The LSM Design Space and its Read Optimizations

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

Niv Dayan
University of Toronto
Toronto, ON, Canada
nivdayan@cs.toronto.edu

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

2023 IEEE 39th International Conference on Data Engineering (ICDE) | 979-8-3503-2227-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICDE55515.2023.00273

***Abstract*—Log-structured merge (LSM) trees have emerged as one of the most commonly used storage-based data structures in modern data systems as they offer high throughput for writes and good utilization of storage space. However, LSM-trees were not originally designed to facilitate efficient reads. Thus, state-of-the-art LSM engines employ numerous optimization techniques to make reads efficient. The goal of this tutorial is to present the fundamental principles of the LSM paradigm along with the various optimization techniques and hybrid designs adopted by LSM engines to accelerate reads.**

Toward this, we first discuss the basic LSM operations and their access patterns. We then discuss techniques and designs that optimize point and range lookups in LSM-trees: (i) index and (ii) filter data structures, (iii) caching, and (iv) read-friendly data layouts. Next, we present the performance tradeoff between writes and reads, outlining the rich design space of the LSM paradigm and how one can navigate it to improve query performance. We conclude by discussing practical problems and open research challenges. This will be a 1.5-hour tutorial.

***Index Terms*—LSM-trees; Key-value stores; Storage engine**

I. Introduction

LSM-Trees are Everywhere. The log-structured merge (LSM) paradigm has emerged as one of the most popular storage paradigms for modern data stores. This is because LSM-trees (i) offer high throughput for writes by employing out-of-place ingestion to the database [65], while (ii) their immutable file structure allows for good utilization of storage space [27]. Thus, LSM-trees are widely used in state-of-the-art relational and NoSQL data stores, including LevelDB [32] and BigTable [14] at Google, RocksDB [29] and MyRocks [61] at Facebook, X-Engine [33] at Alibaba, DynamoDB [25] at Amazon, AsterixDB [2], Cassandra [5], HBase [4], and Accumulo [3] at Apache, ScyllaDB [77], Speedb [79], Pliops [22], in influential but now discontinued systems like Voldemort [52] and bLSM [78], as well as, in other spatial [45, 60, 84] and time-series data systems [41, 47].

The LSM Paradigm and its Tradeoffs. The founding idea of LSM trees is to *buffer ingestions* in memory and flush them to storage using *large sequential writes*, facilitating fast ingestion by avoiding random writes. However, the eagerness of compaction creates a stringent cost contention between the costs of reads and writes. While compacting more eagerly means that queries need to search fewer components, it increases the amortized cost of writes and thus counter-balances the benefits of LSM-tree in the first place [6, 7].

Attempting to reduce the lookup cost comes at the cost of increased memory footprint and CPU-intensive computations. Thus, a lot of work over the past decade focused on how to facilitate faster reads. State-of-the-art LSM engines use several **auxiliary in-memory data structures** such as filters, indexes, and block-based caches, implement **workload-aware block-based caching** and **prefetching** techniques, and adopt **read-friendly designs** and **tunings** to improve lookup performance.

A Rich Design Space. A closer look into the internals of production LSM engines reveals numerous design decisions that determine the performance of a storage engine, such as the storage data layout, the implementation of the auxiliary in-memory components that facilitate competitive reads, and how to allocate memory between the buffer and other in-memory components [18, 75, 76]. Together, these design choices constitute the vast design space of LSM engines.

Goal of the Tutorial. In this tutorial, we discuss the internals and design decisions that drive state-of-the-art LSM engines, highlighting the optimization techniques adopted to make reads better. We reveal the rich design space of the LSM paradigm and provide insights on how to build, tune, and optimize LSM-based systems and navigate their design space.

Tutorial Overview

The 1.5-hour tutorial, broken down into 3 modules.

Module I: LSM Basics. (*25 mins*) This module discusses the LSM background and and its core read/write tradeoffs.

(i) *The LSM structure and its key operating principles* (5 mins). Outlines the in-memory and storage-based components of LSM-based storage engines, and discusses the fundamental operating principles of the log-structured merge paradigm [18, 27, 58, 65, 76].

(ii) *Access patterns of the fundamental LSM operations* (10 mins). Presents the workflows for the internal operations (*flush* and *compaction*) [27, 58, 76] and external operations (*put*, *get*, *scan*, and *delete*) [19, 20, 58, 65, 73] of an LSM-based data store.

(iii) *The read vs. write tradeoff in LSM-trees* (10 mins). Outlines the key optimization techniques used in state-of-the-art LSM engines to improve query performance, focussing particularly on the LSM compaction design space [15, 20, 21, 23, 24, 36, 69, 76, 87].

Module II: Optimizing Reads. (*40 mins*) The second module outlines various optimization techniques that make point and range queries efficient.

2375-026X/23/$31.00 ©2023 IEEE
DOI 10.1109/ICDE55515.2023.00273

(i) *Techniques to improve point query performance* (15 mins). Outlines the various filter data structures (and index structures) used in LSM engines to improve point lookup performance [18, 19, 23, 26, 50].

(ii) *Techniques to improve range query performance* (10 mins). Outlines the different filter and index data structures proposed in the literature to facilitate efficient range queries in LSM engines [59, 91–93].

(iii) *Prefetching and caching techniques; learned indexing* (5 mins). Presents new techniques to employ caching and learned approaches to speed up queries [1, 13, 14, 17, 31, 44, 46, 48, 55, 60, 80, 81, 94].

(iii) *Impact of LSM shapes on read performance* (10 mins). Outlines the impact of lazy and greedy merging policies on read performance, and provides intuitions on the influence of the common LSM shapes on reads [18, 23, 76].

Module III: LSM Tuning. (*25 mins*) This module presents the research on tunability and navigability of the LSM design space to ensure optimal design choices for diverse workloads.

(i) *Navigating the LSM design space and tradeoff curve* (15 mins). Outlines techniques to navigate the read-write tradeoff curve by constructing hybrid LSM shapes based on the workload [19, 21, 37–39, 54, 57].

(ii) *Workload-aware tuning and robust tuning* (5 mins). Discusses workload and performance-driven tuning techniques LSM engines, including discussions on robust LSM tunings [35, 56].

(iii) *Opportunities and open research challenges* (5 mins)

Output. The expected outcomes of this tutorial are:

- Understanding of the LSM paradigm, the LSM design space, and its key operating principles.
- Insights about the point and range query paths and the role of auxiliary in-memory data structures in optimizing reads.
- Understanding the performance tradeoff in LSM-trees.
- Appreciation for the rich design space of LSMs, along with intuitions about how to navigate this design space.
- Exposure to open research challenges on the LSM paradigm.

Target Audience. The target audience includes graduate students who aspire to try their hands on LSM-based systems, database researchers who want a primer on LSM, and practitioners who want to better tune their systems.

II. Tutorial Narrative

A. Module I: LSM Basics

The LSM Design. LSM-trees store data as key-value pairs, where a *key* refers to a unique object identifier, and the data associated with it is the *value*. Entries in an LSM-tree are typically stored and accessed based on the key. LSM-trees were conceptualized to have a hierarchical data layout with one component in memory and several components – a collection of *sorted runs* – on storage [65]. For an LSM-tree with L levels, the first level (Level 0) resides in memory, while the remaining levels (Levels 1 to $L-1$) reside in storage [20]. Data in the memory-resident component is moved to the storage-component iteratively during data layout re-organization.

1) Basic Operations: We now present the basic operations supported by LSM-based key-value stores.

Internal operations are triggered by the storage engine to re-structure the data layout: (i) *flushing* the memory buffer to storage, and (ii) *compacting* the storage-resident sorted runs.

Flush: LSM-trees accumulate incoming entries in a mutable in-memory buffer to amortize the storage access cost. When the memory buffer reaches capacity, the entries are sorted on the key, and *flushed* to the storage as an immutable file [58].

Compaction: When a storage-resident level reaches capacity, all or part of the data from that level is moved to the next level, sort-merging the entries that overlap within the key range. This process, termed *compaction*, limits the number of sorted runs in a tree, and in the process, garbage collects logically invalidated entries [27, 58].

External operations are triggered by the application in order to access or update data via *puts*, *gets*, and *scans*.

Put: LSM-trees ingest data in an out-of-place manner; thus, updates are treated similarly to inserts. Inserts, updates, and deletes are first *put* into the memory buffer, before being moved to storage in an opportunistic way.

Get: A *get* returns the most recent version of an entry. A get begins at the memory buffer and traverses the tree from the youngest sorted run to the oldest one. 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 the matching key.

Scan: A *scan* returns the latest version of all range of keys after scanning and merging the contents of all qualifying sorted runs. Typically, during range lookups, an iterator is assigned for each run, and the runs are scanned in parallel. For consistency reasons, a scan operates over a version (or snapshot) of the data. A version is the collection of files that were active and live at the time the scan began.

2) The Read vs. Write Tradeoff: LSM-trees exhibit an intrinsic tradeoff between reads and writes, the understanding of which is critical for performance tuning. Varying the **number of buffer components**, **the buffer size**, and **implementation of the buffer** allows one to navigate the read-update-memory tradeoff [7], and optimize performance based on the workload and the performance target [5, 28, 29, 29, 33, 77, 85].

To avoid merging the buffer components with storage-resident runs eagerly, a **tiered variant of the LSM** was introduced by Jagadish *et al.* [42]. The tiered design, with multiple sorted runs with overlapping key-ranges in a level, 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 [4, 5, 29, 76, 77]. Recent research has proposed a new set of **hybrid data layouts** where the shallower (smaller) level(s) have a tiered layout and the lower (larger) levels have a leveled layout [20, 36, 37, 87].

For better load balancing, some LSM engines **partition the key space** and store the partitions in separate trees [34, 56, 64, 67]. For tail latency sensitive applications, many LSM engines

have adopted a **partial compaction** strategy, where one (or few) file is compacted at a time [29, 32, 33, 75, 76]. For such systems, the design decision on **which file(s) to compact** affects ingestion performance [74, 76]. Recent research has also proposed to separate the storage of keys from values to improve search time at the expense of additional accesses for queries [12, 49, 53, 88].

B. Module II: Optimizing Reads

The write-optimized LSM design leads to suboptimal read performance. To ameliorate this, LSM engines employ in-memory auxiliary structures, like filters, indexes, and caches.

1) Basic Indexing and Block-based Caching: Without help from any auxiliary data structures, LSM-trees would perform several superfluous I/Os for every lookup. To avoid issuing a binary search in storage for each sorted run during a lookup, LSM-trees maintain **fence pointers** (a special form of Zonemaps [62]), which allow to access the relevant key range at each run with just one storage access [27]. Such light-weight data structures are typically pre-fetched to memory in an opportunistic way. LSM-trie eliminates the fence pointers by using the immutable files as hash tables, reducing the CPU overhead for index navigation [87]. Bourbon [17] uses **learned indexing** to improve the performance of fence pointers.

Another way of improving lookup performance is by using **block-level caching** [71]. Commercial LSM engines use a block cache (which, for example, is frequently configured to be 12GB in RocksDB [27]) that can be tuned to retain in memory the first few levels of a tree, the frequently accessed hot data blocks, and even the filter and index blocks. Since compactions involve a lot of data movement, it is rather frequent that the hot pages that are compacted are invalidated, forcing us to fetch the same data in memory [82]. To address this problem and retain hot pages in memory, Leaper [90] introduces an ML-aided predictive mechanism to identify pages from recently compacted files and **prefetch those in block cache** immediately after compaction.

2) Point Query Filters: In the worst-case, even with fence pointers, a point lookup may need to probe every sorted run in a tree [18, 27, 58], leading to superfluous I/Os. Thus, to bound the cost of point lookups, state-of-the-art LSM engines maintain **Bloom filters** in memory [29]. Bloom filters allow a lookup to skip probing a run altogether, if the filter-lookup returns negative [58]. Further, block-based Bloom filters [66] reduce CPU cache misses while partitioned filters allow for more granular in-memory caching [89]. Several new filter designs and LSM-specific filter optimization techniques have been proposed. ElasticBF [50] and Modular Bloom filters [63] address access skew by employing **multiple small filter units per Bloom filter**. Ribbon filter [26] introduces a better **tradeoff of index time vs. space utilization** at the expense of additional CPU work. On the other hand, sharing hash calculations [95] reduces the CPU cost per query. Cuckoo filters [30] are also used as a Bloom filter replacement in SlimDB [68] and Chucky [23]. Note that other approximate set membership data structures can be used as Bloom filter replacements [10, 11, 16]. A different approach is taken by PinK [40] which avoids using Bloom filters altogether by pinning the keys from the shallower LSM-levels in memory.

3) 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 the tree. Therefore, range filters are crucial to prevent unnecessary storage accesses while executing range queries on LSMs. Prefix filters **use fixed-length key-prefixes** to answer long range membership queries [70]. SuRF [91, 92] is a **succinct trie-based filter** that supports storing variable length prefixes of keys, thus, allowing fewer false positives for long range queries. Rosetta [59] 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. REMIX reduces the CPU costs for range scans by maintaining an index on the domain order of entries across multiple runs [93]. SNARF is an array-based range filter that uses a distribution-aware model and compressed bit array to efficiently address numeric range queries [83].

4) Advanced Indexing Techniques: In memory, the tight-loop search during lookups leads to multiple key comparisons triggering several cache misses and increasing CPU utilization [86]. Thus, many systems maintain an additional hash map for each page which allows a key to be found in constant time. Several approaches have also focused on optimizing reads on secondary (non-key) attributes through **secondary indexing** techniques [55, 60, 80, 81, 94].

In addition to classical indexes, recent work also focuses on using **learned index on LSM-trees** to reduce memory footprint and to speed up the in-memory computation associated with indexes (e.g., Bourbon [17]). RadixSpline [46] proposes a **learned hash-based index** that can be constructed in a single pass over the data with no support for inserts. Such indexes are beneficial for the LSM design (i) due to their *low training time* that does not affect the ingestion throughput and (i) because their read-only nature can *exploit the immutability of files* in LSMs. While other learned indexes, such as PGM [31] and RMI [48], support inserts, they can still be used as a read-only index on LSMs. However, they require multiple passes to learn the data distribution, which increases the construction time. In a recent study [1], Google reported the superiority of such indexes compared to fence pointers in production systems.

5) Optimizing Memory Allocation: The memory allocation between the buffer and the filters can be tuned to improve read performance. Usually, in production systems, the Bloom filters are assigned the same memory footprint across all levels of the LSM tree. Monkey [18, 19] proposes an **optimal memory allocation strategy** (i) across the Bloom filters within a tree and (ii) between filters and buffers to navigate the read-write tradeoff space. Chucky [23] discusses the same with fingerprints and variable hash bucket sizes for **succinct cuckoo filters** on LSMs. Luo *et al.* [54, 57] outlined how to optimally **allocate memory between the memory buffer and the block cache** to improve query performance. FloDB [9] introduces a two-level buffer to facilitate efficient in-memory search.

C. Module III: LSM Tuning

Commercial LSM engines expose hundreds of tuning knobs to the developers, and together, these variable components constitute the LSM design space. The space is vast, and it is important to quantify the impact of LSM designs and tunings to improve performance.

1) Navigating the LSM Design Space: Idreos *et al.* [38, 39] introduced a notion of breaking down data structures into *first-order primitives*, which has been seminal in exploration of the LSM design space. Based on this, the design continuum [37] outlines a **larger design space for LSMs** by exploring different design elements, in terms of (i) the storage data layout, (ii) the data access patterns, and (iii) main memory allocation. Sarkar *et al.* [74, 76] have introduced 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*, highlighting how compactions affect the performance of LSM engines in terms of ingestion, point and range lookups, and space and write amplification.

LSM-Bush [21] introduces a **continuum for data layouts**, where LSMs are configured to have an arbitrary number of sorted runs in each level based on worst-case cost modeling. Cosine [15] breaks away from worst-case cost modeling and introduces (i) *distribution-aware I/O models* and (ii) *learning-based concurrency models*, which allow for **accurate navigation of the LSM design space**.

2) Robust LSM Designs: The ever-evolving application requirements and wide-spread adoption of shared computing infrastructures (e.g., private or public clouds) have added a considerable amount of uncertainty between the expected and the observed workloads. Toward this, Huynh *et al.* [35] propose a **robust LSM tuning** approach to minimize the worst-case performance loss in a *neighborhood* of the expected workload. CruiseDB [51] proposes an adaptive admission mechanism to improve the LSM-shape, offering enhanced tail performance. Luo *et al.* [56] proposed a **throttling mechanism for compactions** to offer predictably stable performance, diminishing the performance instability due to overloading underlying storage. Silk+ prevents latency spikes in LSM stores running heterogeneous workloads [8]. DLC introduces a load-aware compaction policy to increase performance stability in LSM-based storage engines [43].

3) Open Challenges: In the final part of the tutorial, we present opportunities and open challenges in LSM research. Despite recent efforts, LSMs continue to suffer from high write amplification. Thus, **reducing write amplification** in LSMs remains an open challenge. Further, identifying the **optimal compaction policies** based on the workload and the LSM-tuning is an interesting research avenue. A first step toward this would involve extensive workload-aware modeling for each compaction primitive. Another interesting research direction involves **on-line storage data layout transformation** subject to workload changes. This encapsulates the key intuitions of robust LSM tuning and hybrid LSM data layouts. Reducing the duration and the variance of write-stalls when flushing is also a key challenge. Last but not least, LSM engines are frequently used in transactional settings, and a detailed analysis of the interplay of compaction with transactional semantics is still missing from the literature.

III. Related Work

Prior tutorials on storage engine designs primarily focused on expressing the design principles and optimizations adopted in modern key-value systems [6, 36, 37]. Athanassoulis and Idreos [6] presented a tutorial that focused on the three-way tradeoff constructed by the **R**ead cost, the **U**pdate cost, and the **M**emory footprint that binds every data system. Idreos and Kraska [37] presented a tutorial outlining the general design trends toward building auto-tuning and self-designing data systems. The tutorial by Idreos and Callaghan [36] discussed the core design principles and components of modern key-value storage engines. Sarkar and Athanassoulis [72] focus on the techniques adopted in modern LSM engines to improve ingestion performance and reduce write amplification. On the other hand, this tutorial focuses on *techniques to optimize read performance of LSM-trees*. We discuss various approaches adopted in state-of-the-art LSM engines to improve read performance, the read vs. write tradeoff, and insights about the LSM design space and tuning.

IV. Presenters

Subhadeep Sarkar is a post-doctoral associate at Boston University. His research focuses on building efficient data systems by navigating the design space of data structures and algorithms. Prior to this, he was a post-doctoral researcher at INRIA, Rennes and completed his PhD from Indian Institute of Technology 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.

Niv Dayan is an assistant professor at the University of Toronto. His work combines practical and theoretical approaches for designing storage engines and their underlying data structures. He was previously a research scientist at Pliops and a post-doctoral researcher at Harvard and Copenhagen University. He completed his PhD at the IT University of Copenhagen.

Manos Athanassoulis is an assistant professor at Boston University, with expertise on data management, novel hardware, and data systems design and tuning. Prior to BU, Manos was a postdoctoral researcher at Harvard, 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.

Acknowledgment. This work was funded by NSF grants IIS-1850202 & IIS-2144547, a Facebook Faculty Research Award, and a Meta gift.

REFERENCES

[1] 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.

[2] 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.

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

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

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

[6] 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.

[7] 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.

[8] 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.

[9] O. Balmau, R. Guerraoui, V. Trigonakis, and I. Zablotchi. FloDB: Unlocking Memory in Persistent Key-Value Stores. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 80–94, 2017.

[10] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. Don't Thrash: How to Cache Your Hash on Flash. *Proceedings of the VLDB Endowment*, 5(11):1627–1637, 2012.

[11] 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.

[12] H. H. W. Chan, Y. Li, P. P. C. Lee, and Y. Xu. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1007–1019, 2018.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] N. Dayan, Y. Rochman, I. Naiss, S. Dashevsky, N. Rabinovich, E. Bortnikov, I. Maly, O. Frishman, I. B. Zion, Avraham, M. Twitto, U. Beitler, E. Ginzburg, and M. Mokryn. The End of Moore's Law and the Rise of The Data Processor. *Proceedings of the VLDB Endowment*, 14(12):2932–2944, 2021.

[23] 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.

[24] N. Dayan, T. Weiss, S. Dashevsky, M. Pan, E. Bortnikov, and M. Twitto. Spooky: Granulating LSM-Tree Compactions Correctly. *Proceedings of the VLDB Endowment*, 15(11):3071–3084, 2022.

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

[27] 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.

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

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

[30] 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.

[31] 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.

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

[33] 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.

[34] 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.

[35] A. Huynh, H. A. Chaudhari, E. Terzi, and M. Athanassoulis. Endure: A Robust Tuning Paradigm for LSM Trees Under Workload Uncertainty. *Proceedings of the VLDB Endowment*, 15(8):1605–1618, 2022.

[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] 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.

[39] 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.

[40] J. Im, J. Bae, C. Chung, Arvind, and S. Lee. PinK: High-speed In-storage Key-value Store with Bounded Tails. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 173–187, 2020.

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

[42] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti. Incremental Organization for Data Recording and Warehousing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 16–25, 1997.

[43] P. Jin, J. Li, and H. Long. DLC: A New Compaction Scheme for LSM-tree with High Stability and Low Latency. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 547–557, 2021.

[44] 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)*, pages 57:1—-57:14, 2017.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] Y. Li, Z. Liu, P. P. C. Lee, J. Wu, Y. Xu, Y. Wu, L. Tang, Q. Liu, and Q. Cui. Differentiated Key-Value Storage Management for Balanced I/O Performance. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 673–687, 2021.

[50] 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.

[51] J. Liang and Y. Chai. CruiseDB: An LSM-Tree Key-Value Store with Both Better Tail Throughput and Tail Latency. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 1032–1043, 2021.

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

[53] 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.

[54] 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.

[55] 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.

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

[57] 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.

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

[59] 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.

[60] 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 (IEEE BigData)*, pages 455–460, 2020.

[61] 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.

[62] 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.

[63] J. H. Mun, Z. Zhu, A. Raman, and M. Athanassoulis. LSM-Tree Under (Memory) Pressure. In *Proceedings of the International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, 2022.

[64] 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.

[65] 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.

[66] F. Putze, P. Sanders, and J. Singler. Cache-, hash-, and space-efficient bloom filters. *ACM Journal of Experimental Algorithmics*, 14, 2009.

[67] 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.

[68] 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.

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

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

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

[72] S. Sarkar and M. Athanassoulis. Dissecting, Designing, and Optimizing LSM-based Data Stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 2489–2497, 2022.

[73] 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.

[74] 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*, pages 2429–2432, 2022.

[75] 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.

[76] 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.

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

[78] 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.

[79] Speedb. Online reference. *https://github.com/speedb-io/speedb*.

[80] 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.

[81] 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 *Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 11–20, 2015.

[82] 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 *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 68–79, 2017.

[83] K. Vaidya, T. Kraska, S. Chatterjee, E. R. Knorr, M. Mitzenmacher, and S. Idreos. SNARF: A Learning-Enhanced Range Filter. *Proceedings of the VLDB Endowment*, 15(8):1632–1644, 2022.

[84] 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.

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

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

[87] 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.

[88] G. Xanthakis, G. Saloustros, N. Batsaras, A. Papagiannis, and A. Bilas. Parallax: Hybrid Key-Value Placement in LSM-based Key-Value Stores. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pages 305–318, 2021.

[89] M. Yabandeh. Partitioned Index/Filters. *http://rocksdb.org/blog/2017/05/12/partitioned-index-filter.html*, 2017.

[90] 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.

[91] 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.

[92] 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.

[93] W. Zhong, C. Chen, X. Wu, and S. Jiang. REMIX: Efficient Range Query for LSM-trees. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 51–64, 2021.

[94] 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 *Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA)*, volume 10177 of *Lecture Notes in Computer Science*, pages 87–102, 2017.

[95] Z. Zhu, J. H. Mun, A. Raman, and M. Athanassoulis. Reducing Bloom Filter CPU Overhead in LSM-Trees on Modern Storage Devices. In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*, pages 1:1–1:10, 2021.