PebblesDB: Building Key-Valued Stores using Fragmented Log-Structured Merge Trees

April 18, 2025

1 Introduction

One fundamental problem that remains is the high write amplification of key-value stores for write-intensive workloads.

Keep in mind how does pebblesdb solve write amplification

Figure 1 shows the high write amplification (ratio of total IO to total user data written) that occurs in several widely-used key-value stores when 500 million key-value pairs are inserted or updated in random order.

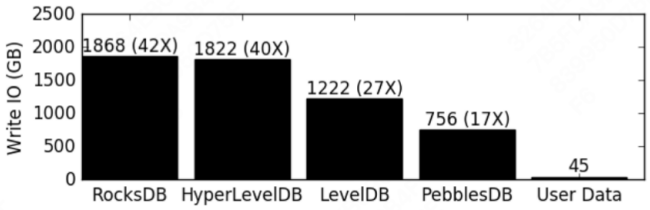

Figure 1: Write Amplification. The figure shows the total write IO (in GB) for different key-value stores when 500 million key-value pairs (totaling 45 GB) are inserted or updated. The write amplification is indicated in parenthesis.

Conventional wisdom is that reducing write amplification requires sacrificing either write or read throughput. In today's low-latency, write-intensive environments , users are not willing to sacrifice either.

2 Fragmented Log-Structured Merge Trees

The challenge is to achieve three goals simultaneously:

- low write amplification
- high write throughput
- good read performance

2.1 Guards

In the classical LSM, each level contains sstables with disjoint key ranges (i.e., each key will be present in exactly one sstable). The chief insight in this work is that maintaining this invariant is the root cause of write amplification, as it forces data to be rewritten in the same level.

The FLSM data structure discards this invariant: each level can contain multiple sstables with overlapping key ranges, so that a key may be present in multiple sstables. To quickly find keys in each level, FLSM organizes the sstables into guards

Each level contains multiple guards. Guards divide the key space (for that level) into disjoint units. Each guard G_i has an associated key K_i , chosen from among keys inserted into the FLSM. Each level in the FLSM contains more guards than the level above it; the guards get progressively more fine-grained as the data gets pushed deeper and deeper into the FLSM. As in a skiplist, if a key is a guard at a given level i, it will be a guard for all levels $> i$.

Each guard has a set of associated sstables. Each sstable is sorted. If guard G_i is associated with key K_i and guard G_{i+1} with K_{i+1}, an sstable with keys in the range $[K_i, K_{i+1})$ will be attached to G_i . Sstables with keys smaller than the first guard are stored in a special sentinel guard in each level. The last guard G_n in the level stores all sstables with keys $\geq K_n$. Guards within a level never have overlapping key ranges. Thus, to find a key in a given level, only one guard will have to be examined.

In FLSM compaction, the sstables of a given guard are (merge) sorted and then fragmented (partitioned), so that each child guard receives a new sstable that fits into the key range of that child guard in the next level.

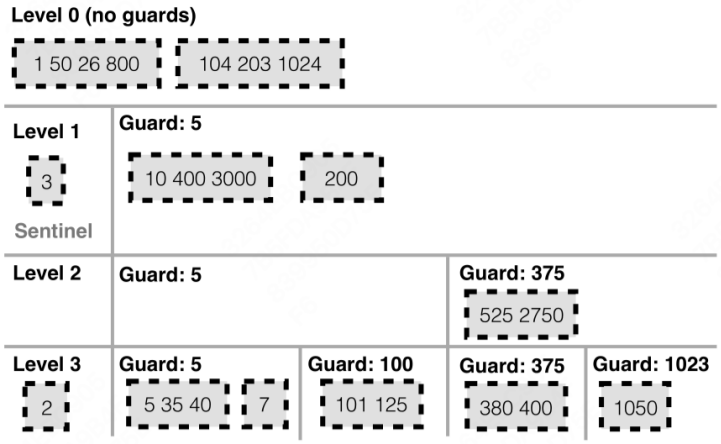

Figure 3: FLSM Layout on Storage. The figure illustrates FLSM's guards across different levels. Each box with dotted outline is an sstable, and the numbers represent keys.

Example 2.1. Figure 2.1 shows the state of the FLSM data structure after a few `put()` operations.

- A `put()` results in keys being added to the in-memory memtable (not shown). Eventually, the memtable becomes full, and is written as an sstable to Level 0. Level 0 does not have guards, and collects together recently written sstables.
- The number of guards increases as the level number increases. The number of guards in each level does not necessarily increase exponentially.
- Each level has a sentinel guard that is responsible for sstables with keys $<$ than the first guard. In Figure 2.1, sstables with keys < 5 are attached to the sentinel guard.
- Data inside an FLSM level is partially sorted: guards do not have overlapping key ranges, but the sstables attached to each guard can have overlapping key ranges.

2.2 Selecting Guards

- Since: FLSM performance is significantly impacted by how guards are selected. In the worst case, if one guard contains all sstables, reading and searching such a large guard (and all its constituent sstables) would cause an un-acceptable increase in latency for reads and range queries.
- Therefore: guards are not selected statically; guards are selected probabilistically from inserted keys, preventing skew.

Guard Probability. When a key is inserted into FLSM, guard probability determines if it becomes a guard. Guard probability `gp(key, i)` is the probability that key becomes a guard at level `i`. For example, if the guard probability is 1/10, one in every 10 inserted keys will be randomly selected to be a guard. The guard probability is designed to be lowest at Level 1 (which has the fewest guards), and it increases with the level number (as higher levels have more guards). Selecting guards in this manner distributes guards across the inserted keys in a smooth fashion that is likely to prevent skew.

Much like skip lists, if a key K is selected as a guard in level i, it becomes a guard for all higher levels $i+1$, $i+2$ etc. The guards in level $i+1$ are a strict superset of the guards in level i. Choosing guards in this manner allows the interval between each guard to be successively refined in each deeper

Othter schemes for selecting guards. Probability does not take into account the amount of IO that will result from partitioning sstables during compaction. FLSM could potentially select new guards for each level at compaction time such that sstable partitions are minimized; however, this could introduce skew.

2.3 Inserting and Deleting Guards

When guards are selected, they are added to an in-memory set termed the **uncommitted** guards. Sstables are not partitioned on storage based on (as of yet) uncommitted guards; as a result, FLSM reads are performed as if these guards did not exist. At the next compaction cycle, sstables are partitioned and compacted based on both old guards and uncommitted guards; any sstable that needs to be split due to an uncommitted guard is compacted to the next level. At the end of compaction, the uncommitted guards are persisted on storage and added to the full set of guards. Future reads will be performed based on the full set of guards.

We note that in many of the workloads that were tested, guard deletion was not required. A guard could become empty if all its keys are deleted, but empty guards do not cause noticeable performance degradation as `get()` and range query operations skip over empty guards. Nevertheless, deleting guards is useful in two scenarios: when the guard is empty or when data in the level is spread unevenly among guards. In the second case, consolidating data among fewer guards can improve performance.

Guard deletion is also performed asynchronously similar to guard insertion. Deleted guards are added to an in-memory set. At the next compaction cycle, sstables are re-arranged to account for the deleted guards. Deleting a guard G at level i is done lazily at compaction time. During compaction, guard G is deleted and sstables belonging to guard G will be partitioned and appended to either the neighboring guards in the same level i or child guards in level $i + 1$. Compaction from level i to $i + 1$ proceeds as normal (since G is still a guard in level i + 1). At the end of compaction, FLSM persists metadata indicating G has been deleted at level i. If required, the guard is deleted in other levels in a similar manner. Note that if a guard is deleted at level i, it should be deleted at all levels $< i$; FLSM can choose whether to delete the guard at higher levels $> i$.

2.4 FLSM Operations

2.4.1 Get Operations

A `get()` operation first checks the in-memory memtable. If the key is not found, the search continues level by level, starting with level 0. During the search, if the key is found, it is returned immediately. To check if a key is present in a given level, binary search is used to find the single guard that could contain the key. Once the guard is located, its sstables are searched for the key. Thus, in the worst case, a get() requires reading one guard from each level, and all the sstables of each guard.

2.4.2 Range Queries

FLSM first identifies the guards at each level that intersect with the given range. Inside each guard, there may be multiple sstables that intersect with the given range; a binary search is performed on each sstable to identify the smallest key overall in the range. Identifying the next smallest key in the range is similar to the merge procedure in merge sort; however, a full sort does not need to be performed. When the end of range query interval is

reached, the operation is complete, and the result is returned to the user.

2.4.3 Put Operations

A `put()` operation adds data to an in-memory memtable. When the memtable gets full, it is written as a sorted sstable to Level 0. When each level reaches a certain size, it is compacted into the next level. In contrast to compaction in LSM stores, FLSM avoids sstable rewrites in most cases by partitioning sstables and attaching them to guards in the next level.

2.4.4 Key Updates and Deletions

Similar to LSM, updating or deleting a key involves inserting the key into the store with an updated sequence number or a deletion flag respectively. Reads and range queries will ignore keys with deletion flags. If the insertion of a key resulted in a guard being formed, the deletion of the key does not result in deletion of the related guard; deleting a guard will involve a significant amount of compaction work. Thus, empty guards are possible.

2.4.5 Compaction

When a guard accumulates a threshold number of sstables, it is compacted into the next level. The sstables in the guard are first (merge) sorted and then partitioned into new sstables based on the guards of the next level; the new sstables are then attached to the correct guards.

Note that in most cases, FLSM compaction does not rewrite sstables. This is the main insight behind how FLSM reduces write amplification. New sstables are simply added to the correct guard in the next level. There are two exceptions to the no-rewrite rule.

1. At the highest level (e.g,. Level 5) of FLSM, the sstables have to be rewritten during compaction; there is no higher level for the sstables to be partitioned and attached to.

2. For the second-highest level (e.g,. Level 4), FLSM will rewrite an sstable into the same level if the alternative is to merge into a large sstable in the highest level (since we cannot attach new sstables in the last level if the guard is full). The exact heuristic is rewrite in second-highest-level if merge causes 25× more IO.

FLSM compaction is trivially parallelizable because compacting a guard only involves its descendants in the next level; the way guards are chosen

in FLSM guarantees that compacting one guard never interferes with compacting another guard in the same level.

2.5 Tuning FLSM

`max_sstables_per_guard`

2.6 Limitations

2.7 Asymptotic Analysis

2.7.1 Model

We use the standard Disk Access Model (DAM) and assume that each read-/write operation can access a block of size B in one unit cost. To simplify the model, we will assume a total of n data items are stored.

2.7.2 FLSM Analysis

Consider a FLSM where the guard probability is $1/B$ (so the number of guards in level $i+1$ is in expectation B times more than the number of guards in level i). Since the expected fan-out of FLSM is B, with high probability, an FLSM with n data items will have $H = \log_B n$ levels. It is easy to see that each data item is written just once per level (it is appended once and never re-written to the same level), resulting in a write cost of O (H) = O (logB n). Since in the DAM model, FLSM writes a block of B items at unit cost, the total amortized cost of any put operation is O (H /B) = O ((logB n)/B) over its entire compaction lifetime. However, FLSM compaction in the last level does re-write data. Since this last level re-write will occur with high probability O (B) times then the final total amortized cost of any put operation is O ((B + logB n)/B). The guards in FLSM induce a degree B Skip List. A detailed theoretical analysis of the B-Skip List data structure shows that with high probability each guard will have O (B) children, each guard will have at most O (B) sstables, and each sstable

3 Building PebblesDB Over FLSM

3.1 Improving Read Performance

bloom filter...

3.2 Improving Range Query Performance

3.2.1 Seek-Based Compaction

Compaction triggered by a threshold number of consecutive `seek()` operations

3.2.2 Parallel Seeks

4 Evaluation

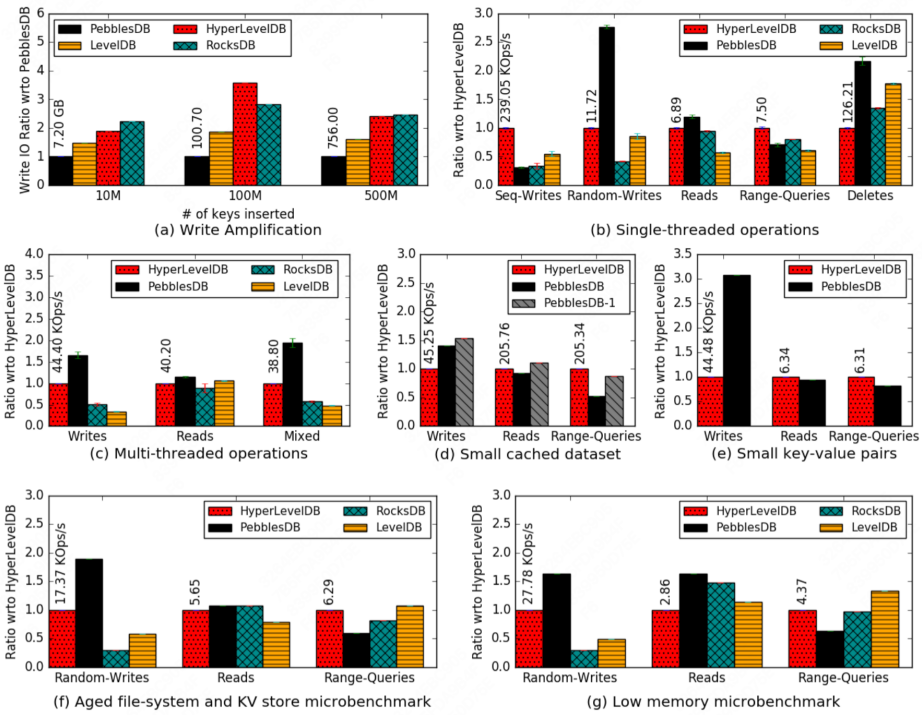

Figure 4: Micro-benchmarks. The figure compares the throughput of several key-value stores on various micro-benchmarks. Values are shown relative to HyperLevelDB, and the absolute value (in KOps/s or GB) of the baseline is shown above the bar. For (a), lower is better. In all other graphs, higher is better. PebblesDb excels in random writes, achieving 2.7× better throughput, while performing 2.5× lower IO.

5 Problems

6 References