Enabling Timely and Persistent Deletion in LSM-Engines

SUBHADEEP SARKAR, Brandeis University, USA
TARIKUL ISLAM PAPON, Boston University, USA
DIMITRIS STARATZIS, TileDB, Inc., USA
ZICHEN ZHU and MANOS ATHANASSOULIS, Boston University, USA

8

Data-intensive applications have fueled the evolution of **log-structured merge (LSM)** based key-value engines that employ the *out-of-place* paradigm to support high ingestion rates with low read/write interference. These benefits, however, come at the cost of *treating deletes as second-class citizens*. A delete operation inserts a *tombstone* that invalidates older instances of the deleted key. State-of-the-art LSM-engines do not provide guarantees as to how fast a tombstone will propagate to *persist the deletion*. Further, LSM-engines only support deletion on the sort key. To delete on another attribute (e.g., timestamp), the entire tree is read and re-written, leading to undesired latency spikes and increasing the overall operational cost of a database. Efficient and persistent deletion is key to support: (i) streaming systems operating on a window of data, (ii) privacy with latency guarantees on data deletion, and (iii) *en masse* cloud deployment of data systems.

Further, we document that LSM-based key-value engines perform suboptimally in the presence of deletes in a workload. Tombstone-driven logical deletes, by design, are unable to purge the deleted entries in a timely manner, and retaining the invalidated entries perpetually affects the overall performance of LSM-engines in terms of space amplification, write amplification, and read performance. Moreover, the potentially unbounded latency for persistent deletes brings in critical privacy concerns in light of the data privacy protection regulations, such as the *right to be forgotten* in EU's GDPR, the *right to delete* in California's CCPA and CPRA, and *deletion right* in Virginia's VCDPA. Toward this, we introduce the delete design space for LSM-trees and highlight the performance implications of the different classes of delete operations.

To address these challenges, in this article, we build a new key-value storage engine, *Lethe*$^+$, that uses a very small amount of additional metadata, a set of new delete-aware compaction policies, and a new physical data layout that weaves the sort and the delete key order. We show that *Lethe*$^+$ supports any user-defined threshold for the delete persistence latency offering *higher read throughput* (1.17×–1.4×) and *lower space amplification* (2.1×–9.8×), with a modest increase in write amplification (between 4% and 25%) that can be further amortized to less than 1%. In addition, *Lethe*$^+$ supports efficient range deletes on a *secondary delete key* by dropping entire data pages without sacrificing read performance or employing a costly full tree merge.

CCS Concepts: • **Information systems** → **Data structures**; **Record storage systems**; *Database design and models*; *Query languages*; *Database management system engines*; • **Security and privacy**;

Additional Key Words and Phrases: Key-value stores, LSM-trees, data deletion, data privacy

This work was done while Subhadeep Sarkar was a post-doctoral associate at Boston University.
This work was done while Dimitris Staratzis was a graduate researcher at Boston University.
Authors' addresses: S. Sarkar, Brandeis University, 415 South Street, Waltham, MA 02453, USA; email: subhadeep@brandeis.edu; T. I. Papon, Z. Zhu, and M. Athanassoulis, Boston University, 665 Commonwealth Avenue, Boston, MA 02215, USA; emails: {papon, zczhu, mathan}@bu.edu; D. Staratzis, TileDB, Inc., 1 Broadway, Cambridge, MA 02142, USA; email: dimitris.staratzis@tiledb.com.

0362-5915/2023/08-ART8 $15.00
https://doi.org/10.1145/3599724

ACM Reference format:
Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. 2023. Enabling Timely and Persistent Deletion in LSM-Engines. *ACM Trans. Datab. Syst.* 48, 3, Article 8 (August 2023), 40 pages.
https://doi.org/10.1145/3599724

1 INTRODUCTION

Systems are Optimized for Fast Data Ingestion. Modern data systems process an unprecedented amount of data generated by a variety of applications that include data analytics, stream processing, and the new breed of information-centric technologies such as the Internet of Things [75, 78] and 5G [23, 35]. Cloud-based latency-sensitive applications like live video streaming [45], real-time health monitoring [64], e-commerce transactions [44], social network analysis [71], and online gaming [57] generate large volumes of data at a high velocity that requires **hybrid transactional/analytical processing (HTAP)** [10, 60, 63]. Modern commercial data store designs are, thus, driven by three principle objectives: (i) the raw data size is larger than the main memory size which means a significant proportion of the data resides on disk, (ii) fast writes, and (iii) fast reads with predictable tail latencies. Toward this, for the past decade, one of the main data management research challenges has been to design data systems that can sustain fast data ingestion rate and process queries at low latency [6, 12, 63]. To achieve this, modern data stores reduce read/write interference by employing *out-of-place* ingestion [14, 15, 26, 43, 52, 54, 65, 83]. Out-of-place data stores buffer the incoming data stream in memory and writes them back to the disk lazily to amortize the cost of writes while facilitating fast query processing.

LSM-based Key-Value Stores. A classical *out-of-place* data structure is the **log-structured merge (LSM)** tree [62], in which data is stored on a disk as immutable files, and updates and deletes are handled out-of-place [26, 58, 62, 67, 84]. LSM-trees buffer incoming data entries in main memory, and periodically flush this buffer as an *immutable sorted run* on durable storage [26, 58, 62, 67, 84]. As more sorted runs accumulate, they are iteratively sort-merged to form fewer yet larger sorted runs. This process, termed *compaction*, reduces the number of sorted runs accessed during a read query with amortized merging cost. Every compaction sort-merges existing sorted runs from consecutive levels and discards any invalid entries. LSM-trees are adopted by several modern systems including LevelDB [41] and BigTable [22] at Google, RocksDB [33] at Facebook, X-Engine [44] at Alibaba, Voldemort [56] at LinkedIn, Dynamo [30] at Amazon, Cassandra [8], HBase [9], and Accumulo [7] at Apache, and bLSM [81] and cLSM [40] at Yahoo. Relational data systems have been increasingly adopting LSM-style of updates. MyRocks [34] uses RocksDB as storage engine and SQLite4 [82] has experimented with LSM-trees in its storage layer. The out-of-place paradigm has also been adopted by many commercial data stores like Vertica [52, 83], Snowflake [25], Vectorwise [93], and TileDB [65, 85], and research designs like MaSM [14] and FD-tree [54] employ out-of-place updates to facilitate hybrid workloads.

The Challenge: Out-of-place Deletes. LSM-based storage engines use the out-of-place paradigm for any write operation, including ingestion (inserts), modification (updates), and deletion (deletes). As a result, a delete (update) is implemented by inserting additional meta-data that logically invalidates the older target entries [21]. We refer to this process as *logical deletes (updates).* Both logical deletes and updates show a complex three-way trade-off involving read performance, update performance, and main memory footprint [16]. Logical deletes, however, have wider implications in terms of (i) space amplification, (ii) read cost, (iii) write amplification, and (iv) privacy considerations, and hence, is the primary focus of this work (Figure 1(a)).

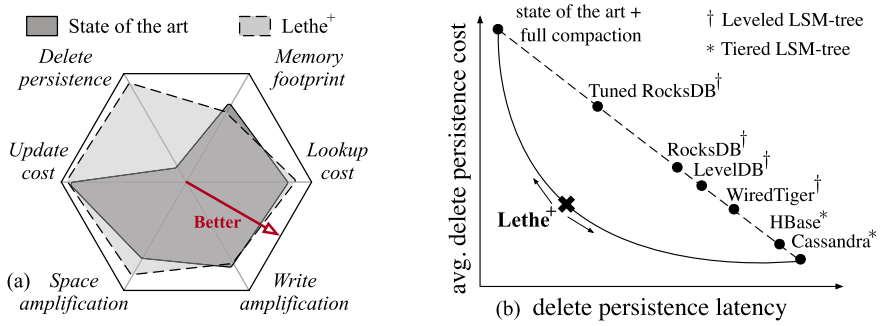

Fig. 1. (a) *Lethe*+ strikes an optimal balance between the latency/performance for timely delete persistence in LSM-trees, and (b) supports timely delete persistence by navigating the latency/cost trade-off.

In particular, a logical delete inserts a *tombstone* that invalidates all the older entries for a target key, with the expectation that the matching older entries will *eventually* be *persistently deleted*. In practice, the *delete persistence latency* is driven by (a) the system design choices and (b) workload characteristics. Neither can be fully controlled during execution, therefore, providing latency guarantees for persistent deletion of user data within a specific *delete persistence threshold* (termed *persistent timely deletion*) is nearly impossible in state-of-the-art LSM-based storage engines.

In fact, LSM-trees have a potentially unbounded delete persistence latency. In order to limit it, current designs employ a costly *full-tree compaction* on a periodic basis, typically, every 15–30 days. Such full-tree compactions are highly undesirable as they incur superfluous storage I/Os, which interfere with read performance, add to the write amplification, and result in performance unpredictability [20, 44]. Some LSM-engines, such as RocksDB, have the option of triggering compactions based on a *time-to-live* that may be configured to represent the delete persistence threshold [33, 70]. While this can guarantee timely persistence of deletes, it comes at the cost of high write amplification. This is because, upon expiration of the time-to-live, the file chosen for compaction must be compacted through all the subsequent levels so that all tombstones contained in the file are propagated to the last level and then purged. The I/O cost for such a reactive approach for persistent timely deletion can be as high as that of full-tree compactions, which makes it also undesirable for production systems.

Deletes in LSM-trees. LSM-trees are employed as the storage layer for relational systems [34], streaming systems [5, 46, 86], and pure key-value storage [18, 61, 89]. As a result, **an LSM delete operation may be triggered by various logical operations, not limited to user-driven deletes**. For example, ZippyDB, which is a distributed key-value engine that stores metadata for images and videos, processes 25.2M delete requests over a 24-hour window, which is 6% of the entire workload [21]. Deletes are also triggered by workloads that involve *periodic data migration* [59], streaming operations on a *running window* [44, 51], or entail *cleanup during data migration* [59]. In particular, dropping tables from an LSM-based data store with multiple column families is typically realized through a range delete operation [59]. In a time-series scenario that stores data as sorted on ingestion-timestamp, every update translates to a delete followed by the insertion of the new version [20]. Below, we distill the common concepts of three frequent delete use cases.

Scenario 1: An e-commerce company *EComp* stores its order details sorted by *order_id* in an LSM-tree, and needs to delete the order history for a particular user. Within the system, this delete request is translated to a set of point and range deletes on the *sort key*, i.e., *order_id*.

Scenario 2: A data company *DComp* stores its operational data in an LSM-tree with *document_id* as the sort key. As most of the data are relevant only for D days, *DComp* wants to delete all data with a *timestamp* that is older than D days (and archive them). At the same time, *DComp* frequently

accesses the documents using *document_id*, hence, the sort key (*document_id*) is different from the delete key (*timestamp*).

Scenario 3: *BComp* is a big data analytics company that runs a relational API on top of an LSM storage. To realize updates on attributes other than the primary key at the storage layer, *BComp* needs to translate every update to (i) a delete of the indexed key and (ii) an insert of the new indexed key for the same row, converting an update-intensive workload to a delete-intensive one.

The State of the Art and Why that is Not Enough. The three aforementioned scenarios highlight the limitations of state-of-the-art LSM-based storage engines. Modern commercial LSM-engines cannot efficiently support *EComp* (Scenario 1) and *BComp* (Scenario 3) for two reasons. First, as deletes insert tombstones (retaining the physical entries), they **increase space amplification**. Second, retaining superfluous entries in the tree **adversely affects read performance** because read queries have to discard potentially large collections of invalid entries, which further "pollute" the filter metadata [44], and **increase write amplification** because invalid entries are repeatedly compacted. Further, LSM-engines are ill-suited for *DComp* from the second scenario because they cannot efficiently support a range deletion in a delete key other than the sort key (termed *secondary range deletes*). Instead, they employ a *full-tree compaction*, which causes an excessive number of **wasteful I/Os** while reading, merging, and re-writing the sorted files of the *entire* tree [44].

Delete persistence latency. In order to be able to report that a delete *persisted*, the corresponding tombstone has to reach the last level of the tree through iterative compactions to discard all invalidated entries. The time elapsed between the insertion of the tombstone in the tree and the completion of the last-level compaction is termed *delete persistence latency*. **LSM logical deletes do not provide delete persistence latency guarantees**, hence *EComp* and *BComp* cannot offer such guarantees to their users. In order to add a hard limit on delete persistence latency, current designs employ a costly full-tree compaction **that interferes with read performance and predictability while increasing the operational cost** remarkably.

Privacy through deletion. Having unbounded delete persistence latency does not ensure timely deletion of the physical entries from a database and may lead to a breach of data privacy. For example, it was recently reported that Twitter retains user messages years after they have been deleted, even after user accounts have been deactivated [88]. With the new data privacy protection acts being enforced across different countries and for several states within the United States, limiting the end-to-end data lifecycle has become critical [31, 74]. In particular, data retention policies such as the *right to be forgotten* in EU's GDPR [1], the *right to delete* in California's CCPA and CPRA [2, 3], and *deletion right* in Virginia's VCDPA [4] coming into play, physically deleting the users' data within a finite threshold has emerged as a fundamental challenge for several data companies.

LSM-engines are not optimized when it comes to providing delete persistence latency guarantees or supporting secondary range deletes. In our interactions with engineers working on LSM-based production systems, we learned that periodic deletes of a large fraction of data based on timestamp are very frequent. To quote an engineer working on XEngine [44, 90], "*Applications may keep data for different durations (e.g., 7 or 30 days) for their own purposes. But they all have this requirement for deletes every day. For example, they may keep data for 30 days, and daily delete data that turned 31-days old, effectively purging 1/30 of the database every day.*" This deletion is performed with a full-tree compaction. To further quote the same team, "*Forcing compactions to set a delete latency threshold, leads to significant increase in compaction frequency, and the observed I/O utilization often peaks. This quickly introduces performance pains.*" For large data companies, deleting 1/7 or 1/30 of their database accounts for several GBs or TBs that is required to be persistently removed daily. The current approach of employing full-tree compactions is suboptimal as it (1) causes high latency spikes for reads, (2) increases write amplification, and (3) adds significantly

to the overall operational cost of the database. Full-tree compactions are, therefore, heavily undesired in production systems, particularly when serving latency-sensitive applications. The goal of this work is to address these challenges while **retaining the benefits of LSM-based design** and reusing a major part of the existing design and system codebase.

"Full tree compactions should be avoided."

The objective of this work is to design an LSM-based data store that (i) provides latency guarantees for delete persistence without compromising the overall system performance and (ii) is able to support secondary deletes efficiently for applications with different *delete key* and *sort key*.

The Solution: *Lethe*$^+$. We propose *Lethe*$^+$[1], a new LSM-based key-value store that offers efficient deletes without compromising the benefits of LSM-trees. *Lethe*$^+$ pushes the boundary of the traditional LSM design space by adding delete persistence as a new design goal, and is able to meet user requirements for delete persistence latency. Figures 1(a) and (b) show a qualitative comparison between state-of-the-art LSM-engines [8, 9, 33, 41, 89] and *Lethe*$^+$ with respect to the efficiency and cost of timely persistent deletes. *Lethe*$^+$ introduces two new LSM design components: *FADE* and *KiWi*$^+$.

FADE (**Fa**st **De**letion) is a new family of compaction strategies that prioritize files for compaction based on (a) the number of invalidated entries contained, (b) the age of the oldest tombstone, and (c) the range overlap with other files. FADE uses this information to decide *when* to trigger a compaction on *which* files, to purge invalid entries within a threshold.

While FADE allows *Lethe*$^+$ to persist deletes in a time-bound manner, it does not improve the secondary range delete performance. The key intuitions behind facilitating efficient secondary range deletes are: (a) to avoid full-tree compactions and (b) to do so without hurting read performance. To achieve this, we propose **Key Wea**ving Storage Layout **(*KiWi*)**, which is a new continuum of physical layouts that offers tunable secondary range delete performance without causing latency spikes, by introducing the concept of *delete tiles*. An LSM-tree level consists of several sorted files that logically form a sorted run. KiWi augments the design of each file with several delete tiles, each containing several data pages. A delete tile is sorted on the secondary (delete) key, while each data page remains internally sorted on the sort key. Having Bloom filters at the page level, and fence pointers for both the sort key and the secondary delete key, KiWi facilitates secondary range deletes by dropping entire pages from the delete tiles, with a constant factor increase in false positives. Maintaining the pages as sorted on the sort key also means that once a page is in memory, read queries maintain the same efficiency as the state of the art. We further improve the trade-off between the costs of secondary range deletes and point lookups by introducing KiWi$^+$. KiWi$^+$ takes into account the workload characteristics (composition and distribution) and the LSM-tuning (page size, size ratio, and Bloom filter size) to propose the optimal size of a delete tile separately for every tree-level. The level-wise optimal data layout proposed by KiWi$^+$ pushes the Pareto frontier constituted by the costs of secondary range deletes and (short) read queries farther, close to the optimum.

Putting everything together, *Lethe*$^+$ presents the design of a deletion-aware LSM-engine that offers timely and efficient deletion of user data while improving read performance, supports user-defined delete latency thresholds, and enables practical secondary range deletes.

Additional Materials with respect to Conference Publication. *Lethe*$^+$ is an extension of our prior work on delete-conscious LSM-engines [17, 73, 79]. In this article, we (i) formally prove that FADE persists primary deletes within a given delete persistence threshold and (ii) justify the assignment of the **time-to-live (TTL)** assigned to every LSM-level by FADE through theoretical

[1] *Lethe* (pronounced as: /ˈliːθiː/), the Greek mythological river of oblivion, signifies efficient deletion.

verification (Section 4.1). Further, (iii) we introduce a level-wise interweaved data layout, KiWi$^+$, that takes into consideration the workload characteristics and the LSM-tuning to identify the optimal data layout for each level in an LSM-tree. We present this new hierarchical data layout in Section 4.3 along with its performance implications. The proposed design pushes the Pareto frontier for the costs of primary reads and secondary range deletes closer to the theoretical optimum, thereby, improving the overall performance of an LSM-engine significantly. We also (iv) present the analysis for KiWi$^+$ for tiered LSM-tree designs (Section 4.3), which demonstrates the applicability of *Lethe*$^+$ in commercial engines such as HBase and Cassandra. Finally, (v) we performed a large number of new experiments with KiWi and KiWi$^+$ by varying (a) the proportion of empty and (b) non-empty point lookups, (c) the proportion of short range queries, and (d) the size ratio of the tree to understand their implications on performance. We further enrich this discussion by varying (e) the proportion as well as (f) the selectivity of secondary range deletes. We present the new results in Section 5.

Contributions. Below we present the contributions of the paper.

- We introduce the taxonomy of deletes in modern key-value workloads and highlight the challenges in realizing the different classes of deletes in LSM-based storage engines.
- We analyze the implications of out-of-place logical deletes on the performance of LSM-engines (in terms of read performance, space amplification, and write amplification), and on data privacy (in terms of deletion cost and guarantees on timely data deletion).
- We introduce FADE that ensures efficient and timely persistence of primary deletes without hurting performance or increasing resource consumption.
- We introduce Key Weaving Storage Layout, an interweaved data layout based on the sort and delete attributes, that supports efficient secondary range deletes.
- We improve further on the interweaved layout to introduce KiWi$^+$ which takes into account the workload characteristics and the LSM-tuning to have different data layouts in different levels on an LSM-tree.
- We present the design of *Lethe*$^+$ that integrates FADE and KiWi$^+$ in a state-of-the-art LSM-engine and enables fast deletes with a tunable balance between delete persistence latency and the overall performance of the system.
- We demonstrate that *Lethe*$^+$ offers delete latency guarantees, having up to 1.4× higher read throughput. The higher read throughput is attributed to the significantly lower space amplification (up to 9.8× for only 10% deletes) because it purges invalid entries faster. These benefits come at the cost of 4%–25% higher write amplification.
- Further, we demonstrate that via KiWi$^+$, *Lethe*$^+$ is the first LSM-based storage engine to support efficient secondary range deletes while achieving optimal workload execution time.
- Finally, we propose an extension of SQL to support declarative on-demand and retention-based deletes using either arbitrary or pre-defined delete persistence thresholds.

2 LSM BACKGROUND

This section provides the necessary background on LSM-trees. A more detailed document on LSM-tree background can be found in [58, 72, 77].

Basics. LSM-trees store key-value pairs, where a *key* refers to a unique object identifier, and the data associated with it, is referred to as *value*. For relational data, the primary key acts as the key, and the remaining attributes in a tuple constitute the value. As entries are sorted and accessed by the key, we refer to it as the *sort key*. For an LSM-tree with L levels, we assume that its first level (Level 0) is an in-memory buffer and the remaining levels (Level 1 to $L-1$) are disk-resident. We adopt the notations from the literature [26, 58].

Buffering Inserts and Updates. Inserts, updates, or deletes are buffered in memory. A delete (update) to a key that exists in the buffer, deletes (replaces) the older key in-place, otherwise the delete (update) remains in memory to invalidate any existing instances of the key on the disk-resident part of the tree. Once the buffer reaches its capacity, the entries are sorted by key to form an *immutable sorted run* and are flushed to the first disk-level (Level 1). When a disk-level reaches its capacity, all runs within that level are sort-merged and pushed to the next level. To bound the number of levels in a tree, runs are arranged in exponentially growing levels on disk. The capacity of Level i ($i \geq 1$) is greater than that of Level $i-1$ by a factor of T, termed the *size ratio* of the tree.

Compaction Policies: Leveling and Tiering. Classically, LSM-trees support two merging policies: leveling and tiering. In leveling, each level may have at most one run, and every time a run in Level $i-1$ ($i \geq 1$) is moved to Level i, it is greedily sort-merged with the run from Level i, if it exists. With tiering, every level must accumulate T runs before they are sort-merged. During a sort-merge (compaction), entries with a matching key are consolidated and only the most recent valid entry is retained [32, 72, 80]. Recently hybrid compaction policies fuse leveling and tiering in a single tree to strike a balance between the read and write throughput based on workload specifications [28, 29].

Partial Compaction. To amortize latency spikes from compactions in larger levels, state-of-the-art LSM-engines organize runs into smaller files, and perform compactions at the granularity of files instead of levels [32]. If Level i grows beyond a threshold, a compaction is triggered and one file from Level i is chosen to be *partially compacted* with files from Level $i+1$ that have an overlapping key-range. Deciding which file to compact depends on the *file picking policy* adopted by the storage engine design [76, 80]. For instance, to optimize write throughput, we select files from Level i with minimal overlap with files in Level $i+1$, to minimize both write amplification and compaction time. Partial compactions spread the total number of I/Os performed during compactions of an entire level over time to multiple smaller compactions [80].

Querying LSM-Trees. A point lookup begins at the memory buffer and traverses the tree from the smallest disk-level to the largest one. For tiering, within a level, a lookup moves from the most to the least recent tier. The lookup terminates when it finds the first matching entry. A range lookup returns the most recent versions of the target keys by sort-merging the qualifying key ranges across all runs in the tree.

Optimizing Lookups. Read performance is optimized using **Bloom filters (BFs)** and fence pointers [77]. In the worst case, a lookup needs to probe every run. To reduce this cost, LSM-engines use one BF per run in main memory [26, 33]. *Bloom filters* allow a lookup to skip probing a run altogether if the filter-lookup returns negative. In practice, for efficient storage, BFs are maintained at the granularity of files [32]. *Fence pointers* store the smallest key per disk page in memory [26], to quickly identify which page(s) to read for a lookup, and perform up to one I/O per run for point lookups. In addition to these, production LSM-engines also maintain fence pointers (also termed block indexes) for every file to reduce the I/Os needed for point and range queries.

3 THE IMPACT OF DELETES

We now describe the design space of deletes in LSM-trees and analyze their implications on the performance of a storage engine. Throughout this section, we refer to Figures 2, 3, and 4 to illustrate the delete design space.

3.1 Delete Design Space

We first present the taxonomy of delete operations in LSM-trees, and then we discuss how deletes are realized in state-of-the-art LSM-engines [17]. Further, we analyze the implications of deletes

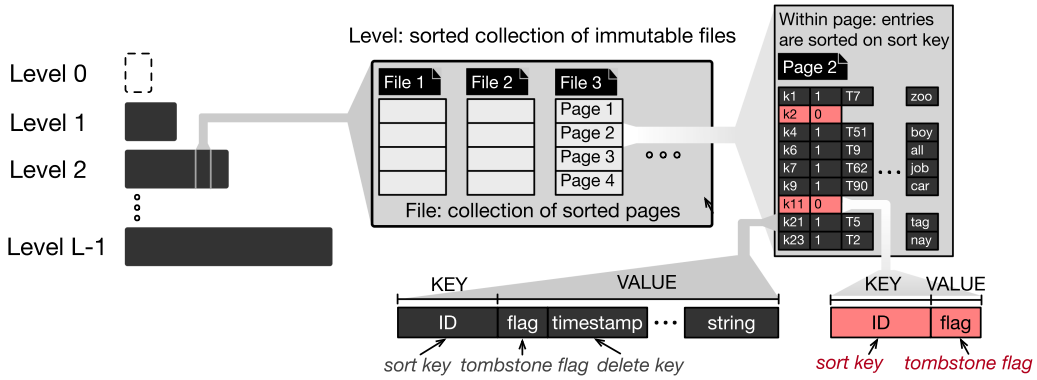

Fig. 2. An LSM-tree stores data across several levels of exponentially larger capacity. Each level is a collection of sorted immutable files, each of which contains several pages of data.

on the overall performance of an LSM-engine in terms of space amplification, point and range lookup performance, write amplification, and write stalls [92]. We also highlight the impact of out-of-place deletes on data privacy in terms of timely deletion of user data [17, 79].

3.1.1 Primary Deletes. Primary deletes refer to *delete operations that are issued on the sort key* of an LSM-based key-value data store. LSM-trees organize the data on disk based on the sort key (or simply, the key), and deletes issued on a particular key or a key range are common in key-value workloads [21]. Figure 2 shows a leveled LSM-tree, the structure of a key-value pair, and a tombstone. A key-value pair contains typically many attributes as part of the value, and a tombstone consists of the (deleted) key and the tombstone flag. In LSM-trees, an entry at Level i is always more recent than an entry with the same key at Level j, if $j > i$. LSM-trees exploit this to logically delete using tombstones that supersede older entries with a matching key. Tombstones are physically purged only after they reach the last level of an LSM-tree, deleting the target entries in the intermediate levels during compactions. Primary deletes can be further classified as *point* and *range* deletes based on the number of entries invalidated by the deletion operation.

Point Deletes. A primary point delete operation is realized logically by inserting a point tombstone (or simply, a tombstone) against the key to be deleted (Figure 3). Within the memory buffer, a tombstone may replace any older entry with a matching key in-place.[2] On disk, the tombstones are stored within a run in sorted order along with other key-value pairs. During compaction, a tombstone deletes older entries with the same key and is retained as there might be more (older) entries with the same delete key in subsequent compactions (Figure 3), unless we compact it with the last level.

Persistence of logical point deletes. A tombstone is *eventually* discarded during its compaction with the last level of the tree, making the logical delete *persistent*. Production-scale LSM-engines typically employ a partial compaction routine to amortize the cost for compactions and avoid prolonged write stalls. Once a level reaches a nominal saturation, instead of compacting all data from that level, the partial compaction routines select a file from that level to be merged with the overlapping files of the immediately following level. The file is selected in various ways, with the two most common ways being (a) in a round-robin manner [41] and (b) the one with the lowest overlap in an attempt to minimize the merging cost [33].

[2]In practice, whether or not a tombstone replaces a matching entry in place within the buffer, depends on the buffer implementation. If the buffer is realized as a log, then all entries are preserved until the buffer is flushed.

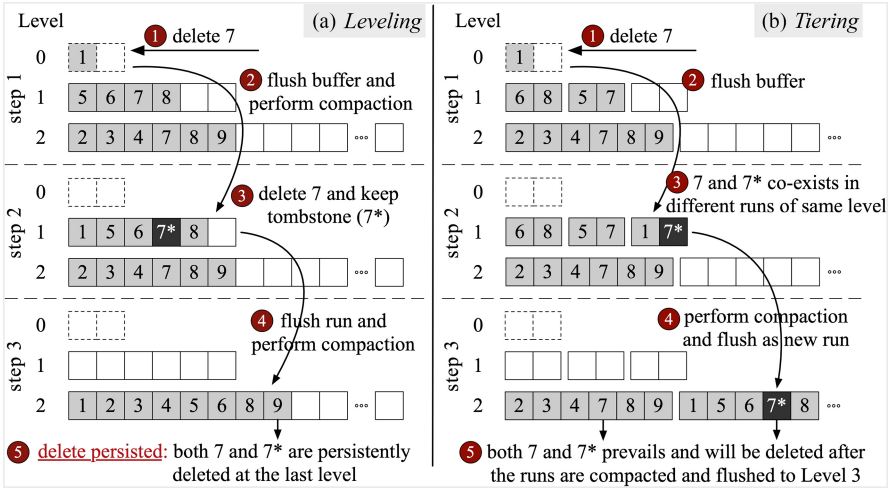

Fig. 3. In an LSM-tree, for every tombstone, there can be (a) one matching entry per level for leveling or (b) one matching entry per tier per level (T per level) for tiering, where $T = 3$ in this example.

Range Deletes. Deleting using a range of keys is also common in many real-life key-value workloads [21, 59]. However, realizing range deletes is non-trivial, as (i) the I/O cost for in-place range deletes is remarkably high due to write stalls, and (ii) out-of-place range delete solutions bear an intrinsic trade-off with the cost for both point and range queries. Thus, most production systems do not support primary range deletes [40, 41, 47, 81, 89]. Some systems, like RocksDB, adopt the out-of-place solution to support range delete operations by generating special *range tombstones* that are stored in a separate *range tombstone block* within the disk files [33]. In addition, a histogram is maintained in memory that stores information on the ranges of deleted keys along with their deletion-timestamp to ensure query correctness. During data access, every point and range query must consult this histogram to guarantee that the query result set does not contain any invalid data. Thus, while range tombstones allow realizing range deletes efficiently, it comes at the cost of increased latency for point and range queries [20, 59].

Persistence of logical range deletes. Similar to point deletes, range deletes are persisted when the files that contain the range tombstones are compacted with the last level of an LSM-tree. Thus, in practice, a complete full-tree compaction is periodically employed to ensure delete persistence [44]. During such compactions, all reads and writes to the data store are stalled, which results in latency spikes [20, 59].

Persistence Latency. The latency for persisting a logical delete depends on the workload and the data size. The left part of Figure 4 illustrates the operation "*delete all entries with* `ID = k`". Within the system, the operation inserts a tombstone, k^*, that logically *invalidates* k. On disk, there might be several entries with a matching key k, and these entries may be located at any level between 1 and L. Thus, to ensure persistent deletion of all matching entries, k^* must participate in L compactions, one at each level of a leveled LSM-tree. Since compactions are triggered when a level reaches a nominal capacity, the *rate of unique insertions* is effectively driving the compactions. The size of a level grows exponentially with T, therefore, a taller tree requires exponentially more unique insertions to propagate a tombstone to the last level. Figure 4 shows the threshold time for persisting logical deletes, denoted by D_{th}, along with the time spent by the tombstone k^* in every level of the LSM-tree. Formally, t_i denotes the time spent by k^* in Level i before it is compacted with Level $i+1$. Thus, $\sum_{i=1}^{L-1} t_i$ corresponds to the cumulative time spent by k^* in the tree before it is

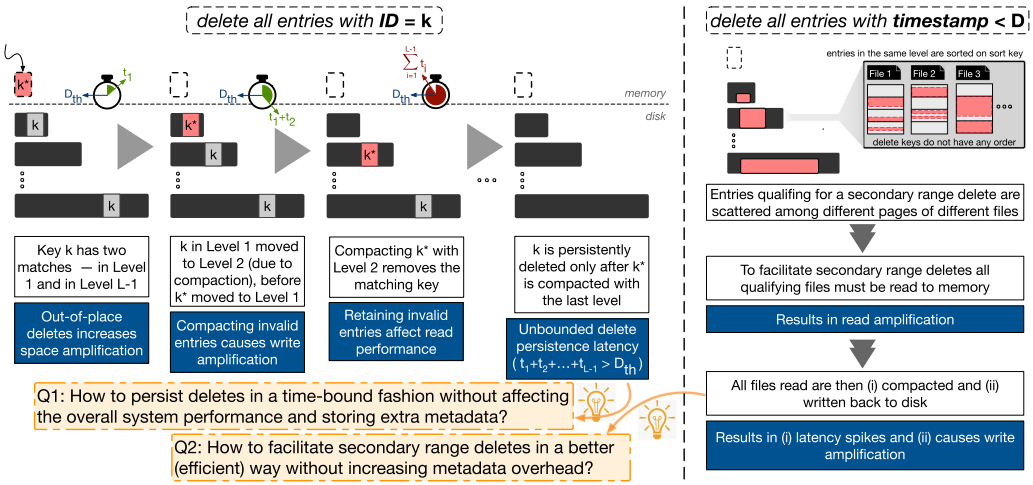

Fig. 4. The problem space of out-of-place deletes in LSM-trees.

persisted, and thus denotes the delete persistence latency for k*. State-of-the-art LSM-tree-based data stores are unable to provide any latency guarantees for delete persistence as t_i is typically driven by factors that are difficult to control. Overall, we observe that the delete persistence latency depends on (i) the rate of unique insertions, (ii) the current height of a tree, (iii) the size ratio of a tree, and (iv) the compaction file picking policy. Thus, in practice, $\sum_{i=1}^{L-1} t_i$ may be significantly larger than D_{th}, in effect, violating the requirements for timely persistence of deletes.

Adversarial Workloads. Tombstones may be recycled in intermediate levels of the tree leading to *unbounded delete persistence latency* and perpetual retention of invalid entries [20]. For example, a workload that mostly modifies hot data (in the first few levels) will grow the tree very slowly, keeping its structure mostly static. Another example is a workload with interleaved inserts and deletes, with the deletes issued on a few recently inserted entries that are at the smaller levels. In both cases, a newly inserted tombstone may be recycled in compactions high up the tree that consolidate entries rather than propagate towards the last level.

3.1.2 Secondary Deletes. We refer to deletes based on an attribute other than the sort key as *secondary deletes*. In many practical use cases, we may need to organize data based on a sort key (e.g., the `RowID`), but we have deletes on a different attribute (e.g., `timestamp`). Secondary point deletes are not native of key-value workloads. In fact, the most prudent approach to support secondary point deletes is to construct a secondary index on the (secondary) delete key and convert them secondary point deletes to primary point deletes. On the other hand, **secondary range deletes** are quite common in practice. Consider the operation "*delete all entries that are older than D days*", similar to the second scenario from the introduction. In Figure 2, we highlight the sort key (`ID`) and the delete key (`timestamp`) of a key-value pair. As the entries in a tree are sorted on the sort key, an entry with a qualifying delete key may be anywhere in the tree, and this delete pattern is not efficiently supported. Rather, systems often resort to full-tree compactions (right part of Figure 4).

3.1.3 Limitations of the State of the Art. In state-of-the-art LSM-engines, deletes are considered as "second-class citizens". In practice, to ensure time-bounded persistence of logical deletes and to facilitate secondary range deletes, data stores resort to *periodic full-tree compaction* [24, 44]. However, this is an extremely expensive solution as it involves superfluous disk I/Os, increases write amplification and results in latency spikes. To reduce excessive I/Os, RocksDB implements a

file selection policy based on the number of tombstones [33]. This reduces the number of invalid entries, but it does not offer persistent delete latency guarantees.

3.2 Implications of Out-of-place Deletes

Next, we quantify the implications of out-of-place deletes on read performance, and space and write amplification.

Model Details. We assume an LSM-tree with size ratio T, that stores N entries across $L+1$ levels. The size of the memory buffer is $M = P \cdot B \cdot E$, where P is the number of disk pages in the buffer, B is the number of entries per page, and E is the average size of an entry. The capacity of this tree is $\sum_{i=0}^{L} M \cdot T^i$, where $M \cdot T^i$ is the capacity of Level i. The N entries inserted in the tree includes δ_p point tombstones and δ_r range tombstones that have an average selective of σ. Table 1 shows all the parameters used in our modeling.

3.2.1 Space Amplification. Deletes increase space amplification by (i) the tombstones and (ii) the invalidated entries (for every key, there might be several invalid versions). Space amplification increases storage cost and the overhead for data organization (sorting) and processing (read I/Os during compaction). Commercial databases often report space amplification of about 11% [59], however, this corresponds to $T = 10$, a single point in the vast design continuum.

Analysis. Following prior work [28], we define space amplification as the ratio between the size of superfluous entries and the size of the unique entries in the tree, $s_{amp} = \frac{\texttt{csize}(N) - \texttt{csize}(U)}{\texttt{csize}(U)}$, where $\texttt{csize}(N)$ is the cumulative size of all entries and $\texttt{csize}(U)$ is the cumulative size of all unique entries. Note that $s_{amp} \in [0, \infty)$, and that if all inserted keys are unique there is no space amplification.

Without Deletes. Assume a workload with inserts and updates (but no deletes) for a leveled LSM-tree. In the worst case, all entries in levels up to $L-1$ can be updates for the entries in Level L, leading to space amplification $O(1/T)$. For a tiered LSM-tree, the worst case is when the tiers of a level overlap, and the first $L-1$ levels contain updates for Level L. This leads to space amplification $O(T)$.

With Deletes. If the size of a tombstone is the same as the size of a key-value entry, the asymptotic worst-case space amplification remains the same as that with updates for leveling. However, in practice, a tombstone is orders of magnitude smaller than a key-value entry. We introduce the tombstone size ratio $\lambda = \frac{\texttt{size}(tombstone)}{\texttt{size}(key\text{-}value)} \approx \frac{\texttt{size}(key)}{\texttt{size}(key)+\texttt{size}(value)}$, where $\texttt{size}(key)$ and $\texttt{size}(value)$ is the average size of a key and an entry, respectively. λ is bounded by $(0, 1]$, and a smaller λ implies that a few bytes (for tombstones) can invalidate more bytes (for key-values) and lead to larger space amplification given by $O(\frac{(1-\lambda)\cdot N+1}{\lambda \cdot T})$. For tiering, in the worst case, tombstones in the recent-most tier can invalidate all entries in that level, resulting in space amplification $O(\frac{N}{1-\lambda})$.

3.2.2 Read Performance. Point tombstones are hashed to the BFs the same way as valid keys, and thus increase the **false positive rate (FPR)** for the filters as well as the I/O cost for point lookups. Also, the deleted entries cause the range queries to scan invalid data before finding the qualifying keys. Consider that a range delete with 0.5% selectivity over a 100GB database invalidates 500MB, which might have to be scanned (and discarded) during query execution.

Analysis: Point Lookups. A point lookup probes one (or more) BF before performing any disk I/O. The FPR of a BF depends on the number of bits allocated to the filter in the memory (m) and the number of entries (N) hashed into the filter, and is given by $e^{-m/N \cdot (ln(2))^2}$. For leveling, the average worst-case point lookup cost on non-existing entries is $O(e^{-m/N})$, and for tiering, the cost becomes $O(T \cdot e^{-m/N})$ [26]. For lookups on existing entries, this cost increases by 1 as the lookup

Table 1. *Lethe*$^+$ Parameters

Symbol	Description	Reference value
N	Number of entries inserted in tree (including tombstones)	2^{20} entries
L	Number of tree-levels on disk with N entries	3 levels
T	Size ratio of the LSM-tree	10
P	Size of memory buffer in disk pages	512 disk pages
B	Number of entries in a disk page	4 entries
E	Average size of a key-value entry	1024 bytes
M	Memory buffer size	16 MB
m	Total main memory allocated to BFs	10 MB
ϕ	False positive rate of BF with N entries in the tree	-
I	Ingestion rate of unique entries in tree	1024 entries/sec
s	Selectivity of a long range lookup	-
δ_p	Number of point deletes issued	3×10^5 entries
λ	Tombstone size/average key-value size	0.1
δ_r	Number of range deletes issued	10^3 entries
σ	Average selectivity of range deletes	5×10^{-4}
N_δ	Approximate number of entries after persisting deletes	-
L_δ	Number of tree-levels on disk with N_δ entries	-
ϕ_δ	False positive rate of BF with N_δ entries in the tree	-
h	Number of disk pages per delete tile	16 disk pages

has to probe at least one page. Since tombstones are hashed into the BFs, retaining tombstones and invalid entries increases their FPR, thus hurting point lookups.

Analysis: Range Lookups. A range query on the sort key reads and merges all qualifying disk pages. The I/O cost of a *short range query* accessing at most two pages per level is $O(L)$ for leveling and $O(L \cdot T)$ for tiering. The I/O cost for long range lookups depends on the selectivity of the lookup range, and is $O(s \cdot N/B)$ for leveling and $O(s \cdot T \cdot N/B)$ for tiering. When answering range queries, tombstones and invalid entries have to be read and discarded, slowing down the range queries.

3.2.3 Write Amplification. Before being consolidated, an invalid entry may participate in multiple compactions. Repeatedly compacting invalid entries increases write amplification, which is particularly undesirable for installations that the durable storage has limited write endurance [59].

Analysis. We define write amplification, w_{amp} as the ratio of the total bytes written on disk that correspond to unmodified entries to the total bytes written corresponding to new or modified entries, $w_{amp} = \frac{\texttt{csize}(N^+) - \texttt{csize}(N)}{\texttt{csize}(N)}$. N^+ is the number of all the entries written to disk including the entries re-written as unmodified after a compaction. For leveling, every entry participates on average in $T/2$ compactions per level which makes $N^+ = N \cdot L \cdot T/2$. For tiering, every entry is written on disk once per level, implying $N^+ = N \cdot L$. Thus, w_{amp} for leveled and tiered LSM-trees are given by $O(L \cdot T)$ and $O(T)$, respectively. Note that, as the data size increases, entries participate in more compactions unmodified including invalid entries further increasing write amplification.

3.2.4 Persistence Latency and Data Privacy. The lack of guarantees in persistence latency has severe implications on data privacy. With new data privacy protection acts [2, 39] and the increased protection of rights like the *right-to-be-forgotten*, data companies are legally obliged to persistently delete data offering guarantees [31] and rethink the end-to-end data lifecycle [74].

Analysis. We define *delete persistence latency* as the worst-case time required, following the insertion of a tombstone, to ensure that the tree is void of any entry with a matching (older) key to that of the tombstone. This time depends on the insertion rate of unique key-value entries (I) and the height of the tree ($L-1$), and is the time needed to insert the minimum number of unique keys that is sufficient to trigger enough compactions. For leveling, delete persistence latency is $O(\frac{T^{L-1} \cdot P \cdot B}{I})$ and for tiering is $O(\frac{T^{L} \cdot P \cdot B}{I})$. This shows that for an LSM-tree with a large number of entries (T^L) that is built by an update-intensive workload, the delete persistence latency can be remarkably high.

3.3 Implications of the Storage Layout

Every file of an LSM-tree is sorted using the sort key. While this supports read, update, and delete queries on the sort key it cannot support operations on a secondary attribute.

Secondary Range Deletes. Secondary range deletes refer to the operation of deleting a range on entries based on a delete key that is different from the sort key. In a key-value store, the delete key is typically part of the "value" field of an entry. As there is no ordering of the entries qualifying for a secondary range delete operation, they might be scattered across several levels of the tree, across several files in every level, and across several pages within each file. Thus, such operations can only be supported by eagerly performing a full-tree compaction. This stalls all write operations, causing huge latency spikes. The cost incurred by a secondary range delete depends on the total number of data pages on disk, and is independent of the selectivity of the range delete operation. Irrespective of the merging strategy, this cost is $O(N/B)$, where B is the page size.

4 PERSISTING DELETES: *LETHE*$^+$

Design Goals. *Lethe*$^+$ aims (i) to provide persistence guarantees for point and range deletes and (ii) to enable practical secondary range deletes. We achieve the first design goal by introducing FADE, a family of delete-aware compaction strategies. We achieve the second goal by introducing Key Weaving Storage Layout, a new continuum of physical data layouts that arranges entries on disk in an interweaved fashion based on both the sort and the delete key. *Lethe*$^+$ achieves these goals using a minimal amount of extra metadata, without hurting overall performance, while offering better resource utilization in the presence of deletes.

4.1 FADE

We first introduce the *FADE* family of compaction strategies that ensures that all tombstones are persisted within a delete persistence threshold (D_{th}). D_{th} is typically specified by the application or user [31, 74] as part of the service level agreement (SLA) that concerns the data retention policy. All data streams bound by the same data retention SLA, have the same delete persistence latency.

4.1.1 Overview. Compactions in LSM-trees influence their behavior by dictating their space amplification, write amplification, point and range read performance, and delete persistence latency. FADE uses additional information about the *age of a file's tombstones* and the *estimated invalidated entries per tombstone* to ensure that every tombstone adheres to the user/application-provided D_{th} by assigning to every file a ***time-to-live* (TTL)**. As long as D_{th} is respected, FADE offers different strategies for secondary optimization goals including minimizing write amplification, minimizing space amplification, or maximizing system throughput.

4.1.2 Time-to-Live. To ensure that all delete operations issued on an LSM-tree are persisted before D_{th}, FADE propagates the tombstones through all intermediate levels to the last level within that threshold from its insertion. FADE achieves this by assigning a smaller TTL, d_i, for every file

in every Level i, such that $\sum_{i=1}^{L-1} d_i = D_{th}$ (we do not have TTL-based flush for the memtable). By assigning a TTL for the tombstones of a level, FADE may need to trigger compactions proactively based on this TTL. We highlight that *the number of compactions triggered by FADE depends on how we break down the delete persistence threshold (D_{th}) into the level-TTLs (d_i).* A naïve way to allocate TTL to the levels in a tree is to use $d_i = D_{th}/(L-1)$. While this may guarantee that a tombstone reaches the last level within D_{th}, it also leads to increased compaction time and resource starvation as larger levels have exponentially more files; hence, a large number of files may exhaust their TTL simultaneously. While this may be facilitated through simultaneous compactions using thread- and storage-level parallelism, in practice, the degree of this parallelism is bounded and limited by the number of physical computing cores available in a system. Moreover, performing a high number of compactions concurrently (i) saturates the device bandwidth and (ii) stalls ongoing operations for long durations, thus increasing tail latency. Hence, to avoid needing a high number of concurrent compactions for timely deletion, we assign exponentially increasing TTLs to the tree levels moving downward, and by doing so, we guarantee that the number of files exhausting the respective TTL remains uniform over time. Below, we prove that this exponential assignment is indeed optimal in the sense that it minimizes the number of compactions triggered, effectively, reducing the amount of extra work performed while ensuring timely delete persistence.

Determining the Optimal Allocation Strategy for d_i. We formulate finding the optimal d_i for every level of a tree as an optimization problem. We assume an LSM-tree with L levels on disk that uses partial compaction, and all intermediate disk levels are always nearly saturated (i.e., adding one additional file would cause a level to be saturated). Further, we assume a uniform rate of ingestion to the database which fills up the LSM-buffer every c time units, triggering a buffer flush. This means that for an LSM-tree with partial compaction (where every level is always *almost* saturated), a buffer flush every c time units triggers a cascade of compactions, moving one file from each of those levels (except the last one) to the next one. Now, assume that at steady state, Level i contains f_i files ($0 < i < L$), with $f_i = f_1 \cdot T^{i-1}$. Among these, g_i ($g_i \leq f_i$) files are assumed to have the same age for the oldest tombstone contained in them, and thus, are expected to exhaust the level TTL (d_i) at the same time. Assuming uniform distribution on the ingested keys, we have $g_i = g_1 \cdot T^{i-1}$, $\forall 0 < i < L$. This is because each file from Level i is expected to overlap with T files in Level $i+1$ in terms of the key domain. All tombstones ingested to the LSM-tree are bound by the same delete persistence threshold, D_{th}.

Now the goal of our optimization is to distribute D_{th} among the $L-1$ disk levels so that the number of compactions triggered by FADE (and not by level saturation) is minimized while ensuring all tombstones reach the last level (and are persistently removed) within D_{th}. The objective function can be expressed as follows.

$$\begin{aligned} &arg \min_{d_1,\ldots,d_{L-1}} \sum_{i=1}^{L-1} \mathbb{E}\left[\#\text{compactions triggered by FADE in Level } i\right] \\ &s.t. \sum_{i=1}^{L-1} d_i = D_{th}, d_i \geq 0,\ \forall i \in 1, 2, \ldots, L-1 \end{aligned} \tag{1}$$

For each Level i, the number of compactions triggered internally by the LSM-tree due to saturation within the level-TTL d_i is d_i/c. During compaction, a file from the saturated level is chosen uniformly and randomly, and thus, for a file that contains a tombstone with timestamp $t \leq d_i$, the probability that is chosen is $1/f_i$. However, if it is not chosen after d_i/c compactions, FADE will trigger a TTL-based compaction that would move the specific file to Level $i+1$. We can treat it as a Bernoulli distribution with probability $(1-1/f_i)^{d_i/c}$. Now, for the g_i similar files in Level i, the

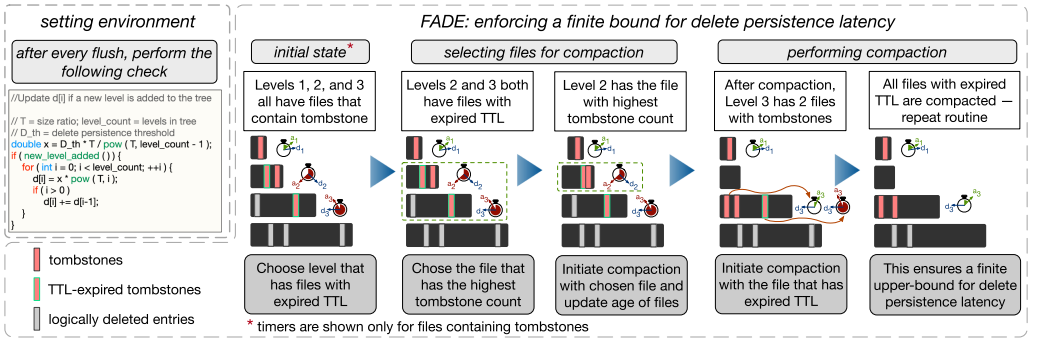

Fig. 5. FADE persists tombstones within the delete persistence threshold, thus improving overall performance.

expected number of TTL-driven compactions triggered by FADE is given by $g_i \cdot (1-1/f_i)^{d_i/c}$ based on the linearity of expectation. In other words, we can transform the above objective function as follows.

$$arg \min_{d_1,\ldots,d_{L-1}} \sum_{i=1}^{L-1} g_i \cdot \left(1 - \frac{1}{f_i}\right)^{d_i/c} = arg \min_{d_1,\ldots,d_{L-1}} \sum_{i=1}^{L-1} g_1 \cdot T^{i-1} \cdot \left(1 - \frac{1}{f_1 \cdot T^{i-1}}\right)^{d_i/c}$$

Neglecting higher order terms, we approximate $(1 - \frac{1}{f_1 \cdot T^{i-1}})^{d_i/c}$ as $e^{\frac{-d_i/c}{f_1 \cdot T^{i-1}}}$. Further, since g_1 is a constant positive integer, we can write down the Lagrange multiplier as below. We define $L(d_1, \ldots, d_{L-1}; \lambda)$ as follows:

$$L(d_1, \ldots, d_{L-1}; \lambda) = \sum_{i=1}^{L-1} T^{i-1} \cdot e^{\frac{-d_i/c}{f_1 \cdot T^{i-1}}} + \lambda \cdot \left(\sum_{i=1}^{L-1} d_i - D_{th}\right)$$

Next, we take the partial derivative in terms of d_i and equating it to 0, we have:

$$T^{i-1} \cdot e^{\frac{-d_i/c}{f_1 \cdot T^{i-1}}} \cdot \left(-\frac{c}{f_1 \cdot T^{i-1}}\right) + \lambda = -\frac{c}{f_1} \cdot e^{\frac{-d_i/c}{f_1 \cdot T^{i-1}}} + \lambda = 0$$

Observe that λ, f_1, c, T are constant, and as this equation holds for every d_i, we further have:

$$\frac{f_1}{c} \cdot \lambda = e^{\frac{-d_1}{c \cdot f_1 \cdot T^0}} = e^{\frac{-d_2/c}{c \cdot f_1 \cdot T^1}} = \cdots = e^{\frac{-d_{L-1}/c}{c \cdot f_1 \cdot T^{L-2}}}$$

which indicates that d_i/T^{i-1} should be a constant, and thus, d_i should grow by a factor of T. This outlines the optimal allocation strategy for D_{th} adopted by FADE.

Updating d_i. For a given tree height, every file is assigned a TTL depending on the level it is being compacted into. As more data entries are inserted, the tree might grow in height. At that point, the TTLs throughout the tree have to be updated. Note that d_i is updated only when a new level is added to the tree, and as the tree grows in height, updating d_i becomes exponentially less frequent. FADE only performs a lightweight check after every buffer flush to check if the height of the tree has increased. Figure 5 shows how to update d_i when a new level is added.

4.1.3 FADE Metadata. Tombstones for point deletes are stored along with valid key-value pairs, and range tombstones are stored in a separate block. In addition to the tombstones, FADE requires the values of two metrics per file: (i) the *age* of the oldest tombstone contained (a^{max}) and (ii) the *estimated invalidation count* (b) of the file tombstones. After every flush, a file is assigned with its current a^{max} and b.

In practice, LSM-engines store file metadata, including (i) the file creation timestamp and (ii) the distribution of the entries per file in the form of a histogram. For example, RocksDB assigns a monotonically increasing insertion-driven *sequence number* (`seqnum`) to all entries, and stores the number of entries (`num_entries`) and point tombstones (`num_deletes`) for every file. FADE takes advantage of this existing metadata. It uses `seqnum` to compute a^{max} and uses `num_entries` and `num_deletes` to compute b. Thus, in practice, FADE has no additional metadata footprint.

Computing a^{max}. The a^{max} of a file f, termed a_f^{max}, is the age of the oldest (point or range) tombstone contained in a file, and is calculated using the difference between the current system time and time the oldest tombstone of that file was inserted in the memory buffer. File without tombstones have $a_f^{max} = 0$. Storing a_f^{max} requires one timestamp (8 bytes) per file, a negligible overhead.

Computing b. The estimated number of invalidated entries by the tombstones of a file f, termed b_f, is calculated using (i) the exact count of point tombstones in the file (p_f) and (ii) an estimation of the entries of the entire database invalidated by the range tombstones of the file (rd_f), as $b_f = p_f + rd_f$. It is not possible to accurately calculate rd_f without accessing the entire database, hence, we *estimate* this value using the system-wide histograms that are already maintained by the data store. The value of b_f is computed on the fly without needing any additional metadata. For a key-domain bounded in $[K^{min}, K^{max}]$, we have $\int_{K^{min}}^{K^{max}} \varphi(x)dx = 1$, where $\varphi(x)$ gives the probability density function for the keys inserted. We assume $\varphi(x)$ remains the same throughout the whole tree. We estimate the number of keys in a tree within a range (r^{min}, r^{max}) by $N' \cdot \int_{r^{min}}^{r^{max}} \varphi(x)dx$, where N' denotes the cumulative number of entries in levels $i+1$ through L. Thus, for a file in Level 1 that contains q non-overlapping range tombstones, the number of entries to be deleted by the range tombstones is approximated by $N' \cdot \sum_{j=1}^{q} \int_{r_j^{min}}^{r_j^{max}} \varphi(x)dx$. Therefore, as a file gets compacted and pushed further down the tree, the number of entries that are affected by a range tombstone decreases exponentially with the tree-level. For a file f in Level i, Equation (2) estimates b_f.

$$b_f^{(i)} = p_f + N \cdot \left(1 - \frac{1}{T^{L-i}}\right) \cdot \sum_{j=1}^{q} \int_{r_j^{min}}^{r_j^{max}} \varphi(x)dx \tag{2}$$

However, in practice, this function may vary across levels due to the presence of point deletes, range deletes, and secondary range deletes in a workload. In Equation (3), we estimate $b_f^{(i)}$ using only the number of logically valid keys in the tree. In practice, we track the number of files per level, and we just estimate $b_f^{(i)}$ in a bottom-up manner starting from the last level.

$$b_f^{(i)} = p_f + \left(N \cdot \left(1 - \frac{1}{T^{L-i}}\right) - \sum_{k=i+1}^{L} (\text{\#files in level k}) \cdot b_f^{(k)}\right) \cdot \frac{T-1}{T} \cdot \sum_{j=1}^{q} \int_{r_j^{min}}^{r_j^{max}} \varphi(x)dx \tag{3}$$

Updating a^{max} and b. Similarly to all file metadata, a^{max} and b are first computed when a file is created after a buffer flush. Thereafter, for newly compacted files, a^{max} and b are recomputed before they are written back to disk. When a compaction simply moves a file from one disk level to the next without physically sort-merging (i.e., when there are no overlapping keys), b remains unchanged and a^{max} is recalculated based on the time of the latest compaction. Note that since all metadata is in memory, this does not cause an I/O.

4.1.4 Compaction Policies. Compactions ensure that both insert and read costs are amortized. For every compaction, there are two policies to be decided: the *compaction trigger policy* and the *file selection policy*. State-of-the-art LSM-engines initiate a compaction when a level is saturated

(i.e., larger than a nominal size threshold) and either pick a file at random, or the one with the smallest overlap with the subsequent level to minimize the merging cost.

Compaction Trigger. FADE augments the state-of-the-art by triggering a compaction, not only when a level is saturated, but also when a file has an expired TTL. FADE triggers a compaction in a level that has at least one file with expired TTL regardless of its saturation. If no TTL has expired, but a level is saturated, a compaction in that level is triggered.

File Selection. FADE decides *which* files to compact based on the trigger that invoked it. It has three modes: (i) the ***saturation-driven trigger and overlap-driven file selection*** **(SO)**, which is similar to the state of the art and optimizes for write amplification, (ii) the ***saturation-driven trigger and delete-driven file selection*** **(SD)**, which selects the file with the highest b to ensure that as many tombstones as possible are compacted and to optimize for space amplification, and (iii) the ***delete-driven trigger and delete-driven file selection*** **(DD)**, which selects a file with an expired tombstone to adhere to D_{th}. A tie in SD and DD is broken by picking the file that contains the oldest tombstone, and a tie in SO by picking the file with the most tombstones. In case of a tie among levels, the smallest level is chosen for compaction to avoid write stalls during compaction. For a tie among files of the same level, FADE chooses the file with the most tombstones.

4.1.5 Implications on Performance. We now analyze the impact and cost of FADE by quantifying its implications on the performance of the storage engine. FADE guarantees that all point and range tombstones are persisted by the time their lifetime reaches D_{th} ($\forall f, a_f^{max} < D_{th}$). We refer to the size of the tree as N and to the size of the tree that has all entries persisted within D_{th} as N_δ.

Assume that N entries including tombstones corresponding to δ_p point deletes, and δ_r range deletes with an average selectivity σ, are inserted to a tree over an arbitrary duration of $d^* \geq D_{th}$. FADE guarantees that after d^*, all point and range deletes $d^* - D_{th}$ duration prior to the current time are persisted. N_δ denotes the total number of entries present the tree after d^*, and as $d^* \to \infty$ and $d^* >> D_{th}$, $N_\delta \to N - 2\delta_p - (1 + \sigma) \cdot \delta_r$.

Space amplification. FADE removes tombstones and logically invalidated entries from the tree on a rolling basis by compacting them in a time-bound fashion. By doing so, it diminishes the space amplification caused by out-of-place deletes, limiting s_{amp} to $O(1/T)$ for leveling and $O(T)$ for tiering, even for a workload with deletes.

Write amplification. Ensuring delete persistence within D_{th}, forces compactions on files with expired TTLs. Therefore, during a workload execution, initially FADE leads to momentary spikes in write amplification. The initial high write amplification, however, is amortized over time. By eagerly compacting tombstones, FADE purges most invalidated entries. Thus, future compactions involve fewer invalidated entries, leading to smaller write amplification which is comparable to the state of the art, as we show in Section 5. For an update-heavy workload a single delete invalidates several entries, and persisting a tombstone faster reduces the superfluous disk I/Os resulting from re-writing of the invalidated entries.

Read performance. FADE has a marginally positive effect on read performance. By compacting invalidated entries and point tombstones, FADE reduces the number of entries hashed in the BFs, leading to smaller overall FPR for a given amount of available memory, hence, the cost for point lookups on existing and non-existing keys is improved asymptotically (Table 2). In the case that N_δ entries can be stored in $L_\delta < L$ levels on disk, the lookup cost will benefit by accessing fewer levels. Long range lookup cost is driven by the selectivity of the query, and this cost is lower for FADE as timely persistence of deletes causes the query iterator to scan fewer invalidated entries.

Table 2. Comparative Analysis of State of the Art and FADE (▲ = better, ▼ = worse, ● = same, ♦ = tunable)

Metric	State-of-the-art [26, 28]		FADE				KiWi				*Lethe*$^+$ (with FADE and KiWi$^+$ integrated)			
	Leveling	Tiering	Leveling		Tiering		Leveling		Tiering		Leveling		Tiering	
Entries in tree	$O(N)$	$O(N)$	$O(N_\delta)$	▲	$O(N_\delta)$	▲	$O(N)$	●	$O(N)$	●	$O(N_\delta)$	▲	$O(N_\delta)$	▲
Space amplification without deletes	$O(1/T)$	$O(T)$	$O(1/T)$	●	$O(T)$	●	$O(1/T)$	●	$O(T)$	●	$O(1/T)$	●	$O(T)$	●
Space amplification with deletes	$O(\frac{(1-\lambda)\cdot N+1}{\lambda \cdot T})$	$O(\frac{N}{1-\lambda})$	$O(1/T)$	▲	$O(T)$	▲	$O(\frac{(1-\lambda)\cdot N}{\lambda \cdot T})$	●	$O(\frac{N}{1-\lambda})$	●	$O(1/T)$	▲	$O(T)$	▲
Total bytes written to disk	$O(N \cdot E \cdot L \cdot T)$	$O(N \cdot E \cdot L)$	$O(N_\delta \cdot E \cdot L_\delta \cdot T)$	▲	$O(N_\delta \cdot E \cdot L_\delta)$	▲	$O(N \cdot E \cdot L \cdot T)$	●	$O(N \cdot E \cdot L)$	●	$O(N_\delta \cdot E \cdot L_\delta \cdot T)$	▲	$O(N_\delta \cdot E \cdot L_\delta)$	▲
Write amplification	$O(L \cdot T)$	$O(L)$	$O(L \cdot T)$	●	$O(L)$	●	$O(L \cdot T)$	●	$O(L)$	●	$O(L \cdot T)$	●	$O(L)$	●
Delete persistence latency	$O(\frac{T^{L-1}\cdot P\cdot B}{I})$	$O(\frac{T^{L}\cdot P\cdot B}{I})$	$O(D_{th})$	▲	$O(D_{th})$	▲	$O(\frac{T^{L-1}\cdot P\cdot B}{I})$	●	$O(\frac{T^{L}\cdot P\cdot B}{I})$	●	$O(D_{th})$	▲	$O(D_{th})$	▲
Zero result point lookup cost	$O(e^{-m/N})$	$O(e^{-m/N} \cdot T)$	$O(e^{-m/N_\delta})$	▲	$O(e^{-m/N_\delta} \cdot T)$	▲	$O(h \cdot e^{-m/N})$	▼	$O(h \cdot e^{-m/N} \cdot T)$	▼	$O(e^{-m/N_\delta} \cdot \prod_{i=1}^{L} T^{\frac{1}{T^{L-i}}})$	♦	$O(T \cdot e^{-m/N_\delta} \cdot \prod_{i=1}^{L} T^{\frac{1}{T^{L-i}}})$	♦
Non-zero result point lookup cost	$O(1)$	$O(1+e^{-m/N} \cdot T)$	$O(1)$	●	$O(1+e^{-m/N_\delta} \cdot T)$	▲	$O(1+h \cdot e^{-m/N})$	▼	$O(1+h \cdot e^{-m/N} \cdot T)$	▼	$O(1+e^{-m/N_\delta} \cdot \prod_{i=1}^{L} T^{\frac{1}{T^{L-i}}})$	♦	$O(1+T \cdot e^{-m/N_\delta} \cdot \prod_{i=1}^{L} T^{\frac{1}{T^{L-i}}})$	♦
Short range point lookup cost	$O(L)$	$O(L \cdot T)$	$O(L_\delta)$	▲	$O(L_\delta \cdot T)$	▲	$O(h \cdot L)$	▼	$O(h \cdot L \cdot T)$	▼	$O(P \cdot \sum_{i=1}^{L} T^i/h_i)$	♦	$O(P \cdot \sum_{i=1}^{L} T^{i+1}/h_i)$	♦
Long range point lookup cost	$O(\frac{s\cdot N}{B})$	$O(\frac{T\cdot s\cdot N}{B})$	$O(\frac{s\cdot N_\delta}{B})$	▲	$O(\frac{T\cdot s\cdot N_\delta}{B})$	▲	$O(\frac{s\cdot N}{B})$	●	$O(\frac{T\cdot s\cdot N}{B})$	●	$O(\frac{s\cdot N_\delta}{B})$	▲	$O(\frac{T\cdot s\cdot N_\delta}{B})$	▲
Insert/Update cost	$O(\frac{L\cdot T}{B})$	$O(\frac{L}{B})$	$O(\frac{L_\delta\cdot T}{B})$	▲	$O(\frac{L_\delta}{B})$	▲	$O(\frac{L\cdot T}{B})$	●	$O(\frac{L}{B})$	●	$O(\frac{L_\delta\cdot T}{B})$	▲	$O(\frac{L_\delta}{B})$	▲
Secondary range delete cost	$O(N/B)$	$O(N/B)$	$O(N_\delta/B)$	▲	$O(N_\delta/B)$	▲	$O(\frac{N}{B\cdot h})$	▲	$O(\frac{N}{B\cdot h})$	▲	$O(\frac{N_\delta}{B\cdot h})$	♦	$O(\frac{N_\delta}{B\cdot h})$	♦

Persistence Guarantees. FADE ensures that all tombstones inserted into an LSM-tree and flushed to the disk will always be compacted with the last level within the user-defined D_{th} threshold. Any tombstone retained in the **write-ahead log (WAL)** is consistently purged if the WAL is purged at a periodicity that is shorter than D_{th}, which is typically the case in practice. Otherwise, we use a dedicated routine that checks all live WALs that are older than D_{th}, copies all live records to a new WAL, and discards the records in the older WAL that made it to the disk.

In practice, the attained threshold might be $D_{th} + \epsilon$, as *Lethe*$^+$ can trigger TTL-driven compactions only after every buffer flush. Thus, systems use as threshold $D_{th} - \epsilon$ to ensure timely delete persistence. The value of ϵ depends on the frequency of buffer flushes from the memory which is a function of the ingestion rate of entries (or unique entries, if entries are updated in place within the buffer). FADE ensures that all tombstones inserted in a tree are compacted with the last tree-level within a given time duration that is specified by the deletes persistence latency. The key enabling components that allow FADE to achieve this are (i) a TTL for every tree-level that is tuned based on the design of a data store and (ii) a new compaction trigger and file picking policy that greedily compacts files that have an expired TTL. The complete algorithm for FADE is presented in Algorithm 1.

Practical Values for D_{th}. The delete persistence threshold of different applications vary widely. In commercial systems, LSM-engines are forced to perform a full-tree compaction every 7, 30, or 60 days based on the SLA requirements [44].

Blind Deletes. A tombstone against a key that does not exist or is already invalidated causes a *blind delete*. Blind deletes ingest tombstones against keys that do not exist in a tree, and these superfluous tombstones affect the performance of point and range queries [44]. This incurs additional storage overhead, reduces the accuracy of the Bloom filters of the files that will carry this tombstone to the last level during its lifetime, and leads to unnecessary re-writes increasing the amortized write amplification. In addition, excessive tombstones adversely affect range queries, as the range query iterator has to go over all data, including invalid entries and tombstones of the value range, before discarding the unnecessary entries [44].

A simple solution to avoid blind deletes is to perform a point lookup on the target key, and insert a tombstone only if the lookup reports a positive match. This, however, requires at least one I/O for every delete issued on existing key, and up to $O(1 + e^{-m/N})$ for leveling and $O(1 + T \cdot e^{-m/N})$ I/Os for tiering in the average worst-case. To avoid blind deletes, FADE probes the corresponding BFs and inserts a tombstone only if the filter probe returns positive. Thus, FADE may insert only a very small fraction of blind deletes driven by the FPR.

4.2 Key Weaving Storage Layout (KiWi)

To facilitate secondary range deletes, we introduce *KiWi*, a continuum of physical storage layouts that arranges the data on disk in an **interweaved sorted order** on the sort key and delete key.

ALGORITHM 1: FADE

Input: delete persistence latency (D_{th}); levels in tree (L_{old}); size ratio (T); size of memory buffer (M)

```
FADE():
begin
    L_new = getCurrentTreeLevel()
    d_0 = 0
    if L_new > L_old then
        for i ∈ [1 : L_new − 1] do
            d_i = d_{i−1} + D_th · (T − 1)/(T^{L_new−1} − 1) · T^{i−1}
    for i ∈ [1 : L_new − 1] do
        csize(i) = 0, ttl_i = 0, cap_i = M · T^i
        for j ∈ [1 : getFileCountInLevel(i)] do
            csize(i) += size(j)
            if d_i < age_j then
                ttl_i++
        score[i] = csize(i)/cap_i + ttl_i
    compact_level = getLevelToCompact(score[])
    compact_file = getFileToCompact(compact_level)
    initiate compaction with compact_file
getLevelToCompact(score[]):
begin
    c_level = score[0]
    for i ∈ [1 : L_new − 1] do
        if score[i] > score[i − 1] then
            c_level = i
    return c_level
getFileToCompact(compact_level):
begin
    j = 0
    for i ∈ [1 : getFileCountInLevel(compact_level)] do
        if d_{compact_level} ≥ age_i then
            return f[j]
        else
            f_age[j++] = age_i
            f_density[j++] = b_i
    return f[0]
```

KiWi supports secondary range deletes without performing a full-tree compaction, at the cost of minimal extra metadata and a tunable penalty on read performance.

4.2.1 The Layout. Figure 6 presents the internal structure of KiWi. Essentially, KiWi adds one new layer in the storage layout of LSM trees. In particular, in addition to the levels of the tree, the files of a level, and the page of a file, we now introduce *delete tiles* that belong to a file and consist of pages. In the following discussion, we use $\mathcal{S}$ to denote the sort key and $\mathcal{D}$ for the delete key.

Level Layout. The structure of the levels remains the same as that of the state-of-the-art LSM trees. Every level is a collection of files containing non-overlapping ranges of $\mathcal{S}$. The order between files in a level follows $\mathcal{S}$. Formally, if $i < j$, all entries in file i have smaller $\mathcal{S}$ than those in file j.

File Layout. The contents of the file are *delete tiles*, which are collections of pages. Delete tiles contain non-overlapping ranges of $\mathcal{S}$, hence from the perspective of the file, the order of the delete

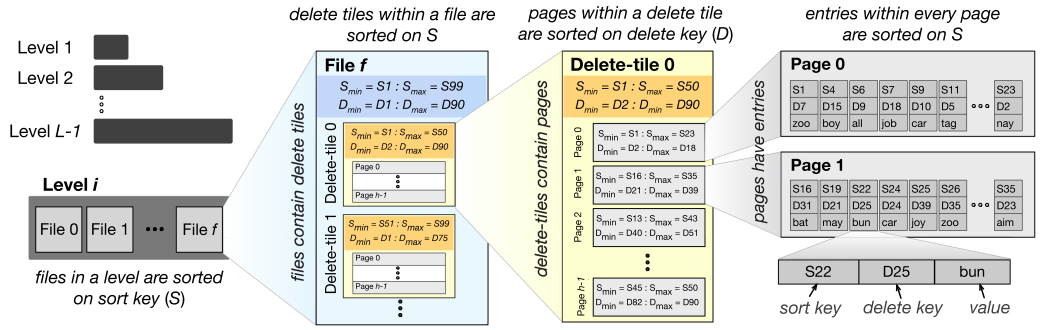

Fig. 6. Key Weaving Storage Layout stores data in an interweaved fashion on the sort and delete key to facilitate efficient secondary range deletes without hurting read performance.

tiles also follows $\mathcal{S}$. Formally, if $k < l$, all entries in delete tile k have smaller $\mathcal{S}$ than those in file l.

Delete Tile Layout. Contrary to the above, the pages of a delete tile are sorted on $\mathcal{D}$. Formally, for $p < q$, page p of a given delete tile contains entries with smaller $\mathcal{D}$ than page q, while we have no information about $\mathcal{S}$. Organizing the contents of a tile ordered on the delete key $\mathcal{D}$ allows us to quickly execute range deletes because the entries under deletion are always clustered within contiguous pages of each tile, which can be dropped in their entirety.

Page layout. The order of entries *within each page* does not affect the performance of secondary range deletes, however, it significantly affects lookup cost, once a page is fetched to memory. To facilitate quick in-memory searches within a page [87], we sort the entries of each page based on $\mathcal{S}$.

4.2.2 Facilitating Secondary Range Deletes. KiWi exploits the fact that within a delete tile, the disk pages are sorted on the delete key. Hence, the entries targeted by a secondary range delete populate contiguous pages of each tile (in the general case of every tile of the tree). The benefit of this approach is that these pages can be dropped without having to be read and updated. Rather, they are removed from the otherwise immutable file and released to be reclaimed by the underlying file system. We call this a *full page drop*. Pages containing entries at the edge of the delete range might also contain some valid entries. These pages are read to memory, and the valid entries are identified with a tight `for`-loop on $\mathcal{D}$ (since they are sorted on $\mathcal{S}$). We call these *partial page drops*. The cost of reading and re-writing these pages is effectively the I/O cost of secondary range deletes with KiWi. In practice, the number of partially dropped pages is limited to one page per delete tile on average. The KiWi algorithm is presented in Algorithm 2.

4.2.3 Tuning and Metadata. We now discuss the tuning knobs and the metadata of KiWi.

Delete Tile Granularity. Every file contains a number of delete tiles, and each tile contains a number of pages. The basic tuning knob of KiWi is the number of pages per delete tile (h), which affects the granularity of delete ranges that can be supported. For a file with P disk pages, the number of delete tiles per file is P/h. The smallest granularity of a delete tile is when it consists of only a single disk page, i.e., for $h = 1$. In fact, $h = 1$ creates the same layout as the state of the art, as all contents are sorted on $\mathcal{S}$ and every range delete needs to update all data pages. As h increases, delete ranges with delete fraction close to $1/h$ will be executed mostly by full drops. On the other hand, for higher h read performance is affected. The optimal decision for h depends on the workload (frequency of lookups and range deletes), and the tuning (memory allocated to BFs and size ratio).

ALGORITHM 2: KiWi

Input: FPR of BFs (ϕ_δ); file size (f_{size}); frequency of empty point queries (f_{EPQ}); frequency of non-empty point queries (f_{PQ}); frequency of short range queries (f_{SRQ}); frequency of secondary range deletes (f_{SRD})

```
KiWi(file f, source level L_i):
begin
    h^opt = GetDeleteTileGranularity()
    select set of files F from target level L_{i+1} overlapping with f
    Merge all keys in F and repartition them as a set of files, F', each of size f_size
    for f_j ∈ F' do
        for every consecutive h^opt entries in f_j do
            sort entries based on delete key D and partition into P pages
            for each page in P do
                sort entries based on sort key S
    persist F' to L_{i+1} on disk
    if L_{i+1} exceeds capacity then
        FADE()
        choose file f_next from level L_{i+1} for compaction
        KiWi(f_next, L_{i+1})
GetDeleteTileGranularity():
begin
    levels = getCurrentTreeLevel()
    pages = getCurrentPageCountInTree()
    h^opt = 1
    if f_SRD ≥ 1 then
        h^opt = ((f_SRD · pages)/(φ_δ · levels · (f_EPQ + f_PQ) + levels · f_SRQ))^0.5
    return round(h^opt)
```

Bloom Filters and Fence Pointers. We next discuss Bloom filters and fence pointers in the context of KiWi.

Bloom filters. KiWi maintains BFs on $\mathcal{S}$ at *the granularity of disk page*. Maintaining separate BFs per page requires no BF reconstruction for full page drops, and light-weight CPU cost for partial page drops. The same overall FPR is achieved with the same memory consumption when having one BF per page, since a delete tile contains no duplicates [13].

Fence pointers. KiWi maintains fence pointers on $\mathcal{S}$ that keep track of the smallest sort key for every delete tile. Fence pointers on $\mathcal{S}$, aided by the BFs, accelerate lookups. To support secondary range deletes, for every delete tile, KiWi maintains in memory a separate fence pointer structure on $\mathcal{D}$. We refer to this as *delete fence pointers*. The delete fence pointers store the smallest $\mathcal{D}$ of every page enabling full page drops of the corresponding pages without loading and searching the contents of a delete tile.

Memory Overhead. While KiWi does not require any additional memory for BFs, it maintains two fence pointer structures – one on $\mathcal{S}$ per delete tile and one on $\mathcal{D}$ per page. Since the state of the art maintains fence pointers on $\mathcal{S}$ per page, the space overhead for KiWi is the additional metadata per tile. Assuming $sizeof(\mathcal{S})$ and $sizeof(\mathcal{D})$ are the sizes in bytes for $\mathcal{S}$ and $\mathcal{D}$ respectively, the space overhead is:

$$\begin{aligned} & Mem_{KiWi} - Mem_{SoA} \\ &= \tfrac{N}{B \cdot h} \cdot sizeof(\mathcal{S}) + \tfrac{N}{B} \cdot sizeof(\mathcal{D}) - \tfrac{N}{B} \cdot sizeof(\mathcal{S}) \\ &= \#delete\_tiles \cdot (sizeof\,(\mathcal{S}) + h \cdot (sizeof\,(\mathcal{D}) - sizeof\,(\mathcal{S}))), \end{aligned}$$

where Mem_{SoA} and Mem_{KiWi} represents the memory overhead due to BFs and fences pointers in the state-of-the-art LSM-engnies and in KiWi, respectively. Note that if $sizeof(\mathcal{S}) = sizeof(\mathcal{D})$, the space overhead for KiWi is only one sort key per delete tile, and if $sizeof(\mathcal{D}) < \frac{h-1}{h} \cdot sizeof(\mathcal{S})$, KiWi could lead to less overall size of metadata.

4.2.4 CPU Overhead. KiWi navigates an intrinsic trade-off between the CPU cost for additional hashing for Bloom filters and the I/O cost associated with data movement to and from disk. For non-zero result point queries, KiWi performs $L \cdot h/2$ times more hash calculations compared to the state of the art, and $L \cdot h$ times in case of zero-result point queries. In practice, commercial LSM-engines, such as RocksDB, use only a single MurmurHash hash digest to calculate which Bloom filter bits to set [59, 91]. This reduces the overall cost of hash calculation by almost one order of magnitude. We measured the time to hash a single 64-bit key using the MurmurHash to be $80ns$, which is significantly smaller than the SSD access latency of $100\mu s$ [66]. This allows *Lethe*$^+$ to strike a navigable trade-off between the CPU and I/O costs, and for the optimal value of h, *Lethe*$^+$ achieves a significantly superior overall performance as compared to the state of the art.

4.2.5 Implications on Performance. KiWi offers a tunable trade-off between the cost of secondary range deletes and that of lookups, but does not influence write performance (including space and write amplifications).

Point Lookup. A point read follows the same search algorithm as in the state of the art [27]. In every level, a lookup searches the fence pointers (on $\mathcal{S}$) to locate the delete tile that may contain the search key. Once a delete tile is located, the BF for each delete tile page is probed. If a probe returns positive, the page is read to memory and binary searched, since the page is sorted on $\mathcal{S}$. If the key is found, the query terminates. If not, the I/O was due to a false positive, and the next page of the tile is fetched. The I/O cost for a query on an existing entry is $O(1 + h \cdot e^{-m/N})$ for leveling and $O(1 + T \cdot h \cdot e^{-m/N})$ for tiering. A zero-result lookup, however, has to read all the pages within a delete tile in the worst case, and the cost for this in leveling and tiering is given by $O(h \cdot e^{-m/N})$ and $O(h \cdot e^{-m/N} \cdot T)$, respectively.

Range Lookup. In general, a range lookup may span several delete tiles spread in one or more consecutive files. KiWi affects the performance of range lookups only at the terminal delete tiles that contain the bounds on the range – all delete tiles in between that are subsumed by the range always need to be read to memory. For the terminal delete tiles, the lookup needs to scan on average $h/2$ more pages per tile instead of only the qualifying pages. Thus, the cost of short range lookups for KiWi becomes $O(h \cdot L)$ for leveling and $O(h \cdot L \cdot T)$ for tiering. For long range lookups, the increase in cost gets amortized over the number of qualifying delete tiles, and remains the same asymptotically, i.e., $O(s \cdot N/B)$ and $O(T \cdot s \cdot N/B)$ for leveling and tiering, respectively.

Secondary Range Lookups. With the interweaved layout, KiWi can also support efficient range lookups on the delete key. While state-of-the-art LSM-engines need to maintain a secondary index on the delete key, they still pay a heavy cost for scanning across many scattered pages. KiWi utilizes the ordering of the data on the delete key and can realize secondary range queries at a much lower I/O cost.

4.2.6 Navigable Design. A fundamental design goal for KiWi is to **navigate the trade-off** between the cost of *secondary range deletes* and *lookups*. KiWi offers a navigable continuum of storage layouts that can be tuned to obtain the optimal value for h based on the workload characteristics and performance requirements. Assuming that the workload can be described by the fractions of (a) empty point queries f_{EPQ}, (b) non-empty point queries f_{PQ}, (c) short range queries f_{SRQ}, (d) long range queries f_{LRQ} with selectivity s, (e) secondary range deletes f_{SRD}, and (f) ingestion

operations (i.e., inserts, updates, and deletes) f_I, we can compare the cost running this workload for *Lethe*$^+$ and the state of the art.

Cost for State of the Art. We plug in the average worst-case cost in Equation (4) based on Table 2 to estimate the I/O cost for running a workload in a state-of-the-art LSM-engine [26–29, 33, 34, 41, 44, 89].

$$Cost_{SoA} = f_{EPQ} \cdot \phi \cdot L + f_{PQ} \cdot (1 + \phi \cdot L) + f_{SRQ} \cdot L + f_{LRQ} \cdot s \cdot N/B + f_{SRD} \cdot N/B + f_I \cdot L \cdot T/B \tag{4}$$

Note that the cost of point queries depends on the FPR of Bloom filters, denoted by ϕ. The value of ϕ depends on the implementation of the Bloom filters, i.e., the amount of memory assigned to the filters, which in turn, affects its FPR. State-of-the-art LSM-based production systems [33, 34, 41, 44, 89] assume a fixed number of memory (bits-per-entry) for all Bloom filters in a tree, and thus, have a fixed FPR for all levels in a tree, given as $\phi = e^{-\frac{bits}{entries} \cdot \ln(2)^2}$. For systems that implement optimal memory distribution by allocating optimal bits-per-entry to the Bloom filters of each level in an LSM-tree [26–29], the FPR for the tree is computed as:

$$\phi = \frac{T^{\frac{T}{T-1}}}{T} \cdot e^{-\frac{bits}{entries} \cdot \ln(2)^2} \tag{5}$$

As these systems do not implement delete tiles (i.e., $h = 1$), the costs do not depend on h.

Cost for KiWi. In contrast, the cost of queries and secondary range deletes in KiWi depends on the delete tile granularity, h, and the overall workload execution cost is computed as shown in Equation (6).

$$Cost_{KiWi} = f_{EPQ} \cdot \phi_\delta \cdot L_\delta \cdot h + f_{PQ} \cdot (1 + \phi_\delta \cdot L_\delta \cdot h) + f_{SRQ} \cdot L_\delta \cdot h + f_{LRQ} \cdot s \cdot N_\delta/B + f_{SRD} \cdot N_\delta/(B \cdot h) + f_I \cdot L_\delta \cdot T/B \tag{6}$$

As discussed in Section 4.2.5, in the worst case, the cost of empty point lookups and short range queries increases by a factor of h, whereas that for secondary range deletes diminishes by a factor of h, in the worst case. ϕ_δ denotes the false positive rate of the BFs when the logically invalidated entries are purged in a time-bound manner (i.e., within D_{th}) by FADE. The cost for non-empty point lookups is influenced marginally in the presence of BFs as lookups on existing keys must always perform one disk I/O to fetch the page containing the target entry. KiWi can support any Bloom filter implementation.

Optimal Delete Tile Granularity for KiWi. For workloads with secondary range deletes, KiWi can tune the storage layout to find the optimal value for the delete tile granularity. This is achieved by using the frequency of point and short range read operations relative to the frequency of secondary range deletes, as the cost of ingestion and long range queries is not a function of h. In order to find the optimal value of h, we minimize the cost of KiWi. We begin by re-writing Equation (6) as follows.

$$\begin{aligned} Cost_{KiWi} &= h \cdot [\phi_\delta \cdot L_\delta \cdot (f_{EPQ} + f_{PQ}) + f_{SRQ} \cdot L_\delta] + \frac{1}{h} \cdot \frac{N_\delta}{B} \cdot f_{SRD} + \frac{N_\delta}{B} \cdot s \cdot f_{LRQ} + \frac{T}{B} \cdot L_\delta \cdot f_I \\ &= Q_1 \cdot h + Q_2/h + C_1 \end{aligned} \tag{7}$$

where $Q_1 = \phi_\delta \cdot L_\delta \cdot (f_{EPQ} + f_{PQ}) + f_{SRQ} \cdot L_\delta$ and $Q_2 = N_\delta/B \cdot f_{SRD}$ are the quotients of the terms depending on h and $C_1 = N_\delta/B \cdot s \cdot f_{LRQ} + T/B \cdot L_\delta \cdot f_I$ is the sum of the remaining terms independent of h. Next, we differentiate Equation (7) with respect to h to minimize the cost.

$$\frac{\partial Cost_{KiWi}}{\partial h} = Q_1 - Q_2/h^2 \tag{8}$$

The minimum cost that leads to the optimal delete tile granularity, h^*, is given by:

$$h^* = \sqrt{Q_2/Q_1} = \left(\frac{f_{SRD} \cdot N_\delta / B}{\phi_\delta \cdot L_\delta \cdot (f_{EPQ} + f_{PQ}) + L_\delta \cdot f_{SRQ}} \right)^{1/2} \tag{9}$$

The optimal delete tile granularity computed based on Equation (9) is rounded off to its nearest integer with the bounds for h^* being $[1, P]$, where P is the number of pages in a file. Equation (9) also shows that the data layout proposed by KiWi depends only on (i) the proportions of lookups and secondary range deletes in a workload, (ii) the number of entries in a tree (levels) and (iii) in each disk page, (iv) the size ratio of the tree, and (v) the FPR of the Bloom filters. Thus, for a fixed LSM tuning, KiWi is only affected by three workload parameters, the frequency of point lookups, short range queries, and secondary range delete parameters. *The Pareto frontier for KiWi is constructed by the relative proportion of (short) reads and secondary range deletes in a workload.*

4.3 Level-Wise Key Weaving Storage Layout (KiWi$^+$)

We now go one step further than KiWi to push the Pareto frontier constructed by the costs of (short) reads and secondary range deletes. We introduce *KiWi*$^+$ that applies KiWi on a per-level basis to find the optimal data layout for each level of an LSM-tree. KiWi$^+$ takes into account that the non-empty point lookups may *terminate early*, i.e., at any intermediate level of a tree, and further enhances the non-empty point lookup performance while retaining all the benefits of KiWi.

4.3.1 The KiWi$^+$ Layout. The data layout principles for KiWi$^+$ remain the same as that of KiWi, as shown in Figure 6. However, unlike KiWi, where the number of pages in each delete tile (h^*) is the same regardless of which tree-level the tile belongs to, the delete tile granularity for each level may be different in KiWi$^+$, subject to the (specification and temporality of the) workload and LSM tuning. The intuition is that the capacity of shallower levels in a tree is exponentially decreasing, and thus their impact of the secondary range lookup performance is significantly smaller the impact of the larger levels. Moreover, for workloads with even a small degree of temporality, the expected number of non-empty point lookups terminating early at a shallower level is significantly higher. Driven by these LSM-properties, KiWi$^+$ optimizes the overall performance of an LSM-tree by finding the optimal delete tile granularity for each level separately.

Delete Tile Granularity. The basic tuning knob for KiWi$^+$ is h_i, which denotes the delete tile granularity for Level i in a tree. The number of delete tiles in a file in Level i, therefore, becomes P/h_i, and $h_i \in [1, P]$. The intuition is that shallower levels, which hold fewer entries, are expected to serve a relatively larger proportion of non-empty point lookups, and thus, the delete tile granularity will be smaller. A smaller h_i for shallower levels will boost the point lookup performance significantly for both empty and non-empty queries, and its impact on the secondary range lookup performance is expected to be marginal. The complete algorithm for KiWi$^+$ is presented in Algorithm 3.

4.3.2 Memory Overhead. The Bloom filter implementation for KiWi$^+$ remains the same as that in the state of the art (and KiWi). However, by having a smaller delete tile granularity for shallower levels, KiWi$^+$ reduces the memory overhead due to fence pointers. In particular, as KiWi$^+$ has fewer delete tiles overall as compared to KiWi, it reduces the memory required for delete fence pointers which are maintained at the granularity of delete tile.

$$Mem_{KiWi^+} - Mem_{SoA} = P \cdot \sum_{i=1}^{L} \frac{T^i}{h_i} \cdot sizeof(\mathcal{S}) + \frac{N}{B} \cdot sizeof(\mathcal{D}) - \frac{N}{B} \cdot sizeof(\mathcal{S}) \tag{10}$$

where Mem_{SoA} and Mem_{KiWi^+} represent the memory overhead due to BFs and fences pointers for the state of the art and KiWi, respectively. In practice, as KiWi$^+$ has fewer delete tiles overall as compared to KiWi, it reduces the memory required for delete fence pointers which are maintained at the granularity of delete tile.

4.3.3 CPU Overhead. KiWi$^+$ reduces the CPU overhead by reducing the average number of Bloom filter probes performed for point lookups. For shallower levels with smaller delete tile granularity, we expect the amount of filter probes per point lookup to reduce in KiWi$^+$ as compared with KiWi. Theoretically, KiWi$^+$ performs $\sum_{i=1}^{L} h_i/2$ disk I/Os on average for non-empty point lookups and twice that number for empty point lookups.

4.3.4 Implications on Performance. While the ingestion and long range lookup performance of KiWi$^+$ remains the same as those of KiWi, KiWi$^+$ strikes a better balance between the cost of secondary range deletes and that of (short) lookups.

Point Lookup. Point lookups are realized significantly faster in KiWi$^+$ while performing few disk I/Os. With different delete tile granularity at different levels of a tree, the average worst-case cost for non-empty point lookups in KiWi$^+$ is given as $O(1 + \sum_{i=1}^{L-1} h_i \cdot \phi_i)$ for leveling and $O(1 + T \cdot \sum_{i=1}^{L-1} h_i \cdot \phi_i)$ for tiering. For empty point lookups this cost is given as $O(\sum_{i=1}^{L} h_i \cdot \phi_i)$ for leveling and $O(T \cdot \sum_{i=1}^{L} h_i \cdot \phi_i)$ for tiering.

Range Lookup. The long range lookup performance for KiWi$^+$ remains the same as that for the state of the art (and KiWi). However, the short range lookup performance is affected by the delete tile granularity, as a short range lookup may have to read up to h_i pages per level on average before constructing the result set. Thus, the cost for short range lookups for KiWi$^+$ becomes $O(\sum_{i=1}^{L} h_i)$ for leveling and $O(T \cdot \sum_{i=1}^{L} h_i)$ for tiering.

KiWi$^+$ is essentially a design superset of KiWi, hence, when tuned optimally based on the workload characteristics, KiWi$^+$ offers better overall performance compared to KiWi.

4.3.5 Navigating the Trade-Off with KiWi$^+$. With the level-wise key interweaved layout, KiWi$^+$ can navigate the trade-off between (short) lookups and secondary range deletes elegantly while improving the overall performance of an LSM-based storage engine. KiWi$^+$ offers a navigable continuum of storage layouts that can be tuned to extract maximum performance from LSM-based storage engines if the workload characteristics and performance requirements are known *a priori*. Below, we show how we obtain the level-wise optimal delete tile granularity with KiWi$^+$ and how that compares against the state of the art.

Cost for KiWi$^+$. The costs of point and short range lookups and that of secondary range deletes in KiWi$^+$ depend on the delete tile granularity for each level of the tree. Equation (11) shows the expected cost (in terms of disk I/Os) for Level i ($1 \leq i \leq L$) of a tree while executing a workload.

$$\begin{aligned} Cost_{KiWi_i^+} = f_{EPQ} \cdot \phi_\delta \cdot h_i + f_{PQ} \cdot [E_i \cdot (1 + \phi_\delta \cdot (h_i - 1)/2) + (1 - E_i - E_i') \cdot \phi_\delta \cdot h_i] \\ + f_{SRQ} \cdot h_i + f_{LRQ} \cdot s \cdot P \cdot T^i + f_{SRD} \cdot P \cdot T^i/h_i + f_I \cdot T/B \end{aligned} \tag{11}$$

where E_i and E_i' are defined as follows. A non-empty point lookup terminates in Level i with probability E_i; in a level earlier than Level i with probability E_i', and with probability $E_i'' = 1 - E_i - E_i'$ in levels following Level i. The total workload executing cost can be estimated as $\sum_{i=1}^{L} Cost_{KiWi_i^+}$.

Computing the Probability of a Query Terminating in a Given Level. For a workload that has the same ingestion and point lookup distributions, the probability terms E_i, E_i', and E_i'' can be estimated as follows.

$$E_i = \frac{T^i}{\sum_{j=0}^{L} T^j} \qquad E_i' = \sum_{k=0}^{i-1} E_k = \frac{\sum_{j=0}^{i-1} T^j}{\sum_{j=0}^{L} T^j}$$

ALGORITHM 3: KiWi$^+$

Input: FPR of BFs (ϕ_δ); file size (f_{size}); size ratio of the tree (T); frequency of empty point queries (f_{EPQ}); frequency of non-empty point queries (f_{PQ}); frequency of short range queries (f_{SRQ}); frequency of secondary range deletes (f_{SRD})

```
KiWiPlus(file f, source level L_i):
begin
    h_i^opt = GetDeleteTileGranularityPlus(i + 1)
    select set of files F from target level L_{i+1} overlapping with f
    Merge all keys in F and repartition them as a set of files, F', each of size f_size
    for f_j ∈ F' do
        for every consecutive h_i^opt entries in f_j do
            sort entries based on delete key D and partition into P pages
            for each page in P do
                sort entries based on sort key S
    persist F' to L_{i+1} on disk
    if L_{i+1} exceeds capacity then
        FADE()
        choose file f_next from level L_{i+1} for compaction
        KiWiPlus(f_next, L_{i+1})
GetDeleteTileGranularityPlus(i):
begin
    h_i^opt = 1
    if f_SRD ≥ 1 then
        E_i = T^i / Σ_{j=0}^{L} T^j
        E_i'' = Σ_{j=i+1}^{L} T^j / Σ_{j=0}^{L} T^j
        h_i^opt = ((f_SRD · P · T^i)/(φ_δ · (f_EPQ + f_PQ · E_i'') + φ_δ · f_PQ · E_i/2 + f_SRQ))^0.5
    return round(h_i^opt)
```

For a tree with partial compactions, as all levels are always nearly saturated, the number of entries in Level i is T times larger than that in Level $i-1$ ($1 \le i \le L$). Thus, E_i and E'_i can be estimated by the fraction of data contained in Level i and the cumulative fraction of data contained in Level 0 through Level $i-1$, respectively. The probability the target entry is found at a level larger than Level i, i.e., E''_i, can be simply computed as complementary probability of $E_i + E'_i$, as shown below.

$$E''_i = 1 - E_i - E'_i = 1 - \frac{T^i}{\sum_{j=0}^{L} T^j} - \frac{\sum_{j=0}^{i-1} T^j}{\sum_{j=0}^{L} T^j} = 1 - \frac{T^i + \sum_{j=0}^{i-1} T^j}{\sum_{j=0}^{L} T^j} = \frac{\sum_{j=0}^{L} T^j - T^i - \sum_{j=0}^{i-1} T^j}{\sum_{j=0}^{L} T^j} = \frac{\sum_{j=i+1}^{L} T^j}{\sum_{j=0}^{L} T^j}$$

Optimal Delete Tile Granularity for KiWi$^+$. In order to find the optimal value of h_i for each level, we minimize the per-level cost of KiWi$^+$. We begin by re-writing Equation (11) as follows.

$$\begin{aligned} Cost_{KiWi^+_i} &= f_{EPQ} \cdot \phi_\delta \cdot h_i + f_{PQ} \cdot [E_i \cdot (1 + \phi_\delta \cdot (h_i - 1)/2) + (1 - E_i - E'_i) \cdot \phi_\delta \cdot h_i] \\ &\quad + f_{SRQ} \cdot h_i + f_{LRQ} \cdot s \cdot P \cdot T^i + f_{SRD} \cdot P \cdot T^i / h_i + f_I \cdot T/B \\ &= h_i \cdot [f_{EPQ} \cdot \phi_\delta + f_{PQ} \cdot E''_i \cdot \phi_\delta + f_{PQ} \cdot E_i \cdot \phi_\delta/2 + f_{SRQ}] + f_{SRD} \cdot P \cdot T^i / h_i \\ &\quad + f_{LRQ} \cdot s \cdot P \cdot T^i + f_I \cdot T/B \\ &= Q_3 \cdot h_i + Q_4/h_i + C_2 \end{aligned} \tag{12}$$

where $Q_3 = f_{EPQ} \cdot \phi_\delta + f_{PQ} \cdot E''_i \cdot \phi_\delta + f_{PQ} \cdot E_i \cdot \phi_\delta/2 + f_{SRQ}$ and $Q_4 = f_{SRD} \cdot P \cdot T^i$ are the quotients of the terms depending on h_i and $C_2 = f_{LRQ} \cdot s \cdot P \cdot T^i + f_I \cdot T/B$ is the sum of the terms that are

independent of h_i. Differentiate Equation (7) with respect to h_i, we get:

$$\frac{\partial Cost_{KiWi_i^+}}{\partial h_i} = Q_3 - Q_4/h_i^2 \tag{13}$$

The optimal delete tile granularity for Level i, h_i^*, is computed as follows.

$$h_i^* = \sqrt{Q_4/Q_3} = \left(\frac{f_{SRD} \cdot P \cdot T^i}{\phi_\delta \cdot (f_{EPQ} + f_{PQ} \cdot E_i'') + \phi_\delta \cdot f_{PQ} \cdot E_i/2 + f_{SRQ}}\right)^{1/2} \tag{14}$$

4.4 Extending KiWi and KiWi$^+$ for Tiered LSMs

We now show how we can apply the interweaved data layout of KiWi and KiWi$^+$ to tiered LSM-tree variants. For this, we update the costs of point lookup, range lookup, and data ingestion, based on Table 2. The optimality analysis stays almost the same as that of a leveled LSM-tree.

4.4.1 KiWi for Tiered LSMs. First, we present the analysis for KiWi, where we have a fixed delete tile granularity (h) across all levels of the tree. The costs of running the workload with a state-of-the-art tiered LSM-tree and for a tiered LSM-tree with KiWi data layout are given by Equation (15) and (16), respectively.

$$Cost_{SoA} = f_{EPQ} \cdot T \cdot \phi + f_{PQ} \cdot (1 + T \cdot \phi) + f_{SRQ} \cdot T \cdot L + f_{LRQ} \cdot T \cdot s \cdot N/B + f_{SRD} \cdot N/B + f_I \cdot L/B \tag{15}$$

$$\begin{aligned} Cost_{KiWi} = {} & f_{EPQ} \cdot T \cdot \phi_\delta \cdot h + f_{PQ} \cdot T \cdot (1 + \phi_\delta \cdot h) + f_{SRQ} \cdot T \cdot L_\delta \cdot h + f_{LRQ} \cdot T \cdot s \cdot N_\delta/B \\ & + f_{SRD} \cdot N_\delta/(B \cdot h) + f_I \cdot L_\delta/B \end{aligned} \tag{16}$$

Re-arranging Equation (16), we have:

$$\begin{aligned} Cost_{KiWi} &= h \cdot [\phi_\delta \cdot T \cdot (f_{EPQ} + f_{PQ}) + f_{SRQ} \cdot T \cdot L_\delta] + \frac{1}{h} \cdot \frac{N_\delta}{B} \cdot f_{SRD} \\ &\quad + \frac{N_\delta}{B} \cdot s \cdot T \cdot f_{LRQ} + \frac{1}{B} \cdot L_\delta \cdot f_I + f_{PQ} \cdot T \\ &= Q_1 \cdot h + Q_2/h + C_1 \end{aligned} \tag{17}$$

where $Q_1 = \phi_\delta \cdot T \cdot (f_{EPQ} + f_{PQ}) + f_{SRQ} \cdot T \cdot L_\delta$ and $Q_2 = N_\delta/B \cdot f_{SRD}$ are the quotients of the terms dependent on h and $C_1 = N_\delta/B \cdot s \cdot T \cdot f_{LRQ} + 1/B \cdot L_\delta \cdot f_I + f_{PQ} \cdot T$. Next we re-apply the optimization technique in Equation (9) to obtain the optimal delete tile granularity (h^*):

$$h^* = \sqrt{Q_2/Q_1} = \left(\frac{f_{SRD} \cdot N_\delta/B}{\phi_\delta \cdot T \cdot (f_{EPQ} + f_{PQ}) + L_\delta \cdot T \cdot f_{SRQ}}\right)^{1/2} \tag{18}$$

This shows that h^* in the leveling layout should be as $\sqrt{T}\times$ larger than the tiering layout.

4.4.2 KiWi$^+$ for Tiered LSMs. Lastly, we compute the optimal level-wise delete tile granularity for KiWi$^+$. For this, we revise Equation (11) as follows.

$$\begin{aligned} Cost_{KiWi_i^+} &= f_{EPQ} \cdot T \cdot \phi_\delta \cdot h_i + f_{PQ} \cdot T \cdot [E_i \cdot (1 + \phi_\delta \cdot (h_i - 1)/2) + (1 - E_i - E_i') \cdot \phi_\delta \cdot h_i] \\ &\quad + f_{SRQ} \cdot T \cdot h_i + f_{LRQ} \cdot T \cdot s \cdot P \cdot T^i + f_{SRD} \cdot P \cdot T^i/h_i + f_I \cdot 1/B \\ &= h_i \cdot T \cdot [f_{EPQ} \cdot \phi_\delta + f_{PQ} \cdot E_i'' \cdot \phi_\delta + f_{PQ} \cdot E_i \cdot \phi_\delta/2 + f_{SRQ}] + f_{SRD} \cdot P \cdot T^i/h_i \\ &\quad + f_{LRQ} \cdot T \cdot s \cdot P \cdot T^i + f_I \cdot 1/B \\ &= Q_3 \cdot h_i + Q_4/h_i + C_2 \end{aligned} \tag{19}$$

where $Q_3 = f_{EPQ} \cdot T \cdot \phi_\delta + f_{PQ} \cdot E_i'' \cdot \phi_\delta + f_{PQ} \cdot E_i \cdot \phi_\delta/2 + f_{SRQ}$ and Q_4, C_2 remain the same as Equation (12). We then re-apply the minimization Equation (14) to derive

$$h_i^* = \sqrt{Q_4/Q_3} = \frac{1}{\sqrt{T}} \cdot \left(\frac{f_{SRD} \cdot P \cdot T^i}{\phi_\delta \cdot (f_{EPQ} + f_{PQ} \cdot E_i'') + \phi_\delta \cdot f_{PQ} \cdot E_i/2 + f_{SRQ}} \right)^{1/2} \tag{20}$$

The result is consistent with the conclusion we have for KiWi, that is, h_i^* in the leveling layout should be $\sqrt{T}\times$ larger than the tiering layout.

4.5 *Lethe*$^+$

Lethe$^+$ puts together the benefits of FADE and KiWi$^+$ to better support deletes in LSM-trees. For a given workload and a persistence delete latency threshold, *Lethe*$^+$ offers maximal performance, by carefully tuning the cost of persistent deletes, and their impact on the overall performance of the system. The key tuning knobs are (i) the delete persistence threshold (D_{th}) and (ii) delete tile granularity for the L levels of the tree (h_i). The delete persistence threshold is specified as part of the data retention SLA, and *Lethe*$^+$ sets the TTL for the tree-levels to ensure timely persistence.

For a workload with secondary range deletes, *Lethe*$^+$ tunes the storage layout to find the optimal value for the delete tile granularity using the frequency of read operations relative to the frequency of secondary range deletes. In order to find the optimal value of h_i for each level, we minimize the cost of *Lethe*$^+$ as presented in Equation (14).

For example, for a 400GB database with 1KB entry size, 4KB page size, and 1MB buffer/file size, if between two range deletes we execute 50M point queries of any type, 10K short range queries, and have $\phi = 0.0082$ and $T = 10$, using Equation (14), we have that the optimal values of h_i are: $h_1 = h_2 = 2$, $h_3 = 7$, $h_4 = 18$, $h_5 = 29$, and $h_6 = 36$. With the interweaved data layout generated by KiWi$^+$, we minimize the overall cost of running the workload, as we will see in the experimental analysis.

Implementation. *Lethe*$^+$ is implemented on top of RocksDB which is an open-source LSM-based key-value store widely used in the industry [32, 33]. The current implementation of RocksDB is implemented as leveling (only Level 1 is implemented as tiered to avoid write-stalls) and has a fixed size ratio (defaults to 10). We develop a new API for *Lethe*$^+$ to have fine-grained control on the infrastructure. The API allows us to initiate compactions in RocksDB based on custom triggers and design custom file picking policies during compactions. RocksDB already stores metadata for every file, which includes the number of entries and deletes and the *age* for each file. The delete persistence threshold is taken as a user-input at setup time and is used to dynamically set the level-TTLs.

5 EVALUATION

We evaluate *Lethe*$^+$ against state-of-the-art LSM-tree designs for a range of workloads with deletes and different delete persistence thresholds.

Experimental Setup. We use a server with two Intel Xeon Gold 6230 2.1GHz processors each having 20 cores with virtualization enabled and with 27.5MB L3 cache, 384GB of RDIMM main memory at 2933MHz and 240GB SSD.

Default Setup. Unless otherwise mentioned the experimental setup consists of an initially empty database with ingestion rate at 2^{10} entries per second. The size of each entry is 1KB, and the entries are uniformly and randomly distributed across the key domain and are inserted in random order. The size of the memory buffer is 1MB (implemented as a skip list). The size ratio for the LSM-tree is set to 10, and we use 10 bits per entry for the Bloom filters. To determine the raw performance, write operations are considered to have a lower priority than compactions. For all

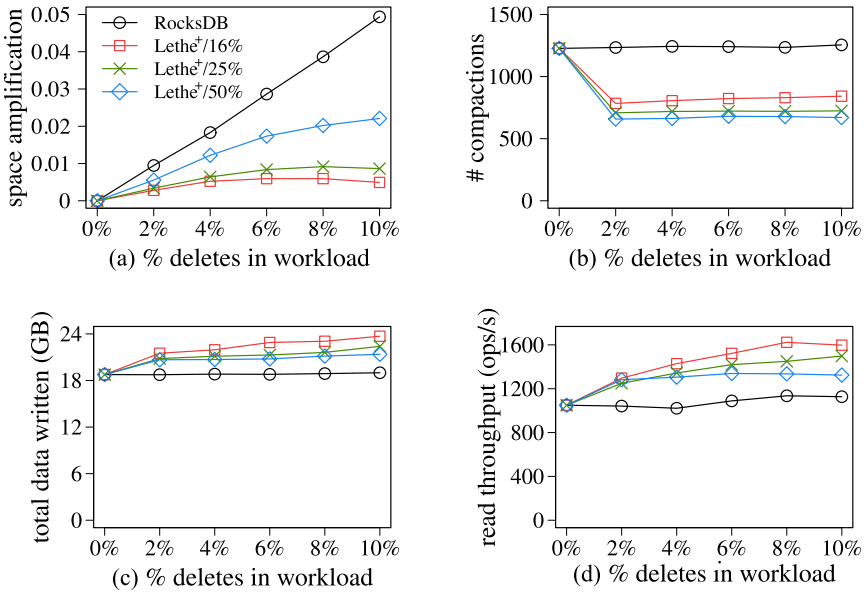

Fig. 7. *Lethe*+ reduces space amplification (a) and performs fewer larger compactions (b, c) to persist deletes timely, and in the process, by purging logically invalidated entries eagerly also improves read throughput (d).

experiments performed, the implementation for *Lethe*+ differs from the RocksDB setup in terms of only the compaction trigger and the file picking policy. We have both block cache and direct I/O enabled and the WAL disabled. Deletes are issued only on keys that have been inserted in the database and are uniformly distributed within the workload. We insert 1GB data in the database with compactions given a higher priority than writes. The delete persistence threshold is set to 16.67%, 25%, and 50% of the experiment's run-time. This experiment mimics the behavior of a long-running workload. The delete persistence threshold values chosen for experimentation are representative of practical applications, where the threshold is 2 months (16.67%), 3 months (25%), 6 months (50%), respectively, for a commercial database running for 1 year [31]. All lookups are issued after the whole database is populated.

Metrics. The compaction related performance metrics including (i) *total number of compactions performed*, (ii) *total bytes compacted*, (iii) *number of tombstones present in a tree*, and the (iv) *age of files containing tombstones* are measured by taking a snapshot of the database after the experiment. *Space* and *write amplification* are then computed using the equations from Sections 3.2.1 and 3.2.3.

Workload. Given the lack of delete benchmarks, we designed a synthetic workload generator, which produces a variation of YCSB Workload A, with 50% general updates and 50% point lookups. In our experiments, we vary the percentage of deletes between 2% to 10% of the ingestion.

5.1 Achieving Timely Delete Persistence

***Lethe*+ Reduces Space Amplification.** We first show that *Lethe*+ significantly reduces space amplification by persisting deletes timely. We set up this experiment by varying the percentage of deletes in a workload for both RocksDB and *Lethe*+, for three different delete persistence thresholds. The plot is shown in Figure 7(a). For a workload with no deletes, the performances of *Lethe*+ and RocksDB are identical. This is because in the *absence of deletes*, *Lethe*+ performs compactions triggered by level-saturation, choosing files with minimal overlap. In the *presence of deletes*, driven

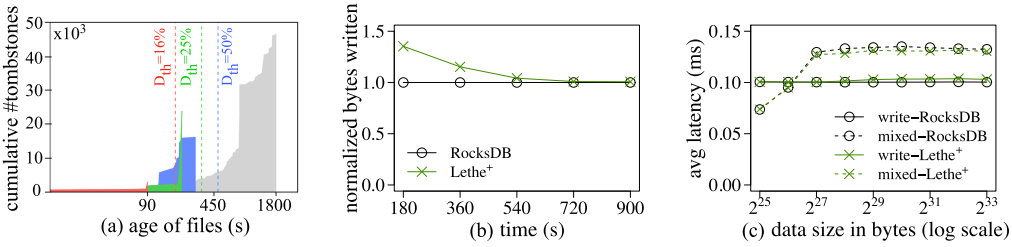

Fig. 8. (a) *Lethe*+ always persists deletes in a timely manner at the cost of a slightly higher write amplification. The higher write amplification in *Lethe*+ gets amortized over time (b), which allows *Lethe*+ to scale similarly to RocksDB (c).

by the delete persistence threshold (D_{th}), *Lethe*+ compacts files more frequently to ensure compliance with the threshold. It deletes the logically invalidated entries persistently, and in the process, diminishes the space amplification in the database. Even when D_{th} is set to 50% of the workload execution duration, *Lethe*+ reduces space amplification by about 48%. For shorter D_{th}, the improvements in space amplification are further pronounced by *Lethe*+.

***Lethe*+ Performs Fewer Compactions.** Figures 7(b) and (c) show that *Lethe*+ performs fewer compactions as compared to RocksDB, but compacts more data during every compaction. *Lethe*+ performs compactions on a rolling basis based on D_{th}. After each experiment, *Lethe*+ was found to have fewer files on disk as compared to RocksDB. This is because, *Lethe*+ compacts invalidated entries in a greedy manner, and for a workload with even a small fraction (2%) of deletes, it reduces the number of compactions performed by 45%, as shown in Figure 7(b). However, while compacting files with expired TTLs, the chosen file may overlap with a relatively higher number of files from the target level, and thus *Lethe*+ compacts 4.5% more data when D_{th} is set as 50% of the experiment's run-time, as shown in Figure 7(c).

***Lethe*+ Achieves Better Read Throughput.** In this experiment, we show that *Lethe*+ offers a superior read performance as compared to RocksDB. For this experiment, we populate the database with 1GB data and then issue point lookups on existing entries. Note that the lookups may target entries have been deleted by a tombstone after they were inserted. With more deletes in the workload, the number of invalidated entries (including tombstones) hashed into the BFs increases. *Lethe*+ purges these superfluous entries by persisting them in a time-bound manner, and thus, cleans up the BFs and improves their FPR when operating on fixed filter memory budget. A lookup on a persistently deleted key returns negative without performing a disk I/O to read a tombstone. Overall, Figure 7(d) shows that *Lethe*+ improves lookup performance by up to 17% for workloads with deletes.

***Lethe*+ Ensures Timely Delete Persistence.** Figure 8(a) shows the distribution of the tombstones ages at the end of the experiment to demonstrate that *Lethe*+ ensures timely persistent deletion. The X-axis shows the age of all files that contain tombstones, and the Y-axis shows the cumulative number of tombstones at the instant of the snapshot with the age corresponding to the X-axis value or smaller. The goal of *Lethe*+ is to have fewer tombstones of smaller age than the state of the art, with all tombstones having age less than D_{th}. We show that in comparison with RocksDB, *Lethe*+ persists between 40% and 80% more tombstones, and does so while honoring the delete persistence threshold. For D_{th} = 50% of the experiment's run-time, while RocksDB has ~40,000 tombstones (i.e., ~40% of all tombstones inserted) distributed among files that are older than D_{th}, *Lethe*+ persists all deletes within the threshold.

Write Amplification gets Amortized for *Lethe*+. This experiment demonstrates that the higher write amplification caused by the initial eager merging of *Lethe*+ is amortized over time. For *Lethe*+,

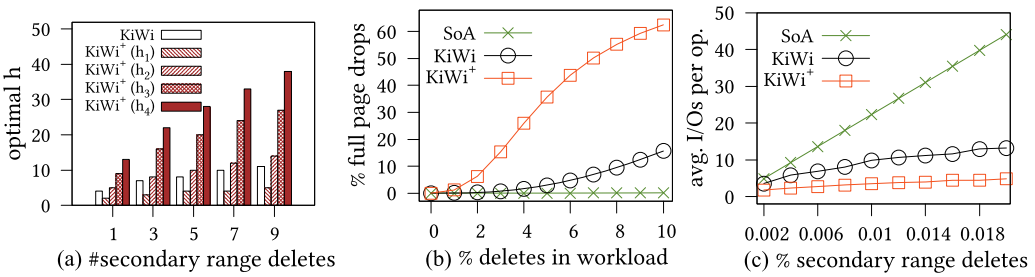

Fig. 9. (a) While KiWi has the same delete tile granularity (h) across all levels of the tree, KiWi$^+$ assigns the delete tile granularity optimally for each level, and by doing so, KiWi$^+$ (b) achieves significantly superior performance for secondary range deletes, as well as (c) for the overall workload execution.

we set D_{th} to 60 seconds and take snapshots at an interval of 180 seconds during the execution of the experiments. At every snapshot, we measure the cumulative bytes written over the past intervals. We measure the same metric for RocksDB (that does not support setting a D_{th}), and use it to normalize the bytes written, and then we plot the normalized bytes written against time (across snapshots) in Figure 8(b). We observe that due to eager merging, *Lethe*$^+$ writes 1.4× more data compared to RocksDB in the first snapshot. However, by persisting invalid entries upfront, *Lethe*$^+$ purges superfluous entries from the tree, and hence, compacts fewer entries during subsequent compactions. This reduces the normalized writes by *Lethe*$^+$ over time. At the end of the experiment, we observe that *Lethe*$^+$ writes only 0.7% more data as compared to RocksDB. In this experiment, we set the D_{th} to be 15× smaller than the experiment duration to model the worst case. In practice, insertions in LSM-engines continue for much longer (even perpetually) and D_{th} is set to a small constant duration. In this scenario, *Lethe*$^+$'s write amplification will be quickly amortized.

***Lethe*$^+$ Scales Similarly to the State of the Art.** This experiment shows that *Lethe*$^+$ and the state of the art follow the same performance trends as data volume grows. We set up this experiment with the default configuration, and we vary data size. In addition to YCSB Workload A, which is used to compute the mixed workload latency, we use a write-only workload to measure write latency. Figure 8(c), shows the average latency for both workloads with data size on the X-axis. We observe that the write latency for RocksDB and *Lethe*$^+$ is not affected by data size. Due to the initial increased write amplification of *Lethe*$^+$, its write latency is 0.1–3% higher than that of RocksDB. For the mixed workload, however, *Lethe*$^+$ improves the average latency by 0.5–4%. This improvement is primarily due to the higher read throughput achieved by *Lethe*$^+$, as shown in Figure 7(d). For smaller data sizes, most data entries are stored in memory or the first disk level, which reduces read latency significantly.

5.2 Secondary Range Deletes

Next, we evaluate the secondary range delete performance for KiWi$^+$ and compare it against KiWi and the state of the art.

Setup. Unless otherwise mentioned, the workload comprises 0.001% secondary range delete operations along with 1% short range queries and 50% point queries. Each file is 1MB in size and has 256 pages where the size of every page is 4KB. The default selectivity of secondary range deletes is 5%, and every short range query is expected to access at most 2 pages per level.

KiWi$^+$ sets a level-wise optimal interweaved data layout. The data layout suggested by KiWi has the same delete tile granularity (h) across all levels of the tree. While a global decision is able to improve performance over the state of the art, it does not account for the temporality

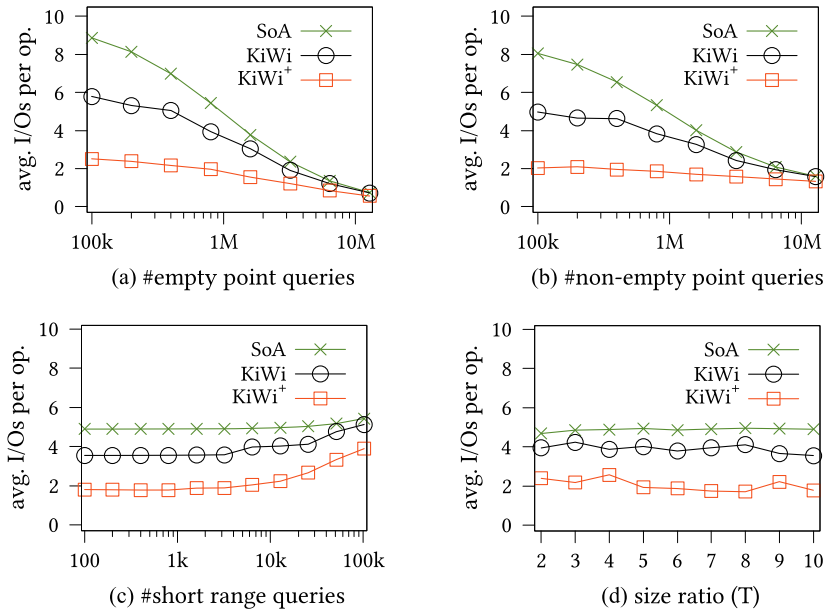

Fig. 10. We vary the number of (a) empty and (b) non-empty point query, (c) short-range query, and (d) the size ratio of a tree to analyze their impact on the performance of *Lethe*$^+$ compared to the state of the art.

and distribution of the point queries, leading to suboptimal designs. In Figure 9(a), we observe that KiWi$^+$ assigns a smaller delete tile granularity for smaller levels, which then increases as we move to larger levels. This way, KiWi$^+$ pays a smaller expected cost when probing the smaller levels, which also has a lower probability of having the target entry. KiWi$^+$ takes into account the probability of the entry being present in a particular level when coming up with the optimal data layout, and by doing so, it achieves a level-wise optimal interweaved data layout. Figure 9(a) also shows how KiWi$^+$ adapts to different workloads by navigating the continuum of storage layouts.

KiWi$^+$ Achieves Superior Secondary Range Delete Performance. Figure 9(b) shows that with the workload-aware and level-wise optimal data layout, KiWi$^+$ outperforms both the state of the art and KiWi in terms of secondary range delete performance. For this experiment, we vary the selectivity of a secondary range deletes, i.e., the fraction of the database that is deleted, and measure the number of pages that can be *fully dropped* during the operation. Full page drops do not require reading the page to memory, and thus, a higher value along the y-axis is desirable. State-of-the-art data layouts, which are based on the sort key alone, are unable to facilitate secondary range deletes efficiently, as they have to perform a full tree compaction, reading and rewriting every disk page. The interweaved data layout of KiWi is able to cluster some of the qualifying data, and as the proportion of delete grows, KiWi marks an increasing number of pages for full drop, thus, avoiding issuing I/Os to access those pages. For a workload with 10% entries under deletion, KiWi reduces the cost of secondary range deletes by $\approx$ 15%, while KiWi$^+$ further reduces the secondary range delete by 4×, leading to a 60% overall benefit when compared to the state of the art. The benefit of KiWi$^+$ grows up to 31× for a smaller fraction of deletes. For a workload with 2% entries under deletion, KiWi$^+$ is able to mark 26.3× more pages for full drop as compared to KiWi. Overall, the fine-grained nature of KiWi$^+$ is able to better capture a variety of workloads and reduces the cost of secondary range deletes between 4× and 31×.

***Lethe*$^+$ Improves Overall Workload Performance.** Figure 9(c) shows that *Lethe*$^+$ improves the overall workload performance using KiWi$^+$ to navigate the storage layouts. We observe that as the fraction of secondary range deletes in a workload increases (while keeping point lookups and short range queries constant at 50M and 10K, respectively), *Lethe*$^+$ reduces the average number of disk I/Os performed per operation and achieves a higher throughput for the workload. Compared to the state of the art, KiWi performs 27.4% fewer I/Os per operation for a workload with only 0.002% of secondary range deletes, which is further reduced by another 36% when KiWi$^+$ is employed As the proportion of secondary range deletes increases, the benefits of both KiWi and KiWi$^+$ become significantly pronounced. As we increase the proportion of secondary range deletes by one order of magnitude (to 0.02%), we observe that the average I/Os performed in KiWi and KiWi$^+$ reduces by 70% and 89%, respectively, compared to the state of the art. Overall, KiWi$^+$'s workload-aware level-wise optimal data layout allows it to further reduce the average operational cost over KiWi.

Analyzing the Effects of Different Workload Components. In this set of experiments, we vary the different workload components to quantify the sensitivity of *Lethe*$^+$ to each operation type. For the first two experiments, we fix the number of secondary range deletes and short-range queries, and we increase the empty (Figure 10(a)) and non-empty (Figure 10(b)) point queries. In both figures, we observe that as the number of point queries increases, the performance of KiWi and KiWi$^+$ converges to that of the state of the art. For KiWi, with 100K point queries, the average I/Os per operation is reduced by ∼38%, which is further reduced by another ∼37% for KiWi$^+$. However, as the point query count increases to 10M, the performance of KiWi converges with that of the state of the art, and that of KiWi$^+$ offers ∼1% improvement. This convergence happens because for a higher proportion of (empty or non-empty) point queries (f_{EQP} or f_{PQ}), the objective function of Equation (14) has a larger value at the denominator, leading to a smaller optimal value for the delete tile granularity (h_i^*). As $h_i^* \rightarrow 1$ for most levels, the data interweaved data layout essentially becomes the same as the one of the state of the art. KiWi$^+$, with its level-wise optimization, is able to have $h_L^* > 1$ for the last level and, thus, still holds a slight advantage over KiWi. Further, note that as the proportion of point queries increases, the average I/Os per operation reduces. This is due to the fact that point queries are significantly cheaper than secondary range deletes and range queries, hence the average operation cost decreases.

Next, we vary the number of short-range queries between 100 and 100K. In Figure 10(c), we observe that as the number of short-range queries increases, the performance benefits on KiWi and KiWi$^+$ decreases and converges to the performance of the state of the art. This is in line with Equation (14), i.e., as the proportion of short-range queries (f_{SRQ}) increases, the increase in the denominator on the right-hand side reduces the optimal delete tile granularity (h_i^*). However, as the I/O cost of a short-range query is larger than the average I/O cost of the workload, an increase in the proportion of short-range queries leads to an increase in the overall I/O cost. Thus, unlike what we observed in the previous two experiments, in Figure 10(c), we observe that the convergence happens toward a higher I/O per query cost. Note that both KiWi and KiWi$^+$ remain faster than the state of the art.

***Lethe*$^+$'s Benefits Are Not Affected By Size Ratio.** In Figure 10(d), we observe that as we vary the size ratio (T) of the LSM-tree, while keeping other tuning knobs unchanged, the relative benefits of *Lethe*$^+$ remain. Note that increasing the size ratio leads to trees with fewer levels. Figure 10(d) shows that the relative benefit offered by KiWi over the state of the art remains between 12.7% and 27.5% (20% on average) as we change the size ratio of the tree from 2 to 10, effectively transitioning from a tree with 12 levels to one with 4 levels. KiWi$^+$ further adds to the benefits by reducing the average I/Os performed per operation between 47.3% and 65.6% (58% on average) compared to the state of the art. This is because the number of pages accessed during a secondary range delete does not depend on the height of the tree, and in state-of-the-art data layouts, we would have to access

all pages in the tree. On the other hand, *Lethe*$^+$ clusters the qualifying entries into fewer pages in a workload-aware manner, and by doing so, it reduces the I/Os per operation for any tree height.

6 SQL SUPPORT FOR TIMELY DATA DELETION

While the legal regulations outline the users' rights for timely deletion of their personal data, and *Lethe*$^+$ outlines the infrastructure necessary to persist logical deletes in a timely manner at the system level, there is still a lack of support at the *query language* level that allows us to translate the user deletion requirements to directives to the underlying system. To bridge this gap, we need to (i) extract the user requirements from the policy layer and (ii) address the fact that the state-of-the-art query languages do not (yet) describe user deletion preferences. Toward this, we present a new set of SQL extensions that express the policy requirements for both retention-driven deletes and on-demand data deletion, bridging the deletion policy with the system capabilities to support timely data deletion [73]. To do this, we augment both the **data definition language (DDL)** and the **data manipulation language (DML)** parts of SQL.

Enabling Retention-Driven Deletes. To support retention-driven data deletion, we extend (i) the `CREATE TABLE` DDL and (ii) the `INSERT` DML in SQL. The `CREATE TABLE` statement now allows an application developer to specify the different retention durations supported as a table-property.

```
CREATE TABLE R (column1 type1, column2 type2, ...)
WITH RET_DUR FIXED (t1 <ret1>, t2 <ret2>, ...);
```

The above SQL statement creates a table `R` that supports retention-based deletes with specific retention durations of `ret1`, `ret2`, and so on. The `WITH RET_DUR` clause is optional, and is only necessary for tables that need to support deletes with predefined retention durations. When a table supports a predefined set of retention durations, each `INSERT` statement can use only one of those. For example, a table that is configured to support retention durations of 30 days and 60 days (`CREATE TABLE R (...) WITH RET_DUR FIXED (t1 '30 days', t2 '60 days');`), can only receive inserts with retention durations `t1` or `t2`. An ingestion without a retention period explicitly mentioned is kept perpetually following the logic of a classical insert. Note that, in general, the predefined retention durations stem from the delete-SLAs that a specific application requires. Following is the syntax for inserts.

```
INSERT INTO R (val1, val2, ...)
WITH RET_DUR t<i>;
```

Support for Arbitrary Retention Durations. We further extend SQL to support *arbitrary* retention durations for deletes. To do so, we add the `ARBITRARY` keyword to both the `CREATE TABLE` and `INSERT` statements. Supporting arbitrary retention durations is common in distributed frameworks that replicate data across physical data stores in different geographic locations, each bound by different regulatory requirements. Below, we present the full syntax for the proposed SQ extensions.

```
CREATE TABLE R (column1 type1, column2 type2, ...)
WITH RET_DUR
{ARBITRARY | FIXED (t1 <ret1>, t2 <ret2>, ...)};

INSERT INTO R (val1, val2, ...)
WITH RET_DUR { <t> | t<i> } ;
```

Note that having a pre-defined set of retention durations provides more information to the system compared to allowing arbitrary durations. As a result, it allows the system to better prepare to offer efficient retention-driven deletes.

Enabling Timely On-Demand Deletion. To support on-demand data deletion in a timely manner, we introduce the keyword DPT which denotes the *delete persistence threshold*, that is the maximum delay between a logical delete and its persistence. Each table can provide support for several such user-defined thresholds. Similarly to retention-based deletes, we also extend SQL to support arbitrary delete persistence thresholds when they are not specified *a priori*. Below, we outline the modifications to the DDL and DML necessary to support on-demand timely deletion requests.

```
CREATE TABLE S (column1 type1, column2 type2, ...)
WITH DPT
{ARBITRARY | FIXED (d1 <dpt1>, d2 <dpt2>, ...)};

DELETE FROM S WHERE (...)
WITH DPT { <d> | d<i> };
```

Table S can support several DPTs (dpt1, dpt2, etc.), if the delete persistence thresholds are specified beforehand, and applications can trigger on-demand deletion with any delete persistence threshold through the DELETE command. Similarly to retention-driven deletes, timely persistent on-demand deletion is easier to handle from a storage engine if the delete persistence thresholds supported are known *a priori* during the table creation.

Putting Everything Together. Putting the proposed DDL extensions together, a table can support multiple (pre-defined or arbitrary) thresholds for both retention-based and on-demand deletes. The complete syntax for CREATE TABLE is as follows.

```
CREATE TABLE T (column1 type1, column2 type2, ...)
WITH RET_DUR
{ARBITRARY | FIXED (t1 <ret1>, t2 <ret2>, ...)};
WITH DPT
{ARBITRARY | FIXED (d1 <dpt1>, d2 <dpt2>, ...)};
```

Note that retention-based deletes come from the *application requirements*, and on-demand deletion requests are issued *by the user*. Further, note that while these SQL extensions allow us to express deletion preferences, they rely on the system layer to correctly realize them.

7 RELATED WORK

Deletion in Relational Systems. Past work on data deletion on relational systems focuses on bulk deletes [19, 36, 55]. Efficient bulk deletion relies on similar techniques as efficient reads: sorting or hashing data to quickly locate, and ideally collocate, the entries to be deleted. Efficient deletion has also been studied in the context of spatial data [53] and view maintenance [11]. Contrary to past work, *Lethe*$^+$ aims to support a user-provided delete persistence latency threshold. Some research have also focussed on building workload-aware data structure and deletion-friendly designs [48, 49, 68, 69]. The research primarily targeted data structures, such as B$^+$-trees, that realize deletes in place. The problem of timely delete persistence, however, is particular to data structures that follow the out-of-place delete/update paradigm.

Self-Destructing Data. In addition, past research has proposed to automatically make data disappear when specific conditions are met. Vanish is a scheme that ensures that all copies of certain data become unreadable after a user-specified time, without any specific action on the part of a user [37, 38]. Kersten [50] and Heinis and Ailamaki [42] have proposed the concept of forgetting in data systems through biology-inspired mechanisms as a way to better manage storage space and for efficient data analysis capabilities, as the data generation trends continue to increase. Contrary to this, *Lethe*$^+$ supports timely data deletion that is set by users/applications.

8 CONCLUSION

In this work, we point out that state-of-the-art LSM-based key-value stores perform suboptimally for workloads with even a small proportion of deletes. In fact, these systems are unable to provide any latency guarantees on when a logical delete may be persisted, leading to potentially unbounded delete persistence latency by design. Further, state-of-the-art LSM engines can not support range deletes (secondary range deletes) on an attribute that is different from the sort key. To address these, we build *Lethe*$^+$, a deletion-aware LSM-based engine that introduces FADE, a new family of compaction strategies, and KiWi$^+$, a continuum of physical data storage layouts. FADE ensures that all logical deletes are persisted in a timely manner, within a user-defined delete persistence threshold. Timely persistence of logical deletes improves read throughput and reduces space amplification, at the expense of a modest increase in write amplification. KiWi$^+$ offers efficient secondary range deletes, which can be tuned based on the expected workload. Further, we propose a set of SQL extensions that serve as the bridge between the deletion user requirements as augmented by the recent privacy policies and the system-level support for primary and secondary persistent deletes.

REFERENCES

[1] 2016. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46/EC. *Official Journal of the European Union (Legislative Acts)* (2016), L119/1–L119/88.

[2] 2018. California Consumer Privacy Act. *Assembly Bill No. 375, Chapter 55* (2018).

[3] 2020. The California Privacy Rights Act of 2020. https://thecpra.org/. (2020).

[4] 2021. Virginia Consumer Data Protection Act. https://www.sullcrom.com/files/upload/SC-Publication-Virginia-Second-State-Enact-Privacy-Legislation.pdf. (2021).

[5] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1792–1803. https://doi.org/10.14778/2824032.2824076

[6] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. 2014. H2O: A hands-free adaptive store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1103–1114. https://doi.org/10.1145/2588555.2610502

[7] Apache. 2023. Accumulo. https://accumulo.apache.org/.

[8] Apache. 2023. Cassandra. http://cassandra.apache.org.

[9] Apache. 2023. HBase. http://hbase.apache.org/.

[10] Raja Appuswamy, Manos Karpathiotakis, Danica Porobic, and Anastasia Ailamaki. 2017. The case for heterogeneous HTAP. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*. http://cidrdb.org/cidr2017/papers/p21-appuswamy-cidr17.pdf.

[11] Chandrabose Aravindan and Peter Baumgartner. 1997. A rational and efficient algorithm for view deletion in databases. In *Proceedings of the International Symposium on Logic Programming (ILPS)*. 165–179.

[12] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 2016. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 583–598. https://doi.org/10.1145/2882903.2915231

[13] Manos Athanassoulis and Anastasia Ailamaki. 2014. BF-Tree: Approximate tree indexing. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1881–1892. http://www.vldb.org/pvldb/vol7/p1881-athanassoulis.pdf.

[14] Manos Athanassoulis, Shimin Chen, Anastasia Ailamaki, Phillip B. Gibbons, and Radu Stoica. 2011. MaSM: Efficient online updates in data warehouses. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 865–876. http://dl.acm.org/citation.cfm?id=1989323.1989414.

[15] Manos Athanassoulis, Shimin Chen, Anastasia Ailamaki, Phillip B. Gibbons, and Radu Stoica. 2015. Online updates on data warehouses via judicious use of solid-state storage. *ACM Transactions on Database Systems (TODS)* 40, 1 (2015).

[16] Manos Athanassoulis, Michael S. Kester, Lukas M. Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark Callaghan. 2016. Designing access methods: The RUM conjecture. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 461–466. http://dx.doi.org/10.5441/002/edbt.2016.42.

[17] Manos Athanassoulis, Subhadeep Sarkar, Tarikul Islam Papon, Zichen Zhu, and Dimitris Staratzis. 2022. Building deletion-compliant data systems. In *IEEE Data Engineering Bulletin*. 21–36.

[18] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. 53–64. https://doi.org/10.1145/2254756.2254766

[19] Bishwaranjan Bhattacharjee, Timothy Malkemus, Sherman Lau, Sean Mckeough, Jo-Anne Kirton, Robin Von Boeschoten, and John Kennedy. 2007. Efficient bulk deletes for multi dimensionally clustered tables in DB2. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 1197–1206.

[20] Mark Callaghan. 2020. Deletes are fast and slow in an LSM. http://smalldatum.blogspot.com/2020/01/deletes-are-fast-and-slow-in-lsm.html.

[21] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 209–223.

[22] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A distributed storage system for structured data. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 205–218. http://dl.acm.org/citation.cfm?id=1267308.1267323.

[23] Cisco. 2018. Cisco global cloud index: Forecast and methodology, 2016-2021. *White Paper* (2018). https://tinyurl.com/CiscoGlobalCloud2018.

[24] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's globally-distributed database. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 251–264. https://www.usenix.org/system/files/conference/osdi12/osdi12-final-16.pdf.

[25] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The snowflake elastic data warehouse. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 215–226. https://doi.org/10.1145/2882903.2903741

[26] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal navigable key-value store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 79–94. https://doi.org/10.1145/3035918.3064054

[27] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2018. Optimal Bloom filters and adaptive merging for LSM-trees. *ACM Transactions on Database Systems (TODS)* 43, 4 (2018), 16:1–16:48. https://doi.org/10.1145/3276980

[28] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 505–520. https://doi.org/10.1145/3183713.3196927

[29] Niv Dayan and Stratos Idreos. 2019. The log-structured merge-bush & the wacky continuum. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 449–466. https://doi.org/10.1145/3299869.3319903

[30] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review* 41, 6 (2007), 205–220. https://doi.org/10.1145/1323293.1294281

[31] Amol Deshpande and Ashwin Machanavajjhala. 2018. ACM SIGMOD blog: Privacy challenges in the post-GDPR world: A data management perspective. http://wp.sigmod.org/?p=2554.

[32] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing space amplification in RocksDB. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*. http://cidrdb.org/cidr2017/papers/p82-dong-cidr17.pdf.

[33] Facebook. 2021. RocksDB. https://github.com/facebook/rocksdb.

[34] Facebook. 2023. MyRocks. http://myrocks.io/.

[35] Gartner. 2017. Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016. https://tinyurl.com/Gartner2020.

[36] Andreas Gärtner, Alfons Kemper, Donald Kossmann, and Bernhard Zeller. 2001. Efficient bulk deletes in relational databases. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 183–192. https://doi.org/10.1109/ICDE.2001.914827

[37] Roxana Geambasu, Tadayoshi Kohno, Amit A. Levy, and Henry M. Levy. 2009. Vanish: Increasing data privacy with self-destructing data. In *Proceedings of the USENIX Security Symposium*. 299–316.

[38] Roxana Geambasu, Amit A. Levy, Tadayoshi Kohno, Arvind Krishnamurthy, and Henry M. Levy. 2010. Comet: An active distributed key-value store. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 323–336.

[39] Michelle Goddard. 2017. The EU General Data Protection Regulation (GDPR): European regulation that has a global impact. *International Journal of Market Research* 59, 6 (2017), 703–705. https://doi.org/10.2501/IJMR-2017-050
[40] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. 2015. Scaling concurrent log-structured data stores. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*. 32:1–32:14. https://doi.org/10.1145/2741948.2741973
[41] Google. 2021. LevelDB. https://github.com/google/leveldb/.
[42] Thomas Heinis and Anastasia Ailamaki. 2015. Reconsolidating data structures. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 665–670. https://doi.org/10.5441/002/edbt.2015.66
[43] Sándor Héman, Marcin Zukowski, and Niels J. Nes. 2010. Positional update handling in column stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 543–554. http://dl.acm.org/citation.cfm?id=1807167.1807227.
[44] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An optimized storage engine for large-scale e-commerce transaction processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 651–665. https://doi.org/10.1145/3299869.3314041
[45] Yan Huang, Tom Z. J. Fu, Dah-Ming Chiu, John C. S. Lui, and Cheng Huang. 2008. Challenges, design and analysis of a large-scale p2p-vod system. In *Proceedings of the ACM SIGCOMM 2008 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Seattle, WA, USA, August 17-22, 2008*. 375–388. https://doi.org/10.1145/1402958.1403001
[46] Fabian Hueske. 2018. State TTL for Apache Flink: How to limit the lifetime of state. *Ververica* (2018).
[47] HyperLevelDB. 2023. Online reference. https://github.com/rescrv/HyperLevelDB.
[48] Theodore Johnson and Dennis Shasha. 1989. Utilization of b-trees with inserts, deletes and modifies. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*. 235–246. https://doi.org/10.1145/73721.73745
[49] Theodore Johnson and Dennis E. Shasha. 1993. B-trees with inserts and deletes: Why free-at-empty is better than merge-at-half. *Journal of Computer and System Sciences (JCSS)* 47, 1 (1993), 45–76. https://doi.org/10.1016/0022-0000(93)90020-W
[50] Martin L. Kersten. 2015. Big data space fungus. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR), Gong show talk*. https://www.monetdbsolutions.com/sites/default/files/CIDR2015Kersten.pdf.
[51] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream processing at scale. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 239–250. https://doi.org/10.1145/2723372.2742788
[52] Andrew Lamb, Matt Fuller, and Ramakrishna Varadarajan. 2012. The Vertica analytic database: C-store 7 years later. *Proceedings of the VLDB Endowment* 5, 12 (2012), 1790–1801. http://dl.acm.org/citation.cfm?id=2367518.
[53] Jui-Tine Lee and Geneva G. Belford. 1992. An efficient object-based algorithm for spatial searching, insertion and deletion. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 40–47. https://doi.org/10.1109/ICDE.1992.213207
[54] Yinan Li, Bingsheng He, Jun Yang, Qiong Luo, Ke Yi, and Robin Jun Yang. 2010. Tree indexing on solid state drives. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 1195–1206. http://dl.acm.org/citation.cfm?id=1920841.1920990.
[55] Timo Lilja, Riku Saikkonen, Seppo Sippu, and Eljas Soisalon-Soininen. 2007. Online bulk deletion. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 956–965. https://doi.org/10.1109/ICDE.2007.368954
[56] LinkedIn. 2023. Voldemort. http://www.project-voldemort.com.
[57] Honghui Lu. 2004. Peer-to-peer support for massively multiplayer games. In *Proceedings of the Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*. https://doi.org/10.1109/INFCOM.2004.1354485
[58] Chen Luo and Michael J. Carey. 2020. LSM-based storage techniques: A survey. *The VLDB Journal* 29, 1 (2020), 393–418. https://doi.org/10.1007/s00778-019-00555-y
[59] Abhishek Madan and Andrew Kryczka. 2018. DeleteRange: A new native RocksDB operation. https://rocksdb.org/blog/2018/11/21/delete-range.html.
[60] C. Mohan. 2016. Hybrid transaction and analytics processing (HTAP): State of the art. In *Proceedings of the International Workshop on Business Intelligence for the Real-Time Enterprise (BIRTE)*.
[61] MongoDB. 2023. Online reference. http://www.mongodb.com/.
[62] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385. http://dl.acm.org/citation.cfm?id=230823.230826.
[63] Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. 2017. Hybrid transactional/analytical processing: A survey. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1771–1775. https://doi.org/10.1145/3035918.3054784
[64] Alexandros Pantelopoulos and Nikolaos G. Bourbakis. 2010. Prognosis: A wearable health-monitoring system for people at risk: Methodology and modeling. *IEEE Trans. Information Technology in Biomedicine* 14, 3 (2010), 613–621. https://doi.org/10.1109/TITB.2010.2040085

[65] Stavros Papadopoulos, Kushal Datta, Samuel Madden, and Timothy Mattson. 2016. The TileDB array data storage manager. *Proceedings of the VLDB Endowment* 10, 4 (2016), 349–360. https://doi.org/10.14778/3025111.3025117
[66] Tarikul Islam Papon and Manos Athanassoulis. 2021. A parametric I/O model for modern storage devices. In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*.
[67] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. 497–514. https://doi.org/10.1145/3132747.3132765
[68] Aneesh Raman, Konstantinos Karatsenidis, Subhadeep Sarkar, Matthaios Olma, and Manos Athanassoulis. 2022. BoDS: A benchmark on data sortedness. In *Proceedings of the TPC Technology Conference on Performance Evaluation \& Benchmarking (TPCTC)*.
[69] Aneesh Raman, Subhadeep Sarkar, Matthaios Olma, and Manos Athanassoulis. 2023. Indexing for near-sorted data. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*.
[70] RocksDB. 2020. FIFO compaction style. https://github.com/facebook/rocksdb/wiki/FIFO-compaction-style.
[71] Takeshi Sakaki, Makoto Okazaki, and Yutaka Matsuo. 2010. Earthquake shakes Twitter users: Real-time event detection by social sensors. In *Proceedings of the International Conference on World Wide Web (WWW)*. 851–860. https://doi.org/10.1145/1772690.1772777
[72] Subhadeep Sarkar and Manos Athanassoulis. 2022. Dissecting, designing, and optimizing LSM-based data stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2489–2497. https://doi.org/10.1145/3514221.3522563
[73] Subhadeep Sarkar and Manos Athanassoulis. 2022. Query language support for timely data deletion. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 429–434.
[74] Subhadeep Sarkar, Jean-Pierre Banâtre, Louis Rilling, and Christine Morin. 2018. Towards enforcement of the EU GDPR: Enabling data erasure. In *Proceedings of the IEEE International Conference of Internet of Things (iThings)*. 1–8.
[75] Subhadeep Sarkar, Subarna Chatterjee, and Sudip Misra. 2018. Assessment of the suitability of fog computing in the context of internet of things. *IEEE Transactions on Cloud Computing (TCC)* 6, 1 (2018), 46–59. https://doi.org/10.1109/TCC.2015.2485206
[76] Subhadeep Sarkar, Kaijie Chen, Zichen Zhu, and Manos Athanassoulis. 2022. Compactionary: A dictionary for LSM compactions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2429–2432. https://doi.org/10.1145/3514221.3520169
[77] Subhadeep Sarkar, Niv Dayan, and Manos Athanassoulis. 2023. The LSM design space and its read optimizations. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*.
[78] S. Sarkar and S. Misra. 2016. Theoretical modelling of fog computing: A green computing paradigm to support IoT applications. *IET Networks* 5, 2 (2016), 23–29. https://doi.org/10.1049/iet-net.2015.0034
[79] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, and Manos Athanassoulis. 2020. Lethe: A tunable delete-aware LSM engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 893–908. https://doi.org/10.1145/3318464.3389757
[80] Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. 2021. Constructing and analyzing the LSM compaction design space. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2216–2229. http://vldb.org/pvldb/vol14/p2216-sarkar.pdf.
[81] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: A general purpose log structured merge tree. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 217–228. https://doi.org/10.1145/2213836.2213862
[82] SQLite4. 2023. Online reference. https://sqlite.org/src4/.
[83] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel R. Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2005. C-store: A column-oriented DBMS. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 553–564. http://dl.acm.org/citation.cfm?id=1083592.1083658.
[84] Risi Thonangi and Jun Yang. 2017. On log-structured merge for solid-state drives. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 683–694. https://doi.org/10.1109/ICDE.2017.121
[85] TileDB. 2023. Online reference. https://tiledb.io.
[86] Quoc-Cuong To, Juan Soto, and Volker Markl. 2017. A survey of state management in big data processing systems. *CoRR* abs/1702.0 (2017). arXiv:1702.01596
[87] Peter Van Sandt, Yannis Chronis, and Jignesh M. Patel. 2019. Efficiently searching in-memory sorted arrays: Revenge of the interpolation search?. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 36–53. https://doi.org/10.1145/3299869.3300075
[88] Zack Whittaker and Natasha Lomas. 2019. Even years later, Twitter doesn't delete your direct messages. https://techcrunch.com/2019/02/15/twitter-direct-messages/ (2019). https://techcrunch.com/2019/02/15/twitter-direct-messages/.

[89] WiredTiger. 2021. Source code. https://github.com/wiredtiger/wiredtiger.
[90] Lei Yang, Hong Wu, Tieying Zhang, Xuntao Cheng, Feifei Li, Lei Zou, Yujie Wang, Rongyao Chen, Jianying Wang, and Gui Huang. 2020. Leaper: A learned prefetcher for cache invalidation in LSM-tree based storage engines. *Proceedings of the VLDB Endowment* 13, 11 (2020), 1976–1989.
[91] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical range query filtering with fast succinct tries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 323–336. https://doi.org/10.1145/3183713.3196931
[92] Z. Zhu, S. Sarkar, and M. Athanassoulis. 2023. Acheron: Persisting tombstones in LSM engines. *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2023). https://doi.org/10.1145/3555041.3589719
[93] Marcin Zukowski and Peter A. Boncz. 2012. Vectorwise: Beyond column stores. *IEEE Data Engineering Bulletin* 35, 1 (2012), 21–27. http://sites.computer.org/debull/A12mar/vectorwise.pdf.

Received 5 October 2022; accepted 27 March 2023