Bi-directional Log-Structured Merge Tree

Xin Zhang, Qizhong Mao, Ahmed Eldawy, Vagelis Hristidis, Yihan Sun
{xzhan261,qmao002,eldawy}@ucr.edu,{vagelis,yihans}@cs.ucr.edu
University of California, Riverside
Riverside, California, USA

ABSTRACT

The Log-Structured Merge (LSM) Tree has become a popular storage scheme for modern NoSQL and New SQL database systems. The LSM-tree scheme achieves high write throughput by first buffering writes in memory, then flushing them to the disk with sequential I/O. LSM-tree is an out-of-place structure, so the key range of a level in the tree can overlap with those of other levels. This negatively impacts range query performance, as multiple levels have to be scanned. Note that range queries are fundamental operators for other types of queries such as joins or spatiotemporal queries. To improve the read performance of LSM-trees, this paper proposes the *Bi-directional LSM-tree*, which differs from the classical LSM-tree in that hot records can move to higher levels to improve the overall LSM organization and benefit future range queries. The Bi-directional LSM-tree reuses the work performed during range queries to selectively generate a special type of components, called *sentinel components*. Our experiments show that the Bi-directional LSM-tree can save more than 10% of disk I/O compared to a standard Leveled LSM-tree.

CCS CONCEPTS

• **Information systems → Database management system engines**.

KEYWORDS

LSM-tree Database, Read Performance, Range Query

ACM Reference Format:
Xin Zhang, Qizhong Mao, Ahmed Eldawy, Vagelis Hristidis, Yihan Sun. 2022. Bi-directional Log-Structured Merge Tree. In *34th International Conference on Scientific and Statistical Database Management (SSDBM 2022), July 6–8, 2022, Copenhagen, Denmark.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3538712.3538730

1 INTRODUCTION

Due to the growing popularity of write-intensive workloads, the Log-Structured Merge (LSM) Tree [10] is widely used in modern database systems, such as AsterixDB [1], LevelDB [5], and RocksDB [4]. To achieve high write throughput, the LSM-tree buffers writes into components in memory and *flush*es the memory components to disk via sequential I/O when it is full. An LSM-tree may need to read multiple disk components to answer a query, as it generally has higher read amplification than traditional B^+-trees [6]. To reduce the read amplification, LSM-tree performs *compaction*s (a.k.a. merges) regularly to organize the records into a sorted order.

One of the most widely used LSM-tree architectures is *Leveled*, which was first implemented in LevelDB, and later adopted in RocksDB [4] A Leveled LSM-tree groups disk components into levels, and disk components in each level have disjoint key ranges. The level size is controlled by a hyper-parameter *size ratio* (or fan factor) T, such that the maximum size of level $i+1$ ($i \geq 1$) is T times larger than that of level i. When level i is full, a disk component in this level is selected to merge with all disk components in level $i+1$ that overlap with the selected component. Via compactions, records (usually cold) can only be moved from upper (and smaller) levels to lower (and larger) levels in a single-directional flow. This design can effectively reduce the read amplification of point queries.

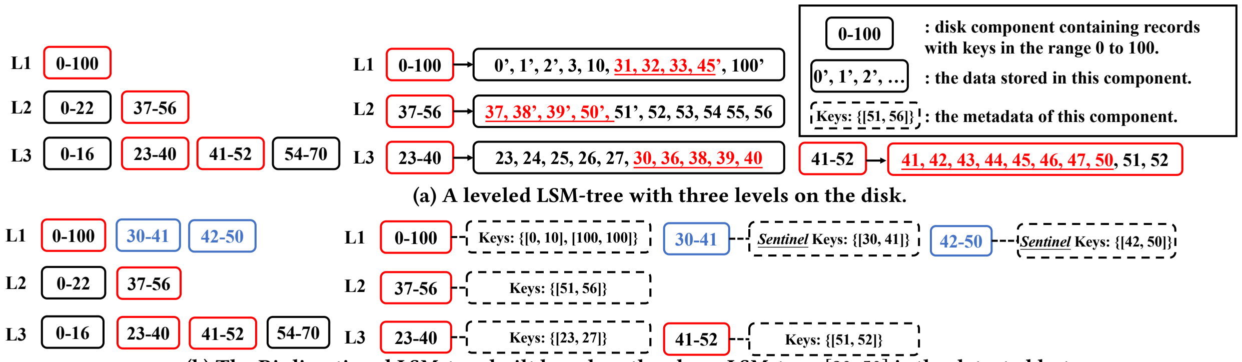
Although leveled LSM-tree has good point query performance, its range query performance is sub-optimal [13], which impacts several types of queries that use range queries (scans) as a basic operator, such as joins or spatial queries. The matching records may be distributed across many components in almost all levels, as illustrated in Figure 1a. Given a range query $Q = [32, 44]$, the red components are *operational components* which may contain matching records in the range from 32 to 44. The system must read the red components from every level to answer the range query Q. Besides, more random I/O also leads to lower cache hit rate, which further reduces range query throughput. Compactions can alleviate this problem to some extent, but they are expensive.

To improve the performance of range queries, we propose the *Bi-directional LSM-tree*, which selectively generates components on higher levels (referred to as *sentinel components*) to optimize the execution of future hot range queries. This effectively means that records can also move to higher levels, in contrast to the traditional LSM scheme. To minimize the cost for creating sentinel components, we piggyback off the work already performed during range queries, that is, the scan of disk components to obtain a stream of sorted and distinct records (thus handling anti-matters/tombstones [1]). These sentinel components, which are created only for hot ranges, are then used to answer future queries by providing more sequential disk access. A lightweight in-memory data structure is used to maintain the hotness of each range. In Figure 1b, the blue components are sentinel components built based on the hot range [30, 50]. Sentinel components can directly answer any future range queries in [30, 50] without checking components in the lower levels. Compared with the LSM-tree in Figure 1a, Bi-directional LSM-tree saves disk I/O and provides more sequential disk access.

In summary, this work makes the following contributions:

- We proposed the Bi-directional LSM-tree, which utilizes sentinel components created from range queries to improve range query performance (Section 3).

SSDBM 2022, July 6–8, 2022, Copenhagen, Denmark
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9667-7/22/07.
https://doi.org/10.1145/3538712.3538730


L1 0-100
L2 0-22 37-56
L3 0-16 23-40 41-52 54-70
L1 0-100 → 0', 1', 2', 3, 10, 31, 32, 33, 45', 100'
L2 37-56 → 37, 38', 39', 50', 51', 52, 53, 54 55, 56
L3 23-40 → 23, 24, 25, 26, 27, 30, 36, 38, 39, 40
41-52 → 41, 42, 43, 44, 45, 46, 47, 50, 51, 52
0-100 : disk component containing records with keys in the range 0 to 100.
0', 1', 2', ... : the data stored in this component.
Keys: {[51, 56]} : the metadata of this component.
L1 0-100 30-41 42-50
L2 0-22 37-56
L3 0-16 23-40 41-52 54-70
L1 0-100 Keys: {[0, 10], [100, 100]} 30-41 Sentinel Keys: {[30, 41]} 42-50 Sentinel Keys: {[42, 50]}
L2 37-56 Keys: {[51, 56]}
L3 23-40 Keys: {[23, 27]} 41-52 Keys: {[51, 52]}

(a) A leveled LSM-tree with three levels on the disk.

(b) The Bi-directional LSM-tree built based on the above LSM-tree. [30, 50] **is the detected hot range.**

Figure 1: Each cell is a disk component. Given a range query, the red cells are regular disk components (operational components) that contain query results. The blue components are sentinel components built eagerly based on the hot range.

- We implemented Bi-directional LSM-tree on a simulator, and compared it with a standard Leveled LSM-tree. We showed that our proposed method achieves lower total disk I/O than the standard Leveled LSM-tree (Section 4).

2 RELATED WORK

The optimization of LSM-tree has become a hot topic in the research community [7]. Recent work has focused on reducing the write amplification [2], merge operations [11, 12], compaction policy [8, 9], and query performance [3, 12, 13].

TRIAD [2] keeps hot data in memory for longer and only flushes cold data to disk. Once the cold records are compacted to lower levels in TRIAD, they cannot become hot anymore. Like TRIAD, our method also optimizes LSM-tree by data hotness, but we allow cold records to move back to upper levels if they become hot again. dCompaction [11] simulates real compactions by virtual compactions, which only modifies the metadata to avoid heavy disk I/O during real compactions. Our method also updates metadata instead of rewriting original disk components to reduce the overheads. LSbM-tree [12] introduced an on-disk buffer to reduce cache and improve query performance. It might not work well on range queries and introduces extra overhead when querying uncached cold records. Our method can benefit both range queries and point queries. Monkey [3] proposed a co-tuning compaction policy to optimize the balance between the cost of updates and reads with a given memory budget. REMIX [13] built extra indexes to improve the range query performance but introduced high maintenance cost when write happens. The system needs to rebuild the index due to the updates. Building sentinel components can benefit future flush and merge operations, but LSbM-tree, Monkey, REMIX cannot achieve these benefits.

3 BI-DIRECTIONAL LSM-TREE

To improve the read performance of LSM-tree, we propose Bi-directional LSM-tree (denoted as BiLSM in the rest), a variant of Leveled LSM-tree. BiLSM uses *sentinel components* to store hot record ranges. In this section, we first discuss the properties of sentinel components, and how to build sentinel components. Then, we present an architectural overview and basic operations of BiLSM.

3.1 Sentinel Components

Let's denote *regular components* as the disk components created by flushes or compactions. *Sentinel components* are special regular components created from range queries with a sentinel flag. The system creates sentinel components based on a range K when it identifies K as a hot range. Then, the system marks this range K (changing its metadata) as *disabled* in the regular components whose key ranges overlap with K. In Figure 1b, assume each regular component can store 10 records, we created the two blue sentinel components based on the hot range [30, 50] and updated "keys" in the metadata of the four red regular components.

Compared to regular components, sentinel components have two useful properties: 1) *they have the newest version of the records in its key range with respect to any component in lower levels*, and 2) *the key range of a sentinel component does not overlap with any other sentinel component*. These two properties indicate that any future range queries that are fully contained in the sentinel components[1] only need to visit the sentinel components without going deeper into the BiLSM. By gathering hot ranges in sentinel components, BiLSM can access fewer components than a standard LSM-tree to answer a query, greatly reducing disk I/O. The system can also utilize more sequential I/O by accessing more "compact" sentinel components on the top level of the BiLSM, instead of resulting in random I/O accessing all levels in the tree.

Creating Sentinel Components. A naive approach is to create sentinel components for every range query. However, this leads to two problems: 1) too many small sentinel components if there are many short range queries, leading to high overhead to manage them and scan among them; 2) too costly to create sentinel components for cold records which will not be queried frequently. Hence, we need to carefully identify ranges that (mostly) contain hot data and create sentinel components for them.

To identify the hot range from an LSM-tree, we divide the whole key space into non-overlapping *Bucket Ranges* and propose a cost model to track the access frequency of each Bucket Range. We first equally split the whole key space into a set of disjoint contiguous Bucket Ranges. For each Bucket Range,we maintain a reader counter and a writer counter to keep track of the hotness information in this Bucket Range. The hotness information is the score of the read counter minus the score of the writer counter, which is the *the estimated benefit*. The write counter stores the number of updates in this bucket range in a certain time window. The read counter is

[1]In fact, this is also true for any future lookup query.

the accumulated predicted number of disk I/O in this Bucket Range saved by using BiLSM.

Assume size ratio $T(T > 1.0)$ and block size B bytes. In a Leveled LSM-tree of $L(L \geq 1)$ levels, given a range query, assume the system reads s bytes from level 1 and sT^{i-1} bytes from level $i(i \geq 1)$. To answer the range query, $Blocks_{Leveled} = \sum_{i=1}^{L} (\lceil sT^{i-1}/B \rceil + 1)$ blocks must be read in the worst case. In a BiLSM, a total of $\sum_{i=1}^{L} sT^{i-1}$ bytes must be read from level 1 only, which is $Blocks_{BiLSM} = \lceil (\sum_{i=1}^{L} sT^{i-1})/B \rceil + 1$ blocks. Since $Blocks_{Leveled} \geq \lceil (\sum_{i=1}^{L} sT^{i-1})/B \rceil + \sum_{i=1}^{L} 1 = \lceil (\sum_{i=1}^{L} sT^{i-1})/B \rceil + 1 + (L-1) = Blocks_{BiLSM} + (L-1)$, BiLSM can save up to $L - 1$ blocks than a Leveled LSM-tree for one range query. Therefore, we take $L - 1$ as the predicted number of saved disk blocks.

Compared with the regular LSM-tree, BiLSM pays the extra write cost for sentinel components creation, but potentially saves future disk I/O if the same query (or its sub-range) is performed again. We define a *cost function* to decide whether to create sentinel components for a Bucket Range or not. We have a tunable threshold θ. For a bucket range r, if $estimated\ benefit > \theta \times cost_{creation}$, the cost model will decide to build sentinel components for this range. The creation cost ($cost_{creation}$) of range r is the number of blocks that will be written to build sentinel components. A sentinel component is valid until the next flush. Therefore, we estimated the benefit of range r by the total number of future accesses in r before the next flush happens. We assume the data distribution in the workload is similar within several adjacent time windows. The cost function uses the previous time windows' information to estimate the future time windows' information. For a given bucket range r, the cost function computes the estimated benefit for future reads if we create a sentinel component for r. Function (1) computes the estimated benefit for r:

$$FB_i = \frac{OB_i}{OW_{total}} \times FW_{total} \tag{1}$$

In the above functions, FB_i is the future benefit for range r. OB_i is the accumulated expected number of saved disk blocks in r, which is stored in the read counter. OW_{total} is the existing write cost for the whole range, which is stored in the write counter. FW_{total} is the future write cost of the whole space, which is the number of pages that can be filled in the memory component before the next flush happens. We use the total memory component size minus its current size to be FW_{total}. Since FW_{total} can be computed at any time from the memory component, for a given bucket range r, the cost function can estimate the future benefit FB_i at any timestamp before the next flush happens.

3.2 Architecture Overview

Figure 2 shows the overall framework of the BiLSM. The left part in Figure 2 is a BiLSM, the middle part shows the workflow of processing the workload, and the right part is the cost model to detect hot ranges and make the decision to build sentinel components. Figure 2 did not include operations that are related to the memory component of BiLSM. But the query executor of BiLSM also checks the memory component. The goal of BiLSM is to reduce the disk I/O, and the cost of accessing the memory component can be neglected compared to disk access.

Read Operations. The BiLSM contains two types of read operations: Get (point queries) and Multi-Get (range queries). BiLSM handles the point queries similar to a standard LSM-tree. Reading starts from the memory component to the bottom level in the disk and stops when finds the query answer in the middle. BiLSM handles the range queries different with standard LSM-tree. In Figure 2, the system first checks if regular components are flushed after creating sentinel components. If "Yes", the query executor accesses the new regular components. Then, checks if Q is fully contained in the key ranges of all sentinel components. If "Yes", the query executor directly outputs the query results only from sentinel components and new regular components (if any). If "No", the query executor performs the scan like a standard LSM-tree. Finally, updates the hotness information of the query range according to the cost model.

Write Operations. BiLSM contains sentinel components writes and users writes, both write operations update writer counter in cost model (in Figure 2 right part). After updating the hotness information and the cost model returns a hot range, BiLSM directly inserts new sentinel components into the top level. Like a standard LSM-tree, all the users write operations inserts to the memory component first. If the memory component is full, BiLSM flushes it to the disk. When the top is full, BiLSM merges old regular components and overlapping sentinel components to lower levels. If there is no newly flushed component, sentinel components can stay in the top level and let the top level overflow.

4 EXPERIMENTS

To evaluate the performance of BiLSM, we implemented a simulator for Leveled LSM-tree and BiLSM to compare their I/O cost on eight synthetic workloads.

4.1 Experimental Settings

We tested eight workloads, each containing a loading phase of different settings to create an initial LSM-tree. Each record is a key-value pair of 1 kB (20 bytes key and 1004 bytes dummy value). The whole key space is between $[1, 10^6]$. No sentinel components were created in the loading phase.

Workload *a-d* all had a read-only (denoted by *R*) phase. Workload *b* and *d* had a loading phase of 1 million unique records. Workload *a* and *c* had a loading phase of 1.09M records where 1M were insertions and 90k were updates. Each disk component stored 2000 records. Workload *a* and *b* contained 100k range queries which the length was 6000, workload *c* and *d* contained 5k range queries which the length was 200.

Workload *e-h* all had a mix of read and write (denoted by *M*) phase, they had a loading phase of 0.99M records of equal size. Workload *e* and *f* contained 90k range queries which length was 6000 and 10k insertions, workload *g-h* contained 23k range queries which length was 120 and 10k insertions.

For all the eight workloads, the hot queries (denoted by *H*) were generated based on uniform distribution, the cold queries (denoted by *C*) were disjoint with each other and each only appeared once. In workload *e* and *g*, 90% of 10k were updates of the records in loading phase and 10% were new insertions. The insertions or updates might happen in the queries' range, we simulated this scenario in the mixed workload *e* and *g*. If updates are involved in a workload, there will be obsolete data. The experiment with "Obsolete = Yes" (denoted by *O*) simulated the obsolete data. We took a linear function to

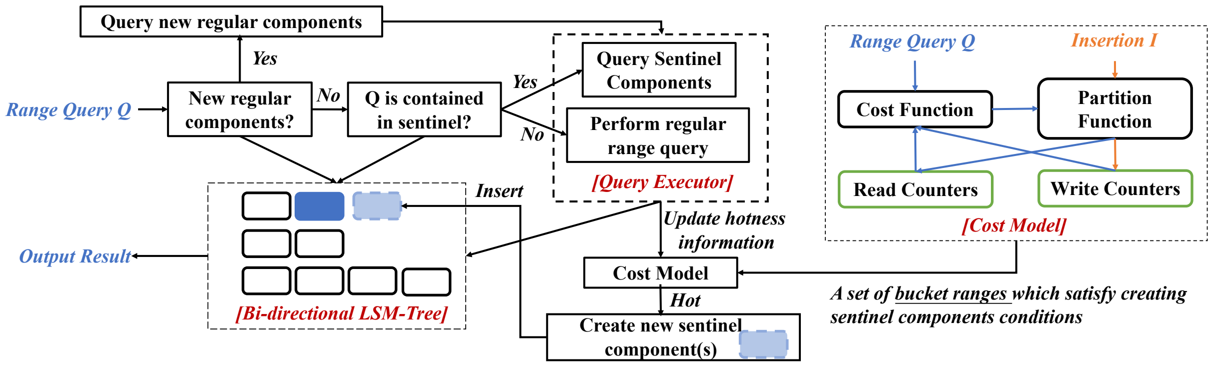

Query new regular components
Yes
Range Query Q
New regular components?
No
Q is contained in sentinel?
Yes
No
Query Sentinel Components
Perform regular range query
[Query Executor]
Insert
Output Result
[Bi-directional LSM-Tree]
Update hotness information
Cost Model
Hot
Create new sentinel component(s)
Range Query Q
Insertion I
Cost Function
Partition Function
Read Counters
Write Counters
[Cost Model]
A set of bucket ranges which satisfy creating sentinel components conditions

Figure 2: The framework of maintaining and using the BiLSM.

partition the whole key space into 500 Bucket Ranges of 2000 keys each. The threshold θ to create sentinel component was 1.

4.2 Results Analysis

We compared BiLSM with standard Leveled LSM-tree (denoted as *Leveled*) and summarized the results in Table 1. The results indicated that BiLSM has better read performance than, or is at least competitive to LSM-tree. In particular, we have the four observations: (1) In workloads containing hot ranges (a, b, e, and f), BiLSM can save more than 10% of disk I/O than Leveled LSM-tree. (2) Even for workloads without hot ranges (c and g), BiLSM can still outperform Leveled LSM-tree. Although the queries are cold, the underlying Bucket Ranges can be hot. (3) Building sentinel components can merge hot data early than regular flushes happen and eliminate the old versions of hot data in the lower levels (by updating the metadata file). In mixed workloads, the write amplification of hot data in BiLSM is smaller than in LSM-tree. Overall, mixed workloads have more benefits than read-only workloads. In Table 1, workload e and f benefited more than workload a and b. (4) If a workload contains updates and the updated records are queried frequently, by merging more eagerly to create sentinel components, obsolete data can be removed more effectively to reduce the total data size, saving more disk I/O during reads (workload e saves more disk I/O than f).

To conclude, BiLSM can reduce more I/O by accessing a smaller size of data and reducing the write amplification of the hot data. BiLSM can have benefits if the workload contains the hot accessed range and can benefit more in the mixed workload. The best scenarios of BiLSM are mixed workloads with hot queries, which are workloads e and f.

5 CONCLUSION AND FUTURE WORK

We present Bi-directional LSM-tree to move data in both top-down and bottom-up directions in an LSM-tree. Hot data is moved from lower levels to upper levels with sentinel components, improving the system's read performance. Paying write cost to create sentinel components beforehand but save lots of disk I/Os for future reads, flushes, and compactions. The results showed that Bi-directional LSM-tree outperform Leveled LSM-tree in almost all settings in the four read-only workloads and four mixed workloads.

Currently, sentinel components can only be generated from range queries, we are planning to add support for point queries to generate sentinel components as well. A better function (probably a learned function) is needed to partition the whole key space into Bucket Ranges, which can handle different key distributions. In the future, we will test our approach on a real system, like RocksDB [4].

Workload	Query Set		Total I/O Cost ($\times 10^3$)			Build SCs
	R/M	H/C	O	Leveled	BiLSM	Benefit	
a	R	H	Y	7,050	6,213	↓ 12%	380
b	R	H	N	7,050	6,156	↓ 13%	380
c	R	C	Y	2,115	2,031	↓ 4%	380
d	R	C	N	2,049	2,049	–	0
e	M	H	Y	3,535	2,834	↓ 20%	450
f	M	H	N	3,300	2,784	↓ 16%	436
g	M	C	Y	856	842	↓ 2%	87
h	M	C	N	756	756	–	0

Table 1: The total disk I/O cost for different workloads. I/O cost: the number blocks read and written. e.g. total I/O cost = read + written 7050 $\times 10^3$ blocks, write cost to build sentinel components = written 380 blocks. SCs: sentinel components. R: Real-only. M: Mixed. H: Hot. C: Cold. O: Obsolete.

ACKNOWLEDGMENTS

This work was supported by the National Science Foundation (NSF) under grants IIS-1838222, IIS-2046236, and CCF-2103483.

REFERENCES

[1] Sattam Alsubaiee et al. 2014. AsterixDB: a scalable, open source BDMS. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1905–1916.
[2] Oana Balmau. 2017. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *ATC*. USENIX, 363–375.
[3] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal navigable key-value store. In *SIGMOD*. 79–94.
[4] Facebook. 2012. RocksDB. https://rocksdb.org/
[5] Google. 2011. LevelDB. https://github.com/google/leveldb
[6] Bradley C Kuszmaul. 2014. A comparison of fractal trees to log-structured merge (LSM) trees. *Tokutek White Paper* (2014).
[7] Chen Luo and Michael J Carey. 2020. LSM-based storage techniques: a survey. *The VLDB Journal* 29, 1 (2020), 393–418.
[8] Qizhong Mao et al. 2019. Experimental Evaluation of Bounded-Depth LSM Merge Policies. In *Big Data*. IEEE, 523–532.
[9] Qizhong Mao et al. 2021. Comparison and evaluation of state-of-the-art LSM merge policies. *The VLDB Journal* 30, 3 (2021), 361–378.
[10] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.
[11] Feng-Feng Pan, Yi-Liang Yue, and Jin Xiong. 2017. dcompaction: Speeding up compaction of the lsm-tree via delayed compaction. *Journal of Computer Science and Technology* 32, 1 (2017), 41–54.
[12] Dejun Teng et al. 2018. A low-cost disk solution enabling LSM-tree to achieve high performance for mixed read/write workloads. *TOS* 14, 2 (2018), 1–26.
[13] Wenshao Zhong et al. 2021. REMIX: Efficient Range Query for LSM-trees. In *FAST*. USENIX, 51–64.