typically employ a no-steal buffer management policy [36]. That is, 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 to each disk component that indicate the range 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, this timestamp-based approach does not work anymore since each component is further range-partitioned. To address this, a typical approach, used in LevelDB [4] and RocksDB [6], 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.

2.3 Cost Analysis

To help understand the performance trade-offs of LSM-trees, we can turn to the cost analysis of writes, point lookups, range queries, and space amplification presented in [24,25]. 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.

Let the size ratio of a given LSM-tree be T, and suppose the LSM-tree contains L levels. In practice, for a stable LSM-tree where the volume of inserts equals the volume of deletes, L remains static. Let B denote the page size, that is, the number of entries that each data page can store, and let P denote the number of pages of a memory component. As a result, a memory component will contain at most $B \cdot P$ entries, and level i ($i \geq 0$)will contain at most $T^{i+1} \cdot B \cdot P$ entries. Given N total entries, the largest level contains approximately $N \cdot \frac{T}{T+1}$ entries since it is T times larger than the previous level. Thus, the number of levels for N entries can be approximated as $L = \lceil \log_T (\frac{N}{B \cdot P} \cdot \frac{T}{T+1}) \rceil$.

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 merged $T-1$ 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. However, Bloom filters can greatly improve the point lookup cost. For a zero-result point lookup, i.e., for a search for a non-existent key, all disk I/Os are caused by Bloom filter false positives. 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 $O(e^{-\frac{M}{N}})$ [18]. Thus, the I/O cost of a zero-result point lookup will be $O(L \cdot e^{-\frac{M}{N}})$ for leveling and $O(T \cdot L \cdot e^{-\frac{M}{N}})$ for tiering. 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 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 $\frac{s}{B} > 2 \cdot L$, otherwise it is *short* [24,25]. 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 (almost) 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 us examine the space amplification of an LSM-tree, which is defined as the overall number of entries divided by the number of unique entries[5]. 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 set of keys. As a result, the worst case space amplification for tiering will be $O(T)$. In practice, the space amplification is an important factor to consider when deploying storage systems [28], as it directly impacts the storage cost for a given workload.

The cost complexity of LSM-trees is summarized in Table 1. Note how the size ratio T impacts the performance of leveling and tiering differently. In general, leveling is opti-

[5] The original analysis presented in [24,25] defines the space amplification to be the overall number of obsolete entries divided by the number of unique entries. We slightly modified the definition to ensure that the space amplification is no less than 1.

Table 1: Summary of Cost Complexity of LSM-trees

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

mized for query performance and space utilization by maintaining one component per level. However, components must be merged more frequently, which will incur a higher write cost by a factor of T. In contrast, tiering is optimized for write performance by maintaining up to T components at each level. This, however, will decrease query performance and worsen space utilization by a factor of T. As one can see, the LSM-tree is highly tunable. For example, by changing the merge policy from leveling to tiering, one can greatly improve write performance with only a small negative impact on point lookup queries due to the Bloom filters. However, range queries and space utilization will be significantly impacted. As we proceed to examine the recent literature on improving LSM-trees, we will see that each makes certain performance trade-offs. Actually, based on the RUM conjecture [14], each access method has to make certain trade-offs among the read cost (R), update cost (U), and memory or storage cost (M). It will be important for the reader to keep in mind the cost complexity described here to better understand the trade-offs made by the proposed improvements.

3 LSM-tree Improvements

In this section we present a taxonomy for use in classifying the existing research efforts on improving LSM-trees. We then provide an in-depth survey of the LSM-tree literature that follows the structure of the proposed taxonomy.

3.1 A Taxonomy of LSM-tree Improvements

Despite the popularity of LSM-trees in modern NoSQL systems, the basic LSM-tree design suffers from various drawbacks and insufficiencies. We now identify the major issues of the basic LSM-tree design, and further present a taxonomy of LSM-tree improvements based on these drawbacks.

Write Amplification. Even though LSM-trees can provide much better write throughput than in-place update structures such as B^+-trees by reducing random I/Os, the leveling merge policy, which has been adopted by modern key-value stores such as LevelDB [4] and RocksDB [6], still incurs relatively high write amplification. High write amplification not only limits the write performance of an LSM-tree but also reduces the lifespan of SSDs due to frequent disk writes. A large body of research has been conducted to reduce the write amplification of LSM-trees.

Merge Operations. Merge operations are critical to the performance of LSM-trees and must therefore be carefully implemented. Moreover, merge operations can have negative impacts on the system, including buffer cache misses after merges and write stalls during large merges. Several improvements have been proposed to optimize merge operations to address these problems.

Hardware. In order to maximize performance, LSM-trees must be carefully implemented to fully utilize the underling hardware platforms. The original LSM-tree has been designed for hard disks, with the goal being reducing random I/Os. In recent years, new hardware platforms have presented new opportunities for database systems to achieve better performance. A significant body of recent research has been devoted to improving LSM-trees to fully exploit the underling hardware platforms, including large memory, multi-core, SSD/NVM, and native storage.

Special Workloads. In addition to hardware opportunities, certain special workloads can also be considered to achieve better performance in those use cases. In this case, the basic LSM-tree implementation must be adapted and customized to exploit the unique characteristics exhibited by these special workloads.

Auto-Tuning. Based on the RUM conjecture [14], no access method can be read-optimal, write-optimal, and space-optimal at the same time. The tunability of LSM-trees is a promising solution to achieve optimal trade-offs for a given workload. However, LSM-trees can be hard to tune because of too many tuning knobs, such as memory allocation, merge policy, size ratio, etc. To address this issue, several auto-tuning techniques have been proposed in the literature.

Secondary Indexing. A given LSM-tree only provides a simple key-value interface. To support the efficient processing of queries on non-key attributes, secondary indexes must be maintained. One issue in this area is how to maintain a set of related secondary indexes efficiently with a small overhead on write performance. Various LSM-based secondary indexing structures and techniques have been designed and evaluated as well.

Based on these major issues of the basic LSM-tree design, we present a taxonomy of LSM-tree improvements, shown in Figure 7, to highlight the specific aspects that the existing research efforts try to optimize. Given this taxon-

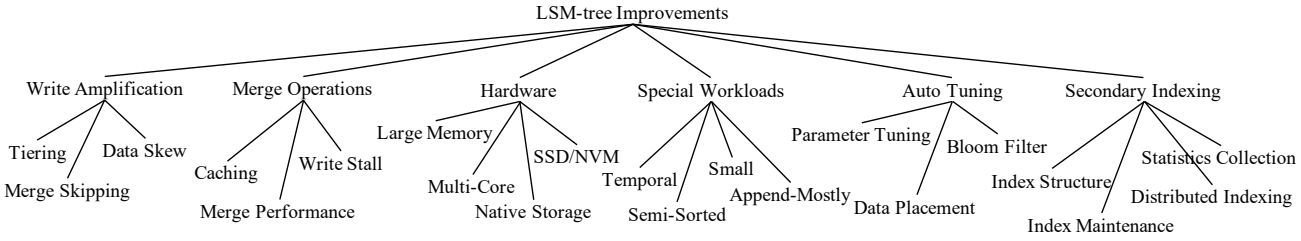

Fig. 7: Taxonomy of LSM-tree improvements

omy, Table 2 further classifies the LSM-tree improvements in terms of each improvement's primary and secondary concerns. With this taxonomy and classification in hand, we now proceed to examine each improvement in more depth.

3.2 Reducing Write Amplification

In this section, we review the improvements in the literature that aim to reduce the write amplification of LSM-trees. Most of these improvements are based on tiering since it has much better write performance than leveling. Other proposed improvements have developed new techniques to perform merge skipping or to exploit data skews.

3.2.1 Tiering

One way to optimize write amplification is to apply tiering since it has much lower write amplification than leveling. However, recall from Section 2.3 that this will lead to worse query performance and space utilization. The improvements in this category can all be viewed as some variants of the partitioned tiering design with vertical or horizontal grouping discussed in Section 2.2.2. Here we will mainly discuss the modifications made by these improvements.

The WriteBuffer (WB) Tree [12] can be viewed as a variant of the partitioned tiering design with vertical grouping. It has made the following modifications. First, it relies on hash-partitioning to achieve workload balance so that each SSTable group roughly stores the same amount of data. Furthermore, it 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 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 (LWC-tree) [78, 79] adopts a similar partitioned tiering design with vertical grouping. It further presents a method to achieve workload balancing of SSTable groups. Recall that under the vertical grouping scheme, SSTables are no longer strictly fixed-size since they are produced based on the key ranges of the overlapping groups at the next level instead of based on their sizes. In the LWC-tree, if a group contains too many entries, it will shrink the key range of this group after the group has been merged (now temporarily empty) and will widen the key ranges of its sibling groups accordingly.

PebblesDB [58] also adopts a partitioned tiering design with vertical grouping. The major difference is that it determines the key ranges of SSTable groups using the idea of guards as inspired by the skip-list [55]. Guards, which are the key ranges of SSTable groups, are selected probabilistically based on inserted keys to achieve workload balance. Once a guard is selected, it is applied lazily during the next merge. PebblesDB further performs parallel seeks of SSTables to improve range query performance.

dCompaction [53] 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 merges. 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 to too many SSTables is encountered during query processing. In general, dCompaction delays a merge operation until multiple SSTables can be merged together, and thus it can also be viewed as a variant of the tiering merge policy.

As one can see, the four structures described above all share a similar high-level design based on partitioned tiering with vertical grouping. They mainly differ in how workload balancing of SSTable groups is performed. For example, the WB-tree [12] relies on hashing, but doing so gives up the ability of supporting range queries. The LWC-tree [78, 79] dynamically shrinks the key ranges of dense SSTable groups, while PebblesDB [58] relies on probabilistically selected guards. In contrast, dCompaction [53] offers no built-in support for workload balancing. It is not clear how skewed SSTable groups would impact the performance of these struc-

Table 2: Classification of existing LSM-tree improvements (♣ denotes primary category, △ denotes secondary categories)

Publication	Write Amplification	Merge Operations	Hardware	Special Workloads	Auto Tuning	Secondary Indexing
WB-tree [12]	♣					
LWC-tree [78,79]	♣					
PebblesDB [58]	♣					
dCompaction [53]	♣				△	
Zhang et al. [82]	♣					
SifrDB [50]	♣	△	△			
Skip-tree [81]	♣					
TRIAD [16]	♣					
VT-tree [64]		♣				
Zhang et al. [84]		♣				
Ahmad et al. [8]		♣				
LSbM-tree [68,69]		♣				
bLSM [61]		♣				
FloDB [15]	△		♣			
Accordion [19]	△		♣			
cLSM [34]			♣			
FD-tree [44]			♣			
FD+tree [71]		△	♣			
MaSM [13]	△		♣	△		
WiscKey [46]	△		♣			
HashKV [20]	△		♣			
Kreon [54]	△		♣			
NoveLSM [39]			♣			
LDS [49]			♣			
LOCS [74]			♣			
NoFTL-KV [73]			♣			
LHAM [51]				♣		
LSM-trie [76]	△			♣		
SlimDB [59]	△			♣		
Mathieu et al. [48]	△			♣		
Lim et al. [45]					♣	
Monkey [25,26]					♣	
Dostoevsky [24]	△				♣	
Thonangi and Yang [70]	△				♣	
ElasticBF [83]					♣	
Mutant [80]					♣	
LSII [75]						♣
Kim et al. [42]						♣
Filter [11]						♣
Qader et al. [57]						♣
Diff-Index [66]						♣
DELI [67],						♣
Luo and Carey [47]						♣
Ildar et al. [7]						♣
Joseph et al. [29]						♣
Zhu et al. [85]						♣
Duan et al. [30]						♣

tures, and future research is needed to understand this problem and evaluate these workload balancing strategies.

The partitioned tiering design with horizontal grouping has been adopted by Zhang et al. [82] and SifrDB [50]. SifrDB also proposes an early-cleaning technique to reduce disk space utilization during merges. During a merge operation, SifrDB incrementally activates newly produced SSTables and deactivates the old SSTables. SifrDB further exploits I/O parallelism to speedup query performance by examining multiple SSTables in parallel.

3.2.2 Merge Skipping

The skip-tree [81] 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

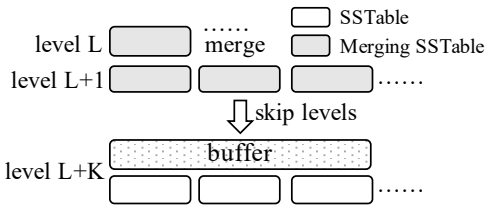

Fig. 8: Merge in skip-tree: entries from a lower level can be directly pushed to the mutable buffer of a higher level

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. As shown in Figure 8, 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 does not appear in any of the intermediate levels $L+1,\cdots,L+K-1$. This condition can be tested efficiently by checking the Bloom filters of the intermediate levels. The skip-tree further performs write-ahead logging to ensure durability of the entries stored in the mutable buffer. To reduce the logging overhead, the skip-tree only logs the key plus the ID of the original SSTable and prevents an SSTable from being deleted if it is referenced by any key in the buffer. Although merge skipping is an interesting idea to reduce write amplification, it introduces non-trivial implementation complexity to manage the mutable buffers. Moreover, since merge skipping essentially reduces some merges at the intermediate levels, it is not clear how the skip-tree would compare against a well-tuned LSM-tree by reducing the size ratio.

3.2.3 Exploiting Data Skew

TRIAD [16] 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. As a result, when hot keys are updated, old versions can be discarded directly without writing them 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. However, range query performance will still be negatively impacted since entries are not sorted in the log.

3.2.4 Summary

Tiering has been widely used to improve the write performance of LSM-trees, but this will decrease query performance and space utilization, as discussed in Section 2.3. The existing tiering-based improvements mainly differ in how SSTables are managed, either by vertical grouping [12, 53,58,78,79] or horizontal grouping [50,82]. It is not clear how these different grouping schemes impact system performance and it would be useful as future work to study and evaluate their impact. The skip-tree [81] and TRIAD [16] propose several new ideas to improve write performance, ideas that are orthogonal to tiering. However, these optimizations bring non-trivial implementation complexity to real systems, such as the mutable buffers introduced by the skip-tree and the use of transaction logs as flushed components by TRIAD.

All of the improvements in this category, as well as some improvements in the later sections, have claimed that they can greatly improve the write performance of LSM-trees, but their performance evaluations have often failed to consider the tunability of LSM-trees. That is, these improvements have mainly been evaluated against a default (untuned) configuration of LevelDB or RocksDB, which use the leveling merge policy with a size ratio of 10. It is not clear how these improvements would compare against well-tuned LSM-trees. To address this, one possible solution would be to tune RocksDB to achieve a similar write throughput to the proposed improvements by changing the size ratio or by adopting the tiering merge policy and then evaluating how these improvements can improve query performance and space amplification. Moreover, these improvements have primarily focused on query performance; space amplification has often been neglected. It would be a useful experimental study to fully evaluate these improvements against well-tuned baseline LSM-trees to evaluate their actual usefulness. We also hope that this situation can be avoided in future research by considering the tunability of LSM-trees when evaluating the proposed improvements.

3.3 Optimizing Merge Operations

Next we review some existing work that improves the implementation of merge operations, including improving merge performance, minimizing buffer cache misses, and eliminating write stalls.

3.3.1 Improving Merge Performance

The VT-tree [64] 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 it again. Even though stitching improves merge performance for certain workloads, it has a number of drawbacks. First, it can cause fragmentation since pages are no longer continuously stored on disk. To alleviate this problem, the VT-tree introduces a stitching threshold K so that a stitching operation is triggered only when there are at least K continuous pages from an input SSTable. Moreover, since the keys in stitched pages are not scanned during a merge operation, a Bloom filter cannot be produced. To address this issue, the VT-tree uses quotient filters [17] since multiple quotient filters can be combined directly without accessing the original keys.

Zhang et al. [84] proposed a pipelined merge implementation to better utilize CPU and I/O parallelism to improve merge performance. The key observation is that a merge operation contains multiple phases, including the read phase, merge-sort phase, and write phase. The read phase reads pages from input SSTables, which will then be merge-sorted to produce new pages during the merge-sort phase. Finally, the new pages will be written to disk during the write phase. Thus, the read phase and write phase are I/O heavy while the merge-sort phase is CPU heavy. To better utilize CPU and I/O parallelism, the proposed approach pipelines the execution of these three phases, as illustrated by Figure 9. In this example, after the first input page has been read, this approach continues reading the second input page (using disk) and the first page can be merge-sorted (using CPU).

3.3.2 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. A simple write-through cache maintenance policy cannot solve this problem. If all of the pages of the new component were cached during a merge operation, a lot of other working pages would be evicted, which will again cause buffer cache misses.

Ahmad et al. [8] conducted an experimental study of the impact of merge operations on system performance. They found that merge operations consume a large number of CPU and I/O resources and impose a high overhead on query

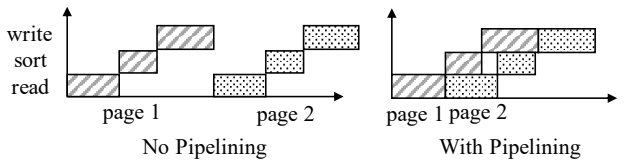

Fig. 9: Pipelined merge example: multiple input pages can be processed in a pipelined fashion

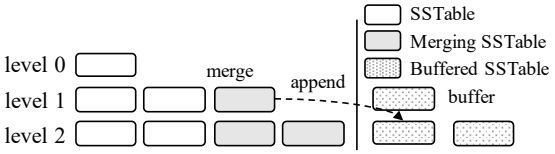

Fig. 10: LSbM-tree: the merged component is added to a buffer of the next level instead of being deleted immediately

response times. To address this, this work proposed to offload large merges to remote servers to minimize their impact. After a merge operation is completed, a smart cache warmup algorithm is used to fetch the new component incrementally to minimize buffer cache misses. The idea is to switch to the new component incrementally, chunk by chunk, to smoothly redirect incoming queries from the old components to the new component. As a result, the burst of buffer cache misses is decomposed into a large number of smaller ones, minimizing the negative impact of component switching on query performance.

One limitation of the approach proposed by Ahmad et al. [8] is that merge operations must be offloaded to separate servers. The incremental warmup algorithm alone was subsequently found to be insufficient due to contention between the newly produced pages and the existing hot pages [68, 69]. To address this limitation, the Log-Structured buffered Merge tree (LSbM-tree) [68, 69] proposes an alternative approach. As illustrated by Figure 10, 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. Note that there is no need to add old SSTables at level $L+1$ into the buffer, as the SSTables at $L+1$ all come from level L and the entries of these old SSTables will have already been added to the buffer before. 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 does not incur any extra disk I/O during a merge operation since it only delays the deletion of the old SSTables. However, this approach is mainly effective for skewed workloads where only a small range of keys are frequently accessed. It can introduce extra overhead for queries accessing cold data that are not cached, especially for range queries since they cannot benefit from Bloom filters.

3.3.3 Minimizing Write Stalls

Although the LSM-tree offers a much higher write throughput compared to traditional B^+-trees, it often exhibits write stalls and unpredictable write latencies since heavy operations such as flushes and merges run in the background. bLSM [61] 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. This eventually cascades to limit the maximum write speed at the memory component and eliminates large write stalls. However, bLSM itself has several limitations. 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.

3.3.4 Summary

The improvements in this category optimize the implementation of merge operations in terms of performance, buffer cache misses, and write stalls. To speedup merge operations, the VT-tree [64] introduces the stitching operation that avoids copying input pages if applicable. However, this may cause fragmentation, which is undesirable for hard disks. Moreover, this optimization is incompatible with Bloom filters, which are widely used in modern LSM-tree implementations. The pipelined merge implementation [84] improves merge performance by exploiting CPU and I/O parallelism. It should be noted that many LSM-based storage systems have already implemented some form of pipelining by exploiting disk read-ahead and write-behind.

Ahmed et al. [8] and the LSbM-tree [68,69] present two alternative methods to alleviating buffer cache misses caused by merges. However, both approaches appear to have certain limitations. The approach proposed by Ahmed et al. [8] requires dedicated servers to perform merges, while the LSbM-tree [68,69] delays the deletion of old components that could negatively impact queries accessing cold data. Write stalls are a unique problem of LSM-trees due to its out-of-place update nature. bLSM [61] is the only effort that attempts to address this problem. However, bLSM [61] only bounds the maximum latency of writing to memory components. The end-to-end write latency can still exhibit large variances due to queuing. More efforts need to be done to improve the performance stability of LSM-trees.

3.4 Hardware Opportunities

We now review the LSM-tree improvements proposed for different hardware platforms, including large memory, multi-core, SSD/NVM, and native storage. A general paradigm of these improvements is to modify the basic design of LSM-trees to fully exploit the unique features provided by the target hardware platform to achieve better performance.

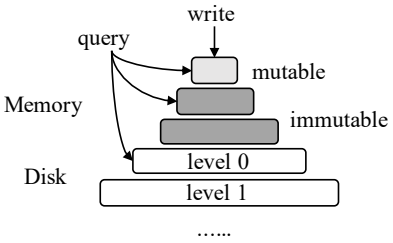

Fig. 11: Accordion's multi-layer structure

3.4.1 Large Memory

It is beneficial for LSM-trees to have large memory components to reduce the total number of levels, as this will improve both write performance and query performance. However, managing large memory components brings several new challenges. 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. In contrast, 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.

FloDB [15] 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. However, FloDB suffers from two major problems. First, it is not efficient for workloads containing both writes and range queries due to their contention. Second, the skip-list may have a large memory footprint and lead to lower memory utilization.

To address the drawbacks of FloDB, Accordion [19] uses a multi-layer approach to manage its large memory components. In this design (Figure 11), there is a small mutable memory component in the top level to process writes. When the mutable memory component is full, instead of being flushed to disk, it is simply flushed into a (more compact) 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. Note that in-memory flush and merge operations do not involve any disk I/O, which reduces the overall disk I/O cost by leveraging large memory.

3.4.2 Multi-Core

cLSM [34] optimizes for multi-core machines and presents new concurrency control algorithms for various LSM-tree operations. It organizes LSM components into a concurrent linked list to minimize blocking caused by synchronization. Flush and merge operations are carefully designed so that they only result in atomic modifications to the linked list that will never block queries. When a memory component becomes full, a new memory component is allocated while the old one will be flushed. To avoid writers inserting into the old memory component, a writer acquires a shared lock before modifications and the flush thread acquires an exclusive lock before flushes. cLSM also supports snapshot scans via multi-versioning and atomic read-modify-write operations using an optimistic concurrency control approach that exploits the fact that all writes, and thus all conflicts, involve the memory component.

3.4.3 SSD/NVM

Different from traditional hard disks, which only support efficient sequential I/Os, new storage devices such as solid-state drives (SSDs) and non-volatile memories (NVMs) support efficient random I/Os as well. NVMs further provide efficient byte-addressable random accesses with persistence guarantees.

The FD-tree [44] uses a similar design to LSM-trees to reduce random writes on SSDs. One major difference is that the FD-tree exploits fractional cascading [22] to improve query performance instead of Bloom filters. For the component at each level, the FD-tree additionally stores fence pointers that point to each page at the next level. For example in Figure 12, the pages at level 2 are pointed at by fence pointers with keys 1, 27, 51, 81 at level 1. After performing a binary search at level 0, a query can follow these fence pointers to traverse all of the levels. However, this design introduces additional complexity to merges. When the component at level L is merged into level $L+1$, all of the previous levels 0 to $L-1$ must be merged as well to rebuild the fence pointers. Moreover, a point lookup still needs to perform disk I/Os when searching for non-existent keys, which can be mostly avoided by using Bloom filters. For these reasons, modern LSM-tree implementations prefer Bloom filters rather than fractional cascading[6].

The FD+tree [71] improves the merge process of the FD-tree [44]. In the FD-tree, when a merge happens from level 0 to level L, new components at levels 0 to L must be created, which will temporarily double the disk space. To address this, during a merge operation, the FD+tree incrementally activates the new components and reclaims pages from the old components that are not used by any active queries.

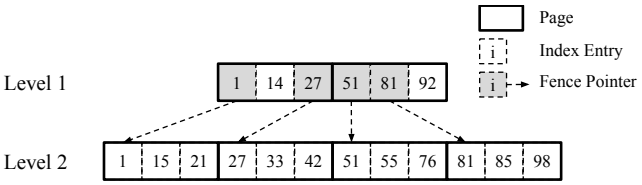

Fig. 12: Example FD-tree structure

MaSM (materialized sort-merge) [13] is designed for supporting efficient updates for data warehousing workloads by exploiting SSDs. MaSM first buffers all updates into an SSD. It uses the tiering merge policy to merge intermediate components with low write amplification. The updates are then merged back to the base data, which resides in the hard disk. MaSM can be viewed as a simplified form of the lazy leveling merge policy proposed by Dostoevsky [24], as we will see later in this survey. Moreover, since MaSM mainly targets long range queries to support data warehousing workloads, the overhead introduced by intermediate components stored in SSDs is negligible compared to the cost of accessing the base data. This enables MaSM to only incur a small overhead on queries with concurrent updates.

Since SSDs support efficient random reads, separating values from keys becomes a viable solution to improve the write performance of LSM-trees. This approach was first implemented by WiscKey [46] and subsequently adopted by HashKV [20] and SifrDB [50]. As shown in Figure 13, WiscKey [46] stores key-value pairs into an append-only log and the LSM-tree simply serves as a primary index that maps each key to its location in the log. While this can greatly reduce the write cost by only merging keys, range query performance will be significantly impacted because values are not sorted anymore. Moreover, the value log must be garbage-collected efficiently to reclaim the storage space. In WiscKey, garbage-collection is performed in three steps. First, WiscKey scans the log tail and validates each entry by performing point lookups against the LSM-tree to find out whether the location of each key has changed or not. Second, valid entries, whose locations have not changed, are then appended to the log and their locations are updated in the LSM-tree as well. Finally, the log tail is truncated to reclaim the storage space. However, this garbage-collection process has been shown to be a new performance bottleneck [20] due to its expensive random point lookups.

HashKV [20] introduces a more efficient approach to garbage-collect obsolete values. The basic idea is to hash-partition the value log into multiple partitions based on keys and to garbage-collect each partition independently. In order to garbage-collect a partition, HashKV performs a group-by

[6] RocksDB [6] supports a limited form of fractional cascading by maintaining the set of overlapping SSTables at the adjacent next level for each SSTable. These pointers are used to narrow down the search range when locating specific SSTables during point lookups.

operation on the keys to find the latest value for each key. Valid key-value pairs are added to a new log and their locations are then updated in the LSM-tree. HashKV further stores cold entries separately so that they can be garbage-collected less frequently.

Kreon [54] exploits memory-mapped I/O to reduce CPU overhead by avoiding unnecessary data copying. It implements a customized memory-mapped I/O manager in the Linux kernel to control cache replacement and to enable blind writes. To improve range query performance, Kreon reorganizes data during query processing by storing the accessed key-value pairs together in a new place.

NoveLSM [39] is an implementation of LSM-trees on NVMs. NoveLSM adds an NVM-based memory component to serve writes when the DRAM memory component is full so that writes can still proceed without being stalled. It further optimizes the write performance of the NVM memory component by skipping logging since NVM itself provides persistence. Finally, it exploits I/O parallelism to search multiple levels concurrently to reduce lookup latency.

3.4.4 Native Storage

Finally, the last line of work in this category attempts to perform native management of storage devices, such as HDDs and SSDs, to optimize the performance of LSM-tree implementations.

The LSM-tree-based Direct Storage system (LDS) [49] bypasses the file system to better exploit the sequential and aggregated I/O patterns exhibited by LSM-trees. The on-disk layout of LDS contains three parts: chunks, a version log, and a backup log. Chunks store the disk components of the LSM-tree. The version log stores the metadata changes of the LSM-tree after each flush and merge. For example, a version log record can record the obsolete chunks and the new chunks resulting from a merge. The version log is regularly checkpointed to aggregate all changes so that the log can be truncated. Finally, the backup log provides durability for in-memory writes by write-ahead logging.

LOCS [74] is an implementation of the LSM-tree on open-channel SSDs. Open-channel SSDs expose internal I/O parallelism via an interface called channels, where each channel functions independently as a logical disk device. This allows applications to flexibly schedule disk writes to leverage the available I/O parallelism, but disk reads must be served by the same channel where the data is stored. To exploit this feature, LOCS dispatches disk writes due to flushes and merges to all channels using a least-weighted-queue-length policy to balance the total amount of work allocated to each channel. To further improve the I/O parallelism for partitioned LSM-trees, LOCS places SSTables from different levels with similar key ranges into different channels so that these SSTables can be read in parallel.

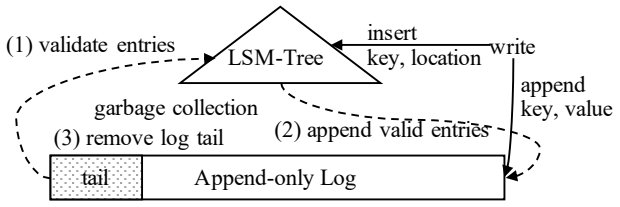

Fig. 13: WiscKey stores values into an append-only log to reduce the write amplification of the LSM-tree

NoFTL-KV [73] proposes to extract the flash translation layer (FTL) from the storage device into the key-value store to gain direct control over the storage device. Traditionally, the FTL translates the logical block address to the physical block address to implement wear leveling, which improves the lifespan of SSDs by distributing writes evenly to all blocks. NoFTL-KV argues for a number of advantages of extracting FTL, such as pushing tasks down to the storage device, performing more efficient data placement to exploit I/O parallelism, and integrating the garbage-collection process of the storage device with the merge process of LSM-trees to reduce write amplification.

3.4.5 Summary

In this subsection, we have reviewed the LSM-tree improvements exploiting hardware platforms, including large memory [15,19], multi-core [34], SSD/NVM [13,20,39,44,46, 54,71], and native storage [49,74,73]. To manage large memory components, both FloDB [15] and Accordion [19] take a multi-layer approach to limit random writes to a small memory area. The difference is that FloDB [15] only uses two layers, while Accordion [19] uses multiple layers to provide better concurrency and memory utilization. For multi-core machines, cLSM [34] presents a set of new concurrency control algorithms to improve concurrency.

A general theme of the improvements for SSD/NVM is to exploit the high random read throughput while reducing the write amplification of LSM-trees to improve the lifespan of these storage devices. The FD-tree [44] and its successor FD+tree [71] propose to use fractional cascading [22] to improve point lookup performance so that only one random I/O is needed for searching each component. However, today's implementations generally prefer Bloom filters since unnecessary I/Os can be mostly avoided by point lookups. Separating keys from values [20,46,54] can significantly improve the write performance of LSM-trees since only keys are merged. However, this leads to lower query performance and space utilization. Meanwhile, values must be garbage-collected separately to reclaim disk space, which is similar to the traditional log-structured file system design [60]. Finally, some recent work has proposed to perform native management of storage devices, including HDDs [49] and

SSDs [73, 74], which can often bring large performance gains by exploiting the sequential and non-overwriting I/O patterns exhibited by LSM-trees.

3.5 Handling Special Workloads

We now review some existing LSM-tree improvements that target special workloads to achieve better performance. The considered special workloads include temporal data, small data, semi-sorted data, and append-mostly data.

The log-structured history access method (LHAM) [51] improves the original LSM-tree to more efficiently support temporal workloads. The key improvement made by LHAM is to attach a range of timestamps to each component to facilitate the processing of temporal queries by pruning irrelevant components. It further guarantees that the timestamp ranges of components are disjoint from one another. This is accomplished by modifying the rolling merge process to always merge the records with the oldest timestamps from a component C_i into C_{i+1}.

The LSM-trie [76] is an LSM-based hash index for managing a large number of key-value pairs where each key-value pair is small. It proposes a number of optimizations to reduce the metadata overhead. The LSM-trie adopts a partitioned tiering design to reduce write amplification. Instead of storing the key ranges of each SSTable directly, the LSM-trie organizes its SSTables using the prefix of their hash values to reduce the metadata overhead, as shown in Figure 14. The LSM-trie further eliminates the index page, instead assigning key-value pairs into fixed-size buckets based on their hash values. Overflow key-value pairs are assigned to underflow buckets and this information is recorded in a migration metadata table. The LSM-trie also builds a Bloom filter for each bucket. Since there are multiple SSTables in each group at a level, the LSM-trie clusters all Bloom filters of the same logical bucket of these SSTables together so that they can be fetched using a single I/O by a point lookup query. In general, the LSM-trie is mainly effective when the number of key-value pairs is so large that even the metadata, e.g., index pages and Bloom filters, cannot be totally cached. However, the LSM-trie only supports point lookups since its optimizations heavily depend on hashing.

SlimDB [59] targets semi-sorted data in which each key contains a prefix x and a suffix y. It supports normal point lookups, given both the prefix and the suffix, as well as retrieving all the key-values pairs sharing the same prefix key x. To reduce write amplification, SlimDB adopts a hybrid structure with tiering on the lower levels and leveling on the higher levels. SlimDB further uses multi-level cuckoo filters [32] to improve point lookup performance for levels that use the tiering merge policy. At each level, a multi-level cuckoo filter maps each key to the ID of the SSTable where

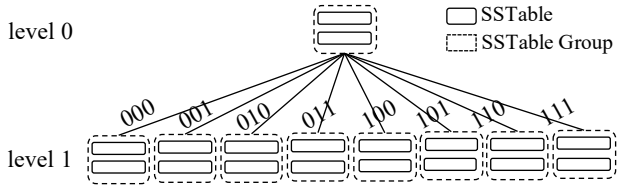

Fig. 14: The LSM-trie uses the prefix of hash values to manage SSTables. In this example, each level uses three bits to perform partitioning.

the latest version of the key is stored so that only one filter check is needed by a point lookup. To reduce the metadata overhead of SSTables, SlimDB uses a multi-level index structure as follows: It first maps each prefix key into a list of pages that contain this prefix key so that the key-value pairs can be retrieved efficiently given a prefix key. It then stores the range of suffix keys for each page to efficiently support point lookup queries based on both prefix and suffix keys.

Mathieu et al. [48] proposed two new merge policies optimized for append-mostly workloads with a bounded number of components. One problem of both leveling and tiering is that the number of levels depends on the total number of entries. Thus, with an append-mostly workload, where the amount of data keeps increasing, the total number of levels will be unbounded in order to achieve the write cost described in Section 2.3. To address this, this work studied the theoretical lower bound of the write cost of an online merge policy for an append-mostly workload given at most K components. It further proposed two merge policies, MinLatency and Binomial, to achieve this lower bound.

The four improvements presented here each target a specialized workload. It should be noted that their optimizations may be useless or even inapplicable for general purpose workloads. For example, the LSM-trie [76] only supports point lookups, while SlimDB [59] only supports a limited form of range queries by fetching all values for a prefix key. The adoption of these optimizations should be chosen carefully based on the given workload.

3.6 Auto-Tuning

We now review some research efforts to develop auto-tuning techniques for the LSM-tree to reduce the tuning burden for the end-user. Some techniques perform co-tuning of all parameters to find an optimal design, while others focus on some specific aspect such as merge policies, Bloom filters, or data placement.

3.6.1 Parameter Tuning

Lim et al. [45] presented an analytical model that incorporates the key distribution to improve the cost estimation of

LSM-tree operations and further used this model to tune the parameters of LSM-trees. The key insight is that the conventional worse-case analysis (Section 2.3) fails to take the key distribution into consideration. If a key is found to be deleted or updated during an early merge, it will not participate in future merges and thus its overall write cost will be reduced. The proposed model assumes a priori knowledge of the key distribution using a probability mass function $f_X(k)$ that measures the probability that a specific key k is written by a write request. Given p total write requests, the number of unique keys is estimated using its expectation as $Unique(p) = N - \sum_{k \in K} (1 - f_X(k))^p$, where N is the total number of unique keys and K is the total key space. Based on this formula, the total write cost for p writes can be computed by summing up the cost of all flushes and merges, except that duplicates keys, if any, are excluded from future merges. Finally, the cost model is used to find the optimal system parameters by minimizing the total write cost.

Monkey [25,26] co-tunes the merge policy, size ratio, and memory allocation between memory components and Bloom filters to find an optimal LSM-tree design for a given workload. The first contribution of Monkey is to show that the usual Bloom filter memory allocation scheme, which allocates the same number of bits per key for all Bloom filters, results in sub-optimal performance. The intuition is that the T components at the last level, which contain most of the data, consume most of the Bloom filter memory but their Bloom filters can only save at most T disk I/Os for a point lookup. To minimize the overall false positive rates across all of the Bloom filters, Monkey analytically shows that more bits should be allocated to the components at the lower levels so that the Bloom filter false positive rates will be exponentially increasing. Under this scheme, the I/O cost of zero-result point lookup queries will be dominated by the last level, and the new I/O cost becomes $O(e^{-\frac{M}{N}})$ for leveling and $O(T \cdot e^{-\frac{M}{N}})$ for tiering. Monkey then finds an optimal LSM-tree design by maximizing the overall throughput using a cost model similar to the one in Section 2.3 considering the workload's mix of the various operations.

3.6.2 Tuning Merge Policies

Dostoevsky [24] shows that the existing merge policies, that is, tiering and leveling, are sub-optimal for certain workloads. The intuition is that for leveling, the cost of zero-result point lookups, long range queries, and space amplification are dominated by the largest level, but the write cost derives equally from all of the levels. To address this, Dostoevsky introduces a lazy-leveling merge policy that performs tiering at the lower levels but leveling at the largest level. Lazy-leveling has much better write cost than leveling, but has similar point lookup cost, long range query cost, and space amplification to leveling. It only has a worse short range query cost than leveling since the number of components is increased. Dostoevsky also proposes a hybrid policy that has at most Z components in the largest level and at most K components at each of the smaller levels, where Z and K are tunable. It then finds an optimal LSM-tree design for a given workload using a similar method as Monkey [25]. It is worth noting that the performance evaluation of Dostoevsky [24] is very thorough; it was performed against well-tuned LSM-trees to show that Dostoevsky strictly dominates the existing LSM-tree designs under certain workloads.

Thonangi and Yang [70] formally studied the impact of partitioning on the write cost of LSM-trees. This work first proposed a ChooseBest policy that always selects an SSTable with the fewest overlapping SSTables at the next level to merge to bound the worst case merge cost. Although the ChooseBest policy outperforms the unpartitioned merge policy in terms of the overall write cost, there are certain periods when the unpartitioned merge policy has a lower write cost since the current level becomes empty after a full merge, which reduces the future merge cost. To exploit this advantage of full merges, this work further proposed a mixed merge policy that selectively performs full merges or partitioned merges based on the relative size between adjacent levels and that dynamically learns these size thresholds to minimize the overall write cost for a given workload.

3.6.3 Dynamic Bloom Filter Memory Allocation

All of the existing LSM-tree implementations, even Monkey [25], adopt a static scheme to manage Bloom filter memory allocation. That is, once the Bloom filter is created for a component, its false positive rate remains unchanged. Instead, ElasticBF [83] dynamically adjusts the Bloom filter false positive rates based on the data hotness and access frequency to optimize read performance. Given a budget of k Bloom filter bits per key, ElasticBF constructs multiple smaller Bloom filters with $k_1, \cdots, k_n$ bits so that $k_1 + \cdots + k_n = k$. When all of these Bloom filters are used together, they provide the same false positive rate as the original monolithic Bloom filter. ElasticBF then dynamically activates and deactivates these Bloom filters based on the access frequency to minimize the total amount of extra I/O. Their experiments reveal that ElasticBF is mainly effective when the overall Bloom filter memory is very limited, such as only 4 bits per key on average. In this case, the disk I/Os caused by the Bloom filter false positives will be dominant. When memory is relatively large and can accommodate more bits per key, such as 10, the benefit of ElasticBF becomes limited since the number of disk I/Os caused by false positives is much smaller than the number of actual disk I/Os to locate the keys.

3.6.4 Optimizing Data Placement

Mutant [80] optimizes the data placement of the LSM-tree on cloud storage. Cloud vendors often provide a variety of storage options with different performance characteristics and monetary costs. Given a monetary budget, it can be important to place SSTables on different storage devices properly to maximize system performance. Mutant solves this problem by monitoring the access frequency of each SSTable and finding a subset of SSTables to be placed in fast storage so that the total number of accesses to fast storage is maximized while the number of selected SSTables is bounded. This optimization problem is equivalent to a 0/1 knapsack problem, which is N/P hard, and can be approximated using a greedy algorithm.

3.6.5 Summary

The techniques presented in this category aim at automatically tuning LSM-trees for given workloads. Both Lim et al. [45] and Monkey [25, 26] attempt to find optimal designs for LSM-trees to maximize system performance. However, these two techniques are complimentary to each other. Lim et al. [45] uses a novel analytical model to improve the cost estimation but only focuses on tuning the maximum level sizes of the leveling merge policy. In contrast, Monkey [25, 26], as well as its follow-up work Dostoevsky [24], co-tune all parameters of LSM-trees to find an optimal design but only optimize for the worst-case I/O cost. It would be useful to combine these two techniques together to enable more accurate performance tuning and prediction.

Dostoevsky [24] extends the design space of LSM-trees with a new merge policy by combining leveling and tiering. This is very useful for certain workloads that require efficient writes, point lookups, and long range queries with less emphasis on short range queries. Thonangi and Yang [70] proposed to combine full merges with partitioned merges to achieve better write performance. Other tuning techniques focus on some aspects of the LSM-tree implementation, such as tuning Bloom filters by ElasticBF [83] and optimizing data placement by Mutant [80].

3.7 Secondary Indexing

So far, we have discussed LSM-tree improvements in a key-value store setting that only contains a single LSM-tree. Now we discuss LSM-based secondary indexing techniques to support efficient query processing, including index structures, index maintenance, statistics collection, and distributed indexing.

Before we present these research efforts in detail, we first discuss some basic concepts for LSM-based secondary indexing techniques. In general, an LSM-based storage system will contain a primary index with multiple secondary indexes. The primary index stores the record values indexed by their primary keys. Each secondary index stores the corresponding primary keys for each secondary key using either a composite key approach or a key list approach. In the composite key approach, the index key of a secondary index is the composition of the secondary key and the primary key. In the key list approach, a secondary index associates a list of primary keys with each secondary key. Either way, to process a query using a secondary index, the secondary index is first searched to return a list of matching primary keys, and those are then used to fetch the records from the primary index if needed. An example of LSM-based secondary indexing is shown in Figure 15. The example User dataset has three fields, namely Id, Name, and Age, where Id is the primary key. The primary index stores full records indexed by Id, while the two secondary indexes store secondary keys, i.e., Name and Age, and their corresponding Ids.

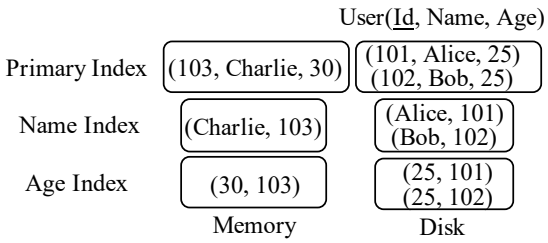

Fig. 15: Example LSM-based secondary indexes

3.7.1 Index Structures

The Log-Structured Inverted Index (LSII) [75] is an index structure designed for exact real-time keyword search on microblogs. A query q searches for the top K microblogs with the highest scores, which are computed as the weighted sum of significance, freshness, and relevance. To support efficient query processing, each keyword in a disk component stores three inverted lists of primary keys in descending order of significance, freshness, and frequency, respectively. Storing three inverted lists enables queries to be processed efficiently via the threshold algorithm [31], which stops query evaluation once the upper bound of the scores of the unseen microbiologs is lower than the current top K answers. However, only one inverted list is stored in the memory component since documents in the memory component often have high freshness and most of them will be accessed by queries. Moreover, storing multiple inverted lists would significantly increase the memory component's write cost.

Kim et al. [42] conducted an experimental study of LSM-based spatial index structures for geo-tagged data, including LSM-tree versions of the R-tree [35], Dynamic Hilbert

B$^+$-tree (DHB-tree) [43], Dynamic Hilbert Value B$^+$-tree (DHVB-tree) [43], Static Hilbert B$^+$-tree (SHB-tree) [33], and Spatial Inverted File (SIF) [40]. An R-tree is a balanced search tree that stores multi-dimensional spatial data using their minimum bounding rectangles. DHB-trees and DHVB-trees store spatial points directly into B$^+$-trees using space-filling curves. SHB-trees and SIFs exploit a grid-based approach by statically decomposing a two-dimensional space into a multi-level grid hierarchy. For each spatial object, the IDs of its overlapping cells are stored. The difference between these two structures is that an SHB-tree stores the pairs of cell IDs and primary keys in a B$^+$-tree, while a SIF stores a list of primary keys for each cell ID in an inverted index. The key conclusion of this study is that there is no clear winner among these index structures, but the LSM-based R-tree performs reasonably well for both ingestion and query workloads without requiring too much tuning. It also handles both point and non-point data well. Moreover, for non-index-only queries, the final primary key lookup step is generally dominant since it often requires a separate disk I/O for each primary key. This further diminished the differences between these spatial indexing methods.

Filters [11] augment each component of the primary and secondary indexes with a filter to enable data pruning based on a filter key during query processing. A filter stores the minimum and maximum values of the chosen filter key for the entries in a component. Thus, a component can be pruned by a query if the search condition is disjoint with the minimum and maximum values of its filter. Though a filter can be built on arbitrary fields, it is really only effective for time-correlated fields since components are naturally partitioned based on time and are likely to have disjoint filter ranges. Note that some special care is needed to maintain filters when a key is updated or deleted. In this case, the filter of the memory component must be maintained based on both the old record and the new record so that future queries will not miss new updates. Consider the example in Figure 16, which depicts a filtered primary LSM-tree. After upserting the new record (k1, v4, T4), the filter of the memory component becomes [T1, T4] so that future queries will properly see that the old record (k1, v1, T1) in the disk component has been deleted. Otherwise, if the filter of the memory component were only maintained based on the new value T4, which would be [T3, T4], a query with search condition T $\leq$ T2 would erroneously prune the memory component and thus actually see the deleted record (k1, v1, T1).

Qadar et al. [57] conducted an experimental study of LSM-based secondary indexing techniques including filters and secondary indexes. For filters, they evaluated component-level range filters and Bloom filters on secondary keys. For secondary indexes, they evaluated two secondary indexing schemes based on composite keys and key lists. Depending on how the secondary index is maintained, the key list

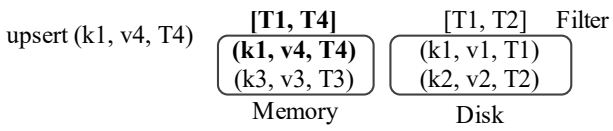

Fig. 16: Filter update example: the dataset contains three fields, a primary key (k), a value (v), and a creation time (T) that is also the filter key

scheme can be further classified as being either eager or lazy. The eager key list scheme always reads the previous list to create a full new list with the new entry added and inserts the new list into the memory component. The lazy key list scheme simply maintains multiple partial lists at each component. The experimental results suggest that the eager inverted list scheme incurs a large overhead on data ingestion because of the point lookups and high write amplification. When the query selectivity becomes larger, that is, when the result set contains more entries, the performance difference between the lazy key list scheme and the composite key scheme diminishes, as the final point lookup step becomes dominant. Finally, filters were found to be very effective with small storage overhead for time-correlated workloads. However, the study did not consider cleaning up secondary indexes in the case of updates, which means that secondary indexes could return obsolete primary keys.

3.7.2 Index Maintenance

A key challenge of maintaining LSM-based secondary indexes is handling updates. For a primary LSM-tree, an update can blindly add the new entry (with the identical key) into the memory component so that the old entry is automatically deleted. However, this mechanism does not work for a secondary index since a secondary key value can change during an update. Extra work must be performed to clean up obsolete entries from secondary indexes during updates.

Diff-Index [66] presents four index maintenance schemes for LSM-based secondary indexes, namely sync-full, sync-insert, async-simple, and async-session. During an update, two steps must be performed to update a secondary index, namely inserting the new entry and cleaning up the old entry. Inserting the new entry is very efficient for LSM-trees, but cleaning up the old entry is generally expensive since it requires a point lookup to fetch the old record. Sync-full performs these two steps synchronously during the ingestion time. It optimizes for query performance since secondary indexes are always up-to-date, but incurs a high overhead during data ingestion because of the point lookups. Sync-insert only inserts new data into secondary indexes, while cleaning up obsolete entries lazily by queries. Async-simple performs index maintenance asynchronously but guarantees its eventual execution by appending updates into an asynchronous

update queue. Finally, async-session enhances async-simple with session consistency for applications by storing new updates temporarily into a local cache on the client-side.

Deferred Lightweight Indexing (DELI) [67] enhances the sync-insert update scheme of Diff-Index [66] with a new method to cleanup secondary indexes by scanning the primary index components. Specifically, when multiple records with identical keys are encountered when scanning primary index components, the obsolete records are used to produce anti-matter entries to clean up the secondary indexes. Note that this procedure can be naturally integrated with the merge process of the primary index to reduce the extra overhead. Meanwhile, since secondary indexes are not always up-to-date, queries must always validate search results by fetching records from the primary index. Because of this, DELI cannot support index-only queries efficiently since point lookups must be performed for validation.

Luo and Carey [47] presented several techniques for the efficient exploitation and maintenance of LSM-based auxiliary structures, including secondary indexes and filters. They first conducted an experimental study to evaluate the effectiveness of various point lookup optimizations, including a newly proposed batched lookup algorithm that accesses components sequentially for a batch of keys, stateful B^+-tree search cursors, and blocked bloom filters [56]. They found that the batched lookup algorithm is the most effective optimization for reducing random I/Os, while the other two are mainly effective for non-selective queries at further reducing the in-memory search cost. To maintain auxiliary structures efficiently, two strategies were further proposed. The key insight is to maintain and exploit a primary key index, which only stores primary keys plus timestamps, to reduce disk I/Os. A validation strategy was proposed to maintain secondary indexes lazily in the background, eliminating the synchronous point lookup overhead. Queries must validate the primary keys returned by secondary indexes either by fetching records directly from the primary index or by searching the primary key index to ensure that the returned primary keys still have the latest timestamps. Secondary indexes are cleaned up efficiently in the background using the primary key index to avoid accessing full records; the basic idea for cleanup is to search the primary key index to validate whether each secondary index entry still has the latest timestamp, as in query validation. Compared to DELI [67], the validation strategy [47] significantly reduces the I/O cost for cleaning up secondary indexes since only the primary key index is accessed. A mutable-bitmap strategy was also introduced to efficiently maintain a primary index with filters. It attaches a mutable bitmap to each disk component so that old records can be directly marked as deleted, thereby avoiding the need to maintain filters based on old records.

3.7.3 Statistics Collection

Absalyamov et al. [7] proposed a lightweight statistics collection framework for LSM-based systems. The basic idea is to integrate the task of statistics collection into the flush and merge operations to minimize the statistics maintenance overhead. During flush and merge operations, statistical synopses, such as histograms and wavelets, are created on-the-fly and are sent back to the system catalog. Due to the multi-component nature of LSM-trees, the system catalog stores multiple statistics for a dataset. To reduce the overhead during query optimization, mergeable statistics, such as equi-width histograms, are merged beforehand. For statistics that are not mergeable, multiple synopses are kept to improve the accuracy of cardinality estimation.

3.7.4 Distributed Indexing

Joseph et al. [29] described two basic implementations of distributed secondary indexes on top of HBase [3], namely global secondary indexes and local secondary indexes, based on the two common approaches to indexing data in a parallel database. A global secondary index is implemented as a separate table that stores secondary keys plus their corresponding primary keys, and it is maintained using co-processors provided by HBase (similar to database triggers). This approach is easy to implement, but incurs a higher communication cost during data ingestion since a secondary index partition may be stored at a separate node from the primary index partition. A local secondary index avoids the communication cost during data ingestion by co-locating each secondary index partition together with the corresponding primary index partition. However, the downside for HBase is that this approach has to be implemented from scratch. Moreover, all partitions of a local secondary index must be searched, eve n for highly selective queries, since a local secondary index is partitioned by primary (not secondary) keys.

Zhu et al. [85] introduced an efficient approach for loading global secondary indexes using three steps: First, the primary index at each partition is scanned and sorted to create a local secondary index. Meanwhile, the statistics of the secondary key are collected to facilitate the next step. Second, based on the collected statistics from the first stage, the index entries of the secondary index will be range-partitioned and these partitions will be assigned to physical nodes. Finally, based on the assigned secondary key range, each node fetches secondary keys and their primary keys from all other nodes, which can be done efficiently by scanning the local secondary index built in the first stage.

Duan et al. [30] proposed a lazy maintenance approach for materialized views on distributed LSM-trees. The basic idea is to append new updates into a delta list of the material-