Dissecting, Designing, and Optimizing LSM-based Data Stores

Subhadeep Sarkar
Boston University, MA, USA
ssarkar1@bu.edu

Manos Athanassoulis
Boston University, MA, USA
mathan@bu.edu

ABSTRACT

Log-structured merge (LSM) trees have emerged as one of the most commonly used disk-based data structures in modern data systems. LSM-trees employ out-of-place ingestion to support high throughput for writes, while their immutable file structure allows for good utilization of disk space. Thus, the log-structured paradigm has been widely adopted in state-of-the-art NoSQL, relational, spatial, and time-series data systems. However, despite their popularity, there is a lack of pedagogical textbook-like material on LSM designs. The goal of this tutorial is to present the fundamental principles of the LSM paradigm along with a digest of optimizations and new designs proposed in recent research and adopted by modern LSM engines. This will serve as introductory material for non-experts, and as a roadmap to cutting-edge LSM results for the LSM-aware researchers and practitioners.

Toward this, we first discuss in detail the basic operations (inserts, updates, deletes, point and range queries), their access patterns, and their paths through the LSM data structure. We then dive into the details of recent research on optimizing each of those operations. We first discuss techniques and designs that optimize data ingestion in LSM-trees and the performance tradeoff constructed by writes and reads for the LSM engines. Finally, we present the rich design space of the log-structured paradigm and outline how to navigate it and tune LSM-based systems. We conclude with a discussion on open challenges on LSM systems. This will be a 1.5-hour tutorial.

CCS CONCEPTS

• **Information systems → Data layout**; **Data access methods**; **Data structures**; **Key-value stores**.

KEYWORDS

LSM-tree, NoSQL, Storage engine, Design space, Compaction, Write optimization, Data layout, Tuning

ACM Reference Format:
Subhadeep Sarkar and Manos Athanassoulis. 2022. Dissecting, Designing, and Optimizing LSM-based Data Stores. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3514221.3522563

1 INTRODUCTION

Driving Application Trends. Modern data store designs are driven by three fundamental trends. (A) Modern applications are storing increasingly more data, which is further fueled by the decreasing price of storage and memory. (B) As a result, many data stores perform more insertions than read queries in their lifetime [115]. (C) The global move to cloud-based data management further supports immutability-based systems [51]. Putting these trends together, a large number of applications opt to employ at their storage layer log-structured merge-trees, a highly ingestion-optimized design.

Log-Structured Merge-Trees (LSM-Trees). LSM-Trees are designed to support high ingestion throughput for disk-based systems [93]; however, this comes at the cost of lower read performance or additional memory/storage footprint [13, 14]. To address this, the LSM paradigm has been heavily optimized since its inception. The founding idea of LSM-trees is to **batch writes in memory** and write on disk only large chunks of sorted data in order to achieve *high write throughput*. The files on disk are never edited in place, rather, **updates and deletes** are applied **lazily** on the **immutable files** that allows the files to be compact. Finally, in order to offer competitive read performance, the design is augmented with **auxiliary in-memory data structures** for each immutable file.

LSM-Trees are Everywhere. The superior ingestion performance and the highly tunable nature of the LSM paradigm, has turned LSMs as one of the most popular storage paradigms for modern data stores including state-of-the-art relational and NoSQL data stores, as well as other systems like spatial [65, 86, 123] and time-series data systems [60, 67]. For example, LevelDB [49] and BigTable [25] at Google, RocksDB [44] at Facebook, X-Engine [53] at Alibaba, Voldemort [77] at LinkedIn, Dynamo [36] at Amazon, AsterixDB [7] Cassandra [11], HBase [10], and Accumulo [9] at Apache, and bLSM [115] and cLSM [48] at Yahoo, all rely on LSM-trees.

A Rich Design Space. A closer look into the internals of LSM-engines reveal numerous design decisions in the works. These decisions determine the structure of a tree (data layout), the periodicity of merges/compactions, the granularity of merges/compactions, distribution of main memory between the buffer and other auxiliary in-memory components, the decision of which auxiliary data structures to use, and even when to physically delete invalidated data objects. Such design choices influence the performance of LSM-engines along several axes; but, are often abstracted from the user due to their complexity and the design-intricacies involved.

Goal of the Tutorial. In this tutorial, we open the black-box of LSM-based systems, and we present each design decision in detail. This deep understanding of the internals of LSM-trees will allow young graduate students, seasoned researchers, and practitioners to better understand how to build and tune an LSM system, and enable them to get an overview of numerous optimizations that have been proposed and adopted in state-of-the-art LSM systems.

Tutorial Overview

We will provide a 1.5-hour tutorial, broken down into 3 modules.

Module I: LSM Basics. (*25 mins*) The first module provides the necessary background on LSM-trees along with an account of the optimization techniques adopted in state-of-the-art LSM-engines to facilitate efficient query processing.

(i) *Basic structure and operating principles* (5 mins). Outline the in-memory and disk-based components of LSM-based data stores and discuss their operating principles [31, 39, 83, 93, 113].

(ii) *LSM operations* (10 mins). Present the workflows for the internal operations (*flush* and *compaction*) [39, 83, 111, 113], and the external operations (*put*, *get*, *scan*, and *delete*) [32, 33, 83, 93, 109].

(iii) *Point and range query optimizations* (10 mins). Outline the different optimization techniques used in LSM-engines to facilitate efficient lookups [31, 32, 35, 37, 76, 84, 131, 132].

Module II: Optimizing Ingestion. (*45 mins*) The second module presents an account of optimizations adopted across state-of-the-art LSM-engines to improve the overall ingestion performance.

(i) *In-memory optimizations* (5 mins). Summarize the modifications made to the in-memory LSM-component in order to facilitate faster ingestion [11, 24, 43, 44, 49, 53, 114].

(ii) *Disk data layout optimizations* (15 mins). Present the disk data layout optimization techniques, including hybrid data layouts [26, 33–35, 56, 102, 127], separating keys from values [30, 56, 78], and partitioning the key space [54, 81, 98].

(iii) *Optimizing compactions* (10 mins). Discuss different compaction design choices (*how many* and *which data blocks to compact*), and their impact on ingestion [6, 7, 19, 38, 71, 111, 113].

(iv) *Compaction design space* (10 mins). Present the LSM compaction design space and provide key insights on the performance implications of compactions [95, 111, 113, 129, 130, 133].

(v) *Enabling parallelism* (5 mins). Present the techniques that exploit thread-level and device-level parallelism to improve ingestion [28, 43, 61, 70, 74, 75, 107, 121, 122, 124, 134, 135].

Module III: Tuning and Navigating the LSM Design Space. (*20 mins*) In the final module, we present an account of research on tunability and navigability of the LSM design space for ensuring optimal design choices for diverse workloads.

(i) *Performance space* (5 mins). Outline the performance space of LSM-engines along with the inherent multi-way tradeoff between the read, write, and delete costs, the memory footprint, and resource utilization [13, 14, 23, 32, 38, 112].

(ii) *Navigating the LSM design space* (5 mins). Outline techniques that navigate the read-write tradeoff curve based on workload composition [32, 34, 57–59, 79, 82].

(iii) *Robust LSM tunings* (5 mins). Discuss techniques for robust LSM tunings that can offer predictable performance in case of workload-shifts [55, 81].

(iv) *Privacy through timely data deletion* (5 mins). Present insights on designing privacy-aware LSM-engines that protects data privacy through timely persistent data deletion [85, 101, 112].

(v) Finally, conclude with a discussion on the opportunities and open challenges in LSM research.

Output. The expected outcome of this tutorial are as follows.

- Understanding of the log-structured merge (LSM) paradigm and its operating principles.
- Understanding the tradeoff between reads, writes, and memory, and their implications on performance of an LSM-engine.
- Insights about the LSM write-path, the data layout re-organization policies, and the optimizations adopted to optimize writes.
- Insights about the point and range read-paths and the role of auxiliary in-memory data structures in optimizing reads.
- Appreciation for the different LSM tuning knobs and construction of the LSM design space from the tuning knobs, along with intuitions about how to navigate this design space subject to workload changes and performance requirements.
- Exposure to open research questions on the LSM paradigm.

To our knowledge, this is the first tutorial that (i) explores the log-structured merge paradigm to (ii) construct and analyze the LSM design space and (iii) provides useful insights about LSM tunings to optimize performance. There is also a lack of textbook materials/supplements that discuss the LSM paradigm. We believe, this tutorial will benefit researchers and practitioners to better understand the operating principles and performance tradeoffs associated with the LSM paradigm, and thus, help make informed decisions to extract near-optimal performance from LSM engines.

2 TUTORIAL NARRATIVE

2.1 Module I: LSM Basics

LSMs as Index Structure. The LSM paradigm was introduced by O'Neil *et al.* as a low-cost disk-based index data structure to sustain high ingestion rates over an extended period of time [93]. However, commercial adoption of LSM-trees for indexing began after nearly a decade with Google's Bigtable [25]. The resurgence of LSM-trees can be attributed to the birth of big data-centric technologies, such as NoSQL systems [44, 73, 90, 92, 116], cloud computing [8, 29, 46, 50, 62], mobile computing [52, 96], and the Internet of things [63, 68, 110]. With an ever-growing data volume, in need for fast ingestion and analysis, LSM-trees offered multi-faceted benefits over existing index structures, and thus, were adopted quickly as a storage layer solution for many NoSQL [6, 10, 11, 44, 49, 53, 88, 99, 115] and relational [12, 42, 72, 87, 125] systems.

Basic Structure. LSM-trees store data in form of key-value pairs, where a *key* refers to a unique object identifier, and the data associated with it, is referred to as *value*. When storing schema-based relational data, the primary key of a relation acts as the key, and the remaining attributes together constitute the value. The entries in an LSM-tree are typically sorted and accessed based on the key.

LSM Components. LSM-trees were conceptualized to have a hierarchical data layout with one tree-like component in memory and a larger second component on disk [93]. Typically, for an LSM-tree with L levels, the first level (Level 0) is retained in memory and the remaining levels (Level 1 to $L-1$) are disk-resident [33]. The data in the in-memory structure is moved to the disk-component iteratively through a process of data layout re-organization.

2.1.1 *Operating Principles.*

Below, we present the key operating principles of LSM-based storage engines.

A. Batched Ingestion. Incoming inserts, updates, or deletes are first buffered within the in-memory component of an LSM-tree. Once the buffer reaches capacity, the entries are sorted by the key and are moved to the first tree-level (Level 0) on disk. Batching writes in memory (i) enables high throughput for writes, while (ii) facilitating fast lookups for workloads with temporality.

B. Out-of-Place Updates and Deletes. LSM-trees follow the out-of-place paradigm by design. Updates and deletes are handled as new inserts, and are applied lazily to the base data. This allows for high write throughput, as unlike in-place read-modify-writes, out-of-place updates and deletes do not incur write-stalls.

C. Immutable File Structure. On disk, LSM-trees maintain the data in *immutable sorted files*. This means, modifications to an entry entails re-writing of the corresponding file anew, with the older file being marked for garbage collection. Immutable file structure allows LSM-tree to achieve high disk space utilization, as within a file, entries are written compactly without any free space maintained.

D. Periodic Data Layout Re-Organization. Each level on disk is assigned a capacity which typically grows exponentially with the levels. When a level reaches capacity, all or part of its data is sort-merged with data from the next level with an overlapping key-range. This process of data layout re-organization is called *compaction*. Compactions (i) bound the number of sorted components or *runs* on disk, thereby, facilitating fast lookups, while (ii) reducing space amplification through periodic garbage collection.

E. LSM Invariant. A key property is that the contents of a Level i are more recent than the contents of the deeper, larger Level $i + 1$.

2.1.2 Basic Operations. We classify the basic LSM-operations into two classes: (i) internal operations and (ii) external operations.

Internal operations refer to the operations triggered by a storage engine in order to re-structure the data layout: (i) *flushing* the memory buffer to disk, and (ii) *compacting* the sorted runs on disk.

Flush. Every time the memory buffer reaches capacity, a new buffer is created to receive the next batch of inserts, to avoid write stalls. Entries in the older buffer are sorted on the key, and are written to disk as an immutable file. This process is called *flushing* [83].

Compaction. Every time a level has an incoming immutable file (note that flushing creates incoming immutable files for Level 1) the LSM-tree checks whether the current level goes beyond its capacity. If this is the case, all or part of the data from that level is moved to the next level. As data in different levels can be overlapping in the key domain, the participating entries are merged, retaining only the latest version of each key. This process, termed *compaction*, limits the number of sorted runs in a tree, and in the process, garbage collects logically invalidated entries [39, 83].

Compaction Policies. Classically, LSM-trees support two compaction policies: leveling and tiering [111, 113]. 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 merged with the run from Level i, if it exists. With *tiering*, every level must accumulate multiple runs before they are merged. In §2.2, we discuss state-of-the-art hybrid compaction strategies and their performance implications.

External operations refer to the operations that are triggered by an application. The fundamental external operations supported by LSM-based key-value stores are *puts*, *gets*, *scans*, and *deletes*.

Put. LSM-trees ingest data in an out-of-place manner; thus, updates are treated similarly to inserts. Incoming entries are first *put* into the memory buffer, before being moved to disk in an opportunistic way. Update to a key that exists in the buffer, immediately replaces the older entry in place; otherwise, the update is propagated to the disk and is lazily applied to the logically invalidated target data.

Get. A *get* or *point lookup* returns the most recent version of an entry of the desired key. 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, as *the LSM-invariant ensures that the latest version of an entry is always retained in the youngest sorted run containing an entry with a matching key.*

Scan. A *range lookup* or *scan* returns the most recent versions of all keys within a range. To facilitate range queries, all sorted runs in an LSM-tree are scanned, and merged, while returning only the latest version for each key. Typically, during range lookups, an iterator is assigned for each run, and the runs are scanned in parallel.

Delete. LSM-trees realize *deletes* logically by inserting a special type of entry called a *tombstone*. A tombstone contains the key to be deleted, and its short (typically, only a byte-long) value field is used to distinguish it from regular key-value entries. The logically invalidated entries are garbage collected only after they are compacted with a matching tombstone. Before this, both the invalidate entries and the tombstone co-exist in different files of the database.

2.1.3 Optimizing Reads. The write-optimized design of LSM-trees leads to suboptimal read performance. To ameliorate this, state-of-the-art LSM-engines employ several in-memory light-weight auxiliary data structures, such as filters, indexes, and block caches.

Indexing and Block-based Caching. Without help from any auxiliary data structures, LSM-trees would perform several superfluous disk I/Os for every lookup. Thus, virtually any LSM-tree design is supported by **fence pointers** (a special form of Zonemaps [89]), that store information about the smallest and largest keys in every disk page [39]. Such light-weight data structures are typically pre-fetched to memory in an opportunistic way. To further improve the performance of in-memory key search [126], several advanced indexing techniques have been proposed in the literature [5, 24, 30, 47, 66, 69]. Several approaches have also focussed on optimizing reads on secondary (non-key) attributes through **secondary indexing** techniques [64, 80, 86, 97, 117, 118, 136].

Another way of improving lookup performance is by using **block-level caching** [105]. Commercial LSM engines use a block cache (which, for example, defaults at 12GB for RocksDB) that can be configured to retain in memory the first few levels of a tree, the frequently accessed hot data blocks, and/or even the filter and index blocks [39]. Since, compactions involve a lot of data movement, it is rather frequent that the hot data pages are evicted from block cache during compactions [119]. To address this problem and retain hot pages in memory, Leaper introduces an ML-aided predictive mechanism to identify pages from recently compacted files and **prefetch those in block cache** immediately after compaction [128].

Point Query Filters. In the worst-case, even in presence of fence pointers, a point lookup may need to probe every sorted run in a tree [39, 83], leading to superfluous I/Os. Thus, to further reduce the cost of point lookups, state-of-the-art LSM-engines maintain **Bloom filters** in memory [44]. Bloom filters are maintained at the granularity of sorted runs, and allow a lookup to skip probing a run altogether, if the filter-lookup returns negative [39]. Dayan *et al.* **optimizes the memory allocation to filters of different tree-levels** to minimize the expected number I/O cost for point lookups [31, 32]. Several new filter designs and LSM-specific filter optimization techniques have been proposed, including ElasticBF [76] that addresses access skew by employing **multiple small filter units per Bloom filter**. Ribbon filter [37] introduces a better **tradeoff of index time vs. space utilization** at the expense of additional CPU work. Chucky [35] builds a **cuckoo filter-based single updatable index** that serves as both a filter and an index, while the **hash sharing** [137] technique re-uses the same hash-digest across levels during lookups, to reduce the overall CPU cost. Note that other approximate set membership data structures can also be potentially useful as Bloom filter replacements [18, 27, 45].

Range Query Filters. LSMs are by design not optimized for range queries as data within a given range can be scattered across all levels of a tree. Therefore, range filters are crucial to prevent unnecessary disk access while executing range queries on LSMs [84, 103, 131, 132]. The study of range filters on LSMs can be broadly divided into two parts: optimizing for long and short range queries. Prefix filters **use fixed-length key-prefixes** to answer long range membership queries [103]. SuRF [131, 132] is a **succinct trie-based filter** that supports storing variable length prefixes of keys, thus, allowing fewer false positives for long range queries. Rosetta [84] introduces a **range filter comprising of a hierarchy of Bloom filters** that can logically construct a segment tree to detect differences in longer prefixes, which is a better fit for short range queries.

Optimizing Memory Allocation. The memory allocation between the buffer and the filters can be tuned to maximize read performance. Monkey [31] points out that as the number of entries grows exponentially with levels, the cost of data movement due to false positives becomes higher in shallower levels. Monkey proposes an **optimal memory allocation** strategy (i) across the Bloom filters within a tree and (ii) between the buffer and the filters to navigate the RUM tradeoff space. Chucky [35] discusses the same with variable hash bucket sizes for **succinct cuckoo filters** on LSMs. More research results on balancing both read and write performance by allocating memory appropriately are discussed in the next module.

2.2 Module II: Optimizing Writes

The high ingestion throughput offered by LSM-trees comes at the cost of increased write amplification [20]. There have been several lines of work that aim to improve the overall ingestion performance by (i) buffer-level modifications, (ii) changing the data layout, (iii) tuning the compaction algorithms, and/or (iv) increasing parallelism. In this module, we present an account of such endeavors that improve ingestion performance in LSM-based storage engines.

2.2.1 In-Memory Optimizations. Varying the **number of buffer components** in memory allows for accumulation of more entries before flushing is required. This allows LSM-engines to withstand heavy bursts of ingestion without hurting the tail write latency [42, 44, 49]. In addition, many LSM-engines also allow developers to **change the buffer size** to ameliorate write stalls and improve ingestion throughput [11, 24, 44, 53, 114, 125].

Another design optimization adopted in commercial engines varies the **implementation of the buffer** based on the workload and performance requirement. For example, RocksDB allows developers to implement the memory buffer as a (i) *vector*, (ii) *skiplist*, (iii) *hash-skiplist*, or (iv) *hash-linkedlist* [43], each of which offers very different performance. A *vector* implementation offers the highest ingestion throughput for write-only workloads; however, its performance degrades in presence of interleaved reads. A skip-list buffer offers better performance for such mixed workloads.

2.2.2 Disk Data Layouts. To avoid merging the buffer components with the runs on disk after every flush, a **tiered variant of the LSM** was introduced by Apache Cassandra [11]. The tiered design, with multiple sorted runs with overlapping key-ranges in a level, reduces data movement due to compactions. This allows for (i) faster data ingestion and (ii) reduced write amplification; but, comes at the cost of (iii) increased query cost and (iv) increased space amplification, as the tiered design has more sorted runs overall [10, 11, 22, 104, 113, 114]. Recent research has proposed a new set of **hybrid data layouts** where the shallower level(s) have a tiered layout and the larger and lower levels have a leveled layout [33, 56, 102, 127]. By default, RocksDB has tiering in the first level and leveling in the rest, as this allows for withstanding bursts of ingestion [102]. Dostoevsky introduces an LSM-structure where only the last level is leveled, with all the intermediate levels as tiered [33].

Reducing disk accesses, and in turn, data movement from disk, has also been a key line of research that aims to reduce write amplification [30, 57, 78]. WiscKey introduces an SSD-conscious data layout by **decoupling the storage of keys from values** [78]. The LSM-tree simply stores the keys along with pointers to the values, while the values are stored in a separate log file. This significantly reduces (4×) write amplification during ingestion, while facilitating up to 100× faster data loading. Idreos *et al.* introduced a continuum of storage layouts, which includes **access frequency-based hybrid index structures** with LSM-based indexes created on the hot data and B-tree based indexes on colder, larger levels [57].

Another way to reduce data movement is by **partitioning the key space** and storing the partitions in separate trees [98]. PebblesDB proposes to partition the key domain and introduces a fragmented LSM-structure that improves the ingestion throughput by reducing the overall data movement during compactions [54, 81, 91, 98]. Nova-LSM uses a similar partitioning algorithm to **shard the data across multiple storage components** in a distributed framework, thereby, reducing superfluous data movement [54].

2.2.3 Tuning Compactions. While some LSM-engines, such as AsterixDB [6, 7], compact all data in a level during compactions, such strategies entail heavy bursts of disk I/Os periodically, causing prolonged, undesired write stalls. Thus, many state-of-the-art LSM-engines have adopted a **partial compaction** strategy, where data is stored in multiple files within a level [44, 49, 53, 112]. During compactions, one (or few) file is compacted at a time, amortizing the I/O cost for compactions by reducing data movement [113].

For systems with partial compaction, the design decision on **which file(s) to compact** affects ingestion performance [19, 21]. Performing compactions at a smaller granularity allows for picking files with the least overlap with the next level for compaction [38, 71]. For delete-intensive workloads [23], picking files with the highest density of tombstones purges the logically invalidated entries early, thus, reducing write amplification [40, 41, 71, 113].

Among system-specific solutions to improve ingestion performance, LSM-based systems allow developers to **assign writes a higher priority than compactions**. This prevents write stalls to some degree; however, accumulating buffers without flushing them and sorted runs without compacting them, violates the LSM data layout, and would incur bursts of disk I/Os in the future, causing latency spikes [100]. To prevent write stalls, some LSM engines perform **compactions using background threads** [44, 125]. **Background compactions** are useful for I/O intensive scenarios; however, this is a highly system-aware operation, and enabling it without due analysis may cause slow down. Balmau *et al.* introduced a **bandwidth scheduler to avoid inference between flush and compaction**, thereby, preventing write stalls [16, 17].

2.2.4 The LSM Compaction Design Space. In prior work, we highlight that compactions affect the performance of LSM-engines in terms of ingestion, point and range lookups, space and write amplification, and deletes [113]. Toward this, we introduce a set of first-order **compaction primitives**: (i) the *compaction trigger*, (ii) the *data layout*, (iii) the *compaction granularity*, and (iv) the *data movement policy*. These primitives formally define any existing or completely new compaction strategies. We will summarize the experimental evaluation of multiple compaction strategies to provide key insights about (i) the effects of the compaction primitives, (ii) the impact of workload composition and distribution on compactions, and (iii) the impact of LSM tuning on compactions. Other efforts on compaction performance optimization include **grouped compaction** strategies [133], **light-weight compaction** policies [129, 130], and **delayed opportunistic compaction** strategies [95].

2.2.5 Increasing Parallelism. Finally, with the emergence of new storage devices and multi-core systems, several efforts have aimed to exploit newer, faster storage and compute architectures to make LSMs better. Several systems allow for **multi-threaded flushes and compactions** to accelerate ingestion [43, 70, 107, 124]. Partitioning and sharding-based solutions also take advantage to intra-node and inter-node operational parallelism to enhance performance. LSM-designs tuned for SSD/NVM devices achieve superior ingestion performance by **batching writes and performing compactions opportunistically** [28, 61, 74, 75, 120–122, 134, 135].

2.2.6 Other Endeavors. Chandramouli *et al.* introduces FASTER, a log-structured storage, that improves the *read-modify-write* performance [24]. Along with a log-structured storage, FASTER maintains an **in-memory hash table that maps keys to disk blocks**. FASTER achieves significantly better read performance at the price of a higher memory footprint and a higher cost for range queries. State-of-the-art systems also support read-modify-write operations, which are particularly useful for stream processing use cases [106].

2.3 Module III: LSM Tuning and Navigating the LSM Design Space

The RUM conjecture highlights the inherent three-way tradeoff constructed by the **R**ead cost, the **U**pdate cost, and the **M**emory footprint [13, 14]. Any given design presents a navigable tradeoff in terms of the RUM costs, which can be tuned to best match the expected workload to get the optimal performance of the specific design. Commercial LSM-engines expose hundreds of tuning knobs to the developers, and together, these variable components constitute the LSM design space. Navigating the LSM design space is critical; however, the vastness of this design space makes this process complex and extremely difficult. Toward this, recent approaches have attempted to break down the LSM black box to understand the implications of the different LSM components, operations, and tuning knobs on performance. This module presents an account of such efforts along with a discussion on open research challenges.

2.3.1 Navigating the Read-Write Tradeoff. The notion of breaking data structure designs into *first-order primitives* was first introduced in Data Calculator [58, 59], and this has been pivotal in exploration of the LSM-tree design space. Dayan *et al.* identified that allocation of main memory plays a critical role on LSM performance, and that, by design, LSM-trees suboptimally co-tune the data layout, the memory buffer size, and the filter memory size [32]. The work introduces a **workload-aware main memory allocation** technique that determines the optimal main memory size and filter size for different tree-levels based on the proportion of ingestion, and empty and non-empty point lookups. By doing so, this work moves the read-write tradeoff curve closer to the Pareto optimal.

The work on the design continuum [57] outlines a **larger design space for LSMs** by exploring different design elements, in terms of (i) the disk data layout, (ii) the data access patterns, and (iii) main memory allocation. Luo *et al.* proposes to optimally **allocate memory between the memory buffer and the block cache** to improve query performance [79, 82]. LSM-Bush puts everything together and introduces a **continuum for data layouts**, where an LSM-tree can be configured to have an arbitrary number of sorted runs in each level [34]. Finally, Cosine breaks away from worst-case based cost modeling and proposes (i) *distribution-aware I/O models* and (ii) *learning-based concurrency models* facilitating **accurate navigation of the LSM design space** [26].

2.3.2 Robust LSM Designs. Navigating the read-write tradeoff to tune LSM-trees allows to achieve the best possible performance for a given workload. This approach has been utilized repeatedly in the research discussed in Sections 2.2, and 2.3.1. However, the advent of new volatile applications and the increasing adoption of shared infrastructure (e.g., private or public clouds), add a degree of uncertainty between the expected and the observed workloads. To address this, recent research proposed a **robust LSM tuning** that formulates the tuning problem as a *min-max problem*, where the goal is to minimize the worst-case performance in a *neighborhood* of the expected workload [55]. In addition to robust tuning, stable performance depends also on the implementation of compaction. Luo *et al.* [81] proposed a **throttling mechanism for compactions** to guarantee that the merging devices operate just at the point prior to saturation. This leads to predictably stable performance, alleviating the instability due to overloading underlying storage.

2.3.3 Privacy-Aware LSM Designs. Next, we discuss the importance of designing privacy-aware data systems with an emphasis on LSM-engines. Similarly to other out-of-place systems, LSM-trees realize deletes through logical invalidation, and this has critical implications on data privacy. Lethe [112] introduces a new family of compaction strategies that **persistently delete logically invalidated data objects within a threshold duration**; thereby, complying with the legal regulations for *timely data deletion* [1–4, 15, 108, 109]. For workloads with unique inserts only, practical systems provide an API for **single deletes**, which removes a tombstone after it is compacted with the first matching key [101]. While some systems also support **range delete** operations, current implementations fail to provide latency bounds on persistent data deletion [85].

2.3.4 Open Challenges. In the final part of the tutorial, we present the opportunities and open challenges in LSM research.

(1) *Reducing write amplification.* Despite recent efforts, LSM designs continue to suffer from high write amplification. Optimizing write amplification without breaking the notion of immutability, remains a key goal.
(2) *Workload-aware compaction.* Identifying optimal compaction strategies based on workload and LSM-tuning is an open research problem. A first step toward this would involve extensive workload-aware modeling for each compaction primitive.
(3) *Data layout transformation.* Another interesting research avenue involves on-line data layout transformation subject to workload changes. This encapsulates the key intuitions of robust LSM tuning and hybrid LSM data layouts.
(4) *Performance predictability.* Reducing the duration and the variance of write-stalls when flushing is also a key goal.
(5) *Privacy-aware LSMs.* Similarly to deletes, LSM-trees realize updates through logical invalidation. Persistent and timely purging entries invalidated by updates is an open challenge, especially as it is hard to differentiate them from (new) inserts. Further, supporting timely and persistent deletes on secondary attributes is hard in LSM engines, particularly for point secondary deletes [15, 108].

3 TARGET AUDIENCE

This tutorial will offer an in-depth presentation of the log-structured merge (LSM) design paradigm along with the extensive recent research on how to optimize it. The target audience includes graduate students that aspire to do research on the topic of LSM-based systems (NoSQL, relational, spatial, or others), database researchers that want to quickly dive in the details of the LSM architecture, and practitioners that want to have a detailed summary of the state-of-the-art results on LSM. The audience will also be exposed to current challenges and open questions on LSM systems.

4 RELATED WORK

Prior tutorials on storage engine designs primarily focussed on expressing the key design principles and optimization techniques adopted in modern NoSQL and key-value systems [13, 56, 57, 94]. Athanassoulis and Idreos presented a tutorial that focussed on identifying and defining the intrinsic three-way tradeoff constructed by the **R**ead cost, the **U**pdate cost, and the **M**emory footprint [13]. Every state-of-the-art NoSQL, NewSQL, relational, and non-relational data system is bound by the RUM tradeoff, which in turn, helps in construction of the design space of data structures for LSM engines. Özcan *et al.* explored this tradeoff from the perspective of the fundamental architectural properties of modern hybrid transactional/analytical processing (HTAP) systems [94]. Idreos and Kraska presented a tutorial outlining the design trends and recent research advancements in auto-tuning and self-designing data systems from the perspective of data structures, algorithms, and query optimization [57]. Another recent tutorial by Idreos and Callaghan discusses the core design principles and components of modern key-value storage engines [56]. In contrast, in this tutorial, we specifically focus on LSM-trees, one of the most commonly used data structures in modern data stores, and dissect their principles, design goals, and optimization techniques. To the best of our knowledge, this is the first tutorial that focusses on deconstructing the log-structured merge paradigm and analyzing the LSM design space to provide useful insights for researchers and practitioners.

5 PRESENTERS

Subhadeep Sarkar is a post-doctoral associate at Boston University advised by Manos Athanassoulis. His research focuses on improving the performance of modern data systems by reasoning about the read-write tradeoff of the underlying data structures and algorithms. In a broader scope, his research interests span data systems, storage layouts, access methods, and their intersection with data privacy. Before joining Boston University, Subhadeep was a postdoctoral researcher at INRIA, Rennes (France). He completed his PhD from Indian Institute of Technology (IIT) Kharagpur. Subhadeep was selected as one of the young scientists to attend the Heidelberg Laureate Forum in 2016. In 2015, he received the second best PhD thesis award at the IDRBT Doctoral Colloquium.

Manos Athanassoulis is an assistant professor of Computer Science at Boston University where he leads the Data-intensive Systems and Computing laboratory, and co-founded the BU Massive Data Algorithms and Systems Group. His research focuses on building data systems that efficiently exploit modern hardware (computing units, storage, and memories), are deployed in the cloud, and can adapt to the workload both at setup time and, dynamically, at runtime. In addition, Manos works on understanding the tradeoffs of new computing and storage hardware focusing on the impact of reprogrammable hardware, and novel storage and non-volatile memory devices, and how to exploit them in data management pipelines. Before joining Boston University, Manos was a postdoctoral researcher at Harvard School of Engineering and Applied Sciences, obtained his PhD from EPFL, Switzerland, and spent one summer at IBM Research, Watson. Manos' research has been recognized by awards like "Best of SIGMOD" in 2016, "Best of VLDB" in 2010 and 2017, and "Most Reproducible Paper" at SIGMOD in 2017. He is also the recipient of an NSF CRII Award, a Facebook Faculty Research Award, a Cisco Research Award, a RedHat Research Award, and a Swiss NSF Postdoc Fellowship.

ACKNOWLEDGEMENT

This work was partially funded by NSF under Grant No. IIS-1850202 and a Facebook Faculty Research Award. We are thankful to Dimitris Staratzis, Zichen Zhu, and Tarikul Islam Papon for their contributions in different phases of this project.

REFERENCES

[1] 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)*, pages L119/1 – L119/88, 2016.

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

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

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

[5] H. Abu-Libdeh, D. Altınbüken, A. Beutel, E. H. Chi, L. Doshi, T. Kraska, Xiaozhou, Li, A. Ly, and C. Olston. Learned Indexes for a Google-scale Disk-based Database. In *Proceedings of the Workshop on ML for Systems at NeurIPS*, 2020.

[6] 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.

[7] 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.

[8] Amazon. Cloud Storage. *https://aws.amazon.com/what-is-cloud-storage/*.

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

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

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

[12] M. Athanassoulis, S. Chen, A. Ailamaki, P. B. Gibbons, and R. Stoica. MaSM: Efficient Online Updates in Data Warehouses. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 865–876, 2011.

[13] 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.

[14] 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.

[15] M. Athanassoulis, S. Sarkar, T. I. Papon, Z. Zhu, and D. Staratzis. Building Deletion-Compliant Data Systems. In *IEEE Data Engineering Bulletin*, 2022.

[16] 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.

[17] 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.

[18] A. Breslow and N. Jayasena. Morton Filters: Faster, Space-Efficient Cuckoo Filters via Biasing, Compression, and Decoupled Logical Sparsity. *Proceedings of the VLDB Endowment*, 11(9):1041–1055, 2018.

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

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

[21] M. Callaghan. Name that compaction algorithm. *http://smalldatum.blogspot.com/2018/08/name-that-compaction-algorithm.html*, 2018.

[22] M. Callaghan. Summarizing the different implementations of tiered compaction. *http://smalldatum.blogspot.com/2021/12/summarizing-different-implementations.html*, 2021.

[23] 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.

[24] B. Chandramouli, G. Prasaad, D. Kossmann, J. J. Levandoski, J. Hunter, and M. Barnett. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 275–290, 2018.

[25] 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.

[26] S. Chatterjee, M. Jagadeesan, W. Qin, and S. Idreos. Cosine: A Cloud-Cost Optimized Self-Designing Key-Value Storage Engine. In *In Proceedings of the Very Large Databases Endowment*, 2022.

[27] H. Chen, L. Liao, H. Jin, and J. Wu. The Dynamic Cuckoo Filter. In *Proceedings of the IEEE International Conference on Network Protocols (ICNP)*, pages 1–10, 2017.

[28] Y. Chen, Y. Lu, F. Yang, Q. Wang, Y. Wang, and J. Shu. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 1077–1091, 2020.

[29] B. Dageville. Snowflake Data Cloud. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2021.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] N. Dayan and M. Twitto. Chucky: A Succinct Cuckoo Filter for LSM-Tree. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 365–378, 2021.

[36] 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.

[37] P. C. Dillinger and S. Walzer. Ribbon filter: practically smaller than Bloom and Xor. *CoRR*, 2103.02515, 2021.

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

[39] 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.

[40] S. Dong, A. Kryczka, Y. Jin, and M. Stumm. Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 33–49, 2021.

[41] S. Dong, A. Kryczka, Y. Jin, and M. Stumm. RocksDB: Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications. *ACM Trans. Storage*, 17(4):26:1—-26:32, 2021.

[42] Facebook. MyRocks. *http://myrocks.io/*.

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

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

[45] B. Fan, D. G. Andersen, M. Kaminsky, and M. Mitzenmacher. Cuckoo Filter: Practically Better Than Bloom. In *Proceedings of the ACM International on Conference on emerging Networking Experiments and Technologies (CoNEXT)*, pages 75–88, 2014.

[46] D. Feinberg, M. Adrian, and A. Ronthal. The Future of Database Management Systems Is Cloud. *https://www.gartner.com/document/3941821*, 2019.

[47] P. Ferragina and G. Vinciguerra. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment*, 13(8):1162–1175, 2020.

[48] 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.

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

[50] B. Hayes. Cloud computing. *Communications of the ACM*, 51(7):9–11, 2008.

[51] P. Helland. Immutability Changes Everything. *Communications of the ACM*, 59(1):64–70, 2016.

[52] R. Herbster, S. DellaTorre, P. Druschel, and B. Bhattacharjee. Privacy Capsules: Preventing Information Leaks by Mobile Apps. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys 2016, Singapore, June 26-30, 2016*, pages 399–411, 2016.

[53] 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.

[54] H. Huang and S. Ghandeharizadeh. Nova-LSM: A Distributed, Component-based LSM-tree Key-value Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 749–763, 2021.

[55] A. Huynh, H. A. Chaudhari, E. Terzi, and M. Athanassoulis. Endure: A Robust Tuning Paradigm for LSM Trees Under Workload Uncertainty. *CoRR*, 2110.13801, 2021.

[56] 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.

[57] 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.

[58] S. Idreos, K. Zoumpatianos, M. Athanassoulis, N. Dayan, B. Hentschel, M. S. Kester, D. Guo, L. M. Maas, W. Qin, A. Wasay, and Y. Sun. The Periodic Table of Data Structures. *IEEE Data Engineering Bulletin*, 41(3):64–75, 2018.

[59] S. Idreos, K. Zoumpatianos, B. Hentschel, M. S. Kester, and D. Guo. The Data Calculator: Data Structure Design and Cost Synthesis from First Principles and Learned Cost Models. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 535–550, 2018.

[60] Influxdata. In-memory indexing and the Time-Structured Merge Tree (TSM). *https://docs.influxdata.com/influxdb/v1.8/concepts/storage_engine/*, 2021.

[61] S. Kannan, N. Bhat, A. Gavrilovska, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In H. S. Gunawi and B. Reed, editors, *2018 {USENIX} Annual Technical Conference, {USENIX} {ATC} 2018, Boston, MA, USA, July 11-13, 2018*, pages 993–1005. {USENIX} Association, 2018.

[62] K. Keahey. Chameleon: Experimental Platform for Cloud Computing Research. *https://www.chameleoncloud.org/media/cms_page_media/17/GENI-lex.pdf*, 2018.

[63] M. L. Kersten. Big Data Space Fungus. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR), Gong show talk*, 2015.

[64] J. Kim, S. Lee, and J. S. Vetter. PapyrusKV: a high-performance parallel key-value store for distributed NVM architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017, Denver, CO, USA, November 12 - 17, 2017*, pages 57:1—-57:14, 2017.

[65] Y.-S. Kim, T. Kim, M. J. Carey, and C. Li. A Comparative Study of Log-Structured Merge-Tree-Based Spatial Indexes for Big Data. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 147–150, 2017.

[66] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann. RadixSpline: a single-pass learned index. In *Proceedings of the International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM@SIGMOD)*, pages 5:1—-5:5, 2020.

[67] H. Kondylakis, N. Dayan, K. Zoumpatianos, and T. Palpanas. Coconut Palm: Static and Streaming Data Series Exploration Now in your Palm. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2019.

[68] H. Kopetz. Internet of Things. In *Real-Time Systems*, pages 307–323. Springer, 2011.

[69] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The Case for Learned Index Structures. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 489–504, 2018.

[70] A. Kryczka. Thread Pool. *https://github.com/facebook/rocksdb/wiki/Thread-Pool*, 2017.

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

[72] B. C. Kuszmaul. A Comparison of Fractal Trees to Log-Structured Merge (LSM) Trees. *Tokutek White Paper*, 2014.

[73] A. Lamb, M. Fuller, and R. Varadarajan. The Vertica Analytic Database: C-Store 7 Years Later. *Proceedings of the VLDB Endowment*, 5(12):1790–1801, 2012.

[74] B. Lepers, O. Balmau, K. Gupta, and W. Zwaenepoel. KVell: the design and implementation of a fast persistent key-value store. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 447–461, 2019.

[75] B. Lepers, O. Balmau, K. Gupta, and W. Zwaenepoel. Kvell+: Snapshot Isolation without Snapshots. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 425–441, 2020.

[76] Y. Li, C. Tian, F. Guo, C. Li, and Y. Xu. ElasticBF: Elastic Bloom Filter with Hotness Awareness for Boosting Read Performance in Large Key-Value Stores. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 739–752, 2019.

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

[78] 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.

[79] C. Luo. Breaking Down Memory Walls in LSM-based Storage Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 2817–2819, 2020.

[80] C. Luo and M. J. Carey. Efficient Data Ingestion and Query Processing for LSM-Based Storage Systems. *Proceedings of the VLDB Endowment*, 12(5):531–543, 2019.

[81] C. Luo and M. J. Carey. On Performance Stability in LSM-based Storage Systems. *Proceedings of the VLDB Endowment*, 13(4):449–462, 2019.

[82] C. Luo and M. J. Carey. Breaking Down Memory Walls: Adaptive Memory Management in LSM-based Storage Systems. *Proceedings of the VLDB Endowment*, 14(3):241–254, 2020.

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

[84] 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.

[85] A. Madan and A. Kryczka. DeleteRange: A New Native RocksDB Operation. *https://rocksdb.org/blog/2018/11/21/delete-range.html*, 2018.

[86] Q. Mao, M. A. Qader, and V. Hristidis. Comprehensive Comparison of LSM Architectures for Spatial Data. In *Proceedings of the IEEE International Conference on Big Data (BigData)*, pages 455–460, 2020.

[87] Y. Matsunobu, S. Dong, and H. Lee. MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph. *Proceedings of the VLDB Endowment*, 13(12):3217–3230, 2020.

[88] 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.

[89] G. Moerkotte. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 476–487, 1998.

[90] MongoDB. Online reference. *http://www.mongodb.com/*.

[91] P. Muth, P. E. O'Neil, A. Pick, and G. Weikum. The LHAM Log-structured History Data Access Method. *The VLDB Journal*, 8(3-4):199–221, 2000.

[92] M. A. Olson, K. Bostic, and M. I. Seltzer. Berkeley DB. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 183–191, 1999.

[93] 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.

[94] F. Özcan, Y. Tian, and P. Tözün. Hybrid Transactional/Analytical Processing: A Survey. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1771–1775, 2017.

[95] F. Pan, Y. Yue, and J. Xiong. dCompaction: Delayed Compaction for the LSM-Tree. *Int. J. Parallel Program.*, 45(6):1310–1325, 2017.

[96] H. Park and K. Shim. FAST: Flash-aware external sorting for mobile database systems. *Journal of Systems and Software*, 82(8):1298–1312, 2009.

[97] M. A. Qader, S. Cheng, and V. Hristidis. A Comparative Study of Secondary Indexing Techniques in LSM-based NoSQL Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 551–566, 2018.

[98] 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.

[99] 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.

[100] RocksDB. Low Priority Write. *https://github.com/facebook/rocksdb/wiki/Low-Priority-Write*, 2019.

[101] RocksDB. Single Delete. *https://github.com/facebook/rocksdb/wiki/Single-Delete*, 2019.

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

[103] RocksDB. Prefix Bloom Filter. *https://github.com/facebook/rocksdb/wiki/Prefix-Seek#configure-prefix-bloom-filter*, 2020.

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

[105] RocksDB. Block Cache. *https://github.com/facebook/rocksdb/wiki/Block-Cache*, 2021.

[106] RocksDB. Merge Operator. *https://github.com/facebook/rocksdb/wiki/Merge-Operator*, 2021.

[107] RocksDB. RocksDB Tuning Guide. *https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide*, 2021.

[108] S. Sarkar and M. Athanassoulis. Query Language Support for Timely Data Deletion. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2022.

[109] 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.

[110] S. Sarkar, S. Chatterjee, and S. Misra. Assessment of the Suitability of Fog Computing in the Context of Internet of Things. *IEEE Transactions on Cloud Computing (TCC)*, 6(1):46–59, 2018.

[111] S. Sarkar, K. Chen, Z. Zhu, and M. Athanassoulis. Compactionary: A Dictionary for LSM Compactions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2022.

[112] 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.

[113] S. Sarkar, D. Staratzis, Z. Zhu, and M. Athanassoulis. Constructing and Analyzing the LSM Compaction Design Space. *Proceedings of the VLDB Endowment*, 14(11):2216–2229, 2021.

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

[115] 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.

[116] M. I. Seltzer. Berkeley DB: A Retrospective. *IEEE Data Engineering Bulletin*, 30(3):21–28, 2007.

[117] W. Tan, S. Tata, Y. R. Tang, and L. L. Fong. Diff-Index: Differentiated Index in Distributed Log-Structured Data Stores. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 700–711, 2014.

[118] Y. R. Tang, A. Iyengar, W. Tan, L. L. Fong, L. Liu, and B. Palanisamy. Deferred Lightweight Indexing for Log-Structured Key-Value Stores. In *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2015, Shenzhen, China, May 4-7, 2015*, pages 11–20, 2015.

[119] 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.

[120] R. Thonangi, S. Babu, and J. Yang. A Practical Concurrent Index for Solid-State Drives. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM)*, pages 1332–1341, 2012.

[121] R. Thonangi and J. Yang. On Log-Structured Merge for Solid-State Drives. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 683–694, 2017.

[122] T. Vinçon, S. Hardock, C. Riegger, J. Oppermann, A. Koch, and I. Petrov. NoFTL-KV: TacklingWrite-Amplification on KV-Stores with Native Storage Management. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 457–460, 2018.

[123] A.-V. Vo, N. Konda, N. Chauhan, H. Aljumaily, and D. F. Laefer. Lessons Learned with Laser Scanning Point Cloud Management in Hadoop HBase. In *Proceedings of EG-ICE*, 2019.

[124] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong. An Efficient Design and Implementation of LSM-Tree based Key-Value Store on Open-Channel SSD. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 16:1–16:14, 2014.

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

[126] F. Wu. Improving Point-Lookup Using Data Block Hash Index. *https://rocksdb.org/blog/2018/08/23/data-block-hash-index.html*, 2018.

[127] X. Wu, Y. Xu, Z. Shao, and S. Jiang. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 71–82, 2015.

[128] 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.

[129] T. Yao, J. Wan, P. Huang, X. He, Q. Gui, F. Wu, and C. Xie. A Light-weight Compaction Tree to Reduce I/O Amplification toward Efficient Key-Value Stores. In *Proceedings of the IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2017.

[130] T. Yao, J. Wan, P. Huang, X. He, F. Wu, and C. Xie. Building Efficient Key-Value Stores via a Lightweight Compaction Tree. *ACM Transactions on Storage (TOS)*, 13(4):29:1–29:28, 2017.

[131] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 323–336, 2018.

[132] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo. Succinct Range Filters. *ACM Transactions on Database Systems (TODS)*, 45(2):5:1—-5:31, 2020.

[133] W. Zhang, Y. Xu, Y. Li, and D. Li. Improving Write Performance of LSMT-Based Key-Value Store. In *22nd IEEE International Conference on Parallel and Distributed Systems, ICPADS 2016, Wuhan, China, December 13-16, 2016*, pages 553–560, 2016.

[134] W. Zhang, X. Zhao, S. Jiang, and H. Jiang. ChameleonDB: a key-value store for optane persistent memory. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, pages 194–209, 2021.

[135] Z. Zhang, Y. Yue, B. He, J. Xiong, M. Chen, L. Zhang, and N. Sun. Pipelined Compaction for the LSM-Tree. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*, pages 777–786, 2014.

[136] Y. Zhu, Z. Zhang, P. Cai, W. Qian, and A. Zhou. An Efficient Bulk Loading Approach of Secondary Index in Distributed Log-Structured Data Stores. In *Database Systems for Advanced Applications - 22nd International Conference, DASFAA 2017, Suzhou, China, March 27-30, 2017, Proceedings, Part I*, volume 10177 of *Lecture Notes in Computer Science*, pages 87–102, 2017.

[137] Z. Zhu, J. H. Mun, A. Raman, and M. Athanassoulis. Reducing Bloom Filter CPU Overhead in LSM-Trees on Modern Storage Devices. *http://disc-projects.bu.edu/documents/DiSC-TR-Reducing-BF-Overhead-in-LSM.pdf*, 2021.