Constructing and Analyzing the LSM Compaction Design Space

Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, Manos Athanassoulis
Boston University, MA, USA
ssarkar1@bu.edu,dstara@bu.edu,zczhu@bu.edu,mathan@bu.edu

ABSTRACT

Log-structured merge (LSM) trees offer efficient ingestion by appending incoming data, and thus, are widely used as the storage layer of production NoSQL data stores. To enable competitive read performance, LSM-trees periodically re-organize data to form a tree with levels of exponentially increasing capacity, through iterative *compactions*. Compactions fundamentally influence the performance of an LSM-engine in terms of write amplification, write throughput, point and range lookup performance, space amplification, and delete performance. Hence, choosing the *appropriate compaction strategy* is crucial and, at the same time, hard as the LSM-compaction design space is vast, largely unexplored, and has not been formally defined in the literature. As a result, most LSM-based engines use a fixed compaction strategy, typically hand-picked by an engineer, which decides *how* and *when* to compact data.

In this paper, we present the design space of LSM-compactions, and evaluate state-of-the-art compaction strategies with respect to key performance metrics. Toward this goal, our first contribution is to introduce a set of four design primitives that can formally define any compaction strategy: (i) the compaction trigger, (ii) the data layout, (iii) the compaction granularity, and (iv) the data movement policy. Together, these primitives can synthesize both existing and completely new compaction strategies. Our second contribution is to experimentally analyze 10 compaction strategies. We present 12 observations and 7 high-level takeaway messages, which show how LSM systems can navigate the compaction design space.

PVLDB Reference Format:
Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, Manos Athanassoulis. Constructing and Analyzing the LSM Compaction Design Space. PVLDB, 14(11): 2216 - 2229, 2021. doi:10.14778/3476249.3476274
PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://disc.bu.edu/lsm-compaction.

1 INTRODUCTION

LSM-based Key-Value Stores. Log-structured merge (LSM) trees are widely used today as the storage layer of modern NoSQL key-value stores [36, 42, 45]. LSM-trees employ the *out-of-place* paradigm to achieve fast ingestion. Incoming key-value pairs are buffered in main memory, and are periodically flushed to persistent storage as *sorted immutable runs*. As more runs accumulate on disk, they are sort-merged to construct fewer yet longer sorted runs. This process is known as *compaction* [30, 42]. To facilitate fast *point lookups*, LSM-trees use auxiliary in-memory data structures (Bloom filters and fence pointers) that help to reduce the average number of disk I/Os performed per lookup [21, 22]. Because of these advantages, LSM-trees are adopted by several production key-value stores including LevelDB [32] and BigTable [17] at Google, RocksDB [30] at Facebook, X-Engine [34] at Alibaba, WiredTiger at MongoDB [62], CockroachDB at Cockroach Labs [18], Voldemort [40] at LinkedIn, DynamoDB [25] at Amazon, AsterixDB [3], Cassandra [8], HBase [7], Accumulo [6] at Apache, and bLSM [56] and cLSM [31] at Yahoo. Academic systems based on LSM-trees include Monkey [21], SlimDB [47], Dostoevsky [22, 23], LSM-Bush [24], Lethe [51], Silk [11, 12], LSbM-tree [58], SifrDB [44], and Leaper [63].

Proceedings of the VLDB Endowment, Vol. 14, No. 11 ISSN 2150-8097.
doi:10.14778/3476249.3476274

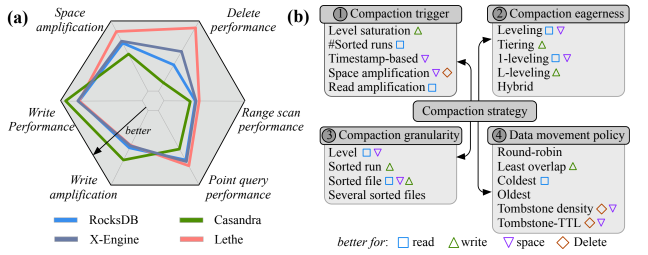


Fig. 1: (a) The different compaction strategies adopted in state-of-the-art LSM-engines lead to the diverse performances offered by the engines; (b) The taxonomy of LSM compactions in terms of the design primitives.

Compactions in LSM-Trees. Compactions in LSM-trees are employed periodically to *reduce read and space amplification at the cost of write amplification* while ensuring data consistency and query correctness [9, 10]. A compaction merges two or more sorted runs, between one or multiple levels to ensure that the LSM-tree maintains levels with exponentially increasing sizes [45]. Compactions are typically invoked when a level reaches its capacity, at which point, the compaction routine moves data from the saturated level to the next one, that has an exponentially larger capacity. Any duplicate entries (resulting from *updates*) and invalidated entries (resulting from *deletes*) are removed during a compaction, retaining only the logically correct (latest valid) version [28, 51]. Compactions dictate *how* and *when* disk-resident data is re-organized, and thereby, influence the physical data layout on the disk. Fig. 1(a) presents qualitatively the performance implications of the various compaction strategies adopted in state-of-the-art LSM-engines.

The Challenge: Hand-Picking Compaction Strategies. Despite compactions being critical to the performance of LSM-engines, the process of *choosing an appropriate compaction strategy* requires a human in the loop. In practice, decisions on *"how to (re-)organize data on disk"*, and thereby, *"which compaction strategies to implement or use"* in a production LSM-based data store are often subject to the expertise of the engineers or the database administrators (DBAs). This is largely due to two reasons. First, the process of compaction in LSM-trees is often treated as a black-box and is rarely exposed as a tunable knob [61]. While the LSM-compaction design space is vast, the lack of a formal template for compactions leads to heavily relying on individual expertise, leaving a large part of the design space unexplored. Second, there is a lack of analytical and experimental data on how compactions influence the performance of an LSM-engine subject to the underlying design of the storage engine and the workload characteristics. Hence, it is difficult, even for experts, to answer design questions such as:

(i) My LSM-engine is offering lower write performance than expected: *Would a change in the compaction strategy help? If yes, which strategies should be used?*
(ii) The workload we used to process has changed: *How does this affect the read throughput of my system? Is there a compaction strategy that can improve the read throughput?*
(iii) We are due to design a new LSM-engine for processing a specific workload: *How should I compact my data for best overall performance? Is there a compaction strategy that I must avoid?*

Relying on human expertise to hand-pick the appropriate compaction strategies for each application does not scale, especially for large-scale system deployments.

Contributions. To this end, in this work, we formalize the design space of compactions in LSM-based storage engines. Further, we experimentally explore this space, and based on this, we present 7 high-level takeaway messages, and 12 observations that serve as a comprehensive set of guidelines for LSM-compactions, and lay the groundwork for compaction tuning and automation.

Conceptual Contribution: Constructing the Compaction Design Space. We identify the defining characteristics of a compaction, or compaction *primitives*: (i) the **trigger** (i.e., *when* to compact), (ii) the **data layout** (i.e., *how* to organize the data after compaction), (iii) the **granularity** (i.e., *how much* data to compact at a time), and (iv) the **data movement policy** (i.e., *which* data to compact). Together, the four primitives define when and how to compact data in an LSM-tree. Fig. 1(b) presents the taxonomy of LSM-compactions along with the various options for each of the design primitives.

Experimental Contribution 1: Unifying the Experimental Infrastructure of Multiple Compaction Strategies. To establish a consistent experimental platform, we integrate several state-of-the-art compaction strategies into a unified codebase, based on the widely adopted open-source RocksDB [30] LSM-engine. This integration bridges wild variations of implementation and configuration knobs of different compaction strategies across different LSM-engines. Further, we implement each compaction strategy through the prism of the aforementioned four primitives on top of the same data store to ensure an apples-to-apples comparison. We implement *ten state-of-the-art compaction strategies* that are popular among production and academic systems, and are key to the understanding of the LSM-compaction design space. We implement these strategies through significant modifications to the latest RocksDB codebase [30], and expose *more than a hundred design knobs* to enable custom configuration and to ensure a fair evaluation.

Experimental Contribution 2: Analyzing the Compaction Design Space. We provide a comprehensive experimental analysis of the LSM-compaction design space, which quantifies the impact of each of the design primitives on a number of performance metrics. This experimental analysis also serves as a roadmap for selecting a compaction strategy subject to the workload characteristics and performance goals. We perform more than 2000 experiments with 10 compaction strategies to take a deep dive on the following.

- *Performance Implications.* We quantify the impact of compactions on LSM performance in terms of ingestion throughput, query latency, space and write amplification, and delete efficacy in §5.1.
- *Workload Influence on Compactions.* While the composition and (ingestion and access) distribution of the workload influence the compaction performance, deciding which compaction strategy to employ is workload-agnostic in existing systems. To analyze the workload's impact on compactions performance, we experiment with a number of representative workloads by varying (i) the size of ingested data, (ii) the proportion of ingestion and lookups, (iii) the proportion of empty and non-empty point lookups, (iv) the selectivity of range queries, (v) the fraction of updates and (vi) deletes, (vii) the key-value size, as well as (viii) the workload distribution (uniform, normal, and Zipfian) in §5.2.
- *Tuning Influence on Compactions.* LSM tuning typically focuses on knobs like memory buffer size, page size, and size ratio which are not believed to be connected with compaction performance. We experiment with these knobs to uncover when compactions are affected (and when not) by these knobs in §5.3.
- *Answering Design Questions.* Finally, throughout §5 we present various observations and key insights of our experimental evaluation, and in §6 we discuss a roadmap for designing and choosing compaction in LSM-engines.

This work defines the LSM compaction design space and presents a thorough account of how the different primitives affect the overall performance of a storage engine.

Key Takeaways. Finally, the high-level key takeaways from our analysis are the following.

A. There is no perfect compaction strategy. When it comes to selecting a compaction strategy for an LSM-engine, there is *no single best*. Thus, a compaction strategy needs to be custom-tailored to specific combinations of workload, LSM tuning, and performance goals.

B. It is important to look into the compaction "black-box". To understand the performance implications of LSM compactions, it is crucial to "open the black-box", and treat them as a set of design primitives. Following this approach, we reason about the performance implications of each design primitive independently. We identify common pitfalls given a workload and a target performance.

C. The right compaction strategy can significantly boost performance. Switching between compaction strategies as the workload and/or the performance goals shift can boost the performance of an LSM-engine significantly. Understanding the behavior and performance implications of the compaction primitives allows for modifications to existing codebases to invoke the appropriate compaction strategy.

2 BACKGROUND

We now present the necessary background of LSM-trees. A more detailed survey on LSM-basics can be found in the literature [21, 42].

LSM-Basics. To support fast data ingestion, LSM-trees buffer incoming inserts, updates, and deletes (i.e., ingestion, in general) within main memory. Once the memory buffer becomes full, the entries contained are sorted on the key and the buffer is flushed as a *sorted run* to the disk-component of the tree. In practice, a sorted run is a collection of one or more *immutable files* that have typically the same size. 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 [21, 42]. On disk, each Level i ($i > 1$) has a capacity that is larger than that of Level $i-1$ by a factor of T, where T is the size ratio of the tree.

LSM-Compactions. To limit the number of sorted runs on disk (and thereby, to facilitate fast lookups and better space utilization), LSM-trees periodically sort-merge runs (or parts of a run) from a Level i with the overlapping runs from Level $i+1$. This process of data re-organization and creating fewer longer sorted runs on disk is known as *compaction*. However, the process of sort-merging data requires the data to be moved back and forth between the disk and main memory. This results in write amplification, which can be as high as 40× in state-of-the-art LSM-based data stores [46].

Partial compactions. To amortize data movement, and thus, avoid latency spikes, state-of-the-art LSM-engines organize data into smaller files, and perform compactions at the granularity of files instead of levels [28]. If Level i grows beyond a threshold, a compaction is triggered and one file (or a subset of files) from Level i is chosen to be compacted with files from Level $i+1$ that have an overlapping key-range. This process is known as *partial compaction*. Fig. 2 presents a comparative illustration of the full compaction and partial compaction routines in LSM-trees.

Querying LSM-Trees. Since LSM-trees realize updates and deletes in an out-of-place manner, multiple entries with the same key may exist in a tree with only the recent-most version being valid.

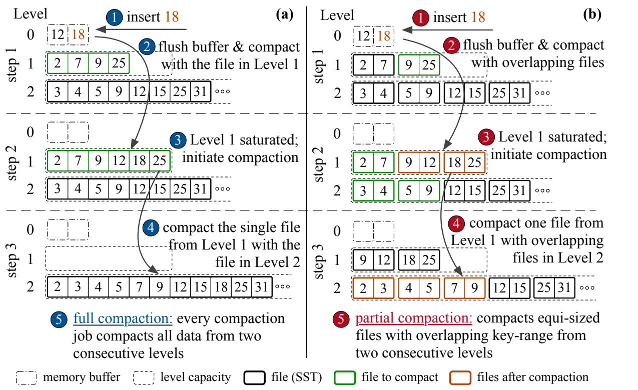
Point lookups. A point lookup starts at the memory buffer and traverses the tree from the smallest level to the largest one, and from the youngest to the oldest run within a level. A lookup terminates immediately after a matching key is found. To limit the number of runs a lookup probes, state-of-the-art LSM-engines use in-memory data structures, such as Bloom filters and fence pointers [23, 30].

Range scans. A range scan requires sort-merging the runs qualifying for a range query across all levels of the tree. The runs are sort-merged in memory and the latest version for each qualifying entry is returned while discarding all older, logically invalidated versions.

Deletes in LSM-Trees. A point delete operation is realized by inserting a special type of key-value entry, known as a *tombstone*, that *logically* invalidates the target entries without necessarily disturbing them. During compactions, a tombstone purges any older entries with a matching key. A delete is eventually considered as *persistent* once the corresponding tombstone reaches the last tree-level, at which point the tombstone can be safely dropped. The time taken to persistently delete a data object from an LSM-based data store depends on process of data re-organization. Compactions, thus, also play a critical role in timely and persistent deletion of entries, especially in light of the new data privacy regulations [1, 26, 38, 50, 54].

3 THE COMPACTION DESIGN SPACE

In this section, we identify the *design primitives* that provide a structured decomposition of arbitrary compaction strategies. This allows us to create the taxonomy of the universe of LSM compaction strategies, including all the classical as well as new ones.



Level
(a)
(b)
1 insert 18
2 flush buffer & compact with the file in Level 1
3 Level 1 saturated; initiate compaction
4 compact the single file from Level 1 with the file in Level 2
5 full compaction: every compaction job compacts all data from two consecutive levels
2 flush buffer & compact with overlapping files
4 compact one file from Level 1 with overlapping files in Level 2
5 partial compaction: compacts equi-sized files with overlapping key-range from two consecutive levels
step 1
step 2
step 3
memory buffer
level capacity
file (SST)
file to compact
files after compaction

Fig. 2: (a) When invoked, the classical full compaction routine compacts whole levels at a time, while (b) partial compactions perform compactions at the granularity of files.

3.1 Compaction Primitives

We define a compaction strategy as *an ensemble of design primitives that represents the fundamental decisions about the physical data layout and the data (re-)organization policy.* Each primitive answers a fundamental design question.

1) *Compaction trigger*: **When** to re-organize the data layout?
2) *Data layout*: **How** to lay out the data physically on storage?
3) *Compaction granularity*: **How much** data to move at-a-time during layout re-organization?
4) *Data movement policy*: **Which** block of data to be moved during re-organization?

Together, these design primitives define *when* and *how* an LSM-engine re-organizes the data layout on the persistent media. The proposed primitives capture any state-of-the-art LSM-compaction strategy and also enables synthesizing new or unexplored compaction strategies. Below, we define these four design primitives.

3.1.1 *Compaction Trigger.*

Compaction triggers refer to the set of events that can initiate a compaction job. The most common compaction trigger is based on the *degree of saturation* of a level in an LSM-tree [3, 30–32, 35, 56, 57]. The degree of saturation for Level i ($1 \le i \le L-1$) is typically measured as the ratio of the number of bytes of data stored in Level i to the theoretical capacity in bytes for Level i. Once the degree of saturation goes beyond a pre-defined threshold, one or more immutable files from Level i are marked for compaction. Some LSM-engines use the file count in a level to compute degree of saturation [32, 34, 35, 49, 55]. Note that the file count-based degree of saturation works only when all immutable files are of equal size, or for systems that have a tunable file size. The "#sorted runs" compaction trigger, triggers a compaction if the number of sorted runs (or "tiers") in a level goes past a predefined threshold, regardless of the size of a level.

Other compaction triggers include the *staleness of a file*, the *tombstone-based time-to-live*, and *space* and *read amplification*. For example, to ensure propagation of updates and deletes to the deeper levels of a tree, some LSM-engines assign a time-to-live (TTL) for each file during its creation. Each file can live in a level for a bounded time, and once the TTL expires, the file is marked for compaction [30]. Another delete-driven compaction trigger ensures

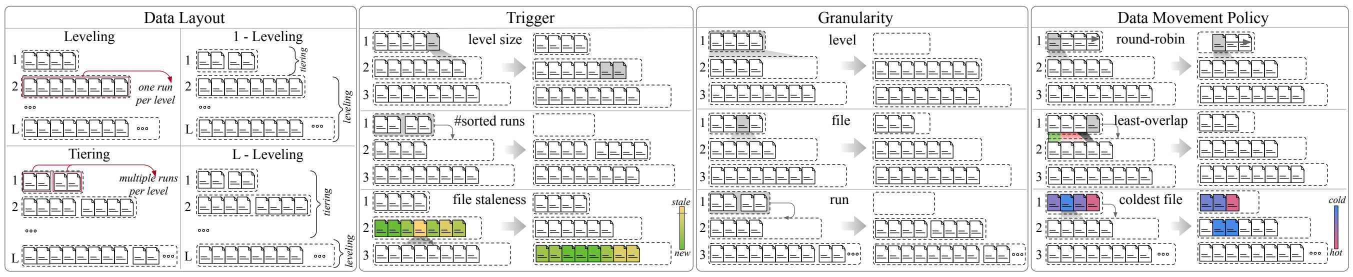


Fig. 3: The primitives that define LSM compactions: trigger, data layout, granularity, and data movement policy.

bounded persistence latency of deletes in LSM-trees through a different timestamp-based scheme. Each file containing at least one tombstone is assigned a special time-to-live in each level, and up on expiration of this timer, the file is marked for compaction [51]. Below, we present a list of the most common **compaction triggers**:

i) ***Level saturation***: level size goes beyond a nominal threshold
ii) ***#Sorted runs***: sorted run count for a level reaches a threshold
iii) ***File staleness***: a file lives in a level for too long
iv) ***Space amplification (SA)***: overall SA surpasses a threshold
v) ***Tombstone-TTL***: files have expired tombstone-TTL

3.1.2 ***Data layout***. The data layout is driven by the compaction eagerness, and determines the data organization on disk by controlling the number of sorted runs per level. Compactions move data between storage and memory, consuming a significant portion of the device bandwidth. There is, thus, an inherent competition for the device bandwidth between ingestion (external) and compaction (internal) – a trade-off depending on the eagerness of compactions.

The data layout is commonly classified as *leveling* and *tiering* [21, 22]. With leveling, once a compaction is triggered in Level i, the file(s) marked for compaction are merged with the overlapping file(s) from Level $i + 1$, and the result is written back to Level $i + 1$. As a result, Level $i + 1$ ends up with a (single) longer sorted run of immutable files [30–32, 34, 35, 56]. For tiering, each level may contain more than one sorted runs with overlapping key domains. Once a compaction is triggered in Level i, all sorted runs in Level i are merged together and the result is written to Level $i + 1$ as a new sorted run without disturbing the existing runs in that level [3, 7, 8, 30, 55, 57]. A hybrid design is proposed in Dostoevsky [23] where the last level is implemented as leveled and all the remaining levels on disk are tiered. A generalization of this idea is proposed in the literature as a continuum of designs [24, 37] that allows each level to separately decide between leveling and tiering. Among production systems, RocksDB implements the first disk-level (Level 1) as tiering [49], and it is allowed to grow perpetually in order to avoid write-stalls [11, 12, 14] in ingestion-heavy workloads. Below is a list of the most common options for **the data layout**:

i) ***Leveling***: one sorted run per level
ii) ***Tiering***: multiple sorted runs per level
iii) **1-*leveling***: *tiering* for Level 1; *leveling* otherwise
iv) ***L-leveling***: *leveling* for last level; *tiering* otherwise
v) ***Hybrid***: a level can be *tiering* or *leveling* independently

3.1.3 ***Compaction Granularity***. Compaction granularity refers to the amount of data moved during a single compaction job. One way to compact data is by sort-merging and moving all data from a level to the next level – we refer to this as *full compaction* [2, 3, 58, 62]. This results in periodic bursts of I/Os due to large data movement during compactions, and as a tree grows deeper, the latency spikes are exacerbated causing prolonged write stalls. To amortize the I/O costs due to compactions, leveled LSM-based engines employ *partial compaction* [30, 32, 34, 45, 51, 55], where instead of moving a whole level, a smaller granularity of data participates in every compaction. The granularity of data can be a single file [28, 34, 51] or multiple files [2, 3, 8, 45] depending on the system design and the workload. Note that, partial compaction does not radically change the total amount of data movement due to compactions, but amortizes this data movement uniformly over time, thereby preventing undesired latency spikes. A compaction granularity of "sorted runs" applies principally to LSMs with lazy merging policies. Once a compaction is triggered in Level i, all sorted runs (or tiers) in Level i are compacted together, and the resulting entries are written to Level $i + 1$ as a new immutable sorted run. Below, we present a list of the most common **compaction granularity** options:

i) ***Level***: all data in two consecutive levels
ii) ***Sorted runs***: all sorted runs in a level
iii) ***Sorted file***: one sorted file at a time
iv) ***Several sorted files***: several sorted files at a time

3.1.4 ***Data Movement Policy***. When *partial compaction* is employed, the data movement policy selects which file(s) to choose for compaction. While the literature commonly refers to this decision as *file picking policy* [27], we use the term *data movement* to generalize for any possible data movement granularity.

A naïve way to choose file(s) is at random or by using a round-robin policy [32, 35]. These data movement policies do not focus on optimizing for any particular performance metric, but help in reducing space amplification. To optimize for read throughput, many production data stores [30, 34] select the "coldest" file(s) in a level once a compaction is triggered. Another common optimization goal is to minimize write amplification. In this policy, files with the least overlap with the target level are marked for compaction [13, 27]. To reduce space amplification, some storage engines choose files with the highest number of tombstones and/or updates [30]. Another delete-aware approach introduces a tombstone-age driven file picking policy that aims to timely persist logical deletes [51]. Below, we present the list of the common **data movement policies**:

i) ***Round-robin***: chooses files in a round-robin manner
ii) ***Least overlapping parent***: file with least overlap with "parent"
iii) ***Least overlapping grandparent***: as above with "grandparent"
iv) ***Coldest***: the least recently accessed file
v) ***Oldest***: the oldest file in a level
vi) ***Tombstone density***: file with #tombstones above a threshold
vii) ***Tombstone-TTL***: file with expired tombstones-TTLs

Database	Data layout	Compaction Trigger					Compaction Granularity				Data Movement Policy							
		Level saturation	#Sorted runs	File staleness	Space amp.	Tombstone-TTL	Level	Sorted run	File (single)	File (multiple)	Round-robin	Least overlap (+1)	Least overlap (+2)	Coldest file	Oldest file	Tombstone density	Expired TS-TTL	N/A (entire level)
RocksDB [30], Monkey [22]	Leveling / 1-Leveling	✓		✓					✓	✓		✓		✓	✓	✓		
	Tiering		✓		✓	✓		✓										✓
LevelDB [32], Monkey (J.) [21]	Leveling	✓							✓		✓	✓	✓					
SlimDB [47]	Tiering	✓							✓	✓								✓
Dostoevsky [23]	*L*-leveling	✓^{L}	✓^{T}				✓^{L}	✓^{T}				✓^{L}						✓^{T}
LSM-Bush [24]	Hybrid leveling	✓^{L}	✓^{T}				✓^{L}	✓^{T}				✓^{L}						✓^{T}
Lethe [51]	Leveling	✓				✓			✓	✓		✓					✓	
Silk [11], Silk+ [12]	Leveling	✓							✓	✓	✓							
HyperLevelDB [35]	Leveling	✓							✓		✓	✓	✓					
PebblesDB [46]	Hybrid leveling	✓							✓	✓								✓
Cassandra [8]	Tiering		✓	✓		✓		✓										✓
	Leveling	✓				✓			✓	✓		✓				✓	✓	
WiredTiger [62]	Leveling	✓					✓											✓
X-Engine [34], Leaper [63]	Hybrid leveling	✓							✓	✓		✓				✓		
HBase [7]	Tiering		✓					✓										✓
AsterixDB [3]	Leveling	✓					✓											✓
	Tiering		✓					✓										✓
Tarantool [57]	*L*-leveling	✓^{L}	✓^{T}				✓^{L}	✓^{T}										✓
ScyllaDB [55]	Tiering		✓	✓		✓		✓										✓
	Leveling	✓				✓			✓	✓		✓				✓	✓	
bLSM [56], cLSM [31]	Leveling	✓							✓		✓							
Accumulo [6]	Tiering	✓	✓			✓		✓										✓
LSbM-tree [58, 59]	Leveling	✓					✓											✓
SifrDB [44]	Tiering	✓								✓								✓

Table 1: Compaction strategies in state-of-the-art systems. [✓^{L}: for levels with leveling; ✓^{T}: for levels with tiering.]

3.2 Compaction as an Ensemble of Primitives

Every compaction strategy takes one or more values for each of the four primitives. The trigger, granularity, and data movement policy are multi-valued primitives, whereas data layout is single-valued.

For example, a common LSM design [2] has a **leveled** LSM-tree (*data layout*) that compacts **whole levels** at a time (*granularity*) once a **level reaches a nominal size** (*trigger*). This design does not implement many subtle optimizations including partial compactions, and by definition, does not need a data movement policy. A more complex example is the compaction strategy for a **leveled** LSM-tree (*data layout*) in which compactions are performed at the *granularity* of a **file**. A compaction is *triggered* if either (a) a **level reaches its capacity** or (b) a **file containing tombstones is retained in a level longer than a pre-set TTL** [51]. Once triggered, the *data movement policy* chooses (a) **the file with the highest density of tombstones**, if there is one or (b) **the file with the least overlap with the parent level**, otherwise.

The Compaction Design Space Cardinality. Two compaction strategies are considered different from each other if they differ in at least one of the four primitives. Compaction strategies that differ in only one primitive, can have vastly different performance when subject to the same workload while running on identical hardware. Plugging in some typical values for the cardinality of the primitives, we estimate the cardinality of the compaction universe as $>10^4$, a vast yet largely unexplored design space. Table 1 shows a representative part of this space, detailing the compaction strategies used in more than twenty academic and production systems.

Compactions Analyzed. For our analysis and experimentation, we select ten representative compaction strategies that are prevalent in production and academic LSM-based systems. We codify and present these candidate compaction strategies in Table 2. `Full` represents the compaction strategy for leveled LSM-trees that compacts entire levels upon invocation. `LO+1` and `LO+2` denote two partial compaction routines that choose a file for compaction with the smallest overlap with files in the parent $(i+1)$ and grandparent $(i+2)$ levels, respectively. `RR` chooses files for compaction in a round-robin fashion from each level. `Cold` and `Old` are read-friendly strategies that mark the coldest and oldest file(s) in a level for compaction, respectively. `TSD` and `TSA` are delete-driven compaction strategies with triggers and data movement policies that are determined by the density of tombstones and the age of the oldest tombstone contained in a file, respectively. `Tier` represents a variant of tiered data layout, where compactions are triggered when either (a) the number of sorted runs in a level or (b) the estimated space amplification in the tree reaches certain thresholds. This interpretation of tiering is also referred to as *universal compaction* in systems like RocksDB [39, 49]. Finally, `1-Lvl` represents a hybrid data layout where the first disk level is realized as *tiered* while the others as *leveled*. This is the default data layout for RocksDB [39, 48].

4 BENCHMARKING COMPACTIONS

We now discuss our experimental platform, how we integrated new compactions policies, and our measurement methodology.

4.1 Standardization of Compaction Strategies

We choose RocksDB [30] as our experimental platform, as it (i) is open-source, (ii) is widely used across industry and academia, (iii) has a large active community. To ensure fair comparison we implement all compaction strategies under the same LSM-engine.

Implementation. We integrate our codebase into RocksDB v6.11.4. We assign to *compactions a higher priority than writes* to accurately benchmark them, while always maintaining the LSM structure [53].

Compaction Trigger. The default compaction trigger for (hybrid) leveling in RocksDB is level saturation [48], and for the universal compaction is space amplification [49]. RocksDB also supports delete-driven compaction triggers, specifically whether the #tombstones in a file goes beyond a threshold. We further implement a trigger based on the tombstones age to facilitate timely deletes [51].

Data layout. By default, RocksDB supports only two different data layouts: *hybrid leveling* (tiered first level, leveled otherwise) [48] and a variation of *tiering* (with a different trigger), termed *universal compaction* [49]. We also implement pure *leveling* by limiting the number of first-level runs to one, and triggering a compaction when the number of first-level files is more than one.

Compaction Granularity. The granularity for leveling is *file* and *sorted runs* for tiering. To implement classical leveling, we mark all files of a level for compaction. We ensure that ingestion may resume only after all the compaction-marked files are compacted thereby replicating the behavior of the full compaction routine.

Data Movement Policy. RocksDB (v6.11.4) provides four different data movement policies: a file (i) with least overlap with its parent level, (ii) least recently accessed, (iii) with the oldest data in a level, and (iv) that has more tombstones than a threshold. We also implement partial compaction strategies that choose a file (v) in a round-robin manner, (vi) with the least overlap with its grandparent level, and (vii) based on the age of the tombstones in a file.

Primitives	Full [3, 58, 62]	L0+1 [22, 30, 51]	Cold [30]	Old [30]	TSD [30, 34]	RR [31, 32, 35, 56]	L0+2 [32, 35]	TSA [51]	Tier [8, 33, 47]	1-Lvl [30, 39, 48]
Trigger	level saturation	level sat.	level sat.	level sat.	1. TS-density 2. level sat.	level sat.	level sat.	1. TS age 2. level sat.	1. #sorted runs 2. space amp.	1. #sorted runs^T 2. level sat.^{L}
Data layout	leveling	leveling	leveling	leveling	leveling	leveling	leveling	leveling	tiering	hybrid
Granularity	levels	files	files	files	files	files	files	files	sorted runs	1. sorted runs^T 2. files^L
Data movement policy	N/A	least overlap. parent	coldest file	oldest file	1. most tombstones 2. least overlap. parent	round-robin	least overlap. grandparent	1. expired TS-TTL 2. least overlap. parent	N/A	1. N/A^T 2. least overlap. parent^L

Table 2: Compaction strategies evaluated in this work. [^{L}: levels with leveling; ^{T}: levels with tiering.]

Designing the Compaction API. We expose the compaction primitives through a new API as configurable knobs. An application can configure the desired compaction strategy and initiate workload execution. The API also allows the application to change the compaction strategy for an existing database. Overall, our experimental infrastructure allows us (i) to ensure an identical underlying structure while setting the compaction benchmark, and (ii) to tune and configure the design of the LSM-engine as necessary.

4.2 Performance Metrics

We now present the performance metrics used in our analysis.

Compaction Latency. The compaction latency includes the time taken to (i) identify the files to compact, (ii) read the participating files to memory, (iii) sort-merge (and remove duplicates from) the files, (iv) write back the result to disk as new files, (v) invalidate the older files, and (vi) update the metadata in the manifest file [30]. *The RocksDB metric* `rocksdb.compaction.times.micros` *is used to measure the compaction latency.*

Write Amplification (WA). The repeated reads and writes due to compaction cause high WA [46]. We formally define WA as *the number of times an entry is (re-)written without any modifications to disk during its lifetime. We use the RocksDB metric* `compact.write.bytes` *and the actual data size to compute WA.*

Write Latency. Write latency is driven by the device bandwidth utilization, which depends on (i) write stalls due to compactions and (ii) the sustained device bandwidth. *We use the* `db.write.micros` *histogram to measure the average and tail of the write latency.*

Read Amplification (RA). RA is the ratio between the total number of disk pages read for point lookups and the pages that should be read *ideally. We use* `rocksdb.bytes.read` *to compute RA.*

Point Lookup Latency. Compactions determine the position of the files in an LSM-tree which affects point lookups on entries contained in those files. *Here, we use the* `db.get.micros` *histogram.*

Range Lookup Latency. The range lookup latency depends on the selectivity of the range query, but is affected by the data layout. *We also use the* `db.get.micros` *histogram for range lookups.*

Space Amplification (SA). SA depends on the data layout, compaction granularity, and the data movement policy. SA is defined as *the ratio between the size of logically invalidated entries and the size of the unique entries in the tree* [23]. *We compute SA using the size of the database and the size of the logically valid entries.*

Delete Performance. We measure the degree to which the tested compaction strategies persistently delete entries within a time-limit [51] in order to analyze the implications of compactions from a privacy standpoint [1, 26, 38, 50, 54, 60]. *We use the RocksDB file metadata* `age` *and a delete persistence threshold.*

4.3 Benchmarking Methodology

We now discuss the methodology for varying the key input parameters for our analysis: *workload* and the *LSM tuning*.

4.3.1 ***Workload.*** A typical key-value workload comprises of five primary operations: inserts, updates, point lookups, range lookups, and deletes. Point lookups target keys that may or may not exist in the database – we refer to these as *non-empty* and *empty point lookups*, respectively. Range lookups are characterized by their *selectivity*. To analyze the impact of each operation, we vary the *fraction* of each operation as well as their qualitative characteristics (i.e., selectivity and entry size). We further vary the *data distribution* of ingestion and queries focusing on (i) uniform, (ii) normal, and (iii) Zipfian distributions. Overall, our custom-built benchmarking suite is a superset of the influential YCSB benchmark [19] as well as the insert benchmark [15], and supports a number of parameters that are missing from existing workload generators, including deletes. Our workload generator exposes over 64 degrees of freedom, and is available via GitHub [52] for dissemination, testing, and adoption.

4.3.2 ***LSM Tuning.*** We further study the interplay of LSM tuning and compaction strategies. We consider questions like *which compaction strategy is appropriate for a specific LSM design and a given workload?* To answer such questions we vary in our experimentation key LSM tuning parameters, like (i) the memory buffer size, (ii) the block cache size, and (iii) the size ratio of the tree.

5 EXPERIMENTAL EVALUATION

We now present the key experimental results using the ten compaction strategies listed in Table 2.

Goal of the Study. Our analysis aims to answer the following three fundamental questions:

i) Performance implications: How do compactions affect the overall performance of LSM-engines?
ii) Workload influence: How do workload distribution and composition influence compactions, and thereby, the performance of LSM-engines?
iii) Tuning influence: What is the interplay between LSM compactions and tuning?

Ultimately, the goal of this study is to help practitioners and researchers to make informed decisions when deciding which compaction strategies to support and use in an LSM-based engine.

Experimental Setup. For our experiments, we use an AWS EC2 server with t2.2xlarge instances (virtualization: hardware virtual machine) [5]. Each virtual machine has 8 Intel Scalable Processors (vCPUs) at 3.0GHz, 32GB of DIMM RAM, 45MB of L3 cache, and runs Ubuntu 20.04 LTS. For storage, we attach a 40GB SSD volume with 4000 provisioned IOPS (volume type: io2) [4]. For experiments with data size larger than 16GB, we switch to a 500GB SSD.

Default Setup. Unless otherwise mentioned, all experiments are performed on a RocksDB setup with an LSM-tree of size ratio 10 [21, 27, 30]. The memory buffer is implemented as a hash skiplist [29]. The size of the write buffer is set to 8MB which can hold up to 512 16KB disk pages [21, 22, 27, 34]. Fence pointers are maintained for each disk page, and Bloom filters are constructed for every file with 10 bits memory allocated for every entry [22, 24]. Additionally, we have 8MB block cache (RocksDB default) assigned for data, filter, and index blocks [27]. To capture the true raw performance of RocksDB as an LSM-engine, we (i) assign compactions a higher priority than writes, (ii) enable direct I/Os for both read and write operations, (iii) limit the number of memory buffers (or memtables) to two (one immutable and one mutable), and (iv) set the number of background threads responsible for compactions to 1.

Workloads. Unless otherwise mentioned, ingestion and lookups are uniformly generated, and the average size of a key-value entry is 128B with 4B keys [20, 41, 43]. We vary the number of inserts, going up to 2^{28}. As compaction performance proves to be agnostic to data size, and in the interest of experimenting with many configurations, we perform our base experiments with 10M inserts [16, 21], both interleaved and serial with respect to lookups. Further specifications of the workloads are presented before each set of experiments.

Presentation. For each experiment, we present the primary observations (**O**) along with key takeaway (**TA**) messages. In the interest of space, we limit our discussion to the most interesting results. Further, note that `TSD` and `TSA`, fall back to `LO+1` in absence of deletes, and thus, are omitted from the experiments without deletes.

5.1 Performance Implications

We first analyze the implications of compactions on the ingestion, lookup, and overall performance of an LSM-engine.

5.1.1 *Data loading*.

In this experiment, we insert 10M key-value entries uniformly generated into an empty database to quantify the raw ingestion and compaction performance.

O1: Compactions Cause High Data Movement. Fig. 4(a) shows that the overall (read and write) data movement due to compactions is significantly larger than the actual size of the data ingested. Among the leveled LSM-designs, `Full` moves 63× (32× for reads and 31× for writes) the data originally ingested. The data movement is significantly smaller for `Tier`, however, it remains 23× of the data size. The data movement for `1-Lvl` is similar to that of the leveled strategies in partial compaction. These observations conforms with prior work [46], but also highlight the problem of *read amplification due to compactions* leading to poor device bandwidth utilization.

O2: Partial Compaction Reduces Data Movement at the Expense of Increased Compaction Count. We now shift our attention to the different variations of leveling. Fig. 4(a) shows that leveled partial compaction leads to 34%–56% less data movement than `Full`. The reason is twofold: (1) A file with no overlap with its parent level, is only logically merged. Such *pseudo-compactions* require simple metadata (file pointer) manipulation in memory, and no I/Os. (2) A smaller compaction granularity reducing overall data movement by choosing a file with (i) the least overlap, (ii) the most updates, or (iii) the most tombstones for compaction. . Specifically, `LO+1` (and `LO+2`) is designed to pick files with the least overlap with the parent $i+1$ (and grandparent $i+2$) level. They move 10%–23% less data than other partial compaction strategies.

Fig. 4(b) shows that the partial compaction strategies as well as `1-Lvl` perform 4× more compaction jobs than `Full`, which is equal to the number of tree-levels. Note that for an LSM-tree with partial compaction, every buffer flush triggers cascading compactions to all L levels, while in a full-level compaction system this happens when a level is full (every T compactions). Finally, since both `Tier` and `Full` are full-level compactions the compaction count is similar.

TA I: *Larger compaction granularity leads to fewer but larger compactions.* *Full-level compactions perform about $1/L$ times fewer compactions than partial compaction routines, however, full-level compaction moves nearly $2L$ times more data per compaction.*

O3: Full Leveling has the Highest Mean Compaction Latency. As expected, `Full` compactions have the highest average latency (1.2–1.9× higher than partial leveling, and 2.1× than tiering). The mean compaction latency is observed to be directly proportional to the average amount of data moved per compaction. `Full` can neither take advantage of pseudo-compactions nor optimize the data movement during compactions, hence, on average the data movement per compaction remains large. `1-Lvl` provides the most predictable performance in terms of compaction latency. Fig. 4(c) shows the mean compaction latency for all strategies as well as the median (P50), the 90^{th} percentile (P90), the 99^{th} percentile (P99), and the maximum (P100). The tail compaction latency largely depends on the amount of data moved by the largest compaction jobs triggered during the workload execution. We observe that the tail latency (P90, P99, P100) is more predictable for `Full`, while partial compactions, and especially, tiering have high variability due to differences in the data movement policies.

The compaction latency presented in Fig. 4(c) can be broken to IO time and CPU time. We observe that the CPU effort is about 50% regardless of the compaction strategy. During a compaction, CPU cycles are spent in (1) obtaining locks and taking snapshots, (2) merging the entries, (3) updating file pointers and metadata, and (4) synchronizing output files post compaction. Among these, the time spent to sort-merge the data in memory dominates.

The Tail Write Latency is Highest for Tiering. Fig. 4(d) shows that the tail write latency is highest for tiering. The tail write latency for `Tier` is ~2.5× greater than `Full` and 5–12× greater than partial compactions. Tiering in RocksDB [49] optimizes for writes and opportunistically seeks to compact all data to a large single level. This design achieves lower average write latency (Fig. 5(b)) at the expense of prolonged write stalls in the worst case, which is when the overlap between two consecutive levels is very high. `Full` also has 2–5× higher tail write stalls than partial compactions because when multiple consecutive levels are close to saturation, a buffer flush can result in a cascade of compactions. `1-Lvl` too has a higher tail write latency as the first level is realized as tiering.

TA II: `Tier` *may cause prolonged write stalls.* *Tail write stall for* `Tier` *is ~$25ms$, while for partial leveling (*`Old`*) it is as low as $1.3ms$.*

5.1.2 *Querying the Data*.

In this experiment, we perform 1M point lookups on the previously generated preloaded database (with

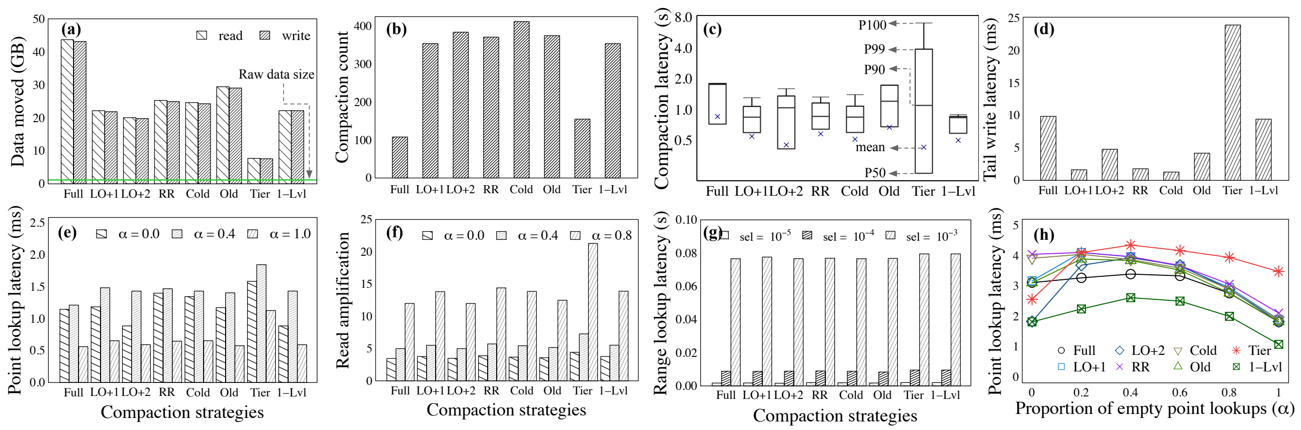


Fig. 4: Compactions influence the ingestion performance of LSM-engines heavily in terms of (a) the overall data movement, (b) the compaction count, (c) the compaction latency, and (d) the tail latency for writes, as well as (e, f) the point lookup performance. The range scan performance (g) remains independent of compactions as the amount of data read remains the same. Finally, the lookup latency (h) depends on the proportion of empty queries (α) in the workload.

10M entries). The lookups are uniformly distributed in the domain and we vary the fraction of empty lookups α between 0 and 1. Specifically, $\alpha = 0$ indicates that we consider only non-empty lookups, while for $\alpha = 1$ we have lookups on non-existing keys. We also execute 1000 range queries, while varying their selectivity.

O4: The Point Lookup Latency is Highest for Tiering and Lowest for Full-Level Compaction. Fig. 4(e) shows that point lookups perform the best for `Full`, and the worst for tiering. The mean latency for point lookups with tiering is between 1.1–1.9× higher than that with leveled compactions for lookups on existing keys, and ~2.2× higher for lookups on non-existing keys. Note that lookups on existing keys must always perform at least one I/O per lookup (unless they are cached). For non-empty lookups in a tree with size ratio T, theoretically, the lookup cost for tiering should be T× higher than its leveling equivalent [21]. However, this *worst-case* cost is not always accurate; in practice it depends on (i) the block cache size and the caching policy, (ii) the temporality of the lookup keys, and (iii) the implementation of the compaction strategies. RocksDB-tiering has overall fewer sorted runs than textbook tiering. Taking into account the block cache and temporality in the lookup workload, the observed tiering cost is less than T× the cost observed for `Full`. In addition, `Full` is 3%–15% lower than the partial compaction routines, because during normal operation of `Full` some levels might be entirely empty, while for partial compaction all levels are always close to being full. Finally, we note that the choice of data movement policy does not affect the point lookup latency significantly, which always benefits from Bloom filters (10 bits-per-key) and the block cache (0.05% of the data size).

Point Lookup Latency Increases for Comparable Number of Empty and Non-Empty Queries. A surprising result for point lookups that is also revealed in Fig. 4(e) is that they perform worse when the fraction of empty and non-empty lookups is balanced. Intuitively, one would expect that as we have more empty queries (that is, as α increases) the latency would decrease since the only data accesses needed by empty queries are the ones due to Bloom filter false positives [21]. To further investigate this result, we plot in Fig. 4(h) the 90th percentile ($P90$) latency which shows a similar curve for point lookup latency as we vary α. In our configuration each file uses 20 pages for its Bloom filters, 4 pages for its index blocks, and that the false positive is $FPR = 0.8\%$. A non-empty query needs to load the Bloom filters of the levels it visits until it terminates. For all intermediate levels, it accesses the index and data blocks with probability FPR, and then fetches the index and data blocks for the target level. On the other hand, an empty query probes the Bloom filters of all levels before returning an empty result. Note that for each level it also accesses the index and data blocks with FPR. The counter-intuitive shape is a result of the non-empty lookups not needing to load the Bloom filters for all levels when $\alpha = 0$ and the empty lookups accessing index and data only when there is a false positive when $\alpha = 1$. Fig. 4(h) also shows the highly predictable point lookup performance of `1-Lvl`.

TA III: *The point lookup latency is largely unaffected by the data movement policy.* *In presence of Bloom filters (with high enough memory) and small enough block cache, the point query latency remains largely unaffected by the data movement policy as long as the number of sorted runs in the tree remains the same. This is because block-wise caching of the filter and index blocks reduces the time spent performing disk I/Os significantly.*

O5: Read Amplification is Influenced by the Block Cache Size and File Structure, and is Highest for Tiering. Fig. 4(f) shows that the read amplification across different compaction strategies for non-empty queries ($\alpha = 0$) is between 3.5 and 4.4. This is attributed to the size of filter and index blocks which are 5× and 1× the size of a data block, respectively. Each non-empty point lookup fetches between 1 and L filter blocks depending on the position of the target key in the tree, and up to $L \cdot FPR$ index and data blocks. Further, the read amplification increases exponentially with α, reaching up to 14.4 for leveling and 21.3 for tiering (for $\alpha = 0.8$). Fig. 4(f) also shows that the estimated read amplification for point lookups is between 1.2× and 1.8× higher for `Tier` than for leveling strategies. This higher read amplification for `Tier` is owing to the larger number of sorted runs in the tree, and is in line with **O4**.

The Effect of Compactions on Range Scans is Marginal. To answer a range query, LSM-trees instantiate multiple *run-iterators*

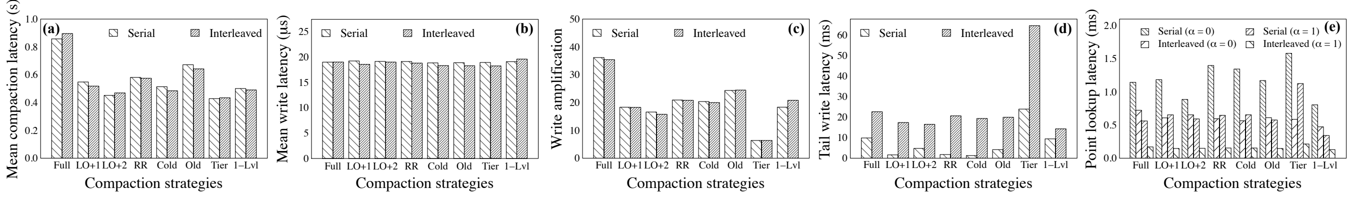


Fig. 5: (a-c) The average ingestion performance for workloads with interleaved inserts and queries is similar to that of an insert-only workload, but (d) with worse tail performance. However, (e) interleaved lookups are significantly faster.

scanning all sorted runs containing qualifying data. Thus, its performance depends on (i) the iterator scan time (which relates to selectivity) and (ii) the time to merge the data. The number of sorted runs in a leveled LSM-tree remains the same, which results in similar range query latency for all leveled variations, especially for larger selectivity (Fig. 4(g)). Note that without updates or deletes, the amount of data qualifying for a range query remains largely identical for different data layouts despite the number of runs being different. The ~5% higher average range query latency for `Tier` is attributed to the additional I/Os needed to handle partially qualifying disk pages from each run ($O(L \cdot T)$ in the worst case).

5.1.3 ***Executing mixed workloads.*** We now discuss the performance implications when ingestion and queries are mixed. We interleave the ingestion of 10M unique key-value entries with 1M point lookups. The ratio of empty to non-empty lookups is varied across experiments. All lookups are performed after $L - 1$ levels are full. Fig. 5 compares side by side the results for serial and interleaved execution of workloads with same specifications.

O6: Mixed Workloads have Higher Tail Write Latency. Figures 5(a) and (b) show that the mean latency of compactions that are interleaved with point queries is only marginally affected for all compaction strategies. This is also corroborated by the write amplification remaining unaffected by mixing reads and writes as shown in Fig. 5(c). On the other hand, Fig. 5(d) shows that the tail write latency is increased between 2–15×. This increase is attributed to (1) the need of point queries to access filter and index blocks that requires disk I/Os that compete with writes and saturate the device, and (2) the delay of memory buffer flushing during lookups.

Interleaving Compactions and Point Queries Helps Keeping the Cache Warm. Since in this experiment we start the point queries when $L - 1$ levels of the tree are full, we expect that the interleaved read query execution will be faster than the serial one, by $1/L$ (25% in our configuration) which corresponds to the difference in the height of the trees. However, Fig. 5(e) shows this difference to be between 26% and 63% for non-empty queries and between 69% and 81% for empty queries. The reasons interleaved point query execution is faster than expected are that (1) about 10% of lookups terminate within the memory buffer, without requiring any disk I/Os, and (2) the block cache is pre-warmed with filter, index, and data blocks cached during compactions. Fig. 5(d) and 5(e) show how `1-Lvl` brings together *the best of both worlds* and offer reasonably good ingestion and lookup performance simultaneously.

TA IV: ***Compactions help lookups by warming up the caches.*** *As the file metadata is updated during compactions, the block cache is warmed up with the filter, index, and data blocks, which helps subsequent point lookups.*

5.2 Workload Influence

Next, we analyze the implications of the workloads on compactions.

5.2.1 ***Varying the Ingestion Distribution.*** In this experiment, we use an interleaved workload that varies the ingestion distribution (*Zipfian* with $s = 1.0$, *normal* with 34% standard deviation), and has uniform lookup distribution. We use a variant of the Zipfian distribution, called *PrefixZipf*, where the key prefixes follow a Zipfian distribution while the suffixes are generated uniformly. This allows us to avoid having too many updates in the workload.

Ingestion Performance is Agnostic to Insert Distribution. Figures 4(a), 6(a), and 6(e) show that the total data movement during compactions remains virtually identical for (unique) insert-only workloads generated using uniform, PrefixZipf, and normal distributions, respectively. Further, we observe that the mean and tail compaction latencies are agnostic of the ingestion distribution (Fig. 4(c), 6(b), and 6(f) are almost identical as well). As long as the data distribution does not change over time, the entries in each level follow the same distribution and the overlap between different levels remains the same. *Therefore, for an ingestion-only workload the data distribution does not influence the choice of compaction strategy.*

O7: Insert Distribution Influences Point Queries. Figure 6(c) shows that while tiering has a slightly higher latency for point lookups, the relative performance of the compaction strategies is close to each other for any fraction of non-empty queries in the workload (all values of α). This is because when empty queries are drawn uniformly from the key domain, the level-wise metadata and index blocks help to entirely avoid a vast majority of unnecessary disk accesses (including fetching index or filter blocks). In Fig. 6(d), we observe that the read amplification remains comparable to that in Fig. 4(f) (*uniform* ingestion) for $\alpha = 0$ and even $\alpha = 0.4$. However, for $\alpha = 0.8$, the read amplification in Fig. 6(d) becomes 65%-75% smaller than in the case of uniform inserts. The I/Os performed to fetch the filter blocks is close to zero. *This shows that all compaction strategies perform equally well while executing an empty query-heavy workload on a database pre-populated with PrefixZipf inserts.* In contrast, when performing lookups on a database pre-loaded with normal ingestion, the point lookup performance (Fig. 6(g)) largely resembles its uniform equivalent (Fig. 4(h)), as the ingestion-skewness is comparable. The filter and index block hits are ~ 10% higher for the normal distribution compared to uniform for larger values of α, which explains the comparatively lower read amplification shown in Fig. 6(h). This plot also shows the first case of unpredictable behavior of `L0+2` for $\alpha = 0$ and $\alpha = 0.2$. We observe more instances of such unpredictable behavior for `L0+2`, which probably explains why it is rarely used in new LSM stores. Once again, for both the compaction and tail lookup performance, `1-Lvl` offers highly predictable performance.

5.2.2 ***Varying the Point Lookup Distribution.*** In this experiment, we change the point lookup distribution to **Zipfian** and **normal**, while keeping the ingestion distribution as uniform.

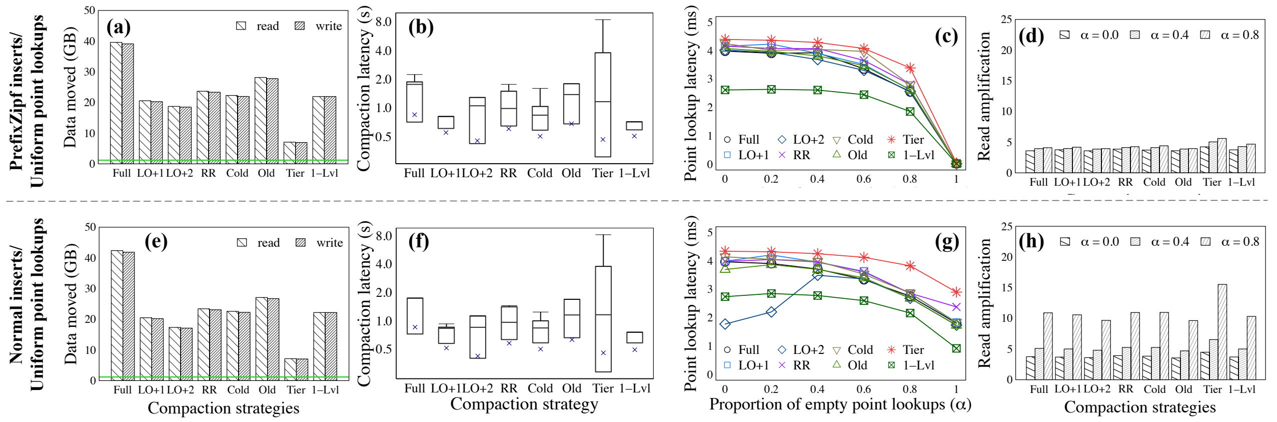


Fig. 6: As the ingestion distribution changes to (a-d) PrefixZipf and (e-h) normal with standard deviation, the ingestion performance of the database remains nearly identical with improvement in the lookup performance.

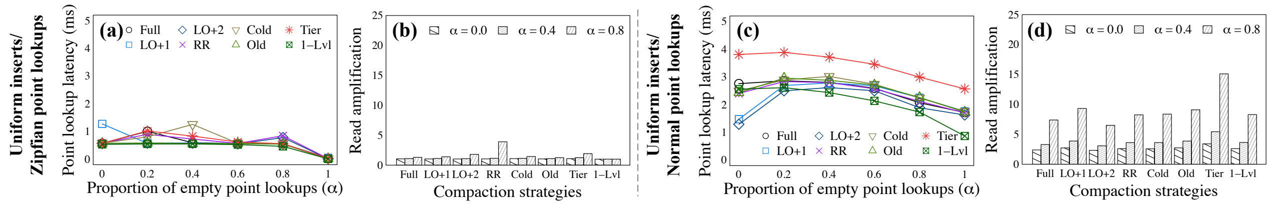


Fig. 7: Skewed lookup distributions like Zipfian (a, b) and normal (c, d) improve the lookup performance dramatically in the presence of a block cache and with the assistance of Bloom filters.

The Distribution of Point Lookups Significantly Affects Performance. Zipfian point lookups on uniformly populated data leads to low latency point queries for all compaction strategies, as shown in Fig. 7(a) because the block cache is enough for the popular blocks in all cases, as also shown by the low read amplification in Fig. 7(b). On the other hand, when queries follow the normal distribution, partial compaction strategies `LO+1` and `LO+2` dominate all other approaches, while `Tier` is found to perform significantly slower than all other approaches, as shown in Fig. 7(c) and 7(d).

TA V: ***For skewed ingestion/lookups, all compaction strategies behave similarly in terms of lookup performance.*** *While the ingestion distribution does not influence its performance, heavily skewed ingestion or lookups impacts query performance due to block cache and file metadata.*

5.2.3 *Varying the Proportion of Updates.*

We now vary the update-to-insert ratio, while interleaving queries with ingestion. An update-to-insert ratio 0 means that all inserts are unique, while a ratio 8 means that each unique insert receives 8 updates on average.

O8: For Higher Update Ratio Compaction Latency for Tiering Drops; `LO+2` Dominates the Leveling Strategies. As the fraction of updates increases, the mean compaction latency decreases significantly for tiering because we discard multiple updated entries in every compaction (Fig. 8(a)). We observe similar but less pronounced trends for `Full` and `LO+2`, while the remaining leveling strategies remain largely unchanged. *Overall, larger compaction granularity helps to exploit the presence of updates by invalidating more entries at a time.* Among the leveling strategies, `LO+2` performs best as it moves ~20% less data during compactions, which also affects write amplification as shown in Fig. 8(b).

As the fraction of updates increases, all compaction strategies including `Tier` have lower tail compaction latency. Fig. 8(c) shows that `Tier`'s tail compaction latency drops from 6× higher than `Full` to 1.2× for an update-to-insert ratio of 8, which demonstrates that `Tier` is most suitable for update-heavy workloads. We also observe that lookup latency and read amplification also decrease for update-heavy workloads.

The Point Lookup Latency Stabilizes with the Level Count. Fig. 8(d) shows that as the update-to-insert ratio increases, the mean point lookup latency decreases sharply before stabilizing. The initial sharp fall in the latency is attributed to a decrement in the number of levels (from 4 to 3) in the LSM-tree, when the update-to-insert ratio increases from 0.4 to 1. The latency then stabilizes because non-empty point lookups perform at least one disk I/O, which, in turn, dominates the overall lookup cost.

TA VI: ***Tiering dominates the performance for update-intensive workloads.*** *When subject to update-intensive workloads,* `Tier` *exhibits superior compaction performance along with comparable lookup performance (as leveled LSMs), which allows it to dominate the overall performance space.*

5.2.4 *Varying Delete Proportion.*

We now analyze the impact of deletes, which manifest as out-of-place invalidations with special entries called tombstones [51]. We keep the same data size and vary the proportion of point deletes in the workload. All deletes are issued on existing keys and are interleaved with the inserts.

`TSD` and `TSA` Offer Superior Delete Performance. We quantify the efficacy of deletion using the number of tombstones at the end of the workload execution. The lower this number, the faster deleted data has been purged from the database, which in turn reduces space, write, and read amplification. Fig. 8(e) shows that `TSD` and `TSA` maintain the fewer tombstones at the end of the experiment. For a workload with 10% deletes, `TSD` purges 16% more tombstones than `Tier` and 5% more tombstones than `LO+1` by picking the files that have a tombstone density above a pre-set threshold for compaction.

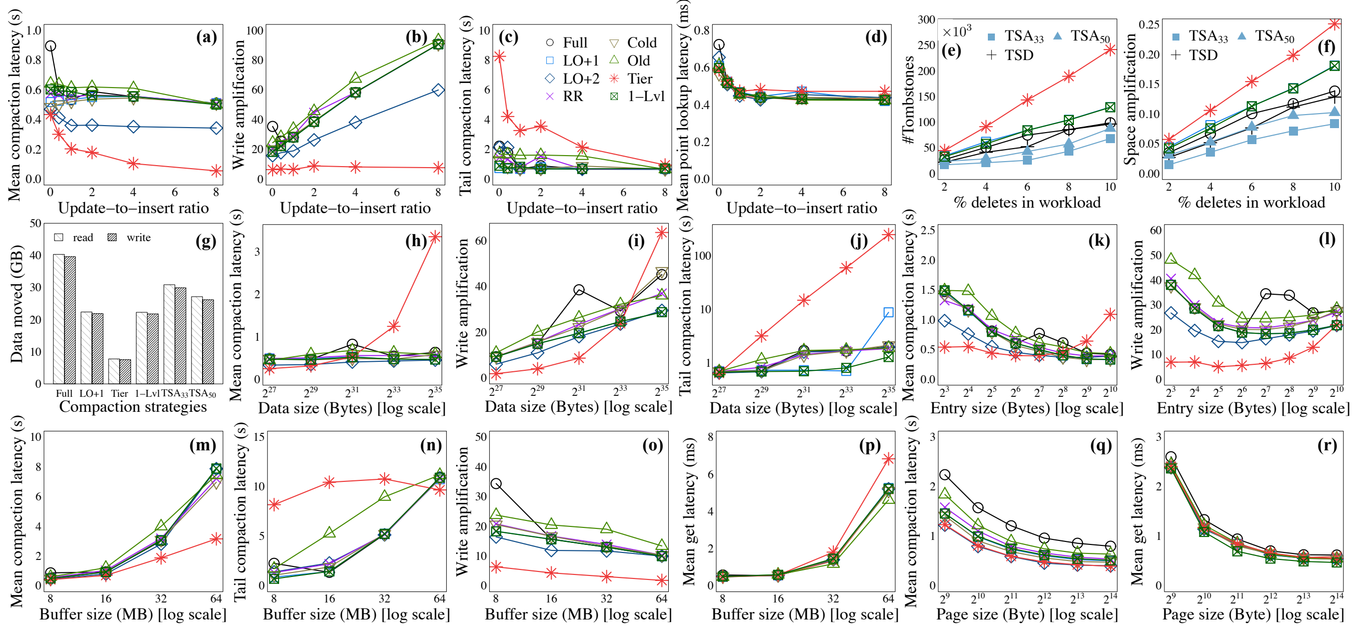


Fig. 8: Experiments with varying workload and data characteristics (a-l) and LSM tuning (m-r) show that there is no perfect compaction strategy – choosing the appropriate compaction strategy is subject to the workload and the performance goal.

For TSA, we experiment with two different thresholds for delete persistence: TSA_{33} and TSA_{50} is set to 33% and 50% of the experiment run-time, respectively. As TSA guarantees persistent deletes within the thresholds set, it compacts more data aggressively, and ends up with 7–10% fewer tombstones as compared to TSD. Full manages to purge more tombstones than any partial compaction routine, as it periodically compacts entire levels. Tier retains the highest number of tombstones as it maintains the highest number of sorted runs overall. As the proportion of deletes in the workload increases, the number of tombstones remaining the LSM-tree (after the experiment is over) increases. TSA and TSD along with Full scale better than the partial compaction routines and tiering. By compacting more tombstones, TSA and TSD also purge more invalid data reducing space amplification, as shown in Fig. 8(f).

O9: Optimizing for Deletes Comes at a (Write) Cost. The reduced space amplification offered by TSA and TSD is achieved by compacting the tombstones eagerly, which increases the overall amount of data moved due to compaction. Fig. 8(g) shows that TSD and TSA_{50} compacts 18% more data than the write optimized LO+1 (for TSA_{33} this becomes 35%). Thus, TSD and TSA are useful when the objective is to (i) persist deletes timely or (ii) reduce space amplification caused by deletes.

TA VII: TSD *and* TSA *are tailored for deletes.* *TSA and TSD, by design, choose files with tombstones for compactions to reduce space amplification. TSA ensures timely persistent deletion by compacting more data eagerly for smaller persistence thresholds, which increases the write amplification.*

5.2.5 *Varying the Ingestion Count.*

We now report the scalability results by varying the data size from 2^{27}B to 2^{35}B.

O10: Tier Scales Poorly Compared to Leveled and Hybrid Strategies. The mean compaction latency scales sub-linearly for all compaction strategies barring Tier, as shown in Fig. 8(h). *The relative advantages of compaction strategies with leveled and hybrid data layouts remain similar regardless of the data size.* This observation is further backed up by Fig. 8(i) which shows how write amplification scales. We also observe that the advantages of the RocksDB-implementation of tiering (i.e., *universal compaction*) [49] diminishes as the data size grows beyond 8GB. Fig. 8(j) shows that as the data size increases, the tail compaction latency for Tier increases, as the worst-case overlap between files from consecutive levels increase significantly. This makes Tier unsuitable for latency-sensitive applications. When the data size reaches 2GB, Full triggers a *cascading compaction* that writes all data to a new level, causing spikes in write amplification and compaction latency.

5.2.6 *Varying Entry Size.*

Here, we keep the key size constant (4B) and vary the value from 4B to 1020B to vary the entry size.

O11: For Smaller Entry Size, Leveling Compactions are More Expensive. Smaller entry size increases the number of entries per page, which in turn, leads to (i) more keys to be compared during merge and (ii) bigger Bloom filters that require more space per file and more CPU for hashing. Fig. 8(k) shows these trends. We also observe similar trends for write amplification in Fig. 8(l) and for query latency. They both decrease as the entry size increases. However, as the overall data size increases with the entry size, we observe the compaction latency and write amplification to increase steeply for Tier (similarly to Fig. 8(h) and (i)).

5.3 LSM Tuning Influence

In the final part of our analysis, we discuss the interplay of compactions with the standard LSM tunings knobs, such as memory buffer size, page size, and size ratio.

O12: Compactions with Tiering Scale Better with Buffer Size. Fig. 8(m) shows that as the buffer size increases, the mean compaction latency increases across all compaction strategies. The size of buffer dictates the size of the files on disk, and larger file size leads to more data being moved per compaction. Also, for larger file size, the filter size per file increases along with the time spent for hashing, which increases compaction latency. Further, as the buffer size increases, the mean compaction latency for `Tier` scales better than the other strategies. Fig. 8(n) shows that the high tail compaction latency for `Tier` plateaus quickly as the buffer size increases, and eventually crossovers with that for the eagerer compaction strategies when the buffer size becomes 64MB.

We also observe in Fig. 8(o) that among the partial compaction routines `Old` experiences an increased write amplification throughout, while `L0+1` and `L0+2` consistently offer lower write amplification and guarantee predictable ingestion performance. Fig. 8(p) shows that as the memory buffer size increases, the mean point lookup latency increases superlinearly. This is because, for larger memory buffers, the files on disk hold a greater number of pages, and thereby, more entries. Thus, the combined size of the index block (one index per page) and filter block (typically, 10 bits per entry) per file grows proportionally with the memory buffer size. The time elapsed in fetching the index and filter blocks causes the mean latency for point lookups to increase significantly.

All Compaction Strategies React Similarly to Varying the Page Size. In this experiment, we vary the logical page size, which in turn, changes the number of entries per page. The smaller the page size, the larger the number of pages per file – meaning more I/Os are required to access a file on the disk. For example, when the page size shrinks from 2^{10}B to 2^9B, the number of pages per file doubles. With smaller page size, the index block size per file increases as more pages should be indexed, which also contributes to the increasing I/Os. Thus, an increase in the logical page size, reduces the mean compaction latency, as shown in Fig. 8(q). In Fig. 8(r), we observe that as the page size increases, the size of the index block per file decreases, and on average fewer I/Os are performed to fetch the metadata block overall for every point lookup.

Miscellaneous Observations. We also vary LSM tuning parameters such as the size ratio, the memory allocated to Bloom filters, and the size of the block cache. We observe that changing the values of these knobs affects the different compaction strategies similarly, and hence, does not influence the choice of the appropriate compaction strategy for any particular set up.

6 DISCUSSION

The design space detailed in Section 3 and the experimental analysis presented in Section 5 aim to offer to database researchers and practitioners the necessary insights to make educated decisions when selecting compaction strategies for LSM-based data stores.

Know Your LSM Compaction. LSM-trees are considered "write-optimized", however, in practice their performance strongly depends on *when and how compactions are performed.* We depart from the notion of *treating compactions as a black-box*, and instead, we formalize *LSM compactions* as an ensemble of four fundamental compaction primitives. This allows us to reason about each of these primitives and navigate the *LSM compaction design space* in search of the appropriate compaction strategy for a workload or for custom performance goals. Further, the proposed compaction design space provides the necessary intuitions about how simple modifications (like data movement policy or compaction granularity) to an existing engine (like RocksDB) can be key to achieving significant performance improvement or cost benefits. For instance, RocksDB can modularize their compaction implementation by decoupling the code logic for every primitive. This will not only expose the primitives as tunable knobs, but will facilitate synthesizing and testing new compaction algorithms tailored to a developer's requirements.

Avoiding the Worst Choices. We discuss how to avoid common pitfalls. For example, tiering is often considered as the write-optimized variant, however, we show that it comes with high tail latency, making it unsuitable for applications that need worst-case performance guarantees. Also, applications requiring stable performance should avoid `L0+2` due to its unpredictable performance. On the other hand, partial compactions with leveling, and especially, hybrid leveling (e.g., `1-Lvl`) offer the most stable performance.

Adapting with Workloads. In prior work tiering is used for write-intensive use-cases, while leveling offers better read performance. However, in practice, in mixed HTAP-style workloads, lookups have a strong temporal locality, and are essentially performed on recent hot data. In such cases, the block cache is frequently proved to be enough for holding the working set and eliminate the need for other costly optimizations for read queries.

Exploring New Compaction Strategies. Ultimately, this work lays the groundwork for exploring the vast design space of LSM compactions. A key intuition we developed during this analysis is that contrary to existing designs, LSM-based systems can benefit by employing different compaction primitives at different levels, depending on the exact workload and the performance goals. The compaction policies we experimented with already support a wide range of metrics they optimize for including system throughput, worst-case latency, read, space, and write amplification, and delete efficiency. Using the proposed design space, new compaction strategies can be designed with new or combined optimization goals. We also envision systems that automatically select compaction strategies on the fly depending on the current context and workload.

7 CONCLUSIONS

LSM-based engines offer efficient ingestion and competitive read performance, while being able to manage various optimization goals like write and space amplification. A key internal operation that is at the heart of how LSM-trees work is the process of *compaction* that periodically re-organizes the data on disk.

We present the *LSM compaction design space* that uses four primitives to define compactions: (i) compaction trigger, (ii) the data layout, (iii) compaction granularity, and (iv) the data movement policy. We map existing approaches in this design space and we select several representative policies to study and analyze their impact on performance and other metrics including write/space amplification and delete latency. We present an extensive collection of observations, and we lay the groundwork for LSM systems that can more flexibly navigate the design space for compactions.

ACKNOWLEDGMENT

We thank the reviewers for their valuable feedback. We are particularly thankful to Guanting Chen for his contributions in the early stages of this project. This work was partially funded by National Science Foundation under Grant No. IIS-1850202 and a Facebook Faculty Research Award.

REFERENCES

[1] California Consumer Privacy Act of 2018. *Assembly Bill No. 375, Chapter 55*, 2018.

[2] W. Y. Alkowaileet, S. Alsubaiee, and M. J. Carey. An LSM-based Tuple Compaction Framework for Apache AsterixDB. *Proceedings of the VLDB Endowment*, 13(9):1388–1400, 2020.

[3] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. R. Borkar, Y. Bu, M. J. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y.-S. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V. J. Tsotras, R. Vernica, J. Wen, and T. Westmann. AsterixDB: A Scalable, Open Source BDMS. *Proceedings of the VLDB Endowment*, 7(14):1905–1916, 2014.

[4] Amazon. EBS volume types. *https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-volume-types.html.*

[5] Amazon. EC2 Instance Types. *https://aws.amazon.com/ec2/instance-types/.*

[6] Apache. Accumulo. *https://accumulo.apache.org/.*

[7] Apache. HBase. *http://hbase.apache.org/.*

[8] Apache. Cassandra. *http://cassandra.apache.org*, 2021.

[9] M. Athanassoulis and S. Idreos. Design Tradeoffs of Data Access Methods. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Tutorial*, 2016.

[10] M. Athanassoulis, M. S. Kester, L. M. Maas, R. Stoica, S. Idreos, A. Ailamaki, and M. Callaghan. Designing Access Methods: The RUM Conjecture. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 461–466, 2016.

[11] O. Balmau, F. Dinu, W. Zwaenepoel, K. Gupta, R. Chandhiramoorthi, and D. Didona. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 753–766, 2019.

[12] O. Balmau, F. Dinu, W. Zwaenepoel, K. Gupta, R. Chandhiramoorthi, and D. Didona. SILK+ Preventing Latency Spikes in Log-Structured Merge Key-Value Stores Running Heterogeneous Workloads. *ACM Trans. Comput. Syst.*, 36(4):12:1–12:27, 2020.

[13] M. Callaghan. Compaction priority in RocksDB. *http://smalldatum.blogspot.com/2016/02/compaction-priority-in-rocksdb.html*, 2016.

[14] M. Callaghan. Compaction stalls: something to make better in RocksDB. *http://smalldatum.blogspot.com/2017/01/compaction-stalls-something-to-make.html*, 2017.

[15] M. Callaghan. The Insert Benchmark. *http://smalldatum.blogspot.com/2017/06/the-insert-benchmark.html*, 2017.

[16] Z. Cao, S. Dong, S. Vemuri, and D. H. C. Du. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 209–223, 2020.

[17] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 205–218, 2006.

[18] CockroachDB. Open Isuue: Storage: Performance degradation caused by kv tombstones. *https://github.com/cockroachdb/cockroach/issues/17229*, 2017.

[19] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pages 143–154, 2010.

[20] Y. Dai, Y. Xu, A. Ganesan, R. Alagappan, B. Kroth, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 155–171, 2020.

[21] N. Dayan, M. Athanassoulis, and S. Idreos. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 79–94, 2017.

[22] N. Dayan, M. Athanassoulis, and S. Idreos. Optimal Bloom Filters and Adaptive Merging for LSM-Trees. *ACM Transactions on Database Systems (TODS)*, 43(4):16:1–16:48, 2018.

[23] N. Dayan and S. Idreos. 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*, pages 505–520, 2018.

[24] N. Dayan and S. Idreos. The Log-Structured Merge-Bush & the Wacky Continuum. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 449–466, 2019.

[25] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. *ACM SIGOPS Operating Systems Review*, 41(6):205–220, 2007.

[26] A. Deshpande and A. Machanavajjhala. ACM SIGMOD Blog: Privacy Challenges in the Post-GDPR World: A Data Management Perspective. *http://wp.sigmod.org/?p=2554*, 2018.

[27] S. Dong. Option of Compaction Priority. *https://rocksdb.org/blog/2016/01/29/compaction_pri.html*, 2016.

[28] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum. Optimizing Space Amplification in RocksDB. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2017.

[29] Facebook. MemTable. *https://github.com/facebook/rocksdb/wiki/MemTable*, 2021.

[30] Facebook. RocksDB. *https://github.com/facebook/rocksdb*, 2021.

[31] G. Golan-Gueta, E. Bortnikov, E. Hillel, and I. Keidar. Scaling Concurrent Log-Structured Data Stores. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 32:1–32:14, 2015.

[32] Google. LevelDB. *https://github.com/google/leveldb/*, 2021.

[33] HBase. Online reference. *http://hbase.apache.org/*, 2013.

[34] G. Huang, X. Cheng, J. Wang, Y. Wang, D. He, T. Zhang, F. Li, S. Wang, W. Cao, and Q. Li. 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*, pages 651–665, 2019.

[35] HyperLevelDB. Online reference. *https://github.com/rescrv/HyperLevelDB.*

[36] S. Idreos and M. Callaghan. Key-Value Storage Engines. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 2667–2672, 2020.

[37] S. Idreos, N. Dayan, W. Qin, M. Akmanalp, S. Hilgard, A. Ross, J. Lennon, V. Jain, H. Gupta, D. Li, and Z. Zhu. Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2019.

[38] T. Kraska, M. Stonebraker, M. L. Brodie, S. Servan-Schreiber, and D. J. Weitzner. SchengenDB: A Data Protection Database Proposal. In *Heterogeneous Data Management, Polystores, and Analytics for Healthcare - VLDB 2019 Workshops, Poly and DMAH, Los Angeles, CA, USA, August 30, 2019, Revised Selected Papers*, volume 11721 of *Lecture Notes in Computer Science*, pages 24–38, 2019.

[39] A. Kryczka. Compaction Styles. *https://github.com/facebook/rocksdb/blob/gh-pages-old/talks/2020-07-17-Brownbag-Compactions.pdf*, 2020.

[40] LinkedIn. Voldemort. *http://www.project-voldemort.com.*

[41] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 133–148, 2016.

[42] C. Luo and M. J. Carey. LSM-based Storage Techniques: A Survey. *The VLDB Journal*, 29(1):393–418, 2020.

[43] S. Luo, S. Chatterjee, R. Ketsetsidis, N. Dayan, W. Qin, and S. Idreos. Rosetta: A Robust Space-Time Optimized Range Filter for Key-Value Stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 2071–2086, 2020.

[44] F. Mei, Q. Cao, H. Jiang, and J. Li. SifrDB: A Unified Solution for Write-Optimized Key-Value Stores in Large Datacenter. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*, pages 477–489, 2018.

[45] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[46] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 497–514, 2017.

[47] K. Ren, Q. Zheng, J. Arulraj, and G. Gibson. SlimDB: A Space-Efficient Key-Value Storage Engine For Semi-Sorted Data. *Proceedings of the VLDB Endowment*, 10(13):2037–2048, 2017.

[48] RocksDB. Leveled Compaction. *https://github.com/facebook/rocksdb/wiki/Leveled-Compaction*, 2020.

[49] RocksDB. Universal Compaction. *https://github.com/facebook/rocksdb/wiki/Universal-Compaction*, 2020.

[50] S. Sarkar, J.-P. Banâtre, L. Rilling, and C. Morin. Towards Enforcement of the EU GDPR: Enabling Data Erasure. In *Proceedings of the IEEE International Conference of Internet of Things (iThings)*, pages 1–8, 2018.

[51] S. Sarkar, T. I. Papon, D. Staratzis, and M. Athanassoulis. Lethe: A Tunable Delete-Aware LSM Engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 893–908, 2020.

[52] S. Sarkar, D. Staratzis, Z. Zhu, and M. Athanassoulis. K-V Workload Generator. *https://github.com/BU-DiSC/K-V-Workload-Generator*, 2021.

[53] S. Sarkar, D. Staratzis, Z. Zhu, and M. Athanassoulis. LSM Compaction Analysis. *https://github.com/BU-DiSC/LSM-Compaction-Analysis*, 2021.

[54] M. Schwarzkopf, E. Kohler, M. F. Kaashoek, and R. T. Morris. Position: GDPR Compliance by Construction. In *Selected Papers from VLDB Workshop on Polystore Systems for Heterogeneous Data in Multiple Databases with Privacy and Security Assurances (POLY)*, volume 11721 of *Lecture Notes in Computer Science*, pages 39–53, 2019.

[55] ScyllaDB. Online reference. *https://www.scylladb.com/.*

[56] R. Sears and R. Ramakrishnan. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 217–228, 2012.

[57] Tarantool. Online reference. *https://www.tarantool.io/.*

[58] D. Teng, L. Guo, R. Lee, F. Chen, S. Ma, Y. Zhang, and X. Zhang. LSbM-tree: Re-Enabling Buffer Caching in Data Management for Mixed Reads and Writes. In *37th IEEE International Conference on Distributed Computing Systems, ICDCS*

2017, Atlanta, GA, USA, June 5-8, 2017, pages 68–79, 2017.

[59] D. Teng, L. Guo, R. Lee, F. Chen, Y. Zhang, S. Ma, and X. Zhang. A Low-cost Disk Solution Enabling LSM-tree to Achieve High Performance for Mixed Read/Write Workloads. *ACM Trans. Storage*, 14(2):15:1—-15:26, 2018.

[60] L. Wang, J. P. Near, N. Somani, P. Gao, A. Low, D. Dao, and D. Song. Data Capsule: A New Paradigm for Automatic Compliance with Data Privacy Regulations. In *Heterogeneous Data Management, Polystores, and Analytics for Healthcare - VLDB 2019 Workshops, Poly and DMAH, Los Angeles, CA, USA, August 30, 2019, Revised Selected Papers*, volume 11721 of *Lecture Notes in Computer Science*, pages 3–23, 2019.

[61] WiredTiger. Merging in WiredTiger's LSM Trees. *https://source.wiredtiger.com/develop/lsm.html*, 2021.

[62] WiredTiger. Source Code. *https://github.com/wiredtiger/wiredtiger*, 2021.

[63] L. Yang, H. Wu, T. Zhang, X. Cheng, F. Li, L. Zou, Y. Wang, R. Chen, J. Wang, and G. Huang. Leaper: A Learned Prefetcher for Cache Invalidation in LSM-tree based Storage Engines. *Proceedings of the VLDB Endowment*, 13(11):1976–1989, 2020.