Tiered-Indexing: Optimizing Access Methods for Skew

The VLDB Journal (2025) 34:45
https://doi.org/10.1007/s00778-025-00928-6

REGULAR PAPER

Tiered-Indexing: Optimizing Access Methods for Skew

Xinjing Zhou[1] · Xiangpeng Hao[2] · Xiangyao Yu[2] · Michael Stonebraker[1]

Received: 3 April 2024 / Revised: 10 February 2025 / Accepted: 8 May 2025 / Published online: 24 May 2025

Abstract
Real-world DBMS workloads invariably exhibit skewed access patterns, where a small number of "hot" records are accessed much more frequently than the remaining "cold" records. Page-oriented data structures, such as B+trees, dynamic hash tables, heap files, and LSM-tree, are sub-optimal in terms of memory utilization under skewed access conditions. Hot records might be co-located with cold ones on pages in the data structure. Caching those lukewarm pages in the buffer pool lowers memory utilization due to the mismatch of caching granularity (page) and access granularity (record), leading to sub-optimal performance. Recently, the 2- TREE approach was proposed to improve caching efficiency for B+trees using record-level migration. In this paper, we generalize the 2- TREE approach to TIERED- INDEXING that can be applied to common buffer-managed data structures to efficiently handle skew using record migration. Using this architecture, we extend hash tables, heap files, and LSM-trees with I/O-efficient record migration. Moreover, we design a general mechanism to ensure data structure consistency for TIERED- INDEXING data structures during record migration using optimistic lock coupling. Compared to traditional 1- TIER and state-of-the-art record-caching designs, we observe significant throughput and memory utilization improvement across B+tree, hash table, heap file, and LSM-tree under skewed workloads.

Keywords Buffer management · Skew · Indexing · B+tree · LSM-tree

1 Introduction

Skew is invariably prevalent in real-world DBMS workloads. The working set of a workload has a much higher access frequency than the remaining cold data [3, 5, 6, 43, 52]. In most of these workloads, users access data at record granularity (e.g., tuple, index entry, or key-value pair) which is typically much smaller than a page [3, 6, 43]. Consequently, "lukewarm" pages containing both hot and cold records arise, as each page in the buffer pool may be pinned by a single hot record. This results in cold records occupying precious memory space and lowering overall memory utilization. Such lukewarm pages are widespread across various buffer-managed data structures, though for different reasons. For instance, in B+trees, hot records might be scattered across leaf pages if the access pattern does not align with the tree's key ordering [10], rather than concentrated in one or two small sub-ranges. In contrast, heap files store data with no particular order, so hot records can be dispersed randomly in the file. Hash-based data structures (e.g., dynamic hash tables) also distribute hot records throughout the hash buckets by design. In log-structured merge trees (LSM trees) [39], compactions may push read-hot records to lower levels, where they become co-located with colder records, creating lukewarm pages. Moreover, if a previously cold record becomes hot again - as often happens under shifting skew distributions-lukewarm pages will re-emerge, and searches may have to traverse most or all levels of the hierarchy, further increasing query latency.

This paper introduces TIERED- INDEXING, a general approach to improve buffer-managed data structures under skewed access. To represent an index structure, TIERED-INDEXING maintains a hierarchy of index structures with different hotness that share a buffer pool. The index structure on each tier must support the same set of operations as

✉ Xinjing Zhou
xinjing@mit.edu

Xiangpeng Hao
xiangpeng.hao@wisc.edu

Xiangyao Yu
yxy@cs.wisc.edu

Michael Stonebraker
stonebraker@csail.mit.edu

[1] MIT CSAIL, Cambridge, US

[2] University of Wisconsin-Madison, Madison, US

the original 1-tier structure. TIERED- INDEXING actively performs efficient inter-tier record migration based on record hotness.

TIERED- INDEXING is not the first attempt at solving this problem. One line of effort investigated record caching [2, 17, 31, 34, 44, 53], which caches hot records from pages in primary storage. However, TIERED- INDEXING has several advantages over record caching. First, a record cache cannot be used for range queries [44, 51], requiring users to balance the memory budget between a record cache and a page cache. Sub-optimal examples of such balancing include TreeLine [53], which dedicates all memory to a record cache, leaving no memory for range scan. Instead, TIERED- INDEXING dedicates all the memory to the buffer pool and can flexibly use buffer frames to serve both point and scan queries. Second, supporting durable writes for a record cache is challenging: it either needs to write through the cache (slow) or implement record-level recovery along with page-level recovery protocol (complex) [2, 53]. As a result, many record caches [31, 34, 44] are read-only. In contrast, TIERED- INDEXING can leverage widely adopted page-based recovery [22, 37] to support durable writes easily and efficiently. Finally, record caching [44] for LSM-tree does not address the problem of sub-optimal record placement in the hierarchy. In contrast, TIERED- INDEXING applies record migration to LSM-tree to improve the placement.

TIERED- INDEXING is a generalization of 2- TREE [56], which maintains two B+trees with record-level data migration. However, 2- TREE [56] focuses on clustered B+trees and lacks support for concurrency and a comparison against state-of-the-art record-caching designs [31, 44, 53]. This paper covers all the missing pieces.

Using the TIERED- INDEXING architecture, we propose 2- HASH and 2- HEAP, respectively, as the 2- TIER skew-optimized hash table and heap file with tailored and I/O-efficient migration strategies. 2- HASH and 2- HEAP outperform 1- TIER designs significantly on skewed workloads. The proposed migration strategies are critical for high memory utilization under skew, fast reaction to hot spot distribution shift, and efficient data loading. The strategy for 2- HASH can even significantly improve data loading performance, which is a challenge for 1- TIER hash tables.

Furthermore, we present an N- TIER structure, called BILSM- TREE, adapted from LSM-tree by adding record migration to improve buffer pool hit rate. The challenge is to perform the migration without degrading its write efficiency and range scan performance. BILSM- TREE achieves this by introducing new upward data migration paths and three techniques to selectively and adaptively migrate records according to their hotness at low cost. Such migration also reduces the number of trees that need to be accessed during point reads and improves query latency. BILSM- TREE equals or beats traditional LSM-trees and LSM-trees with record-caching on skewed point reads, blind writes, updates, and range scans.

Previous work [56] did not support maintaining data structure consistency during migration with multiple concurrent threads. This is not a problem for 1- TIER designs. To fill this gap, we designed a concurrency control mechanism based on the well-known optimistic concurrency control for TIERED- INDEXING structures with minimal overhead for read operations.

We summarize our contributions as follows:

- We generalize 2- TREE and present a TIERED- INDEXING architecture that optimizes memory utilization for buffer-managed access methods under skew.
- We present 2- HASH and 2- HEAP as 2- TIER data structures with I/O-efficient migration strategies that greatly improve the performance of skewed workloads and data loading.
- We present BILSM- TREE, an LSM-tree variant and a realization of an N- TIER structure. BILSM- TREE adaptively migrates records in the tree hierarchy to improve memory utilization and read performance at low cost. Unlike the record-caching approach, BILSM- TREE does not negatively impact the performance of range scan queries.
- We propose a concurrency control scheme to ensure that TIERED- INDEXING structures remain consistent during concurrent migration and other operations.
- We thoroughly evaluate TIERED- INDEXING designs against 1- TIER and record-caching designs on skewed workloads.

2 RELATED WORK AND MOTIVATION

This section discusses recent work in main-memory databases, record caching, log-structured storage structures, and buffer management related to this paper. This section also explains why previous work does not address the skew problem sufficiently.

Record Caching. One general approach for improving memory utilization of page-based storage engines is maintaining separate in-memory caches for hot records [2, 17, 24, 31, 34, 44, 53]. Although such 1- TIER design with record cache resembles TIERED- INDEXING in terms of structure, they differ in the following ways:

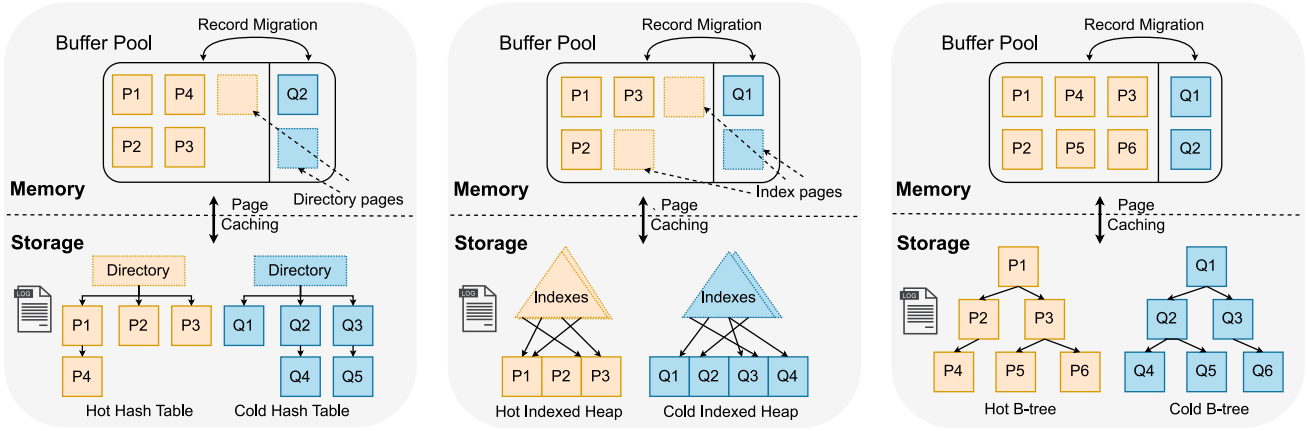
- **Per-Record Space Overhead**. Many record-cache-based designs (e.g., AntiCaching, TreeLine, RocksDB) stored records in dynamically allocated memory regions to support variable-sized records. They also typically maintain records in a LRU list and hash table which both add per-

record metadata space overhead. For example, RocksDB requires 64 bytes of metadata per in-memory record for LRU and in-memory hash table. TreeLine requires 24 bytes per record which is smaller compared to RocksDB due to its use of Clock replacement policy. All of these allocations also create internal fragmentation from the memory allocator. In the case of 2-BTree, the records are stored inline in the leaves of buffer-managed B-tree with 3-bit metadata for tracking hotness. Although these can be thought of as implementation details, they have a considerable impact on the actual resident memory footprint per record, especially considering small records in the OLTP setting.

– **Durable Writes and Recovery**. Most record caches are read-only for simplicity. To support durable modifications, one either needs to do a costly write-through operation to the record cache or devise a recovery mechanism that works for both the record cache and the page-based storage engine. Different from a record cache, our design builds on a page-based buffer manager only. Therefore, we can leverage well-known page-based recovery mechanisms [22, 37] to support durable writes and recover easily and efficiently.
– **Range Query.** Record caching is also not as versatile as page caching [51] as most record caches are designed to support point queries [31, 44]. Record cache could potentially accelerate query on a range if all records in the range are hot and cached. TIERED- INDEXING differs from record caching in that we build only on a buffer pool whose buffer frames can serve both point and range queries. One could argue that a record cache implemented with a sorted structure (e.g, B+tree or skip list) might help with range scan operation. This is true in the case of B+tree. For LSM-tree, the range scan operation requires merging overlapping ranges in all levels on storage. Even if the record cache is sorted, the range scan operation still needs to scan all blocks in the overlapped ranges of all the levels on storage. In this case, the sortedness of the record cache does not really help at saving I/Os. Whereas if the buffer pool is given more memory, it could potentially cache more sorted blocks of SST files, reducing the number of I/Os.
– **Memory Budget Tuning.** While a record cache that only supports point query surely improves the memory utilization for skewed point queries, one needs to decide on the amount of memory budget allocated to the record cache. Allocating too much memory to record cache harms the performance of range queries as fewer buffer frames can be used to serve range queries. TIERED- INDEXING dedicates all the memory budget to the buffer pool and therefore can adapt to the workloads as it uses the buffer pool to serve both point and range queries.
– **I/O Efficient Migration**. As we demonstrated in the experimental evaluation section, naive eviction using a general LRU policy or Clock policy from a record cache without paying attention to the data structure characteristics has a large impact on I/O performance. TIERED- INDEXING introduces I/O efficient record migration tailored for each of the four widely-used data structures from the hot tier to the cold tier, significantly outperforming general LRU policy.

Main-Memory DBMS. Another line of related work is main-memory databases [13, 23, 25, 42] where data are managed at tuple granularity. They have also been extended to support data sets larger than memory capacity [12, 15, 41]. Anti-Caching [12] manages hot tuples in memory and evicts cold tuples to disk. This design maximizes memory utilization. However, it keeps all indexes in memory, limiting its scalability [54]. Also, it cannot handle range scans efficiently because tuples on the disk are unordered. Unlike Anti-Caching, Siberia [15], completely separates hot tuples from cold ones by storing them in a hot store (in memory) and a cold store (on disk). Our approach differs from Siberia's in two major ways. First, Siberia does not migrate data from the cold store to the hot store upon read. This is sub-optimal for read-only or changing workloads. Second, Siberia identifies hot data through offline analysis of access logs. In contrast, our lightweight online approaches require fewer computing resources and result in more timely adaptation to workload changes. Virtual memory and OS paging [41] have also been exploited to support larger-than-memory databases. In contrast, TIERED- INDEXING builds on modern buffer managers that scale better and degrade more gracefully [27]. Furthermore, our record migration addresses the above-mentioned limitations. Our approach builds on the buffer manager interface and does not keep any metadata of evicted data in memory. Therefore, our approach is more scalable. Second, our approach can easily support efficient range scan. Third, our record-level migration is bidirectional.

Log-Structured Storage. Another line of related work is log-structured storage [9, 24, 30, 39], which focuses on improving write performance. One classic example is the log-structured-merge tree (LSM-tree [39]). The idea behind the LSM-tree is to turn random writes into sequential ones by first buffering writes in memory and then later sort-merging them with data on disk. Modern LSM-trees [11, 44] organize data using multiple levels on disk, where recently written data are stored on higher levels. Data migrates from top to bottom through flush and merge operations triggered by writes. TRIAD [4] reduces the merging overhead of LSM-tree by keeping frequently updated records in memory. Bw-tree [32] is a design that adopts log-structured storage in a buffer-managed B+tree. Updates to a tree node are processed by



(a) 2-Hash (b) 2-Heap (c) 2-Tree

Fig. 1 2- TIER designs for dynamic hash table, indexed heap files, and B+tree

chaining them as delta records off of base tree nodes. Bw-tree trades off read performance for write performance [49]. Log structuring has also been studied in the context of hash tables on persistent memory. Plush [48] optimizes the write performance of hash tables on persistent memory [1] by employing log-structuring to reduce the write amplification. Although TIERED- INDEXING also employs multiple structures, it actively performs upward data migration, which keeps hot records at the higher tiers of the hierarchy for maximum performance and good memory utilization. And TIERED- INDEXING considers record hotness for downward migration to ensure hot records stay at the highest tier.

Buffer Management. Buffer managers [14] play a crucial role in caching hot data in database systems. However, when data fit in memory, they become a bottleneck [21]. There have been recent efforts to remove such bottlenecks. LeanStore [27] and Umbra [38] eliminates the need for page lookup table using pointer swizzling [19, 50], achieving in-memory performance. VMCache [26] leverages virtual memory hardware to manage the buffer pool more efficiently. There are also designs [47, 55] that incorporate persistent memory as a second layer of memory for better cost-effectiveness. All these efforts assume data structures are page-based. Therefore, they still suffer from memory utilization issues with skewed workloads. Our approach introduces record-level migration on top of buffer managers and benefits from these advances.

3 TWO-TIER DATA STRUCTURES

This section presents 2- HASH and 2- HEAP as skewed-optimized 2- TIER hash tables and indexed heap file under TIERED- INDEXING architecture. We also cover 2- TREE for completeness. We emphasize new migration strategies tailored for each data structure as they are critical for achieving good performance. For convenience, we use "record" throughout the rest of the paper in multiple contexts. We use record to represent a keyed database tuple or an index entry that stores the primary key and/or the location of the tuple.

3.1 Design Overview

Figure 1 illustrates three classic data structures after applying the 2- TIER design principles. Generally, a single data structure is decomposed into two. All of these designs follow the principle that they share a single buffer pool which runs its own replacement algorithm at the page level. To differentiate hot and cold tiers, the buffer pool is logically partitioned into two regions: hot and cold. The hot region takes up the majority of the buffer pool capacity. The buffer frames in this region cache pages from the hot tiers of the proposed designs and are not considered for eviction. The frames in the cold region cache pages from the cold tier and are considered for eviction. Note that the buffer frames in the hot region are still considered for the checkpointing process, where dirty pages are written back to the secondary storage to ensure that the WAL does not grow indefinitely. In addition to running a page-level replacement algorithm that tries to keep hot pages in memory, 2- TIER design runs a record-level data migration algorithm on top of the buffer pool to ensure that hot records stay in the hot tier while cold ones stay in the cold tier. The record eviction happens when the size of the hot tiers exceeds the capacity of the hot buffer pool region. Each record in the hot tier is extended with one reference bit, one dirty bit, and one deletion bit for the purpose of such migration. The reference bit is set when accessed. TIERED- INDEXING design combined with record-level migration ensures high memory utilization for buffer frames in the hot tier. We next dive into the details of each design.

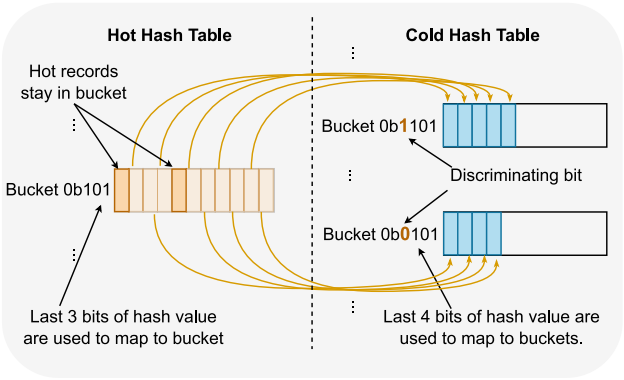


Fig. 2 Illustration of 2- HASH downward record migration using a shared hash function; the cold hash table (2^4 buckets) is twice as large as the hot hash table (2^3 buckets); Evicted records from bucket 0b101 are split into two group by the fourth bit of hash value and migrated to two buckets in the cold hash table

3.2 2-HASH Design

In this work, we focus on linear hashing [33]. We assume that the number of buckets in the hash table is always a power of two. This is typically true in dynamic hash schemes, such as extendible hashing [16] and linear hashing [33]. In 2- HASH, a hot hash table is maintained to keep hot records, while a cold hash table keeps cold records. When the hot hash table exceeds a size threshold, an eviction algorithm is run to migrate the coldest records from the hot hash table to the cold hash table.

Downward Migration. The design goal of downward migration is to minimize I/O overhead during eviction. This might seem difficult at first glance, as each evicted record from the hot hash table might get randomly distributed to buckets in the cold hash table. Therefore, similar to 1- TIER hash table, each record is potentially going to incur an I/O on a small buffer pool on eviction-heavy workloads such as data loading and distribution shift.

Shared Hash Function. However, we make a key observation that in these dynamic hashing schemes, records in the same bucket have the same hash value suffix bits. If we use the same hash function between the hot hash table and the cold hash table, we can reduce the number of I/Os significantly. As shown in the example in Figure 2, by sharing the hash function between hash tables, records from the same bucket in the hot hash table will only end up in two buckets in the cold hash table when they get migrated. If the two tables are independent, every record migrated from the hot table could potentially end up in a different hash bucket of the cold table. More generally, let N and M be the number of buckets in the hot and cold hash table. Let B be the average number of records that can be stored in a hash bucket. For a sufficiently loaded hot hash table, B is typically close to the capacity of the hash bucket. Assuming uniform hash functions, we can derive the average number of buckets that will be accessed in the cold hash table when moving all records from a bucket in the hot table as $min(B, \lceil \frac{M}{N} \rceil)$ if the hash function is shared. For independent hash tables, the cost is simply B. Therefore, sharing the hash function results in an asymptotically better design for eviction. Note that $\frac{M}{N}$ is the size ratio between the cold and hot hash tables.

This observation leads us to a more efficient eviction strategy by leveraging such locality. When the hot hash table exceeds the size threshold and cannot accommodate new records, 2- HASH scans the table buckets by bucket order (e.g., bucket 0b100 is examined before bucket 0b101) for eviction. Within each bucket, records are examined for eviction according to the reference bit. Since such scanning typically examines records in a bucket consecutively, the I/O overhead for eviction is significantly reduced when sharing a hash function between hot and cold tiers. Furthermore, sequentially scanning buckets of the hot table also results in more sequential I/O patterns on the cold table if the pages representing adjacent buckets (e.g., 0b101 is adjacent to 0b100) of the cold table are stored consecutively on storage devices.

Although our design favors sharing the hash function between the hot and cold hash tables, it does not make further assumptions about the implementation of the hash table. Both linear hashing and extendable hashing schemes can be used as individual tables.

3.3 2-HEAP Design

We now discuss how to apply TIERED- INDEXING with two tiers to heap files, obtaining 2- HEAP. In this work, we only consider indexed heaps, or non-clustered indexes, which are heap files with indexes built on top of them. This is because, without indexing, the only access method for a heap is a sequential scan. This is not practical for OLTP or key-value workloads. Note that 2- HEAP is also applicable to key-value stores based on an indexed log [8, 9, 35, 40] since a log file is similar to a heap file in terms of storing payloads. We assume the index maintains pointers (page id + slot id) to the records (tuple) stored in the heap file. As shown in Figure 1, 2- HEAP consists of two heaps, each with its own set of indexes. The index could be a B+tree, hash table, or LSM-tree. Similarly, the hot heap maintains the hot tuples along with its indexes. Upon memory pressure, records are removed from the hot heap and migrated to the cold heap. Index maintenance on both heaps is done during the migration step.

Downward Migration. As illustrated in Figure 3, one natural way (Strategy ①) to evict records is by scanning the heap record by record, utilizing the sequential scan interface provided by a standard heap file. This is simple and intuitive. However, this might not be optimal. This is because the I/O

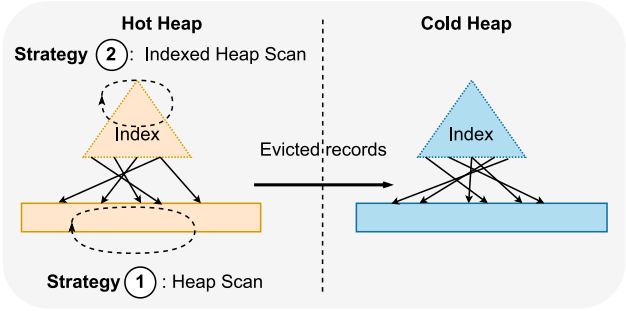


Fig. 3 Illustration of 2- HEAP downward record migration

pattern of maintenance of the index on the cold heap might be random because records evicted from the hot heap have no order. A better strategy is to perform an indexed scan on the heap (Strategy ②). The indexed scan allows scanning tuples in the heap based on the order within the index. This is based on the observation that the indexes of the hot and cold heap share the key space. This allows for a more sequential I/O pattern on index maintenance of the cold heap compared to heap scan. Although the access pattern to the hot heap could be random due to the indexed scan, there is hardly any I/O cost, since the hot heap is well-cached in the buffer pool.

Note that if more than one index is maintained on the heap, this strategy would only reduce the overhead of one index on the cold heap. However, it is strictly better than the heap scan strategy, where every index has to be maintained at a relatively higher cost. We also emphasize that such settings are common in practice. For example, most relations in TPC-C only have one index. Heap-based key-value stores [8, 35] typically have only one index.

3.4 2B+tree Design

For completeness, we also include the description of 2B+tree introduced in [56]. 2B+tree consists of two B+trees. The hot B+tree in 2B+tree maintains hot records, while the cold B+tree stores mostly cold records that cannot be kept in the hot tier. When the hot B+tree exceeds a size threshold, a record-level migration algorithm is run on the hot tier to evict the coldest records to the cold B+tree using the buffer pool. 2B+tree leverages range scan operation supported by B+tree to implement an approximate clock-based replacement strategy. During eviction, it walks the hot B+tree in key order and evicts records without the reference bit on. Since the hot tier is mostly cached in the buffer pool, such a scan incurs almost no I/O overhead. Moreover, such scan-based record eviction produces an I/O pattern that is relatively sequential when accessing the cold B+tree. This is because the hot and cold B+trees both index on the same key space. Evicted records from the hot B+tree typically fall within a key range represented by a small number of consecutive leaf pages in the cold B+tree. Thus, I/O overhead is often amortized over a batch of record insertions/updates to the cold B+tree.

3.5 Upward Migration and Exclusivity

For upward data migration of all the data structures, we adopted the same sampling-based approach proposed in 2- TREE [56] where only the sampled records are moved upward. We define the sampling rate as $F (0 < F \leq 1)$. As data becomes hot, its access frequency increases and it will be more likely to end up in the sample set. Hence, our approach provides a tunable level of thrash resistance with minimal memory and CPU overhead. Our approach also presents a trade-off between the cache warm-up rate and the level of thrash resistance. A large sampling rate warms the cache quickly while providing little resistance to thrash. In contrast, a small sampling rate delivers thrash resistance by sacrificing the warm-up rate. This is a general technique applicable to all three data structures.

Another important decision when migrating records upwards is determining how many copies of a record to keep in the system. Under an exclusive policy, the system only keeps one copy of the record in either the hot or cold tier. When migrating data from the cold tier upward, the record is erased from the cold tier. In contrast, under an inclusive policy, the record is retained in the cold tier when copied to the hot tier. The inclusive policy avoids modifying the cold tier during migration, which benefits IO-bound workloads. However, it results in a larger overall tier size as duplication increases. Finding the best sampling rate is a typical knob-tuning task that is out of scope for this paper, and many automatic techniques [45, 46, 55] are available. For simplicity, in this paper, we manually find the sampling rate that works well in our evaluation. We leave systematic tuning of such parameters as future work.

3.6 Durability and Recovery

One advantage of using a buffer manager for both the hot and cold tiers is that the durability and recovery can be easily supported. Since both tiers are page-based, we reuse the prevalent physiological write-ahead-logging and ARIES [37] protocol for the durability and recovery of the three proposed designs. We notice that migration does not change logical user content (e.g., tuples) but only changes the physical representation of the data structures. Therefore, we use system transactions [18] as a general approach to ensure the atomicity and durability of migration. System transactions do not change the user content and only modify the physical representation of user content(e.g., B+tree node split). System transactions are lightweight as they do not need to force log records to stable storage upon commit. A system transaction could consist of migrating a single record during normal

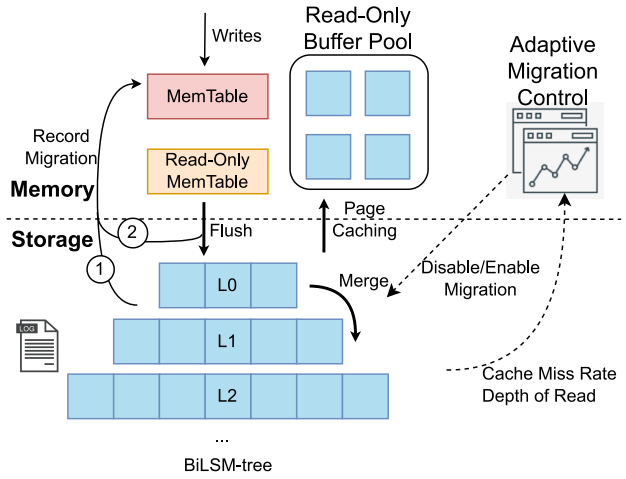


Fig. 4 Illustration of Bi-directional LSM-tree.

upward migration operation or multiple records during downward migration. For each migration system transaction, we record redo and undo log records of pages changed by the transaction to log file. When a system transaction commits, it also records a commit record. However, these log records are not forced to disk upon commit. Instead, the hardening of these records is implicitly carried out by later user transactions that force its log records to storage.

4 BIDIRECTIONAL LSM-TREE

In this section, we present a new LSM-tree variant called **BiLSM-tree**: **Bi**-directional **LSM-tree**. BiLSM- tree is a realization of an N- Tier data structure under Tiered-Indexing. Besides the downward data migration that exists in a traditional LSM-tree, BiLSM- tree adds two additional upward data migration paths that promote hot records to be closer to the top of the hierarchy for point read operation, as shown in Figure 4. Since LSM-tree does not support in-place update to pages on storage, such migration has to first go through the memtable, similar to a user write. This brings a challenge: migration incurs write overhead even for read-only operations. We next propose three techniques in BiLSM- tree for achieving migration with low overhead.

Path ①: Exponential Sampling Rate. To reduce write overhead caused by migration, path ① identifies and migrates only warm record read from storage. We define the depth $D(0 \leq D)$ of a point read as the number of files accessed to complete the operation. BiLSM- tree identifies warm records using a sampling-based approach similar to the proposal in 2- Tree. Different from a constant sampling rate in 2- Tree, BiLSM- tree set the sampling rate as $B \cdot R^D$ where $B(0 < B \leq 1)$ is the base sampling rate for the first file in the hierarchy, and $R(1 < R)$ is the growth factor between consecutive levels. R serves a similar purpose as the size ratio between consecutive levels in LSM-tree. Therefore, the sampling rate is exponential with respect to the depth of the point read. Records served from a file deeper in the hierarchy will have a higher probability of being migrated. The intuition is that warm records deeper in the hierarchy benefit more from migration as they are more likely to be clustered with cold records. Shallow records will be less likely to be migrated as they are expected to be clustered well with hot records. Furthermore, we distinguish point reads that incur cache miss from those without cache miss. For the latter, we set its base sampling rate (B_{hit}) to be much smaller than that of the former (B_{miss}). This reflects the fact that cache hits often signify the record being in the working set. Therefore we should use an even smaller sampling rate to further reduce unnecessary migrations.

Path ②: Early Migration. The second data migration path is triggered by a flush of a read-only memtable. During a flush, BiLSM- tree identifies records accessed frequently and migrates them directly to the active memtable without writing them to storage as part of the flush. This serves two purposes. First, it retains hot data at the highest level to reduce the performance impact caused by a flush. Second, it reduces write overhead by reducing the amount of data being written to storage since hot records will very quickly get migrated through path ① again. BiLSM- tree identifies read-hot records by maintaining a reference bit per record stored in the memtable. Point reads served by memtable set the reference bit. During a flush, BiLSM- tree scans through records in the memtable and migrates records with the reference bit on. To accommodate write-heavy workloads, BiLSM- tree migrates at most half of the records to the active memtable. This ensures the active memtable has enough space for buffering writes from user. This also ensures that the amount of data flushed to storage will not be tiny and will be at least half the size of the memtable. This retains the asymptotic property of the LSM-tree look-up. To handle the case where newly-written records get repetitively migrated through path ② and never get persisted on storage, BiLSM- tree only migrates records through path ② when they have already been migrated through path ①. Therefore, BiLSM- tree keeps another bit (migrated) per record for such a purpose. Records migrated through path ① will have this bit on.

Adaptive Migration Control. Note that even with these optimizations applied, our upward migration still has a negative impact when the workload is stable and being run long enough. Therefore, we introduce a control mechanism based on hill-climbing to determine if the migration is helpful and react accordingly. We use a combination of two system metrics to capture performance: M = average block cache miss rate and D = average depth of point read divided by the num-

```
OnControlStep(t,miss_rate,depth,M,TH_L, TH_H):
W.Add(f(miss_rate,depth;t)) # Keeps a window
   of past M observations smoothed by Double
   Moving Average
L = fit a trend line over the window W using
   linear regression
S = Number of records in the database that
   are covered by 90% of the accesses #
   Periodically computed by sampling
if migration not disabled and
   num_records_migrated >= S and |L.slope| <=
    TH_L:
    disable migration
    num_records_migrated = 0
else if migration disabled and |L.slope| >=
   TH_H:
    enable migration
```

Fig. 5 Adaptive Migration Control for BILSM- TREE

ber of levels in the system. The former signifies the memory utilization of the system, and the latter is a good proxy for how well the placement of records is in the tree hierarchy. We define a benefit function f at time t as $f(M, D; t) = M + D$. This function is able to capture both memory utilization and record placement quality over time, and it generally decreases as the migration continues. BILSM- TREE monitors block cache miss rate and average depth of point read in the system. BILSM- TREE periodically executes the control algorithm described in Figure 5 with values of the observed M and D since the last execution. The algorithm maintains a window of observations of function f. We apply double moving average to the window to smooth out spikes caused by flush and merges. The algorithm then fits a trend line over the window using linear regression. When the absolute value of the slope of the trend line is below a threshold value TH_L, it suggests the improvement brought by migration is plateauing. Therefore, BILSM- TREE stops the migration, at which point it becomes an ordinary LSM-tree. When the absolute value of slope is above another threshold TH_H, it suggests a drastic change in the shape of LSM-hierarchy or a distribution shift. Therefore, BILSM- TREE re-enables the migration. Furthermore, for each migration cycle, BILSM- TREE will perform at least S migrations where S is the size of the working set contributing to 90% of the accesses. This is to ensure migration covers the majority of the records in the working set. S is roughly obtained by sampling the accesses and computing the 90th percentile of the access distribution. For example, for a Zipfian skew factor of 0.9, the 90th percentile of the accesses covers 34% of records in the database and $S = 0.34 \cdot N$ where N is the number of records in the database.

4.1 Discussion On Tuning Parameters

We next share our experience on tuning the parameters introduced in the previous section. First of all, increasing B_{hit} and B_{miss} generally results in more aggressive upward migration, hence faster stabilization. The growth factor R and B_{hit}/B_{miss} together determine the migration probability on each level, much like the capacity of each level in an LSM-tree is controlled by the size ratio between consecutive levels and the capacity of memtable. We find a small B_{miss} in the range of $(0.01, 0.1]$ and R in the range of $[1.5, 2.5]$ generally strikes a good balance in the migration overhead and time to reach stabilization. Furthermore, the system is expected to serve many more requests on block cache hit than miss in unit time due to lower memory latency. Hence, to avoid aggressive upward migration on cache hit causing high write overhead and churn, it is recommended that B_{hit} is at least an order of magnitude smaller than B_{miss} to capture the latency difference of memory and storage device. The threshold parameters TH_L and TH_H in our adaptive control algorithm are designed to capture significant shifts or stability in workload behavior by monitoring trends in cache miss rates and average read depth. A value of zero for TH_L means the benefit curve is a horizontal line, reaching a stable performance. However, in practice, it is hard to reach such a state as the migration and compaction will likely cause throughput fluctuation. Hence it is recommended that TH_L to set to a value close to zero. We find the value of 0.01 works well. TH_H is designed to capture a significant shift (large slope) of the benefit curve, suggesting the change in workload or LSM-tree shape. We find a value of 0.1 works well for detecting changes in workload and tree shape.

The discussed recommendations and the impact of adjusting these parameters are obtained through manual tuning. Hence it is by no means a thorough exploration of the knob space. We leave the systematic exploration of the knob space as future work.

4.2 Durability

BILSM- TREE can leverage system transactions for durability introduced in Section 3.6 because record migration in BILSM- TREE is simply moving a user record to a higher place in the hierarchy. Like a user write, such migration requires writing a log record to the write-ahead-log for the memtable for recovery. However, it does not require the log record to be forced into storage. The log records generated by migrations can be piggybacked with the forcing of log records from user writes, which reduces overhead.

4.3 Comparison to LSM-tree with Record-Caching

A common approach to improve traditional LSM-tree under skew is record caching. For example, RocksDB has a row cache feature that keeps frequently accessed records of SST files in memory. BILSM- TREE distinguishes itself from such design in the following aspects.

Range Scan Query.The record cache in such a design can only serve point queries, reducing the memory budget from the block cache, which is essential for range scan queries and compactions. BILSM- TREE, on the other hand, dedicates all the memory to the block cache. This ensures the maximum cache space to be used for scan queries. BILSM- TREE then adaptively migrates hot records to the top of the hierarchy, improving the utilization for skewed point queries. As demonstrated in the experimental evaluation, BILSM- TREE significantly improves the skewed point queries with minimal impact on the performance of range scan queries.

Space Overhead. Keeping a row cache also potentially results in a higher space/memory overhead. For example, RocksDB's row cache uses `<row_cache_id, sst_file_id, seq_no, user key>` as the key to hot record. Therefore, the user key could possibly be stored in both the record cache and the block cache. BILSM- TREE does not duplicate such keys as there is no record cache. BILSM- TREE uses 2-bit metadata per record in the memtable to track read hotness and avoid repeated migration, translating to more overall cache space.

5 HANDLING CONCURRENCY

One important problem not addressed in 2- TREE [56] is that migrations introduce inconsistency when there are concurrent migrations and concurrent read/write/scan operations in the workloads. For example, thread X misses the hot tier on reading key K and tries to read from the cold tier. However, it is possible that another thread, Y migrated K from cold to hot before the read of thread X on the cold tier even starts. From the perspective of thread X, the record with key K does not exist, whereas, in fact, it does. Similar anomalies also exist in BILSM- TREE.

One strategy to protect against this kind of concurrency anomaly is locking. Specifically, we can use a lock table to store record-level locks, shown in Figure 6. Before searching for a keyed record in the two tiers, a read lock associated with that key must be obtained. Similarly, before performing any write operation (insert/delete/update) on a keyed record, a write lock must be obtained. During the migration of a record, its write lock must be obtained before proceeding to the real migration. This protocol correctly serializes concurrent operations on a keyed record. However, it also introduces

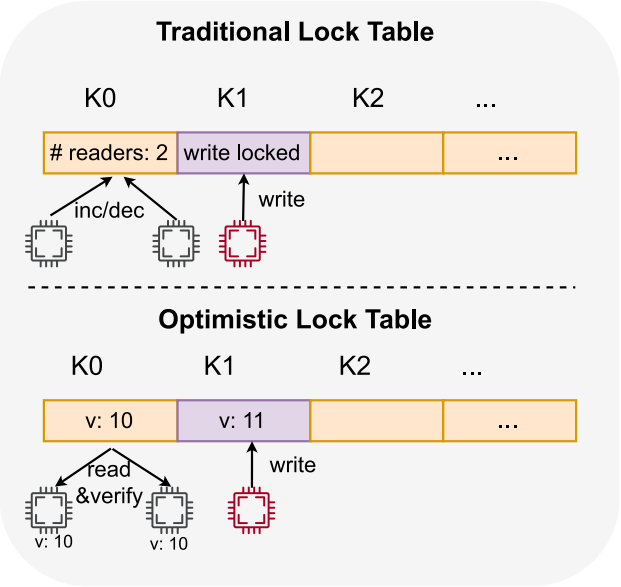


Fig. 6 Illustration traditional locking and our proposed optimistic locking

cache contention, because even read-only operations would now require writes to shared memory for keeping track of the number of readers. The problem is particularly pronounced in OLTP workloads, where a small set of keys may receive a majority of accesses due to skew. We next show how to eliminate this overhead using optimistic locking during read operations.

5.1 Optimistic Locking for Migration

To address this problem, we propose a general and more scalable approach using optimistic concurrency control. We took inspiration from optimistic lock coupling [7, 28, 29], a technique that has been used to synchronize concurrent accesses to physical nodes of a tree data structure with high scalability for reads. We show how to adapt this technique to the logical level in 2- TIER and BILSM- TREE designs to provide high scalability for read-heavy workloads.

Instead of keeping track of the number of readers for each key, our approach associates a version number with each key in a lock table, shown in Figure 6. For read operations, reader threads first obtain a private snapshot of the version number associated with a key and proceed to actually access the data structures. The version number is only atomically incremented by writer threads that perform writes (insert/delete/update) or migration at the start and end of their operation. Hence, an odd version number indicates that the key is being write-locked. This serializes the writers. At the end of the read operation, readers perform a verification step on the private version snapshot against the latest version of the key in the lock table. The operation passes the check if

```
1  LOOKUP(key):
2  restart:
3  v_guard = lock_table.load_version(key)
4  record = find key in hot tier
5  if record exists:
6      if record.ref == false:
7          set record.ref to true to retain
       hotness
8      if lock_table.validate_version(key,
       v_guard) == false:
9          goto restart
10     if record.deleted: return null
11     else return record.payload
12 # record does not exist in the hot tier
13 status, record = UPMIGRATION(key, OP::LOOKUP,
       null)
14 if status == RETRY or lock_table.
       validate_version(key, v_guard.v) == false:
15     goto restart
16 if status == NON_EXISTENT: return null
17 else: return record.payload
18
```

Fig. 7 Lookup Procedure.

they match. If the versions do not match, it indicates that at least one write operation initiated since the reader read the version. In this case, the reader restarts the read operation.

5.2 Basic Operations

We now describe the basic operations supported by TIERED-INDEXING and BILSM- TREE designs using optimistic locking.

Lookup Operation. All proposed designs support keyed lookup operations. As shown in Figure 7, the procedure begins by obtaining a version snapshot from the optimistic lock table. It then searches for the key in the hot tier. For 2- HEAP, the search involves finding the record stored in the heap using the index. If the record is found in the hot tier, we set the reference bit of the record to retain hotness and return the value (Line 7 of Figure 7). Before returning the result to the user, a validation step (Line 8) is performed to check if the version changed. If so, it indicates that there is a conflicting operation on the record, and the operation is retried. If the key does not exist in the hot tier, the search continues in the cold tier. If found, the upward migration (Figure 9) algorithm is performed, which probabilistically moves the record to the hot tier. Note that, both lookup and write operations may trigger upward migration. Therefore, before the actual migration is performed, the write lock must be obtained (Line 8-9 of Figure 9). The lookup algorithm for BILSM- TREE is very similar, as shown in Figure 8. The difference is that the migration sampling rate is a function (P) of the depth of the read and migrated records are copied over to the memtable. Note that optimistic locking also works for the migration in BILSM- TREE.

```
1  LOOKUP_bilsmt(key):
2  restart:
3  v_guard = lock_table.load_version(key)
4  record, depth, hit_or_miss = find key in lsm-
       tree.
5  if migration_enabled and random(0, 1) <= P(
       depth, hit_or_miss):
6      if v_guard.upgrade_to_write_lock() ==
       false:
7          goto restart
8      migrate record into memtable
9  if lock_table.validate_version(k, v_guard.v)
       == false:
10     goto restart
11 return record.payload
12
```

Fig. 8 Lookup Procedure for BILSM- TREE.

```
1  UPMIGRATION(key, op, new_value, v_guard):
2  record = find key in cold tier
3  if record exists:
4      if op == OP::UPDATE:
5          record.payload = new_value
6          update record in cold tier
7      if random(0, 1) <= D:  # Sampling-based
       Upward Migration
8          if v_guard.upgrade_to_write_lock() ==
       false:
9              return (RETRY, null)
10         migrate record into hot tier
11         if exclusive policy:
12             delete record from cold tier
13     return (SUCCEEDED, record)
14 return (NON_EXISTENT, null)
15
```

Fig. 9 Upward Migration Procedure.

```
1  UPDATE(key, new_value):
2  v_guard = lock_table.write_lock(key) #
       v_guard releases lock when it goes out of
       scope
3  record = find key in hot tier for write
4  if record exists:
5      if record.deleted: return false
6      update record with reference = true,
       dirty = true, payload = new_value in hot
       tier
7      return true
8  status, record = UPMIGRATION(key, OP::UPDATE,
       new_value)
9  if status == NON_EXISTENT: return false
10 else: return true
11
```

Fig. 10 Update Procedure.

Update Operation. For an update operation, shown in Figure 10, it first obtains a write lock associated with the key. Then a search operation is performed on the hot tier. If the key is found, we update the reference bit, dirty bit, and the record. If the key is not found in the hot tier, the update operation continues to the cold tier and executes a similar upward migration.

Insert Operation An insertion operation first checks the existence of a keyed record by using the Lookup operation. This is useful for implementing the primary key in relational databases. If the record does not exist, it inserts a record into the hot tier. Similarly, a write lock must be obtained before the insertion.

Put Operation. A put operation performs an insert that does not check the existence of the record in both tiers. This is useful for loading data or blind updates. Put operations insert records first in the hot tier with the dirty bit set. They are migrated down to the cold tier via eviction. Similarly, the write lock for the record must be obtained before proceeding.

Delete Operation. A delete operation first searches the hot tier. If the record is found, its deleted bit is set. If the record is not found, we insert a placeholder record with an empty payload and the deleted bit set. The delete is applied only to the cold tier when the record is evicted. Note that we do not update the reference bit so that the record will be evicted with a higher probability. This is a blind deletion, as we do not check record existence in the cold tier.

Range Scan Operation. For data structures that support range scan operations, such as clustered B+trees or B+tree-indexed heaps, the scan operation can be implemented by merging the results of scans on individual tiers. However, complications arise due to concurrent migrations and writes. During the scan, both the records and their version numbers are recorded. After merging the individual scan results, each record is validated by checking its version number in the lock table. If a record's version has changed or it was write-locked during the scan, a lookup is initiated to fetch the latest version. To prevent potential repeated lookups in cases where updates might continuously trigger version mismatches, the system employs a hybrid locking strategy: when a record fails the version check or is found to be write-locked, a write lock is obtained for that record. In the common case where there are no concurrent migrations or writes in the scanned range, this approach incurs minimal overhead and does not block concurrent point reads. When concurrent writes or migrations occur, the system falls back to acquiring write locks for the affected keys. Although in the worst case this could result in obtaining write locks for all records in the range, such scenarios are expected to be rare. Further optimizations, such as next-key locking, can also be employed. When merging results, two cases are handled. **Case #1:** a record exists in only one tree, in which case the system returns that record. **Case #2:** the record appears in both trees, in which case the version in the top tree is returned as it reflects the latest state. Note that for BILSM- TREE, which can be implemented on top of RocksDB, we leverage the fact that most of LSM-tree implementations has a built-in MVCC mechanism where a range scan is performed on a point-in-time snapshot of the underlying database. Hence, we do not need to do anything for BILSM- TREE for the range scan operation.

```
EVICTION():
victims = {}
for next record in hot tier considered for eviction:
    if record.ref == true:
        toggle the reference bit of the record
    else:
        victims.add(record, lock_table.load_version(record.key))
        if victims set is large enough:
            break
for (record, v_guard) in victims:
    if v_guard.upgrade_to_write_lock() == false:
        continue
    if record.deleted:
        delete record from cold tier
    elif record.dirty: # write back the latest copy
        upsert record into cold tier
    else: # not dirty
        if exclusive policy:
            insert record into cold tier
    delete record from hot tier
    v_guard.release_write_lock()
```

Fig. 11 Eviction Procedure.

Eviction. We trigger eviction by periodically checking if the size of the hot tier has exceeded a threshold. As shown in Figure 11, the eviction process starts by scanning the hot tier. Note that to ensure consistency, the write-lock of each evicted record must be obtained before the migration is really performed. The process iterates through every record using scan strategies discussed in Section 3. If the reference bit of a record is set, it is toggled off. Otherwise, it is added to the eviction set. Note that in addition to adding the record to the eviction set, we also keep a snapshot of the associated version number in the eviction set. Later we will use it to validate whether the record has been updated or migrated due to concurrent migration and writes. We end the scan operation after the eviction set has collected the desired number of records. For each record in the eviction set, we will first try to acquire its write lock by upgrading from the read lock (version) acquired from the scan step. The upgrade succeeds if the version does not change and an atomic compare-and-swap operation successfully bumps the version number in the lock table. Otherwise, it indicates that there were concurrent write operations that modified the record in the hot tier, and the record in the eviction set is potentially stale. In this case, we skip the record for downward eviction for correctness. After the write lock is acquired, we first apply deletions to the cold tier if the record is delete-marked. Then, all dirty records are upserted into the cold tier. For non-dirty records, they get inserted into the cold tier if the system is configured

with an exclusive migration policy. Lastly, we erase records in the eviction set from the hot tier.

Optimization for Clock-Replacement. Since TIERED- INDEXING designs rely on the clock-replacement algorithm for eviction, the reference bit of records in the hot tier needs to be constantly updated, even for read operations. With logging enabled, updating the reference bit naively would require writing log records. Luckily, the hot tier is completely memory-resident, and the reference bit is transient and does not need persistence for the correctness of the operations supported. Therefore, as an optimization, we can avoid logging when updating the reference bit.

6 IMPLEMENTATION DETAILS

We next describe the implementation details for TIERED-INDEXING. The implementation is opened source [1].

6.1 Buffer Pool

We use the LeanStore [27] buffer pool to implement all proposed 2- TIER designs. We partition the LeanStore buffer pool so that the hot region takes up 90% of the buffer frames and the cold region takes up 10%. This increases memory utilization for the majority part of the buffer pool while also allowing caching most of the index pages of the cold tiers. Note that this partitioning is not optimal for every workload, and we leave tuning to future work.

6.2 Optimistic Locking

We implemented the Optimistic Locking approach described in Section 5.1 using an array of 64K 64-bit version numbers. Records are mapped to a slot by their hashed keys. We use atomic compare-and-swap instructions for the write-locking. Note that this approach could cause conflicts in that different records might be mapped to the same slot. Other implementations, such as hash tables, are entirely possible to reduce such conflicts.

6.3 Data Structures

2B+tree. We use two LeanStore B+trees [27] backed by the buffer pool with the scan-based record migration [56]. The eviction process is triggered when the hot tier consumes more than the size of the hot region of the buffer pool. Our eviction is also run concurrently by multiple workers to maximize storage bandwidth. Each worker scans small key ranges (a few leaf nodes) that are disjoint from each other. Within each key range, the procedure described in Figure 11 is implemented.

2Hash. We compose two linear hash tables on top of the LeanStore with record migration. The eviction process is activated when the hot hash table consumes more than the size of the hot region of the buffer pool. Our eviction is run concurrently by multiple workers to maximize storage bandwidth. Each worker scans a few hash buckets disjoint from each other using procedures described in Section 3.2 and Figure 11. We implemented 2Hash with a shared hash function (2Hash-S) and 2Hash with independent hashing (2Hash-D).

2Heap. We compose two heaps built on top of the LeanStore with record migration. Each heap is equipped with its own indexes (B-trees). The eviction process is activated when the hot heap consumes more than the size of the hot region of the buffer pool. Our eviction is also run concurrently by multiple workers to maximize storage bandwidth. For heap scan eviction strategy, each worker scans pages disjoint from each other. For indexed heap scan, each worker scans key ranges (a few leaf nodes) of the B+tree disjoint from each other. Indexes are maintained during migration. We implemented both 2Heap with heap scan (2Heap-HS) and 2Heap with indexes heap scan (2Heap-IHS).

BILSM-TREE. We built BILSM- TREE on top of RocksDB by adding upward data migration on point read. We added callbacks to flush and read path of memtable in RocksDB to implement migration path ② described in Figure 4. BILSM- TREE monitors block cache miss rate, and average depth of point read every second. We set the lower (TH_L) and upper (TH_H) slope thresholds of adaptive migration control to be 0.01 and 0.1. We maintain a window of 100 observations. We set base sampling rates B_{hit} and B_{miss} to be 0.01 and 0.1. We set the growth factor R to 1.5. We cap the sampling rate to 0.3. Therefore the sampling rate is $min(B_{hit} \cdot R^D, 0.3)$ or $min(B_{miss} \cdot R^D, 0.3)$. We do not perform migration for point reads served by the memtable.

7 EXPERIMENTAL EVALUATION

In this section, we conduct extensive experiments to answer the following questions:

- How do TIERED- INDEXING compare to 1- TIER and record-caching designs on data loading and skewed workloads?
- How do proposed designs react to distribution shifts?
- What is the impact of different degrees of skew and record size?
- How scalable is our proposed optimistic locking compared to the pessimistic one and other baselines?

[1] https://github.com/zxjcarrot/2-Tree/tree/tieredindex

7.1 Baselines

We compare our proposals against the following systems:

1-TIER Designs. We use the B+tree from LeanStore. We implemented a linear hashing scheme with utilization-controlled split on LeanStore. Records that cannot be kept on the primary bucket pages are kept in overflow pages. We implemented an indexed heap on top of LeanStore. The heap file consists of data pages managed by a page directory that contains free space information and the location of each page. By default, we build a B+tree index on top of the heap file. The B+tree indexes the tuples in the heap file on the primary key. Each index entry stores the page ID and slot ID of the indexed tuple within the page.

TreeLine. TreeLine is a key-value store optimized for modern storage [27], which employs an in-memory record cache on top of a page-based B+tree-like structure on disk to optimize for skew. TreeLine dedicates all memory to the record cache. We use Treeline at commit 6a7c35fb for evaluation. The record cache is based on Masstree [36] that provides efficient range scan.

1-TIER Designs with Record Cache. We also implemented a in-memory record cache for both linear hash table and indexed heap. For linear hash table, the record cache uses a hash index with LRU eviction policy. To improve concurrency, we shard the hash index into 24 shards each protected by a lock. To reduce the CPU overhead of maintaining the LRU chain, we sample 10% of the accesses to the hash index to trigger the maintenance of the chain [12]. For indexed heap, the record cache uses an in-memory B+tree implementation [2] based on optimistic lock coupling and epoch-based reclamation. Both caches use the same LRU eviction policy as the linear hash table. To make the cache more versatile, we cache writes (insert/delete/update) in the record cache as well. Both caches adopt an eager migration where each cache miss results in immediate record migration from the on-disk structure. Similarly, we allocate 90% of the memory budget to record cache and leave 10% for buffer pool.

LSM-tree. We use RocksDB [44] v7.11 as the baseline LSM-tree implementation, which uses a block cache (read-only buffer pool) to cache frequently accessed data blocks. We configure 256 MB memory budget for the memtable and leave the rest of the budget to block cache.

LSM-tree with Record Cache (LSM-tree RC) . We enable the row cache feature in RocksDB that caches frequently accessed physical records in SST files. We configure the row cache to use 90% of the memory budget and leave 10% to block cache.

[2] https://github.com/wangziqi2016/index-microbench/tree/master/BTreeOLC

We first examine the loading performance. We measure the average throughput of inserting 100 million records randomly drawn from the key space. The overall results are shown in Figure 12 and Table 1.

7.2 Experimental Setup

All experiments are conducted on a server with an Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz that has 20 cores and 256 GB DRAM. The server is installed with Linux kernel 5.19.12 as the operating system. We use a 1 TB NVMe SSD (INTEL SSDPE2KX010T8) with a read latency of around 110 us for all experiments. To focus solely on buffer pool memory utilization, all systems use direct IO and disable write-ahead logging and the Linux page cache for all experiments.

Workload. We use the Yahoo! Cloud Serving Benchmark (YCSB) as our main workload. By default, we use a data set with 100 million records, each with an 8-byte key and 120 random bytes of payload. The data set is 12.8 GiB in logical size. We evaluated our designs mainly with Zipfian [20] access distribution, typically seen in real-world workloads. By default, we use a Zipfian factor of 0.9 where 80% of the accesses land on ∼10% of the keys in the dataset.

We configure all systems with 16 KB page size. For all experiments, we first load the data and then warm up the systems by running the target workload until the system throughput stabilizes, and finally measure and report the average throughput. By default, 2- TIER designs are configured with an inclusive caching policy and lazy upward migration with a sampling rate F of 0.3, which we found performs reasonably well. All experiments were run with 20 worker threads that perform normal operations and migration.

7.3 Data Loading Performance

We compared vanilla B+tree and TreeLine against 2B+tree. The results are shown in Figure 12. This is a very challenging workload for traditional buffer-managed B+tree. At a small memory budget (2.5 GB and 5 GB), 2B+tree outperformed B+tree/TreeLine by up to 9.8×/12.3× in terms of throughput. This can be explained by the amount of I/O amplification per insert. As shown in Table 1, 2B+tree incurs nearly 29.5×/52.5× less amplification per insert. This is because the eviction of 2B+tree produces a nice semi-sequential I/O pattern and results in only a 0.04 amplification factor on average. TreeLine has the worst loading performance as all memory is dedicated to its record cache, and its eviction algorithm produces a random I/O pattern. Hence, every evicted record requires at least one read and one write I/O request. This emphasizes the importance of a good eviction algorithm. For B+tree, there is a 58% chance that the page it needs is not

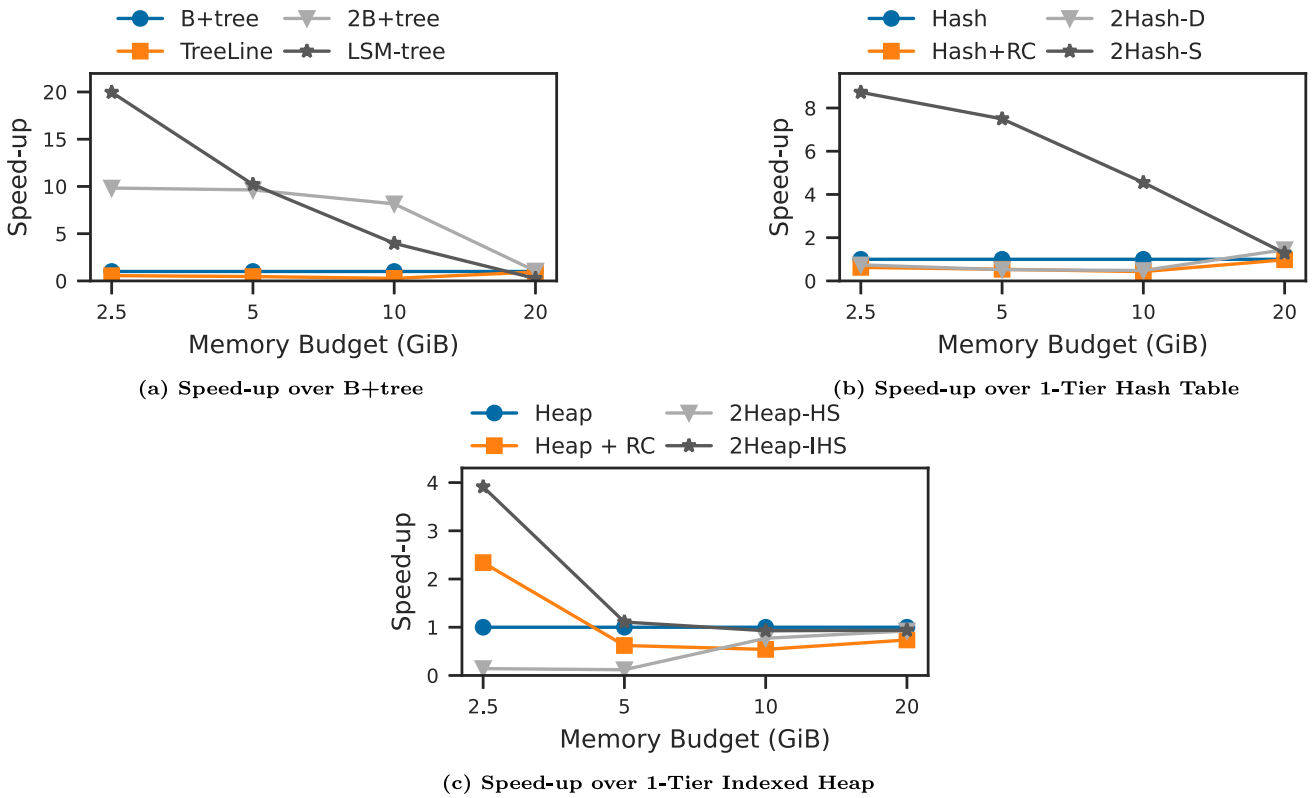


Fig. 12 Relative speed-up of loading 100 million records over 1- TIER designs.

Table 1 Average I/O amplification factor per insert. The factor is calculated as the number of bytes read/written from/to storage (each I/O is 16KB) divided by the logical payload of 128 bytes

System / Memory	2.5 GiB	5 GiB	10 GiB	20 GiB
LSM-tree	5.4	5.4	5.4	5.4
B+tree	151.04	93.44	29.44	0
TreeLine	268.80	248.32	143.36	0
2B+tree	5.12	2.56	1.28	0
Hash	272.64	174.08	58.88	0
Hash + RC	336.5	254.7	119	0
2Hash-D	352.00	262.40	136.96	0
2Hash-S	7.68	3.84	2.56	0
Heap	2.56	1.28	0	0
Heap + RC	1.31	0.68	0.1	0
2Heap-HS	98.56	24.32	0.51	0
2Heap-IHS	1.28	1.28	0	0

in the buffer pool. As we increase the memory budget, the throughput of all systems converges as more pages can be cached.

For reference, in Figure 12a, we also included the throughput of RocksDB, an LSM-tree implementation that is well-known to handle such workloads very well. LSM-tree throughput is constant with respect to the buffer pool capacity as its memtable is designed to be constant in size (256 MB). 2B+tree performs similarly to an LSM-tree using a 5 GiB buffer pool. Another interesting observation is that buffer-managed B+trees can improve the loading throughput with a bigger buffer pool as more data fits in memory. When all data fits in memory, the loading performance of all options outperformed LSM-tree significantly as there is no eviction I/O. However, 2B+tree requires a much smaller buffer pool thanks to its more I/O efficient eviction strategy.

For hash tables, we compared 1- TIER linear hashing with or without record cache against 2- HASH with different hash function choices. The results are shown in Figure 12b. Interestingly, record cache hurts performance compared to a single linear hashing. This is due to the fact that the LRU eviction policy generates a random I/O pattern (see Table 1). This demonstrates the importance of a good eviction algorithm for record-level caching. On the other hand, at a small memory budget (2.5 GB and 5 GB), 2- HASH with a shared hash function (2Hash-S) outperforms 1- TIER hash table and 2- HASH with independent hash functions (2Hash-D) significantly. Looking at the I/O stats in Table 1, 2Hash-S incurs up to 45× less I/O per write compared to the other two designs. 2-Hash-S has a much better I/O pattern during eviction. Similarly, as the buffer pool capacities increase, the throughput of all designs converges as most pages are cached in the buffer pool.

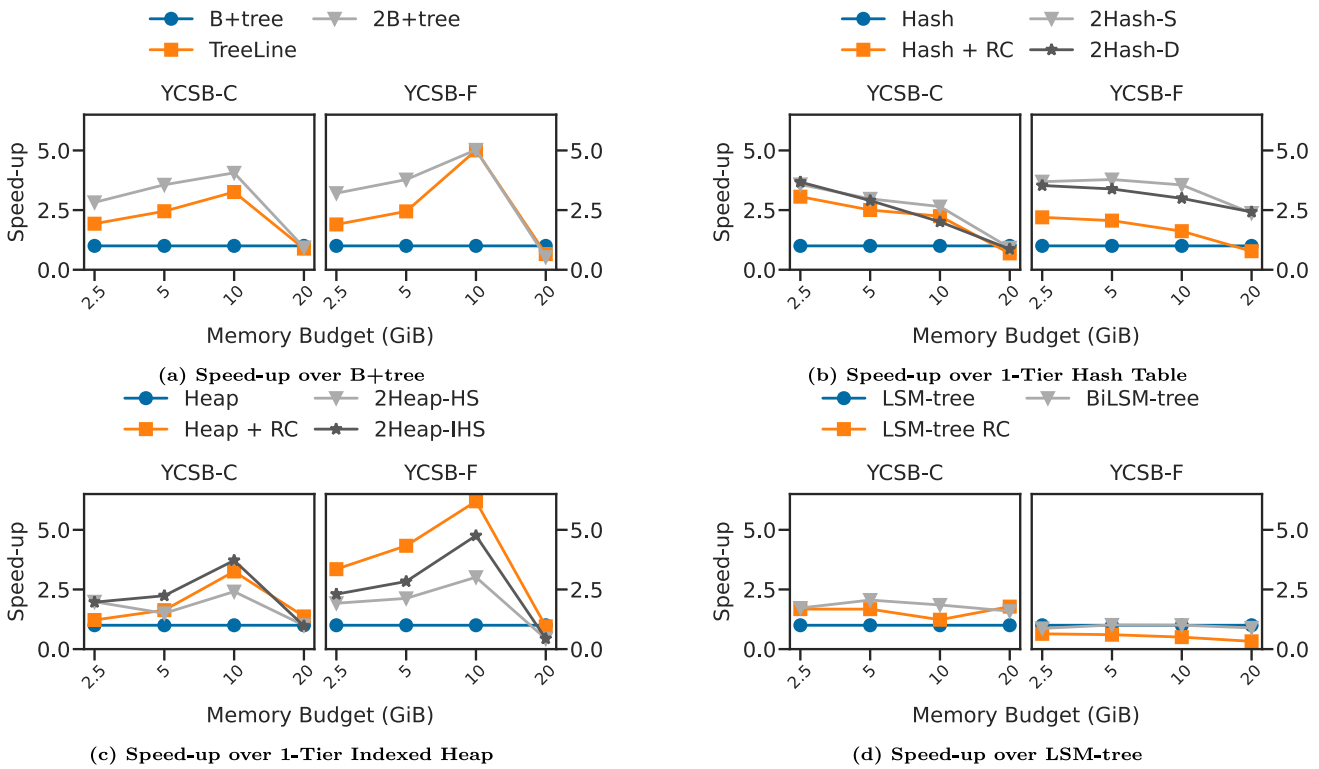


Fig. 13 Relative throughput improvement over 1- TIER designs on YCSB-C(100% Reads) and YCSB-F(50% Reads, 50% Updates)

For heap files, we compared the indexed heap with or without record cache against 2- HEAP with different eviction strategies. The results are shown in Figure 12c. Generally, 1- TIER heap performs the best as records inserted are appended to the heap files, resulting in sequential I/O patterns and requiring little buffer frames. This means that most of the buffer frames can be used to cache pages of the index. 2- HEAP with indexed heap scan (2Heap-IHS) performs similarly to a single heap. It outperforms 2- HEAP with heap scan (2Heap-HS) significantly at small memory. This is mainly because the maintenance of the index on the cold heap causes much more random I/Os for 2Heap-HS, shown in Table 1, due to the fact that only 10% of the buffer pool can be used for caching pages of the index. For 2- HEAP with indexed heap scan, the maintenance of the cold index has a more sequential I/O pattern which means fewer buffer frames are required than a single indexed heap.

We omit the results for variants of LSM-tree for brevity as they show similar results on data loading.

To summarize, TIERED- INDEXING designs with efficient record-level eviction algorithm outperform 1- TIER and record-caching designs significantly on data loading for B+tree and hash tables.

7.4 Steady-State Performance

We next look at the design performance of YCSB-C and YBCS-A workloads, focusing on memory utilization of the buffer pool after the system stabilizes after data loading. Note that YCSB-C consists of 100% point read operations, and YCSB-F consists of 50% read-modify-write operations and 50% point read operations. To quantify memory utilization, we mainly use the average number of I/O issued per operation in the workloads. The results are shown in Figure 13 and Table 2.

7.4.1 2-TIER Designs

For B+tree, shown in Figure 13a, we observe that 2B+tree can outperform a single B+tree by 3.7/4.5× with a 10 GB buffer pool on read-only and update workloads. 2B+tree issues up to 5.4× fewer I/O operations compared to B+tree. 2B+tree is also more efficient than TreeLine. The reasons are twofold. First, TreeLine's record cache has 56 bytes per record space overhead, while 2B+tree only has 1 byte overhead in the hot tier, resulting in more records cached in 2B+tree. Second, the eviction procedure of 2B+tree is more efficient than TreeLine's and issues up to 60% fewer I/O operations for the YCSB-F workload. At 20 GB memory, all systems perform similarly as the entire workload fits in memory. Compared to B+tree and 2B+tree, TreeLine per-

Table 2 Average number of I/O requests per operation in steady state (YCSB-C/YCSB-F).

System / Memory	2.5 GiB	5 GiB	10 GiB	20 GiB
LSM-tree	0.46/0.30	0.32/0.19	0.1/0.08	0/0.02
LSM-tree RC	0.27/0.52	0.19/0.38	0.09/0.25	0/0.18
BiLSM-tree	0.26/0.32	0.15/0.18	0.05/0.09	0/0.03
B+tree	0.54/0.84	0.43/0.68	0.22/0.38	0/0
TreeLine	0.26/0.47	0.17/0.29	0.06/0.07	0/0
2B+tree	0.20/0.30	0.13/0.18	0.05/0.07	0/0
Hash	0.69/0.98	0.49/0.74	0.20/0.34	0/0
Hash+RC	0.38/0.72	0.24/0.49	0.09/0.2	0/0
2Hash-D	0.32/0.44	0.20/0.29	0.09/0.12	0/0
2Hash-S	0.32/0.42	0.20/0.26	0.07/0.10	0/0
Heap	0.8/1.1	0.53/0.29	0.20/0.34	0/0
Heap+RC	0.58/0.38	0.27/0.21	0.06/0.06	0/0
2Heap-HS	0.41/0.58	0.3/0.39	0.08/0.11	0/0
2Heap-IHS	0.41/0.50	0.23/0.29	0.05/0.07	0/0

forms worse on a scan-heavy workload by up to two orders of magnitude, as shown in Figure 16. 2B+tree is outperformed by B+tree by up to 50% due to its merging costs. However, 2+B-tree are able to leverage increased memory. On the other hand, TreeLine cannot leverage the increased memory budget to serve scan queries as it dedicates all memory to the record cache, and there is no page caching to exploit the skew.

For hash tables, the results are shown in Figure 13b. We observe that 2- HASH with a shared hash function (2Hash-S) performs the best. It outperforms the single hash table by up to 3.0/3.7× with a 5 GB buffer pool for YCSB-C and YCSB-F workloads. The two 2- HASH designs both outperform single hash with up to 2.45× less I/O operation. 2Hash-S requires slightly fewer I/O operations than 2Hash-D due to its eviction having a better I/O pattern. With 20 GB memory, all three designs converge on performance. Compared to hash table with record cache, 2Hash-S generally performs better on both YCSB-C and YCSB-F workloads. This is due to 2B+tree having better memory utilization and more I/O efficient evictions, as demonstrated by the I/O statistics in Table 2.

Results for indexed heap files are shown in Figure 13c. We observe that 2- HEAP with an indexed scan (2Heap-IHS) performs the best. It outperforms single indexed heap significantly by up to 3.4/4.6× with a 10 GB buffer pool on read and update workloads. 2Heap-IHS issues 3.4× fewer read I/O operations per operation compared to a single indexed heap, as shown in Table 2. 2Heap-IHS also performs fewer I/O operations than 2- HEAP with a heap scan (2Heap-HS). This is mainly because the former performs 53 × less read I/O operations during the eviction procedure. Heap with record

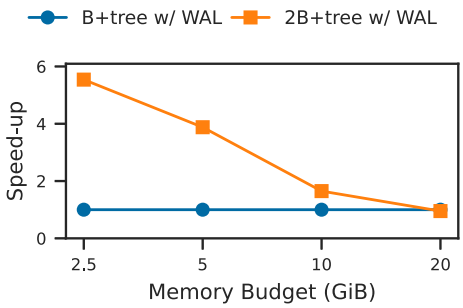


Fig. 14 Relative speed-up of loading 100 million records over B+tree with logging enabled.

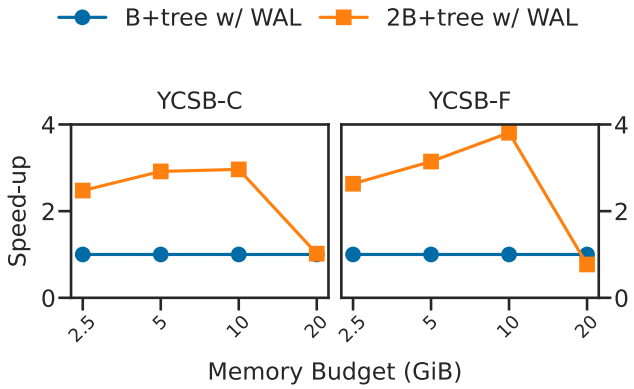


Fig. 15 Relative throughput improvement over B+tree with logging enabled.

cache outperforms 2Heap-IHS in YCSB-F. This is mainly due to that 2Heap maintains two indexes, causing more pages in total and slightly higher I/O cost compared to heap with a record cache. All three designs converge on performance when the buffer pool is 20 GB in size, as all data can be cached in the buffer pool.

In addition, the performance gap between TIERED-INDEXING and 1- TIER designs is generally wider on update (YCSB-F) workload across B+tree, hash table, and heap files. This is because such workloads further penalize the design with lower memory utilization. An update to a disk page can necessitate two I/O operations (one for read and one for eviction).

7.4.2 BILSM-TREE

For LSM-tree, shown in Figure 13d, BILSM- TREE outperformed LSM-tree by up to 2.05× on read-heavy (YCSB-C) workloads. BILSM- TREE performs similarly compared to LSM-tree with record caching on read-only workloads as they both have increased memory utilization. However, for update-heavy workloads (YCSB-F), LSM-tree with record cache is consistently worse than LSM-tree and BILSM-TREE. This is because each update inserts a new record at the top of tree hiearchy and implicitly invalidates an old record in

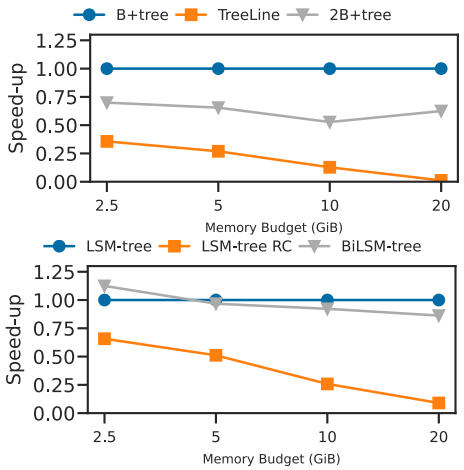


Fig. 16 Relative throughput improvement of Scan-Update (95% Scans, 5% Read-Modify-Write) workload over 1- TIER design

the record cache. Since RocksDB does not update the record cache during writes, it relies on later point reads to cache the record. Therefore, the record cache has a poor hit rate in this workload. Another problem with record caching in LSM-tree is its scan performance. As shown in Figure 16, we ran a scan-heavy workload on LSM-tree designs, varying the memory budget. LSM-tree with record caching is consistently worse than LSM-tree and BILSM- TREE by up to 10×. This is because record cache cannot serve range queries. Although one could dedicate more memory to block cache to improve range scan, the fundamental tension between record cache and block cache still exists.

In summary, both 2- TIER and BILSM- TREE designs are effective in improving the memory utilization of the buffer pool. Unlike LSM-tree with record cache, BILSM- TREE does not sacrifice the performance of range scan and update-heavy workloads.

7.5 Impact of Logging

We next look at the performance impact of enabling write-ahead logging which is based on a parallel write-ahead-logging mechanism proposed in [22] for the buffer manager. Since both TreeLine [53] and 1- TIER designs with record cache lack a logging and recovery strategy for durable record cache, this experiment focuses on comparing 2B+tree against 1- TIER B+tree. We performed the same experiment as we did in previous sections using YCSB-C and YCSB-F. To reduce performance penalty from log persistence, the system implements group commit and writes log records to SSD without issuing a `fsync` at commit.

The results are shown in Figure 14 and Figure 15. Compared to Figure 12a, we can see that the performance gap of data loading shrinks due to the logging overhead. However, the relative improvement is still significant. 2B+tree outperforms B+tree by up to 5.8× at low buffer pool capacity. For steady-state performance shown in Figure 15, 2B+tree outperforms B+tree by up to 2.96× and 3.8× for YCSB-C and YCSB-F workloads. Compared to Figure 13a, the performance improvement is smaller due to the logging overhead.

7.6 Ablation Study of BILSM-TREE

To understand the impact of techniques introduced in BILSM- TREE, we performed an ablation study that incrementally enables these techniques. We ran YCSB-C workload for 3000 seconds with a 5 GB memory budget and measured the average number of bytes written caused by migration per point read operation. The result is shown in Table 3. We start from vanilla LSM-tree, which has zero write overhead with a throughput of 190K/s. By adding a naive migration that migrates every record read from the LSM-tree, it achieves a worse throughput of 185K/s and 67-byte write overhead. When a constant sampling rate of 0.3 is applied to the migration, the throughput jumps to 297K/s with 3.2× less write overhead compared to naive migration. In contrast, exponential sampling in BILSM- TREE reduces the write overhead by 8.4×. Although early migration does not reduce write overhead much, it improves throughput by reducing the impact of flushes. Finally, adaptive migration further reduces write overhead by 2.1× and delivers the highest throughput as it disables migration and reduces performance fluctuation caused by compaction. Note that if the workload is run longer, the write overhead will be further amortized.

Table 3 Ablation Study of BILSM- TREE on YCSB-C.

Baselines	Average # Bytes Written/Lookup	Throughput
LSM-tree	0	190K/s
+ Naive Migration	67	185K/s
+ Const. Sampling [56]	21	297K/s
+ Exp. Sampling	8	308K/s
+ Early Migration	7.9	353K/s
+ Adaptive Migration	3.8	376K/s

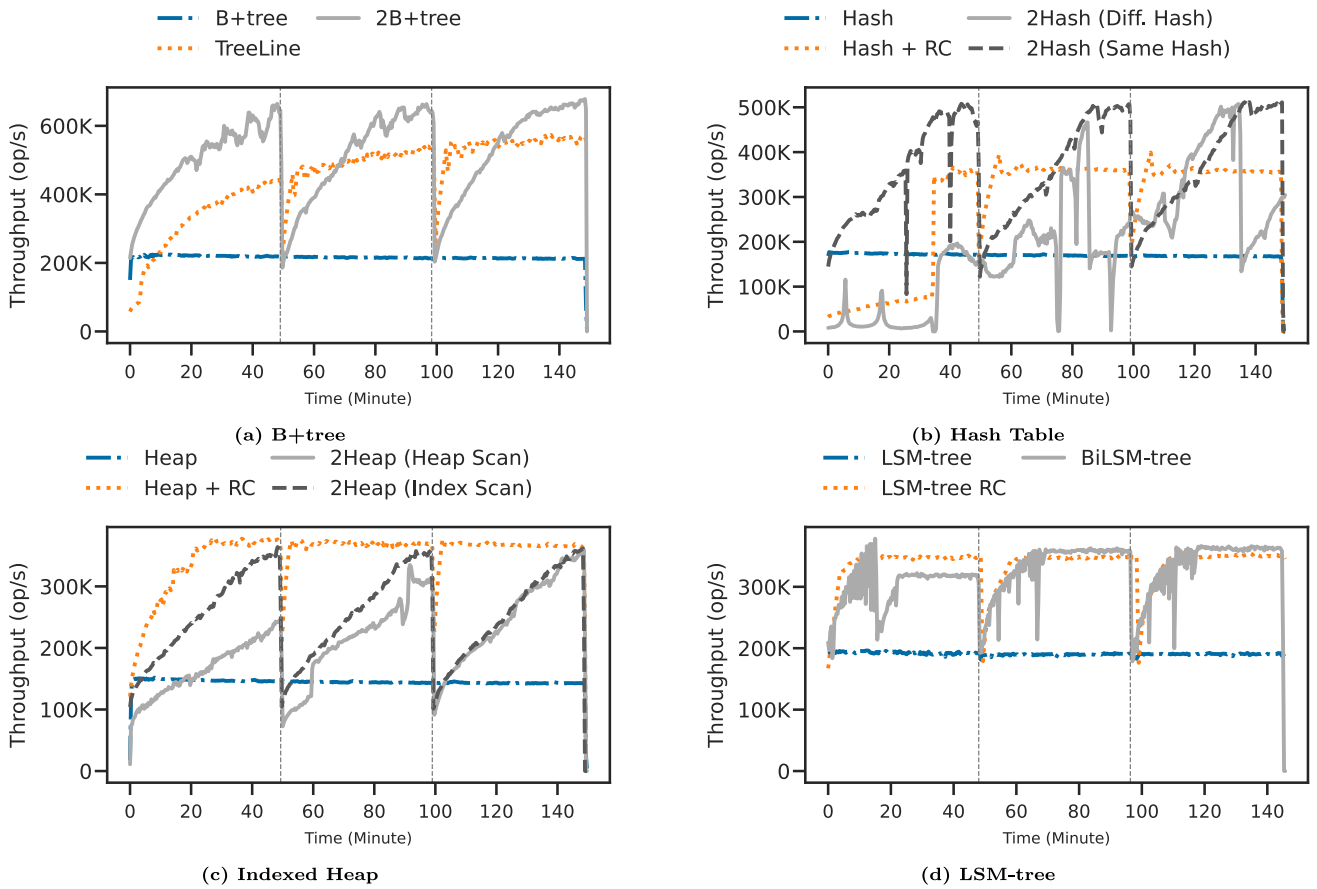


Fig. 17 Performance of all systems on YCSB-C workload over 9000 seconds with shifting hot spots. Hot spots are changed every 3000 seconds. Each data point is an average throughput over a 20-second period.

7.7 Impact of Distribution Shift

We next study the impact of shifting hot spots on all the systems. This experiment intends to measure how fast each system stabilizes on performance after loading and reacts to distribution shifts. We start from a freshly loaded database with a 5 GB memory budget where no upward migration. We then run YCSB-C workload for 9000 seconds. After every 3000 seconds, we randomly permute the keys in the key space to simulate distribution shifts. We measure the average throughput over time. The results are shown in Figure 17.

For B+tree variants, shown in Figure 17a, a single B+tree stabilizes the fastest because it only has page migration. 2B+tree starts off with a similar throughput as B+tree and gradually improves its throughput as the upward migration identifies and migrates hot records. It reaches stability after about 33 minutes. TreeLine starts off worse than B+tree as it is still evicting dirty records, and the eviction is inefficient. TreeLine surpassed B+tree after 20 minutes. Both 2B+tree and TreeLine experience significant performance drops when the distribution changes. This is because 2B+tree/TreeLine replaces the working set one record at a time, and B+tree replaces the working set at the page level, which is much faster.

For hash table variants, shown in Figure 17b, we observe that a single hash table (Hash) has very stable throughput throughout the experiment. We can see that 2- HASH with different hash functions initially performs worse than a single hash table by up to 80% for about 25 minutes. This is mainly because eviction incurs about 1.1 I/O operations per dirty record. Such eviction I/O consumes SSD bandwidth. In contrast, 2- HASH with the same hash function has a much smoother ramp-up curve as eviction issues only 0.1 I/O operations per dirty record on average due to the design of sharing hash function. Both 2- HASH designs perform similarly when the workload has been run for sufficiently long.

For indexed heap variants, shown in Figure 17c, a single heap has little throughput fluctuation during the experiment. 2- HEAP with heap scan initially performs much worse than Heap for about 20 minutes by up to 88%. This is because each eviction issues 0.83 I/O operations on average compared to 0.02 I/O operations on average for 2- HEAP with indexed scan, which reaches the highest throughput much faster. 2-

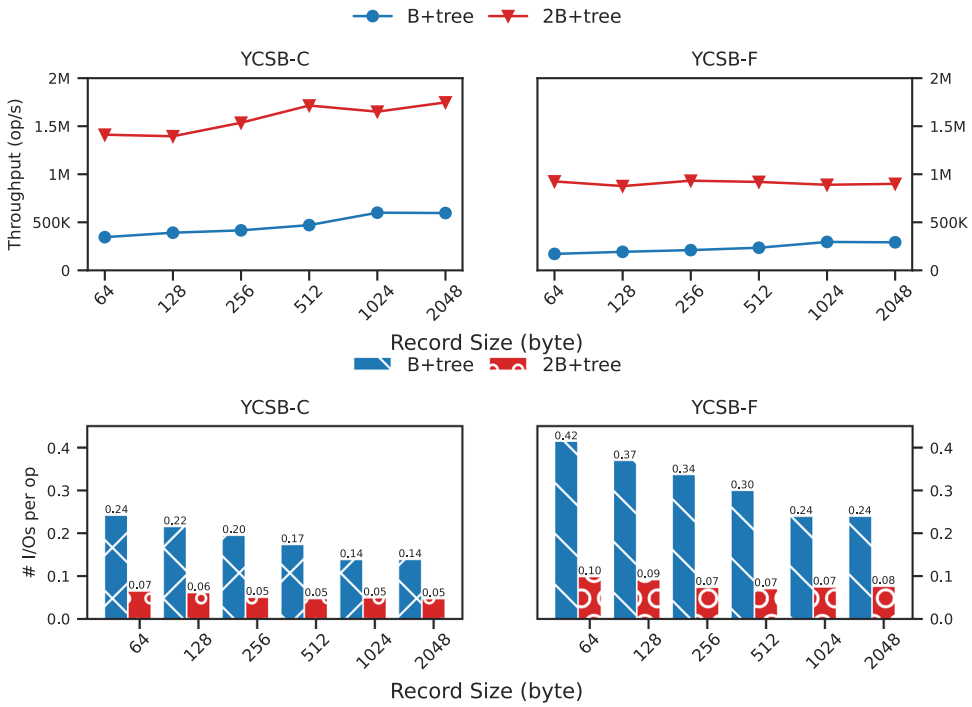


Fig. 18 Performance impact of record size.

HEAP with heap scan catches up throughput during the third phase when it has evicted all the dirty records.

For LSM-tree variants, shown in Figure 17d, LSM-tree has the most stable performance. BILSM- TREE and LSM-tree with record cache improve throughput over time. The migration of BILSM- TREE is turned off after about 20 minutes when BILSM- TREE detects the rate of improvement is small. At this point, BILSM- TREE outperforms LSM-tree by about 1.8× in throughput. At 50-minute, when the distribution shifts, BILSM- TREE detects a slope of the trend line larger than the threshold and restarts the migration procedure to optimize the placement of the records. Therefore, BILSM- TREE reliably detects distribution shifts and reacts quickly.

7.8 Impact of Record Size

Next, we study the impact of record size on the performance of our proposals. Memory utilization is expected to increase for single-structure designs as we increase the record size. This is because as the size gap between pages and records decreases, less memory are wasted. We focus on the clustered B+tree in this experiment as the results for other data structures are similar. We run the YCSB-C and YCSB-F workloads for both B+tree and 2B+tree with a skew factor of 0.9. We vary the record size while keeping the logical size of the data set the same (12.8 GB). We measure the steady-state throughput of B+tree and 2B+tree.

The results are shown in Figure 18. Generally, B+tree throughput increases as we increase the record size from 64 bytes to 2 kilobytes. This is also reflected in the I/O statistics shown in Figure 18. 2B+tree outperforms B+tree by 5× at the smallest record size (64 bytes). This is the point where the memory utilization for B+tree is lowest due to the largest size ratio (256) between page and record. As the record size increases, the gap between 2B+tree and B+tree shrinks. With 2 kilobyte records, 2B+tree still outperforms B+tree by 3×. And the average number of I/O reads is relatively stable for 2B+tree across varying record sizes. This is because our upward migration makes sure that records in the hot tiers are the hottest irrespective of the record size.

In summary, increased record sizes narrow the performance gap between B+tree and 2B+tree. However, 2B+tree is still outperforming B+tree at all record sizes.

7.9 Impact of Skew Level

Next, we study the impact of skew level. We run YCSB-C and YCSB-F workloads for all designs with a 5 GB memory, varying the Zipfian skew factor from 0.1(almost no skew) to 1.1 (highly skewed). For 2- HASH, we focus on the design with a shared hash function. For 2- HEAP, we focus on the design with an indexed heap scan eviction strategy. We chose these because they perform well on both loading and steady-state experiments. The results are shown in Figure 19.

From the results, we observe that under the lowest skew (0.1), both TIERED- INDEXING designs and 1- TIER designs perform similarly. The reasons are twofold. First, our lazy migration strategy significantly reduces the number of unnecessary migrations in the cold tier if the accessed record is

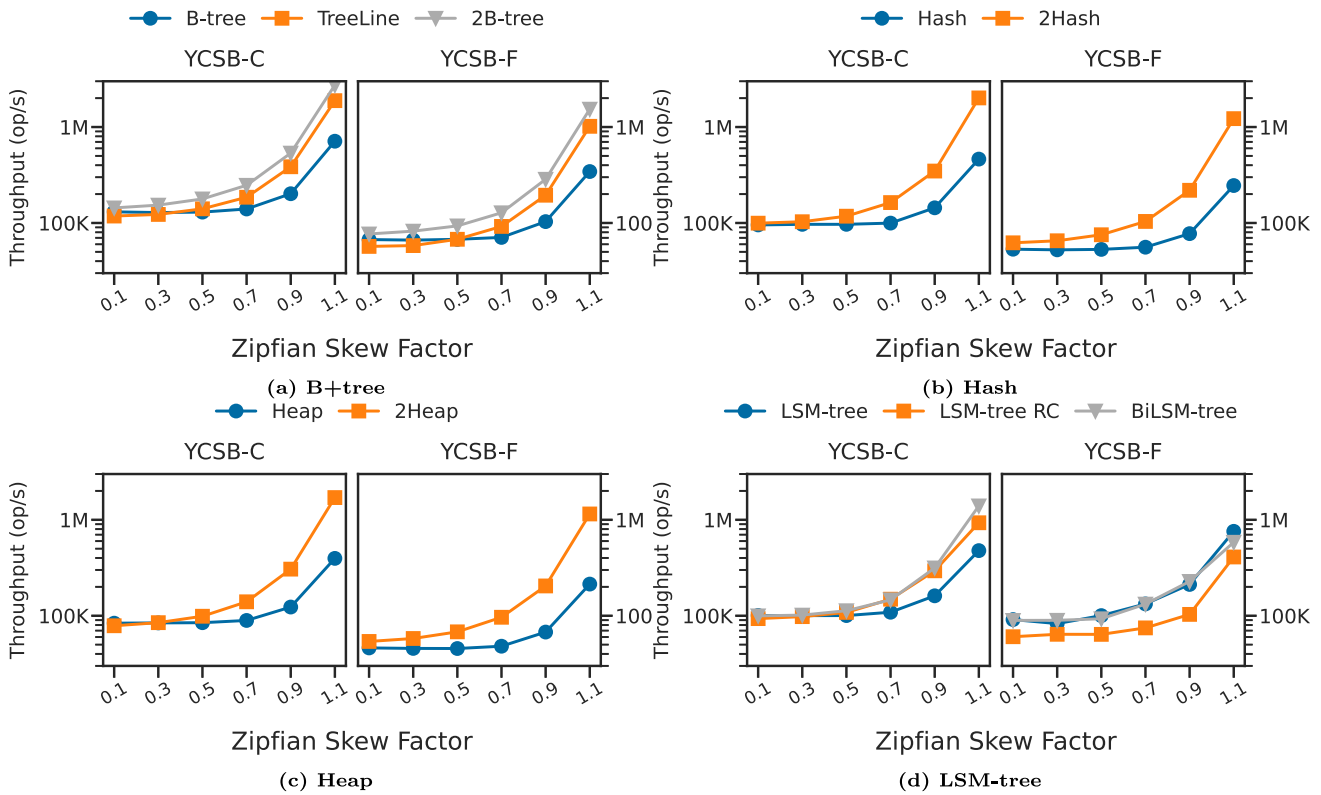


Fig. 19 Performance under varying skew.

cold. Second, our downward migration strategy is efficient in terms of I/O. For example, in 2B+tree, each evicted record takes only 0.04 I/Os on the cold tier on average. As the skew increases, TIERED- INDEXING designs perform much better. 2B+tree outperforms B+tree/TreeLine by up to 4.5/1.5× at skew factor 1.1 on YCSB-F workload. 2- HASH outperforms Hash by up to 4.34/4.98× at skew factor 1.1 on YCSB-C/YCSB-F workloads. All 2- TIER designs perform similarly compared to 1- TIER designs at low skews(0.1-0.3). We only observe significant performance improvements of the 1- TIER design at a high skew (0.9/1.1). BILSM- TREE outperforms LSM-tree YCSB-C by 2.9× on skew factor 1.1. BILSM-TREE even outperforms LSM-tree with record caching by 1.5× on skew factor 1.1. This is because record migration reduces the depth of the point reads by moving hot records to a shallower place in the tree hierarchy which is impossible for record caching.

These results show that 2- TIER and BILSM- TREE designs can outperform 1- TIER and record caching designs in most degrees of skew. 2- TIER designs also do not under-perform with low skew compared to the 1- TIER designs thanks to our efficient migration.

7.10 Multi-Core Scalability

We next study the scalability of our proposed optimistic locking. To isolate I/O latency, we focus on heavily skewed workloads where the working set fits in the main memory. We compare our optimistic approach with a pessimistic one as well as 1- TIER B+tree and TreeLine by running the same YCSB-F and YCSB-C workloads with a 30 GB memory budget. To stress the systems, we vary the skew factor from 0.9 to 0.96 and the number of threads from 1 to 32. With a skew factor of 0.96, 90% of the accesses go to 6.7% of the records in the dataset.

The results are shown in Figure 20. For read-only workloads (Figure 20a), the optimistic approach scales well across all the skew factors and the number of threads in the system. It matches the scalability of the 1- TIER B+tree in LeanStore which employs optimistic lock coupling. This scalability does not change across skew factors. Note that pessimistic locking degrades scalability as the skew factors increase. At 32 threads and a skew factor of 0.96, it is outperformed by the proposed optimistic approach by 1.78 ×. The pessimistic approach performs similarly to TreeLine which uses a per-record read-write mutex for concurrency control. For update-heavy workloads (Figure 20b), the performance of all systems drops significantly with higher skew and more threads. This is due to the inherent physical conflict at the

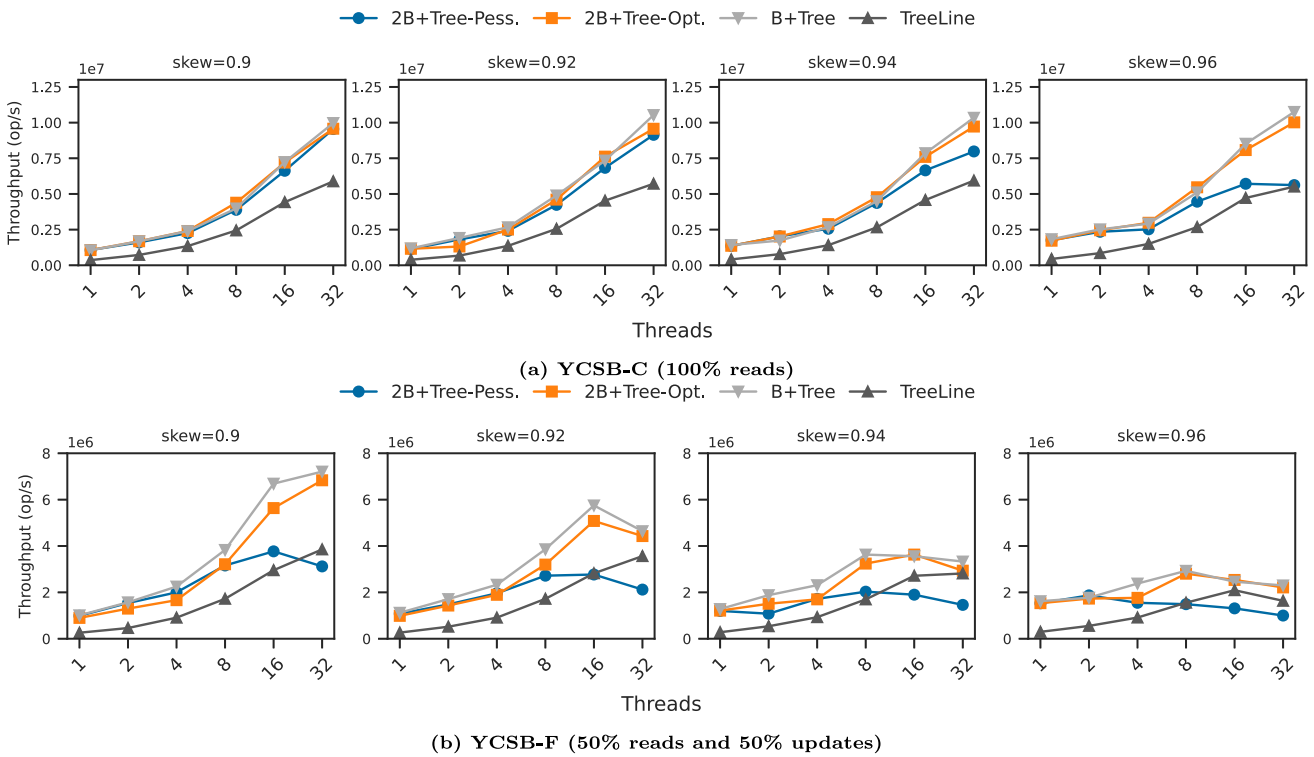


Fig. 20 Scalability of the proposed Optimistic Locking under heavy skews and concurrency.

record level. However, the optimistic approach still performs 2.2× than the pessimistic approach, as the YCSB-F contains reads.

To summarize, we demonstrated that the proposed optimistic approach is scalable for read-heavy workloads under high contention and skew. It performs similarly or better than the pessimistic approach when there are skewed writes in the workload. Optimistic approach has a scalability close to a page-based 1- TIER B+tree using optimistic lock coupling.

7.11 Inclusive versus Exclusive Strategy

Next, we study the impact of different caching strategies during upward migration. Specifically, we compare an inclusive strategy with an exclusive strategy. We run YCSB-C and YCSB-F workloads in all TIERED- INDEXING designs while varying the buffer pool capacity from 2.5 GB to 20 GB. For 2- HASH, we focus on the design with a shared hash function. For 2- HEAP, we focus on the design with an indexed heap scan eviction strategy. This is because they perform well on both loading and steady-state experiments. The results are shown in Figure 21.

One general trend we observe is that for read-only workloads, designs with inclusive strategy outperform designs with exclusive strategy when the buffer pool capacity is below 20 GB. Specifically, we observe up to 70%/58%/55% improvements with an inclusive strategy for 2B+tree/2Hash/2Heap. This is because, with an exclusive strategy, record migration would require modifying pages. For example, for upward migration, the system first needs to remove the record from the cold tier. Similarly, evicting records from the hot tier would require inserting records back into the cold tier, even if the records are not modified at all. These modifications generate writes to the SSD even if the workloads are read-only. Such writes consume the bandwidth of the SSD and lower system performance. For an inclusive strategy, upward migration does not require modification of the cold tier at the expense of data duplication. Similarly, to make space in the hot tier, records that are evicted and are not modified can be simply thrown away without modifying the cold tier. Therefore, an inclusive strategy is superior to an exclusive strategy. For update workloads, the performance gap between inclusive and exclusive strategies is much smaller. This is because the records evicted from the hot tiers are mostly dirty and would require writing back to the cold tier in an inclusive strategy. At 20 GB buffer pool capacity, we observe no significant difference between exclusive strategy and inclusive strategy as all data fits in memory.

8 CONCLUSION

In this paper, we proposed TIERED- INDEXING, a general approach for optimizing the memory utilization of a wide

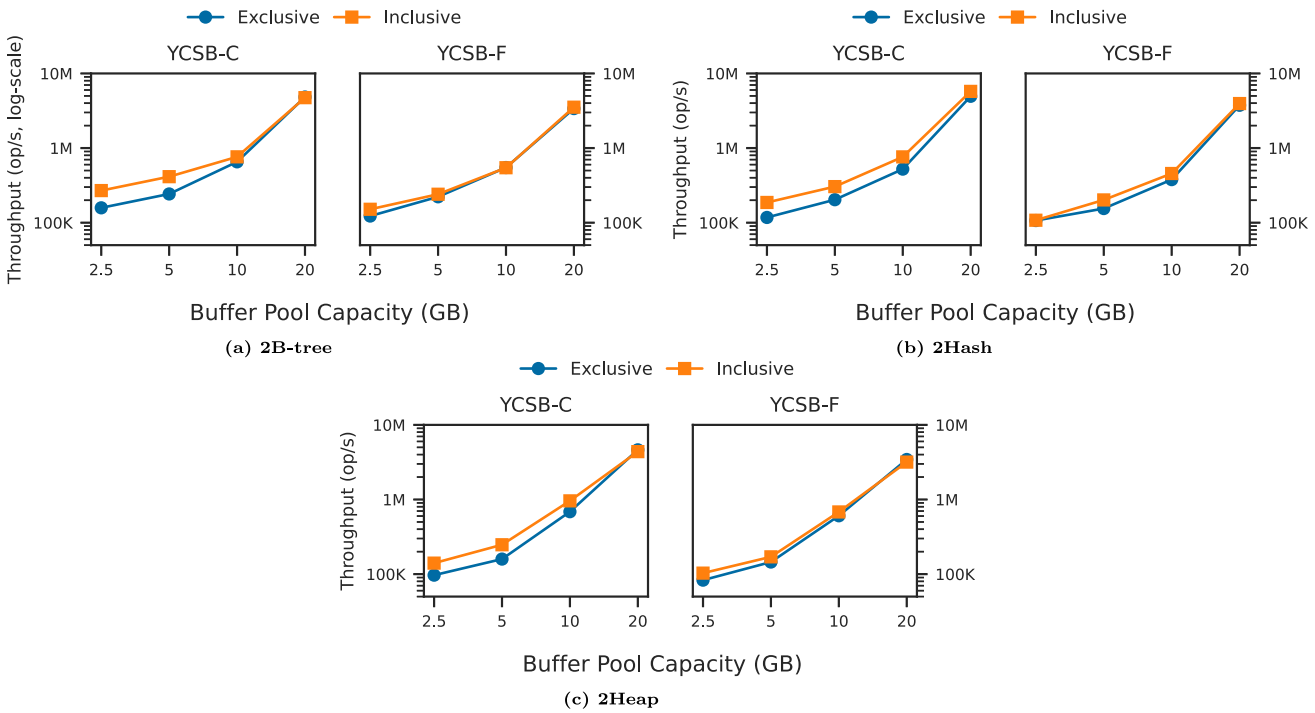


Fig. 21 Performance difference between exclusive and inclusive migration strategies.

range of buffer-managed access methods. Our results show that 2- TIER designs following TIERED- INDEXING outperformed traditional 1- TIER and record-caching designs in terms of memory utilization significantly for skewed workloads. We also presented a N- TIER structure, BILSM- TREE, that optimizes the memory utilization and throughput of LSM-tree. BILSM- TREE beats or equals traditional LSM-tree and LSM-tree with record caching across point read, blind write, update, and range scan. We also demonstrated that 2- TIER designs can outperform 1- TIER designs on write-intensive workloads by an order of magnitude thanks to our efficient migration strategies.

Funding Open Access funding provided by the MIT Libraries.

References

1. A new breakthrough in persistent memory gets its first public demo. https://itpeernetwork.intel.com/new-breakthrough-persistent-memory-first-public-demo/, May 2017
2. Anderson, R., Arun, G., Frank, R.: Oracle rdb's record caching model. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data* (1998), pp. 526–527. https://doi.org/10.1145/276304.276365
3. Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., Paleczny, M.: Workload analysis of a large-scale key-value store. In *SIGMETRICS* (2012), pp. 53–64. https://doi.org/10.1145/2318857.2254766
4. Balmau, O., Didona, D., Guerraoui, R., Zwaenepoel, W., Yuan, H., Arora, A., Gupta, K., Konka, P.: {TRIAD}: Creating synergies between memory, disk and log in log structured {Key-Value} stores. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (2017), pp. 363–375. https://doi.org/10.5555/3154690.3154725
5. Breslau, L., Cao, P., Fan, L., Phillips, G., Shenker, S.: Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM* (1999), vol. 1, IEEE, pp. 126–134. https://doi.org/10.1109/INFCOM.1999.749260
6. Cao, Z., Dong, S., Vemuri, S., Du, D. H.: Characterizing, modeling, and benchmarking {RocksDB}{Key-Value} workloads at facebook. In *FAST* (2020), pp. 209–223. https://doi.org/10.5555/3386691.3386712
7. Cha, S.K., Hwang, S., Kim, K., Kwon, K.: Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In VLDB **1**, 181–190 (2001). https://doi.org/10.5555/645927.672375
8. Chan, H. H., Li, Y., Lee, P. P., Xu, Y.: Hashkv: Enabling efficient updates in {KV} storage via hashing. In 2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18) (2018), pp. 1007–1019. https://doi.org/10.5555/3277355.3277451

9. Chandramouli, B., Prasaad, G., Kossmann, D., Levandoski, J., Hunter, J., Barnett, M.: Faster: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data* (2018), pp. 275–290. https://doi.org/10.1145/3183713.3196898
10. Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In *SoCC* (2010), pp. 143–154. https://doi.org/10.1145/1807128.1807152
11. Dean, J., Ghemawat, S.: LevelDB. http://leveldb.googlecode.com
12. DeBrabant, J., Pavlo, A., Tu, S., Stonebraker, M., Zdonik, S.: Anti-caching: A new approach to database management system architecture. *Proc. VLDB Endow. 6*, 14 (Sept. 2013), 1942–1953. https://doi.org/10.14778/2556549.2556575
13. Diaconu, C., Freedman, C., Ismert, E., Larson, P.-A., Mittal, P., Stonecipher, R., Verma, N., Zwilling, M.: Hekaton: SQL Server's Memory-optimized OLTP Engine. In *SIGMOD* (2013), pp. 1243–1254. https://doi.org/10.1145/2463676.2463710
14. Effelsberg, W., Haerder, T.: Principles of database buffer management. ACM Transactions on Database Systems (TODS) **9**(4), 560–595 (1984). https://doi.org/10.1145/1994.2022
15. Eldawy, A., Levandoski, J., Larson, P.-Å.: Trekking through siberia: Managing cold data in a memory-optimized database. *PVLDB 7*, 11 (2014), 931–942. https://doi.org/10.14778/2732967.2732968
16. Fagin, R., Nievergelt, J., Pippenger, N., Strong, H.R.: Extendible hashing-a fast access method for dynamic files. ACM Transactions on Database Systems (TODS) **4**(3), 315–344 (1979). https://doi.org/10.1145/320083.320092
17. Gilad, E., Bortnikov, E., Braginsky, A., Gottesman, Y., Hillel, E., Keidar, I., Moscovici, N., Shahout, R.: Evendb: optimizing key-value storage for spatial locality. In *Proceedings of the Fifteenth European Conference on Computer Systems* (2020), pp. 1–16. https://doi.org/10.1145/3342195.3387523
18. Graefe, G.: A survey of b-tree logging and recovery techniques. ACM Transactions on Database Systems (TODS) **37**(1), 1–35 (2012). https://doi.org/10.1145/2109196.2109197
19. Graefe, G., Volos, H., Kimura, H., Kuno, H., Tucek, J., Lillibridge, M., Veitch, A.: In-memory performance for big data. *VLDB 8*, 1 (2014). https://doi.org/10.14778/2735461.2735465
20. Gray, J., Sundaresan, P., Englert, S., Baclawski, K., Weinberger, P. J.: Quickly generating billion-record synthetic databases. In *SIGMOD* (1994), pp. 243–252. https://doi.org/10.1145/191843.191886
21. Harizopoulos, S., Abadi, D. J., Madden, S., Stonebraker, M.: Oltp through the looking glass, and what we found there. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2008), SIGMOD '08, Association for Computing Machinery, p. 981–992. https://doi.org/10.1145/1376616.1376713
22. Haubenschild, M., Sauer, C., Neumann, T., Leis, V.: Rethinking logging, checkpoints, and recovery for high-performance storage engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020), pp. 877–892. https://doi.org/10.1145/3318464.3389716
23. Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S., Jones, E. P. C., Madden, S., Stonebraker, M., Zhang, Y., Hugg, J., Abadi, D. J.: H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. In *VLDB* (2008), pp. 1496–1499. https://doi.org/10.14778/1454159.1454211
24. Kanellis, K., Chandramouli, B., Venkataraman, S.: F2: Designing a key-value store for large skewed workloads. *arXiv preprint* arXiv:2305.01516 (2023)
25. Kemper, A., Neumann, T.: HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE* (2011), ICDE, pp. 195–206. https://doi.org/10.1109/ICDE.2011.5767867
26. Leis, V., Alhomssi, A., Ziegler, T., Loeck, Y., Dietrich, C.: Virtual-memory assisted buffer management. Proceedings of the ACM on Management of Data **1**(1), 1–25 (2023). https://doi.org/10.1145/3588687
27. Leis, V., Haubenschild, M., Kemper, A., Neumann, T.: Leanstore: In-memory data management beyond main memory. In *ICDE* (2018), IEEE, pp. 185–196. https://doi.org/10.1109/ICDE.2018.00026
28. Leis, V., Haubenschild, M., Neumann, T.: Optimistic lock coupling: A scalable and efficient general-purpose synchronization method. IEEE Data Eng. Bull. **42**(1), 73–84 (2019)
29. Leis, V., Scheibner, F., Kemper, A., Neumann, T.: The art of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware* (2016), pp. 1–8. https://doi.org/10.1145/2933349.2933352
30. Levandoski, J., Lomet, D., Sengupta, S.: The Bw-Tree: A B-tree for new hardware platforms. *ICDE* (2013), 302–313. https://doi.org/10.1109/ICDE.2013.6544834
31. Levandoski, J., Lomet, D. B., Sengupta, S., Stutsman, R., Wang, R.: High performance transactions in deuteronomy. In *CIDR* (2015)
32. Levandoski, J. J., Lomet, D. B., Sengupta, S.: The bw-tree: A b-tree for new hardware platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)* (2013), IEEE, pp. 302–313. https://doi.org/10.1109/ICDE.2013.6544834
33. Litwin, W.: Linear hashing: a new tool for file and table addressing. In VLDB **80**, 1–3 (1980). https://doi.org/10.5555/1286887.1286911
34. Lomet, D.: Cost/performance in modern data stores: How data caching systems succeed. In *Proceedings of the 14th International Workshop on Data Management on New Hardware* (2018), pp. 1–10. https://doi.org/10.1145/3211922.3211927
35. Lu, L., Pillai, T.S., Gopalakrishnan, H., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Wisckey: Separating keys from values in ssd-conscious storage. ACM Transactions on Storage (TOS) **13**(1), 1–28 (2017). https://doi.org/10.5555/2930583.2930594
36. Mao, Y., Kohler, E.: Morris, pp. 183–196. Cache craftiness for fast multicore key-value storage. EuroSys, R. T. (2012)
37. Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., Schwarz, P.: ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM TODS **17**(1), 94–162 (1992). https://doi.org/10.1145/128765.128770
38. Neumann, T., Freitag, M. J.: Umbra: A disk-based system with in-memory performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings* (2020), www.cidrdb.org
39. O'Neil, P., Cheng, E., Gawlick, D., O'Neil, E.: The log-structured merge-tree (lsm-tree). Acta Informatica **33**(4), 351–385 (1996). https://doi.org/10.1007/s002360050048
40. Sheehy, J., Smith, D.: Bitcask: A log-structured hash table for fast key/value data. *Basho White Paper* (2010)
41. Stoica, R., Ailamaki, A.: Enabling efficient os paging for main-memory oltp databases. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware* (2013), pp. 1–7. https://doi.org/10.1145/2485278.2485285
42. Stonebraker, M., Madden, S., Abadi, D. J., Harizopoulos, S., Hachem, N., Helland, P.: The end of an architectural era: (it's time for a complete rewrite). In *VLDB* (2007), pp. 1150–1160. https://doi.org/10.5555/1325851.1325981
43. Sundarrajan, A., Feng, M., Kasbekar, M., Sitaraman, R. K.: Footprint descriptors: Theory and practice of cache provisioning in a global cdn. In *CoNEXT* (2017), pp. 55–67. https://doi.org/10.1145/3143361.3143368
44. Team, R.: RocksDB. http://rocksdb.org/
45. Van Aken, D., Pavlo, A., Gordon, G. J., Zhang, B.: Automatic database management system tuning through large-scale machine

learning. In *SIGMOD* (2017), pp. 1009–1024. https://doi.org/10.1145/3035918.3064029
46. Van Aken, D., Yang, D., Brillard, S., Fiorino, A., Zhang, B., Bilien, C., Pavlo, A.: An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems. *Proceedings of the VLDB Endowment 14*, 7 (2021), 1241–1253. https://doi.org/10.14778/3450980.3450992
47. van Renen et. al., A.: Managing non-volatile memory in database systems. In *SIGMOD* (2018). https://doi.org/10.1145/3183713.3196897
48. Vogel, L., van Renen, A., Imamura, S., Giceva, J., Neumann, T., Kemper, A.: Plush: a write-optimized persistent log-structured hash-table. *Proceedings of the VLDB Endowment 15*, 11 (2022), 2895–2907. https://doi.org/10.14778/3551793.3551839
49. Wang, Z., Pavlo, A., Lim, H., Leis, V., Zhang, H., Kaminsky, M., Andersen, D. G.: Building a bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data* (2018), pp. 473–488. https://doi.org/10.1145/3183713.3196895
50. White, S. J.: *Pointer swizzling techniques for object-oriented database systems*. The University of Wisconsin-Madison, 1994
51. Wu, F., Yang, M.-H., Zhang, B., Du, D. H.: {AC-Key}: Adaptive caching for {LSM-based}{Key-Value} stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (2020), pp. 603–615. https://doi.org/10.5555/3489146.3489187
52. Yang, J., Yue, Y., Rashmi, K.: A large scale analysis of hundreds of in-memory cache clusters at twitter. In *OSDI* (2020), pp. 191–208. https://doi.org/10.1145/3468521
53. Yu, G. X., Markakis, M., Kipf, A., Larson, P.-Å., Minhas, U. F., Kraska, T.: Treeline: an update-in-place key-value store for modern storage. *Proceedings of the VLDB Endowment 16*, 1 (2022), 99–112. https://doi.org/10.14778/3561261.3561270
54. Zhang, H., Andersen, D. G., Pavlo, A., Kaminsky, M., Ma, L., Shen, R.: Reducing the storage overhead of main-memory oltp databases with hybrid indexes. In *SIGMOD* (2016), pp. 1567–1581. https://doi.org/10.1145/2882903.2915222
55. Zhou, X., Arulraj, J., Pavlo, A., Cohen, D.: Spitfire: A three-tier buffer manager for volatile and non-volatile memory. In *Proceedings of the 2021 International Conference on Management of Data* (2021), pp. 2195–2207. https://doi.org/10.1145/3448016.3452819
56. Zhou, X., Yu, X., Graefe, G., Stonebraker, M.: Two is better than one: The case for 2-tree for skewed data sets. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, Online Proceedings* (2023)