Memc3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing

July 10, 2025

1 Background

1.1 Memcached Overview

Interface:

- `SET/ADD/REPLACE(key,value)`
- `GET(key)`
- `DELETE(key)`

Hash Table

Memory Allocation: Memcached uses **slab-based memory allocation**. Memory is divided into 1MB pages, and each page is further sub-divided into fixed-length **chunks**. Key-value objects are stored in an appropriately-size chunk. The size of a chunk, and thus the number of chunks per page, depends on the particular slab class. For example, by default the chunk size of slab class 1 is 72 bytes and each page of this class has 14563 chunks; while the chunk size of slab class 43 is 1 MB and thus there is only 1 chunk spanning the whole page.

To insert a new key, Memcached looks up the slab class whose chunk size best fits this key-value object. If a vacant chunk is available, it is assigned to this item; if the search fails, Memcached will execute cache eviction

Cache policy: Each slab class maintains its own objects in an LRU queue.

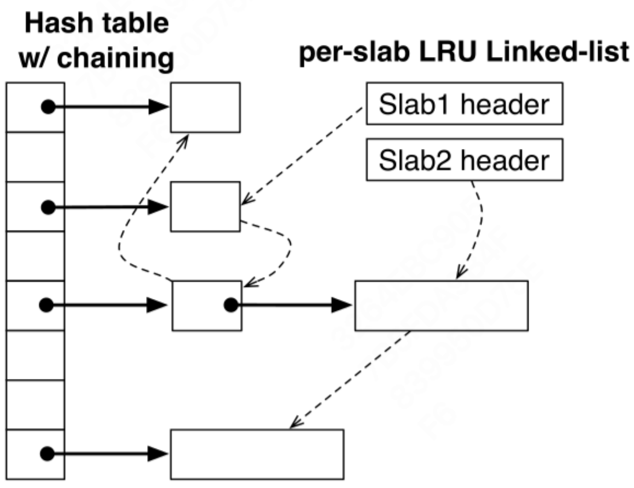

Figure 1: Memcached data structures

Threading

1.2 Real-world Workloads: Small and Read-only Requests Dominate

2 Optimistic Concurrent Cuckoo Hashing

Basic Cuckoo Hashing: The basic idea of cuckoo hashing is to use two hash functions instead of one, thus providing each key two possible locations where it can reside. Cuckoo hashing can dynamically relocate existing keys and refine the table to make room for new keys during insertion.

Our hash table, as shown in Figure 2, consists of an array of **buckets**, each having 4 **slots**. Each slot contains a **pointer** to the key-value object and a short summary of the key called a **tag**. To support keys of variable length, the full keys and values are not stored in the hash table, but stored with the associated metadata outside the table and referenced by the pointer. A null pointer indicates this slot is not used.

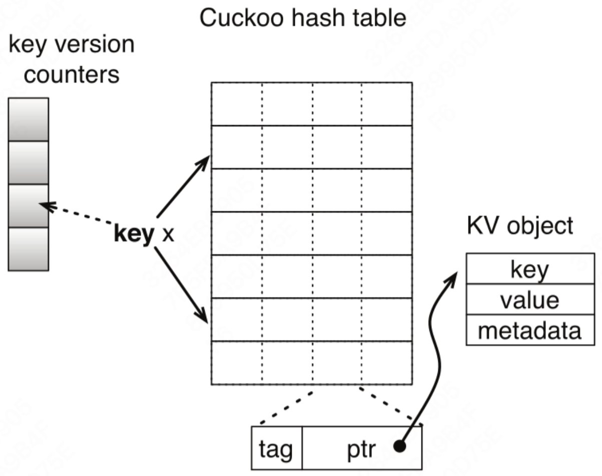

Figure 2: Hash table overview: The hash table is 4-way set-associative. Each key is mapped to 2 buckets by hash functions and associated with 1 version counter; Each slot stores a tag of the key and a pointer to the key-value item. Values in gray are used for optimistic locking and must be accessed atomically.

Each key is mapped to two random buckets, so `Lookup` checks all 8 candidate keys from every slot. To insert a new key x into the table, if either of the two buckets has an empty slot, it is then inserted in that bucket; if neither bucket has space, `Insert` selects a random key y from one candidate bucket and relocates y to its own alternate location. Displacing y may also require kicking out another existing key z, so this procedure may repeat until a vacant slot is found, or until a maximum number of displacements is reached (e.g., 500 times in our implementation). If no vacant slot found, the hash table is considered too full to insert and an expansion process is scheduled. Though it may execute a sequence of displacements, the amortized insertion time of cuckoo hashing is O(1)

2.1 Tag-based Lookup/Insert

We propose a cache-aware technique to perform cuckoo hashing with minimum memory references by using **tags** —a short hash of the keys (one-byte in our implementation).

Cache-friendly Lookup: Checking two buckets on each `Lookup` makes up to 8 (parallel) pointer dereferences. In addition, displacing each key on

`Insert` also requires a pointer dereference to calculate the alternate location to swap, and each `Insert` may perform several displacement operations

Our hash table eliminates the need for pointer dereferences in the common case. We compute a 1-Byte tag as the summary of each inserted key, and store the tag in the same bucket as its pointer. `Lookup` first compares the tag, then retrieves the full key only if the tag matches.

Because each bucket fits in a CPU cacheline, on average each `Lookup` makes only 2 parallel cacheline-sized reads for checking the two buckets.

Cache-friendly Insert: We also use the tags to avoid retrieving full keys on `Insert`. To this end, our hashing scheme computes the two candidate buckets b_1 and b_2 for key x by

$$\begin{aligned} b_1 &= HASH(x) \\ b_2 &= b_1 \oplus HASH(tag) \end{aligned}$$

Now b_1 can be computed by the same formula from b_2 and tag. This property ensures that to displace a key originally in bucket b - no matter if b is b_1 or b_2 - it is possible to calculate its alternative bucket b' from bucket index b and the tag stored in bucket b by

$$b' = b \oplus HASH(tag)$$

2.2 Concurrent Cuckoo Hashing

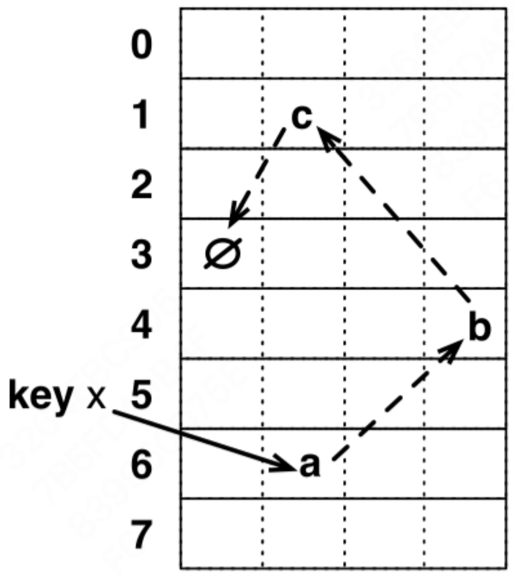

Figure 2: Cuckoo path. ∅ represents an empty slot

Define a **cuckoo path** as the sequence of displaced keys in an `Insert` operation. In Figure 2, $a \Rightarrow b \Rightarrow c$ is one cuckoo path to make one bucket available to insert key x.

Two obstacles:

1. **Deadlock risk (writer/writer)**:
2. **False misses (reader/writer)**

To avoid writer/writer deadlocks, it allows only one writer at a time - a tradeoff we accept as our target workloads are read-heavy. To eliminate false misses, our design changes the order of the basic cuckoo hashing insertion by:

1. *separating discovering a valid cuckoo path from the execution of this path*. We first search for a cuckoo path, but do not move keys during this search phase
2. *moving keys backwards along the cuckoo path*. After a valid cuckoo path is known, we first move the last key on the cuckoo path to the free

slot, and then move the second to last key to the empty slot left by the previous one, and so on. As a result, each swap affects only one key at a time, which can always be successfully moved to its new location without any kickout. {So insert does not affect get}

2.2.1 Optimization: Optimistic Locks for Lookup

Instead of locking on buckets, it assigns a version counter for each key, updates its version when displacing this key on `Insert`, and looks for a version change during `Lookup` to detect any concurrent displacement.

Lock Striping: The simplest way to maintain each key's version is to store it inside each key-value object. This approach, however, adds one counter for each key and there could be hundred of millions of keys. More importantly, this approach leads to a race condition: to check or update the version of a given key, we must first lookup in the hash table to find the key-value object (stored external to the hash table), and this initial lookup is not protected by any lock and thus not thread-safe.

Instead, we create an array of counters (Figure 2). To keep this array small, each counter is shared among multiple keys by hashing (e.g., the i-th counter is shared by all keys whose hash value is i). Our implementation keeps 8192 counters in total (or 32 KB). This permits the counters to fit in cache, but allows substantial concurrent access. It also keeps the chance of a "false retry" (re-reading a key due to modification of an unrelated key) to roughly 0.01%. All counters are initialized to 0 and only read/updated by atomic memory operations to ensure the consistency among all threads.

Optimistic Locking: Before displacing a key, an `Insert` process first increases the relevant counter by one, indicating to the other Lookups an on-going update for this key; after the key is moved to its new location, the counter is again increased by one to indicate the completion. As a result, the key version is increased by 2 after each displacement.

Before a Lookup process reads the two buckets for a given key, it first snapshots the version stored in its counter: If this version is odd, there must be a concurrent `Insert` working on the same key (or another key sharing the same counter), and it should wait and retry; otherwise it proceeds to the two buckets. After it finishes reading both buckets, it snapshots the counter again and compares its new version with the old version. If two versions differ, the writer must have modified this key, and the Lookup should retry.

2.2.2 Optimization: Multiple Cuckoo Paths

Our revised `Insert` process first looks for a valid cuckoo path before swapping the key along the path. Due to the separation of search and execution phases, we apply the following optimization to speed path discovery and increase the chance of finding an empty slot.

Instead of searching for an empty slot along one cuckoo path, our `Insert` process keeps track of multiple paths in parallel. At each step, multiple victim keys are "kicked out," each key extending its own cuckoo path. Whenever one path reaches an available bucket, this search phase completes.

3 Concurrent Cache Management

When serving small key-value objects, this too becomes a major source of space overhead in Memcached, which requires 18 Bytes for each key (i.e., two pointers and a 2-Byte reference counter) to ensure that keys can be evicted safely in a strict LRU order. String LRU cache management is also a synchronization bottleneck, as all updates to the cache must be serialized in Memcached.

This section presents our efforts to make the cache management *space efficient* (1 bit per key) and *concurrent* (no synchronization to update LRU) by implementing an approximate LRU cache based on the CLOCK replacement algorithm. CLOCK is a well-known algorithm; our contribution lies in integrating CLOCK replacement with the optimistic, striped locking in our cuckoo algorithm to reduce both locking and space overhead.

CLOCK Replacement: A cache must implement two functions related to its replacement policy:

- `Update` to keep track of the recency after querying a key in the cache
- `Evict` to select keys to purge when inserting keys into a full cache

Memcached keeps each key-value entry in a doubly-linked-list based LRU queue within its own slab class. After each cache query, `Update` moves the accessed entry to the head of its own queue; to free space when the cache is full, `Evict` replaces the entry on the tail of the queue by the new key-value pair. This ensures strict LRU eviction in each queue, but unfortunately it also requires two pointers per key for the doubly-linked list and, more importantly, all Updates to one linked list are serialized. Every read access requires an update, and thus the queue permits no concurrency even for read-only workloads.

CLOCK approximates LRU with improved concurrency and space efficiency. For each slab class, we maintain a **circular buffer** and a **virtual hand**; each bit in the buffer represents the recency of a differen key-value object: 1 for "recently used" and 0 otherwise. Each `Update` simply sets the recency bit to 1 on each key access; each `Evict` checks the bit currently pointed by the hand. If the current bit is 0, `Evict` selects the corresponding key-value object; otherwise we reset this bit to 0 and advance the hand in the circular buffer until we see a bit of 0.

Integration with Optimisitc Cuckoo Hashing: The `Evict` process must coordinate with reader threads to ensure the eviction is safe. Otherwise, a key-value entry may be overwritten by a new key-value pair after eviction, but threads still accessing the entry for the evicted key may read dirty data. To this end, the original Memcached adds to each entry a 2-Byte reference counter to avoid this rare case. Reading this per-entry counter, the `Evict` process knows how many other threads are accessing this entry concurrently and avoid evicting those busy entries.

Our cache integrates cache eviction with our optimistic locking scheme for cuckoo hashing. When `Evict` selects a victim key x by CLOCK, it first increases key x's version counter to inform other threads currently reading x to retry; it then deletes x from the hash table to make x unreachable for later readers, including those retries; and finally it increases key x's version counter again to complete the change for x. Note that `Evict` and the hash table `Insert` are both serialized (using locks) so when updating the counters they can not affect each other.

With Evict as above, our cache ensures consistent `GET`s by version checking. Each `GET` first snapshots the version of the key before accessing the hash table; if the hash table returns a valid pointer, it follows the pointer and reads the value assoicated. Afterwards, `GET` compares the latest key version with the snapshot. If the verions differ, then `GET` may have observed an inconsistent intermediate state and must retry.

Algorithm 1: Psuedo code of SET and GET

```
SET(key, value)    //insert (key,value) to cache
begin
    lock();
    ptr = Alloc();        //try to allocate space
    if ptr == NULL then
        ptr = Evict();   //cache is full, evict old item
    memcpy key, value to ptr;
    Insert(key, ptr); //index this key in hashtable
    unlock();

GET(key)    //get value of key from cache
begin
    while true do
        vs = ReadCounter(key);   //key version
        ptr= Lookup(key);        //check hash table
        if ptr == NULL then  return NULL ;
        prepare response for data in ptr;
        ve = ReadCounter(key);   //key version
        if vs & 1 or vs != ve then
            //may read dirty data, try again
            continue
        Update(key);          //update CLOCK
        return response
```

4 Problemsp

5 References