LSM-Based Storage Techniques - A Survey

wu

April 16, 2024

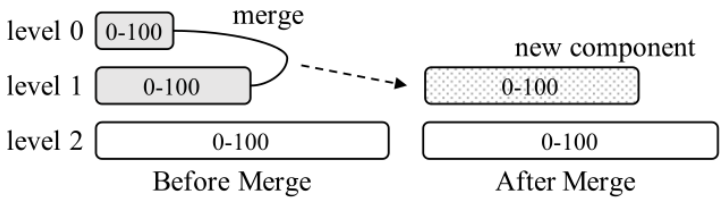
https://doi.org/10.1007/s00778-019-00555-y LSM-tree: bigtable, dynamo, hbase, cassandra, leveldb, rocksdb, asterixdb

1 LSM-tree basics

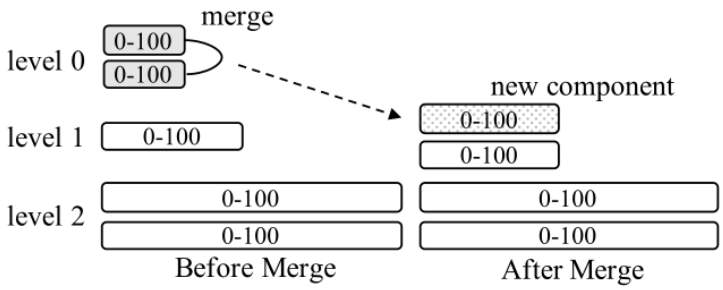
1.1 Basic Structure

- Memory component: concurrent data structure, skip-list/B^+-tree
- Disk component, SSTables: a data block stores key-value pairs ordered by keys, and the index blocks stores the key ranges of all data blocks

Two types of merge policies:



(a) Leveling Merge Policy: one component per level



(b) Tiering Merge Policy: up to T components per level

Figure 1: LSM-tree merge policies

- **Leveling merge policy**: each level only maintains one component, but the component at level L is T times larger than the component at level $L-1$. As a result, the component at level L will be merged multiple times with incoming component at level $L-1$ until it fills up, and it will then be merged into level $L+1$, and it will then be merged into level $L+1$.

  Better query performance.

- **Tiering merge policy**: maintains up to T components per level. When level L is full, its T components are merged together into a new component at level $L+1$.

  Better write performance.

1.2 Well-Known Optimizations

Bloom filter:

- built on top of disk components.
- built for each leaf page for a disk component: a point lookup can first search the non-leaf pages of a B^+-tree to locate the leaf page, where

the non-leaf pages are assumed to be small enough to be cached, and then check the associated Bloom filter before fetching the leaf page.

The false positive rate of a Bloom filter is

$$\left(1 - e^{-kn}/m\right)^k$$

where k is the number of hash functions, n is the number of keys, and m is the total number of bits. And the optimal number of hash functions that minimizes the false positive rate is

$$k = \frac{m}{n} \ln 2$$

In practice, most systems typically use 10 bits/key as a default configuration, which gives a 1% false positive rate.

Partitioning: Partitioning is orthogonal to merge policies, both leveling and tiering can be adapted to supported partitioning. But only the partitioned leveling policy has been fully implemented.

In the partitioned leveling merge policy, pioneered by LevelDB, the disk component at each level is range partitioned into multiple fixed-size SSTables, as in figure 2

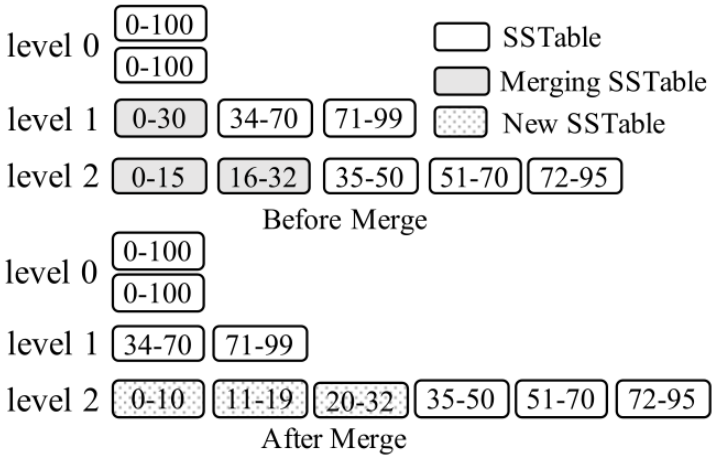


Figure 2: Partitioned leveling merge policy

Each SSTable is labeled with its key range in the figure. To merge an SSTable from level L into level $L+1$, all of its overlapping SSTable at level $L+1$ are selected, and these SSTables are merged with it to produce new SSTables

still at level $L + 1$. Different policies can be used to select which SSTable to merge next at each level.

The partitioned optimization can also be applied to the tiering merge policy. However, one major issue in doing so is that each level can contain multiple SSTables with overlapping key ranges. Two possible schemes can be used to organize the SSTables at each level

1. **Vertical grouping**: groups SSTables with overlapping key ranges together so that the groups have disjoint key ranges
2. **Horizontal grouping**: each logical disk component, which is range-partitioned into a set of SSTables, serves as a group directly

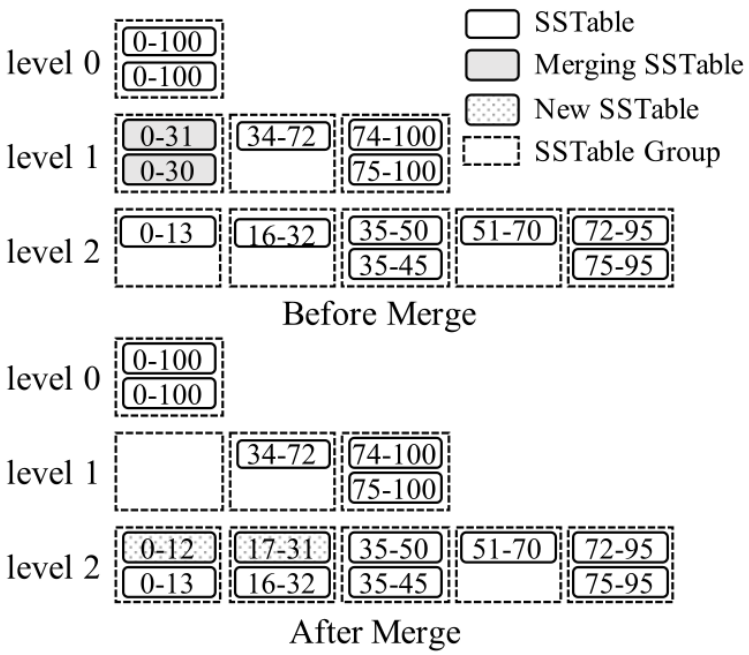


Figure 3: Partitioned tiering with vertical grouping

During a merge operation, all of the SSTables in a group are merged together to produce the resulting SSTables based on the key ranges of the overlapping groups at the next level, which are then added to these overlapping groups.

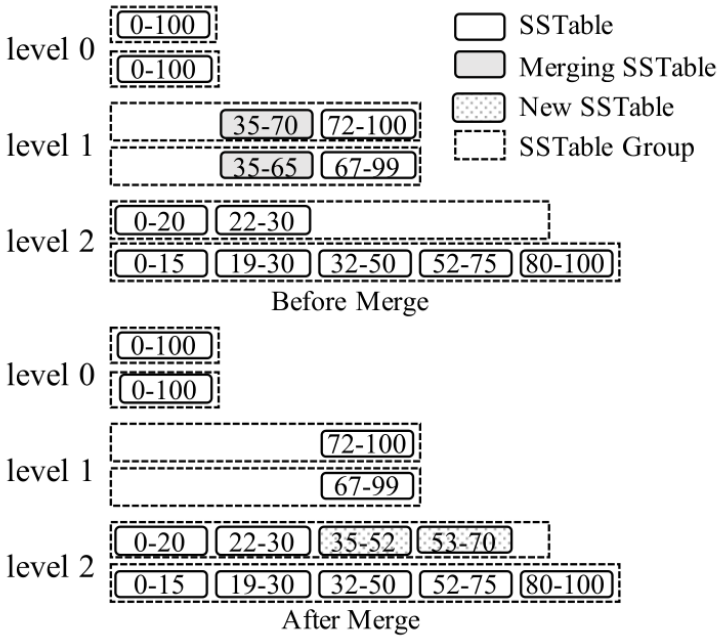


Figure 4: Partitioned tiering with horizontal grouping

Each level L further maintains an active group, which is also the first group, to receive new SSTables merged from the previous level. A merge operation selects the SSTables with overlapping key ranges from all of the groups at a level, and the resulting SSTables are added to the active group at the next level.

1.3 Concurrency Control and Recovery

Depending on the transactional isolation requirement, today's LSM-tree implementations either use a locking scheme or a multi-version scheme. A multi-version scheme works well with an LSM-tree since obsolete can be garbage-collected during merges.

Concurrent flush and merge operations, however, are unique to LSM-tree. These operations modify the metadata of an LSM-tree, e.g., the list of active components. Thus, accesses to the component metadata must be properly synchronized. To prevent a component in use from being deleted, each component can maintain a reference counter. Before accessing the components of an LSM-tree, a query can first obtain a snapshot of active components and increment their in-use counters.

Since all writes are first appended into memory, write-ahead logging (WAL) can be performed to ensure their durability. To simplify the recovery process, existing systems typically employ a **no-steal buffer management policy**: a memory component can only be flushed when all active write transactions have terminated. During recovery for an LSM-tree, the transaction log is replayed to redo all successful transactions, but no undo is needed due to the no-steal policy.

Meanwhile, the list of active disk components must also be recovered in the event of a crash.

- For unpartitioned LSM-trees, this can be accomplished by adding a pair of timestamps of the stored entries.

  This timestamp can be simply generated using local wall-clock time or a monotonic sequence number. To reconstruct the component list, the recovery process can simply find all components with disjoint timestamps. In the event that multiple components have overlapping timestamps, the component with the largest timestamp range is chosen and the rest can simply be deleted since they will have been merged to form the selected component.

- For partitioned LSM-trees, a typical approach is to maintain a separate metadata log to store all changes to the structural metadata, such as adding or deleting SSTables. The state of the LSM-tree structure can then be reconstructed by replaying the metadata log during recovery.

1.4 Cost Analysis

The cost of writes and queries is measured by counting the number of disk I/Os per operation. This analysis considers an unpartitioned LSM-tree and represents a worst-case cost.

Define

$T =$ size ratio of a given LSM-tree

$L =$ levels of the LSM-tree

$B =$ number of entries that each data page can store, page size

$P =$ number of pages of a memory component

As a result, a memory component will contain at most $B \cdot P$ entries., and level i will contain at most $T^{i+1} \cdot B \cdot P$ entries. Given N total entries, the largest level contains approximately $N \cdot \frac{T}{T+1}$. Thus the number of levels for N entries can be approximated as $L = \lceil \log_T \left(\frac{N}{B \cdot P} \cdot \frac{T}{T+1} \right) \rceil$

The write cost, which is also referred to as **write amplification** in the literature, measures the amortized I/O cost of inserting an entry into an LSM-tree. It should be noted that this cost measures the overall I/O cost for this entry to be merged into the largest level since inserting an entry into memory does not incur any disk I/O.

- For leveling, a component at each level will be merge $T-1$ times until it fills up and is pushed to the next level.
- For tiering, multiple components at each level are merged only once and are pushed to the next level directly.

Since each disk page contains B entries, the write cost for each entry will be $O(T \cdot \frac{L}{B})$ for leveling and $O(\frac{L}{B})$ for tiering.

The I/O cost of a query depends on the number of components in an LSM-tree.

- Without Bloom filters, the I/O cost of a point lookup will be $O(L)$ for leveling and $O(T \cdot L)$ for tiering.
- For a zero-result point lookup, suppose all Bloom filters have M bits in total and have the same false positive rate across all levels. With N total keys, each Bloom filter has a false positive rate of $O(e^{-\frac{M}{N}})$. Thus the I/O cost of a zero-result point lookup will be $O(L \cdot e^{-\frac{M}{N}})$ for leveling and $O(T \cdot L \cdot e^{-\frac{M}{N}})$.
- To search for an existing unique key, at least one I/O must be performed to fetch the entry. Given that in practice the Bloom filter false positive rate is much smaller than 1, the successful point lookup I/O cost for both the leveling and tiering will be $O(1)$.

The I/O cost of a range query depends on the query selectivity. Let s be the number of unique keys accessed by a range query. A range query can be considered to be **long** if $\frac{s}{B} > 2 \cdot L$, and **short** otherwise. The distinction is that the I/O cost of a long range query will be dominated by the largest level since the largest level contains most of the data. In contrast, the I/O cost of a short range query will derive equally from all levels since the query must issue one I/O to each disk component. Thus, the I/O cost of a long range query will be $O(\frac{s}{B})$ for leveling and $O(T \cdot \frac{s}{B})$ for tiering. For a short range query, the I/O cost will be $O(L)$ for leveling and $O(T \cdot L)$ for tiering.

Finally, let's examine the space amplification of an LSM-tree, which is defined as the overall number of entries divided by the number of unique entries.

- For leveling, the worst case occurs when all of the data at the first $L-1$ levels, which contain approximately $\frac{1}{T}$ of the total data, are updates to the entries at the largest level. Thus the worst case space amplification for leveling is $O(\frac{T+1}{T})$.
- For tiering, the worst case happens when all of the components at the largest level contain exactly the same of keys. As a result, the worst case space amplification is $O(T)$.

Merge Policy	Write	Point Lookup (Zero-Result/Non-zero)	Short Range Query	Long Ran
Leveling	$O(T \cdot \frac{L}{B}))$	$O(L \cdot e^{-\frac{M}{N}})/O(1)$	$O(L)$	$O(\frac{s}{B})$
Tiering	$O(\frac{L}{B})$	$O(T \cdot L \cdot e^{-\frac{M}{N}})/O(1)$	$O(T \cdot L)$	$O(T \cdot \frac{s}{B})$

2 LSM-tree Improvements

2.1 A Taxonomy of LSM-tree Improvements

- **Write Amplification**:
- **Merge Operations**: Moreover, merge operations can have negative impacts on the system, including buffer cache misses after merges and write stalls during large merges.
- **Hardware**:
- **Special Workloads**:
- **Auto-Tuning**: Based on the RUM conjecture, no access method can be read-optimal, write-optimal, and space- optimal at the same time.
- **Secondary Indexing**:

2.2 Reducing Write Amplification

2.2.1 Tiering

WriteBuffer (WB) tree:

1. hash-partitioning to achieve workload balance so that each SSTable group roughly stores the same amount of data.

2. Organizes SSTable groups into a B^+-tree-like structure to enable self-balancing to minimize the total number of levels. Specifically, each SSTable group is treated like a node in a B^+-tree. When a non-leaf node becomes full with T SSTables, these T SSTables are merged together to form a new SSTables that are added into its child nodes. When a leaf node becomes full with T SSTables, it is split into two leaf nodes by merging all of its SSTables into two leaf nodes with smaller key ranges so that each new node receives about $T/2$ SSTables.

The light-weight compaction tree presents a method to achieve workload balancing of SSTable groups.

dCompaction introduces the concept of virtual SSTables and virtual merges to reduce the merge frequency. A virtual merge operation produces a virtual SSTable that simply points to the input SSTables without performing actual merge. However, since a virtual SSTable points to multiple SSTables with overlapping ranges, query performance will degrade. To address this, dCompaction introduces a threshold based on the number of real SSTables to trigger actual merges. It also lets queries trigger actual merges if a virtual SSTable pointing too many SSTables is encountered during query processing.

2.2.2 Merge Skipping

The skip-tree proposes a merge skipping idea to improve write performance. The observation is that each entry must be merged from level 0 down to the largest level. If some entries can be directly pushed to a higher level by skipping some level-by-level merges, then the total write cost will be reduced.

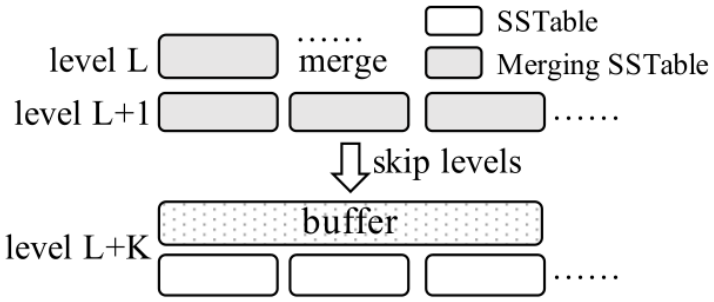


Figure 5: Merge in skip-tree

During a merge at level L, the skip-tree directly pushes some keys to a mutable buffer at level $L+K$ so that some level-by-level merges can be skipped.

Meanwhile, the skipped entries in the mutable buffer will be merged with the SSTables at level $L + K$ during subsequent merges. To ensure correctness, a key from level L can be pushed to level $L + K$ only if this key doe not appear in any of the intermediate levels $L + 1, ..., L + K - 1$. This condition can be tested efficiently by checking the Bloom filters of the intermediate levels.

2.2.3 Exploiting Data Skew

TRIAD reduces write amplification for skewed update workloads where some hot keys are updated frequently. The basic idea is to separate hot keys from cold keys in the memory component so that only cold keys are flushed to disk. Even though hot keys are not flushed to disk, they are periodically copied to a new transaction log so that the old transaction log can be reclaimed

TRIAD also reduces write amplification by delaying merges at level 0 until level 0 contains multiple SSTables.

Finally, it presents an optimization that avoids creating new disk components after flushes. Instead, the transaction log itself is used as a disk component and an index structure is built on top of it to improve lookup performance.

2.2.4 Summary

Tiering has been widely used to improve the write performance of LSM-trees, but this will decrease query performance and space utilization

2.3 Optimizing Merge Operations

2.3.1 Improving Merge Performance

The VT-tree presents a stitching operation to improve merge performance. The basic idea is that when merging multiple SSTables, if the key range of a page from an input SSTable does not overlap the key ranges of any pages from other SSTables, then this page can be simply pointed to by the resulting SSTable without reading and copying.

But is has a number of drawbacks:

1. cause fragmentation since pages are no longer continuously stored: introduce a stitching threshold K so that a stitching operation is triggered only when there are at least K continuous pages from an input SSTable

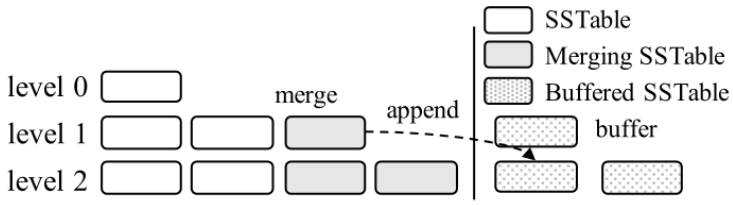
2. Since the keys in stitched pages are not scanned during a merge operation, a Bloom filter cannot be reproduced: use quotient filters since multiple quotient filters can be combined directly without accessing the original keys.

Or we could slightly merge the phases of merge[**?**] .

2.3.2 Reducing Buffer Cache Misses

Merge operations can interfere with the caching behavior of a system. After a new component is enabled, queries may experience a large number of buffer cache misses since the new component has not been cached yet.

The Log-Structured buffered Merge tree:



After an SSTable at level L is merged into level $L+1$, the old SSTables at level L is appended to a buffer associated with level $L+1$ instead of being deleted immediately. The buffered SSTables are searched by queries as well to minimize buffer cache misses, and they are deleted gradually based on their access frequency. This approach is mainly effective for skewed workloads where only a small range of keys are frequently accessed.

2.3.3 Minimizing Write Stalls

bLSM proposes a spring-and-gear merge scheduler to minimize write stalls for the unpartitioned leveling merge policy. Its basic idea is to tolerate an extra component at each level so that merges at different levels can proceed in parallel. Furthermore, the merge scheduler controls the progress of merge operations to ensure that level L produces a new component at level $L+1$ only after the previous merge operation at level $L+1$ has completed.

bLSM was only designed for the unpartitioned leveling merge policy. Moreover, it only bounds the maximum latency of writing to memory components while the queuing latency, which is often a major source of performance variability, is ignored.

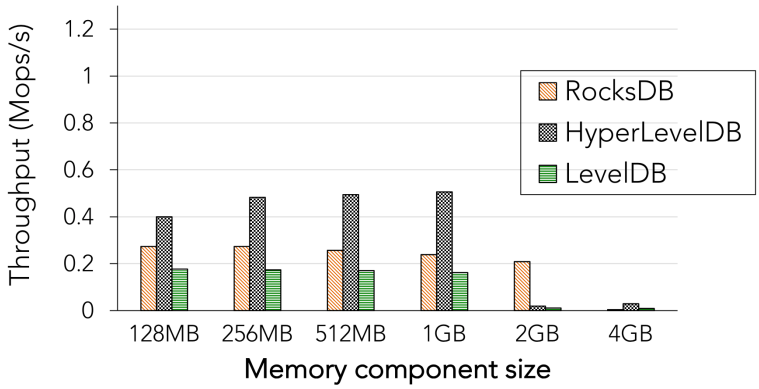
2.3.4 Summary

2.4 Hardware Opportunities

2.4.1 Large Memory

- If a memory component is implemented directly using on-heap data structures, large memory can result in a large number of small objects that lead to significant GC overheads.
- If a memory component is implemented using off-heap structures such as a concurrent B^+-tree, large memory can still cause a higher search cost (due to tree height) and cause more CPU cache misses for writes, as a write must first search for its position in the structure.

Memory component scales badly. Image from FloDB's presentation.



16 threads; write-only workload.

FloDB presents a two-layer design to manage large memory components.

- The top level is a small concurrent hash table to support fast writes, and the bottom level is a large skip-list to support range queries efficiently.
- When the hash table is full, its entries are efficiently migrated into the skip-list using a batched algorithm.

By limiting random writes to a small memory area, this design significantly improves the in-memory write throughput. To support range queries, FloDB requires that a range query must wait for the hash table to be drained so that the skip-list alone can be searched to answer the query.

Problems:

1. Not efficient for workloads containing both writes and range queries.
2. The skip-list may have a large memory footprint and lead to lower memory utilization.

To address the drawbacks of FloDB, Accordion uses a multi-layer approach to manage its large memory components. In this design,

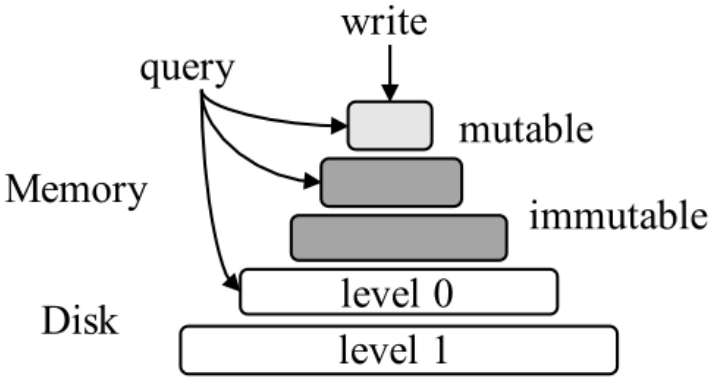


Figure 6: Accordion's multi-layer structure

there is a small mutable memory component in the top level to process writes. When the mutable component is full, instead of being flushed to disk, it is simply flushed into a immutable memory component via an in-memory flush operation. Similarly, such immutable memory components can be merged via in-memory merge operations to improve query performance and reclaim space occupied by obsolete entries.

2.4.2 Multi-Core

3 References

References

[ZYH+14] Zigang Zhang, Yinliang Yue, Bingsheng He, Jin Xiong, Mingyu Chen, Lixin Zhang, and Ninghui Sun. Pipelined compaction for the lsm-tree. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 777–786, 2014.