PAPER SHARE: Optimizing Flash-based Key-value Cache Systems

2025.02.27

Jun

Background:Slab Allocator

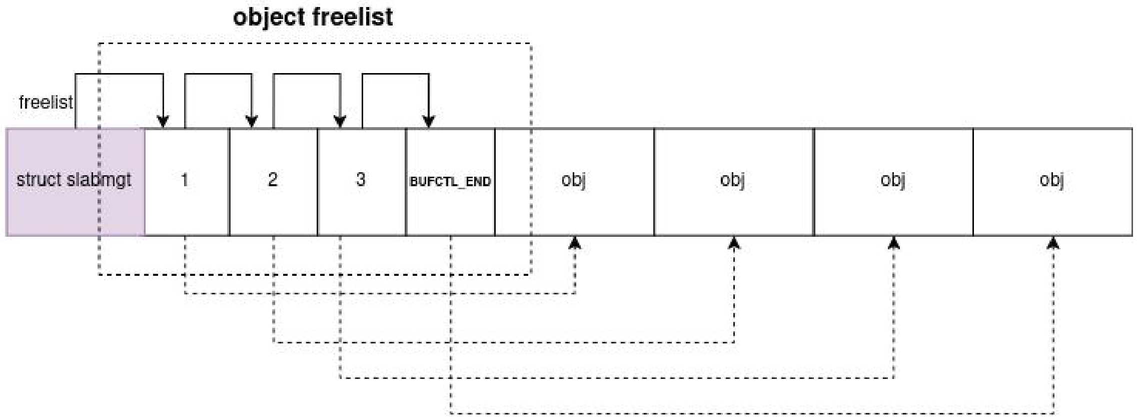
PAPER: The slab allocator: An object-caching kernel memory allocator.

Slab allocator is self-tuning segregatted list in some sense.

Slab: The basic memory management unit(A continuous memory area)

Slot: A slot is an equally-sized memory block within a slab, used for storing object.

Freelist: Freelist manages all free slots. The core of allocation/deallocation is to remove objects frm or return objects back to the freelist.



Background:KV Cache On SSD

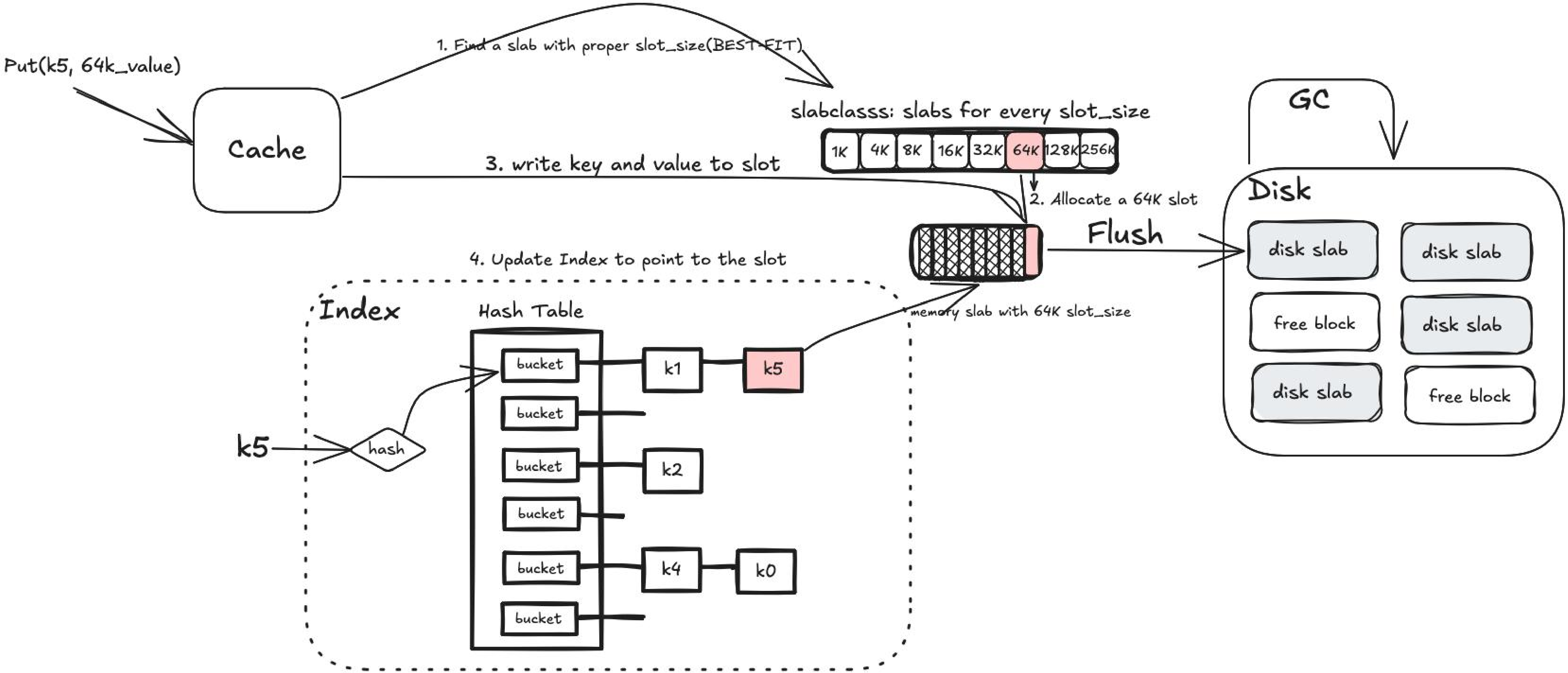
Two key differences:

1. Single layer memory slab allocator -> Two layers(memory and disk) slab allcoator
2. KV Cache has to manage indexes of objects, while kernel slab allcoator doesn't.

Key ideas:

1. Buffer writes in memory slab(mslab) and flush them into disk slab(dslab) later.
2. Index objects using hash table for fast query.

Background:KV Cache On SSD



Issues: Redundency

Issues:

1. Redundent mapping
    1. Cache index and SSD FTL mapping
    2. Redundent SSD page level mapping
2. Double garbage collection(GC)
    1. Cache GC and SSD GC
    2. SSD GC can't free blocks even the data is garbege at Cache level
3. Wasted over-provisioning space(OPS)

Solutions: Bypass SSD

Application Layer: Manage SSD at Aplication Level

1. Redundent mapping -> application level slab/block mapping
2. Double garbage collection(GC) -> application level slab/block GC
3. Wasted over-provisioning space(OPS) -> application level self-tuning OPS

SSD Layer: Customed SSD

1. Bypass redundent FTL functions: mapping, wear-leveling and GC.
2. Retain few FTL functions: error handling, flash control and etc.

Implementation: SSD

1. Simulate Open-Channel SSD by RocksDV KV pair: BlockID -> BlockData
2. Simulate Open-Channel SSD by Linux disk device read/write API for benchmark
3. Implement channel load balancing at application level

```
// RocksDB keys
/nr_blocks
/blocks_per_channel
/nr_channels
/blocks/<channel-id>/<block-id> -> Block Data
```

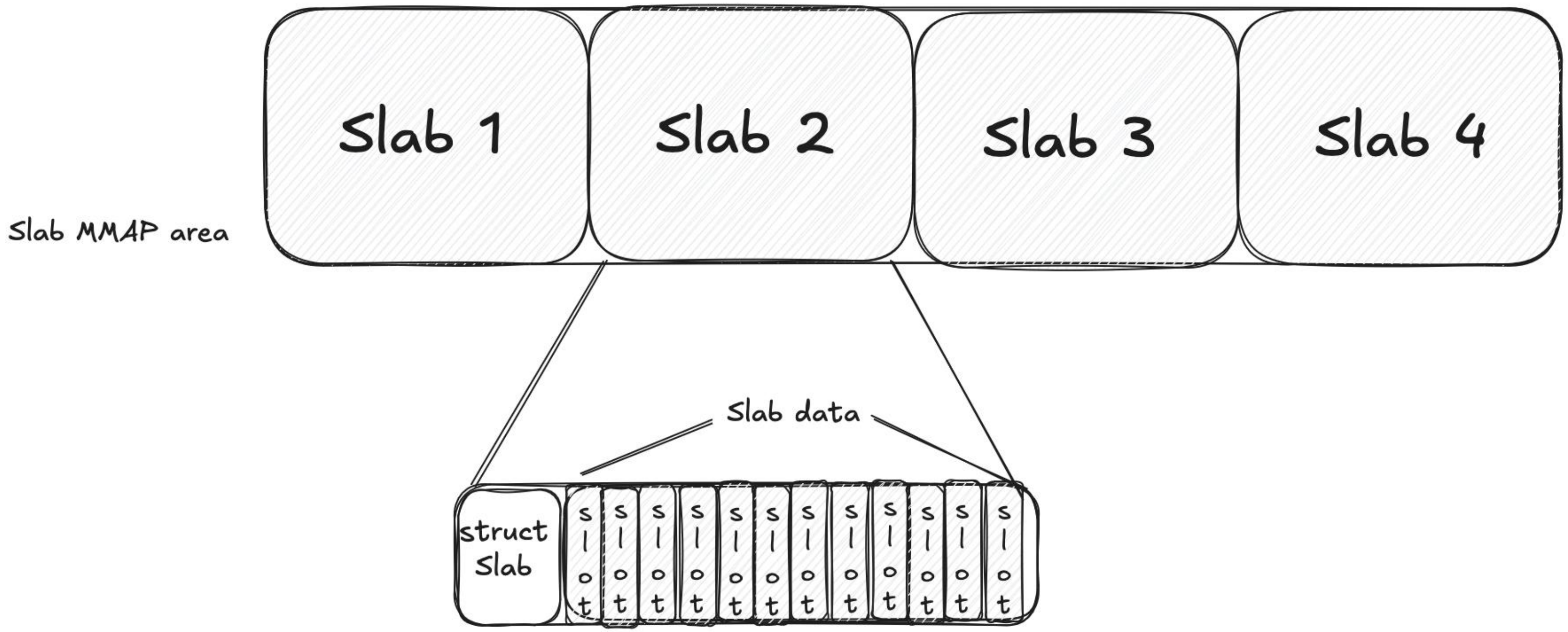
```
// Open disk device, such as /dev/loop0.
fd_ = open(device, O_RDWR | O_DIRECT, 0644);

// The block is located at the disk device offset of block_id * block_size.
offset = block_id * block_size;
// Read block to read_buffer.
nread = pread(fd_, read_buffer, block_size, block_id*block_size);
// Write new block_data to the block
nwrite = pwrite(fd_, block_data, block_size, offset);
```

Implementation: Mapping

Mapping:

1. A slab occupies a block, so a slab is equivalent to a block.

2. SSD page level mapping -> application level slab/block mapping



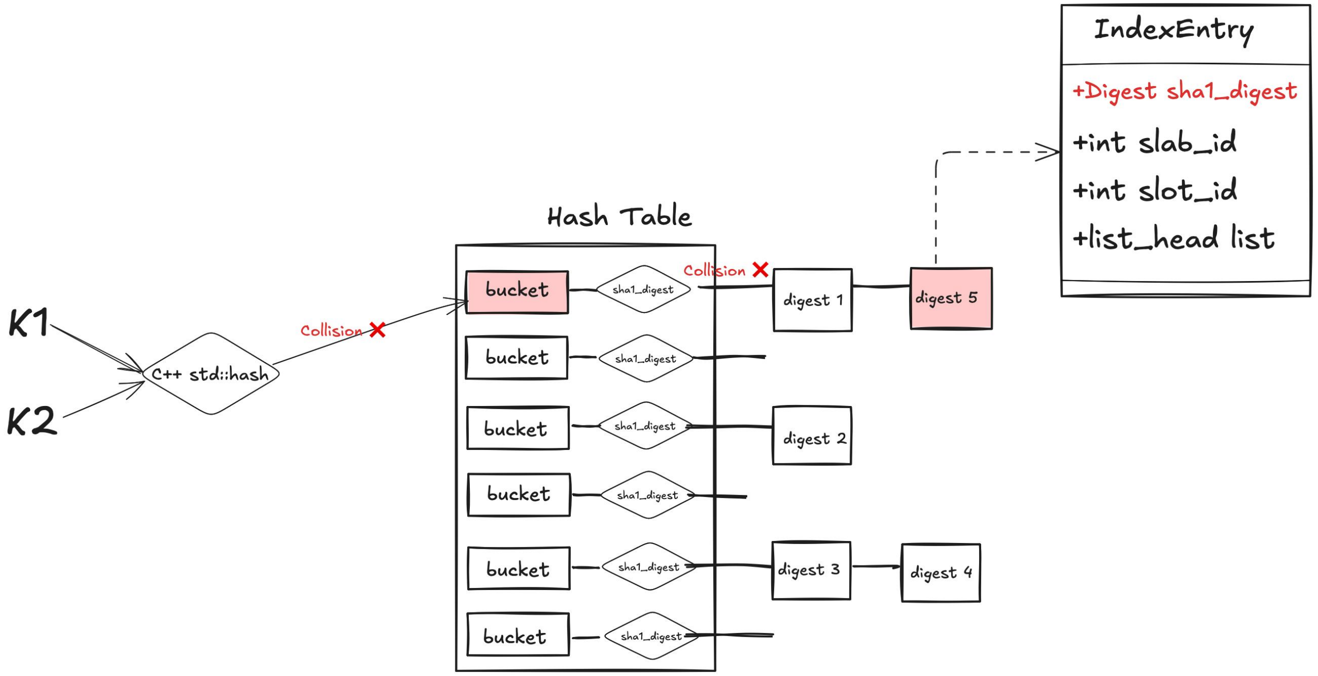
Implementation: Index

1. Store SHA1 digest(20 bytes) instead of the original key to reduce index size.

2. It is possible to map two different keys in a same index entry due to hash collision.

3. PUT overwites the index entry with the same hash value.

4. GET compares the key after reading the KV pair, thereby avoiding returning the conflicting key.



Implementation: Index Efficiency

Efficiency:

1. Current index entry size: 20B SHA1 digest + 8B slab_8 id + 8B slot_id + 8B list pointer = 36B
2. The index entry size can be further reduced.
    1. SHA1 digest -> MD5 digest or other algorithms with shorter length
    2. Store the original key when the key length is less than the hash digest length.
    3. Reduce the linked list pointer to 4 bytes (capable of indexing 4G entries) or even smaller.

Implementation: Concurrency

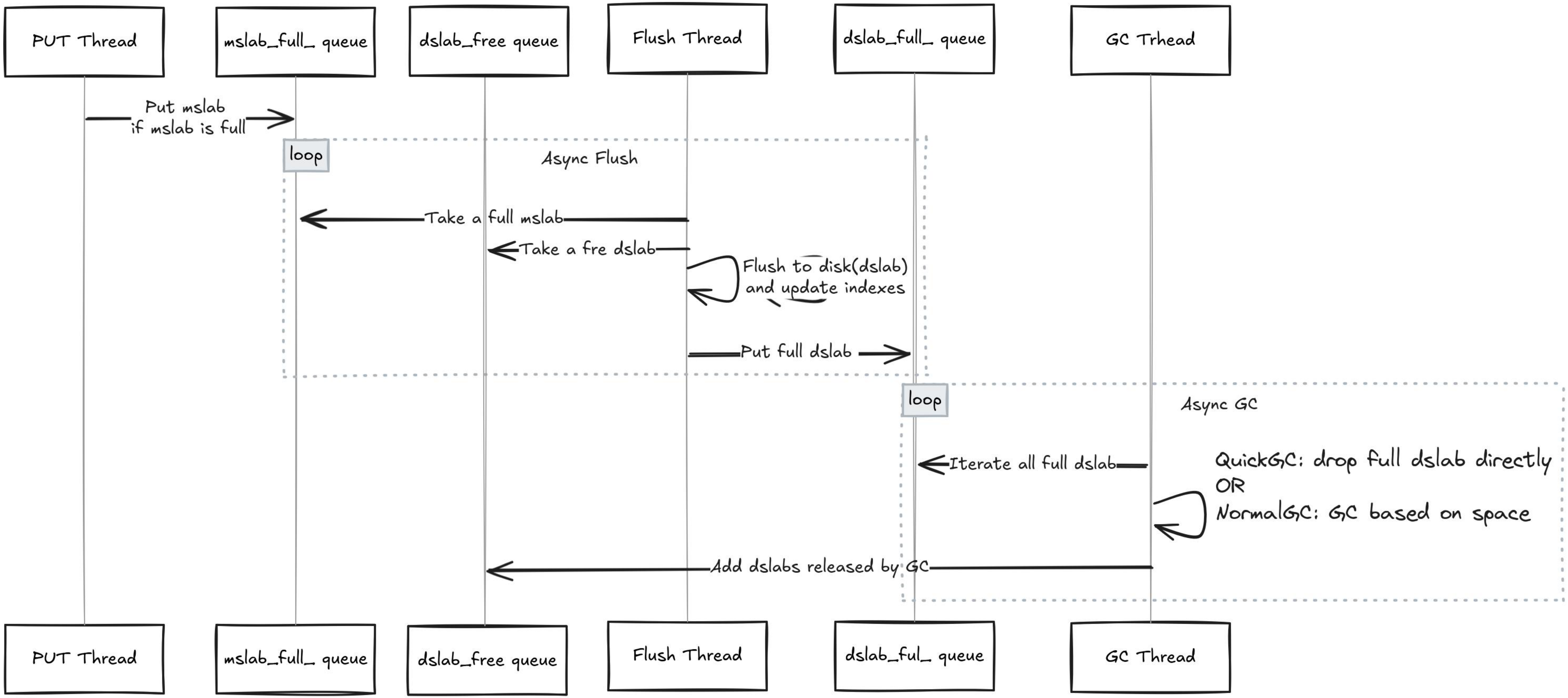
API:

1. PUT and DELETE are serialized through locks.
2. GET can be executed concurrently with PUT/DELETE.

Internals:

1. The critical section mainly involves reading and writing the index.
2. Both flush and GC are executed by background threads.
3. Critical sections do not involve disk IO.

Implement: Concurrency



Implementation: Self-Tuning OPS

Goal: Increase OPS under high write pressure and decrease OPS under low write pressure.

Key Ideas: Tune OPS based on the number of free blocks and watermarks

1. [low_watermark, high_watermark]: system is under light write presure.

2. [0, low_watermark]: system is under high write pressure.

3 [high_watermark, nr_blocks]: system has no write pressure.

Implementation: GC

1. Run self-tuning OPS algorithm in every rounds of GC.

2. NormalGC: Space-based GC to reclaim as much disk space and index entries as possible.

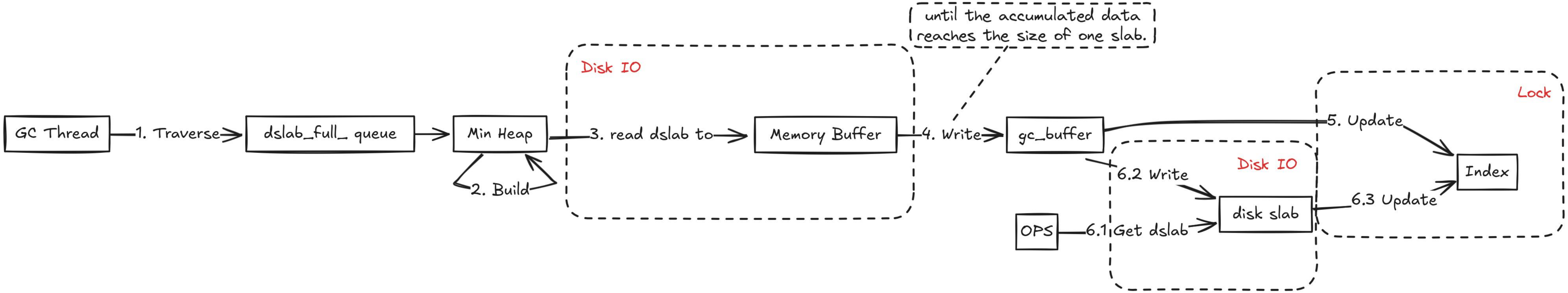
2. QuickGC: Drop slabs directly to response write pressure quickly.

Implementation: QuickGC

1. Drop 1.8*low_watermark full dslabs.
2. Return a portion of the dslab to OPS, doubling the OPS size.
3. The remainning portion is returned to dslab_free_ queue(free blocks).
4. Increase low_watermark/high_watermark to 1.5X.

Implementation: NormalGC

1. GC the full dslabs containing the largest number of obsolete value items.
2. Decrease watermark and OPS linearly.
3. The critical section only involves index modification without disk IO.



Evaluation: Method

1. Google Test benchmark framework
    1. Unit tests: Basic operations, edge cases, concurrency tests and etc.
    2. Benchmark: PUT, GET and DELETE 4*ssd_SIZE data on local SSD.
2. Twitter's twemperf: A memcached benchmark tool
    1. Use Twitter's fatcache as the baseline of performance benchmark.
    2. Replace fatcache's PUT/GET implementation with my own implementation.

Evaluation: Tool

1. /dev/loop: I mounted the file as /dev/loop because fatcache needs to directly open the disk device and read/write to it.

2. twemperf: Send SET/GET commands to Memcached at a specific rate.

Evaluation: Local SSD

Workload: Put and Get 4 SSDs with random key and value size single-threadedly.

Simulated SSD configuration: 200MiB SSD with 200KiB block size.

KVCache configuration: 10MiB Slab memory, 1Mib index memory and 512K hash buckets.

Local SSD:970MiB/s seqwrite and **694MiB/s** for 200KiB block size

Performance Benchmark Result:

write_seconds: 1.6887 read_seconds: 0.446772

Write: 4913.83 ops/s, actual write **473.80** MiB/s

Read: 111314.07 ops/s, actual read **2062.72** MiB/s

Evaluation: Heavy Write Workload

fatcache configuration: 4KiB slab_size, 64MiB index memory, 64MiB slab memory and **1GiB** SSD.

Write **1GiB** data: src/mcperf --sizes=u100,100 --num-calls=10000 --num-conns=100 --call-rate=1000 --conn-rate=10000 --method=put/get --server=0.0.0.0 --port=11211

	KVCache(RocksDB)	KVCache(dev)	fatcache
Put	99871.7 rsq/s	99860.1 rsq/s	99831.8 rsq/s
Get	**97751.2 rsq/s**	**90967.3 rsq/s**	65229.4 rsq/s
Cache-Hit Rate	54%	54%	100%

Evaluation: Heavy Write Workload

1. RocksDB version achives better write performance because RocksDB buffers data in memtable.

2. /dev/loop0 version achives worse performance because of redundent memory copy.

3. OPS occupies 20% of the disk, and quickGC drops 26% slabs, meaning that only 54% of the space is actually used to store objects.

	KVCache(RocksDB)	KVCache(dev)	fatcache	KVCache (SSD*1.2)
Put	99871.7 rsq/s	99860.1 rsq/s	99831.8 rsq/s	**85385.3 req/s**
Get	97751.2 rsq/s	90967.3 rsq/s	65229.4 rsq/s	**73493.8 req/s**
Cache-Hit Rate	54%	54%	100%	68.7%

Evaluation: Heavy Flush/GC Workload

1. Asynchronous flush and garbage collection enable KVCache to achieve higher performance.

2. Fatcache is blocked on flush and GC, causing test timeout.

3. Given enough time for flush and GC, Fatcache will achieve better Get performance.

	KVCache(RocksDB)	KVCache(dev)	fatcache
Put	45907.8 rsq/s	40757.7 rsq/s	**31978.1** rsq/s
Get	32380.4 rsq/s	35981.3 rsq/s	**7386.9**(46565.2) rsq/s

Evaluation: Async Flush and GC

1. Quick GC and self-tunning OPS algorithm is too agile, relaiming too much slabs.

2. GC locks the index hash table, thereby reducing GET performance.

2. Async flush has significent positive impact on PUT/GET performance.

	flush = 0 gc = 0	flush = 0 gc = 1	flush = 1 gc = 0	flush = 1 gc = 1
Put	23991.3 rsp/s	17358.3 rsp/s	46035.5 rsq/s	51114.2 req/s
Get	38286.9 rsp/s	24186.8 rsp/s	69444.2 rsp/s	32965.2 req/s