merged to form a new memory-efficient store

`GET` requests

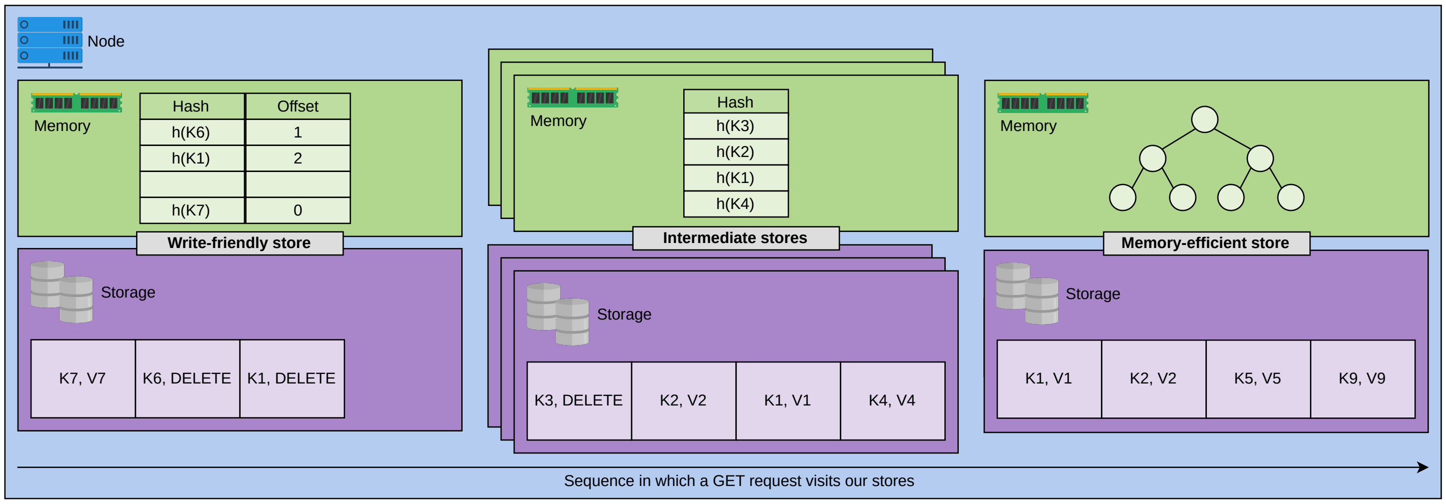
While most of the entries are stored in the memory-efficient store, we need to query stores in an order that ensures our design returns the latest value. We will have to query stores in the same order as the flow of a `PUT` or a `DELETE` request.

For every `GET` request, our design first looks it up in the write-friendly store. Our write-friendly store has a hash filter. If the index for the key from the `GET` request is in the write-friendly store's in-memory hash table, we will look at its offset in memory and return the value corresponding to that position from the in-storage log. If the index does not exist, then it does not exist in the write-friendly store.

Then, our design checks for the same key in all the intermediary stores. These intermediary stores have the same in-memory hash table inherited from their write-friendly store instances. A similar checking process tells us if the key is in an intermediary store. The only difference is that if the key does exist in the intermediary store, we do not need an offset, except that the position of the index on the hash table gives us the in-storage location.

> **Note:** If we encounter a `DELETE` request against a key in the write-friendly store or any of the intermediary stores, we return `NULL`, indicating that the key does not exist in the key-value store.

Lastly, if a key is not in all intermediary stores, our design looks it up in the sorted, memory-efficient store. This process is quick since data in the memory-efficient store is sorted.



A GET request might go from write-friendly store to intermediate stores to the memory-efficient store

Moving forward

We have explained our design in detail. Let's evaluate our design goals in the next lesson.

Back

A Memory-efficient St...

Next

Evaluating and Extend...

Mark as Completed