Data structures to maintain hot data for RocksDB

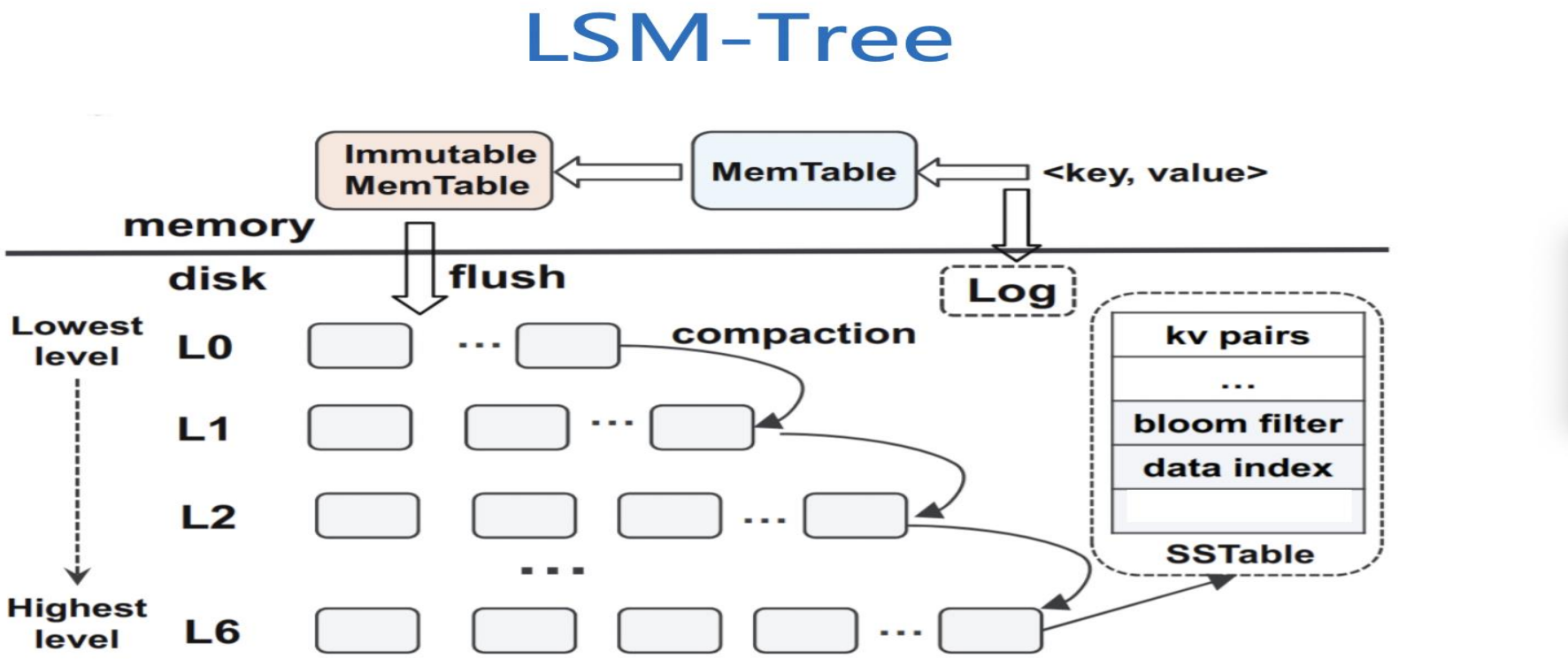
Tanuj Gupta [1]

[1]*Graduate Student, Department of Computer Engineering, UCSC*

Today's Agenda

- Rocksdb Background and Get Operation
- Rocksdb Approach to Address Slow Reads
- Limitations of Current Approaches
- When and How to Mark a Key as Hot Data
- What Data Structures to Use to Store Hot Data
- How to Deal With Relationship of "Hot Data" and "New Values"
- Drawbacks

BACKGROUND -> ROCKSDB



- The architecture of the LSM-tree storage.
- Compared with the B+ tree, LSM tree tradeoff random read performance for write performance.

READ/GET OPERATION IN ROCKSDB ?

Linearly scan all SST files level by level?
Linearly scan all data blocks inside SST file?

ROCKSDB APPROACH FOR SLOW READS

Sorted SST's every level

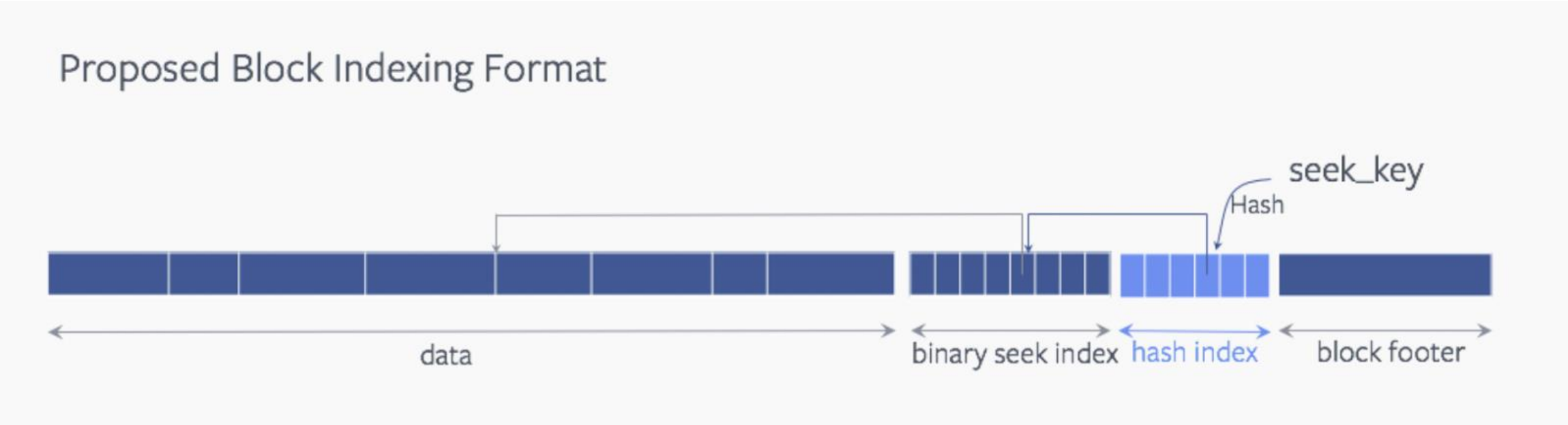
+

<beginning_of_file>
[data block 1]
[data block 2]
...

[data block N]
[meta block 1: filter block]
[meta block 2: index block]
[meta block 3: Hash index block]
[meta block 4: compression dictionary block]
[meta block 5: range deletion block]
[meta block 6: stats block]
[meta block K: future extended block]
[metaindex block]
[Footer]
<end_of_file>

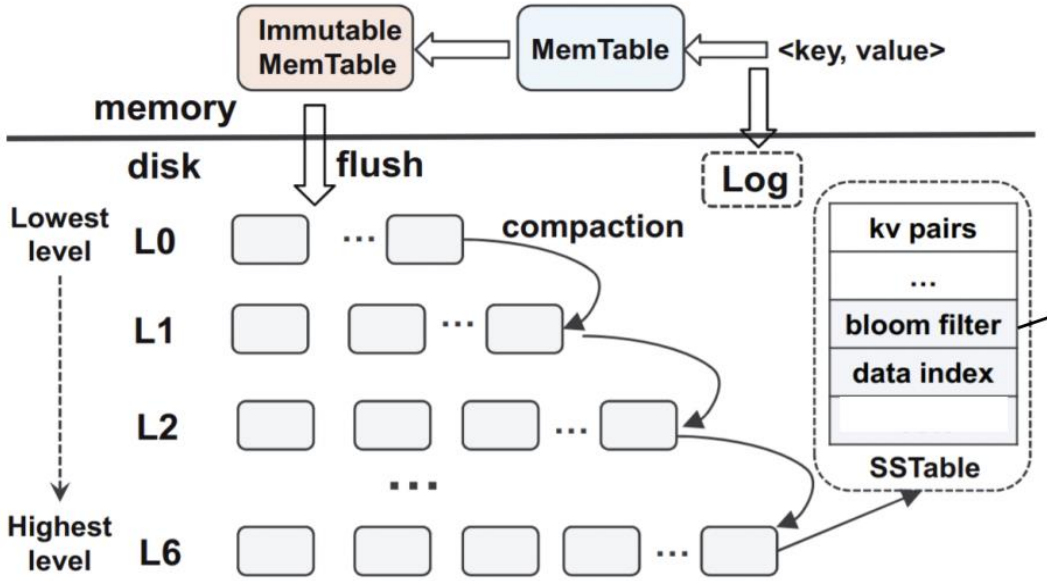
+ CACHING

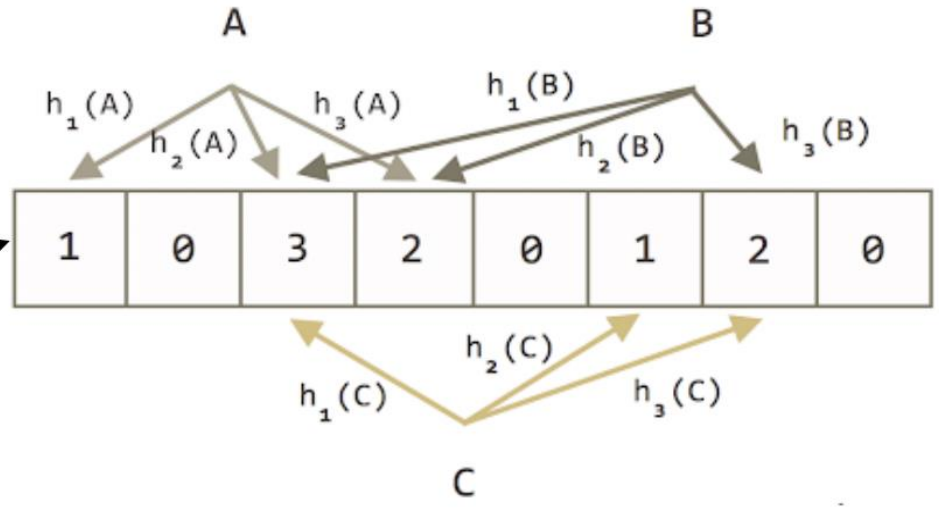
Index Block And Hash Index



Bloom Filters

- The bloom filter (typically) requires 10 bits per key in RocksDB.





- We do not use a Bloom filter at the last level. ◦
  - The last level bloom filter is large (~9X as large as all lower-level Bloom filters combined)

RocksDB Cache

RocksDB uses two implementations of cache:

1) LRUCache

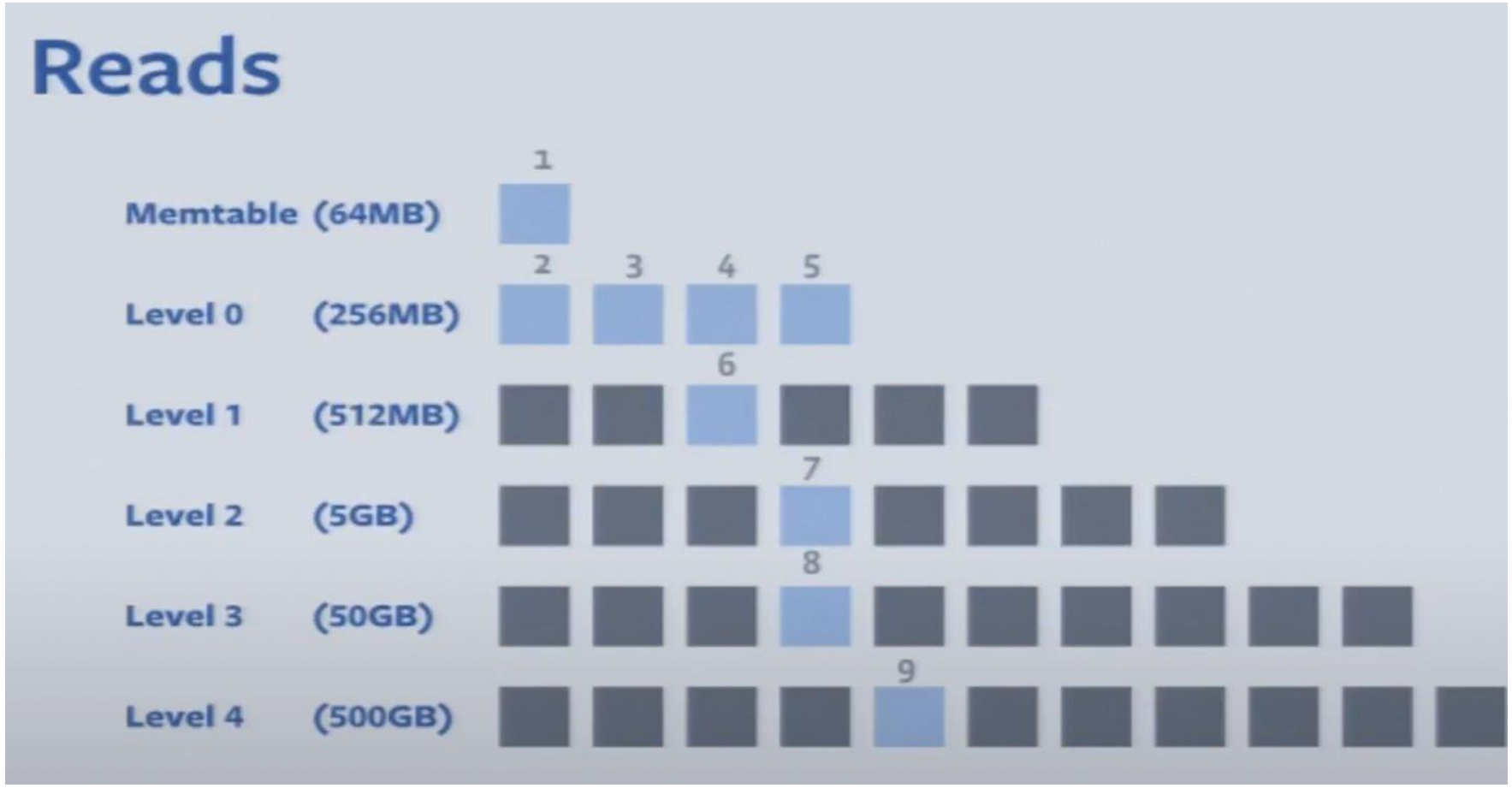
2) ClockCache.

Both types of cache are sharded to mitigate lock contention. Capacity is divided evenly to each shard. By default, each cache will be sharded into at most 64 shards, with each shard having no less than 512k bytes of capacity.

READ/GET OPERATION IN ROCKSDB

- Scan Sharded Cache (in-memory,fast but small)
- Scan Memtable (in-memory but linear)
- Scan l0 level SST files (in-memory but linear)
- Level l1 to L -> (Disk but sorted data)
  - -Read bloom filter block against 1 SST (at least 1 I/O per level)
  - -hash index to find data block O(1)
  - -index block to find data block O(log(n))

READ/GET OPERATION IN ROCKSDB



LIMITATIONS OF CURRENT APPROACHES

- Prioritize new writes
- Keys at lower levels incur high I/O for every read.
- Cache is very small but Data at lower level is huge

- Disk I/O -> L(once per every level) + 1(hash index) + 1 (log(n) index block binary search) +1(data block)

- Memory reads -> Cahe size + memtable size + L0 size

Solution?

Push hot data to higher levels manually.

Challenges:

a) When and how to mark a key as hot data?

b) What data structures to use to store hot data?

c) When to manually write/push the hot data on some higher level?

d) How to deal with the relationship of "hot data" and new values?

WHEN AND HOW TO MARK A KEY AS HOT DATA

WHAT DEFINES HOT DATA?

- 1) The time between the updates of the key is very large. Because of this, the key stays at a lower level at adds to high read latency.
  - some extra operation on write?

- 2) The key is read frequently. This makes the key valuable and important, something we can't afford to have a slow response on.
  - some extra operation on read?

Some extra operation on write?

- Cannot afford to slow writes in RocksDB
- Promises High throughput
- Heavy write based workloads.

Solution?

- So let's define a threshold level Lt. Any data below level Lt, will not be considered as hot data for obvious reasons
- On every read, if the key is found below level Lt, we will increment its count.
- If count reaches above threshold N, mark key as hot data

Where to store these counts?

Goals

- All the keys below a threshold level(Lt) are the possible candidates for hotkeys. Space-efficient structure needed
- Update/increment read count
- Get read count

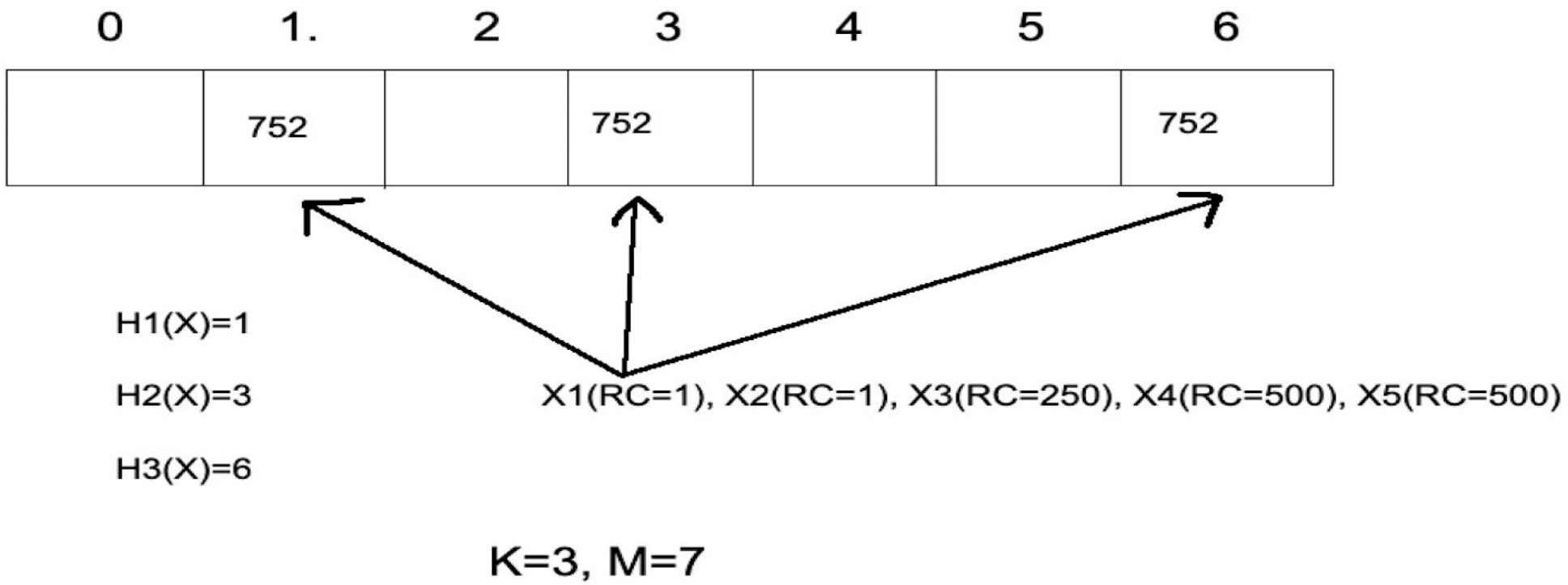
COUTING BLOOM FILTER FOR COUNTS

- Space-efficient, O(1) get and update

Wrong prediction: False positive in Bloom Filters

- May end up selecting the wrong candidate for hot data.
- Will waste a lot of CPU bandwidth and space not-so-important keys

Wrong prediction: False positive in Bloom Filters



Reducing selection probability of the wrong candidate

Use second counting bloom filter

- Only maintains read count for selected candidates from bloom filter 1
- far fewer elements will be competing
- far fewer collisions, and hence fewer chances of the wrong prediction
- Data is hot if counting bloom filter 2 says so

Algorithm

1. Any data below level Lt, will not be considered as hot data for obvious reasons
2. For data above level Lt, Create a counting bloom filter to store the number of possible reads for the key.
3. On read, if data lies at level > Lt and

A. Update/Increment the count of the key against all the hashes in Read_Counting_bloom_filter. (O(1) time)

B. If the count of all hashes of the key in bloom filter is greater than Threshold Number N1 (O(1) time):

If the key is in Possible_Hotdata_Counting_bloom_filter**:**

Update/Increment the count of the key against all the hashes in Possible_Hotdata_Counting_bloom_filter (O(1) time)**.**

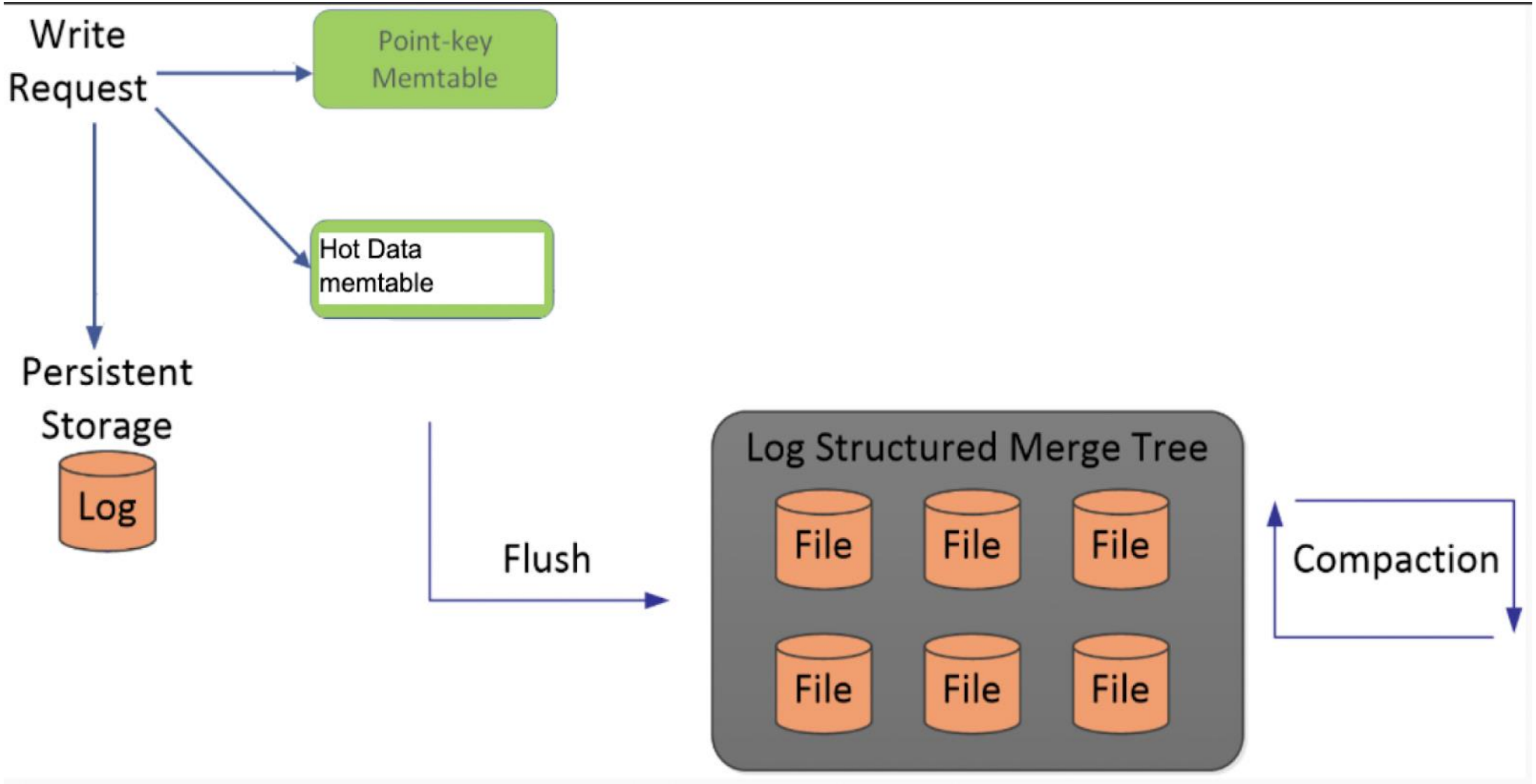
If the count in Possible_Hotdata_Counting_bloom_filter > N2:

Push the key to Hotdata_Memtable (O(1) time)

Else

Push the key to Possible_Hotdata_Counting_bloom_filter (O(1) time)

DATA STRUCTURES TO STORE HOT DATA



Deal with "hot data" and "new values"

Time t1 -> the time when read that selects a key k as hot was initiated

V1 -> the value of this key k at t1

Time t2 -> key was updated to a new value V2

Time t3 -> the time when key k was selected as hot data

Time t4 -> key was updated to a new value V3

Time t5 -> Hot key with value v1 flushed

Challenge ->

- Hot data (V1) flush may overwrite the latest updates of the key (V2 and V3).

Solution ->

- Time ordering between "hot data" and "new values"?
  - -V3 will not be overwritten
- What about V2? Update that happens between t1 and t2?

Write that happened between t1 and t2

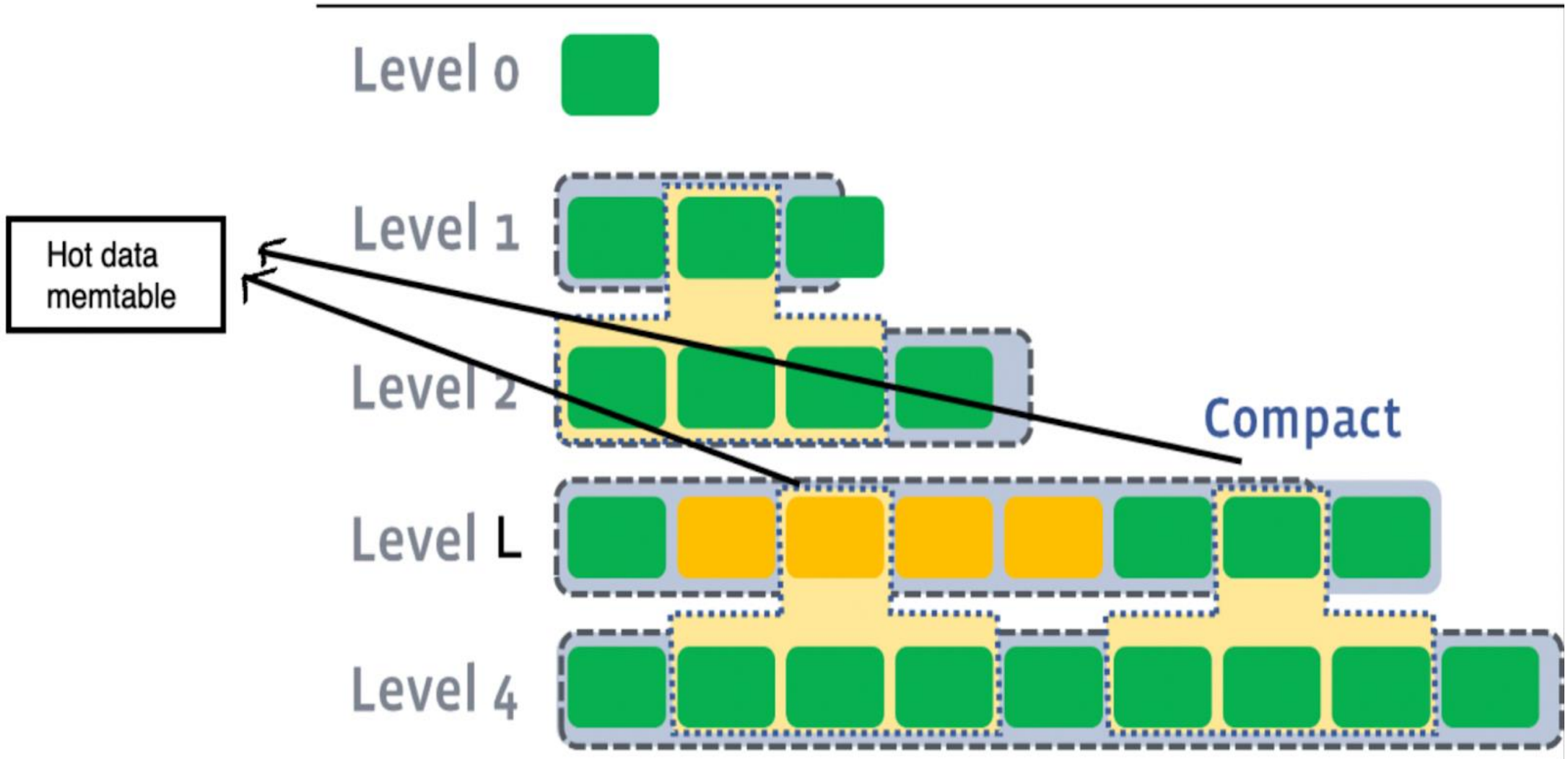
Assumption

- write/update in RocksDB takes time to push down to lower levels
- the time interval between t1 and t2 will be very less in general

Any update or write that happened between t1 and t2(say t3) will not be pushed below certain level L

Solution

- Time ordering of HOT data memtabe writes and all the compactions that push SST from level L to L+1.


Level 0
Level 1
Level 2
Level L
Level 4
Hot data
memtable
Compact

LEVELED VS UNIVERSAL COMPACTION

- Our solution: every compaction from Level L to Level L+1 reads Hot table Memtable.
- Proportional to number of compactions from Level L to Level L+1.

Hence

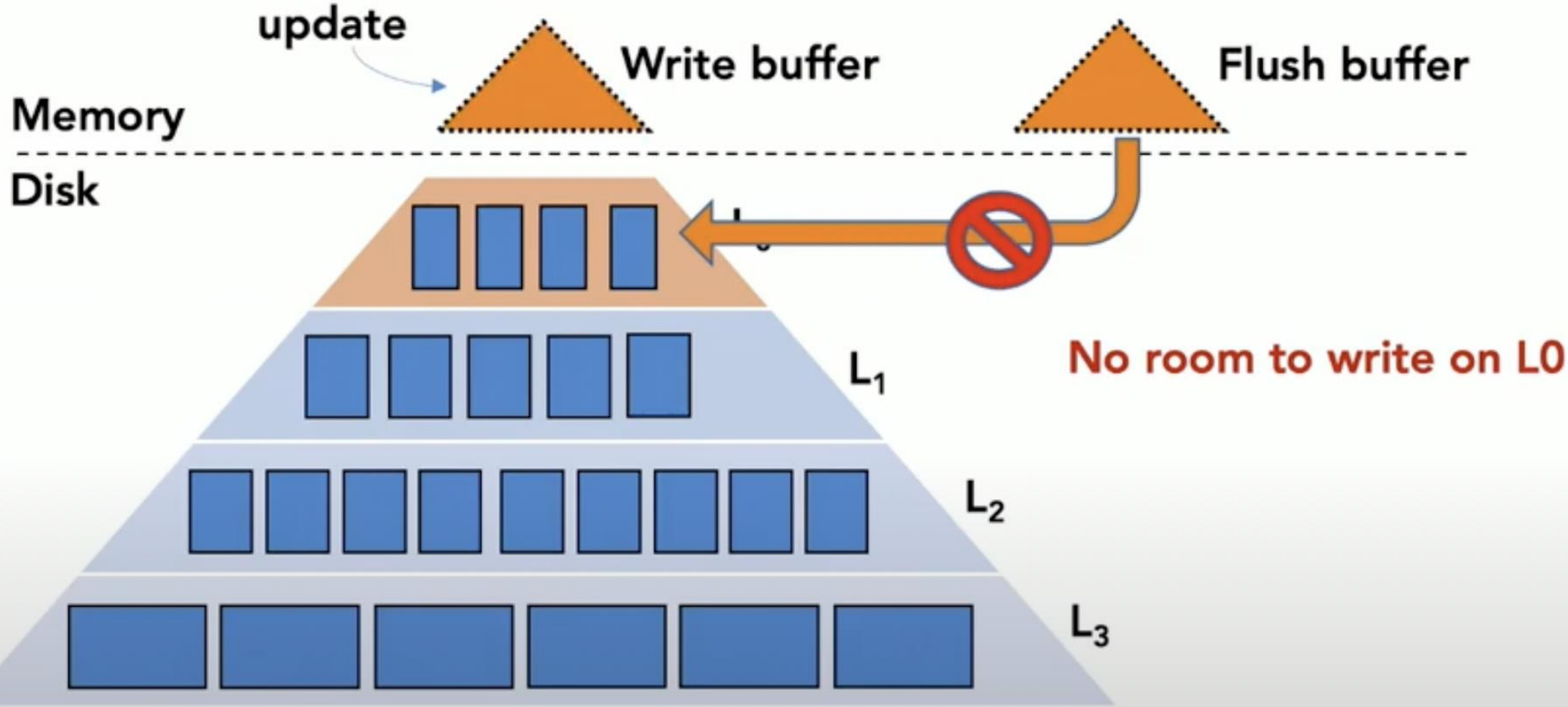
- Work much faster with universal compaction in comparison to leveled compaction.

Internal operations:

1. Flushing. From memory to disk.
2. L0 → L1 compaction. Make room to flush new files.
3. Higher level compactions. ~GC, I/O intensive.

No coordination between internal ops and client ops.

L0 Full, Cannot Flush



SELECTING VALUE FOR LEVEL L

Higher value for L

- Increase read response. Opposite to what we want
- L < Lt

Low value for L

- Assumption fails
- Flushing extra hot table mem table at higher levels will stall writes at level 0.

Somewhere in the middle, that captures best of both the worlds. Something around 3 or 4 seems reasonable.

CONFIGURABLE PARAMETERS

1. **Lt**-> Minimum level to be considered for Hot Data. (Recommended L> 6)
2. **L**-> Level where Hotdata memtable will be flushed. (Recommended L < Lt)
3. **M1,K1**-> Size and hash functions used for Read_Counting_bloom_filter
4. **M2,K3**-> Size and hash functions used for Possible_Hotdata_Counting_bloom_filter
5. **Threshold Number N**-> To decide when to push candidate key to Possible_Hotdata_Counting_bloom_filter.
6. **Threshold Number N2**-> To decide when to push candidate key to Possible_Hotdata_Counting_bloom_filter.

DRAWBACKS

- Targets very specific workloads -> Read heavy
- A Lot of extra contention at level L.
- Adds extra CPU cycles to maintain and update all the data structures used
- Slows down all reads for keys at level>Lt
- Based on the assumption -> “any update or write that happened between t1 and t2(say t3) will still be at level L or higher”

FUTURE WORK

A. Implementation

B. Benchmarking and testing

C. Deal with the new values being written